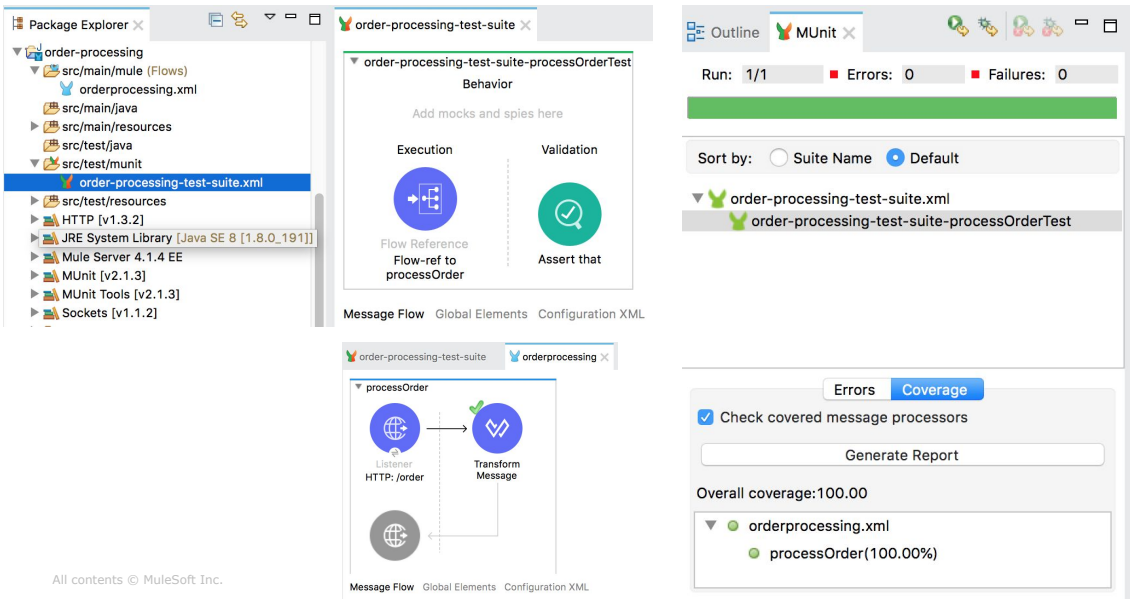
Module 6: Designing Mule Application Testing Strategies

Goal



At the end of this module, you should be able to

- Understand why testing is important in an integration project
- Identify the MuleSoft provided testing framework and tools for testing during the software development lifecycle
- Design testing strategies for a Mule application
- Define test coverage for flows in a Mule application

Designing testing strategies for Mule applications

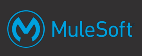
Why applications need testing

- To **find problems early** in the **development cycle** and **fix** them **quickly**
- As executable documentation of the code's behaviour
- To Minimize risk for developers to later make changes
- To make collaboration between developers easier

Why applications need **automated** testing

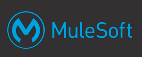
- To find problems quickly **every time** code is changed
- As a necessary step in continuous deployment, continuous integration, and continuous delivery pipelines

The architect's responsibilities related to designing testing strategies



- It is important to build a comprehensive **test suite** of relevant tests for each Mule project
  - To ensure **consistent, predictable, and reliable deployments** and runtime operations
- As an architect, you are responsible for defining these testing strategies, particularly as they relate to Mule applications

Types of testing that are particularly relevant to Mule applications



- Unit testing
  - Verifies the **functionality** of **a specific unit (**often a flow or subflow)
  - If the flow exposes an interface, request data and any external calls or dependencies are usually mocked or stubbed-out
- Integration testing
  - Verifies **all interactions to/from interacting systems** are **functional**
  - Test the actual requests and/or responses exchanged with external systems
- Performance testing
  - Measure, validate, or verify performance characteristics of the system, such as **scalability**, **reliability**, and **resource usage** under **various workload**

Type of tests **not** particular to a Mule application

- Other types of testing that relate more to the general lifecycle of any Java application may not need specific Mule testing tools
  - Such as destructive, resilience, functional, and regression testing
  - These testing strategies may already be in place within the organization for every application deployment

Reflection questions: Unit testing

- What are some examples where unit testing is helpful, and how would you implement unit tests?
- What are the general features of a unit test of
  - An HTTP Request?
  - A database select vs. a database insert, update, or delete operation?
  - A DataWeave transformation?
  - A Scatter-Gather component?
  - A For Each scope and its components?
  - Custom Java code?

