private static final ThreadLocal<RequestSpecification> requestSpecThreadLocal = new ThreadLocal<>();

private static final ThreadLocal<SoftAssert> softAssertThreadLocal = new ThreadLocal<>();

* In your `BaseTest.java` , you've used ThreadLocal to isolate test state:
* This ensures each test thread has its own isolated instances of these objects.

Yes, that's correct! Using ThreadLocal in your BaseTest.java is a great way to ensure thread-safe isolation of test state, particularly in parallel test execution scenarios (e.g., when running tests with TestNG or JUnit in parallel).

Why ThreadLocal is Useful Here:

1. **Thread Safety**:
   * When tests run in parallel, multiple threads may execute tests simultaneously.
   * Without ThreadLocal, sharing RequestSpecification or SoftAssert across threads could lead to race conditions or unintended state modifications.
2. **Isolation**:
   * Each thread gets its own independent copy of RequestSpecification (e.g., for API requests) and SoftAssert (for assertions).
   * This prevents one test from interfering with another's state.
3. **Reusability**:
   * The ThreadLocal instances can be initialized in a @BeforeMethod (TestNG) or @BeforeEach (JUnit) and cleaned up in @AfterMethod/@AfterEach.

### Key Notes:

1. **Cleanup**:
   * Always call ThreadLocal.remove() in the @AfterMethod/@AfterEach to avoid memory leaks (especially in long-running applications or test suites).
2. **Access**:
   * Provide getter methods (e.g., getRequestSpec()) to ensure tests access their thread-local instances safely.
3. **SoftAssert**:
   * Remember to call softAssert.assertAll() at the end of each test to trigger the validation of all collected assertions.
4. **RestAssured**:
   * If using RestAssured, RequestSpecification can be configured once per thread (e.g., with base URL, headers, etc.).

@BeforeMethod // Changed from @BeforeClass to @BeforeMethod

public void setupAwsService() throws IOException {

    // ...

    messagingService = new MessagingService(...);

    messagingService.initialize();

}

**In `SampleApiTest.java` , you've changed from @BeforeClass to @BeforeMethod for AWS service initialization:**

**This ensures each test method gets its own fresh instance of MessagingService .**

System.out.println("AWS messaging service initialized for thread: " + Thread.currentThread().getId());

**You've added thread ID logging for debugging parallel execution:**

**Immutable Configuration**

private final String snsTopicArn;

private final String sqsQueueUrl;

private final Regions region;

private final String awsAccessKey;

private final String awsSecretKey;

private final String awsSessionToken;

In `MessagingService.java` , you've used final fields for configuration values:

### Why Use final Fields for Configuration?

1. **Immutability**:
   * final ensures these values cannot be modified after initialization, making the class thread-safe and predictable.
   * Configuration values (like AWS credentials, ARNs, or URLs) typically don't change at runtime.
2. **Clear Intent**:
   * Signals that these values are set once (e.g., via constructor) and remain constant.
3. **Thread Safety**:
   * No risk of accidental modification, even if the MessagingService is shared across threads.