Salesforce Coding Standards and Best Practices

Table of Contents

[1 Configuration Standard 6](#_Toc385967717)

[1.1 Objects 6](#_Toc385967718)

[Label 6](#_Toc385967719)

[Object Name 6](#_Toc385967720)

[API Name 6](#_Toc385967721)

[Description 6](#_Toc385967722)

[1.2 Fields 6](#_Toc385967723)

[Label 6](#_Toc385967724)

[Name 6](#_Toc385967725)

[API Name 7](#_Toc385967726)

[Description 7](#_Toc385967727)

[Reference Field Name 7](#_Toc385967728)

[1.3 Buttons and Links 7](#_Toc385967729)

[Label 7](#_Toc385967730)

[Name 8](#_Toc385967731)

[Description 8](#_Toc385967732)

[1.4 Page Layouts and Sections 8](#_Toc385967733)

[Page Layout Name 8](#_Toc385967734)

[Section Name 8](#_Toc385967735)

[1.5 Validation / Assignment/ Auto Response Rules/ Sharing Rules 8](#_Toc385967736)

[Name 8](#_Toc385967737)

[Description 9](#_Toc385967738)

[1.6 Workflow Rules and Approval Processes 9](#_Toc385967739)

[Name 9](#_Toc385967740)

[Description 9](#_Toc385967741)

[1.7 Workflow Actions 9](#_Toc385967742)

[Name 9](#_Toc385967743)

[Description 10](#_Toc385967744)

[1.8 Tabs 10](#_Toc385967745)

[Name 10](#_Toc385967746)

[Description 10](#_Toc385967747)

[Tags 10](#_Toc385967748)

[1.9 Profiles 10](#_Toc385967749)

[Name 10](#_Toc385967750)

[Description 10](#_Toc385967751)

[General 10](#_Toc385967752)

[1.10 Roles and Hierarchy 10](#_Toc385967753)

[Name 10](#_Toc385967754)

[1.11 Record Types 11](#_Toc385967755)

[Label 11](#_Toc385967756)

[Name 11](#_Toc385967757)

[Description 11](#_Toc385967758)

[1.12 Deprecation 11](#_Toc385967759)

[2 Apex and VisualForce Naming Convention 11](#_Toc385967760)

[2.1 Code Customization Names 11](#_Toc385967761)

[2.2 Documentation 13](#_Toc385967762)

[3 Best Practices 15](#_Toc385967763)

[3.1 Summary of Best Practices 15](#_Toc385967764)

[3.2 General Guidelines for APEX classes 15](#_Toc385967765)

[3.3 Name Customizations by Project 16](#_Toc385967766)

[3.4 Code for Bulk Processing [Apex Triggers and Classes] 17](#_Toc385967767)

[3.4.1 Handle all records which initiated the code 17](#_Toc385967768)

[3.4.2 Using Collections inside For Loops 18](#_Toc385967769)

[3.4.3 Avoid Queries in For Loop 19](#_Toc385967770)

[3.5 Key Recommendations for Apex Triggers 20](#_Toc385967771)

[3.5.1 No more than one trigger per object 21](#_Toc385967772)

[3.5.2 Turn Trigger into a class 21](#_Toc385967773)

[3.5.3 Disabling trigger behavior 22](#_Toc385967774)

[3.6 Querying Large Data Sets 22](#_Toc385967775)

[3.7 Use Relationship Queries 23](#_Toc385967776)

[3.8 Use Custom Settings 24](#_Toc385967777)

[3.9 Use Custom Labels 26](#_Toc385967778)

[3.10 Profile a Customization's Usage Metrics 27](#_Toc385967779)

[3.11 Use of the Limits Apex Methods to Avoid Hitting Governor Limits 28](#_Toc385967780)

[3.12 Use @future appropriately 29](#_Toc385967781)

[3.13 Use a Value Object for Parameters 30](#_Toc385967782)

[3.14 Use a Value Object or Wrapper as a Cross-Object Container 32](#_Toc385967783)

[3.15 Use a Document or a Static Resource 33](#_Toc385967784)

[3.16 Style Sheet as Static Resource 33](#_Toc385967785)

[4 Exception Handling 34](#_Toc385967786)

[4.1 Basic Exception Handling 34](#_Toc385967787)

[4.2 Throwing Custom Exceptions 35](#_Toc385967788)

[4.3 No Silent Exceptions 35](#_Toc385967789)

[4.4 How to report DML exceptions in controllers 36](#_Toc385967790)

[4.5 When you want to Log your Exceptions 36](#_Toc385967791)

[5 Writing Good Test Methods 39](#_Toc385967792)

[5.1 Best Practices for Writing Test Classes 39](#_Toc385967793)

[5.2 Use Assertions 40](#_Toc385967794)

[5.3 Use the “@isTest” Annotation for Test Classes 42](#_Toc385967795)

[5.4 Use “Test.startTest()” and “Test.stopTest()” on Test Classes 42](#_Toc385967796)

[5.5 Use “Test.isRunningTest()” on Classes with External Callouts 43](#_Toc385967797)

[5.6 Writing Test Methods to Verify Large Datasets 45](#_Toc385967798)

[5.7 Use System.runAs() to execute code as a specific user to test for sharing rules 46](#_Toc385967799)

[6 Visualforce Best Practices 47](#_Toc385967801)

[6.1 Following General Design Guidelines for Application Performance 47](#_Toc385967802)

[6.2 Using Standard Functionality vs. Creating Custom Code 47](#_Toc385967803)

[6.3 Controlling Data Size 47](#_Toc385967804)

[6.4 Lazy Loading 48](#_Toc385967805)

[6.5 Offloading Processing to Asynchronous Tasks 48](#_Toc385967806)

[6.6 Caching Global Data in Custom Settings 49](#_Toc385967807)

[6.7 Writing Efficient Apex and SOQL 49](#_Toc385967808)

[6.8 Writing Efficient Getter Methods 49](#_Toc385967809)

[6.9 Optimizing the View State 49](#_Toc385967810)

[6.10 Optimizing JavaScript 50](#_Toc385967811)

[6.11 Optimizing Image Usage 51](#_Toc385967812)

[6.12 Optimizing Pages for Internet Explorer 51](#_Toc385967813)

[6.13 Visualforce Performance Optimization Case Study 52](#_Toc385967814)

[7 Data Migration 52](#_Toc385967816)

[7.1 Points to be noted for data migration 53](#_Toc385967817)

[8 Appendix 53](#_Toc385967818)

[8.1 Salesforce Limits 53](#_Toc385967819)

[8.1.1 Configuration Limits for Reference 53](#_Toc385967820)

[8.2 Learn SFDC's Object Definition Convention 53](#_Toc385967821)

[8.3 Learn SFDC's URL Convention 54](#_Toc385967822)

[8.4 Learn the Features of the SFDC Migration Tools 55](#_Toc385967823)

[8.5 API Development 56](#_Toc385967824)

# Configuration Standard

Configuration Standards described below mainly cover the naming conventions. The below can be used as a guidance but it’s important to review the standards with the client and ensure we are aligned with the client specific naming conventions if any already exist. Special care needs to be taken when there are multiple Business Units sharing the same salesforce.com instance [like having the shortname of the BU as a prefix to names of the fields, objects etc.,]

## Objects

### Label

The Label of an object is identified by a meaningful word beginning with an uppercase letter. It may not contain any special characters and the total length must be less than 40 characters. The Label should typically represent a precise entity to users. (i.e. Booking, Community User etc.)

### Object Name

Object Name may only contain underscores and alphanumeric characters. It must be unique, begin with an uppercase letter, not end with an underscore, and not contain two consecutive underscores. Typically, SFDC automatically replaces the spaces in Object label to derive Name during object creation/editing. This should be used unless deviation is absolutely necessary.

Custom objects must have unique names within your organization.

### API Name

In the API, the names of custom objects are identified by a suffix of two underscores in the <Name> immediately followed by a lowercase “c” character.

*Example:* A custom object labeled “Customer Support” in the Salesforce user interface is seen as Customer\_Support\_\_c in that organization’s WSDL. API names are also used in Apex code.

### Description

ALL Custom Objects MUST have a detailed description in business terminology.

## Fields

### Label

The Label of a field may contain underscores and alphanumeric characters. It must be unique at the object level, begin with an uppercase letter and words separated by spaces.

### Name

Field Name may only contain underscores and alphanumeric characters. It must be unique at object level, begin with an uppercase letter, not end with an underscore, and not contain two consecutive underscores. The default name given by SFDC is same as the Label with spaces filled with underscores. This should be used unless deviation is absolutely necessary.

### API Name

In the API, the names of custom objects are identified by a suffix of two underscores in the <Name> immediately followed by a lowercase “c” character.

*Example:* A custom object labeled “Customer Support” in the Salesforce user interface is seen as Customer\_Support\_\_c in that organization’s WSDL.

Example:

Label: Client Name

Name: Client\_Name

API Name: Client\_Name\_\_c

### Description

ALL Custom Fields **MUST** have a detailed description in business terminology.

### Reference Field Name

Reference Field contains an ID value that points to a unique record (usually the parent record) on another object. This is analogous to the concept of a foreign key in relational databases. The name of a reference field ends, by convention, with the letters Id.

Example:

“CaseId” OR “OpportunityId”

While referring to parent object fields, typically, the ‘Id’ is replaced by ‘.fieldName’. In the case of custom reference fields, “\_\_c” will be replaced by “\_\_r”. (i.e. Case.Description, Client\_Name\_\_r.Company\_Name\_\_c

## Buttons and Links

Custom buttons are not distinguished from standard buttons in any graphical way. By default, they appear on the right side of Standard Buttons on the page layouts.

### Label

Button/Link Label name

should start with uppercase letter and be short and meaningful without abbreviation and special characters. Button Labels will be defined by the business in the UI Spec.

*Example*: “Update user info” is the label of the Button.

### Name

Names of buttons/links may contain underscores and alphanumeric characters. It must be unique, beginning with an uppercase letter, not ending with an underscore, and not containing two consecutive underscores. Standard practice is to fill the <Label> whitespaces with underscores.

*Example:* “Sample\_Button” is the name of Button.

### Description

ALL Custom Buttons MUST have a detailed description in business terminology.

## Page Layouts and Sections

### Page Layout Name

Page Layout Name should be short and meaningful and start with an uppercase letter. The maximum length for the Name field is 80 characters. It may NOT contain special characters. Often, the name denotes the business function which uses this Eg: “Contact Layout (Marketing)”

### Section Name

Section name should be short and meaningful and start with an uppercase letter. The maximum length for the Name field is 80 characters. It may NOT contain special characters. It should be relevant to the fields contained within.

## Validation / Assignment/ Auto Response Rules/ Sharing Rules

### Name

Name should start with a numeric value for ordering. (Numbers should be “gapped” to allow for later insertion.) After the number, the name should start with an uppercase letter and should be short and meaningful without any abbreviation or special characters. The words in the name should be separated by spaces.

*Note:*

*<API Object Name>\_ can be added to the Name for clarity if needed. This might make the name too long*

*A two lettered abbreviation of the representing the type of component can be appended to the name as well. For example VR for validation rule, WF for Workflow etc.,*

Example:

Sharing Rule: “005 - Role and Subordinates Office 365”

Validation Rule: “010 - Account Validation Rules”

[Or]

<API Object Name>\_VRxx\_<Short Wording>

Example: Account\_VR1\_Date

Assignment Rule: “105 - Hot Account Assignment”

### Description

ALL Custom Rules MUST have a detailed description in business terminology.

## Workflow Rules and Approval Processes

### Name

Name should start with a numeric value for ordering. (Numbers should be “gapped” to allow for later insertion.) After the number, the name should start with an uppercase letter and should be short and meaningful without any abbreviation or special characters. The words in the name should be separated by spaces.

Example:

Workflow: “205 - Account Update Forecast PI”

[Or]

Rule Name: <API Object Name>\_WR205\_<Short Wording>

Approval Processes: “325 - Account Discounting Opportunities”

### Description

ALL Custom Rules MUST have a detailed description in business terminology.