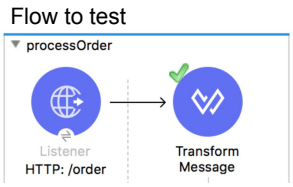
Reflection questions: Performance testing

- You'll learn more about performance testing later in the course, but for now think about these questions
  - What network considerations might affect performance testing?
  - What choice of tools might affect performance testing?
  - What are the tradeoffs of meeting various performance goals, and what other goals might conflict with performance goals?

Testing Mule applications using MUnit

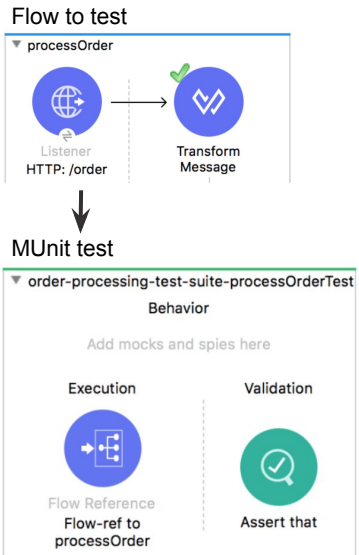
How Mule flows can be tested

- **JUnit** is often used to build unit tests specific functionality of Java code
  - Usually to test one **method** call, or a Java class's constructor
  - External dependencies are replaced, mocked, or stubbed
  - Input data is provided
  - Assertions about the result are tested and recorded
- In a Mule application, a flow is logically equivalent to a Java method



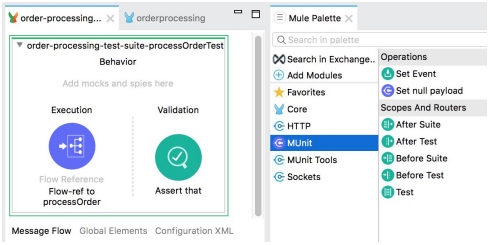
How Mule flows can be tested using MUnit

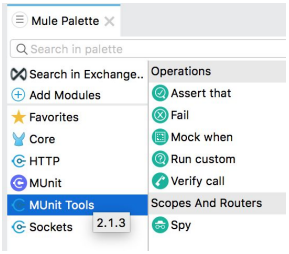
- Anypoint Studio includes **MUnit** to
  - To build **test suites** focused specifically on how flows in a Mule application behave
  - **Automatically** generates testing stub flows for flows
    - The developer then fills in the testing **assertions** (conditions and logic) using Mule event processors from the MUnit modules



Using MUnit to test Mule applications

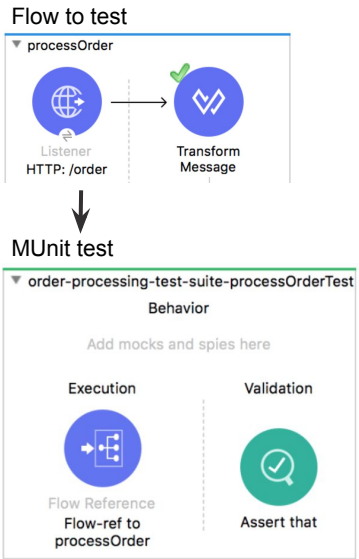
- **MUnit** is a test suite specifically targeted to testing Mule applications inside a Mule runtime
  - Builds test suites focused on Mule application details
  - Each test is itself a Mule flow, using components from the **MUnit** and **MUnit Tools** modules
  - The testing Mule flows execute in a Mule runtime and report results





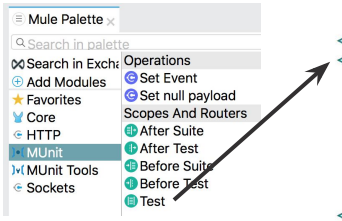
Using MUnit to build Mule application testing suites

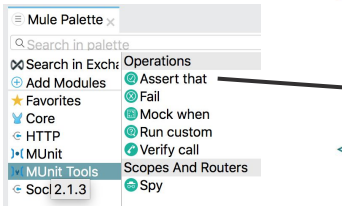
- MUnit allows you to
  - Design and create **whitebox** test cases as Mule flows
    - Tests the internal workings and behaviors of Mule flows
    - Each test is itself a Mule flow
  - Run tests as Mule flows in a Mule runtime
    - **Manually** in Anypoint Studio
    - **Automatically** as part of Maven based **CI/CD** build process
  - View test results and coverage inside Anypoint Studio



