Mastering Application Development with Force.com

This book will help you enhance your skillset and develop complex applications using Force.com. It gets you started with a quick refresher of Force.com’s development tools and methodologies, and moves on to an in-depth discussion of triggers, bulkification, DML order of operations, and trigger frameworks. Next, you will learn to use batchable and schedulable interfaces to process massive amounts of information asynchronously. You will also be introduced to Salesforce Lightning and cover components in detail.

Moving on, the book will focus on testing various apex components: what to test, when to write the tests, and—most importantly—how to test. Next, you will also use command-line tools to authenticate and access the Force.com REST sObject API and the Bulk sObject API.

Finally, you will take an in-depth look at the overarching best practices for architecture (structure) and engineering (code) applications on the Force.com platform.

Who this book is written for
If you’re a Force.com developer with a fundamental understanding of Apex and JavaScript but now want to enhance your skills with tips, tricks, and guidance on the best architectural and engineering practices, then this book is for you.

What you will learn from this book
- Creating triggers, and the best practices for writing them
- Developing a number of Visualforce controllers and extensions
- Integrating third-party web APIs into your Apex code
- Writing unit tests that cover testing bulk safety, user and profile settings, and negative tests
- Creating a Lightning component for use in Salesforce1
- Getting acquainted with various methods for deploying metadata between orgs
- Using the Rest sObject API for access to Salesforce objects and data using the REST methodology
- Learning about overarching architectural considerations such as naming conventions, testing practices, and data modeling practices

Design and develop state-of-the-art applications using Force.com’s powerful development platform

Kevin J. Poorman

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Writing Efficient and Useful Unit Tests'
- A synopsis of the book’s content
- More information on Mastering Application Development with Force.com
Kevin J. Poorman has been working with the Salesforce1 platform since 2008. He has architected and built a number of applications on the platform for enterprises and fortune 500 companies in the advertising industry. He has been a Force.com MVP for 3 years. He is active in the community as a developer user group leader, and he can also regularly be found blogging at http://www.codefriar.com, on Twitter (@codefriar), and on the #salesforce IRC channel as well as on the Salesforce developer forums and Salesforce stack overflow. He has also created and maintains the ngForce library for writing Angular.js apps on the platform. In 2015, Kevin joined the Salesforce Marketing Cloud as a senior customer success architect helping marketers succeed at mobile marketing. He regularly speaks on mobile development using Salesforce1, the Salesforce Marketing Cloud, Ionic, and RubyMotion.
Preface

Salesforce.com's platform is one of the most exciting and unique development platforms for business applications. A lot of Salesforce development can be done declaratively without writing code, but to truly master the platform, you'll need to be able to develop not only declaratively, but also with code. Ultimately, you will need to know when to use which toolset—declarative or code.

It is relatively easy, in a world with Salesforce developer forums, Stack Overflow, and user groups, to find others who have faced the same issues you're facing. As a developer, it's likely that you can cobble together a solution from posted solutions. Understanding and tweaking those posts into a solution for your particular problem, however, requires a greater mastery of the platform.

This book is all about mastering the platform; taking your skills as a developer and tuning them for the unique features of the Salesforce platform. We'll discuss the architecture and code and which tool to use for the job. It's going to be awesome. So let's get started…

What this book covers

Chapter 1, A Conceptual Overview of Application Development on the Salesforce1 Platform, is a quick refresher of the Force.com development tools and methodologies. We'll discuss the concepts of classes, triggers, and unit testing that we'll be mastering in the further chapters.

Chapter 2, Architecting Sustainable Triggers Using a Trigger Framework, will dive deep into why you would need a trigger, when you should and should not use a trigger, and how to architect triggers for maintainability. Additionally, we'll dig into trigger frameworks that provide cleaner, more scalable solutions that solve many of the problems that plague traditional trigger development.
Preface

Chapter 3, *Asynchronous Apex for Fun and Profit*, is all about Apex classes that implement the batchable, scheduleable, and queueable interfaces as well as the @future method Annotation.

Chapter 4, *Lightning Concepts*, discusses the four new features of the Salesforce platform that carry the Lightning moniker. We’ll start with Lightning connect and move on to cover process builder, app builder, and lightning components.

Chapter 5, *Writing Efficient and Useful Unit Tests*, talks about unit testing, which is the single most important activity an application developer has to master. However, writing unit tests is rarely seen as exciting. In this chapter, we’ll look at how to write useful unit tests that help us maintain our application over time.

Chapter 6, *Deploying Your Code*, takes you to the next step—you’ve written and tested your code, now what? This chapter is a tour of the many ways to deploy your application metadata from one org to another. Specifically, we’ll cover the Ant migration toolkit, IDE deployment, and Change sets. Additionally, we’ll briefly touch on packaging as a means of deploying metadata.

Chapter 7, *Using, Extending, and Creating API Integrations*, demonstrates how to use the sObject and bulk APIs provided by Salesforce as well as how to create your own custom REST endpoints with Apex. Finally, we’ll build out a set of classes to make calling external REST APIs as painless as possible.

Chapter 8, *Team Development with the Salesforce1 Platform*, discusses and works through the pitfalls of team development in general and the unique solutions available to us on the Salesforce1 Platform.