## Workflow Actions

### Name

The Name should start with an uppercase letter and should be short and meaningful without any abbreviation or special characters. The words in the Name should be separated by spaces.

Email Alert Description: <API Object Name>\_EAxx\_<Short Wording>

Field update Name: <API Object Name>\_FUxx\_<Field Name>\_<Field new value>

Outbound message Name: <API Object Name>\_OMxx\_<Short Wording>

### Description

ALL Custom Rules MUST have a detailed description in business terminology.

## Tabs

### Name

This will be same as Object plural Label Name. (i.e. Opportunities)

### Description

ALL Custom Tabs MUST have a detailed description in business terminology.

### Tags

Tags can only contain letters, numbers, spaces, dashes, and underscores, and must contain at least one letter or number.

## Profiles

### Name

The Profile Name should start with an uppercase letter and has to be short and meaningful, without any special characters. The words in the Name should be separated by spaces. It Should represent the business function performed by such users.

Example:

Name: “System Administrator”

### Description

ALL Profiles MUST have a detailed description in business terminology.

### General

SFDC best practices recommend starting a new Profile by cloning an existing profile.

## Roles and Hierarchy

### Name

The Name should start with uppercase letter and be short and meaningful without any special characters. The words in the name should be separated by spaces. Roles should be similar to business roles in a company. (i.e. “Sales Manager”)

Example:

Name: “Central management”

## Record Types

### Label

The Label name should start with an uppercase letter be short and meaningful with no special characters. The words in the name should be separated by spaces.

*Example:* “Manual Lead”

### Name

Name of a record type can contain underscores and alphanumeric characters. It must be unique, beginning with an uppercase letter, not ending with an underscore, and not containing two consecutive underscores. Standard practice is to fill the <Label> gaps with underscores.

*Example:* “Manual\_Lead”

### Description

ALL Record Types MUST have a detailed description in business terminology.

## Deprecation

All deprecated items (objects, fields, workflow rules, etc.) MUST be removed prior to moving to QA. The developer should note in the Deployment Worksheet which items require removal as part of the deployment.

# Apex and VisualForce Naming Convention

## Code Customization Names

Code customizations should follow these guidelines during naming:

|  |  |  |
| --- | --- | --- |
| CUSTOMIZATION  GUIDELINES | | |
| Apex Trigger | 1. Should describe the purpose or process of the trigger.  2. Should not be too long; around 5 words or less would be ideal.  3. Should include the name of the SObject that it is associated with.  4. Preferably starts with an action word. |
| e.g. NewContactTask, ValidateAccountInfo |
| Visualforce Page | 1. Should describe the purpose or process of the page.  2. Should not be too long; around 5 words or less would be ideal.  3. Preferably would not include the word “Page”. |
| e.g. CustomerSearch, NewAccountAndContact |
| Visualforce Component | 1. Should describe the purpose or process of the component.  2. Should not be too long; around 5 words or less would be ideal.  3. Preferably would not include the word “Component”. |
| e.g. ProgressBarDialog, CustomDatePicker |
| Apex Class *(Controller)* | 1. Should be the name of the Visualforce page/component with the suffix of “CC”.  2. Should not be too long; around 5 words or less would be ideal.  3. Preferably would not include the word “Controller”. |
| e.g. CustomerSearchCC, NewAccountAndContactCC |
| Apex Class *(Controller Extension)* | 1. Should be the name of the Visualforce page/component with the suffix of “CX”.  2. Should not be too long; around 5 words or less would be ideal.  3. Preferably would not include the word “Controller”. |
| e.g. CaseDetailsCX, OpportunitySearchCX |
| Apex Class *(Utility / Web Service)* | 1. Should describe the purpose or process of the class.  2. Should not have a suffix.  3. Should not be too long; around 5 words or less would be ideal.  4. Preferably would not include either of the words “Class” or “Object”. |
| e.g. CustomSettingsManager, TransactionConstants, LeadProcessor |
| Apex Class *(Test Coverage)* | 1. Should be the name of the class or trigger with the suffix of “\_Test”.  2. Should not be too long; around 5 words or less (excluding the suffix) would be ideal. |
| e.g. NewContactTask\_Test, CustomSettingsManager\_Test, CustomerSearchCC\_Test |

The names of code customizations should follow the Upper Camel-Casing convention for names. This means that the first character of every word should be capitalized:

|  |  |
| --- | --- |
| NAMES IN UPPER CAMEL-CASE | |
| MyContactDisplay |
| MyContactDisplayCC |

The name should not contain underscores except when naming Test Classes *(see Naming Guidelines above)*.

On the other hand, the Lower Camel-Case should be used when naming the methods of a class. This means that the first character of all the words in the name should be capitalized except for the first word. The first word of the method name should ideally be an action word:

|  |  |
| --- | --- |
| NAMES IN LOWER CAMEL-CASE | |
| getAccountName |
| setOpportunityStage |

|  |
| --- |
| Apex Class : Method Naming |
| public with sharing class CustomerSearchCC {  //constructor  public CustomerSearchCC() {  }  //method  public String getCustomerName() {  }  } |

This, of course, does not apply to a class' constructor as it should always match the class' name exactly.

Variable names should be in Lower Camel-Case except when defining constants. Constant names should be in all upper case and words are separated by an underscore:

|  |
| --- |
| Apex Class : Variables and Constants |
| public with sharing class CustomerSearchCC {  //constant  public static final Integer NOT\_FOUND\_ERROR\_CODE = 404;  //variable  private String customerName {get; set;}  } |

## Documentation

All custom code customizations should have the standard header. Currently there are Salesforce provided tools for code documentation; the syntax used for the ApexDoc plug-in is recommended. More on ApexDoc can be found on the following site:

http://techsahre.blogspot.com/2011/01/apexdoc-salesforce-code-documentation.html

For Apex classes and triggers, Visualforce pages and components, the following block tags or attributes should be used:

|  |  |  |
| --- | --- | --- |
| BLOCK TAG | USAGE | |
| @description | Short description of the purpose of the customization |
| @author | Original author of the customization |
| @date | The code creation date in DD.MMM.YYYY |

The following are examples for the different types of customizations:

|  |
| --- |
| Apex Class |
| /\*\*  \* @description Controller for the CustomerSearch page.  \* @author <Name>  \* @date 01.JUN.2011  \*/  public with sharing class CustomerSearchCC {  } |

|  |
| --- |
| Apex Trigger |
| /\*\*  \* @description Inserts a new Task for new Contacts.  \* @author <Name>  \* @date 16.MAY.2011  \*/  trigger NewContactTask on Contact(after insert) {  } |

|  |
| --- |
| Visualforce Page |
| <!--  /\*\*  \* @description Dynamic customer search.  \* @author <Name>  \* @date 01.JUN.2011  \*/  -->  <apex:page controller="CustomerSearchCC">  </apex:page> |

|  |
| --- |
| Visualforce Component |
| <!--  /\*\*  \* @description Shows a progress bar.  \* @author <Name>  \* @date 21.APR.2011  \*/  -->  <apex:component>  </apex:component> |

For class methods, the following block tags should be used:

|  |  |  |
| --- | --- | --- |
| BLOCK TAG | USAGE | |
| @description | Short description of the purpose of the method |
| @author | Author of the latest modification to the method's code |
| @date | The last modified date in DD.MMM.YYYY |
| @param | The input parameter(s) of the method. Each parameter should have its own entry. These should be listed in the order as defined on the method from left to right. |
| @return | The method's return value. Should be excluded for void methods. |

The comment block should not be indented to match the indention of the method.

|  |
| --- |
| Apex Class |
| /\*\*  \* @description Controller for the CustomerSearch page.  \* @author <Name>  \* @date 01.JUN.2011  \*/  public with sharing class CustomerSearchCC {  /\*\*  \* @description Sets the value of customerName.  \* @author <Name>  \* @date 01.JUN.2011  \* @param firstName The customer's first name.  \* @param lastName The customer's surname.  \* @return Accounts that match the search criteria.  \*/  public List<Account> getRecordDetails(String firstName, String lastName){  }  } |

# Best Practices

## Summary of Best Practices

* Code should be Bulkified
* [Apex code must provide proper exception handling](#_Apex_code_must)
* [Using Collections inside For Loops](#_Using_Collections_inside)
* [Avoid SOQL Queries and DML Operations inside FOR Loops](#_Avoid_SOQL_Queries)
* [Avoiding Multiple Queries](#_Avoiding_Multiple_Queries)
* [Use of Helper Class](#_Use_of_Helper)
* [Use of the Limits Apex Methods to Avoid Hitting Governor Limits](#_Use_of_the)
* [Use Asychronous Apex (@future annotation) for logic that does not need to be executed synchronous.](#_Use_Asychronous_Apex)
* [In case of large data sets, a SOQL query for loop must be used](#_In_case_of)
* [Use test.startTest and test.stopTest while writing test classes](#_Use_test.startTest_and)
* [Use System.runAs() to execute code as a specific user to test for sharing rules](#_Use_System.runAs()_to)
* [Use of Hardcoded IDs should be avoided](#_Use_of_Hardcoded)
* Avoiding View State Error in VisualForce Page.

## General Guidelines for APEX classes

* Declare all variables at the beginning of each class.
* All instance variables must be declared either private or protected.
* A class should have limited public variables except 'final' or 'static final'.
* Try to avoid public variables where possible.
* To convert a primitive type to String use String.valueOf() instead of var + "“.
* Avoid any type of hard coding in the code. Use Custom Labels or static Apex Classes to drive any configuration related values.
* Explicitly initialize local and instance variables to their default values
  + String: default value is null.
  + Integer: default value is 0.
  + Double: 0.0 or else it assigns null by default.
  + Boolean: false.
  + Any other Object: null.
  + Date: null.
* Do not instantiate any object before you actually need it. Specifically, do not instantiate objects in variable declarations outside the try block.
* All constants should have uppercase names, with logical sections of the name separated by underscores.
* Local variables should not have the same name as instance variables. In general, variables in a nested scope should not have the same name as variables in outer scopes.
* Use sets, maps, or lists when returning data from the database. This makes your code more efficient because the code makes fewer trips to the database. Avoid writing a SOQL inside a loop.
* All classes that contain methods or variables defined with the webService keyword must be declared as global. If a method, variable or inner class is declared as global, the outer, top-level class must also be defined as global.

## Name Customizations by Project

Recommendation Level: Medium

SFDC supports the use of a Namespace for disambiguation. Unfortunately, Namespaces are only available for customizations that are part of a Managed Package (a.k.a. registered product) for the AppExchange.

An alternative to Namespaces is prefixing a predefined code to the names of all customizations created for a project. Naming customizations after the project provide the following advantages:

1. **Scope:** The presence of the project prefix immediately provides a viewer with a context regarding the customization without having to delve into the source codes.
2. **Manageability:** SFDC supports the creation of Views for customizations. This allows for easy navigation through customizations. Using the prefix as a filter criterion should narrow down the list to customizations created only for the project.
3. **Safeguard:** This comes into play when there are existing customizations on the SFDC org. Through the use of Views, limiting the display should aid in preventing the accidental modification of customizations outside of the project scope.

