Automating Contract Testing with Pact (Consumer/Provider Flow)

1. Why Contract Testing?

- Unit tests test internal logic.
- Integration tests validate component interactions but can become brittle or complex.
- **Contract testing** verifies that service-to-service communication matches expectations **without needing the entire system up**.

3. Consumer-Driven Workflow

1. **Consumer Test Setup** (e.g., frontend or client service):

2. Write Pact tests simulating how the consumer expects to call the provider.

3. Pact generates a **contract** during test execution.

4. **Publish Contract**:

5. The consumer publishes the generated contract to a Pact Broker.

6. **Provider Verification**:

7. The provider downloads the contract from the broker.

8. It runs verification tests to assert it fulfills the contract.

9. **CI/CD Integration**:

10. Both steps (consumer generation & provider verification) run in separate pipelines.

11. Verification status is pushed back to the Pact Broker.

5. Advantages

- Independent verification → safe parallel development.
- Prevents “API drift” where changes silently break clients.
- Allows stubbing during development before actual implementation.

7. Limitations

- Best suited for synchronous HTTP/REST APIs.
- Async messaging support (Kafka/RabbitMQ) is available but more complex.
- Requires discipline to maintain up-to-date contracts.

Conclusion

Contract testing with Pact enforces tight API contracts in a decentralized, loosely coupled service architecture. It shifts responsibility to the **consumer**, ensuring contracts are well-defined, shareable, and verifiable — leading to more reliable integrations and faster feedback loops.