Salesforce Platform CoE

ABB Salesforce Platform Development Guidelines

1 Introduction

This document outlines Salesforce Platform Development Guidelines for all applications running on Salesforce.com platform in ABB (GAR# 9AAG10077), i.e.:

- Salesforce CRM
- Force.com applications
- Apttus Quote
- Salesforce Analytics
- Salesforce Einstein
- Salesforce Marketing Cloud
- Salesforce Field Service
- Pardot

Those architecture guidelines should be enforced by all projects delivering functionalities to ABB Salesforce Platform governed by CoE. Customer will facilitate a training session with supplier's team to explain documented architectural guidelines if required. Customer will verify code & design quality via code scan tools and code review processes. The supplier is responsible to correct the source code & system design before end of SIT phase. Customer will also be responsible in generating these reports and sharing them on timely basis to the project teams. In case the supplier is unable to fix all the issues raised by customer before UAT phase, then the supplier has to agree on a timeline with the customer by when these open issues will be fixed.

1.1 Revisions

Rev. ind.	Date	Description	Name
A	2017-07-24	First Draft	Przemyslaw Bucharowski
B	2017-08-18	Document reviewed	Bartosz Borowiec, Jarosław Kondrat, Tadeusz Hyży
1.0	2017-08-18	Document released	Przemyslaw Bucharowski
1.1a	2017-08-18	Draft version prepared for amendments	Przemyslaw Bucharowski
1.1b	2017-08-31	Changed introduction & added exception handling	Przemyslaw Bucharowski
1.1c	2017-10-10	Added integration user best practices	Przemyslaw Bucharowski
1.1d	2017-10-16	Updated browser best practices	Przemyslaw Bucharowski
1.1	2017-10-27	Document reviewed & released	Bartosz Borowiec, Jarosław Kondrat, Tadeusz Hyży, Przemyslaw Bucharowski
1.2a	2018-03-05	Added new best practices	Przemyslaw Bucharowski
1.2	2018-03-12	Document reviewed & released	Bartosz Borowiec, Jarosław Kondrat, Tadeusz Hyży, Przemyslaw Bucharowski
1.3a	2018-05-23	Added bypass logic, checklist to check if the test class follows architectural guidelines and UserDAO class.	Bartosz Borowiec

1.3	2018-05-30	Document reviewed & released	Bartosz Borowiec, Jarosław Kondrat, Tadeusz Hyży, Przemyslaw Bucharowski

1.2 Terms and Definitions

Production	Salesforce Production environment (https://abb.my.salesforce.com)
Release	Any project that is delivering new functionalities to Salesforce Platform
Customer	ABB

1.3 Roles and responsibilities

Salesforce Platform Lead Solution Architect	Przemyslaw Bucharowski przemyslaw.bucharowski@pl.abb.com
Salesforce Platform Solution Architect	Bartosz Borowiec bartosz.borowiec@pl.abb.com
Salesforce Platform Solution Architect	Jaroslaw Kondrat jaroslaw.kondrat@pl.abb.com
Salesforce Platform Solution Architect	Tadeusz Hyzy tadeusz.hyzy@pl.abb.com

1.4 General Description

The objective of this document is to provide Salesforce developers with consistent Salesforce Platform Development Guidelines & architecture directions.

The targeted audience of this document are:

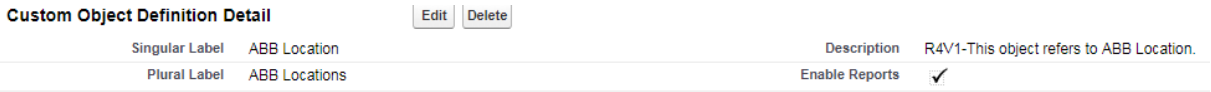
1. Project solution architects
2. Project developers.

2 ABB Salesforce Platform Design Principles

Below you will find key Salesforce design principles that every Release should follow:

- All components updated / created as part of the Release, must have an updated description with the Release version and the change / purpose of the component.

  For example:



- All methods in Apex class must have proper description on the functionality of the method, author details and Release version (this includes changes to existing code & new code introduced by Release). Use code comment syntax to store that information.
- Every time you add a new field, objects, report type etc. add both description and help text. Description is dedicated for your fellow developers and administrators, which may not know the application - explain to them what the field is responsible for, where is it used and put there any other nontrivial information; include Release number. Help text will be useful for your users - this field should have info about allowed data format, validation rules and other information useful for end user.
- Developers must ensure the code is written/tested & executed within Salesforce governor limits.
- Developers must ensure the code is scalable. When introducing functionality e.g. for one pilot country, design it the way it scales up.
- All custom logics developed must be bulkified to support insert/ upsert of bulk data load except EmailHandlers and controlers
- The Salesforce Order of Executions must always be respected when developing business automations. Do not write code that fall into infinite loop.
- Avoid Synchronous Callouts when designing integrations. Use only if very short response time is guaranteed (there is Org wide limit of 10 Calls with >5 seconds response time).Usage of synchronous integration pattern should be justified in written & signed-off by Salesforce Platform Solution Architect during design phase.
- When designing integration, adjust UI to support asynchronous responses, using standard Salesforce methods like: Continuation (Continuation Patterns) or Streaming API (Streaming API).
- All masterdata/globally reusable picklists should utilize Salesforce Picklist Value Sets that will be governed centrally by platform support team. Before introducing new picklist on object, check if Picklist Value Sets already exist.
- When custom buttons are created, they should be Lightning Experience & Salesforce mobile compatible.
- All Salesforce functionalities (especially VisualForce pages & Lightning components) must be developed & tested against mobile devices & ABB selected browsers (Microsoft Internet Explorer 11 & latest Google Chrome) for compatibility & usability.
- All validation rules must use the existing Custom Setting which will bypass the logic for certain profiles. { NOT($Setup.Validation_Override__c.Exclude__c) }
- When new validation rules are included, this must also be updated in the test classes to avoid failures and better code coverage

- System Administrator profile must always have READ & WRITE access to all fields created on platform
- The following profiles/permission sets must always complete read permission (NOT VIEW ALL) on all objects, fields, VisualForce pages and Apex Classes (this step is valid until common profiles are introduced):
    - Profile: “Analytics Cloud Integration User”
    - Profile: “ABB Copy Storm Integration”
    - Permission Set: “System: Local UAM”
- If there are new fields created for an object, the developer should check if there are any integration profiles utilizing this object. If yes, developer must reach out to platform support team and ensure relevant access for this profile is set. For example, any field created on Opportunity must be available for CQP integration.

3 IDE - Integrated Development Environment

3.1 What is IDE and why use it?

IDE - integrated development environment (Wikipedia) – is software which helps developer writing code faster and better. Depends on which IDE developer will choose it will have different helpful tools but developing in Force.com one is common and very useful. Thanks to IDE we can resolve conflicts between server and local code and it will not allow to save us without resolving it.

Basically IDE is the first thing you need to learn to start coding and not only in Force.com but also for other languages (Java, C++, C#, PHP etc.).

3.2 Which for Force.com IDE is best?

It's really depends on preferences but any Force.com IDE is good enough to write Classes, VisualForce Pages or even Lightning Components. Below you can find list of IDE's for Force.com

Name	Paid?	Link
Eclipse Force.com IDE	No	https://developer.salesforce.com/page/Force.com_IDE
JetForcer	No	https://plugins.jetbrains.com/plugin/9238-jetforcer--the-smartest-force-com-ide
Salesforce Extensions for VS Code	No	https://marketplace.visualstudio.com/items?itemName=salesforce.salesforcedx-vscode
Illuminated Cloud	125$ annually	http://www.illuminatedcloud.com/
The Welkin Suite	150$ annually	https://welkinsuite.com/

3.3 What about Notepad++ or MS Notepad?

If your Notepad++ or MS Notepad can handle server <> local file comparing and code merging you can use it but it's not recommend to use any kind of simple text editors to change components from Salesforce.

4 Apex Best Practices

4.1 Dealing with Apex Limits

When starting coding in Salesforce, the one very basic thing that make Apex different from all other programming languages is limits. Because we are working within multi-tenant environment, shared with other organizations instances of Salesforce, Salesforce impose many limits on developer to avoid situation where one organization takes over resources (CPU time, Database access, RAM) dedicated to whole cluster of organizations.