Chapter 9, *My Way—A Prescriptive Discussion of Application Development on Salesforce1*, looks at overarching best practices for Architecture and Engineering of applications on the Force.com platform in depth. Specifically, we discuss the nature of keeping things simple, testing things well, naming things intuitively, and writing maintainable code. While the rest of the book has been descriptive of the best practices, this chapter is an opinionated prescription for developing Salesforce1 applications.
5 Writing Efficient and Useful Unit Tests

Unit testing is the single most important skill an application developer can master. Sadly, writing unit tests is rarely seen as exciting, let alone important. This chapter will focus on testing various Apex components. We will discuss what to test, when to write the tests, and most importantly, how to test. We will cover the following topics:

• Why do we write unit tests?
• What to test—discussing positive, negative, and role-based testing
• When do we write unit tests?
• How do we structure tests for speed and code reuse?
• Mocking
• Tips and tricks for efficient testing
Why do we write unit tests?

Often, it seems that the answer to why do we write unit tests is because Salesforce makes us do it! This, however, is not the reason we should be writing unit tests. All classes in production orgs have to have 75% code coverage and all triggers must have some coverage. Salesforce enforces this for a number of reasons. As the platform evolves and new features are added, Salesforce needs to ensure that your applications will continue to run without an issue. To do this, they employ the hammer, a specialized test harness that allows them to run every unit test in every org twice. First, the hammer runs every test in every org on the current version of the platform. These same tests are then run on the pre-release version of the platform. This helps them ensure that your code not only runs on the new version of the platform, but that it also runs at least as efficiently as it did on the earlier platforms. Even if your unit tests have no assertions and utterly fail to do anything but run code, the hammer's execution of your tests provides valuable insight into the backward compatibility and efficiency of the platform. This illustrates an important point. Contrary to popular belief, the true cost of software is not in development but in maintenance. The Apex software on the Salesforce1 platform is no exception. While there are upfront development costs, the true cost of the software adds up as after months and years after the initial implementation, developers try to add features, fix bugs, or change the implementation of objects and functions. Without hammer and the unit tests it runs, the cost of upgrading the Salesforce platform would be astronomical! Not only for Salesforce, but for every client running code on the platform. With that in mind, let's look at some concrete positive reasons for writing unit tests.

Proving functionality

The classic reason for unit testing is to prove that the code works as you believe it does. Given an addition method accepting two integer numbers we can assert the output of the method when we know the input. Proving the functionality of our code is the foundation of our testing philosophy because it gives us assurance not only that our code's architecture, engineering, and logic are correct, but also that we understand how that code interacts within the greater system.
Reducing the cost of change
As a system evolves, features are added, deprecated, and changed. It's relatively easy for knowledgeable developers to add features. However, it's difficult to safely deprecate features. The highest risk to a system, however, comes from changing the implementation of the given features. This is where unit tests can save vast amounts of time. Imagine being tasked with replacing the implementation of a complex `calculateIncomeTax` method. The current implementation and the new implementation are functionally identical, but your new method is 5x faster. Because both implementations are expected to function identically existing tests are valid proofs of functionality for both implementations. Because of this, replacing the implementation of a complex method becomes safer and easier. Make a change to the implementation, and so long as the tests pass, you can be confident you've not introduced a bug.

Encouraging modular, reusable code
As you start to write unit tests for your code, you'll quickly learn that poorly designed, tightly coupled code is difficult to test. Writing tests to ensure software quality and maintainability, rather than to meet deployment requirements, will force certain design choices on your code. You'll end up with code that is more loosely coupled, with less reliance on state, and is generally simpler. Such code is easier to test, and is therefore easier to functionally prove. Additionally, because it no longer requires maintaining state or tightly coupled relationships with other objects, this type of code is almost always reusable code, which is never a bad thing.

Identifying engineering bugs you didn't write
If you're writing software for a living, you're working to solve complex real-world problems, not to prove your understanding of computer science principles. The situations and problems you're tasked with solving are not simple problems, but nuanced, intricate problems complex enough to justify custom software. This can lead to seemingly crazy bugs that are incredibly hard to solve. There are situations where your code fails because of some other bit of code that fails only on the last day of the month or at midnight on Thursdays. I call these Cinderella bugs because they only happen at midnight on the night of the ball. Writing unit tests help you identify these situations before you experience them in production. Granted, you'll need to show some creativity when writing your tests, but learning to test your billing code at midnight on Thursdays is generally the kind of lesson you only need to learn once.
Documenting expected behavior

It's a truism that the job you have now is likely not your first, nor your last. Inevitably we will face the daunting task of learning a new code base. On top of that, outside of a computer science textbook, when have any of us ever encountered a code base that consistently followed best practices as well as proper and sensible object oriented abstractions? Indeed. When I was working as a consultant, coming up to speed on a org's code base was a weekly chore. One of the best ways I've ever found to get up to speed is to read through the tests. Well written tests demonstrate not only what the code does but also what kinds of data are expected and required for that code to run. Fantastic tests will even walk you through failure scenarios and their causes. Even badly written tests give insight into what the developer was thinking the code should do.

