Salesforce Apex Best Practices

Dhananjay Aher

There are certain rules and best practices for writing Apex Code in Salesforce as a backend developer. It's good to follow these guidelines in order to make your system both scalable and manageable. One more important aspect that any organization needs to think about is Test Coverage. When releasing Salesforce Apex Triggers, you can plan proper backup and testing before release of the application.

1.Bulkify Your Code

Bulkification of your code is the process of making your code able to handle multiple records at a time efficiently. This is mainly true for triggers, where up to 200 records could be run through at once, e.g. from a data load. If your code hasn’t been written to take that into account, it can lead to errors being thrown, or worse, unexpected behavior.

2. Avoid DML/SOQL Queries in Loops

- There are times when you may wish to insert/update multiple records at once, or query sets of records based on specific contexts. You may be tempted to either query or run the DML on these records while within your for loop, because what’s the worst that can happen? Well, quite a lot...

- **SOQL and DML are some of the most expensive operations we can perform within Salesforce Apex, and both have strict governor limits associated with them. So, sticking them into a loop is a recipe for disaster, as we can quickly and unknowingly reach these limits, especially when triggers are involved!**
- **For DML statements, we can shift those statements outside of the loop and instead, within our loop, we can add the records we wish to perform those operations on into a list and perform the DML statement on our list instead. For almost all situations, this is the safest and best approach.**

3. Avoid Hard-coded IDs

- Hardcoding IDs might work fine and without incident when they are developed, but as soon as we migrate our code into production, those IDs will no longer reference the correct record. This is especially true for Record Types created in a sandbox and then migrated into production. Consider the opposite situation: if we create a sandbox from production, our hard-coded IDs will no longer point to the correct records.
- If we wish to utilize record types, instead we can reference them via their DeveloperName which will be consistent across environments.

4. Use a Single Trigger per SObject Type

When multiple triggers are defined on a single object, when a record is saved and the triggers are invoked, the order in which the triggers are run cannot be guaranteed – for all intents and purposes it is random. It’s common for individual actions within a trigger to have an order of priority, or it may have a prerequisite on a previous action having been completed (e.g., assigning a parent lookup which is then expected to be populated in the next action). Having a random trigger order also introduces randomness into our code – this randomness makes it harder for us to debug and develop code since we always have an element of randomness and can no longer accurately replicate scenarios.

5. Use SOQL for Loops

When we query a large set of records and assign the results to a variable, a large portion of our heap can be consumed. While this might be fine during test runs where the volume isn’t as large as in a production environment, as the queried dataset grows in volume, more of our heap will be consumed and the results of a query could easily push it over the limit.

Instead of assigning our query results to a variable, we can place our query directly as the iterable variable in our for loop. This causes some behind the scenes changes to how the query is performed, causing the results to be chunked and processed much more efficiently and transparently, preventing us from running into heap limits we may have previously encountered.

6. Test Multiple Scenarios

- Salesforce mandates that we have at least 75% code coverage when we wish to deploy Apex code into production, and while having a high number of lines covered by tests is a good goal to have, it doesn’t tell the whole story when it comes to testing.
- Writing tests only to achieve the code coverage requirement shows one thing: your code has been run, and it doesn’t actually provide any value other than showing that in a very specific scenario (which may or may not ever happen in practice!).

When writing our tests, we should worry about code coverage less, and instead concern ourselves with covering different use cases for our code, ensuring that we're covering the scenarios in which the code is actually being run. We do this by writing multiple test methods, some of which may be testing the same methods and not generating additional covered lines, each of which runs our code under a different scenario.

7. Avoid Nested Loops

- Rather than using nested loops, a good strategy is to abstract your logic into separate methods (which perform the logic themselves). This way, when we’re looking at a block of code from a higher level, it is far easier to understand. Abstracting our code out like this also has other benefits, such as making it easier to test specific aspects of our code.

8. Have a Naming Convention

- Naming conventions tend to be a very important topic in any developer team. The benefits are clear if everyone follows them, as they make it easier for other people in your team to understand what's going on within an org. The specifics of what makes your (or your team's) naming convention are up to you to decide. But following this will reap plenty of benefits when it comes to maintenance and collaboration – there'll be less head scratching while you figure it out, so you can spend more time on the fun stuff.

9. Avoid Business Logic in Triggers

- When writing triggers, if we place our logic directly in the trigger, it becomes very difficult to test and maintain. Instead, what we should be doing is using our trigger to call classes specifically designed to handle our logic – these can then be easily tested, maintained, and reused. We commonly call these classes TriggerHandlers.

10. Use only Required Events in Trigger

- Add only required Events in Trigger. For example, if you want to execute the Trigger after inserting an Account record, only add after insert event

- trigger AccountTrigger on Account (after insert)
  {
  //Trigger Logic.
  }

11.Use Collections and Avoid SOQL or DML in FOR loops

- Always try not to include SOQL and DML in FOR loops. This is applicable not only in Triggers but also in any Apex Code. Make use of Collections like List, Set, and Maps to store the records or other details in Collection to avoid the SOQL or DML in FOR loops. Use Collections to perform DML operations rather than doing it on an individual record. This helps big time to write efficient code. Also, use FOR loops efficiently to minimize the iterations and make use of break statements in FOR loop wherever applicable.

12.Proper use of Future method

- Use the Future method whenever necessary. The Future method runs asynchronously. Hence, consider the fact that it will execute whenever the resources are available. Ideally, we should use Future methods in Triggers only if we need to make API calls from Trigger.

13.Avoid Recursive Trigger

- When we perform a DML operation on a Record in Trigger which executes the same Trigger again and performs the DML operation again (and so on), then it runs for 16 times and fails due to the Governor Limits. To avoid Recursive Trigger, use two static variables with the default value as false for before and after context. Once the Context-specific handler method is executed for the respective context, set it to true. Before calling the Context-specific handler method, check the value of the static variable, and call the method only if the value is false to avoid the Recursive Trigger.

14.Consider Process Builder and Flows before writing Trigger

- Most of the small Triggers can be replaced with Process Builders. Even the complex Triggers can be replaced with the combination of Process Builder and Flow. Hence, before writing a Trigger, see if you can implement the same functionality with Process Builder and Flows.

thank you!

Dhananjay Aher