4.1.1 DML Statements

4.1.1.1 Avoid DML statements inside FOR Loops

When applying a change to a large set of data (insert, update, delete), it may seem a good idea to perform it on each record separately. Unfortunately, Apex code has a limit of 150 DML statements per context, which makes it impossible for this code to work if we have more than 150 records in contact list:

```
List<Contact> contacts = [Select id, salutation, firstname, lastname, email
                From Contact];

for(Contact c: contacts) {
        c.Description=c.salutation + ' ' + c.firstName + ' ' + c.lastname;
        update c;
}
```

The correct approach is always to create a collection, where we store all changed records and then update them in one DML statement:

```
List<Contact> contacts = [Select id, salutation, firstname, lastname, email
                From Contact];
List<Contact> contactsToUpdate = new List<Contact>{};
for(Contact c: contacts){
                c.Description=c.salutation + ' ' + c.firstName + ' ' + c.lastname;
                contactsToUpdate.add(c);
}
update contactsToUpdate;
```

4.1.1.2 Avoid SOQL queries inside FOR Loops

In Apex we will often encounter a situation, when we receive an Id of the parent object and have to query for its children. It is important to avoid putting such SOQL queries in FOR loops, because of 100 queries per context limit.

So instead of:

```
trigger accountTestTrggr on Account (before insert, before update) {
  for(Account a: Trigger.new) {
    List<Contact> contacts = [Select id, salutation, firstname, lastname, email
                    From Contact Where accountId = :a.Id];
    for(Contact c: contacts) {
      // perform Contact operations
    }
  }
}
```

It is suggested to query for all objects in just one, outside-of-loop query:

```
trigger accountTestTrggr on Account (before insert, before update) {

  List<Account> accountsWithContacts = [Select id, name, (Select id, salutation,
                    Description, firstname, lastname, email From Contacts)
                    From Account Where Id IN :Trigger.newMap.keySet()];
  for(Account a: accountsWithContacts ) {
    for(Contact c: a.Contacts) {
      // perform Contact operations
    }
  }
}
```

As you could see, we also utilized the relationship between Account and Contact to simplify and unify the query. It's a really good habit!

4.1.1.3 Querying Large Data Sets

Two limits are especially important when querying large datasets - number of records retrieved (50 000) and heap size (6MB for synchronous, and 12MB for asynchronous calls). Although the row number limit is a hard one and it is not possible to avoid it in a single context, there is a language construct that helps in reducing the heap size when dealing with great amount of data.

So, instead of putting 40 000 records on a heap with one query:

```
Account[] accts = [SELECT id FROM account];
```

It is suggested to rather divide it into smaller chunks of 200 with for loop SOQL query:

```
for (List<Account> acc : [SELECT id, name FROM account]) {
  acc.size() // returns 200, unless last chunk
}
```

4.1.1.4 Writing Selective SOQL Query

"A query is selective when one of the query filters is on an indexed field and the query filter reduces the resulting number of rows below a system-defined threshold. The performance of the SOQL query improves when two or more filters used in the WHERE clause meet the mentioned conditions.
The selectivity threshold is 10% of the records for the first million records and less than 5% of the records after the first million records, up to a maximum of 333,000 records. In some circumstances, for example with a query filter that is an indexed standard field, the threshold may be higher. Also, the selectivity threshold is subject to change." - help.salesforce.com

Above threshold values are valid for custom indexes. Standard indexes have higher threshold which are 30% for first million and 15% for all records after million, up to maximum of 1 million records.

A non-selective query may cause different programmatic elements (like an Apex trigger or batch Apex class) to fail. When querying large objects, it is important to not exceed the threshold limits.

It can be achieved by:

- Using indexed fields in a query (primary keys, foreign keys, audit dates, custom fields marked as External ID or Unique).
- Analyzing slow queries in Query Plan and optimizing them (Query Plan can be turn on in Developer Console and used in in Query Editor tab of DC)
- Avoiding formula having relations to parents in WHERE clause
- Writing the code so that it does not require immediate access to more than 10% of the object records.
- Avoid using 'NOT' and '!='. Even if the field is indexed—the Force.com query optimizer can't use the index to drive the query.

4.1.2 Code Bulkifying

Because of query number, DML statements, CPU time and other limits, it is a suggested solution to build your methods in classes so they are prepared to handle collections of records instead of single objects.

Not bulkifying the code can lead to violating business logic:

```
trigger accountTestTrggr on Account (before insert, before update) {

  // This only handles the first record in the Trigger.new collection
  // But if more than one Account initiated this trigger, those additional records
```

```
  // will not be processed
  Account acct = Trigger.new[0];
  acct.Description = acct.Name + ':' + acct.BillingState;
}
```

Or violating the limits (DML/CPU time) inside the helper class:

```
class BadPracticeAccountHelper {

  public static void fixContacts(List<Account> accounts) {
    List<Accounts> accWithCons = [SELECT Id, Name, (SELECT Id, Name FROM Contacts)
                        FROM Accounts Where Id in :accounts];
    for(Account a : accWithCons) {
      for(Contact c : a.contacts) {
        helperClass.fixContact(c); // this method may perform costly operations
                        // or even DML statements
      }
    }
  }

  class helperClass {
    public static void fixContact(Contact c) {
      // perform some actions on contact
      update c;
    }
  }
}
```

This classes should be rewritten to:

```
trigger accountTestTrggr on Account (before insert, before update) {

  List<String> accountNames = new List<String>{};
  for(Account a: Trigger.new){
    a.Description = a.Name + ':' + a.BillingState
  }
```

```
}
```

and

```
class GoodPracticeAccountHelper {

  public static void fixContacts(List<Account> accounts) {
    List<Accounts> accWithCons = [SELECT Id, Name, (SELECT Id, Name FROM Contacts)
                        FROM Accounts Where Id in :accounts];
    List<Contact> toUpdate = new List<Contact>(); // to eliminate the update from helper
    for(Account a : accWithCons) {
      toUpdate.addAll(helperClass.fixContacts(a.Contacts));
      }
    }
    update toUpdate;
  }
  class helperClass {
    public static List<Contact> fixContact(List<Contact> cons) {
      // perform some actions on contacts
    }
  }
}
```

4.1.3 Responsible using of @future annotations

Using @future annotation is a good habit, allowing to remove a little bit of workload from main context, but it is a power that must be used with great wisdom and responsibility. There is a limit of 20 future callout per context and 200 times number of Salesforce Licenses per 24 hours. Especially the former is easily exceedable, which means that, similar to DML statements or queries, future callouts should not be made in loops.

As the other methods, @future callouts should also work on Collections of objects instead of single instances to increase efficiency.

4.1.4 Using Workflows or Flows rather than Apex for mail sending

It often seems convenient to perform the mail sending operation from the Apex code with SingleEmailMessage, but it is extremely unscalable solution. Each organization has a limit of 5000 external email addresses send from Apex code, which in medium and large organization can be easily exceeded. Always try to send emails as workflow action, which allows you to omit the restriction, as those mails do not count to the limit.

You can send an unlimited amount of email to your org's internal users, which includes portal users.

You can send mass emails only to contacts, person accounts, leads, and your org's internal users.

4.1.5 Checking the Limits

If you suspect that your code may exceed the limits you may want to check for your current usage and stop the code from executing if you are close to exceeding. It can be achieved with use of Limits class and its methods getLimit… and get… (e.g. getLimitDmlRows and getDMLRows).

4.2 Triggers

Triggers should not have business logic coded. Use Salesforce declarative methods instead (it will be easier to support).

All triggers to have a switch control using Custom Label/ Custom Settings, which can help disable the trigger for maintenance & migration tasks.

All Triggers should utilize the Trigger Factory Framework within their Module-specific Trigger Handler classes.

Each object will have its own Trigger Handler class, with an exception of Master-Detail related objects, where trigger logic is dependent in all contexts, making a shared handler more efficient.

4.2.1 Writing multiple triggers for the same object

Don't do that. This is not a good practice. If possible, avoid writing multiple triggers for the same object, especially for the same event (e.g. before insert).