Tests + code = less likely to produce bugs

Even if you only write basic unit tests to prove your code functions as you expect, you've decreased the chances of having bugs in your code. Why? There are a few reasons, but the basic principle is that you're less likely to make a logic mistake in two places than you are in one place. With each test you write, you're decreasing the risk that you've got a logic bug somewhere in the code you are testing. Writing good tests helps by not only exercising your code, but also by helping you think through your code's interaction with the greater system. This is especially true if you write your test code first! Forcing yourself and your team to design new functionality up front causes you to pause and consider what existing code you can reuse as well as consider the impact this functionality has on the greater system.

What not to test?

I'm often asked by developers who are new to the platform what kinds of thing they should write unit tests for. In general, I find that question rather frustrating. Many factors go into answering that question, and without knowing the specifics of your org, it's impossible to answer comprehensively. On the other hand, it's much easier to answer the question what not to test? While the answer to that question isn't cut and dry either, it's at least simpler to provide guidelines for. With that in mind, here are some things you should most likely not test.
Managed package code
Managed packages have to provide their own unit tests. Additionally, you don't get to see the code from managed packages, so writing unit tests for managed package code is not only difficult, it's fruitlessly redundant as well. However, this isn't carte blanche for not testing your code that interacts with the managed package code. The only time I write unit tests for managed package code, is when I'm documenting a bug in that managed package!

Standard platform features
I know this one seems obvious, but I've come across it numerous times. There's nothing to be gained from testing the basic constructs of the Apex language or built-in system-defined classes and functions. It is possible to test whether or not \texttt{if()} works, but what's the point? What you want to be testing isn't \texttt{if()} but the logical tests that go into that \texttt{if()} statement. Also in this category, we have declarative features, such as rollup summary fields, formulas, and validation rules. We write unit tests in part to prove functionality, but we need not write tests to prove platform functionality.

Aside from testing managed package code and standard platform features, the answer to what to test is \textit{everything}, unless it doesn't make sense Testing, just like software development, requires judgment calls, and if something doesn't seem like it makes sense to test, it probably doesn't. However, you should always maintain the posture of testing everything and ruling out only what clearly doesn't make sense.

When to write unit tests
Adherents of test-driven development argue for writing the tests first. Indeed, the idea behind test-driven development is that tests define acceptance of code. In general, the \textbf{Test-driven development} (TDD) approach is to write a test, see it fail, and then write just enough code to make the test pass. In practice, this is easier said than done. Others argue that you should write your code and then the tests because it's more efficient. I find the most successful developers almost always develop the code and the tests simultaneously, the code influencing the tests and the tests ensuring that the code is written in a maintainable, testable way. So long as the tests are written, and written well, I find it hard to argue for or against writing the tests first. Indeed, I find it most flexible to write the tests at the same time as the code. The central question of unit testing your code isn't when to test or what to test, but how to test.
Structuring unit tests

In general, there is only one overarching pattern for unit testing code on the Salesforce1 platform. The essential differences between the Salesforce1 platform and other software development stacks does make unit testing is a bit different. That said, if you're familiar with unit testing in other languages, this will largely seem familiar to you. Here's the general pattern:

![Diagram of unit testing process]

Using your own data

Each of our test methods needs to create its own test data. While this seems cumbersome and time consuming, it's the only safe way to run unit tests. Relying on existing data in your org is problematic because you cannot ensure that the data will exist in your other orgs. Assuming an account will exist in your production org just because it exists in your development sandbox is a great recipe for frustrating deployments. This often finds its way into our tests through the use of hardcoded IDs. Querying for an account where the ID is x, for instance, not only assumes that the account exists in other orgs, but also that its fields contain the same values, including the ID field.
While creating your own test data can seem cumbersome and time consuming, there are a couple of things you can do to help with that. First and foremost, use a test factory. Test factories provide easy-to-remember methods for generating your test data. Additionally, they reflect the metadata of your org at runtime, allowing the test factory to dynamically know which fields are required and how to populate them intelligently. While there are a number of test factory libraries available, I'm a fan of Daniel Hoechst's Salesforce Test-Factory, which you can find on GitHub at https://github.com/dhoechst/Salesforce-Test-Factory. With his test factory in place, generating your own data for your tests is as simple as a call, like this:

```java
// Calling TestFactory will pre-fill all the fields we need
Account a = (Account)TestFactory.createSObject(new Account());
insert a;

// You can also manually set specific values to be used.
// Any values set in the constructor will override the defaults
Opportunity o = (Opportunity)TestFactory.createSObject(new Opportunity(AccountId = a.Id));

// You can also specify a specific set of overrides for different scenarios
Account a = (Account)TestFactory.createSObject(new Account(), 'TestFactory.AccountDefaults');

// Finally, get a bunch of records for testing bulk
Account[] aList = (Account[])TestFactory.createSObjectList(new Account(), 200);

// You can optionally insert records as created like this:
// Note the final parameter of true.
Account a = (Account) TestFactory.createSObject(new Account(), true);
Contact c = (Contact) TestFactory.createSObject(new Contact(AccountID = a.Id), true);
```
Even with a test factory in place, it can be cumbersome to have twenty lines of code in each test simply setting up data. Here's where the second tip comes into play. While the test factory can provide specific sObjects for us, we often need a collection of related objects to test with. For instance, we may need an opportunity with opportunity line items associated with it for our test. Normally, that would require creating not just the opportunity and opportunity line items but also the account and any contacts we may need. In these situations, it makes sense to establish one or more customized test factories that rely on and build on the functionality of the basic test factory. I like to ensure that everything is easy to find, so I tend to have many sObject specific test factories. Consider the following instance:

```java
@isTest
public without sharing class OpportunityTestFactory {

    public static genOppWithLineItemsAndAccount(Integer numLineItems, Boolean doInsert) {
        Account a = (Account) TestFactory.createSObject(new Account(), doInsert);
        Opportunity o = (Opportunity) TestFactory.createSObject(new Opportunity(accountId=a.id), doInsert);
        List<OpportunityLineItem> olis = new List<OpportunityLineItem>();
        for (Integer i = 0; i < numLineItems; i++) {
            olis.add((OpportunityLineItem) TestFactory.createSObject(new OpportunityLineItem(OpportunityId=o.id)))
        }
        if (doInsert) {
            insert olis;
        }
        return o;
    }
}
```

