Real-World Unit Testing

Get to 100% Coverage, The Right Way

David Reed, Senior Member of Technical Staff, Salesforce.org
d.reed@salesforce.com, @aoristdual


David Reed
Senior Member of Technical Staff,
Release Engineering, Salesforce.org

Forward-Looking Statement

Statement under the Private Securities Litigation Reform Act of 1995

This presentation may contain forward-looking statements that involve risks, uncertainties, and assumptions. If any such uncertainties materialize or if any of the assumptions prove incorrect, the results of salesforce.com, Inc. could differ materially from the results expressed or implied by the forward-looking statements we make. All statements other than statements of historical fact could be deemed forward-looking, including any projections of product or service availability, subscriber growth, earnings, revenues, or other financial items and any statements regarding strategies or plans of management for future operations, statements of belief, any statements concerning new, planned, or upgraded services or technology developments and customer contracts or use of our services.

The risks and uncertainties referred to above include – but are not limited to – risks associated with developing and delivering new functionality for our service, new products and services, our new business model, our past operating losses, possible fluctuations in our operating results and rate of growth, interruptions or delays in our Web hosting, breach of our security measures, the outcome of any litigation, risks associated with completed and any possible mergers and acquisitions, the immature market in which we operate, our relatively limited operating history, our ability to expand, retain, and motivate our employees and manage our growth, new releases of our service and successful customer deployment, our limited history reselling non-salesforce.com products, and utilization and selling to larger enterprise customers. Further information on potential factors that could affect the financial results of salesforce.com, Inc. is included in our annual report on Form 10-K for the most recent fiscal year and in our quarterly report on Form 10-Q for the most recent fiscal quarter. These documents and others containing important disclosures are available on the SEC Filings section of the Investor Information section of our website.

Any unreleased services or features referenced in this or other presentations, press releases or public statements are not currently available and may not be delivered on time or at all. Customers who purchase our services should make the purchase decisions based upon features that are currently available. salesforce.com, Inc. assumes no obligation and does not intend to update these forward-looking statements.

What's in this Talk?

- Framing testing objectives.
- Testing strategies for real code that might not be perfect.
  - No libraries
  - No (required) enterprise design patterns
- Very little discussion of obtaining code coverage.
  - … and we'll talk about why.
- Only Apex testing
  - Jest, LTS, Selenium, Robot, UAT, regression… many other topics!

A Tiny Bit About Code Coverage

It’s a trap!

```
trigger AccountTrigger on Account (before insert) {
    if (Trigger.new[0].Name == 'Star Helix') {
        Trigger.new[0].Industry = 'Security';
        insert new Task(Subject = ‘Follow Up’);
    }
}
```

100% coverage!

… but two bugs unfixed

```
@isTest
public static void runTest() {
    Account a = new Account(Name = ‘Star Helix’);
    insert a;
}
```

A Tiny Bit About Code Coverage

It's a trap!

```
public String getAccountIndustry(Contact c) {
    return c.Account.Industry;
}
public void coverage() {
    Integer i = 0;
    i++;
    i++;
    i++;
}
```

```
@isTest
public static void test() {
    myClass.coverage();
}
```

80% coverage!
… but no validation

A Tiny Bit About Code Coverage

(Limited) Proxy for Tests

- Example test classes get good coverage, but miss serious bugs.
- Code coverage is a machine-friendly proxy for testing.
- Code coverage is a necessary, but not sufficient, indicator of your testing program's efficacy.

- **Optimizing for code coverage is the wrong goal for unit testing.**
  - **…. but we'll get it for free if we follow the right path.**
  - **…. and it can help us identify deficiencies in our tests.**

Testing is a Promise

Testing is a Promise

What are we claiming when we build tests?

- When the tests pass, the code works.
- When the code will fail, the tests will also fail.

Testing is a Promise

Who are we making a promise to? What are the stakes?

We're promising:

- Our future selves.
- Our colleagues in IT.
- Our business users or customers.
- Our leadership.

We're staking:

- Money: production faults can be *very* expensive.
- Trust
  - In the CRM and the data
  - In the IT team

Testing is a Promise

How do we make our promises meaningful - and keep them?

- Tests are comprehensive: all functionality is covered.
- Tests are effective: assertions make verifiable claims.
- Tests are representative of real use and failure scenarios.