First of all, there is no guaranteed order of execution of those trigger, which may cause errors or invalid data at the end when the triggers will overwrite each other results. What's more, all triggers invoked as a result of single DML statement share the same context, which, while having multiple triggers, may easily lead to limit violation, mostly because of using the same queries in each trigger.

You will probably end up with multiple triggers in your organization anyway, because of AppExchange apps you are going to install - there is no reason to increase this number with your own actions.

4.2.2 Trigger Handlers

A good design pattern for writing triggers is a trigger handler, which helps avoiding multiple triggers, support determining order of actions and significantly increase code readability.

The pattern can be implemented in many different ways, but the basic concept is to use separate class for trigger and trigger actions.

In ABB a dedicated framework for trigger handlers has been introduced. This framework is not perfect and will be a subject of a re-factorization but to keep code consistent it is recommended to use it because it will simplify re-factorization in future. Please find an example how to use this trigger factory. This is a part of an Account trigger from ABB implementation.

1) Trigger (Account) :

```
/**
* Trigger for Account
*
*/
trigger AccountTrigger on Account(after update,  before insert, before update) {
        /**
        * trigger runs only for custom label has below value
        *
```

```
                 */
If(!(ClsAccountUtil.isAccMergeFlag) && !System.label
.SYS_RunAccountTrigger.equalsIgnoreCase('NO'))
                 {
                   System.debug('::MR::AccountTrigger');
                   ClsTriggerFactory.createHandler(Account.sObjectType);
                 }

}
```

It is important to mention that this trigger can be disabled via custom label (marked in green)

This trigger calls a trigger factory.

2) Trigger Factory (clsTriggerFactory)

```
public class ClsTriggerFactory {

   (...)

    private static IntTrigger getHandler(Schema.sObjectType soType) {

      (...)

      }else if(soType ==  Account.sObjectType) {
         return new ClsTriggerAccountHandler();
      }

      (...)

   )

}
```

Trigger factory creates and executes trigger handler that inherits from ***ClsTriggerVirtual*** virtual class and implements ***IntTrigger*** interface. It is essential to mention that lines marked in yellow have to be added to ***clsTriggerFactory*** for every single handler.

3) Trigger hander (ClsTriggerAccountHandler)

```
public without sharing class ClsTriggerAccountHandler extends ClsTriggerVirtual {
   public Static Set<Id> processedIdSet = new Set<Id>();
   /**
    * @see IntHelper.beforeTrigger
   */
   public override void beforeTrigger(ClsWrappers.TriggerContext trgCtx) {
     (...)
   }
   /**
    * @see IntHelper.afterTrigger
```

```
    */
    public override void afterTrigger(ClsWrappers.TriggerContext trgCtx) {
      (…)
    }
  (…)
  }
```

In this class implementation of virtual methods (not all only needed) from IntTrigger interface should be provided.

In case new trigger would be created, it is recommended for a developer to go through AccountTrigger implementation and analyze current TriggerFactory pattern.

4.2.3 On/Off Switch For Triggers

This practice is not crucial, but is a good practice that allows to better control over the system.

Sometimes, especially in tests, it is not desired to fire all the triggers, because of additional computing time or because of a lot of additional data necessary for them to not fail. When we are not interested in testing the triggers, but rather other, not involved functionalities, it seems like a good solution to be able to turn off the triggers.

Unfortunately, it is not possible in Apex.

So this is something we need to implement ourselves. The preferable solutions is to create a List Custom Setting for Trigger Managing with fields:

1. Name of the sObject (String)
2. Trigger Enabled (on/off)

It can be extended to separate field for each trigger (e.g. Before Insert Trigger Enabled, After Delete Trigger Enabled) and so on - it depends of granularity you require.

4.2.4 Delegate workload to jobs if possible

Sometimes it is necessary to perform operations that have large computation complexity in triggers. It is a good habit to delegate such workload to jobs and batches from trigger and trigger handler, to avoid exceeding any limits. In large organizations, triggers can have many possibilities and can easily exceed Apex Limits because, as was mentioned, all object triggers share the same context.

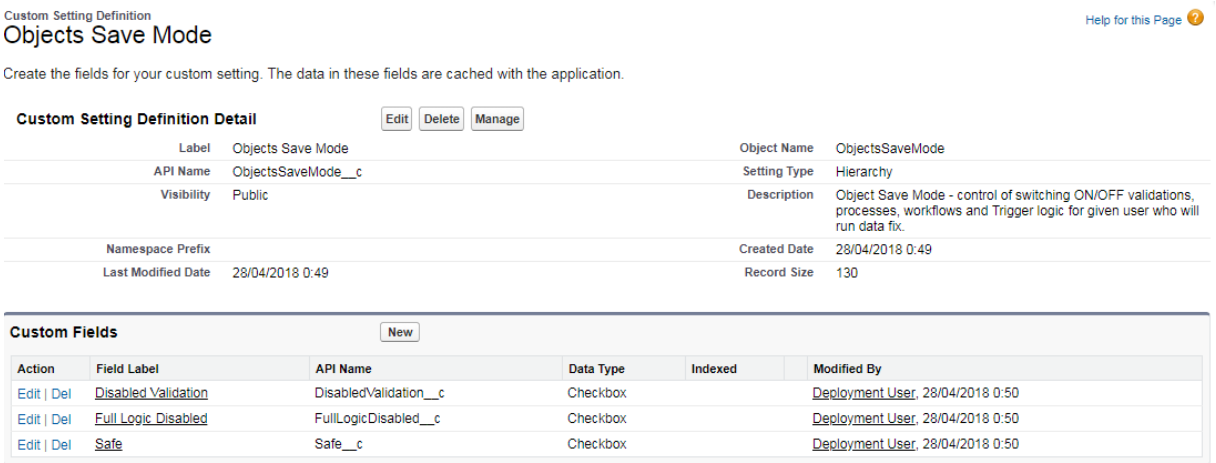
4.3 Implementation of logic bypass.

The architecture of system did not allow to proceed data fixes in background when system is active. This led to downtime of system every time when any data fix was being processed. Because of rapid grow of volume of stored data downtime needed to proceed data-fixes also grows and becomes unacceptable for the business. Moreover in every minor and major release an additional logic is being added to triggers, workflow rules and processes and this logic should be also reflected in data fixes. Recently (Release 2) logic introduced to Lead and Campaign objects allows running data fixes when system is active and allows to disable only “dangerous” part business logic during data fixes execution. In next releases new objects will be added.

Every new validation rule, trigger, workflow lookup filter and process implemented should follow this approach.

4.3.1 Logic bypass for data fixes from architectural perspective:

The custom setting ObjectsSaveMode__c has been introduced. It is a switch that contains 3 checkboxes representing corresponding modes:



Full Logic Enabled (all checkboxes not set) – This is default “objects save mode” for a user. All validations, processes, workflows and triggers will be executed when user with this mode set insert or update an object.

Disabled Validation (DisabledValidation__c Checkbox Set) – This mode will be used mostly by L2/L3 support. We could imagine following scenario when it is needed:

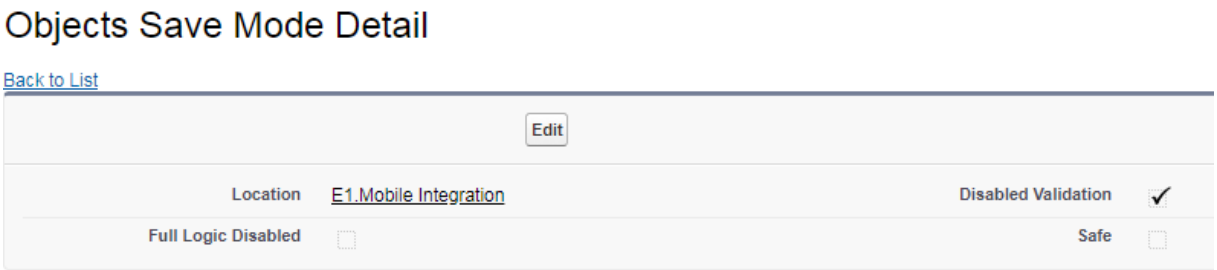
- New validation rule has been introduced. For example there is a validation rule Non_China_EP_Specific_Approver_Selection on a Campaign object (Approver has to be selected for Approval Process other than China with Division EP). This rule has been introduced 09/12/2016.
- Some Campaigns created or modified before 09/12/2016 do not have an approver.
- On 20th of January2 2017 business found that for 25 campaigns created before 09/12/2016 were wrongly updated during reorg and BU has to be changed, and asked L2/L3 to do it. L2/3 realized that 10 campaigns do not have an approver and cannot be updated. Therefore they have to contact owners of these campaigns to set an approvers, before update BU on them.
- With the disabled validation mode they will be able to update these campaigns without contacting campaign owners.

