* Why the Service Layer?
  + It encapsulates API interaction or business logic, promoting separation of concerns, reusability, abstraction, and maintainability.
  + In test frameworks, it simplifies test classes by handling RestAssured logic.
* What Should It Contain?
  + API interaction logic, configuration, error handling, and reusable methods.
  + In an application, it includes business rules and coordination with repositories.  Code to construct and send HTTP requests using RestAssured.
  + Parsing and validating API responses.
  + Handling authentication (e.g., OAuth tokens, API keys).
* What Should It Not Contain?
  + Presentation logic, data access logic, test-specific logic, or excessive HTTP details.
* Industry Practice:
  + Industry frameworks favor a Service Layer for API interactions, initialized in @BeforeClass for flexibility and configurability, as it aligns with scalable, maintainable test architectures.

**BaseTest (Setup & Teardown Layer)**

1. This should:
2. Initialize common configurations (RestAssured.baseURI, auth tokens, logging, etc.)
3. Load config files/properties
4. Handle @BeforeClass, @AfterClass, etc.

**public** **class** BaseTest {

@BeforeClass

**public** **void** setup() {

RestAssured.baseURI = Config.getBaseUri(); // from config

// Set up auth, common headers, etc.

}

}

**Service Layer (API Methods)**

1. This handles:
   1. Encapsulating API endpoints
   2. Request building
   3. Response handling
   4. Logging
   5. Service classes encapsulate API endpoints and request logic

**package** services;

**import** io.restassured.response.Response;

**import** **static** io.restassured.RestAssured.given;

**public** **class** AuthService {

**public** Response login(String username, String password) {

**return** given()

.contentType("application/json")

.body("{ \"username\": \"" + username + "\", \"password\": \"" + password + "\" }")

.when()

.post("/auth/login");

}

**public** Response logout(String authToken) {

**return** given()

.header("Authorization", "Bearer " + authToken)

.when()

.post("/auth/logout");

}

// Other auth-related methods...

}

**Test Layer (TestNG Tests, Extend BaseTest)**

Use services and make assertions. This keeps your test logic clean.

**public** **class** LoginTest **extends** BaseTest {

**private** AuthService authService;

@BeforeClass

**public** **void** init() {

authService = **new** AuthService();

}

@Test

**public** **void** testSuccessfulLogin() {

Response response = authService.login("user", "pass");

response.then().statusCode(200);

Assert.assertNotNull(response.jsonPath().getString("token"));

}

@Test

**public** **void** testTokenRefresh() {

String token = authService.login("user", "pass").jsonPath().getString("refreshToken");

Response response = authService.refreshToken(token);

response.then().statusCode(200);

}

}

👍 Benefits of This Design:

Modular: Reuse services across tests.

Maintainable: Logic is separated; easier to debug.

Scalable: Add more layers like DTOs, custom assertions, or a utility layer.