These kinds of customized test factories grow over time but having them means it's easy to add additional custom methods and easier to write tests using your own data. Two tips on custom test factories. First, always annotate them as @isTest at the class level. This ensures that the code cannot be called in a non-test environment and that it doesn't count against your overall code coverage. Secondly, note how I've maintained the optional insert flag for the method in the preceding code. If your test doesn't rely on triggers and other data manipulation language (DML) based operations, you can often write faster running tests by not inserting or querying. Oftentimes, however, you will have to insert the data, and maintaining that simple Boolean flag in your custom factory methods makes it that much easier to do.
Starting and stopping your tests

One of the unique features of the Salesforce1 platform is its governor limits. These limits are intended to keep you from writing inefficient code and, more importantly, ensure that no one org is using vast amounts of resources at the detriment of other org’s. Having said that, they can often cause problems when writing tests. After all, if you’re creating your own test data, and testing in bulk, you may very well run into the max DML or max query count limits. This is why Salesforce created the platform-unique `startTest` and `stopTest` methods. These methods do more than just delineating the portions of your test that exercise your code; they provide you with two invaluable services. First, calling `startTest()` will reset your governor limits and limit them to just the code between `startTest()` and `stopTest()`. This means that you are free to create your test data without the creation of that data causing you to hit a governor limit. Additionally, it means that when the test runs, the governor limits summary at the end of the log is a summary not of the test itself, but of your tested code. This is incredibly useful for identifying processes and methods that, for instance, sometimes hit the execution time limit.

Secondly, calling `stopTest()` forces all asynchronous code to complete before returning to the test. This is invaluable as you write asynchronous code, such as `@future` methods and queueable Apex. This means that `stopTest()` will force any asynchronous work to complete not at some point in the future, but right now so that you can continue to test. This allows you to follow the same pattern for testing regardless of the tested code’s synchronicity.

Executing your code

After creating test data and calling `test.startTest()`, we are now left with executing our code. This is where our general testing pattern branches into three subpatterns. Each of these subpatterns does execute the code, but each in their own unique and important way. Of the three, the first subpattern is the easiest to understand and what you’re most likely to have run across. I call this types of tests Positive tests because they’re based in the positive assumption that your code is working as intended with valid data. The second subpattern is less intuitive, but far more powerful. I call these Negative tests because they test what happens when exceptions occur. Lastly, there are Permissions tests that test how your code functions with different users, roles, and permission sets. Let’s take a more detailed look at all three.
Positive tests

Positive tests prove expected behavior. That is, they prove that the code functions as it is intended to when given valid input. Succinctly stated, this subpattern states that when given proper inputs, expected outputs are received. Care should be taken to evaluate all valid paths through the code. If your code is structured with conditional logic statements, ensure that you positively test each and every viable path through your code. Let's take a look at some class code and its test to illustrate this:

```java
public class exampleCode {
    public Integer add(Integer one, Integer two) {
        return one + two;
    }
}
```

```java
@isTest
private class exampleCode_Tests {
    @isTest static void test_Add_Postive() {
        exampleCode drWho = new exampleCode();
        Test.startTest();
        Integer testValue = drWho.add(5, 7);
        Test.stopTest();
        System.assertEquals(12, testValue, 'Expected 5+7 to equal 12');
    }
}
```

Our simple `add()` method accepts two integer arguments and returns a single integer in response. Our test follows our basic test pattern, but the simple nature of this method means we don't need to create our own test data here; we can just pass in two integers. You'll note, however, that we're creating the `ExampleCode` object before we start the test. Because this is a positive test, we'll provide the add method with two valid integers, in this case, 5 and 7. Because we're setting the input parameters, after a couple cups of coffee, we can do the math ourselves and assert that the value returned is 12. This is an extremely simple but clear example of how positive tests work.
Negative tests