Full Logic Disabled (FullLogicDissabled__c checkbox set) – all validations, processes, workflows and triggers will not be executed when user with this save mode updates an object. This allows to run data fixes without downtime, but logic from triggers, workflows and processes related to changed fields should be re-implemented in a data fix (batch, external database stored procedure, e.t.c.).

Safe (Save__c checkbox set) – When user with safe mode updates objects all validationrulesd and “dangerous” logic (creating new tasks, update timelines or sending an email to the user) will be disabled. For example opportunity object’s workflow rule “Send Questionaire and Follow-up” creates Task and sends an email; business does not want these tasks and emails to be created and sent for old closed opportunities when these opportunities are being modified during reorg data fixes. Other logic will be enabled, and this will simplify the development of data fixes.

Mode	Validation	Triggers	Workflows, Processes, Flows	Dangerous logic
Full logic enabled (all checkboxes not set)	Enabled	Enabled	Enabled	Enabled
"Disabled validation" checked	Disabled	Enabled	Enabled	Enabled
"Full logic disabled" checked	Disabled	Disabled	Disabled	Disabled
"Safe" checked	Disabled	Enabled	Enabled	Disabled

Example of disabling validations for E1. Mobile Integration user:



4.3.2 Technical Design for logic bypass

New custom settings (ObjectSaveMode__c) has been introduced.

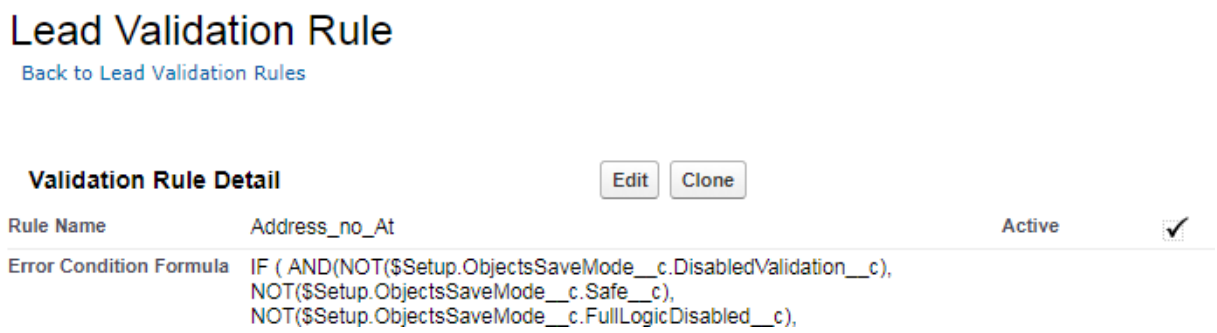
1) This custom settings contain 3 checkbox fields:
   - DisabledValidation__c
   - FullLogicDisabled__c
   - Safe__c
2) Triggers, workflows, processes and validation rules should be written to behave in the following way:

	Disabled Validation	Safe	Full Logic Disabled	
1	0	0	0	Full logic enabled
2	0	0	1	Validation is disabled, dangerous logic is disabled, all triggers are disabled, and workflow and processed are disabled

3	0	1	0	Validation is disabled, dangerous logic is disabled, triggers, workflows and processes are enabled
4	0	1	1	The same as for row 2.
5	1	0	0	Validation is disabled, dangerous logic (is enabled), all workflows, processes and triggers will execute
6	1	0	1	The same as for paragraph 2
7	1	1	0	Validation is disabled (also in triggers), dangerous methods will not run
8	1	1	1	The same as for paragraph 2

3) If all of this flags are set to false (DisabledValidation__c==false && FullLogicDisabled__c==false && Safe__c==false) system works in FullLogicEnabled mode.

   Example usage in validation rule: NOT($Setup.ObjectsSaveMode__c.DisabledValidation__c)



4) In user object there are 3 fields present below which reflect values in custom setting ObjectsSaveMode__c:

- DisabledValidationMode__c
- FullLogicDisabledMode__c
- SafeMode__c

These fields should be checked in “simple” workflow rules, because workflow rules that do not use formulas cannot directly access custom settings.

Workflow Rule Detail Edit Clone Deactivate

Rule Name	STS_Lead_Ext _Queues	Object	Lead
Active	✓	Evaluation Criteria	Evaluate the rule when a record is created, and every time it's edited
Description	STS-PR2 - All leads that are assigned to one of the portal queues must change record type to Prospect. If the Lead owner is queue and contains "-External User-Lead Queue". Work Id - 953		
Rule Criteria	AND(NOT($Setup.ObjectsSaveMode__c.FullLogicDisabled__c),CONTAINS(Owner_Display_Name__c , $Label.STS_External_User_Lead_Queue))		

5) Lookups filters:
Required lookup filters can be used in Salesforce. Filters that blocks updates of the records that do not meets filtering criteria. Unfortunately in lookup filters, formula fields cannot be used therefore a dedicated field on the user has been introduced **DisabledLookupFilter-Mode__c**. This field should be added to required lookup filters to bypass them:

Lookup Filter

Filter Criteria	(Lead: Business Unit CONTAINS ProductGroup: Business Unit) OR (Current User: DisabledLookupFilterMode EQUALS True)
Filter Type	**Required.** The user-entered value must match filter criteria.
Error Message	Business Unit In Product and Lead doesn't match
Lookup Window Text	
Active	✓

4.4 Tests

In Salesforce, it is required to have at least 75% code coverage to be able to deploy the code to production. This requirement is supposed to encourage thoughtful testing of implemented business logic, but can also result in low quality tests that do not do anything else than just execute the methods.

Whenever there is a change in the system logic like adding a validation rule, making fields mandatory, the development team must ensure to update all relevant Test Classes.

Null point error checks must be performed as part of Unit testing during Build phase.

4.4.1 Assertions

Assertions are very important part of tests and although Salesforce does not require to have it in each test - the best practice does.

Few tips about making meaningful assertions:

- Assertions should checked for changes/values enforced by your code, not values enforced by other parts of applications
- Assertions should reference constants rather than literal Strings
- Whenever possible, use the message option in the assertion to give meaningful feedback about test failure, especially with non-obvious assertions (e.g. instead of System.assertEquals(true, false), write System.assertEquals(true, false, “Exception was not thrown”));

4.4.2 Bulk testing

Your code should be able to handle collections instead of single records. The same requirement refers to test code, especially the one dedicated to triggers. The good practice is to test with a little bit more than 200 records, as the triggers usually divide the records in packages of 200.

4.4.3 DataLoaderUtil

There is Util class which will help you creating data for your tests. Class name is **DataLoaderUtil** and it's available on GIT Master Branch. There are help comments above each method but here you have some examples below that you can easily use.

```
/*
 * Create Test User
 * This method returns random string with default 8 length
 * If Integer passed as param X it will return string with X length.
*/
DataLoaderUtil. generateRandomString();

/*
 * Create Test User
 * This method returns inserted User with some default values
 *   @name = generateRandomString()
 *   @profile = 'ABB Standard User READONLY'
 *   @role = 'ABB Global'
 *   @timezone = 'en_US'
 *   @timezonekey = 'America/Los_Angeles'
 *   @encoding = 'UTF-8'
 * There are 6 more overloaded methods (with the same name) that will help override default User
params.
 */
DataLoaderUtil.createUser();

/*
 * Create Contact Object (String sObjectTypeName)
 * This method returns inserted Contact record with some default values, filled required fields and (if
available) pass all validation rules that are written at the beginning of the class
*/
(Contact)DataLoaderUtil.createObject('Contact');

/*
 * Create Contact Object (String sObjectTypeName, Boolean isInsert)
 * This method return Contact record without inserting it into Database
*/
(Contact)DataLoaderUtil.createObject('Contact', false);

/*
```

```
 * Create Contact Object (String sObjectTypeName, Boolean isInsert, Map<String, Object> params)
 * This method return Contact record without inserting it into Database with some field that will be inside that Contact record. Example show how we can fill Description field
*/

Map<String, Object> params = new Map<String, Object>();

params.put('Description', DataLoaderUtil.generateRandomString(55));

(Contact)DataLoaderUtil.createObject('Contact', false, params);

/*
 * Create Contact Object (String sObjectTypeName, Boolean isInsert, Map<String, Object> params, String recordTypeName)
 * This method return Contact record without inserting it into Database with different record type
*/

(Contact)DataLoaderUtil.createObject('Contact', false, new Map<String, Object>(), RecordTypeName');

/*
 * Create Contact Objects (String sObjectTypeName, Boolean isInsert, Integer length, List<Map<String, Object>> paramsList, List<String> recordTypeNamesList)
 * This method returns List of 2 Contacts without inserting them into Database. Parameters can be passed with List<Map<String,Object>>.
*/

DataLoaderUtil.createObjects('Contact', false, 2, new List<Map<String, Object>>(), recordTypeNames);
```

For more examples you can go to DataLoaderUtilTest.