The Lifespan of a Unit Test

Deployment is just the beginning

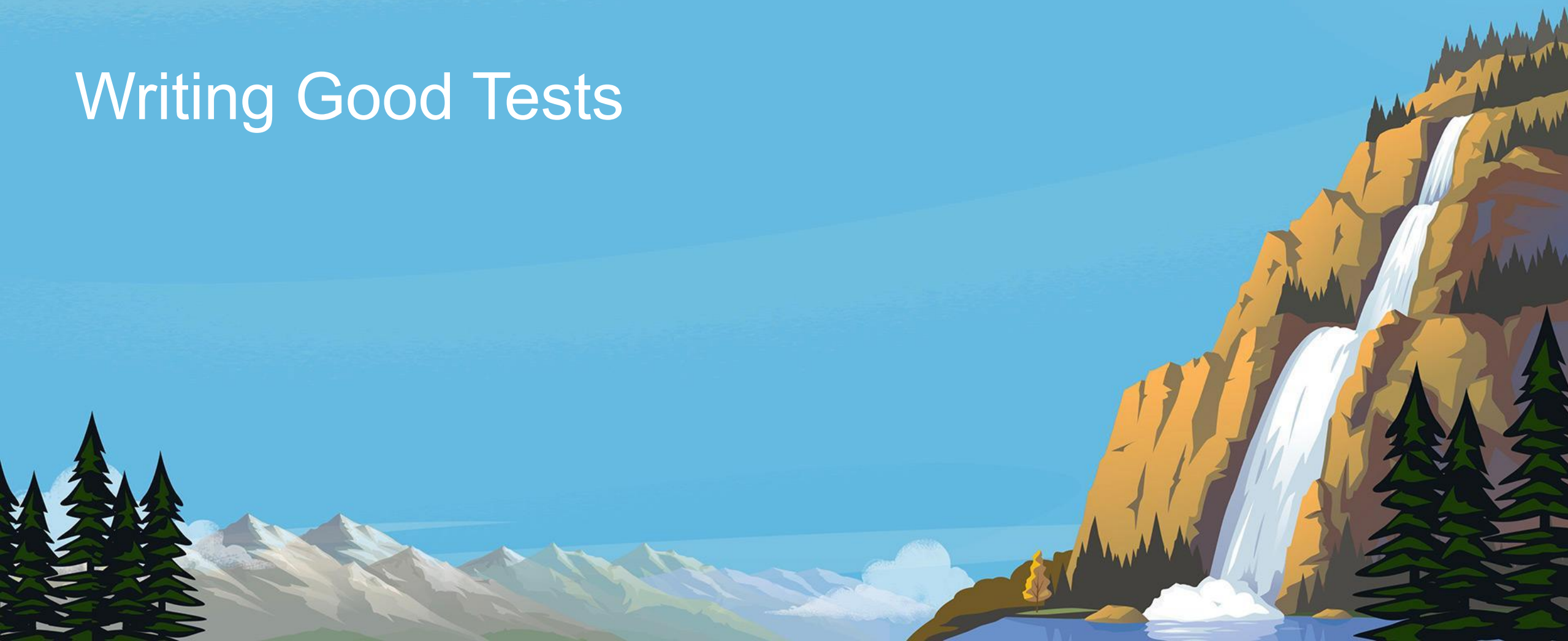
Your test needs to run **1** time to deploy your code to production.

Assume

- The code you're writing today will be in use (*in some form*) for 4 years.
- You deploy to production every two weeks.
- Every production deploy follows at least 3 deploy/test passes in sandboxes.

A unit test you write today could run approximately **4 * 26 * (3 + 1)** times in its lifetime, giving it **416** chances to catch a regression.