MUnit tests are also Mule flows

- The MUnit module provides event scopes and event processors to test flows and event processors within flows

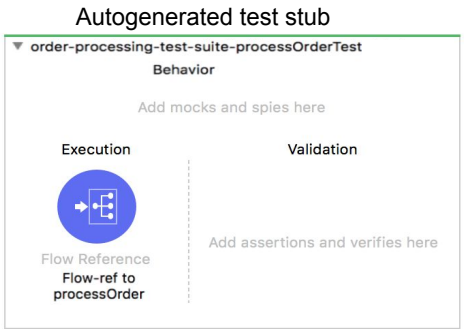





```
<munit:config name="new-test-suite.xml" />
<munit:test name="new-test-suite-ordersUnitTest" description="Test" tags="sales, unit">
    <munit:execution >
        <flow-ref doc:name="Flow-ref to orders"  name="orders"/>
    </munit:execution>
    <munit:validation >
        <munit-tools:assert-that doc:name="Assert that payoad is not null"
        expression="#[payload]" is="#[MunitTools::notNullValue()]" message="Payload is not null"/>
    </munit:validation>
</munit:test>
<munit:test name="new-test-suite-accountBalanceTest" description="Test" >
    <munit:execution >
        <flow-ref doc:name="Flow-ref to accountBalance" name="accountBalance"/>
    </munit:execution>
    <munit:validation >
        <munit-tools:assert-that doc:name="Assert that" expression="#[payload]"
        is="#[MunitTools::notNullValue()]" message="Payload is not null"/>
    </munit:validation>
</munit:test>
```


The structure of autogenerated MUnit tests

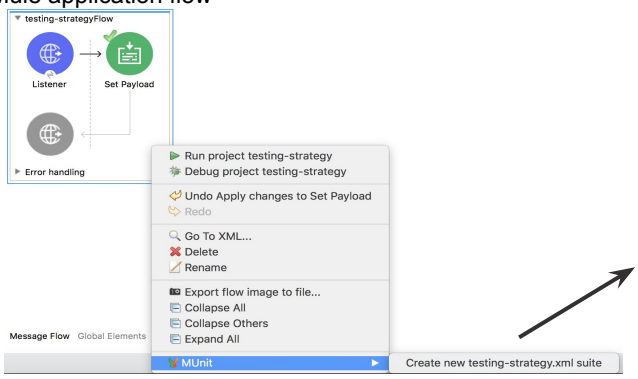
- Each MUnit test case created by Anypoint Studio has
  - An Execution section with a **configured event source**
  - An **empty** Validation section
- Execution section usually calls a flow in the Mule application
  - This bypasses the flow's Event Source



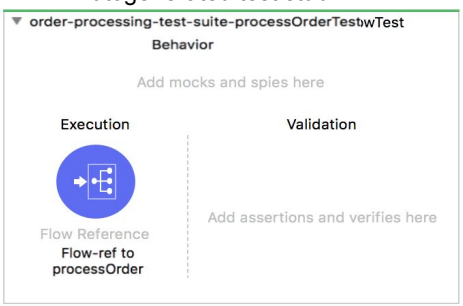
Autogenerating MUnit test suites using Anypoint Studio

- Anypoint Studio can create stubs MUnit test suites from a Mule application's flows
  - The test suite is a new Mule XML configuration file with stub flows
  - This way the testing flows do not clutter the actual Mule app XML files