Validation rules on Object can be predefined inside **DataLoaderUtil** class and we will have no problems in refactoring if new Validation Rule will be added. For example here are some predefined validation rules for Account Object:

```
  private static final Map<String, Object> ACCOUNT_VALIDATION_RULES_MAP = new Map<String, Object> {
    'Name'          => DataLoaderUtil.generateRandomString(8),
    'BillingStreet'   => DataLoaderUtil.generateRandomString(8),
    'BillingCountry'   => User.CountryCode.getDescribe().getPicklistValues()[Integer.valueof(Math.random() * 10)].getLabel(),
    'BillingCity'     => DataLoaderUtil.generateRandomString(8),
```

```
    'BillingPostalCode' => DataLoaderUtil.generateRandomString(2) + '-' + DataLoaderUtil.generateRandomString(3),
    'ABACUS_code__c'   => DataLoaderUtil.generateRandomString(3),
    'Active__c' => true
  };
```

This map will be used for Account.Default (record type) which is defined below:

```
  private static final Map<String, Map<String, Object>> VALIDATION_RULES_MAP = new Map<String, Map<String, Object>> {
    'Account.Default'   => ACCOUNT_VALIDATION_RULES_MAP,
  };
```

4.4.4 Other useful testing practices

- Test not only for positive scenarios, but also for negative ones, especially for possible nulls in data.
- Never use seeAllData annotations unless the test component won't work without it (e.g. some of Chatter API requires it). You don't want your test to depend on data from environment.
- Don't create data in your test class – reuse predefined DataLoaderUtil class.
- Whenever possible, test with runAs(User) method to assure that users have access to and only to specified data.
- Use Test.startTest() and Test.stopTest(). This way you create a new context for the tested method.
- Use comments "given", "when" and "then" in test class to distinguish between: establishing the test data, execution of tested methods and assertions. Explain in comment scenario being tested.

4.4.5 Mandatory checklist to check if the test class follows architectural guidelines:

No	Checkpoint	Comments:
1	* Test method name should be descriptive	
2	* Method cannot be longer than 50 lines	
3	* Test.startTest and Test.stopTest has to be used	
4	* Test should be executed for dedicated user not a "random admin" because it may lead to strange inconsistences during deployment. Please use runAS method	
5	* DataLoaderUtil class should be used to create data and users	
6	* public static Object DataLoaderUtil.prepareDictionaryObject(String dictionaryObjectApiName) - should be used to create dictionary objects like Product_Group__c/ABB_Location__c e.t.c.	At the time when the guide is published these method are not added to master branch of Git. These

		will be introduced in the R3 release
7	* Value from masterdata cannot be hardcoded in the code. It should be created via DataLoaderUtil.getFirstPicklistActiveValue or DataLoaderUtil.getControlingFieldValue and DataLoaderUtil.getDepenentFieldValues.	At the time when the guide is published these method are not added to master branch of Git. These will be introduced in the R3 release
8	* Negative scenario should be incorporated into test class.	
9	* ProfileDAO should be used to find Profiles	
10	* RecordTypeManager should be used to find recordsTypes ids	
11	* commented code should be removed from class	
12	* Objects with the same type should be inserted together. DMLs and SOQLs cannot be wasted	
13	* Asserts should be added to every single test method to check if code is executed in the proper way. Assertions should be very specific and should fail when something goes wrong	
14	* Asserts should be written in a detailed way. For example, when there is a method that retrieves 10 leads from a database it should check more than number or retrieved leads and if the right leads are retrieved and if all requested fields are retrieved.	
15	* System.assert(false,"process cannot reach this point") should be added in test of negative scenario when an exception is expected to be thrown	
16	* There should be a message added to assertions that describes the bug. For example the assertions should look like: System.assert(false, 'User Can assign action plan to a user that is a part of account plan team'); or System.assert(ex.getMessage().contains('It is only possible to change assignment to members of Account Plan Team.'), 'Error message');	
17	*Tests should be executed without human supervision. Debugs statements in tests are a bad practice because the issue should be found by assert and a message related to it should be displayed as assert message (third parameter) of system assert function	
18	*Catch block of a "try catch" statement cannot be empty because if something goes wrong the problem will not be found. There should be at least System.assert(false,message) in catch block of “try-catch” block in a test method.	
19	* All declared variables should be used in the code.	
20	* Batches are async and finish in before Test.stopTest method. All batches should be executed inside Test.startTest() Test.stopTest() block	
21	* All batches should be tested on at least 200 instances to check if the implementation does hit limits of SOQLs and DMLs.	
22	*SeeAllData annotation cannot be used	There are some very rare cases when it must be used but always it

		should be confirmed by an architect or a lead developer and justified in code comments signed by an author and an architect/dev lead

4.5 Other good coding practices

4.5.1 Exception handling

An exception is a special condition that changes the normal flow of program execution. Here are the most common examples:

- 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

All exceptions should be handled gracefully by your code. Apex uses the Try, Catch, Finally construct common to many other programming languages. You "try" to run your code. If there is an exception, you "catch" it and can run some code, then you can "finally" run some code whether you had an exception or not. You can have multiple Catch blocks to catch any of the 20+ different kinds of Apex exceptions.

Whenever possible, replace standard exception message with custom one that is meaningful to business user for particular use case. Include “call to action” in the message (e.g. “Please provide valid phone number” or as last resort: “Contact ABB support” or “Contact your local administrator”).

In code deployed to production, always notify Salesforce Support Mailbox (CH-salesforce.support@abb.com) via email on exception. Do not send notifications to developers’ mailboxes.

4.5.2 Don’t use hardcoded IDs

The IDs change between your development sandbox, test environment and production, so it is absolutely forbidden to hardcode any ID, because the code will immediately stop working once deployed to another organization. In tests, insert your own data.

4.5.3 Record type IDs.

Salesforce does not have a method to find a record ID by a developer name as a key. Only a record label can be used (this is not recommended because a record label can be a subject of a language translation). Therefore a custom framework has been implemented to address this issue (class ***RecordTypeManager***). This class keeps a map of record SObjectType + '.' + rt.DeveloperName in a static variable that is initialized during a first usage. Therefore this Class uses only one SOQL regardless how many times it is used during one session. Please find a short description of methods of this class:

	Method	Description
1.	getRecordTypeId(String sobjectType, String recordTypeName)	Returns id of the record type based on subjectName and record type name. When an Id cannot be found an empty string '' is returned. For example ***RecordTypeManager. getRecordTypeId('Account', 'External_Non_Buying_Account')***
2.	getRecordTypeId(String sobjectType, String recordTypeName)	Returns id of the record type based on subjectName and record type name. When an Id cannot be found an empty string '' is returned. For example ***RecordTypeManager. getRecordTypeId('Account.*External_Non_Buying_Account') –** only one parameter
3.	getRecordTypeDeveloperName (String recordTypeId)	Returns record developer name based on its ID. When a DeveloperName cannot be found an empty string '' is returned. For example: ***getRecordTypeDeveloperName ('01220000000jJUm')***

4.5.4 Use UserDAO.

In many places of our logic system looks for an information of the logged user. UserDAO is a class that should be used to get information about logged user. For example:

In before trigger

User currentuser=[Select Country from User where Id=:userinfo.getuserId() LIMIT 1];

And in after trigger:

User currentuser=[Select Division__c from User where Id=:userinfo.getuserId() LIMIT 1];

This is not a proper solution because we are wasting SOQL and we can hit the governor limits. The solution for that is always use:

User theUser = UserDAO.getLoggedUser().