Prefixes should be limited up to 5 characters, followed by an underscore, to prevent the over lengthening of names. If used, this should be made as part of the project standards issued to every project member and should be checked for compliance and consistency before handing-off for QA.

|  |  |
| --- | --- |
| SAMPLE PROJECT CODE PREFIX FORMAT | |
| ConAP\_MyContactDisplay |
| ConAP\_MyContactDisplayCC |

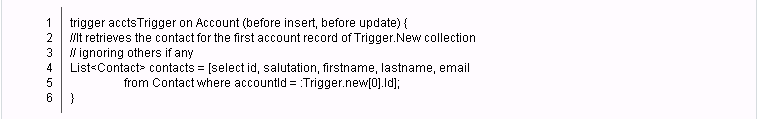
## Code for Bulk Processing [Apex Triggers and Classes]

Recommendation Level: High

**Never assume that a piece of code will only process 1 record at a given time**.

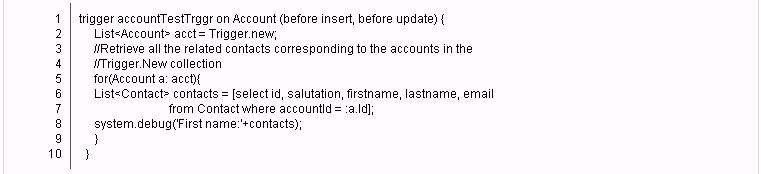
Always process records in batches to efficiently utilize SFDC resources. There is no guarantee that a piece of code will never reach Apex's defined limits, but will help a great deal in preventing it from going near the limit or from frequently reaching those limits. This applies to both Apex classes and triggers **but especially for triggers.**

### Handle all records which initiated the code



The issue is that only one Account record is handled because the code explicitly accesses only the first record in the Trigger.new collection by using the syntax Trigger.new[0]. Instead, the trigger should properly handle the entire collection of Accounts in the Trigger.new collection.

The trigger should handle the entire collection of records which can be achieved as follows



### Using Collections inside For Loops

It is important to use [Apex Collections](http://www.salesforce.com/us/developer/docs/apexcode/index_Left.htm#StartTopic=Content/apex_methods_collections.htm) to efficiently query data and store the data in memory. A combination of using collections and streamlining SOQL queries can substantially help writing efficient Apex code and avoid governor limits.

Here is a sample that uses collections inefficiently

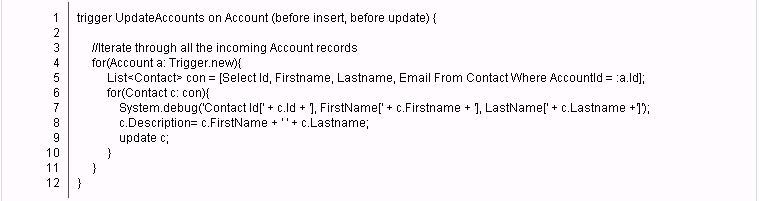
|  |
| --- |
| BAD IMPLEMENTATION |
| trigger accountTrigger on Account (before delete, before insert, before update) {    //This code inefficiently queries the Opportunity object in two seperate queries  List<Opportunity> opptysClosedLost = [select id, name, closedate, stagename  from Opportunity where  accountId IN :Trigger.newMap.keySet() and StageName='Closed - Lost'];    List<Opportunity> opptysClosedWon = [select id, name, closedate, stagename  from Opportunity where  accountId IN :Trigger.newMap.keySet() and StageName='Closed - Won'];    for(Account a : Trigger.new){    //This code inefficiently has two inner FOR loops  //Redundantly processes the List of Opportunity Lost  for(Opportunity o: opptysClosedLost){  if(o.accountid == a.id)  System.debug('Do more logic here...');  }    //Redundantly processes the List of Opportunity Won  for(Opportunity o: opptysClosedWon){  if(o.accountid == a.id)  System.debug('Do more logic here...');  }  }  } | |

The main issue with the previous snippet is the unnecessary querying of the opportunity records in two separate queries. Use the power of the SOQL where clause to query all data needed in a single query. Another issue here is the use of two inner for loops that redundantly loop through the list of opportunity records just trying to find the ones related to a specific account. Look at the following revision:

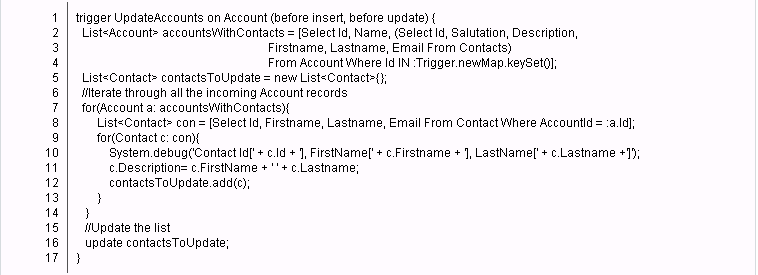
|  |
| --- |
| GOOD IMPLEMENTATION |
| trigger accountTrigger on Account (before delete, before insert, before update) {    //This code efficiently queries all related Closed Lost and  //Closed Won opportunities in a single query.  List<Account> accountWithOpptys = [select id, name, (select id, name, closedate,  stagename from Opportunities where accountId IN :Trigger.newMap.keySet()  and (StageName='Closed - Lost' or StageName = 'Closed - Won'))  from Account where Id IN :Trigger.newMap.keySet()];    //Loop through Accounts only once  for(Account a : accountWithOpptys){    //Loop through related Opportunities only once  for(Opportunity o: a.Opportunities){  if(o.StageName == 'Closed - Won'){  System.debug('Opportunity Closed Won...do some more logic here...');  }else if(o.StageName =='Closed - Lost'){  System.debug('Opportunity Closed Lost...do some more logic here...');  }  }    } | |

### Avoid Queries in For Loop

A common mistake is that queries are placed inside for loop. An individual Apex request gets a maximum of 100 SOQL queries before exceeding that governor limit. When queries are placed inside for loop, a query is executed on each iteration and the governor limit can be easily reached in case of bulk insert/update. Instead, move the SOQL query outside of the for loop and retrieve all the necessary data in a single query.



In the above code snippet, the SOQL query and DML Operations can easily exceed the respective governor limits of 100 and 150, if the trigger is initiated for more than 150 records.

Here is the optimal way to query the contacts in a single query and only perform a single update DML operation****

There may be circumstances where bulk processing cannot be applied but these cases should be few and far between.

***Additional Reading:***

<http://developer.force.com/cookbook/recipe/bulk-processing-records-in-a-trigger>

## Key Recommendations for Apex Triggers

1. Create a single generic trigger for each object rather than a separate trigger for each function. Use the isInsert, isUpdate, isDelete, isBefore and isAfter context variables to determine processing. This allows you to control the order of logic execution and minimize processing of the same records. (e.g. rather than updateOptyTrigger and insertOptyTrigger triggers, create a single trigger for all conditions.)

2. User-defined methods in a trigger cannot be declared as static.

3. Use sets, maps, or lists when returning data from the database. This makes your code more efficient because the code makes fewer trips to the database.

4. You can only set five savepoints in all contexts

5. Execute logic by calling class methods rather than writing the logic within the trigger

6. Triggers must support bulk logic:

* Use the standard query pattern if looping through records
* Use the query-query more pattern if performing DML operations
* Minimize the number of times you loop through all affected records

7. Triggers can be used to prevent DML operations from occurring by calling the addError() method on a record or field. When used on Trigger.new records in insert and update triggers, and on Trigger.old records in delete triggers, the custom error message is displayed in the application interface and logged.

*Note: Bulkfication is a key requirement for trigger which is explained in the above section. Apart from that below are some of the key guidelines for writing efficient triggers*

### No more than one trigger per object

Salesforce.com has documented that **trigger sequence cannot be predefined**. As a best practice create one trigger per object and use comment blocks to separate different logic blocks. By having all logic in one trigger you may also be able to optimize on your SOQL queries.

### Turn Trigger into a class

Trigger code should only be a few lines, around 20 at most. More than that, the code becomes difficult to read and modify. For more complex triggers, create supporting classes instead. This will allow the functionality to be compartmentalized into methods for improved readability and reuse. This also allows for the possible reuse of the class on other places.

|  |  |
| --- | --- |
| SOLUTION : TRIGGER WITH SUPPORTING CLASS | |
| public class AccountHandler {  public static void processNewAccounts(List<Account> acctMap) {  //process new Accounts  }    public static void compareUpdates(Map<ID, Account> newMap, Map<ID, Account> oldMap) {  //compare updates to records  }    public static void checkBeforeDelete(Map<ID, Account> delMap) {  //verify records for deletion  }  } |
| trigger PreprocessAccount on Account(before insert, before update, before delete) {  if (Trigger.isInsert) {  AccountHandler.processNewAccounts(Trigger.new);  } else if (Trigger.isUpdate) {  AccountHandler.compareUpdates(Trigger.newMap, Trigger.oldMap);  } else {  AccountHandler.checkBeforeDelete(Trigger.oldMap);  }  } |
| AVOID : BECOMES UNREADABLE, DIFFICULT TO MAINTAIN | |
| trigger PreprocessAccount on Account(before insert, before update, before delete) {  if (Trigger.isInsert) {    //process new Accounts    } else if (Trigger.isUpdate) {    //compare updates to records    } else {    //verify records for deletion  }  } | |

### Disabling trigger behavior

Use a custom setting to temporarily switch off trigger behavior if needed. This will be very useful wen you want to switch off the triggers for certain profiles or turn of the triggers completely in situations like data migration.

*Note: Triggers can’t be directly disabled in production. It requires a deployment to turn off and on triggers. Controlling behavior using a custom setting will eliminate this problem. It lets you switch off the triggers for certain profiles or turn off the triggers completely in situations like data migration.*

* Create a hierarchical custom setting called DisabledTriggers\_\_c (Setup > Develop > Custom Settings)
* For each behavior, you want to control, create a field using the naming convention [class]\_[method] -- you may have to abbreviate the name a bit now and then due to Force.com limitations on field name lengths. The field must be a check box (boolean) field, which is default unchecked.
* There is at least one data row in the custom setting before deploying any dependent triggers, otherwise you will get a NullpointerException! The first row must be at organization level in the hierarchy.
* In the trigger before calling the business rule, check whether the relevant disabling field of DisabledTriggers\_\_c is true or false as shown in the above example code.

## Querying Large Data Sets

1. Recommendation Level: High

The total number of records that can be returned by SOQL queries in a request is 50,000. If returning a large set of queries causes you to exceed your heap limit, then a SOQL query for loop must be used instead. It can process multiple batches of records through the use of internal calls to query and queryMore. For example, if the results are too large, the `syntax below causes a runtime exception:

|  |
| --- |
| BAD IMPLEMENTATION |
| //A runtime exception is thrown if this query returns enough records to exceed your heap limit.  Account[] accts = [SELECT id FROM account]; | |

Instead, use a SOQL query for loop as in one of the following examples:

|  |
| --- |
| SOLUTION |
| // Use this format for efficiency if you are executing DML statements  // within the for loop. Be careful not to exceed the 150 DML statement limit.  for (List<Account> acct&nbsp;: [SELECT id, name FROM account  WHERE name LIKE 'Acme']) {  // Your code here  update acct;  } | |

## Use Relationship Queries

Recommendation Level: High

When querying on an object and its child objects, use relationship queries or subqueries whenever applicable. This will reduce the number of SOQL calls issued from a customization.

**Sample Requirements:** Retrieve a set of Accounts and the Contacts for each Account.

|  |
| --- |
| SOLUTION 1 : SUBQUERY ON CHILD OBJECT FROM PARENT |
| List<Account> acctList = [SELECT Id, Name, Website, (SELECT Id, Name FROM Contacts)  FROM Account  WHERE (Id IN :acctIDs)]; | |
| SOLUTION 2 : RELATIONSHIP QUERY FROM CHILD OBJECT INCLUDING PARENT FIELDS | |
| List<Contact> contList = [SELECT Id, Name, AccountId, Account.Name, Account.Website  FROM Contact  WHERE (AccountId IN :acctIDs)  ORDER BY AccountId]; | |
| MINIMIZE / AVOID | |
| List<Account> acctList = [SELECT Id, Name, Website  FROM Account  WHERE (Id IN :acctIDs)];  List<Contact> contList = [SELECT Id, Name, AccountId  FROM Contact  WHERE (AccountId IN :acctIDs)  ORDER BY AccountId]; | |

***Additional Reading:***

<http://www.salesforce.com/us/developer/docs/api/Content/sforce_api_calls_soql_relationships.htm>

## Use Custom Settings

Recommendation Level: High

The hard-coding of values into a customization is virtually never viewed as a good practice. Specific values used on code (e.g. for validation, for defaulting, etc.), may change over time or, in the case of SFDC IDs, between Orgs. It would be best if these values are stored in an object that the code can retrieve whenever needed. This makes customizations flexible by being partially configurable; propagating updates to the functionality without having to edit code.