Writing Good Tests



Three Key Pieces to Every Test

- Environment Setup and Data Creation
- Execution of Functionality
- Meaningful Assertions

- These pieces give us our promise!

```
@isTest
public static void runTest() {
    // Test setup
    Account a = [SELECT Id FROM
Account];
    insert new Account(
        Name = 'Test');
    // Start Test
    MyClass.doTheThing(a);
    Test.stopTest();
    a = [SELECT Id, Name FROM
Account];
    System.assertEquals('NotTest',
a.Name, 'name of record');
}
```

Types of Apex Unit Test

Positive tests

- Positive tests
  - Single record
  - Multiple records
  - Bulk/volume tests

```
@isTest
public static void trigger_updates_field() {
    List<Account> accts = new List<Account> {
        new Account(Name = 'Beratnas Gas'),
        new Account(Name = 'Mao-Kwikowski Mercantile'),
        new Account(Name = 'Protogen')
    };

    Test.startTest();
    insert accts;
    Test.stopTest();

    for (Account a : [SELECT Name, Industry FROM Account]) {
        System.assertEquals('Unknown', a.Industry, 'Industry
populated');
    }
}
```

Types of Apex Unit Test

Negative tests

- Failure cases
- Invalid parameters
- Nulls and empty inputs
- Expected exceptions thrown

```
@isTest
public static void handles_null_input() {
    System.assertEquals(null, MyClass.myMethod(null));
}

@isTest
public static void throws_exception_on_null_input() {
    Boolean caught = false;
    try {
        MyClass.myMethod(null);
    } catch (MyClass.InvalidArgumentException e) {
        System.assert(e.getMessage.contains('cannot be null'));
        caught = true;
    }

    System.assertEquals(true, caught, 'received expected
exception');
}
```

Types of Apex Unit Test

No-action tests

- Confirm filter logic is applied correctly
- Ensure only desired records are affected

```
@isTest
public static void trigger_does_not_overwrite_field() {
    List<Account> accts = new List<Account> {
        new Account(Name = 'Mao-Kwikowski Mercantile'),
        new Account(Name = 'Beratnas Gas'),
        new Account(Name = 'Star Helix', Industry =
'Security')
    };

    Test.startTest();
    insert accts;
    Test.stopTest();

    for (Account a : [SELECT Name, Industry FROM Account]) {
        if (a.Name != 'Star Helix') {
            System.assertEquals('Unknown', a.Industry,
'Industry populated');
        } else {
            System.assertEquals('Security', a.Industry,
'Industry not changed');
    }
}
```

Types of Apex Unit Test

Permission and sharing tests

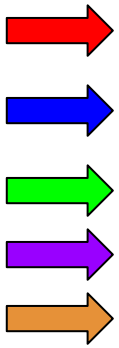
- Check FLS and CRUD enforcement
- Validate behavior in restricted sharing environments
- Typically involves creating Users, may include Permission Set Assignments and Roles.

```
@isTest
public static void handler_queries_visible_records() {
    User u = UserFactory.createUser('Josephus Miller');
    UserFactory.addPermissionSet(u, 'Investigation User');
    System.runAs(u) {
        // Validate that permissions and
        // visibility for `u` are applied as expected.
    }
}
```

Writing Testable Code

Thinking in “Code Path” Terms

One Test : One Path (roughly)



Notice how we see some of the “test types” we discussed earlier come out of pathing.

```
   public void updatePriorities(List<Opportunity> opportunities,
OpportunityMode mode) {
        // Snipped for brevity.
        for (Opportunity o : opportunities) {
            if (accountMap.containsKey(o.AccountId)) {
                Account a = accountMap.get(o.AccountId);

                if (a.Industry == 'Spaceflight') {
                    if (mode == MODE_OVERWRITE ||
String.isBlank(o.Priority__c)) {
                        o.Priority__c = 'Prioritized';
                    }
                } else if (a.Industry == 'Fossil Fuels') {
                    if (mode == MODE_OVERWRITE ||
String.isBlank(o.Priority__c)) {
                        o.Priority__c = 'Low';
                    }
                } else {
                    if (mode == MODE_OVERWRITE ||
String.isBlank(o.Priority__c)) {
                        if (o.Amount > 100000) {
```

Simple Heuristic: Easy to “Path” → Easy to Test

How do we achieve that?

- Factor and modularize
- Reduce “cyclomatic complexity” - flatten logical decision tree
- Remove unnecessary branching

Example 1: Factoring for Testability

Opportunity trigger handler populates “Priority” based on Industry

- Pull out discrete units of logic
- Flatten decision tree
- Abstract logic
- What could be reused?
- What could be called by another client - like a test?