This class stores the information about of all fields related to logged user and queries it during the first usage during transaction. Therefore we are sure that we will spent only one SOQL.

Disclaimer:

The UserDAO class will be implemented during R3.

4.5.5 Use Labels in VisualForce/Frontend, not Constants

When designing a new page, it is important to keep all text in Salesforce Custom Labels. It makes later translations much easier and also allows to modification in production by customer itself. When hardcoding the text, it requires a whole deploy to apply any changes to it.

4.5.6 Utilize seamless Map conversion for Queries

Maps can be very helpful in your coding mission and there is no reason to avoid them.

Especially when Apex offers you such a nice conversion method, immediately creating Id<->SObject pairs from List or Query:

```
Map<Id, sObject> myAwesomeMap = new Map<Id,sObject>(List of sObjects or SOQL Query);
```

4.5.7 Use Custom Metadata or Custom Settings for storing configuration

A common mistake is to create a hardcoded class inside the Apex to handle settings particular application. It is important to consider using (and actually use) either custom settings or

metadata. Custom Settings are especially useful if we want to create different configurations based on hierarchy or we do not have problems with too many queries. They can be reached from Formula fields. Custom Metadata can only be retrieved from Apex, but the retrieving do not count against the SOQL limit. Also, its records are saved as metadata and can be easily deployed between organizations.

It is your choice which one suits your implementation more - just do not create “CasesConfiguration” class in your code.

4.6 Object Oriented approach

New developers should be able to easily read and understand the code and algorithm behind it. I addition to that Apex is an object oriented language and all principles related to object oriented programming should be met.

- Methods should be simple and short. The rule of thumb is when the method does not fit a screen then the method should be spitted into shorted ,less complex methods
- 3 levels of if/ loop is a recommended complexity in for method. For example: when there is an if statement, in a loop, and this loop is placed in another loop that is placed in another if statement the readability of code and algorithm is low and the code should be rewritten or new method should be extracted.
- Reduce a length of the conditional statement.

Sometimes conditions are being checked at the begging of the method and the entire body of the method is being placed in one or more conditional statements. It reduces the code maintainability and readability, for example:

```
Public  Integer someMethod(Integer a, Integer b, Integer c)
   if( b != 0 ){
                 if(a > 0 ){
                    if(b > 0){
                                 if( c > 0) {
              return a/b*c
                       }
                    }

   }
   return 0;
};
```

This kind of code is not readable. Should be changed into:

```
Public  Integer someMethod(Integer a, Integer b, Integer c)
   if( b ==  0 || a <= 0 || c <= 0){
                 return 0
   }
   return a/(b*c);
};
```

- Encapsulation. Good practice is that a method should operate on its local variables and parameters. Introducing class level variable only to communicate between methods is usually a bad practice.

- A method should be responsible for one functionality related to its name. When a method does two separate things it should be spitted to more atomic methods.
- A method name should describe the method functionality. Therefore when the functionality is changed or a new functionally is introduced, the method name also should also be modified to reflect these changes.
- When there is a common group of methods used in many places of the algorithm it is worth to introduce an inner class that encapsulates this functionality.
- Static methods should be used with a special caution because these methods are difficult to mock in unit tests.

4.7 Try not to code

This is the most important advice. In most of the Salesforce organizations, 80% of logic will be achieved with clicks, not code. Before implementing something, thing thrice, if you actually need to put it in the code, when maybe it will be easier and more maintainable, if you implement it with workflow, Process Builder, validation rule, proper profile assignment and similar.

5 Profiles

Recommended design is to limit the number of Profiles and increase the number of Permission Sets.

We will leave only the generic profiles, with no functionality or object access (unless system permission cannot be granted using permission sets):

a. ABB Internal User Profile – used by all standard ABB employees: Sales, Customer Support, Marketing users
b. ABB External User Profile – just to distinguish external users (to be replaced by community profiles)
c. ABB Partner and Customer community Profiles – used by community users (partner & customers)
d. ABB Integration Profile – one common profile used by all integration users with no special permissions, just to limit UI access by setting 'API Enabled' and 'API Only User' permissions. Password should never expire. Integration specific permissions should be granted via permission sets.
e. ABB Basic Admin Profile – non-business users in local administration role (for local user management only)
f. ABB Global Admin Profile (number of those could be extended depending on the needs of S&O team)
g. Standard Salesforce profiles (e.g. System Administrator)

6 Permission sets

Functionality or object access should be driven by Permission Sets. For example: 'Business: Edit Opportunity' Permission Set should include all access required to view and edit Opportunities, enabling any kind of user it's assigned to (regardless of their Profile) to follow the Opportunity Management process.

Before you introduce new permission set, check if existing one covers required level of access.

7 Security

7.1 Securing data model

7.1.1 Data model should be private whenever possible

Data security is always very important factor when developing application. The main concern is that, users should see only subset of data they are allowed to see. Salesforce has advanced permission model, that when configured properly can protect organization data.

As general rule of thumb remember to do not grant more access to user than needed.

7.1.2 Use permission sets for greater flexibility

If you need to grant access among group of user or grant temporary access to some specific part of application permission sets should be used. Remember that user can have many permission sets but only one profile so it's flexible solution to got specialized permissions and maintain them for user. There are many specialized apps on appExchange which can help you to manage easily with permissions among groups of user, for example: The Permissioner, Permission Assignment Rules.

7.1.3 Hide sensitive data with Field Level Security (FLS)

If you need to hide some field for certain profiles or create some non-user/administrative fields the FLS is a way to go for you. Remember that there are some consideration for FLS as workflow can update field restricted by FLS, Roll-Up summary or formula fields can use field restricted by FLS, so read Salesforce documentation carefully to be sure to not provide information to not authorized user by mistake.

7.1.4 Use Apex programing sharing as last resort

It is possible to manually share a record to a user or a group using Apex but before doing so, check twice if your goal cannot be achieved by Salesforce standard sharing possibilities. Programming solution will be always less maintainable as most of the time for additional changes specialized Apex developer will be needed. Standard salesforce sharing feature should be utilized first for better performance and maintenance.

To grant access to records (vertical/horizontal, based on criteria) check possibilities like role hierarchies, sharing rules, manual sharing, public/personal groups, teams, queues.

7.1.5 Enable Modify/View All Data for users with great caution

Consider very careful when granting user with ability to View All Data or Modify All Data.

These privilege should always be enabled only for administrative/knowledgeable users. Granting this for not authorized users can result in information/security leaks as these setting override most of the user profile/sharing access.

7.2 Securing Apex code

7.2.1 Enforcing CRUD and FLS

When rendering VisualForce pages, the platform will automatically enforce CRUD and FLS when the developer references SObjects and SObject fields directly in the VisualForce page but there are often cases where developers use VisualForce to display data derived from an SObject field in an indirect or processed form.

Because VisualForce only sees the return value as a string and not as an SObject field, CRUD and FLS is not automatically enforced and it is necessary to call isAccessible() on the appropriate Describe Field Result in order to manually check the user's CRUD and FLS access.

For more guidance in this topic please refer to official documentation about enforcing CRUD and FLS access and search topics for using isAccessible(), isUpdateable(), isCreateable(), isDeletable() methods.

7.2.2 Be aware of Cross Site Scripting

Cross-site scripting is a vulnerability that occurs when an attacker can insert unauthorized JavaScript, VBScript, HTML, or other active content into a web page viewed by other users. A malicious script inserted into a page in this manner can hijack the user's session, submit unauthorized transactions as the user, steal confidential information, or simply deface the page. Cross-site scripting is one of the most serious and most common attacks against web applications today.

XSS allows malicious users to control the content and code on your site — something only you should be able to do.