Custom Settings are like custom objects with the added advantage of being readily available for use (meaning without having to query for them). Another advantage is that these objects require minimal configuration when it comes to visibility.

**Sample Requirement 1:** If an Account's Classification Code is either 'AC-099' or 'NM-467', assign an initial value of 25,000,000 as its Estimated Worth. Otherwise assign 0 to Estimated Worth.

|  |  |
| --- | --- |
| CUSTOM SETTING API NAME | AccountEstWorth\_\_c |
| RECORDS | |
| Name | EstWorth\_\_c |
| AC-099 | 25000000 |
| NM-467 | 25000000 |

|  |
| --- |
| SOLUTION |
| Map<String, AccountEstWorth\_\_c> acctWorthRefTable = AccountEstWorth\_\_c.getAll();  for (Account acct: Trigger.new) {  if (acctWorthRefTable.containsKey(acct.ClassificationCode\_\_c)) {  AccountEstWorth\_\_c aewRec = acctWorthRefTable.get(acct.ClassificationCode\_\_c);  acct.EstimatedWorth\_\_c = aewRec.EstWorth\_\_c;  } else {  acct.EstimatedWorth\_\_c = 0;  }  } | |
| AVOID | |
| for (Account acct: Trigger.new) {  if (acct.ClassificationCode\_\_c.equals('AC-099') || acct.ClassificationCode\_\_c.equals('NM-467')) {  acct.EstimatedWorth\_\_c = 25000000;  } else {  acct.EstimatedWorth\_\_c = 0;  }  } | |

**Sample Requirement 2:** A Web Service will be used to verify the authenticity of an Account's information.

|  |  |  |
| --- | --- | --- |
| CUSTOM SETTING API NAME | AccountVerifier\_\_c | |
| RECORDS | | |
| Name | Organization | Endpoint\_\_c |
| 00D90000000YRId | Production | <http://acct.client.com/verify> |
| 00D90000000LWxn | Sandbox | <http://dev.client.com/verify> |
| DEFAULT | Default | <http://dev.client.com/verify> |

|  |
| --- |
| SOLUTION |
| public class WebServiceManager {  public static final String WS\_DEFAULT\_SETTING\_NAME = 'DEFAULT';  public static final String WS\_ERROR\_NO\_SETTINGS = 'No available Web Service Custom Setting for ';  public String getWebServiceEndpoint() {  //get the 15 character version of the current Org's ID  String orgID = UserInfo.getOrganizationId().substring(0, 15);  AccountVerifier\_\_c orgWSSettings = AccountVerifier\_\_c.getInstance(orgID);  if (orgWSSettings == null) {  orgWSSettings = AccountVerifier\_\_c.getInstance(WS\_DEFAULT\_SETTING\_NAME);  }  System.assertNotEquals(orgWSSettings, null, WS\_ERROR\_NO\_SETTINGS + orgID);  return orgWSSettings.Endpoint\_\_c;  }  } | |
| BAD IMPLEMENTATION : SFDC IDs SHOULD NEVER BE HARDCODED; URLs CAN CHANGE | |
| public class WebServiceManager {  public static final String WS\_PRODUCTION\_ENDPOINT = 'http://acct.client.com/verify';  public static final String WS\_SANDBOX\_ENDPOINT = 'http://dev.client.com/verify';  public String getWebServiceEndpoint() {  if (UserInfo.getOrganizationId().equals('00D90000000YRId')) {  return WS\_PRODUCTION\_ENDPOINT;  } else {  return WS\_SANDBOX\_ENDPOINT;  }  }  } | |

Additional Reading:

<https://login.salesforce.com/help/doc/en/cs_about.htm>

## Use Custom Labels

Recommendation Level: High

Custom Labels should always be used for any static text that will be displayed on a Visualforce page or email template. Custom Labels provide the following advantages:

1. **Configurable:** Allows for the update of text on a page without editing the markup.
2. **Language Support:** SFDC automatically applies the corresponding user supplied translation of the text if the current user's language is not the Master Language.

**Sample Requirement:** Support Spanish, French, Italian and German for all pages.

|  |  |
| --- | --- |
| CUSTOM LABEL NAME | WelcomeHeader |
| VALUES | |
| Language | Value |
| English (en\_US) | Welcome! |
| Spanish (es) | ¡Bienvenidos! |
| French (fr) | Bienvenue! |
| German (de) | Willkommen! |
| Italian (it) | Benvenuto! |

|  |
| --- |
| SOLUTION |
| <apex:page standardController="Account">  <!-- displays the appropriate text based on the current user's language setting -->  <h1>{!$Label.WelcomeHeader}</h1>  </apex:page> | |

A VF page can also be forced to use a specific language by specifying the appropriate language code to the page's language attribute:

|  |
| --- |
| SOLUTION |
| <apex:page standardController="Account" language="en\_US">  <!-- always displays in English -->  <h1>{!$Label.WelcomeHeader}</h1>  </apex:page> | |

***Additional Reading:***

<https://login.salesforce.com/help/doc/en/cl_about.htm>

## Profile a Customization's Usage Metrics

Recommendation Level: High

Awareness of the SFDC Governors and Limits is a must for a developer to successfully design and create a viable solution. An elegant solution that fails to come under the known limits is a useless solution. Profiling is a tedious yet necessary part of SFDC development especially for complex solutions.

For triggers, profile your solution for 1 record as well as for the maximum 200 records in a batch. For Apex classes, there should be a defined record limit for the project or for the specific customization.

|  |  |
| --- | --- |
| SOLUTION | |
| trigger Sample1Profiler on Sample1\_\_c (before insert) {  System.debug('<w>>>> TRIGGER.NEW.SIZE: ' + Trigger.new.size());  } |
| PROFILE |
| 16:55:38.147 (147892000)|USER\_DEBUG|[2]|DEBUG|<w>>>> TRIGGER.NEW.SIZE: 200  16:55:38.147 (147938000)|SYSTEM\_METHOD\_EXIT|[2]|System.debug(ANY)  16:55:38.466 (148023000)|CUMULATIVE\_LIMIT\_USAGE  16:55:38.466|LIMIT\_USAGE\_FOR\_NS|(default)|  Number of SOQL queries: 0 out of 100  Number of query rows: 0 out of 50000  Number of SOSL queries: 0 out of 20  Number of DML statements: 0 out of 150  Number of DML rows: 0 out of 10000  Number of script statements: 1 out of 200000  Maximum heap size: 0 out of 3000000  Number of callouts: 0 out of 10  Number of Email Invocations: 0 out of 10  Number of fields describes: 0 out of 100  Number of record type describes: 0 out of 100  Number of child relationships describes: 0 out of 100  Number of picklist describes: 0 out of 100  Number of future calls: 0 out of 10  16:55:38.466 (148023000)|CUMULATIVE\_LIMIT\_USAGE\_END |

Be warned. These limits may change after one of SFDC's upgrade cycles so always check the release notes of upcoming releases. Also, aside from Apex Limits, Visualforce adds another limit which is for the View State. The View State holds the values contained on a form in a VF page.

***Additional Reading:***

<http://www.salesforce.com/us/developer/docs/apexcode/Content/apex_gov_limits.htm>

<http://wiki.developerforce.com/index.php/An_Introduction_to_Visualforce_View_State>

<http://www.salesforce.com/us/developer/docs/pages/Content/pages_tools_edit_toolbar.htm>

## Use of the Limits Apex Methods to Avoid Hitting Governor Limits

Apex has a System class called Limits that lets you output debug messages for each governor limit. There are two versions of every method: the first returns the amount of the resource that has been used in the current context, while the second version contains the word limit and returns the total amount of the resource that is available for that context.

The following example shows how to embed these types of statements in your code and ultimately determine if or when you are about to exceed any governor limits. Using either the System Log or Debug Logs, you can evaluate the output to see how the specific code is performing against the governor limits. Additionally, you can embed logic in the Apex code directly to throw error messages before reaching a governor limit. The code sample below has an IF statement to evaluate if the trigger is about to update too many Opportunities.

Here is an example of how you can use a combination of System.debug statements and the Limits Apex class to generate some very useful output as it relates to governor limits and the overall efficiency of your code.

|  |  |
| --- | --- |
| SOLUTION | |
| trigger accountLimitExample on Account (after delete, after insert, after update) {  System.debug('Total Number of SOQL Queries allowed in this Apex code context: ' + Limits.getLimitQueries());  System.debug('Total Number of records that can be queried in this Apex code context: ' + Limits.getLimitDmlRows());  System.debug('Total Number of DML statements allowed in this Apex code context: ' + Limits.getLimitDmlStatements() );  System.debug('Total Number of CPU usage time (in ms) allowed in this Apex code context: ' + Limits.getLimitCpuTime());    // Query the Opportunity object  List<Opportunity> opptys =  [select id, description, name, accountid, closedate, stagename from Opportunity where accountId IN: Trigger.newMap.keySet()];    System.debug('1. Number of Queries used in this Apex code so far: ' + Limits.getQueries());  System.debug('2. Number of rows queried in this Apex code so far: ' + Limits.getDmlRows());  System.debug('3. Number of DML statements used so far: ' + Limits.getDmlStatements());  System.debug('4. Amount of CPU time (in ms) used so far: ' + Limits.getCpuTime());    //NOTE:Proactively determine if there are too many Opportunities to update and avoid governor limits  if (opptys.size() + Limits.getDMLRows() > Limits.getLimitDMLRows()) {  System.debug('Need to stop processing to avoid hitting a governor limit. Too many related Opportunities to update in this trigger');  System.debug('Trying to update ' + opptys.size() + ' opportunities but governor limits will only allow ' + Limits.getLimitDMLRows());  for (Account a&nbsp;: Trigger.new) {  a.addError('You are attempting to update the addresses of too many accounts at once. Please try again with fewer accounts.');  }  }    else{  System.debug('Continue processing. Not going to hit DML governor limits');  System.debug('Going to update ' + opptys.size() + ' opportunities and governor limits will allow ' + Limits.getLimitDMLRows());  for(Account a&nbsp;: Trigger.new){  System.debug('Number of DML statements used so far: ' + Limits.getDmlStatements());      for(Opportunity o: opptys){  if (o.accountid == a.id)  o.description = 'testing';  }    }  update opptys;  System.debug('Final number of DML statements used so far: ' + Limits.getDmlStatements());  System.debug('Final heap size: ' + Limits.getHeapSize());  }  } |
| DEBUG |
| DEBUG|Total Number of SOQL Queries allowed in this Apex code context: 100  DEBUG|Total Number of records that can be queried in this Apex code context: 10000  DEBUG|Total Number of DML statements allowed in this Apex code context: 150  DEBUG|Total Number of CPU usage time (in ms) allowed in this Apex code context: 10000  DEBUG|1. Number of Queries used in this Apex code so far: 1  DEBUG|2. Number of rows queried in this Apex code so far: 0  DEBUG|3. Number of DML statements used so far: 0  DEBUG|4. Amount of CPU time (in ms) used so far: 9  DEBUG|Continue processing. Not going to hit DML governor limits  DEBUG|Going to update 3 opportunities and governor limits will allow 10000  DEBUG|Number of DML statements used so far: 0  DEBUG|Final number of DML statements used so far: 1  DEBUG|Final heap size: 1819 |

## Use @future appropriately

1. Recommendation Level: High

It is critical to write your Apex code to efficiently handle bulk or many records at a time. This is also true for asynchronous Apex methods (those annotated with the @future keyword). The differences between synchronous and asynchronous Apex can be found Governors in Apex Code#Synchronous\_vs\_Asynchronous\_Apex. Even though Apex written within an asynchronous method gets its own independent set of higher governor limits, it still has governor limits. Additionally, no more than ten @future methods can be invoked within a single Apex transaction.