Negative tests are less intuitive and require some additional setup. In return, negative tests prove that our code safely handles errors and exceptions. Because we're intentionally providing inputs to the code that will cause an exception to occur, we need to find a way to safely capture the exception without failing the test. Here, the subpattern works by setting a Boolean variable, say didPass to false in the setup of the test, and executing our to-be-tested code inside a Try/catch block. When our inputs cause the test to throw the exception, our test captures that in the catch block. If the type of exception and messaging we catch matches what we expect, we can set didPass to true. Outside of the Try/catch block and after the stopTest() call, we can assert that our didPass variable is true. That's a lot to take in, so let's look at the following code example:

```java
public class exampleCode {
    public class exampleCodeException {
    }
    public static Integer division(Integer one, Integer two) {
        if (two == 0) {
            throw new exampleCodeException("Dividing by zero makes kittens cry");
        }
        return one / two;
    }
}

private class exampleCode_Tests {
    @isTest static void test_Divide_Negative() {
        Boolean didCatchProperException = false;
        Test.startTest();
        try {
            exampleCode.divide(1, 0);
        } catch (exampleCode.exampleCodeException AwesomeException) {
            didCatchProperException = true;
        }
        Test.stopTest();
        System.assert(didCatchProperException,
                      "Properly caught custom Exception");
    }
}
```
We've added a division method to our `ExampleCode` class, and astute readers will notice the addition of a new custom exception class, `exampleCodeException`. In our test, we follow the same pattern as earlier of starting the test and executing our code. However, in this situation, we wrap the code execution in a `Try/catch` block. We're calling our divide method with an intentionally bad second integer, 0. Our `divide` method is on the lookout, however, for the second parameter being 0. Instead of letting the system throw a divide by zero error, our code throws an `ExampleCodeException` method. It's this `exampleCodeException` method that we're trapping in our test's `catch` block. This allows us to be certain that not just any exception is caught here. If, for instance, a divide by zero exception somehow still occurred, that exception would not be caught and the test would fail. In the end, we set `didCatchProperException`, our previously defined test Boolean to `true` so that we can assert that we did indeed catch the proper kind of exception.

Negative tests are considerably more powerful in the long run, helping ensure modifications still properly handle known exception situations properly. This is especially helpful when modifying the implementation of class methods. Additionally, because they're testing how the code reacts to invalid data, they are the de facto way of testing situations, such as invalid user entered data, fields that legitimately have apostrophes in them, and responses to third-party external web services that your code uses. Does your code properly handle the last name O'Connel? How about a fellow developer accidentally linking a task by `WhatId` instead of `WhoId`? And how does your exchange rate API code handle the sudden lack of Internet connection due to severe weather? Negative tests ensure that you can answer these questions. More importantly, negative tests prove that you've defensively developed against at least those failure scenarios that were identified.

**Permissions-based tests**

Permissions-based testing ensures that your sharing and security model works as you expect it to. Sharing rules and security options are some of the most complicated aspects of Salesforce1 platform development. These are, therefore, some of the most important tests you can write. But wait you say! Isn't writing tests of the security and permissions model just a form of testing native platform functionality? Kind of. Unless otherwise specified, Apex code runs in a system context. In essence, this means that Apex code normally ignores the users' permissions and sharing rules. These tests help us ensure that the permissions and sharing rules are honored by our Apex code.
Effectively, permission tests follow the same general pattern as other tests with one small twist. When we go to execute the code, we'll execute that code with a different user. We're not just going to pick a user at random either. The user we run the test with becomes a crucial data point that we create. To actually run the test with our newly created user, we call the Salesforce1 platform's System.runAs() method. The RunAs method accepts a single user parameter and a block. Anything in that block of code is executed as with the profile, role, and permission sets of the user specified.

Let's look at a basic use case for runAs():

```java
@isTest static void test_getBankAccount_AsUserWithTimeLord() {
    User u = UserTestFactory.getUserWithProfile('TimeLord');
    System.runAs(u){
        Test.startTest();
        // This is executed as our user with the Timelord
        // profile
        Test.stopTest();
    }
    // Assertions
}
```

In this test method, we create a user with a call to our UserTestFactory. This user is set up with a given profile, in this case, the profile TimeLord. This sets us up to test our code as TimeLord. Now, when we execute our code, we can ensure that this profile's permissions are honored.

With this subpattern, we can test not only profiles, but also sharing rules and permission sets. Perhaps more importantly, we can test them both positively and negatively! Let's look at some examples. First, let's update our ExampleCode class with a method we want to permission test:

```java
Public class exampleCode {
    Public class exampleCodeException{}
    Public Integer getBankAccount(Account a){
        // SuperSekr3tBankAccountNum__c is an encrypted field
        a = [SELECT superSekr3tBankAccountNum__c
             FROM Account
             WHERE ID :a.id];
        If(String.ValueOf(a.superSekr3tBankAccountNum__c).contains('*')) {
            Throw new exampleCodeException('Nope!');
        }
        return a.SuperSekr3tBankAccountNum__c;
    }
}
```
Writing Efficient and Useful Unit Tests

