MuleSoft Coding Mastery

MUnit Testing Framework & Batch Processing Review

Verify · Mock · Assert

Topics Covered:

Batch Components (Review) · MUnit Architecture
Mocking & Spying · Assertions · Test Suites

Prepared for:
Professional MuleSoft Developer
December 9, 2025

Contents

Chapter 1

Review: Batch Processing

Note: This section covers the essential 10% review of Batch Processing components.

1.1 Core Components

Batch processing is used for handling large datasets asynchronously and reliably.

1.1.1 1. Batch Job

The top-level container. It manages the lifecycle of the batch process.

- **Input:** Accepts a large payload (e.g., CSV with 10k rows).
- **Internal Process:** Automatically splits the payload into individual records and queues them.
- **Threading:** Multi-threaded by default.

1.1.2 2. Batch Steps

The processing units inside a job.

- **Sequential Steps:** Step 1 runs for all records, then Step 2 runs.
- **Accept Policy:**
  - `NO_FAILURES` (Default): Only process records that succeeded in previous steps.
  - `ONLY_FAILURES`: Only process records that failed previously (Error Handling).

1.1.3 3. Batch Aggregator

Purpose: Accumulates a subset of records to process them in bulk.

- **Why?** Inserting 1,000 records into Salesforce one by one is slow. Aggregating 200 records and doing 1 Bulk Insert is fast.
- **Variable Scope:** Variables created inside the Aggregator are NOT accessible outside of it.

1.2 Architecture Diagram

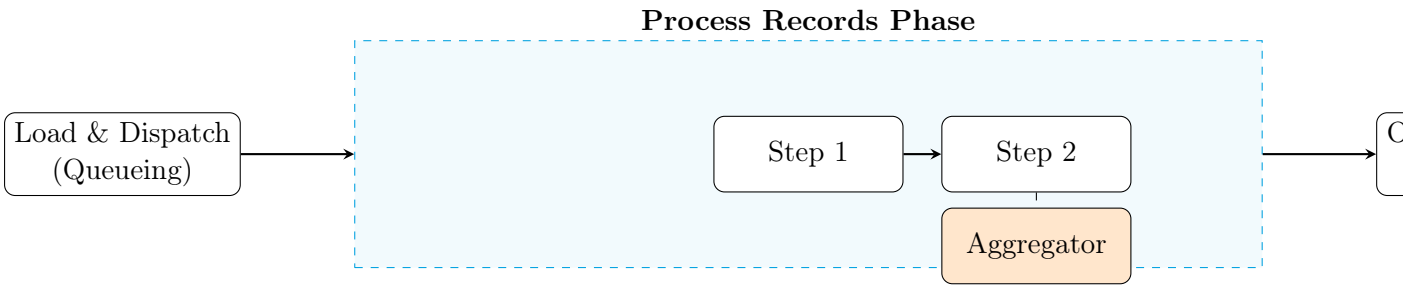

Figure 1.1: Batch Job Lifecycle

Chapter 2

MUnit: The Testing Framework

2.1 What is MUnit?

MUnit is the native testing framework for Mule 4. It allows you to build automated tests for your APIs and Integrations.

- **Purpose:** To ensure your code works as expected and to prevent "regressions" (breaking existing features when adding new ones).
- **How it works:** It starts an embedded Mule Runtime to run the tests. It does **not** deploy the application to CloudHub; it runs locally or in CI/CD.

2.2 Anatomy of a MUnit Test

A MUnit test is split into three scopes:

1. **Behavior (Pre-Conditions):**
   - Define what external systems should do (Mocking).
   - Example: "When the Database Connector is called, don't actually hit the DB. Instead, return this fake JSON."
2. **Execution (Action):**
   - The logic being tested. Usually, a `flow-ref` to the main flow.
3. **Validation (Assertions):**
   - Check the results.
   - Example: "Assert that the payload is not null" or "Verify the Logger was called exactly 1 time."

2.3 Architecture Diagram: MUnit Flow

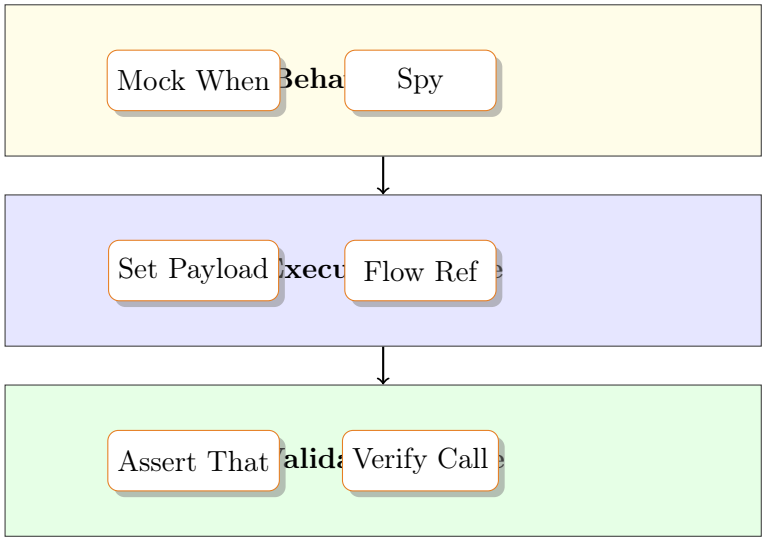

Figure 2.1: The Three Scopes of MUnit

Chapter 3

Mocking, Spying & Verifying

3.1 Mock When (The Isolator)

Definition: Allows you to replace a real processor (like an HTTP Request or DB Select) with a dummy behavior. **Why?** You typically cannot access production databases or third-party APIs during a test build.