7.2.2.1 Unsafe sObject Data Types

Force.com sObjects can be built from a number of primitive data types. When rendering a mergefield or retrieving a field via the API, it's important to understand whether the field contains potentially unsafe or safe content. The following primitive data types can contain unsafe strings:

Primitive Type	Restrictions on values
url	Can contain arbitrary text. The platform will prepend the url with 'http://' if no scheme is provided.
picklist	Can contain arbitrary text, independent of the field definition. Picklist values are not enforced by the schema, and users can modify a picklist value to contain any text via an update call.
text	can contain arbitrary text
textarea	can contain arbitrary text
rich text field	Allows a whitelist of HTML tags, and all other HTML characters are HTML-encoded. Safe to use unencoded in an HTML rendering context but not in any other rendering context (e.g. javascript control characters are not encoded).

Name fields can be arbitrary text, and must be considered unsafe. This also applies to global variables such usernames.

Developers are urged to program defensively. Even if a primitive type (such as an Id) cannot contain control characters, properly output encode the field type based on the rendering context. Output encoding will never result in over encoding and will make your application safe for further refactoring should the controller logic change -- for example, by pulling the Id from a URL parameter rather than from the controller.

7.2.2.2 Built in VisualForce encoding functions

The platform provides the following VisualForce encoding functions:

- JSENCODE -- performs string encoding within a Javascript String context
- HTMLENCODE -- encodes all characters with the appropriate HTML character references so as to avoid interpretation of characters as markup.
- URLENCODE -- performs URI encoding (% style encoding) within a URL component context
- JSINHTMLENCODE -- a convenience method that is equivalent to the composition of HTMLENCODE(JSENCODE(x))

Data may need to be encoded multiple times if it passes through multiple parsers.

As this is very broad topic whenever you need to serialize/deserialize user input or data from external sources please take extra caution whenever and how you will use the data. For more information refer to official documentation on XSS attacks.

7.2.3 Be aware of SOQL Injection

SOQL is much simpler and more limited in functionality than SQL. Therefore, the risks are much lower for SOQL injection than for SQL injection, but the attacks are nearly identical to traditional SQL injection. In summary SQL/SOQL injection involves taking user-supplied input and using those values in a dynamic SOQL query. If the input is not validated, it can include SOQL commands that effectively modify the SOQL statement and trick the application into performing unintended commands.

To prevent a SOQL injection attack, avoid using dynamic SOQL queries. Instead, use static queries and binding variables. If you must use dynamic queries, use the escapeSingleQuotes method to sanitize user-supplied input. This method adds the escape character (\) to all single quotation marks in a string that is passed in from a user. The method ensures that all single quotation marks are treated as enclosing strings, instead of database commands.

7.2.4 Be aware of Cross Site Forgery

Cross-site request forgery (CSRF) occurs when a user visits a malicious web page that makes their browser send requests to your application that the user did not intend. This can be done with the src attribute of the IMG, IFRAME or other tags and more complicated requests, including POSTs, can be made using JavaScript. Because the browser always sends the relevant cookies when making requests, requests like this appear to originate from an authenticated user. The malicious site isn't able to see the results of these requests, but if create, update or delete functionality can be triggered, the malicious site may be able to perform unauthorized actions.

Force.com platform got implemented an anti-CSRF token to prevent attack. Every form includes a parameter containing a random string of characters as a hidden field. When the user submits the form, the platform checks the validity of this string and will not execute the command unless the given value matches the expected value. This feature will protect you when using all of the standard controllers and methods — but only for POST requests. GET requests should be idempotent and are not protected with the anti-forgery token.

To prevent this attack alway use use an <apex:form> block and POST request to update state on server side. It is safe to use getParameters in controller to select and display record but you never should update state of record in this way.

Consider below code as very **BAD PRACTISE:**

…..

```
 // method called with GET method
Id id = ApexPages.currentPage().getParameters().get('id'); // obtain ID from query String
Account obj = [select id, Name FROM Account WHERE id =:id];
delete obj; // update state
```

….

GOOD PRACTICE:

```
// dedicated method
public PageReference deleteAccount() {
  Id id = String.escapeSingleQuotes(ApexPages.currentPage().getParameters().get('id')); // obtain ID from query String
  Account obj = [select id, Name FROM Account WHERE id =:id];
  delete obj;
  return new PageReference(Page.Success_Page);
}
```

7.2.5 Avoid security through obscurity

Term is also known as hiding security secrets behind implementation or components of system.

This kind of security hasn't been proven to work on its own. Consider rather good security by design - system should be designed from the beginning with security in mind. Malicious intent is assumed. User input and often even internals are not trusted. Instead rules are enforced and everything is checked and validated.

7.2.6 Storing Sensitive Data

Sensitive data can include:

- Passwords
- Passphrases
- Encryption keys
- Purchase instruments, such as credit card numbers
- Personal contact information such as names, phone numbers, email addresses, account usernames, physical addresses, and more
- Demographic information such as income, gender, age, ethnicity, education
- In some states and countries: machine identifying information such as MAC address, serial numbers, IP addresses, and more

When storing sensitive information on a machine:

- All authentication secrets must be encrypted when stored on disk. This includes passwords, API Tokens, and OAuth Tokens.
- For client apps running on a desktop, laptop, tablet, or mobile device, store all secrets in the vendor provided key store (keychain in OS X/iOS devices, keystore in Android devices, or in the registry protected with the DP-API on windows devices.) This is a hard requirement to pass the security review.
- For services running on servers that must boot without user interaction, store secrets in a database encrypted with a key not available to the database process. The application

layer should provide the key as needed to the database at runtime or should decrypt/encrypt as needed in its own process space.
- Do not store any cryptographic keys used for protecting secrets in your application code
- Be cautious of the algorithms and ciphers used in any cryptographic operations
- Salt hashes, and if possible store salts and hashes separately
- Leverage strong platform cryptographic solutions
- Check if frameworks/platforms have already addressed the problem
- Use SSL/TLS to transmit sensitive data

API Tokens and OAuth Tokens should not be hardcoded in Apex code, nor in custom labels referenced by that code.

7.3 Use Audit Tools to diagnose potential problems

When dealing with issue on production/sandboxes its good habit to check the audit tools provided by Salesforce to diagnose the root of problem.

If you suspect unauthorized changes to object/application configuration check Setup Audit Trail.

Please note that starting from Winter'16 the SetupAuditTrail object has now been exposed via the Salesforce API's and within Apex via SOQL so you can use sample query to fetch needed information:

List<SetupAuditTrail> stuffDoneByConsultants = [SELECT Id, Action, CreatedBy.Name, CreatedDate, Display, Section FROM SetupAuditTrail WHERE CreatedBy.Email LIKE '%xyzconsulting.com%'];

- For emails problems check emails logs and Deliverability Settings.
- For workflow process and apex errors check Debug Logs.
- For time schedule processes check Paused and Waiting Interviews in Flow section.
- For scheduled jobs problems check Scheduled Jobs in Setup.

Administrators can monitor all login attempts for their organization and enabled portals or communities. The login history page displays the most recent 20,000 attempts. Administrator can download the full information about logins to CSV file.

For more complicated cases use Event Log Files for event monitoring. The event monitoring feature gathers information about your Salesforce org's operational events, which you can use to analyze usage trends and user behavior. You can interact with event monitoring data by querying fields on the EventLogFile object (like EventType and LogDate). You can access log files via https://salesforce-elf.herokuapp.com/

7.4 Backup you data to secure location

Salesforce provide you with tools for exporting data. You can manually start the process or schedule exports with Data Export tool or use Data Loader and schedule exports via API.

It's always good habit to back your data for various reason - simple recovery, audit, archives, and failures. Developers make mistakes and often important files could be deleted. Depending on what files/records are important to you or your client you should store your backups in secure location. Secure password protected location is always better than your local hard drive so have it on your mind.

8 Salesforce mobile Best Practices

Whether you're creating Visualforce pages to add as custom actions or mobile cards, or creating or integrating canvas apps into Salesforce mobile, these principles and practices will help you provide the best experience for your mobile users.

Visualforce pages aren't automatically mobile friendly in the Salesforce mobile app. The standard Salesforce header and sidebar are disabled in favor of the Salesforce mobile controls, and a JavaScript API is available to make it possible for Visualforce pages to connect with Salesforce mobile navigation management. In other respects the pages remain as they are and, although usable within Salesforce mobile, desktop focused Visualforce pages will feel desktop focused.

