Apex Design Patterns and Best Practices

Chuck Tomanek  Senior Systems Analyst
The Force.com platform is a new platform, and while the Apex programming language is syntactically similar to Java, the way you use it and the rules you must follow are very different. The best practices and design patterns that apply to Java may not apply to Force.com and the Apex language. That does not mean that none of the same principles apply. Many of the best practices you would use in Java can apply to Apex. General rules around efficient coding like limiting database queries and minimizing loops still apply. In this paper, I'll share what I have found to be the best practices and good design patterns through the multiple development projects I’ve been a part of.

**The Force.com Platform**

The Force.com platform is not like your typical application server environment. It has a very large set of tools that is available to the developer without writing a single line of code. These tools can be a great help in development as they allow you to implement functionality very quickly and easily. This helps make Force.com a very quick development platform.

While the included toolset that Salesforce gives you is a great positive for developers, there is another difference that is not so appealing. Salesforce uses governor limits to ensure one developer’s code does not run wild and end up hurting the performance of the overall system. Force.com is a multi-tenant architecture, which means all users share the same physical hardware and have their own virtual spaces. Since the hardware is shared between everyone, Salesforce needs to assure no one can affect others outside of their virtual space. They do this by enforcing governor limits or restrictions on things like the number of database calls, script statements, etc. that can happen in a transaction. While these limits are important to the stability of the system as a whole, they can be troublesome to developers who are not used to working with such tight restrictions.

**When To Use Apex vs. Native Functionality**

Whenever possible, it is better to use native functionality over Apex. The reasoning is: it’s quicker, not bound by governor limits (for the most part), no unit tests are required, and it’s easier for the end users to make changes if needed.

When I say native functionality, I’m referring to the toolset that Salesforce gives you, and that you don’t need to write code to implement. Things like validation rules, workflow, and formulas can let a developer implement functionality that would require much more time to implement in Apex.

Validation rules allow a developer to put conditional requirements on just about any fields or objects in Salesforce. So, for example, if you wanted to require a field called Email Format Preference to be populated if the Email Opt-In checkbox was selected, but not require it if the Email Opt-In checkbox wasn’t selected, you could do that easily with a validation rule. A developer could write the validation rule in a few minutes, whereas the exact same functionality using Apex would require a trigger to be written, along with unit tests, and would take significantly longer.

When implementing simple business logic, consider using Workflow instead of writing code. Workflow can be used to perform a conditional action on an object. From Workflow, you can send emails, create tasks, and update fields on the same object. Workflow actions can also be set to execute at future times.

**Coding For Bulk Processes**

While developing in Apex, one of the most important things to remember when designing your classes and triggers is always to code for bulk operations. Apex governor limits can be very problematic when your code doesn’t properly handle processing multiple records in the same transaction. Here are some tips for ensuring your code is ready for collections of data in the same transaction.

Always perform your queries and DML statements on collections, instead of individual records. This is because Apex governor limits control how may queries and DML statements are allowed per transaction. Performing these operations on collections minimizes the amount of times you have to make a query or perform a DML operation. Let’s take the following code for example:
trigger UpdateAccountManagerOnChildAccts on Account (after insert, after update) {
    // Define a map to hold parent accounts, and a list to hold child accounts to update:
    Map<Id, Account> parentMap = new Map<Id, Account>();
    List<Account> childAccounts = new List<Account>();
    // Add the parent accounts to the map:
    for (Account a : trigger.new) {
        parentMap.put(a.Id, a);
    }
    // Query for child accounts, set the Account Manager appropriately and add them to the list to update:
    for (Account child : [SELECT Id, Account_Manager__c, ParentId FROM Account WHERE ParentId IN :parentMap.keySet()]) {
        child.Account_Manager__c = parentMap.get(child.ParentId).Account_Manager__c;
        childAccounts.add(child);
    }
    // Update the list of child accounts with a single DML statement:
    update childAccounts;
}

This trigger will work fine as long as only one record is being passed in at a time. If we receive a bulk update to Accounts, we will run into an error because of governor limits that restrict the amount of queries and updates we’re allowed to do in one transaction.