3.1.1 Configuration Guidelines

- **processor:** The XML tag of the component (e.g., ‘db:select‘).
- **doc:id:** (Recommended) The specific ID of the component to mock.
- **then-return:** What payload/attributes/variables to return.

```
1 <munit-tools:mock-when doc:name="Mock DB" processor="db:select">
2     <munit-tools:with-attributes >
3         <munit-tools:with-attribute whereValue="Select User" attributeName="doc:
    name" />
4     </munit-tools:with-attributes>
5     <munit-tools:then-return >
6         <munit-tools:payload value="#[output application/json --- {'id': 1, '
    name': 'Test'}]" />
7     </munit-tools:then-return>
8 </munit-tools:mock-when>
```

Listing 3.1: Mocking a Database Call

3.2 Spy (The Inspector)

Definition: Allows you to see what happens *before* and *after* a processor executes, **without** changing its behavior. **Why?** To verify that a transformation inside a flow worked correctly before sending data to a connector.

3.3 Verify Call (The Counter)

Definition: Checks how many times a specific processor ran. **Usage:** "Ensure the Email Connector was called exactly once."

Interview Tip: Mock vs Spy

Mock REPLACES the processor (the real code never runs).
Spy WRAPS the processor (the real code runs, but you check data before/after).

Chapter 4

Assertions: Validating Results

4.1 Assert That

This is the primary way to pass or fail a test. It uses **DataWeave Matchers**.

4.1.1 Syntax

```
<munit-tools:assert-that
    expression="#[payload.status]"
    is="#[MunitTools::equalTo('Active')]"
    message="Status should be Active"/>
```

4.1.2 Common Matchers (MunitTools Functions)

You must import ‘MunitTools‘ in DataWeave.

Matcher	Purpose
`equalTo(value)`	Checks for equality.
`notNullValue()`	Ensures data exists.
`withMediaType('application/json')`	Checks format.
`hasKey('id')`	Checks if a JSON object has a key.
`both(M1).and(M2)`	Combines matchers (AND).

Table 4.1: MUnit DataWeave Matchers

4.2 Comparison Example

```
import * from MunitTools
---
payload must [
    haveKey("id"),
    haveKey("email"),
    be(notNullValue())
]
```

Listing 4.1: Complex Assertion

Chapter 5

Mini Project: Testing a Calculator API

5.1 The Scenario

We have a flow ‘calculateFlow‘ that:

1. Receives HTTP ‘ "a": 10, "b": 5, "op": "add" ‘.
2. Validates input.
3. Returns result ‘ "result": 15 ‘.

5.2 The MUnit Test Implementation

5.2.1 Step 1: Execution Scope

We simulate the incoming HTTP request by setting the payload manually.

```
<munit:execution>
    <munit:set-event doc:name="Set Payload">
        <munit:payload value="#[{ 'a': 10, 'b': 5, 'op': 'add' }]" mediaType="
    application/json" />
    </munit:set-event>

    <flow-ref name="calculateFlow" />
</munit:execution>
```

5.2.2 Step 2: Validation Scope

We check if the math was done correctly.

```
<munit:validation>
    <munit-tools:assert-that
        expression="#[payload.result]"
        is="#[MunitTools::equalTo(15)]"
        message="The addition result is incorrect!" />

    <munit-tools:verify-call processor="logger" times="1" />
</munit:validation>
```

5.3 Testing Error Handling

How do we test if an error is thrown correctly?

```
<munit:test name="test-error-scenario" expectedErrorType="MULE:EXPRESSION">
    <munit:execution>
        <munit:set-event>
             <munit:payload value="#[null]" />
        </munit:set-event>
        <flow-ref name="calculateFlow" />
    </munit:execution>
    </munit:test>
```

Listing 5.1: Expected Error Test

Chapter 6

Best Practices & Interview Guide

6.1 MUnit Best Practices

Golden Rules of Testing

- **Isolation:** Never depend on external systems (Sandbox/Prod). Always MOCK DBs, HTTP Requests, and File connectors.
- **Coverage:** Aim for 80%+ code coverage.
- **Naming:** Name tests clearly: '[FlowName]$_{[}$*Scenario*]$_{[}$*ExpectedResult*]'(*e.g.*, '*createUser*$_{i}$*nvalidInput*
- **Files:** Keep test resources (example JSONs) in 'src/test/resources' to keep the XML clean.

6.2 Interview Questions

6.2.1 Q1: Can MUnit test private flows?

Answer: Yes. MUnit can invoke private flows directly via 'flow-ref' in the execution scope, even if they don't have an event source.

6.2.2 Q2: What is the difference between 'mock:when' and 'mock:then' (deprecated)?

Answer: In Mule 3, we had 'mock:when'. In Mule 4, we strictly use 'munit-tools:mock-when'. It looks for a processor matching specific criteria (doc:id, name) and prevents it from running.

6.2.3 Q3: How do you enable Code Coverage in Anypoint Studio?

Answer: Go to Run Configurations → MUnit → Enable "Run coverage reports". It generates an HTML report showing which processors were executed during tests.

6.3 Summary

- **Batch Processing:** Efficiently processes large data in blocks (Load → Process → On-Complete).
- **MUnit:** The standard for quality assurance in MuleSoft.

- **Three Scopes:** Behavior (Mock), Execution (Run), Validation (Assert).
- **Key Tools:** ‘Mock When‘ (Isolate), ‘Spy‘ (Inspect), ‘Assert That‘ (Validate).