Here's a positive test:

```java
private class exampleCode_Tests {
    @isTest static void test_getBankAccount_Positive() {
        exampleCode drWho = new exampleCode();
        User u = UserTestFactory.getUserWithProfile('TimeLord');
        Account a = (Account)TestFactory.createSObject(new Account());
        Integer result;
        System.runAs(u){
            Test.startTest();
            result = drWho.getBankAccount(a);
            Test.stopTest();
        }
        System.assertNotEquals(result, null,
            'Expected The Doctor to have access to bank #');
    }

    In this test, we expect a user with the TimeLord profile to be able to access the
    encrypted bank account number field. On the other hand, we want to ensure that
    other profiles do not have access. With this in mind, we can write a negative test that
    looks like this:

    @isTest
    private class exampleCode_Tests {
        @isTest static void test_getBankAccount_UberForNope() {
            exampleCode Dalek = new exampleCode();
            User u = UserTestFactory.getUserWithProfile('Dalek');
            Account a = (Account)TestFactory.createSObject(new Account());
            Boolean didCatchException = false;
            Integer result;
            System.runAs(u){
                Test.startTest();
                Try {
                    result = Dalek.getBankAccount(a);
                } catch(exampleCode.ExampleCodeException e){
                    if(e.getMessage().containsIgnoreCase('nope')){
                        didCatchException = true;
                    }
                } Test.stopTest();
            }
            System.assert(didCatchException, 'Expected Daleks to be blocked');
        }
    }
```
Our `getBankAccount()` method queries for an encrypted field; if the user doesn't have permission to view it, it will return a masked value. If we detect that masked value, we throw an exception. Like our positive test, this test still requires a custom user with a given profile, but in this case we expect an exception.

Importantly, we’re not limited to testing users with different profiles. We can, and should create tests for Apex-based sharing rules, roles, and permission sets. Permission sets are an incredibly powerful and fine-grained tool for extending permissions on a per-user basis, beyond what a user has from their profile. With permission sets, we can, for instance, establish a singular profile for the entire support staff and grant additional privileges to support managers. If you’re not already a fan of permission sets, check them out, you soon will be! Testing permission sets requires just a few extra lines of code to test. This is a prime example, however, of where a custom test factory becomes an invaluable tool. Let’s look at some code as another example of testing permission sets:

```java
@isTest
private class exampleCode_Tests {
    @isTest static void test_getBankAccount_W_PermSet() {
        exampleCode ClaraOswald = new exampleCode();
        User u = UserTestFactory.getUserWithProfileAndPermSets('Standard User', new List<String>{'companion'});
        Account a = (Account)TestFactory.createSObject(new Account());
        Boolean result;
        System.runAs(u){
            Test.startTest();
            result = ClaraOswald.getBankAccount(a);
            Test.stopTest();
        }
        System.assertNotEquals(result, null,
            'Expected ClaraOswald who has Companion Permissions to have access to the bank account');
    }
}
```

As you can see from the code, our test is nearly identical to our positive profile permission test. In fact, the only difference in this test is the custom `userTestFactory` method we called. Because positive permission tests are so similar, you can often group them together in a data structure. This allows you to write a metatest that iterates over your data structure to test various permissions. This greatly simplifies your testing, allowing you to add an element to your data structure, instead of adding entirely new tests whenever a new permission set or profile is created. Here’s how one such meta-test works:

```java
@isTest
public class accountPermTests {

```
public class PermissionTestData {
    Boolean isProfileTest {get;set;}
    Boolean isPermSetTest {get;set;}
    Boolean isPositiveTest {get;set;}
    String profileName {get;set;}
    String permsetName {get;set;}
    String exceptionTypeName {get;set;}
    String exceptionMessage {get;set;}
    String friendlyMessage {get;set;}
    String assertEqualsValue {get;set;}

    public PermissionTestData(Boolean iisProfileTest,
                           Boolean iisPermSetTest, Boolean iisPositiveTest,
                           String iProfileName, String iPermSetName,
                           String iExceptionTypeName, String iExceptionMessage, String
                           iFriendlyMessage iAssertEqualsValue) {
        this.isProfileTest = iisPositiveTest;
        this.isPermSetTest = iisPermSetTest;
        this.isPositiveTest = iisPositiveTest;
        this.profileName = iProfileName;
        this.permSetName = iPermSetName;
        this.exceptionTypeName = iExceptionTypeName;
        this.exceptionMessage = iExceptionMessage;
        this.friendlyMessage = iFriendlyMessage;
        this.assertEqualsValue = iAssertEqualsValue;
    }
}

private List<PermissionTestData> PTD = new
List<PermissionTestData>();

private List<PermissionTestData> setPopulatedTestData() {
    PTD.add(new PermissionTestData(true, false, true,
                           'support', '', '', '',
                           'Expected this test to pass'));
    PTD.add(new PermissionTestData(true, true, true, 'support',
                           'Support Manager', 'ExampleCodeException',
                           'No access for you',
                           'Did not expect this test to pass as the permission set
                           involved should not pass!'));
}

@isTest static void test_getBankAccount_WithPermSets() {
    for(PermissionTestData p: setPopulatedTestData()) {

exampleCode instance = new ExampleCode();
User u;
Boolean didCatchException;
Integer result;
Account a = (Account)TestFactory.createSObject(new Account());
if(p.isPermSetTest && p.isProfileTest
    && p.profileName != '' && p.permSetName != ''){
    u = UserTestFactory.getUserWithProfileAndPermSets
    (p.profileName, new List<String>{p.permSetName});
} else if (p.isProfileTest && p.profileName != ''){
    u = UserTestFactory.getUserWithProfile(p.profileName);
}
Test.startTest();
System.runAs(u){
    if(p.isPositiveTest) {
        result = instance.getBankAccount(a);
    } else {
        try {
            result = instance.getBankAccount(a);
        } catch(Exception e) {
            if(e.getTypeName() == p.exceptionTypeName &&
            e.getMessage().containsIgnoreCase(p.exceptionMessage)){
                didCatchException = true;
            }
        }
    }
}
Test.stopTest();

if(p.isPositiveTest){
    System.AssertEquals(p.AssertEquals, Result,
    p.friendlyMessage);
} else {
    System.assert(didCatchException, p.friendlyMessage);
}