See GitHub for full code!

```
            for (Opportunity o : opportunities) {
             if (accountMap.containsKey(o.AccountId) && (mode ==
OpportunityMode.MODE_OVERWRITE || String.isBlank(o.Priority__c))) {
                Account a = accountMap.get(o.AccountId);

                if (priorities.containsKey(a.Industry)) {
                    o.Priority__c = priorities.get(a.Industry);
                } else {
                    o.Priority__c = getAmountPriority(o.Amount);
                }
             }
        }

    private String getAmountPriority(Decimal amount) {
        if (amount > 100000) {
            return 'Prioritized';
        } else if (amount > 20000) {
            return 'Normal';
        }

        return 'Low';
```

Ensuring Access for Test Code

Unit tests as API consumer

- Consider exposing smaller units of public API
- Consider exposing private API and state with `@TestVisible`
- Isolate, and limit, external dependencies
- Maintain awareness of unique limitations of platform around testing

Example 2: Exposing State and Private API to Tests

How do we test what we factor out?

Expose methods to unit test, or promote to public API:

```
@TestVisible
private String getAmountPriority(Decimal amount) {
```

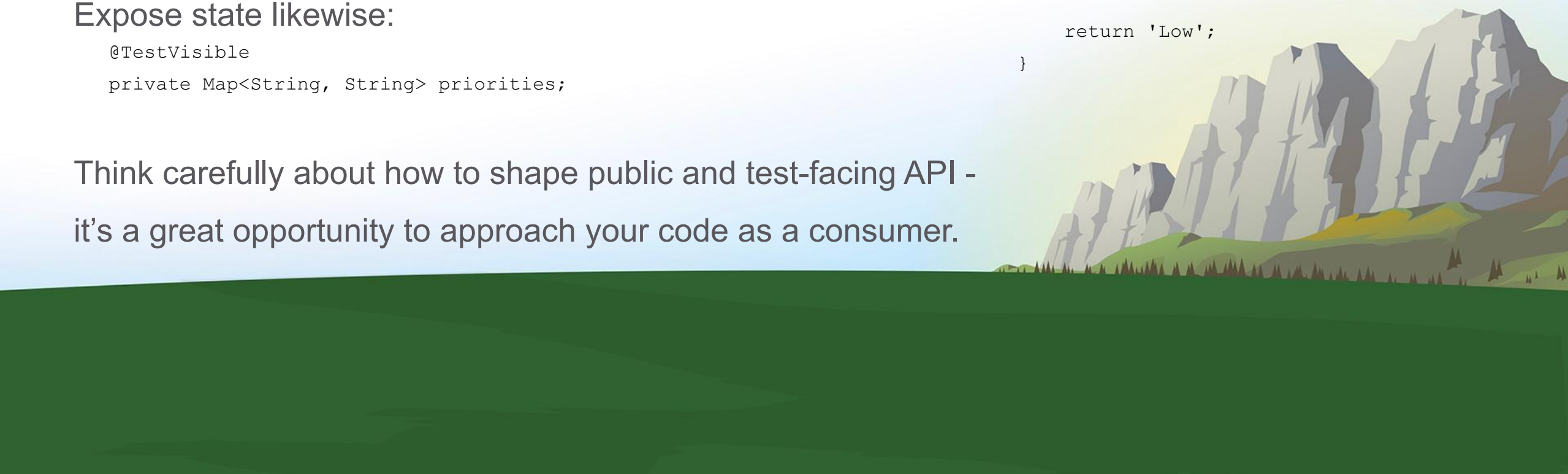
Expose state likewise:

```
@TestVisible
private Map<String, String> priorities;
```

Think carefully about how to shape public and test-facing API - it's a great opportunity to approach your code as a consumer.

```
    private String getAmountPriority(Decimal
amount) {
        if (amount > 100000) {
            return 'Prioritized';
        } else if (amount > 20000) {
            return 'Normal';
        }

        return 'Low';
    }
```



Interlude: Dependency Injection and Mocking

A Quick Introduction

Dependency Injection: Better Isolation for Tested Code

- Isolate external dependencies in helper or utility classes
- Replace those class instances in test context with a mock version.
- Mock returns controlled data and does not execute external functionality.

Example 3: Dependency Injection with Custom Metadata

Several patterns, but common theme

- Factor dependency into separate class
  - Dependency might be a SOQL query, DML, Chatter API call, etc.
- Store instance of dependent class
- Allow test to replace instance with a mocked copy.

