Booker API Test Strategy & Plan

1. Objective -

The primary objective of this test strategy is to ensure that the Booker API is functionally correct, reliable, and robust. The tests validate the complete booking lifecycle (Create → Update → Verify → Delete), bulk operations, filtering mechanisms, and cross-endpoint data consistency.

2. Scope -

- **In Scope:**
  - CRUD operations for bookings
  - Filtering and query parameter validations
  - Data consistency across endpoints
  - Bulk operations and updates
  - Automated regression coverage in CI/CD pipeline
  - Data-driven test scenarios with large dynamic datasets
  - Security testing (SQLi, XSS, Auth bypass)
- **Out of Scope:**
  - UI Testing
  -

3. Test Types -

- **Functional Testing:** Validate API endpoints for correctness, including positive and negative scenarios.
- **End-to-End Testing:** Verify the full booking lifecycle, ensuring operations are consistent and independent.
- **Regression Testing:** Automated scripts cover existing functionality and ensure no regressions are introduced.
- **Bulk Operations Testing:** Validate creation, filtering, and updates of multiple bookings simultaneously.
- **Data Validation:** Ensure field-level correctness and cross-endpoint consistency.

4. Test Approach -

The testing approach is designed for efficiency, reliability, and scalability:

1. **Automation-First Philosophy:**
   - Automated tests implemented in Python with Pytest for rapid and repeatable execution.
   - Parameterized tests maximize scenario coverage without redundant code.
2. **End-to-End Lifecycle Validation:**
   - Tests validate the full booking lifecycle: Create → Update → Verify → Delete.
   - Each step includes verification to ensure data integrity.
3. **Dynamic & Isolated Test Data:**
   - BookingDataBuilder generates unique, dynamic test data for each execution in lower environments (QA/staging).
   - Function-scoped fixtures ensure parallel tests remain isolated, preventing data collisions.
4. **Test Design Logic for Validation :**
   - Bookings are created using the BookingDataBuilder class and stored in a registry (cache) for reference.
   - During test execution, bookings are retrieved from the registry based on Pytest parameters, and the API response is validated against the cached booking to ensure accuracy and data integrity.
4. **Resilient and Reliable Execution:**
   - Retry mechanisms handle transient API delays; polling loops address eventual consistency.
   - Each test cleans up its own data to maintain environment stability.
5. **CI/CD Integration:**
   - Tests integrated with the CI pipeline produce detailed reports and logs.
   - Quick feedback supports continuous integration and deployment cycles.
6. **Performance Awareness:**
   - Response times are monitored to identify potential API bottlenecks.
   - Latency checks included in functional workflows; throughput checks not currently implemented.

This approach ensures high coverage, maintainability, and reliability while demonstrating professional-level API testing practices.

5. Risks & Mitigation -

Risk	Mitigation
API downtime or transient failures	Retry and wait mechanisms in automated tests
Test data conflicts in parallel execution	Dynamic test data and isolated fixtures
CI/CD pipeline instability	Logging and detailed report generation for traceability

6. Test Tools & Framework -

- **Language:** Python 3.12
- **Test Framework:** Pytest
- **API Testing Utilities:** Custom helper classes (BookingDataBuilder, booking_helper)
- **CI/CD Integration:** GitHub Actions
- **Reporting:** Pytest HTML reports with embedded Pie Graph
- **LLM Usage:**
  LLMs are leveraged to brainstorm and refine test design logic, generate inline comments and docstrings for better code readability, and assist in reviewing, refactoring or optimizing Python code. In addition, LLMs are used to enhance reporting by embedding graphical insights into Pytest HTML reports, improving visibility of test results and trends.

7. Deliverables -

- Automated test scripts and helper modules
- Test execution reports (HTML/JSON)
- Documentation: README, Test Strategy, Summary of Findings & Recommendation

8. Reflection Questions & Answer’s:

8.1 Test Data Management in a Production Environment -

In a production environment, the priority is safety, compliance, and reliability rather than dynamic test data generation.

- **Segregated Test Accounts:** Use dedicated, pre-approved test accounts for validation. No real customer data should ever be touched.
- **Anonymized / Masked Data:** When production-like datasets are required, sensitive details must be anonymized or masked to meet privacy and security standards.
- **Version-Controlled Datasets:** Maintain datasets under version control to ensure repeatability and consistency across runs.
- **No Runtime Data Creation:** Automated tests should not generate or alter live production data.

This ensures that production testing is non-intrusive, repeatable, and compliant, while still providing meaningful quality signals.

8.2 Strategies for Test Environment Isolation -

Isolation is key for consistent, reliable testing:

- **Dedicated Environments:** Maintain separate dev, QA, and staging environments to prevent cross-contamination.
- **Containerization & Virtualization:** Leverage Docker or virtual machines to create isolated service instances on-demand.
- **Database Sandboxing:** Use separate databases or schemas per environment and reset data between runs.
- **Mocking External Services:** Stub or mock dependencies to isolate the system under test, reducing variability from external services.

This ensures tests remain reliable, consistent, and unaffected by external dependencies.

8.3 Designing Retry Logic for Flaky Tests -

Flaky tests can mask real issues. A robust retry strategy includes:

- **Selective Retries:** Retry only tests prone to transient failures, not actual defects.
- **Exponential Backoff:** Space out retries intelligently to avoid overloading systems (currently simple retry loops implemented).
- **Retry Limits:** Enforce a maximum number of attempts to prevent endless loops.
- **Detailed Logging & Analysis:** Capture logs for every retry attempt to identify patterns and implement long-term fixes.

This ensures transient issues do not obscure true defects while maintaining traceability and accountability.

8.4 Metrics to Measure API Quality Over Time -

Continuous measurement of API quality drives improvement and confidence:

- **Test Pass/Fail Rate:** Monitor success rates to gauge overall stability.
- **Response Time & Latency:** Track average and percentile response times to detect performance bottlenecks.
- **Error Rate:** Keep an eye on 4xx/5xx errors to identify functional or stability issues.
- **Flaky Test Incidents:** Monitor intermittent failures to tackle underlying instability.
- **API Coverage:** Ensure tests cover all critical endpoints and use cases.
- **Regression Trends:** Analyze recurring failures to proactively prevent regressions.

This ensures ongoing insights into API stability, reliability, and maintainability.