A test setup like this has some upfront costs, namely writing the inner class data structure and thinking through what commonalities exist across your permissions-based tests. In the end though, it's much easier to maintain such a suite of tests, as you can add or modify the .add calls in setPermissionTestData much faster than writing a net-new test.
Assertions

If you take nothing else away from this chapter, let it be this—tests without assertions are not tests, but liabilities. Without the assert calls, you cannot check the outcome of your executed code block to see if it functioned properly. That's why I refer to test methods without assert calls as liabilities. Inevitably, when the code fails for whatever reason, you're accountable for the consequences. At 3am. On a Saturday. It's even worse, if your code is mission critical and fails at the quarter's end! Practice safe testing, use asserts.

The Salesforce1 platform provides us with three basic assertion methods. The first method, `System.Assert(expression)`, evaluates the expression within for a Boolean `true` or `false`. Thus, you can use it like this:

```
System.assert(1 = 1)
System.assert(BooleanVariable)
System.assert(p != np)
```

The other two built-in assertion methods are really shorthand, convenience methods built on top of `System.assert`. They are `System.assertEquals(expected, actual)` and `System.assertNotEquals(expected, actual)`. These often read easier than a simple `System.assert()` call. Here are a few examples:

```
System.assertEquals(1, 1)
System.assertEquals(false, BooleanVariable)
System.assertEquals(P, NP)
```

It's important to remember that each of these built-in assert methods can accept an optional final parameter that I call the friendly message. You should always use the friendly message option, as it will help you debug which assertions and which tests are failing. You will only ever see the friendly message when a test fails. To use the friendly message, simply add a string parameter to the method call; for instance:

```
System.assertEquals(P, NP, 'Oh noes! P = np means all cryptography is flawed!')
```
Creating your own assertion methods

Often, it can be useful to create your own assertion methods. Complex comparisons of objects and situations where multiple assertions must all pass are excellent candidates for custom assertion methods. Creating an assertion method is as simple as creating any other method, but to be used as an assertion, the method must either return true or throw an exception. It's a good idea to maintain the friendly message concept, so it pays to accept a friendly message parameter. If your assertion method needs to throw an exception, use the friendly message as the exception's message. If, for instance, we want to build an assertion method that proves that two contacts are from the same household, we might write it like this:

```java
@isTest
public class customAssertions {
    public class customAssertionException extends Exception {}

    public static Boolean ContactsAreFromSameHousehold(Contact firstContact, Contact secondContact, String friendlyMessage){
        System.assertEquals(firstContact.mailingAddress, secondContact.mailingAddress, 'MailingAddress: ' + friendlyMessage);
        System.assertEquals(firstContact.homePhone, secondContact.homePhone, 'homePhone: ' + friendlyMessage);
        System.assertEquals(firstContact.lovesCrankCalls, secondContact.lovesCrankCalls, 'lovesCrankCalls: ' + friendlyMessage);
        if(firstContact.fullName == secondContact.fullName){
            throw new customAssertionException('Full Names are identical: ' + friendlyMessage)
        }
        return true;
    }
}
```

Here, we're both compiling standard assertions that now must all pass as well as a custom comparison. If any of the three standard assertions called here fail, or if the custom full name comparison fails, an exception is raised and the assertion fails.
We talked about what asserts are and how to use them, so let's take a second to consider when to use them. All tests should have at least one assertion after the call to `stopTest()`. Better tests will have multiple assertions identifying, for instance, that the object returned not only had the proper record type, but that it was properly modified by the executed code. Best yet, are the test methods that include two assertion blocks in addition to the standard after `stopTest()` assertions, the best tests will include a block asserting that your test data was created properly. This is especially important as you try to identify falsely failing tests. If the code works when manually executed via anonymous Apex or the UI, but fails in the tests, you may have a problem with your test data. Asserting that X number of contacts were created helps you identify the problem.

**Mocking**

Mocking is the practice of substituting the standard implementation of classes and methods with stub methods that return a fixed value. Other development stacks have rich and robust mocking capabilities built in. Salesforce1, on the other hand, is slowly expanding into the mocking world. In fact, there's only one built-in mock interface for you to stub. Despite the lack of a robust mocking library built into the platform, the capabilities of the existing mock interface make unit testing HTTP callouts a breeze. Additionally, there are other, third-party, mocking libraries that work with the Salesforce1 platform. Libraries, such as `FFLib_ApexMocks`, found at `https://github.com/financialforcedev/fflib-apex-mocks`, allow you to stub custom objects and methods so long as you have written your class to implant an interface.