Here is a list of governor limits specific to the @future annotation:

* No more than 10 method calls per Apex invocation
* No more than 200 method calls per Salesforce license per 24 hours
* The parameters specified must be primitive dataypes, arrays of primitive datatypes, or collections of primitive datatypes.
* Methods with the future annotation cannot take sObjects or objects as arguments.
* Methods with the future annotation cannot be used in Visualforce controllers in either getMethodName or setMethodName methods, nor in the constructor.

It's important to make sure that the asynchronous methods are invoked in an efficient manner and that the code in the methods is efficient. In the following example,

|  |  |
| --- | --- |
| TRIGGER | |
| trigger accountAsyncTrigger on Account (after insert, after update) {  //By passing the @future method a set of Ids, it only needs to be  //invoked once to handle all of the data.  asyncApex.processAccount(Trigger.newMap.keySet());  } |
| global class asyncApex {    @future  public static void processAccount(Set<Id> accountIds) {  List<Contact> contacts = [select id, salutation, firstname, lastname, email from Contact where accountId IN&nbsp;:accountIds];  for(Contact c: contacts){  System.debug('Contact Id[' + c.Id + '], FirstName[' + c.firstname + '], LastName[' + c.lastname +']');  c.Description=c.salutation + ' ' + c.firstName + ' ' + c.lastname;  }  update contacts;  }  } |

## Use a Value Object for Parameters

Recommendation Level: High

A class' method definition may change during development or later upon the implementation of additional requirements. Updating an existing method's signature will mean breaking all customizations that consume the said method. This will mean more modifications to existing functionality as well as regression testing.

A value object is a simple object that's mostly comprised of properties; the main purpose of which is to serve as a container for relevant information. By using a VO instead of enumerating each parameter, it makes the method “future proof” to an extent as well as increasing the method signature's readability. Any additional information that needs to be passed can be implemented as an additional property on the VO without modifying the signature of the method. This also allows existing customizations which do not use the new property to stay intact and still function as expected by ignoring the additional information.

**Sample Requirement (PHASE 1):** Login to a Web Service using a User Name and Password from an Endpoint to get the Session ID

|  |  |
| --- | --- |
| SOLUTION : STARTING STATE | |
| public class WebServiceLoginVO {    public String username {get; set;}  public String password {get; set;}  public String endpointURL {get; set;}  } |
| public class WebServiceHandler {  public String getWSSessionID(WebServiceLoginVO wsVO) {  //login to web service using VO info and return the session ID  }  } |
| AVOID THIS: MAY LEAD TO METHOD SIGNATURE CHANGES IN FUTURE |
| public class WebServiceHandler {  public String getWSSessionID(String userName, String password, String endpointURL) {  //login to web service using VO info and return the session ID  }  } |

**Sample Requirement (PHASE 2):** Login to a Web Service using a User Name and Password from an Endpoint to get the Session ID. The Timeout Interval may vary depending on the Endpoint.

|  |  |
| --- | --- |
| SOLUTION : MODIFIED STATE. NO CHANGE IN METHOD SIGNATURE | |
| public class WebServiceLoginVO {    public String username {get; set;}  public String password {get; set;}  public String endpointURL {get; set;}  public Integer timeoutInterval {get; set;} //additional parameter  } |
| public class WebServiceHandler {  //no change to the method's signature  public String getWSSessionID(WebServiceLoginVO wsVO) {  //login to web service using VO info and return the session ID  }  } |
| AVOID THIS: CHANGE IN METHOD SIGNATURE |
| public class WebServiceHandler {  //change in the method signature  public String getWSSessionID(String userName, String password, String endpointURL, Integer timeoutInterval) {  //login to web service using VO info and return the session ID  }  } |

If an existing SFDC object (custom or standard) can cover the needed information to be passed to the method, this may be used as well.

## Use a Value Object or Wrapper as a Cross-Object Container

Recommendation Level: Medium

When the need to collect information from various sources into one structure arises, the simple List<SObject> may not suffice. Having a List of generic SObjects will not be helpful when the time to access its contents come to play. A Wrapper Class can be made to hide the objects and present to its consumers, a unified way of accessing its information.

|  |  |
| --- | --- |
| SOLUTION : WRAPPER CLASS WITH CUSTOM EXCEPTION | |
| public class CustomRecordException extends Exception{} |
| public class CustomRecord {  public static final String ERROR\_INVALID\_RECORD = 'Invalid Record. Please supply a valid Account or Contact';  private Account acctRec;  private Contact contRec;    //constructor  public CustomRecord(SObject sObj) {  System.assertNotEquals(sObj, null, ERROR\_INVALID\_RECORD);  if (sObj.getSObjectType() == Account.sObjectType) {  this.acctRec = (Account)sObj;  } else if (sObj.getSObjectType() == Contact.sObjectType) {  this.contRec = (Contact)sObj;  } else {  throw new CustomRecordException(ERROR\_INVALID\_RECORD);  }  }    //accessors  public ID getID() {  if (this.acctRec != null) {  return this.acctRec.Id;  } else {  return this.contRec.Id;  }  }    public String getName() {  if (this.acctRec != null) {  return this.acctRec.Name;  } else {  return (this.contRec.LastName + ', ' + this.contRec.FirstName);  }  }    public String getStreetAddress() {  if (this.acctRec != null) {  return this.acctRec.BillingStreet;  } else {  return this.contRec.MailingStreet;  }  }  } |

If the code has to support more than a handful of objects, the resulting Wrapper Class' code might become too long and complicated. Another way is to use a Value Object instead (*see previous section on Value Objects*). Though with VOs, you lose the SObject's methods and other information not captured in the VO's properties. The VO does make it up with simplicity and flexibility since it can also support non-SObject sources of information.

## Use a Document or a Static Resource

Recommendation Level: Low

For Visualforce pages, the use of Static resources for images, style sheets and other libraries is highly encouraged. Static Resources are easily accessible via VF syntax, reusable and are updateable resources.

On Visualforce Email Templates however, be careful on using Static Resources as they may not behave properly. This may be due to SRs needing the context of SFDC to interpret the VF syntax for SRs properly.

For example: the company logo used on an Email Letterhead. The image of the logo should be created as a Document record rather than a Static Resource as Documents do not require special syntax just the specific URL to the image. This allows the logo to be displayed properly on the email.

## Style Sheet as Static Resource

Recommendation Level: High

Whenever custom style sheets are to be used on a Visualforce page, these should be placed in a CSS file and then uploaded as a Static Resource. SFDC requires this especially for pages rendered as PDF. This also makes the style sheet reusable and has an effect of a cleaner VF markup. The use of inline styling should be discouraged if it can be avoided.

***Additional Reading:***

<http://www.salesforce.com/us/developer/docs/pages/Content/pages_resources.htm>

<http://www.salesforce.com/us/developer/docs/pages/Content/pages_styling_custom.htm>

<http://wiki.developerforce.com/index.php/Delivering_Static_Resources_with_Visualforce>

# Exception Handling

Recommendation Level: High

Apex uses exceptions to note errors and other events that disrupt the normal flow of code execution. throw statements can be used to generate exceptions, while try, catch, and finally can be used to gracefully recover from an exception. Hence, proper error handling helps in dealing any type of situation and thus generating an efficient, error-free code.

Here are the most common examples of exception conditions

• Your code expects a value from something that is currently null

• An insert or update statement fails to pass a custom validation rule you have set

• Assigning a query that returns no records or more than one record to a singleton sObject variable

• Accessing a list index that is out of bounds

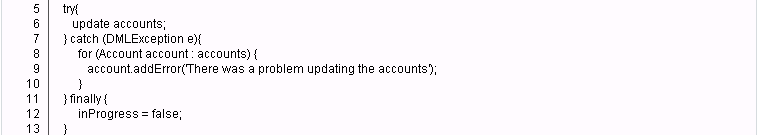
In all these instances we're trying something that the language deems impossible, and an exception is thrown.

## Basic Exception Handling

Use try-catch block when you know the nature of exception and want to handle it accordingly. While handling such exceptions you must display error messages understandable to the end user.

Examples: Use try-catch block for all API calls and based on the severity you interrupt the processing by display the meaningful message on UI.

In remaining cases, where you are not sure of the Exception and its cause, let the exception to be caught by Salesforce. It sends an email message with the details of the Error occurred like method name, line number and type of error. It also displays Error message on the UI, but that may not be meaningful for the end user. The email message helps track the error while debugging.

****

## Throwing Custom Exceptions

You may decide that you want to interrupt your program flow even when a system exception does not occur. In that case you can throw your own custom exception. It's very easy to do - simply create a new exception class by extending the Exception class. Here's an example of a custom exception, and some code that throws the exception:

|  |  |
| --- | --- |
| SOLUTION | |
| public class MyException extends Exception{}  //try, throw and catch  try {  Integer i;  //Your code here  if ( i < 5 ) throw new MyException();  } catch ( MyException e ) {  //Your MyException handling code here  } |

Custom exceptions give you the flexibility to throw exceptions under whatever conditions make sense to you.

## No Silent Exceptions

Error handling using a try-catch block is considered a good practice. However, catching an exception but not doing anything about it is not. It places the user into a false sense of security that everything is proceeding smoothly when in fact it's far from it. The user should be informed of the error if the code was part of a trigger or VF controller. If the exception occurred during a backend process like a batch job or an asynchronous process, these should be logged so admins can later bring them up for review. Optionally, an email can be sent to an identified admin to notify him of the error.

|  |  |
| --- | --- |
| SOLUTION | |
| try {  Account acct = [SELECT Id, Name FROM Account WHERE Id = :recID];  } catch (QueryException qe) {  System.debug('<w>>>>> QE MESSAGE: ' + qe.getMessage());  ApexPage.Messages msg = new ApexPages.Message(ApexPages.Severity.ERROR, System.Label.NoMatchFound);  ApexPages.addMessage(msg);  } |
| BAD IMPLEMENTATION : EXCEPTION IS STILL NOT HANDLED |
| try {  Account acct = [SELECT Id, Name FROM Account WHERE Id = :recID];  } catch (QueryException qe) {  //silent exception  } |

Additional Reading:

<http://wiki.developerforce.com/index.php/An_Introduction_to_Exception_Handling>

## How to report DML exceptions in controllers

|  |
| --- |
| INSTEAD OF THIS |
| try {  upsert myList;  } catch (System.Dmlexception e) {  Apexpages.addMessage(new Apexpages.Message(ApexPages.Severity.ERROR, e.getmessage()));  // Other exception handling...  } |
| DO THIS |
| try {  upsert myList;  } catch (System.Dmlexception e) {  for (Integer i = 0; i < e.getNumDml(); i++) {  Apexpages.addMessage(new Apexpages.Message(ApexPages.Severity.ERROR, e.getDmlMessage(i)));  }  // Other exception handling...  } |

## When you want to Log your Exceptions

Some clients require robust handling of errors to the point where custom objects are made specifically to log all errors. To use custom objects to log and track errors or exceptions, a developer can follow the next couple of steps:

* 1. Create the ExceptionLog custom object with the following custom fields.

|  |  |
| --- | --- |
| Exception Log custom object | |
| Custom Field 1 | The type of exception. Pass in the text string "Trigger" where the exception originated in a trigger and "Class" for a class. |
| Custom Field 2 | The object name. For classes this should be the class name. For triggers this should be the object name |
| Custom Field 3 | Function name. For Classes this should be the function name. For triggers this is the trigger name |
| Custom Field 4 | The exception object itself |

* 1. Create a new public Apex class called "ExceptionLog". Create one static function on this class and name it "add". This function will accept the four input parameters that will match the fields on the custom object created.

|  |  |
| --- | --- |
| SOLUTION | |
| Public ExceptionLog{  .  .  .  static add(String param1, String param2, String param3, String param4){  // Write the code here to create a new instance of the custom sObject ExceptionLog  ExceptionLog\_\_c newExnLog = new ExceptionLog\_\_c();    // Set the values of custom fields equals to the corresponding values from the input parameters  newExnLog.customField1\_\_c = param1;  .  .  .    //Insert the newly created ExceptionLog\_\_c instance  Insert newExnLog;  .  .  .  }  } |