Mule application flow



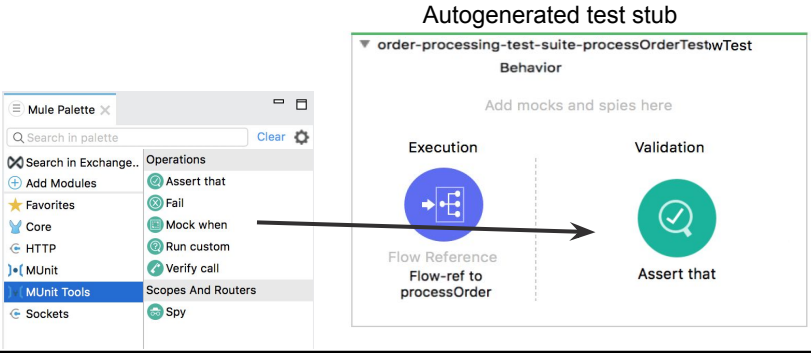
Autogenerated test stub



Design and create test cases

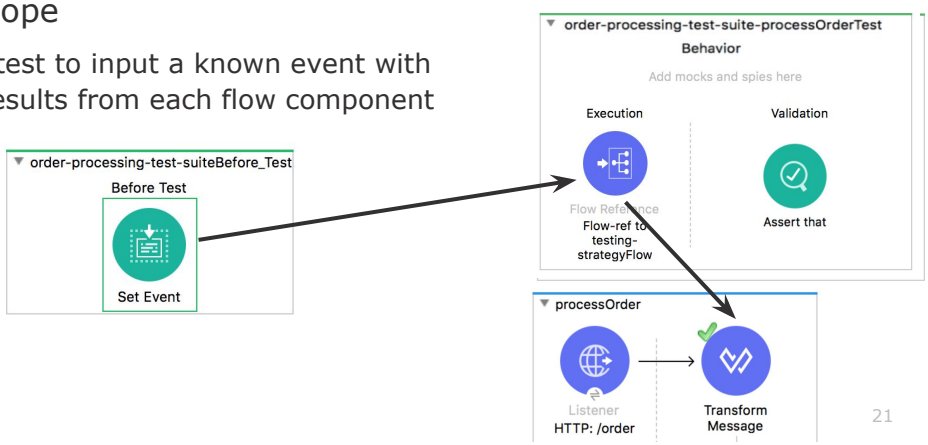
- Validation should be added to each test case
  - To add the logic to compare the expected result vs. the actual result of the test case

Autogenerated test stub



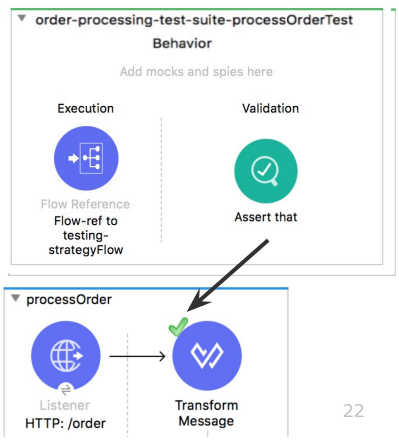
Passing known event data into an MUnit test

- A flow reference in the Execution section bypasses the referenced flow's Event Source
- A **Before Test** scope can set the Mule event passed to the Execution scope
  - Allows the test to input a known event with expected results from each flow component



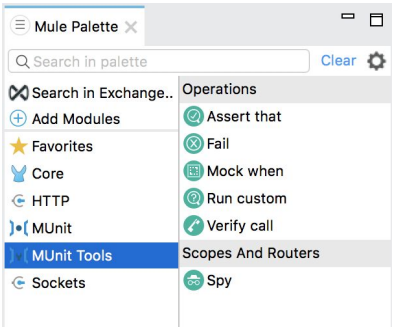
Viewing test results and coverage in Anypoint Studio

- When a flow passes tests, the flow components display a green check mark
- This can also visually show you how much of a Mule application is covered by the test suite or test suites



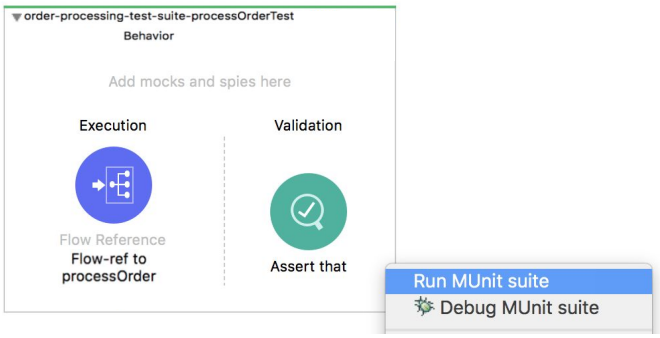
Test components used to build MUnit tests

- **Mock** any processor
  - **Deterministically return expected data or throw a certain error** for certain event processors
    - Focuses the unit test on the particular flow
    - Ignores dependencies outside the flow, such as results of a connector, flow ref, DataWeave transformation, variables, and attributes
- Define **assertions**
  - Set boolean conditions to compare expected vs. actual data at a specific point in a flow
- Spy on flow components
  - A scope to **validate (assert)** the **Mule event state** before and after a flow component