Let's take a deeper look at the `HTTPCalloutMock` interface. Like most interface implanting classes, there are required methods for you to implement. In the case of `HTTPCalloutMock`, we must implement the `response()` method. This method must accept a `HTTPRequest` object as it's parameter and must return a `HTTPResponse` object. How we stub that out is up to us. To use our mock object in a test, we simply call: `test.setMock(MockObj)`. After that mock is set, the next callout made will automatically return our stubbed `HTTPResponse` object. Rather than cluttering up my org with dozens of classes implementing the `HTTPCalloutMock` interface, I like to code a factory class that constructs the mock for me:

```java
@isTest
public with sharing class CalloutMockFactory implements HttpCalloutMock {
    Protected Integer code;
    Protected String status;
    Protected String bodyAsString;
    Protected Blob bodyAsBlob;
    Protected Map<String, String> responseHeaders;
}
```
public CalloutMockFactory(Integer code, String status, String body, Map<String, String> responseHeaders) {
    this.code = code;
    this.status = status;
    this.bodyAsString = body;
    this.bodyAsBlob = null;
    this.responseHeaders = responseHeaders;
}

public HTTPResponse respond(HTTPRequest req) {
    HttpResponse res = new HttpResponse();
    res.setStatusCode(this.code);
    res.setStatus(this.status);
    res.setBody(this.bodyAsString);
    return res;
}

With this factory in place I can easily test callouts by using the factory to return a response as it's needed. For example:

    Test.setMock(HttpCalloutMock.class, new CalloutMockFactory(400, 'Invalid Request', ps(GuidResp_Tests.json_error, null));

**Tips and tricks for efficient testing**

Following the general pattern for creating your own test data outlined here: executing the tested code between `startTest()` and `stopTest()` calls and always including asserts in the tests will set you on a good path for useful, robust tests. This doesn't, however, mean that the tests are fast. To help keep tests fast and your test-code feedback loop tight, keep these tips and tricks in mind:

- If you don't need to, don't insert and query data to and from the database. Starting with winter '13, you can set a value for the `Id` field, so long as you don't try to insert it. This allows you to create an object, set its ID, and create other objects that reference it. The slowest part of any web-based application is historically the database, and Salesforce1 is no exception. If you can cut down your SOQL and DML, your tests will run faster.

- Use `@testVisible`. This annotation allows you to quickly and easily annotate private class variables and methods and then access their values or execute the code during tests. Here's an example of it in use:

    public class exampleCode {
        // Private member variable
        @TestVisible
    }
private static Boolean normallyHidden = true;

// Private method
@TestVisible
private static void cantSeeMe(Integer 3) {
    // do amazing things
}

• Find a general mocking library and use it. The more complex your org becomes, the longer it will take to run your tests. If you're mocking out objects and methods whose return values are crucial to the code being tested but not the tested code itself, you not only add stability to your tests, you know what the mock object or stubbed method will return each time! But, you also speed up your code by bypassing all of the code those objects would be running! Additionally, you should mock every single HTTP callout!

• Avoid using @isTest(seeAllData=true). Here be dragons. This annotation allows your tests to view all of the data in your org. SOQL queries run in tests without the annotation cannot see your existing Accounts, Contacts, and so on. When you annotate testMethod with @isTest(seeAllData=true), all bets are off. It's true that some areas of the platform still require seeAllData=true because those objects are not able to be created in Apex code. For example, you cannot create an approval process in Apex. To test your approval process or the code that submits a record for approval, you'll still have to use @seeAllData=true. This also reinforces the idea that you should be creating all of the needed test data in your test methods!

• Consider investing in a continuous integration system. Many Continuous Integration (CI) systems, such as Jenkins or drone.io, work well with the Salesforce1 platform. These systems run your tests for you, at periodic intervals or after specific events, like a Git commit. They help you keep pace with the other developers in your org and allow for integration sandboxes where your team's changes are merged together and the tests run. This helps you identify when another developer has made a change that breaks what you're working on before you try to deploy it!
• Always test your code with bulk data, but change the volume of data per environment. Proving that your code is bulk safe doesn't have to happen in production. Create a custom setting for TestOptions and create a numeric field titled EnvironmentBulkSize. Reference that custom setting in your test cases as you can see in the following code. Remember to set your sandbox's EnvironmentBulkSize option to 200 but set your production value to something like 5. The less the work, the faster the tests:

```java
@isTest
private class ExampleCodeTests {
    TestOptions__c options;

    @testSetup static void setup() {
        options = TestOptions__c.getInstance('default');
    }

    @isTest static void someTest() {
        Account[] accounts = (Account[])TestFactory.createSObjectList(new Account(),
            options.EnvironmentBulkSize);
    }
}
```

Summary
Throughout this chapter, we covered in great detail the why, when, and how of writing unit tests. There's a lot of information and nuance in this chapter. However, I hope you can walk away with not only a grasp of the fundamental pattern for unit testing, but also its three subpatterns of positive, negative, and permissions-based testing. Remember as well that a test method without at least one assert method is a liability. Furthermore, it's likely not going to be useful in the long run. Use asserts liberally. Assert not only the expected response of the unit of code, but also the data you created. Once your tests are written and passing, you'll be able to deploy your code to production. We'll talk about deployment strategies in our next chapter.
Where to buy this book

You can buy Mastering Application Development with Force.com from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.