* 1. Invoke or call the “add” function on all triggers or apex classes whenever there is code that implements error handling (try-catch block). Provide the necessary values of the input parameters.

|  |  |
| --- | --- |
| SOLUTION | |
| try{    //Your code here  } catch (ListException le) {    //Optional catch of a specific exception type  //Specific exception handling code here  ExceptionLog.add('Trigger', 'This object name', 'This trigger name', le);  } catch (Exception e) {  //Generic exception handling code here  ExceptionLog.add('Trigger', 'This object name', 'This trigger name', e);  } |

The above sample code implementation was done on a Trigger where:

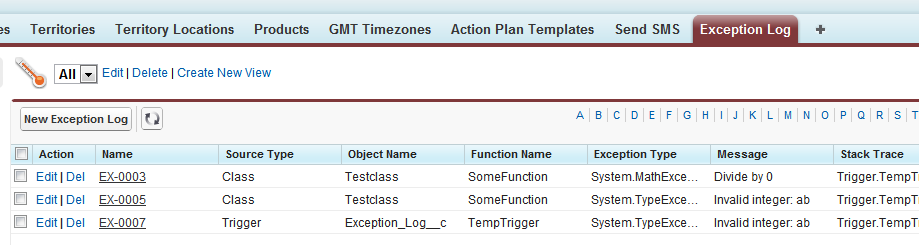
* + - "Trigger" is the type of exception
    - "This object name" is the name of the object that the trigger is based on
    - "This trigger name" is the name of the trigger.
    - le and e are the actual object exceptions

The next example shows how to implement the error logging on Apex classes:

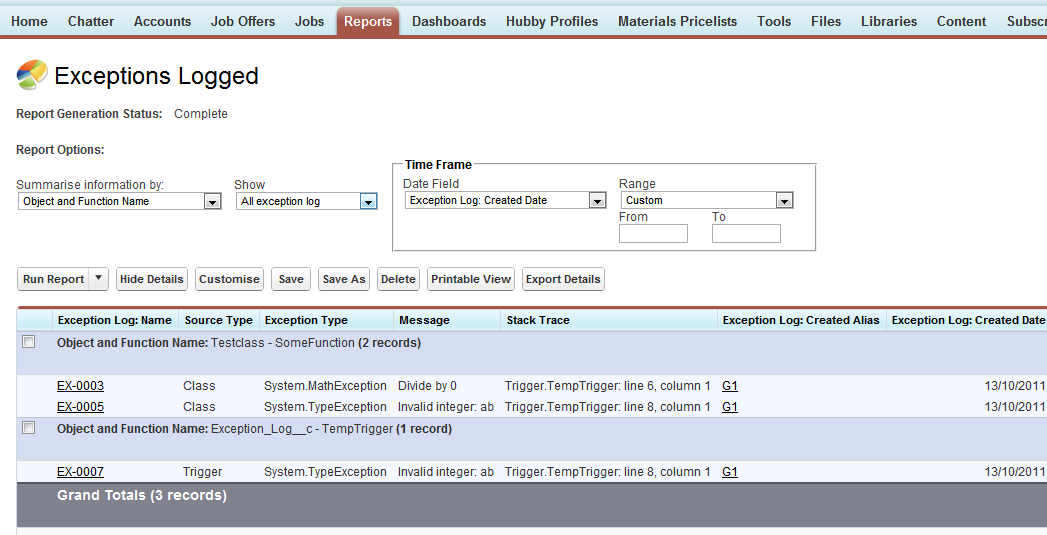
|  |  |
| --- | --- |
| SOLUTION | |
| try{    //Your code here  } catch (Exception e) {  //Generic exception handling code here  ExceptionLog.add(Class, 'This class name', 'This function name', e);  } |

where:

* "This class name" is the name of the class
* "This function name" is the name of the function.
  1. The custom object can then be displayed as a tab on the application and give users (like System administrators) a way to view the logged error/exception records.



* 1. The custom object can also be used on Reports.



Some of the benefits on logging and tracking exceptions by using custom object are as follows :

* + Exceptions aren't lost
  + Historical exceptions can be viewed anytime
  + Historical exceptions can be reported on and find things like the most common problem areas
  + Viewing of Exceptions can be controlled or be shown to specific users(like system admins instead of to the users )

***Additional References:***

<http://wiki.developerforce.com/index.php/An_Introduction_to_Exception_Handling>

<http://www.salesforce.com/us/developer/docs/apexcode/Content/langCon_apex_exception_statements.htm>

# Writing Good Test Methods

To facilitate the development of robust, error-free code, Apex supports the creation and execution of unit tests. Unit tests are class methods that verify whether a particular piece of code is working properly. Unit test methods take no arguments, commit no data to the database, send no emails, and are flagged with the testMethod keyword in the method

*Note: 75% coverage is mandated by salesforce but strive to achieve a higher coverage %. Ensure the test methods truly validate the functionality by having appropriate test data and assertions.*

## Best Practices for Writing Test Classes

* All test methods should reside in a separate class from the class in which the method being tested resides.
* These classes should be appended with the word Test followed by the name of the class being tested, e.g. TestOpportunityServices.
* These classes should all use the @isTest annotation.
* Each method in the production class should have, at a minimum, one corresponding test method in its test class and should be appended by “test”
* There should be a minimum of “Null Pointer Exception test” as part of negative testing for each method, specially the methods that accept parameters.
* A method without an assert statement is not considered a test method. Large number of relevant asserts statement increases confidence in the correct behaviour of business logic.
* There should be a comment with each assert statement explaining what is being tested and what the expected output is
* Only use isTest(SeeAllData = true) on class methods in exceptional cases where there are sObjects that doesn't allow DML operation e.g. PriceBook creation
* No hard coded ids of any sObject in any test method.
* If a Constant needs to be asserted, it’s a best practice to reference that from the Constant class or from Custom Labels or Custom Settings. Using hard coded string in unit tests( or any class for that matter) will trigger failures when things like Picklist values change
* All test data creation should be done from a Utility class. This allows for a streamlined creation of test objects that adhere to all the validation rules.
* Any business logic that needs to be tested should be enveloped within a Test.runAs(user) statement so profile restrictions can be tested. . Using any admin profiles should be avoided.
* All private methods should also have its corresponding unit test method. In the production code, add a public method for each private method and prepend it by “exposeForUnitTest\_”.
* Creating multiple test method for testing that same production code method should be avoided. We want to ensure that our unit test methods are properly testing the logic but the same time the efficiency of the unit test method should not be ignored. All the unit test methods run with every deployment so the cumulative run time should be as small as possible.
* Any asynchronous method testing should include Test.startTest and Test.stopTest. Test.stopTest forces the asynchronous methods to run so the results could be asserted.
* Any exceptions that are caught in the production methods should be tested by feeding the test data that throws exception. Exception Type and error message should be asserted.

**Additional Reading:**

<https://www.salesforce.com/us/developer/docs/apexcode/Content/apex_testing_best_practices.htm>

## Use Assertions

Recommendation Level: High

Assertions are simple tests that can be carried out to ensure that a code set is working with appropriate data. This can range from simple tests for NULL to the verification of an end result.

|  |  |
| --- | --- |
| SOLUTION | |
| public class MyContactUtility {  public static final String ERROR\_INVALID\_LASTNAME = 'ERROR: Please specify a Last Name with at least 2 characters';  public static final String ERROR\_INVALID\_FIRSTNAME = 'ERROR: Please specify a First Name with at least 3 characters';  public static String getFullName(String lastName, String firstName) {  System.assertNotEquals(lastName, null, ERROR\_INVALID\_LASTNAME);  System.assertNotEquals(firstName, null, ERROR\_INVALID\_FIRSTNAME);    System.assert((lastName.length() > 1), ERROR\_INVALID\_LASTNAME);  System.assert((firstName.length() > 2), ERROR\_INVALID\_FIRSTNAME);    return firstName + ' ' + lastName;  }  } |

System.assertEquals() and System.assertNotEquals(), by default, outputs the values of the first and second parameters when it fails. Sort of like an auto-System.debug():

|  |  |
| --- | --- |
| SOLUTION | |
| Integer x = 1;  Integer y = 2;  System.assertEquals(x, y); |
| EXECUTION RESULTS |
| 23:59:19.042 (42374000)|EXECUTION\_STARTED  23:59:19.042 (42440000)|CODE\_UNIT\_STARTED|[EXTERNAL]|execute\_anonymous\_apex  23:59:19.043 (43460000)|SYSTEM\_METHOD\_ENTRY|[4]|System.assertEquals(ANY, ANY)  23:59:19.043 (43628000)|EXCEPTION\_THROWN|[4]|System.AssertException: Assertion Failed: Expected: 1, Actual: 2  23:59:19.043 (43781000)|SYSTEM\_METHOD\_EXIT|[4]|System.assertEquals(ANY, ANY)  23:59:19.043 (43927000)|FATAL\_ERROR|System.AssertException: Assertion Failed: Expected: 1, Actual: 2 |

Assertions would also be ideal for test classes to check the results of the code being covered:

|  |  |
| --- | --- |
| SOLUTION : TRIGGER WITH SUPPORTING CLASS AND TEST CLASS | |
| public class ContactCommHandler {  public static final String CONTACT\_COMM\_EMAIL = 'By Email Only';  public static final String CONTACT\_COMM\_PHONE = 'By Phone Only';  } |
| trigger SetContactCommPref on Contact(before insert) {  for (Contact cont: Trigger.new) {  if (cont.DoNotCall) {  cont.CommunicationMode\_\_c = ContactCommHandler.CONTACT\_COMM\_EMAIL;  } else {  cont.CommunicationMode\_\_c = ContactCommHandler.CONTACT\_COMM\_PHONE;  }  }  } |
| @isTest  private class SetContactCommPref\_Test {  static testMethod void testEmailCommType() {  Contact cont = new Contact(LastName = 'Doe', FirstName = 'John', DoNotCall = TRUE);    Test.startTest();  insert cont;  Test.stopTest();  cont = [SELECT Id, CommunicationMode\_\_c FROM Contact WHERE (Id = :cont.Id)];    System.assertEquals(ContactCommHandler.CONTACT\_COMM\_EMAIL, cont.CommunicationMode\_\_c);  }  static testMethod void testPhoneCommType() {  Contact cont = new Contact(LastName = 'Doe', FirstName = 'John', DoNotCall = FALSE);    Test.startTest();  insert cont;  Test.stopTest();  cont = [SELECT Id, CommunicationMode\_\_c FROM Contact WHERE (Id = :cont.Id)];    System.assertEquals(ContactCommHandler.CONTACT\_COMM\_PHONE, cont.CommunicationMode\_\_c);  }    } |

***Additional Reading:***

<http://www.salesforce.com/us/developer/docs/apexcode/Content/apex_methods_system_system.htm>

<http://wiki.developerforce.com/index.php/An_Introduction_to_Apex_Code_Test_Methods>

<http://wiki.developerforce.com/index.php/How_to_Write_Good_Unit_Tests>

## Use the “@isTest” Annotation for Test Classes

Recommendation Level: High

When writing test coverages, always create a separate class for the test coverage and use the @isTest annotation. The annotation marks the class as a test class which tells SFDC to not count the code as part of the total limit for the customizations. Remember to always use the private modifier for the class.