Running MUnit test suites

- MUnit test cases are deployed and run in a dedicated Mule runtime
  - The Mule runtime is started for the tests by Anypoint Studio or by the MUnit maven plugin
  - After the selected test suite(s) run, this Mule runtime terminates



Using MUnit test suites to automate testing

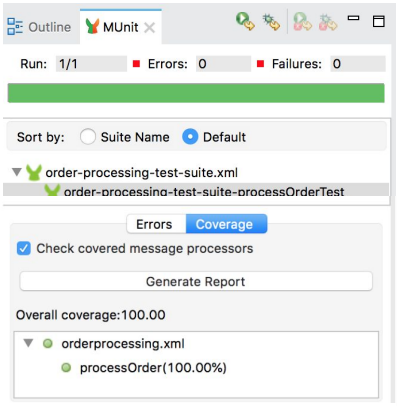
- MUnit test suites can also be run automatically outside of Anypoint Studio
- MUnit is fully integrated with Maven through a Mule Maven plugin
  - Libraries are available in the MuleSoft public nexus
  - MUnit test suites are defined as elements inside a Mule application XML
    - Are triggered during a Maven test phase
  - Works with surefire-reports
- This allows tests to run
  - Automatically as part of any Maven build
    - Such as locally at the developer's machine
  - Whenever code is changed and checked back in to an online repository, or other CI/CD pipeline

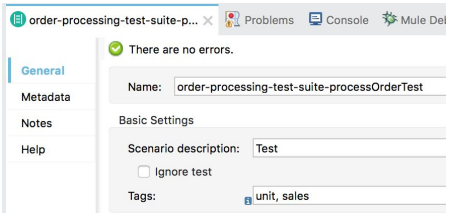
How to mock data to support MUnit tests

- You already saw that MUnit can **mock** any event processor and replace the output with mocked data
  - Including connectors, flow references, DataWeave, variables, and attributes
- There are also **other ways to mock data** going through a Mule flow
  - API Manager provides mocking services for APIs
  - Other third party products, servers, and online sites are commonly used to mock external responses for other types of external systems

MUnit features to select tests to run and view test results

- Code coverage reports
  - Reports the percent of code that was tested, visually in Anypoint Studio and in generated reports
- Test tags
  - Each MUnit test can include tags
  - Can run only tests matching selected tag(s)
  - For example to tag unit vs. integration tests





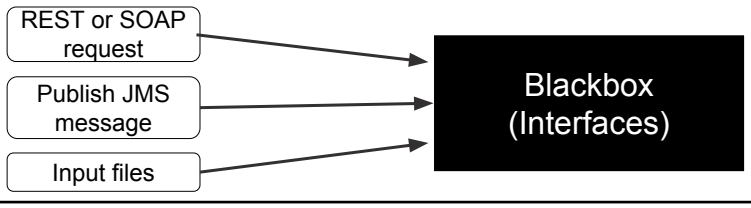
Designing integration and performance tests of Mule applications

MUnit features for integration testing

- MUnit is also a whitebox integration testing framework for Mule applications
  - Assertions
  - Code coverage reports (generated reports)
  - Test tags

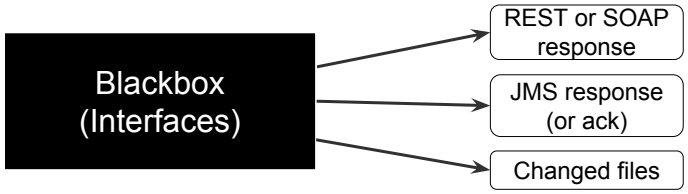
Features of blackbox (functional) integration testing

- Interact with the component-under-test only through its published interface
  - Invoke REST APIs or SOAP web services exposed by the component
  - Send JMS or other messages to a queue or topic on which the component listens
  - Create files in a folder which the component watches
- Have no knowledge of implementation details of the component



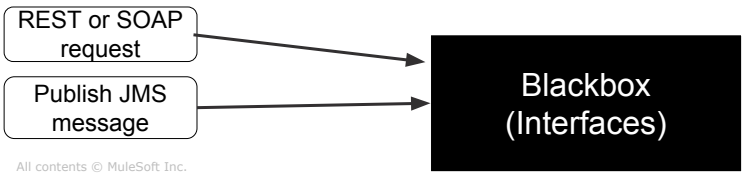
Other features of blackbox (functional) integration testing