To correct this, we need to assure the update statement is performed on a collection of accounts instead of each account individually. Consider using the following code instead:
While the above code is more complicated, and appears to be more process intensive (due to additional loops and processing), it is actually better in the Apex world. The database interactions have been taken out of loops and only one update statement is made regardless of how many records are processed.

The above code also demonstrates another best practice for Apex, which is never to put SOQL queries inside of loops. Doing this allows the same query to be executed multiple times and creates the possibility of breaking another governor limit. Notice in the top part of the example we have the following nested loops:

```apex
for (Account a : trigger.new){
    for (Account child : [SELECT Id, Account_Manager__c FROM Account WHERE ParentId = :a.Id]){...
```

This causes the query inside the second loop to be executed as many times as accounts that are being processed, which could potentially be quite a few. We correct this problem in the second part of the example by adding the accounts to a map. We then use the map to query all the child accounts we may need to process with one query.

**Batch Apex**

If you need to update a large data set, and don’t need it to happen in real time, consider using batch Apex to perform the logic. To use batch Apex, you write a class that implements the Database Batchable interface. This class will contain a start, an execute, and a finish method. Each pass through the execute method has the governor limits reset. So, for example, if you do not specify a different scope, the governor limits will reset after every 200 records that are being processed. This allows for processing of a lot of records, without hitting governor limits. Here is an example of a batch Apex class from the Salesforce Developers Guide:
global class UpdateAccountFields implements Database.Batchable<sObject>{
global final String Query;
global final String Entity;
global final String Field;
global final String Value;

global UpdateAccountFields(String q, String e, String f, String v){
    Query=q;
    Entity=e;
    Field=f;
    Value=v;
}

global Database.QueryLocator start(Database.BatchableContext BC){
    return Database.getQueryLocator(query);
}

global void execute(Database.BatchableContext BC, List<sObject> scope){
    for(Sobject s : scope){
        s.put(Field,Value);
    }
    update scope;
}

global void finish(Database.BatchableContext BC){}
}

Efficient Coding – Avoiding Multiple Queries

Another good idea is to try to avoid redundant queries to the database. The fewer queries you execute in Apex, the more efficient your code is. A good example of this would be if you needed to retrieve a contact’s first name, last name, account name, and account number. We could do that using these two queries:

Contact c = [SELECT FirstName, LastName, AccountId FROM Contact LIMIT 1];
Account a = [SELECT Name, AccountNumber FROM Account WHERE Id = :c.AccountId LIMIT 1];
The above code works, but a better way to do it would be to use a single query to retrieve all the information needed:

```sql
Contact c = [SELECT FirstName, LastName, Account.Name, Account.AccountNumber FROM Contact LIMIT 1];
```

The above query utilizes relationships to retrieve the same result, with just one call to the database.

**Efficient Coding – Using Helper Classes**

To reduce the amount of code you write and to help keep it maintainable, use helper classes for code that may be referenced in multiple triggers or classes. This idea is nothing new and is generally best practice in most programming languages. I feel this idea is critical when developing in Apex due to the flat file structure you have to work with and the fact that you are limited in how much code you can put in to a Salesforce organization.

Helper classes can also be used to pass information between trigger and classes, and even to control whether or not code should execute. For example, you can use a set of static methods to control whether or not a trigger will execute its logic based on whether or not the update is coming from another trigger. The helper class would contain the following methods:

```java
//global class MyHelper {
    private static boolean isFromMyTrigger = false;

    //Method to receive value of the variable being held.
    global static void isFromThisTrigger(boolean fromTrigger) {
        isFromMyTrigger = fromTrigger;
    }

    //Method to return stored value to requesting app.
    global static boolean returnFromTrigger() {
        return isFromMyTrigger;
    }

    //}
You could then use the above class to tell one trigger the update is coming from another trigger and it should not execute. To do that, put the following code in your trigger that is performing the update, just before the update:

```java
MyHelper.isFromThisTrigger(true);
```

Then, put this check in the second trigger to determine if the update came from the first trigger:

```java
if(!MyHelper.returnFromTrigger()){
    //Your logic that should run if the update is NOT from the first trigger goes here.
}
```

**Synchronous vs. Asynchronous and When to Use Each**

Asynchronous Apex involves code that is put into a queue and run when system resources are available. It allows Salesforce to loosen the governor limits, and allows developers to create more process or database intensive applications. Keep in mind, however, that when you use asynchronous code, the changes made by the code are not immediately visible to the end user. For example, if a user creates a lead that requires an asynchronous method to execute and populate a field on that lead, the value applied by the code will not be visible immediately after the record is saved, so the user may think something didn’t work. Here is a visual example from a Salesforce Dreamforce presentation:
An email is correspondence, while messaging is a conversation.

Setting up a method to be asynchronous is easy. You simply add the @future definition above the method declaration. For example:

```apex
@future
class aClass {  
    public static void myFutureMethod(list<id> idList){  
        //Your logic here....  
    }  
}
```
When the above method is executed, it would be placed into a queue and executed when resources are available. A couple of things to note are that asynchronous methods must always be static and cannot return anything. And, you can only pass primitive data types into an asynchronous method. If you try to pass an object, like Lead or Account, you will receive an error.

Some of the reasons to use asynchronous Apex are when you need to perform actions that require a lot of processing, or database interactions.

Some of the reasons to use asynchronous Apex are when you need to perform actions that require a lot of processing, or database interactions. It can be used to perform updates to a record shortly after it has been inserted. When you use triggers, which are synchronous actions, you need to either perform any logic that will update the record before the insert. The problem is you don’t yet have the id for the new record. You can get the id of a new record on an after trigger, but you won’t be able to change the record at all. If you need the new id, and want to perform an update to it, you will most likely have to use asynchronous code.

There are some reasons to avoid using asynchronous Apex in your applications. One reason is you have a limited amount of asynchronous calls you can make in a 24-hour period. The limit is 200 future method calls per user license, per 24-hour period. If you exceed this limit, you will receive an error every time an attempt to make an asynchronous call is made. This can be detrimental if you have triggers that are commonly used and make asynchronous calls.

Another thing to watch out for when using asynchronous Apex is attempting to update a record in an asynchronous method at the same time it is being updated synchronously. This commonly happens when you are doing mass updates to records that use asynchronous methods to update other records of the same type. The method will attempt to update a record that is currently being updated by the mass data load, or vice versa.

For example, let’s say we have a trigger on accounts that makes a call to an asynchronous method that updates all other accounts with the same parent account. If we were to do a mass update to accounts, we may end up updating an account that calls the future method, which in turn updates all the accounts with the same parent account. If the next batch in the mass update contains another account with the same parent account, it may try to update that account at the same time the asynchronous method from the previous batch tries to update that account, and you will receive an error that says something like “unable to obtain exclusive access to this record.” For this reason, I recommend trying to avoid asynchronous calls that update related records of the same object type. If possible, perform the logic synchronously inside the trigger, or consider using scheduled jobs or batch Apex to perform the logic.

Using Limits Methods To Avoid Governor Limit Errors

As you may have noticed, a lot of these best practices involve ways of avoiding governor limits. Governor limits can be one of the most frustrating parts of developing in Apex. There is nothing more aggravating than completing your application, and releasing it to production only to find out that governor limits are being hit when users are working with the application.
An email is correspondence, while messaging is a conversation.

The above code checks to see that the number of DML rows you have used, plus the number of records in a list you want to update, are less than the maximum amount of rows you are allowed to process in a single transaction. It will update the rows if you are within your limit, or do something else, perhaps use an asynchronous call, if you are going to go over your limit.

**Unit Testing Best Practices – Create, Invoke, Check**

Unit testing will undoubtedly be part of any development project you do on the Force.com platform. Before you can move any Apex code to a production organization, you need to have at least 75% code coverage for your code collectively and at least some coverage for every trigger you have. Often times, writing unit tests is an afterthought for a project. There’s a rush to get the application done and tested, and when it comes time to move the application to production, you realize there are no unit tests and you can’t deploy it. When unit testing is planned and properly implemented, it can save a lot of time and headaches when trying to move any application to production.

One idea for proper unit testing on the Force.com platform is to use the create, invoke, check pattern of test development. Always create your data, execute the code you are testing against the test records you just created, and finally assert that the code behaved as expected. Unit tests in Salesforce do not actually commit data to the database. When the test is over, all the test data is removed, so there’s no harm in creating as many test records as you need. One of the main reasons for creating your tests this way is because you should never assume that the data that exists in your test organization exists in your production organization. Creating your test data in your tests helps ensure that your tests will work when you migrate from one organization to another.

It also gives you flexibility to create various scenarios to ensure your code will work under all sorts of conditions. Always use assert statements to ensure your test yielded the results you expected. It is possible to get very high code coverage without actually testing anything. Line coverage can be gained by having your code execute and not throw an exception. However, you won’t have any idea if it actually worked if you don’t have assert statements to prove the results were expected. Remember, unit testing is critical to writing good, maintainable applications, and should never be skimped on.

**Unit Testing Best Practices – test.startTest and test.stopTest**

Unit test governor limits in Apex are not necessarily representative of governor limits when your code runs outside of the test methods. For this reason, it is always a good idea to wrap your logic that invokes the code you are testing in test.startTest() and test.stopTest() methods. An example of how your test should be constructed could look like this:

```apex
if ((limits.getDMLRows() + accountList.size()) < limits.getLimitDmlStatements()){
    update accountList;
} else{
    //Handle this appropriately, perhaps make an asynchronous call to complete the update.
}
```

The above code checks to see that the number of DML rows you have used, plus the number of records in a list you want to update, are less than the maximum amount of rows you are allowed to process in a single transaction. It will update the rows if you are within your limit, or do something else, perhaps use an asynchronous call, if you are going to go over your limit.
An email is correspondence, while messaging is a conversation.

Constructing your tests this way not only allows the code to run with the governor limits in place, it will be held to in real world execution, but it also is the only way to test asynchronous code. Invoking the asynchronous call after the startTest() method, and before the stopTest() method will allow all of the asynchronous code to complete before running the assertion statements and allows you to properly test your code from asynchronous methods.

**Unit Testing Best Practices – Using Separate Test Classes**

On the Force.com platform, you are limited on the total amount of Apex code you can put into an organization. While the limit is high, and most companies will never use all the space, it is still best to conserve when possible. Unit test code should not count against that limit as it is only executed when deploying or testing applications.

In order not to have your unit test code count against your organization’s total limit, you should put all of your test methods in designated test classes. Test classes are special classes in Apex, and are designated by putting "@isTest" above the class declaration like this:

```apex
@isTest
private class MyTestClass {
    static testMethod void myTestMethod() {
        //Create all your test data...
        test.startTest();
        //Invoke the code you are testing....
        test.stopTest();
        //Perform your assertions to ensure your code behaves as expected....
    }
}
```
Separating your unit tests like this can also make your application more maintainable by putting all the unit tests in a central location. Another helpful tip would be to come up with a standard naming convention for test classes to make it easier for future developers to find them.

**Conclusion**

Apex is still a relatively new programming language, and for that reason it can be difficult to get started programming properly. Many of the principles of other languages hold true for Apex, but some others do not. The ideas presented here will no doubt change over time as Apex matures and new features become available. I believe documentation for design patterns and best practices in Apex is still scarce, so hopefully I have given you some ideas you can use in your applications to make them more stable, maintainable, and functional.
References