***Additional Reading:***

<http://www.salesforce.com/us/developer/docs/apexcode/Content/apex_classes_annotation_isTest.htm>

## Use “Test.startTest()” and “Test.stopTest()” on Test Classes

Recommendation Level: High

When writing test coverage, it is highly recommended that the test class setup its own data for the test. This prevents the test class from failing due to dependencies on existing data. Creating data through code consumes part of the allocations for the class and counts against its usage metrics. To discount the test data from the class' usage metrics, the Test class' startTest() and stopTest() methods are used:

|  |
| --- |
| SOLUTION : CHECK THE CLASSIFICATION OF THE RECORD USING ITS ID |
| @isTest  private class MyCustomDisplayCC\_Test {  static testMethod void testDisplay() {  Account acct = new Account(Name = 'Acme Corporation, Inc.', Website='www.acmekabo.om');  insert acct;    Contact cont = new Contact(LastName = 'Coyote', FirstName = 'Wile', Email = 'wile@acmekabo.om');  insert cont;    PageReference mcd = Page.MyCustomDisplay;  Test.setCurrentPage(mcd);  ApexPages.currentPage().getParameters().put('id', acct.Id);    Test.startTest();  MyCustomDisplayCC mcdCC = new MyCustomDisplayCC();    //reload the display  mcdCC.refreshDisplay();  Test.stopTest();  }  } |

**NOTE:** Records created under a testMethod are rolled back after the method's execution.

***Additional Reading:***

<http://www.salesforce.com/us/developer/docs/apexcode/Content/apex_methods_system_test.htm>

<http://wiki.developerforce.com/index.php/An_Introduction_to_Apex_Code_Test_Methods>

<http://wiki.developerforce.com/index.php/How_to_Write_Good_Unit_Tests>

## Use “Test.isRunningTest()” on Classes with External Callouts

Recommendation Level: Medium

A callout to an external service during execution of a test coverage is not allowed. If the class being covered consumes a web service and processes its results then the test coverage will fail. To address this, use the Test class' isRunningTest() method on the class being covered to simulate the web service call:

|  |
| --- |
| SOLUTION |
| public class ExternalWSHandler {  public class ExternalWSInfoVO {  public String param1 {get; set;}  public String param2 {get; set;}  public String param3 {get; set;}  }  public class WSResultsContainer {  public String results;  }    public HTTPRequest buildWSRequest(ExternalWSHandler.ExternalWSInfoVO ewsVO) {  //create the HTTPRequest based on the information in the value object  }    public ExternalWSHandler.WSResultsContainer invokeExternalWS(HTTPRequest req) {  if (Test.isRunningTest()) {  //return a mock-up of the WS call results contained in an instance of WSResultsContainer  } else {  //callout to external web service using the HTTPRequest instance  //return the results of the WS call wrapped in an instance of WSResultsContainer  }  }    public String getResultsInfo(ExternalWSHandler.WSResultsContainer wsrc) {  //process the WS results and return the necessary value  }  } |
| @isTest  private class ExternalWSHandler\_Test {  static testMethod void testDisplay() {    String wsRes;  Test.startTest();  ExternalWSHandler ewsh = new ExternalWSHandler();  ExternalWSHandler.ExternalWSInfoVO ewsVO = new ExternalWSHandler.ExternalWSInfoVO();  ewsVO.param1 = 'value1';  ewsVO.param2 = 'value2';  ewsVO.param3 = 'value3';    HTTPRequest req = ewsh.buildWSRequest(ewsVO);    ExternalWSHandler.WSResultsContainer wsrc = ewsh.invokeExternalWS(req);    wsRes = ewsh.getResultsInfo(wsrc);  Test.stopTest();    System.assertEquals('Expected Value', wsRes);  }  } |

***Additional Reading:***

<http://www.salesforce.com/us/developer/docs/apexcode/Content/apex_methods_system_test.htm>

<http://wiki.developerforce.com/index.php/An_Introduction_to_Apex_Code_Test_Methods>

<http://wiki.developerforce.com/index.php/How_to_Write_Good_Unit_Tests>

## Writing Test Methods to Verify Large Datasets

Since Apex code executes in bulk, it is essential to have test scenarios to verify the Apex being tested is designed to handle large datasets and not just single records. To elaborate, an Apex trigger may be invoked either by a data operation from the user interface or by a data operation from the Force.com SOAP API. The API may send multiple records per batch, leading to the trigger being invoked with several records. Therefore, it is important to have test methods which verify that all Apex code is properly designed to handle larger datasets and does not exceed governor limits.

The example below demonstrates a poorly written trigger which does not handle bulk properly and, therefore, hits a governor limit. Later, the trigger is revised to properly handle bulk datasets.

This is the poorly written contact trigger. For each contact, the trigger performs a SOQL query to retrieve the related account. The invalid element in this trigger is that the SOQL query is within the FOR loop and, therefore, will throw a governor limit exception if more than 100 contacts are inserted/updated.

|  |
| --- |
| BAD TRIGGER |
| trigger contactTest on Contact (before insert, before update) {    for(Contact ct: Trigger.new){    Account acct = [select id, name from Account where Id=:ct.AccountId];  if(acct.BillingState=='CA'){  System.debug('found a contact related to an account in california...');  ct.email = 'test\_email@testing.com';  //Apply more logic here....  }  }    } |

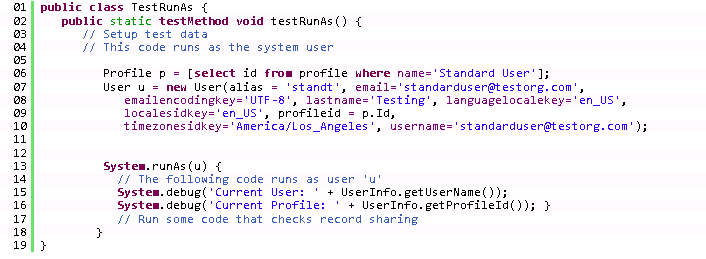
|  |
| --- |
| Test method that tests if this trigger properly handles volume datasets: |
| public class sampleTestMethodCls {  static testMethod void testAccountTrigger(){    //First, prepare 200 contacts for the test data  Account acct = new Account(name='test account');  insert acct;    Contact[] contactsToCreate = new Contact[]{};  for(Integer x=0; x<200;x++){  Contact ct = new Contact(AccountId=acct.Id,lastname='test');  contactsToCreate.add(ct);  }    //Now insert data causing an contact trigger to fire.  Test.startTest();  insert contactsToCreate;  Test.stopTest();  }  }  } |

This test method creates an array of 200 contacts and inserts them. The insert will, in turn, cause the trigger to fire. When this test method is executed, a System.Exception will be thrown when it hits a governor limit. Since the trigger shown above executes a SOQL query for each contact in the batch, this test method throws the exception 'Too many SOQL queries: 101'. A trigger may only execute at most 100 queries.

Note the use of Test.startTest and Test.stopTest. When executing tests, code called before Test.startTest and after Test.stopTest receives a separate set of governor limits than the code called between Test.startTest and Test.stopTest. This allows any data which needs to be setup to do so without affecting the governor limits available to the actual code being tested.

## Use System.runAs() to execute code as a specific user to test for sharing rules

Generally, all Apex code runs in system mode, and the permissions and record sharing of the current user are not taken into account. The system method, System.runAs(), lets you write test methods that change user contexts to either an existing user or a new user. RunAs() functionality tests and verify proper data sharing and data access. But runAs() does not validate CRUD or Field Level Security permissions. An example is illustrated as below:



# Visualforce Best Practices

## Following General Design Guidelines for Application Performance

When designing Visualforce pages, follow these general guidelines to avoid performance impacts.

* Design pages around specific tasks, with a sensible workflow and navigation between tasks.
* Don’t overload pages with functionality and data. Visualforce pages with unbounded data or a large number of components, rows, and fields have poor usability and performance, and they risk hitting governor limits for view state, heap size, record limits, and total page size.
* Push back on requests to include nonessential functionality.
* Build prototypes to validate concerns.

## Using Standard Functionality vs. Creating Custom Code

The programmatic features of the Force.com platform make it easy to customize functionality, but always use standard objects and declarative features whenever possible. Standard objects and declarative features—such as approval processes, visual flows, and workflows—are highly optimized already and don’t count against most governor limits. They typically simplify data models and reduce the number of Visualforce pages necessary for business processes.

## Controlling Data Size

Visualforce pages can’t exceed the 15 MB standard response limit, but even smaller page sizes affect load time. To minimize load times, use the following techniques to further limit the amount of data each page displays.

**Filters**

Use filters to limit the data that SOQL calls in and Apex controllers return.

For example, use AND statements in your WHERE clause or remove null results.

**Keywords**

When creating Apex controllers, use the with sharing keyword to retrieve only the records the user can access.

**StandardSetController Built-In Pagination**

Use the built-in pagination functionality in list controllers to prevent list views from displaying unbounded data. Unbounded data might cause longer load times, hit governor limits, and become unusable as the data set grows. By default, a list controller returns 20 records on the page, but developers often configure list views to display up to 100 records at a time.

**SOQL OFFSET Pagination**

Use the SOQL OFFSET clause to write logic that paginates to a specific subset of results within SOQL.

In addition, avoid using data grids, which display many records with editable fields. Data grids frequently expand to thousands of input components on a page and exceed the maximum view state size, resulting in a Visualforce component tree that’s slow to process.

If your Visualforce page needs a data grid:

* Use pagination and filters.
* Where possible, make data read-only to reduce the view state size.
* Only display essential data for a given record, and provide a link for an Ajax-based details box or separate page to view the rest.

## Lazy Loading

To reduce or delay expensive calculations or data loading, use lazy loading, a technique in which a page loads its essential features first and delays the rest—either briefly or until the user performs an action that requires the information. This technique gives users faster access to essential features, improving the apparent responsiveness of a large page, even though the entire page takes the same total time to load.

To lazy load parts of a Visualforce page:

* Use the rerender attribute on Visualforce components to update the component without updating the entire page.
* Use JavaScript Remoting to call functions in your controller through JavaScript, and to retrieve ancillary or static data.
* Create a custom component to show and hide data according to user actions.

Visualforce Performance: Best Practices Controlling Data Size When lazy loading pages, consider the number of users and amount of data you expect to use the page, and watch out for limits like the concurrent API call limit. For example, if a navigation tree only loads elements as needed, the number of queries might end up out of proportion to the data.

## Offloading Processing to Asynchronous Tasks

Offload expensive processing using asynchronous tasks when that processing is secondary to the purpose of the pages. For example, instead of having a user click a button and wait for a long-running task to complete before seeing a confirmation message, consider queuing the long-running task for asynchronous processing and returning control to the user immediately. Configure the page to notify the user via email or some other means when the task completes.

## Caching Global Data in Custom Settings

Visualforce pages sometimes use calculation results globally; that is, the pages use the same data across users and requests. Improve the performance of such pages by caching calculation results in a custom setting and refreshing the results periodically instead of upon every request. Custom settings are part of an application’s cache and do not require a database query for retrieval. Balance this approach against the time it takes to update custom cached data.

## Writing Efficient Apex and SOQL

The Apex and SOQL that a Visualforce page leverages impacts the overall performance of the page.

When writing Apex or SOQL for use with a Visualforce page:

* Perform calculations in SOQL instead of in Apex, whenever possible.
* Never perform DML inside a loop.
* Filter in SOQL first, then in Apex, and finally in Visualforce.

For example, don’t do the following.

|  |  |
| --- | --- |
| For example, don’t do the following. | |
| <apex:pageBlockTable value="{!positions}" var="position">  <apex:column value="{!position.name}"  rendered="{!IF(position.Status\_\_c == 'Open - Approved', true, false)}"/>  <apex:column value="{!position.Location\_\_c}"  rendered="{!IF(position.Status\_\_c == 'Open - Approved', true, false)}"/>  <apex:column value="{!position.Job\_Description\_\_c}"  rendered="{!IF(position.Status\_\_c == 'Open - Approved', true, false)}"/>  </apex:pageBlockTable> |

## Writing Efficient Getter Methods

Visualforce requests evaluate expressions, action attributes, and other method calls, sometimes calling the getter methods in a class multiple times, especially in post backs (form submissions). Cache the value of a property calculation so that additional calls to access the property can use the cached value, and configure the getters in your Apex classes to only query for data if the object is null.

## Optimizing the View State

To maintain state in a Visualforce page, the Force.com platform includes the state of components, field values, and controller state in a hidden form element. This encrypted string is the view state and has a limit of 135KB. Large view states require longer processing times for each request, including serializing and deserializing, and encryption and decryption. Reducing your view state size causes your pages to load quicker and stall less often.