- Several routes to implementation
  - StubProvider
  - Inheritance-based
  - Interface-based

Example 3: Dependency Injection with Custom Metadata

Interface-based demo: main class contains interface, “delegate” or “helper”

```
@TestVisible
private IOppServiceQueryDelegate delegate = new OppServiceQueryConcreteDelegate();

@TestVisible
private interface IOppServiceQueryDelegate {
    List<Industry_Priority__mdt> getPriorityMetadata();
}

@TestVisible
private class OppServiceQueryConcreteDelegate implements IOppServiceQueryDelegate {
    public List<Industry_Priority__mdt> getPriorityMetadata() {
        return [SELECT Id, Industry__c, Priority__c FROM Industry_Priority__mdt];
    }
}
```

Example 3: Dependency Injection with Custom Metadata

Interface-based demo: mock class implements interface, returns controlled data

```
private class MockQueryDelegate implements OpportunityServiceV2.IOppServiceQueryDelegate {
    public List<Industry_Priority__mdt> getPriorityMetadata() {
        return new List<Industry_Priority__mdt>{
            new Industry_Priority__mdt(
                Priority__c = 'Prioritized',
                Industry__c = 'Spaceflight'
            )
        };
    }
}
```

Example 3: Dependency Injection with Custom Metadata

Interface-based demo: unit test using mock

```
public static void test_v2_sets_priority_from_custom_metadata() {
    Opportunity o = new Opportunity(AccountId = [SELECT Id FROM Account WHERE Industry = 'Spaceflight'].Id,
       Name = 'Expansive Deal', CloseDate = Date.today(), StageName = 'New' );
    OpportunityServiceV2 os = new OpportunityServiceV2();

    os.delegate = new MockQueryDelegate();

    Test.startTest();
    os.updatePriorities_v2(new List<Opportunity>{o}, OpportunityServiceV2.OpportunityMode.MODE_OVERWRITE);
    Test.stopTest();
    System.assertEquals('Prioritized', o.Priority__c, 'opportunity priority');
}
```

Advanced Testing Strategies

Key Implementation Questions

- “How do I execute that in a test?”
  - Careful data design
  - Decomposing asynchrony - play ball with the platform
  - Execute Batches and Queueables synchronously
  - Dependency injection

- “How do I control the test environment?”
  - Mocking
  - Dependency injection

Exception Handlers and Failure Paths

Control input data and parameters

```
try {
    insert new Opportunity(AccountId = a.Id, StageName =
b);
} catch (Exception e) {
    // …
}
```

Pass `null` parameter to force a `DMLException`.

Exception Handlers and Failure Paths

Factor out resource access and inject dependency

```
try {
    List<Contact> contacts = [SELECT Id FROM Contact WHERE
AccountId = :myId FOR UPDATE];
} catch (QueryException e) {
    // …
}
```

Factor this query out and use dependency injection to force an exception - not directly testable.

Complex Asynchronous Code

Some of the toughest testing challenges

- `Test.stopTest()` only supports 1 layer of asynchrony - must decompose complex code.
  - Schedulable calls Batch → two unit tests. (And/or dependency injection).
- No support for chaining `Queueable` or `Batch` classes in a test.
  - Must dependency-inject or gate with `Test.isRunningTest()`
- No support for multiple batch invocations.
  - Control test data suite.
  - Batches on metadata like User require dependency injection or test gating.
- Serialization makes state inspection and dependency injection more challenging.
  - Can't hold a reference to Batch or Queueable instance variable through invocation.

```
MyBatch b = new MyBatch(delegate);
Test.startTest();
Database.executeBatch(b);
Test.stopTest();
System.assertEquals(20, delegate.recordCount);
```

