Apex Best Practices

Apex is a powerful programming language used for custom development in **Salesforce**, but it comes with strict **governor limits** and security rules. Following best practices ensures that your code is **efficient, scalable, maintainable, and secure**.

---

1. Governor Limits & Performance Optimization

Salesforce enforces governor limits to ensure **multi-tenant architecture stability**. Exceeding these limits results in runtime exceptions. Follow these best practices to avoid hitting limits:

1.1 Bulkify Your Code

Bulkification ensures that your Apex code can handle multiple records efficiently in a **single transaction**, rather than processing records one by one.

Bad Practice (SOQL inside a loop - inefficient & will hit limits)

```
for (Account acc : Trigger.new) {
    Account a = [SELECT Id, Name FROM Account WHERE Id = :acc.Id];  // ❌ Inefficient
    a.Name = 'Updated Name';
    update a;
}
```

This will fail if the trigger processes more than 100 records, as **SOQL queries inside a loop** exceed the limit of **100 SOQL queries per transaction**.

Good Practice (Bulkified Query and DML - Efficient & Scalable)

```
List<Account> accList = [SELECT Id, Name FROM Account WHERE Id IN :Trigger.newMap.keySet()]
for (Account acc : accList) {
    acc.Name = 'Updated Name';
}
update accList;  // ✅ Efficient
```

1.2 Avoid SOQL & DML in Loops

SOQL inside a loop **runs multiple queries**, which can hit the **100 SOQL queries per transaction limit**.

DML inside a loop performs multiple database operations, exceeding the **150 DML statements per transaction limit**.

Bad Example

```
for (Contact c : contactList) {
    Contact contact = [SELECT Id, Email FROM Contact WHERE Id = :c.Id];  // ❌ Multiple SOQL
    contact.Email = 'test@example.com';
    update contact;  // ❌ Multiple DML statements
}
```

Good Example (Bulk SOQL & Bulk DML)

```
Map<Id, Contact> contactMap = new Map<Id, Contact>(
    [SELECT Id, Email FROM Contact WHERE Id IN :contactIds]
);
for (Contact c : contactMap.values()) {
    c.Email = 'test@example.com';
}
update contactMap.values();  // ✅ Bulk DML (1 statement)
```

1.3 Use Collections Efficiently

Use **Maps** for quick lookups instead of looping through Lists.

Use **Sets** for unique values and avoiding duplicates.

```
Map<Id, Contact> contactMap = new Map<Id, Contact>();
for (Contact c : [SELECT Id, Email FROM Contact WHERE AccountId IN :accountIds]) {
    contactMap.put(c.Id, c);
}
```

1.4 Limit Data Retrieved in SOQL Queries

- Fetch only necessary fields instead of `SELECT *`.
- Example:

```
List<Account> accList = [SELECT Id, Name FROM Account LIMIT 100];
```

1.5 Use Asynchronous Processing

- Move complex logic to **@future, Queueable, or Batch Apex**.
- Example of @future method:

```
@future
public static void updateContacts(List<Id> contactIds) {
    List<Contact> contacts = [SELECT Id, Email FROM Contact WHERE Id IN :contactIds];
    for (Contact c : contacts) {
        c.Email = 'updated@example.com';
    }
    update contacts;
}
```

---

2. Security Best Practices

Security is critical in **multi-tenant** cloud platforms. Follow these best practices:

2.1 Use "With Sharing" for Security Enforcement

- **Without sharing**, Apex runs in system context, ignoring record-level security.
- Always specify with sharing unless absolutely necessary.

Bad Practice (No Record-Level Security Enforcement)

```
public class AccountService {
    public static List<Account> getAccounts() {
        return [SELECT Id, Name FROM Account];  // ❌ Ignores security
    }
}
```

Good practice(Enforcing Record-Level Security)

```
public with sharing class AccountService {
    public static List<Account> getAccounts() {
        return [SELECT Id, Name FROM Account WHERE Industry = 'Healthcare'];
    }
}
```

2.2 Enforce Field-Level Security (FLS) & Object Permissions

Even with "with sharing", Apex **bypasses field-level security** unless explicitly checked

Bad Practice (Accessing Fields Without Checking FLS)

```
List<Account> accs = [SELECT Name, AnnualRevenue FROM Account];  // ❌ No FLS check
```

Good Practice (Using Security.stripInaccessible)

```
List<Account> accs = Security.stripInaccessible(
    AccessType.READABLE,
    [SELECT Name, AnnualRevenue FROM Account]
).getRecords();  // ✅ Checks FLS before returning data
```

2.3 Avoid Hardcoded IDs

- Never hardcode **Profile IDs, Record Type IDs, or User IDs**.
- Instead, **query them dynamically**:

Id recordTypeId = [SELECT Id FROM RecordType WHERE Name = 'Standard' AND SObjectType = 'Account' LIMIT 1].Id;

---

3. Apex Trigger Best Practices

Triggers execute before/after DML events. Follow best practices to **optimize and structure** trigger execution:

3.1 Use One Trigger Per Object

- Delegate logic to a **Trigger Handler Class**.
- Example:

```
trigger AccountTrigger on Account (before insert, before update) {
    AccountTriggerHandler.handleTrigger(Trigger.new, Trigger.oldMap);
}
```

3.2 Use a Trigger Handler Class

```
public class AccountTriggerHandler {
    public static void handleTrigger(List<Account> newList, Map<Id, Account> oldMap) {
        if (Trigger.isBefore && Trigger.isInsert) {
            validateAccounts(newList);
        }
    }

    private static void validateAccounts(List<Account> accounts) {
        for (Account acc : accounts) {
            if (acc.Name == null) {
                acc.addError('Account Name is required');
            }
        }
    }
}
```

---

4. Exception Handling & Debugging

Exception handling ensures graceful failure recovery.

4.1 Use Try-Catch Blocks

```
try {
    Account acc = new Account(Name = null);
    insert acc;
} catch (DmlException e) {
    System.debug('Error: ' + e.getMessage());
}
```

---

5. Testing Best Practices

Apex test classes ensure **code quality** and prevent future regressions.

5.1 Use @isTest Annotation

```
@isTest
private class AccountTest {
    @isTest
    static void testAccountInsert() {
        Account acc = new Account(Name = 'Test Account');
        insert acc;
        System.assertNotEquals(null, acc.Id);
    }
}
```

5.2 Use Test Setup for Reusable Data

```
@isTest
private class AccountTest {
    @testSetup
    static void setup() {
        Account acc = new Account(Name = 'Test Account');
        insert acc;
    }
}
```

5.3 Use testDataFactory apex class to create test data and use it inside your test class

---

6. Asynchronous Apex Best Practices

For **large data processing**, use:

Type	Use Case
@future	Simple async execution (limited chaining)
Queueable	Chained execution, supports objects
Batch Apex	Large data volumes (50M+ records)
Scheduled Apex	Recurring job execution

Example:

```
public class AsyncExample implements Queueable {
    public void execute(QueueableContext context) {
        System.debug('Processing...');
    }
}
```

7. Integrations & API Best Practices

7.1 Use Named Credentials for Callouts

```
HttpRequest req = new HttpRequest();
req.setEndpoint('callout:MyNamedCredential/service');
```

7.2 Use Platform Events for Scalable Integrations

- Instead of **polling**, use **Platform Events** for real-time updates.

```
EventBus.publish(new MyEvent__e(Message__c = 'Update Occurred'));
```