Use the View State tab in the development mode footer to monitor view state performance, and take the following actions.

* Use the transient keyword in your Apex controllers for variables that aren’t essential for maintaining state and aren’t necessary during page refreshes.
* If you notice that a large percentage of your view state comes from objects used in controllers or controller extensions, consider refining your SOQL calls to return only data that's relevant to the Visualforce page.
* If your view state is affected by a large component tree, try reducing the number of components your page depends on. To reduce view state:
* Use filters and pagination to reduce data requiring state.
* Declare instance variables with a transient keyword when the variable is only useful for the current request. The view state includes all non-transient members in the controller and extension, as well as objects reachable from these non-transient members. Decide if some data can be read-only and use the <apex:outputText> component instead of <apex:inputField>.
* Set the Development Mode and Show View State in Development Mode permissions set to see the View State tab in the development mode footer. The tab displays the distribution of view state. Make sure you know the view state size of each page, and test with large data volumes to determine if issues might occur after deployment.
* Use JavaScript remoting. Unlike the <apex:actionFunction> component, JavaScript Remoting does not require a form component. This technique doesn’t reduce the overall view state of a page, but your page generally performs better without the need to transmit, serialize, and deserialize the view state. The tradeoff is the loss of the re-render attribute and the additional JavaScript code to handle callbacks.

## Optimizing JavaScript

Optimize your JavaScript to ensure efficient delivery to the client, improve caching, and accelerate page display in the browser.

* Consider externalizing JavaScript files. This process increases the number of initial HTTP requests, but it also reduces the size of individual pages and takes advantage of browser caching.
* Build custom versions of JavaScript libraries with only the functions you need. Many open-source JavaScript libraries, such as jQuery, provide this option, which significantly reduces the file size.
* Combine all JavaScript files into a single file to reduce HTTP requests, and remove duplicate functions as they might result in more than one HTTP request and waste JavaScript execution.
* Remove comments and whitespace (spaces, newlines, and tabs), and compress the resulting file for faster downloads.
* Put scripts at the bottom of the page. By loading scripts just before the closing </body> tag, the page can download other components first and render the page progressively.

Note: Only move JavaScript to the bottom of the page if you’re certain it doesn’t have any adverse effects on your page. For example, do not move JavaScript code snippets requiring document.write or event handlers from the <head> element.

* Consider including JavaScript files using a standard HTML <script> tag right before your closing </apex:page> tag instead of using <apex:includeScript> The <apex:includeScript> tag places JavaScript right before the closing </head> element, causing the browser to attempt to load the JavaScript before rendering any other content on the page.
* Use static resources to serve JavaScript files, as well as images, CSS, and other non-changing files. JavaScript and other assets served this way benefit from the caching and content distribution network (CDN) built into Salesforce.

## Optimizing Image Usage

Images are frequently the largest components of a Web page and thus significantly affect performance.

Follow these image tips to minimize the performance impact of images.

* Use fewer images and smaller background textures, and use CSS instead of images whenever possible.
* Use CSS sprites instead of individual images. CSS sprites let you combine a collection of similarly sized graphics, such as buttons and icons, into a single file, and then use the CSS background-image and background-position properties to display portions of the combined image. This technique reduces the number of images used—and thus the number of HTTP requests—and the combined sprite file of images is usually very easily cached.
* Use static resources to serve images, as well as CSS, JavaScript, and other non-changing files. Images and other assets served this way benefit from the caching and content distribution network (CDN) built into Salesforce.
* Compress images. Graphics tools often use default settings that favour visual fidelity over compression and add metadata when saving images. Image compression tools can compress image files 10-30% further without reducing visual quality

## Optimizing Pages for Internet Explorer

If your organization uses Microsoft® Internet Explorer®, optimize your Visualforce pages as follows.

* Avoid the AlphaImageLoader filter. It blocks rendering and freezes the browser while the image downloads. It also increases memory consumption and applies per element, not per image, so the problem multiplies.
* To include additional stylesheets, use <link> instead of @import. Using @import changes the download order and hinders progressive rendering.
* Avoid CSS expressions. While they allow you to set CSS properties dynamically, Internet Explorer evaluates them more often than expected.

## Visualforce Performance Optimization Case Study

To understand how these best practices work together in actual situations, consider a case in which a customer had a Visualforce page with a data grid to collect sales forecasts. The data model for the forecast contained a multilevel object hierarchy. The page also contained calculations to display pivoted data. For an average user, the grid contained roughly 1,500 cells, causing the page to load slowly, and hit heap and view state limits.

If you are in a similar situation and want to optimize page performance:

* Make the page task focused, using it only for input. Using the page for both input and some aggregate reporting adds unnecessary complexity.
* Create a custom object to hold aggregate data for reporting. Removing the formulas needed to display aggregated information reduces the heap size.
* Relax some of the design requirements, most notably pagination.
* Avoid displaying every account on a single page. This practice improves both page load speeds and the view state size.
* Make the data grid cells read-only. Have users click on cells to edit them, and use Ajax to save their edits. These practices have a big impact on view state.

# Data Migration

1. ***Know what order objects should be inserted and what IDs must be preserved in the migration process*** – While provisioning the target system, Pervasive worked with Salesforce, and subject matter experts from all stakeholders on logical object planning to determine which objects were necessary for migration and how they were related.
2. ***Identify quality rules and discriminators*** – Determining rules which the data should follow is essential. Use a data profiler to profile the data and retrieve score cards to analyze what problems have to be dealt with.
3. **Set up a proper mirrored test environment to determine how long it will run, and to catch any volume or time related ‘gotchas’ before show-time** – Migrations can require a lot of time to move data, have a lot of bandwidth and API limitations to consider, and it is also  difficult to undo the migration if a problem is found.
4. **Prepare your users, whether co-workers or customers, for the switch.**– Do some upfront impact analysis of network and related systems. Provide “scheduled maintenance” reminders for anyone still using either the old or new systems.
5. **Perform the migration at an off-peak time when network traffic and risk is lowest**.
6. **Immediately begin validation of the data once the migration is complete. Don’t delay.** If there are any problems, finding them fast is vital.

## Points to be noted for data migration

* During data load sharing rule needs to be deactivated with the help of sales force or After data load sharing rule needs to be created .
* Owner Id should be active for the record which is being loaded.
* In case of users migration, Email alert are **NOT** sent to the users when active users are loaded into sales force  using data loader since ‘Sending Notification email checkbox is not available during the data load. In order to overcome this situation , Please upload the users with active flag set to false.  Then update users active flag to true.
* Object sequence should like profile/user/Account/contact/Opportunity/task/events should be thought through and decided before migration

# Appendix

## Salesforce Limits

### Configuration Limits for Reference

Refer the below link for Salesforce Limits Quick Reference Guide.

<https://ap1.salesforce.com/help/pdfs/en/salesforce_app_limits_cheatsheet.pdf>

## Learn SFDC's Object Definition Convention

Recommendation Level: Medium

Every object on SFDC is assigned a 3 character ID often called a KeyPrefix upon creation. For custom objects, this KeyPrefix is variable; it may change when the object is moved from one Org to another. Standard SFDC objects on the other hand, have constant KeyPrefixes; meaning it’s the same 3 character sequence on any Org.

The following is a partial list of the Standard Object KeyPrefixes on SFDC:

|  |  |  |
| --- | --- | --- |
| STANDARD OBJECT | KEYPREFIX | |
| Account | 001 |
| Contact | 003 |
| Opportunity | 006 |
| Lead | 00Q |
| Task | 00T |
| Event | 00U |
| User | 005 |

The KeyPrefix is used as the first 3 characters of the ID of a record from the object:

|  |  |  |
| --- | --- | --- |
| STANDARD OBJECT | KEYPREFIX | |
| Account | 00190000006N1eR |
| Contact | 00390000005n97M |

These can also be observed on the URLs of the object:

|  |  |  |
| --- | --- | --- |
| TYPE | URL | |
| Account: Tab View | https://na1.salesforce.com/001/o |
| Account: New | https://na1.salesforce.com/001/e |

Knowing the KeyPrefix convention will be helpful when sorting out records with polymorphic keys like the Task's WhatId field:

|  |  |
| --- | --- |
| SOLUTION : CHECK THE CLASSIFICATION OF THE RECORD USING ITS ID | |
| public class SFDCObjectSniffer {  public static final String KEYPREFIX\_ACCOUNT = '001';  public static final String KEYPREFIX\_CONTACT = '003';  public static final String KEYPREFIX\_OPPORTUNITY = '006';    public static String getKeyPrefix(ID recID) {  return ((String)recID).substring(0, 3);  }    public static Boolean isAccountID(ID recID) {  return getKeyPrefix(recID).equals(KEYPREFIX\_ACCOUNT);  }  public static Boolean isContactID(ID recID) {  return getKeyPrefix(recID).equals(KEYPREFIX\_CONTACT);  }  public static Boolean isOpportunityID(ID recID) {  return getKeyPrefix(recID).equals(KEYPREFIX\_OPPORTUNITY);  }  } |

***Additional Reading:***

<http://www.salesforce.com/us/developer/docs/api/Content/sforce_api_calls_describesobjects_describesobjectresult.htm>

## Learn SFDC's URL Convention

Recommendation Level: Low

The most common URL parameter one may encounter on SFDC would be the retURL parameter. The purpose of the URL is usually for the Save or Cancel action on a page. The retURL will contain the encoded version of the originating page's URL where the user will be redirected to upon Save or Cancel.

Another parameter that most SFDC pages support is the isdtp parameter. The isdtp param modifies the layout of the page by removing the sidebar and reducing the font size. The primary purpose of this special parameter is for use on the SFDC Console. isdtp is short for “Is Desktop” which is another name for the Console. This parameter accepts a couple of values:

|  |  |
| --- | --- |
| isdtp VALUE | EFFECT |
| mn | Renders the page using the old SFDC look and feel. |
| vw | Renders the page using the current SFDC look and feel. |
| nv | (Only for the Service Cloud Console) Renders the page using the current SFDC look and feel while adding a Refresh button. |

Though mainly used on the Console, the “isdtp” param may also be used outside of its original container. This can be easily verified by appending the parameter to most standard pages:

<https://na1.salesforce.com/001/o?isdtp=vw>

This renders the Account Tab view using the styling for the Console.

## Learn the Features of the SFDC Migration Tools

Recommendation Level: Medium

SFDC currently provides 3 methods of customization deployment. Each one has its own strength:

|  |  |  |
| --- | --- | --- |
| DEPLOYMENT TOOL | ADVANTAGE | DISADVANTAGE |
| Change Sets  *Requirement: Browser* | - Easy to use  - Independent of local bandwidth  - Historical  - Supports deployment simulation | - Only supports deployment across a Production Org and its Sandboxes  - Deployment progress unclear  - Browser may become unresponsive during deployment  - Tedious hunt-and-select interface |
| Force.com IDE  *Requirement: Java, Force.com IDE* | - Easy to use  - Ad Hoc  - Supports deployment simulation  - Supports deployment across any Org | - Dependent on local bandwidth  - No automatic historical records  - Deployment progress unclear  - IDE may become unresponsive during deployment |
| Force.com Migration Tool  *Requirement: Java, Ant* | - Deployment files may be considered historical  - Reusable deployment files  - Supports deployment simulation  - Supports deployment across any Org  - Option for verbose progress output  - Heavily customizable | - Dependent on local bandwidth  - No automatic historical records  - Requires knowledge of Ant and SFDC Metadata API  - Command-line interface |

***Additional Reading:***

<https://login.salesforce.com/help/doc/en/changesets.htm>

<http://wiki.developerforce.com/index.php/Force.com_IDE>

<http://wiki.developerforce.com/index.php/Migration_Tool_Guide>

## API Development

To develop Web service client applications, it is strongly recommended that you use Salesforce.com Sandbox Edition, which is an exact replica of your Salesforce.com deployment, including all customization and data. For more information, see <http://www.salesforce.com/products/sandbox.jsp.>