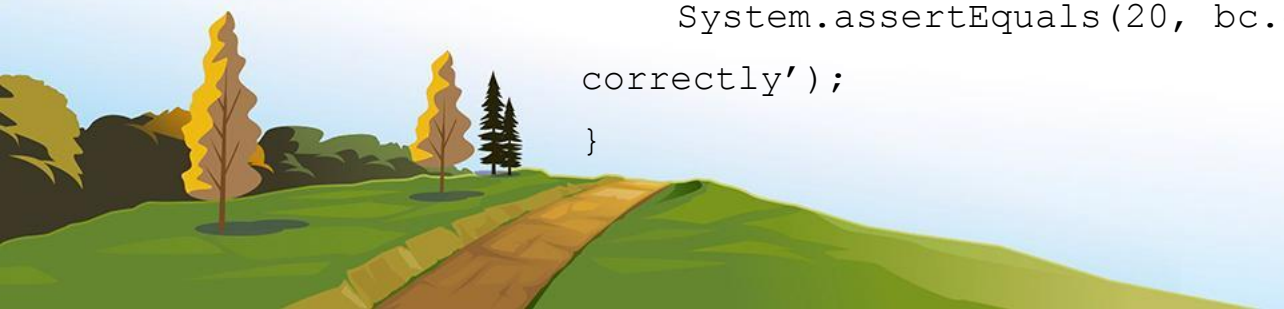
Complex Asynchronous Code

Pragmatic solutions

- Call `execute()` synchronously to allow dependency injection, review of state
- Test multi-layer asynchrony piece by piece.
- Factor logic into a helper class that can be tested synchronously.

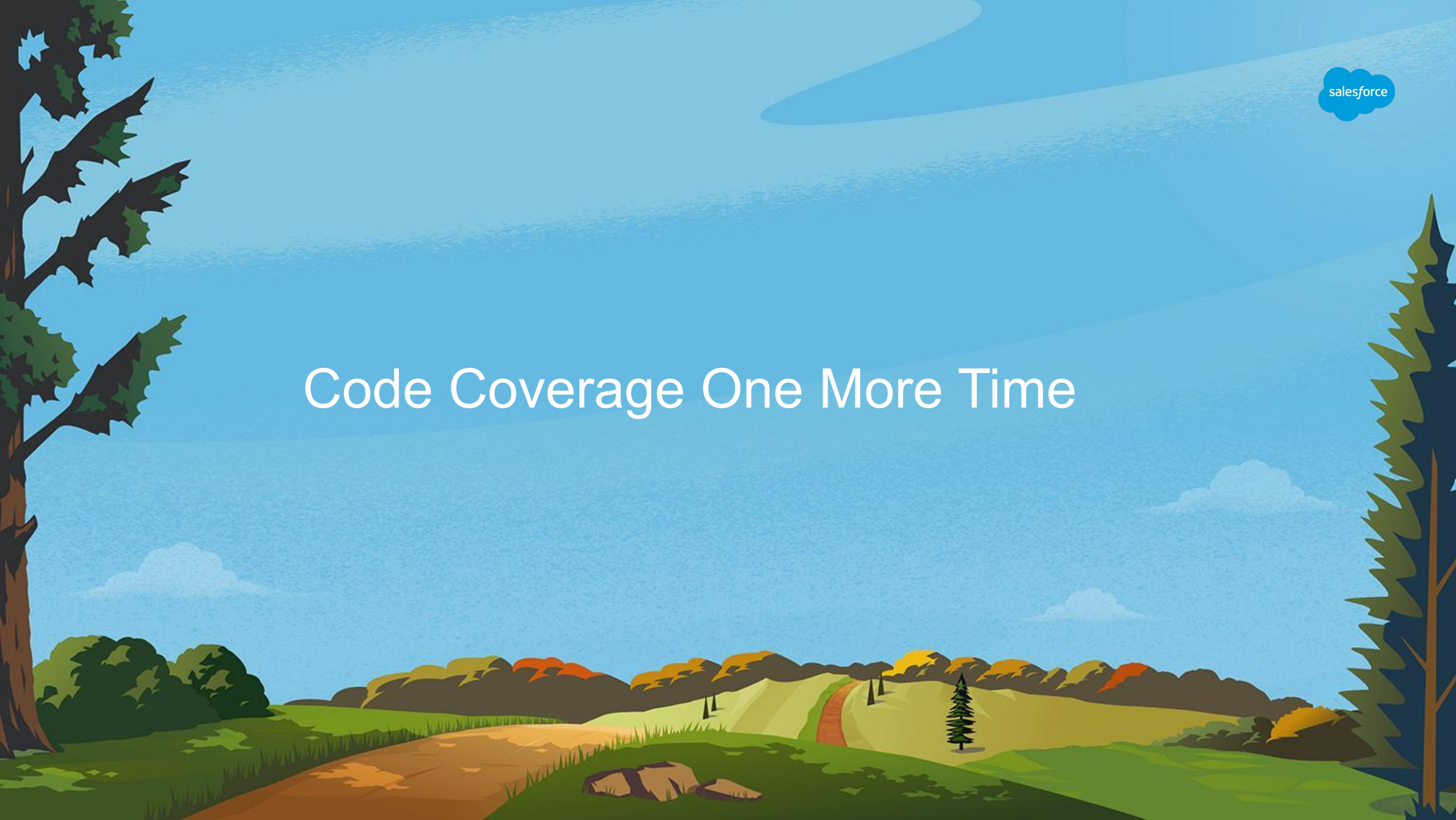
```
@isTest
public static void test_batch_class_state_stored() {
    MyBatchClass bc = new MyBatchClass();
    List<Account> accounts = [SELECT Id, Name FROM Account LIMIT 20];

    Test.startTest();
    bc.execute(null, accounts);
    Test.stopTest();
    System.assertEquals(20, bc.countOfProcessedRecords, 'state stored
correctly');
}
```