- Register the component's response or observable behaviour (side effects)
  - REST or SOAP response
  - Resulting (response or follow-up) JMS messages
  - Disappearance or renaming of consumed files or creation of output files
- Assert that the response/behaviour from/of the component adheres to its published specification
  - RAML definition or WSDL document
  - Human-readable documentation and functional specification



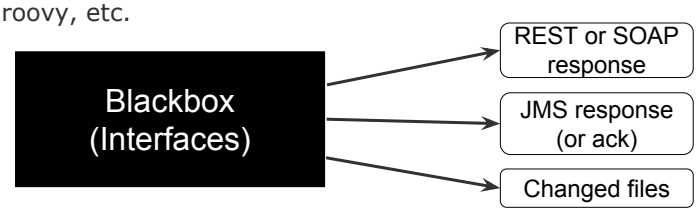
Minimum support needed for Blackbox (functional) integration **testing tools** to **generate requests**

- Ability to trigger the component-under-test through its published interface
  - Send HTTP request: plain, REST, SOAP
  - Send JMS messages



Minimum support needed for Blackbox (functional) integration **testing tools** to **test responses**

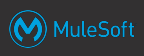
- Ability to record the component's response
  - Receive HTTP responses and JMS messages
- Ability to define assertions that validate the response
  - Validate XML and JSON docs for well-formedness and against a schema
  - Validate HTTP response status codes (200 - 599)
  - Expressions on HTTP or JMS header values, HTTP response body, JMS payload, parts thereof, etc.
    - Requires an expression language or full-fledged programming language, such as JSON path, XPath, Groovy, etc.



Blackbox integration testing should be able to be automated

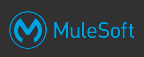
- Should be able to be launched both
  - Manually and interactively
  - Automatically from Continuous Delivery (CI/CD) pipelines
- Tools should support Maven plugins or Jenkins pipeline plugins

Popular tools for blackbox (functional) integration testing



- These tools fulfill most of the previously listed requirements:
  - SOAPUI
    - Open-source and commercial
  - Restlet Client
    - No explicit support for SOAP over plain HTTP and no JMS support
  - REST-assured
    - Integrates with JUnit, so tests are written in Java, Scala or Kotlin
    - No explicit support for SOAP over plain HTTP and no JMS support
      - But any JVM library usable within JUnit tests

Performance Testing



- MuleSoft does not provide particular performance testing tools
- As an architect, to plan for performance testing earlier in development is mandatory for application where performance is KPI
  - JMeter or BlazeMeter or any open source performance testing can be used for performance testing
- Later in the class, the **Optimizing Performance** module has detailed discussion about performance testing and how to measure the performance of a Mule application

Why test coverage is important

- The **MUnit Coverage** feature provides metrics on what percentage of a Mule application's code has been tested by a set of MUnit tests
  - **Flow coverage metrics**
    - Reports how many flows, subflows, and batch jobs were covered
  - **Resource coverage metrics**
    - Reports on each Mule configuration file under src/main/mule
  - **Application overall coverage metrics**
    - An average of the flow and resource coverage metrics
- For reliable CI/CD pipelines, aim for MUnit test coverage of 80% or above

Exercise 6-1: Define a testing strategy for a use case

- Define the types of testing required for a use case
- Document a testing strategy for a use case
- Document the code coverage for test cases

Exercise steps

- Reread the use case and identify the testing requirement for the class use cases
- Define the type(s) of testing required to support the uses cases
- Document the code coverage by an MUnit test suite
  - Note: Due to the time limits of this class, do not document individual test cases

Exercise solution

- Open the exercise solutions file
- Compare your documented testing strategy with the exercise solution's testing strategy

Summary

Summary

- It is important to **identify, define, and implement automated tests early in a project** to help developers to find bugs early in the development lifecycle
- Good testing strategies give **developers confidence** about changing code throughout the entire lifecycle of a Mule application
- With MUnit, unit testing and whitebox integration testing is easily **automated and incorporated with CI/CD** and helps to improve code quality
- Documenting test coverage for application is another important part of testing strategy
- Performance testing is not always required in every Mule application

Reflection questions: Testing Strategy

- What are some examples where unit testing is helpful, and how would you implement unit tests?
- How are integration tests of a flow different from unit tests?
- How is source and target data retrieved vs. mocked for these types of tests?
- What factors affect or limit the effectiveness of an integration test?
- What types of performance tests are common for Mule flows?
- Should testing use average load, worst case load, or some other model?