Other Challenges

- Code using the ConnectAPI (Chatter in Apex) requires `seeAllData=true` to test.
  - Validate behavior with dependency injection.
- Outbound callouts require Mocks.
  - Good platform support.
  - Use Static Resources to supply fixed web service responses.


salesforce
Code Coverage One More Time

The Secret to 100% Code Coverage

- Build well-structured, testable code.
- Write quality test code for each distinct code path.
  - Understanding and accounting for the unique testing strategies of the Salesforce platform.

- **100% code coverage comes for free.**
- If you don’t hit 100%, you missed a code path.


THANK YOU

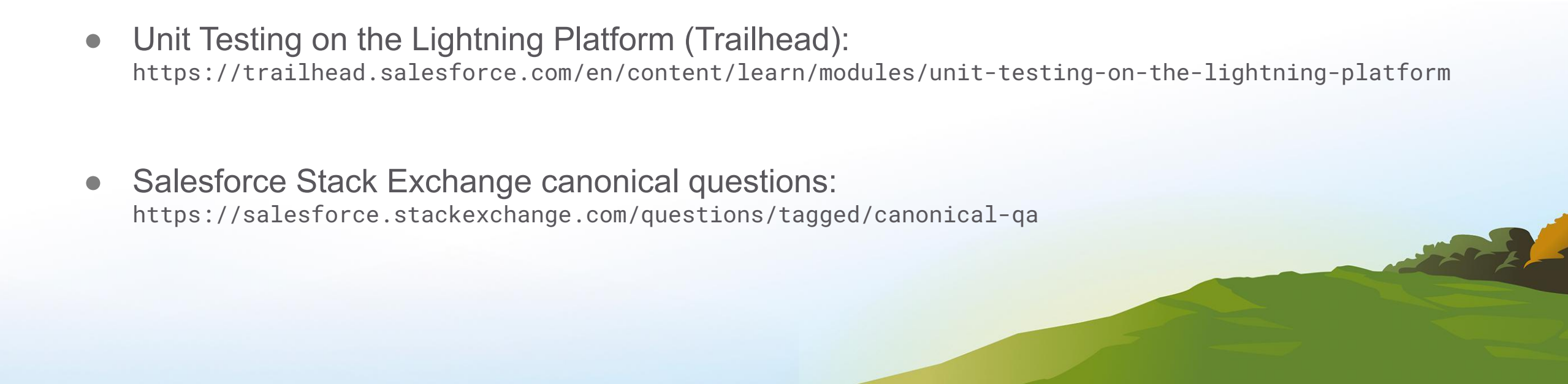
Resources

- Repo for This Talk (contains full example code):
  `https://github.com/davidmreed/phillyforce-testing-examples`

- Month of Testing Series (Salesforce Developer Blog):
  `https://developer.salesforce.com/blogs/2018/05/why-we-test.html`

- Unit Testing on the Lightning Platform (Trailhead):
  `https://trailhead.salesforce.com/en/content/learn/modules/unit-testing-on-the-lightning-platform`

- Salesforce Stack Exchange canonical questions:
  `https://salesforce.stackexchange.com/questions/tagged/canonical-qa`



Text to share feedback on Real-World Unit Testing: Get to 100% Coverage, The Right Way with David Reed

Powered by

mogli sms
INTELLIGENT TEXT MESSAGING

(833) FORCE25
(833) 367-2325

BLAZE5

Testing is Cheap

(with a long time horizon)