8.1 Sharing Visualforce Pages Between Mobile and Desktop

You should revise Visualforce pages that appear in both the Salesforce mobile app and in the full Salesforce site to support both environments. This includes Visualforce pages used as custom actions, and Visualforce pages added to standard page layouts, except those added to the Mobile Cards section of a page layout.

Visualforce pages that need to work in both environments include:

- Pages used as custom actions. Custom actions appear in the action bar in the Salesforce mobile app, and in the publisher menu in the full Salesforce site.
- Pages added to normal page layouts, when Available for Salesforce mobile apps and Lightning Pages is enabled for the page.
- Custom Visualforce buttons or links added to normal page layouts.
- Standard button overrides with Visualforce pages for the New, Edit, View, Delete, and Clone actions. Overriding standard list and tab controls isn't supported in Salesforce mobile. Button overrides won't appear in the Salesforce mobile app unless Available for Salesforce mobile apps and Lightning Pages is enabled for the page.

8.2 Visualforce Components and Features to Avoid in Salesforce mobile

Most core Visualforce components (those components in the apex namespace) function normally within Salesforce mobile. Unfortunately, that doesn't mean they're optimized for mobile, or that every feature works with Salesforce mobile. You can improve the Salesforce mobile user experience of your Visualforce pages by following some straightforward rules.

- In general, avoid structural components, like <apex:pageBlock> and child components, and other components that mimic the Salesforce look and feel, such as <apex:pageBlockTable>. If you must use these components, set them to one column, using <apex:pageBlockSection columns="1">, instead of the default of two columns.
- Avoid wide, non-wrapping components, especially <apex:detail>, <apex:enhancedList>, <apex:listViews>, and <apex:relatedList>, which are all unsupported. Keep device width in mind when creating tables with <apex:dataTable>.
- Avoid using <apex:inlineEditSupport>. Inline editing is a user interface pattern that works well for mouse-based desktop apps, but it's difficult to use on a touch-based device, especially on phones where the screen is small.
- Using <apex:inputField> is fine for fields that display as a basic input field, like text, email, and phone numbers, but avoid using it for field types that use an input widget, such as date and lookup fields.
- Don't use <apex:scontrol>. sControls aren't supported anywhere in Salesforce mobile.

- PDF rendering, by setting renderAs="PDF" on <apex:page>, isn't supported for pages in Salesforce mobile.

8.3 Performance Tuning for Visualforce Pages

Performance is an important aspect of mobile Visualforce pages. Visualforce has a caching mechanism to help you tune the performance of your pages. To enable caching for a page, use this statement:

```
<apex:page cache="true" expires="600">
```

9 Integration Best Practices

When you implement Salesforce, you frequently need to integrate it with other applications. Although each integration scenario is unique, there are common requirements and issues that developers must resolve.

Salesforce allows us to consume SOAP services, it has built-in WSDL to Apex class generator.

Apex does not support any other WSDL constructs, types, or services, including:

- RPC/encoded services
- WSDL files with mulitple portTypes, multiple services, or multiple bindings
- WSDL files that import external schemas.

The following data types are only supported when used as call ins, that is, when an external Web service calls an Apex Web service method. These data types are not supported as callouts, that is, when an Apex Web service method calls an external Web service.

- Blob
- Decimal
- enum

Services that you consume from Salesforce and that you create on Salesforce should work on collections of objects. That way they are slightly more complicated, but more scalable.

Calling external system from trigger is not a good practice, this type of integration should be done asynchronously from batch.

When external system wants to update core application objects, with many child objects (for instance Account), it can lead to Unable to lock row exceptions and data skew. The temporary object should be introduced, which volume would be definitely smaller and it won't have so many child objects. Data from temporary object can be processed in asynchronous manner.

9.1 Integration User Best Practices

Service Account (aka. integration user or system user) needs are determined based on a combination of following factors:

9.1.1 Interfaces

Typically separate service accounts used for push and pull services. This will give the ability to grant read only access for pull services and write access only to push services. Adequate care should be taken when sharing a service account across different interfaces as it could potentially hit certain platform governors' limits.

9.1.2 Operational

Separate service accounts could be considered to meet specific operational requirements. This comes down to individual/group that manages, audits and trouble shoots the interface or the

application serviced by the interface. Different service account will be required, if a clear separation of activity/audit trail is desired for different consumers of Salesforce services/API.

9.1.3 Infrastructure

As a best practice distinct service account should be used across different infrastructure that access Salesforce. This is because it yields to operational efficiency as credentials setup/managed on different infrastructure need to kept in sync. Also this provides a clear demarcation of activities and audit trails across different environments/systems.

9.1.4 Systems

Integration data feeds from different backend systems should use separate service accounts. This facilitates operational efficiency as it allows segregating audit and managing different interfaces at source system level.

9.1.5 Authorization

Different service accounts needed depending access level authorized for an interface. User sessions to Salesforce are mapped to a profile and permission sets that authorize access levels, like CRUD operations on objects including access to specific fields. If we need to tightly manage access levels for each interface we need separate accounts. Also record visibility based on user Role drives the number of service accounts required.

9.2 General best practices

9.2.1 Integration user Profile

See Profiles section for details.

9.2.2 Integration user Permission Sets

Each integration user should have a dedicated permission set defined. This permission set should give CRUD access only to required objects & fields based on the nature of this integration. Permission set should be assigned only to one integration user (they should not be shared between integrations).

9.2.3 Impersonating users

In a traditional sense of system or data integration, never implement solutions that impersonate a licensed Salesforce user. Exposing or exchanging an active user’s session Id should be avoided as much as possible. Auditing and troubleshooting becomes a challenge when real users are impersonated.

9.2.4 Record ownership and Data skew

It is important to bear in mind that record ownership plays a part in visibility and security aspect of Salesforce platform. When new records are created Salesforce assumes (unless explicitly assigned) user in context as record owner. In case records created via. integration service account user becomes the owner. It is important to ensure that too many records are not owned by any integration users.

9.2.5 System Fields

Do not share service accounts across different integration end points. Salesforce tracks user who created and last modified every record. When external systems push data into Salesforce the CURD operation performed on records will stamp service account as last accessed user. This serves as an audit trail of how and when records got touched by different integration feeds. This is one of the reason to consider different service accounts for different integration points.

9.2.6 Role hierarchy and record visibility

Be judicious about assign any Role to integration user. If possible avoid assigning a Role. Role assigned to a user drives record visibility to users above in the role hierarchy. This could potentially affect overall performance when an integration user ends up owner of several millions of records and is also placed deep down in a role hierarchy. Data security and visibility for data pushed via. integration needs extensive care.

9.2.7 Compliance Standards

Consider any enterprise guidelines and compliance (E.g. SOX) standards that are to be met when granting access rights to integration user.

9.2.8 Platform Governors Limits

Finally, be cognizant of platform governors' limits that may impact integrations. There are concurrency and query cursor limits to be aware off, particularly when sharing an integration user across different interfaces.

10 Guidelines for Lightning Experience readiness:

As ABB will be migrating Classic interface to Lightning Experience, please make sure that your code is LEX ready:

1. Do not further develop “Account Contact Roles”. Instead, migrate to „Contacts to Multiple Accounts”.
2. Do not further develop “Campaign Influence”. Migrate to Customizable Campaign Influence, which lets you decide how credit is assigned to each campaign that brings in an opportunity.
3. Do not further develop Custom Buttons and Links—JavaScript. They aren't supported in Lightning Experience. Check https://trailhead.salesforce.com/en/modules/lex_javascript_button_migration
4. When you create new Custom Buttons and Links – URLs, test them in Salesforce Lightning Experience. Right now, users can see these buttons and links in Lightning Experience, but the buttons and links might not work as expected. To run your test, preview Lightning Experience from the Migration Assistant in Salesforce Setup. If you notice problems, update your custom URL buttons and links to ones that are supported in Lightning Experience. Check out the resources below to learn more.
5. Do not further develop Home Page and Sidebar Components. Users can't see these components from Lightning Experience Home. Likewise, Lightning Experience doesn't have a sidebar, so users won't see components there, either. Use the Lightning App Builder to customize Lightning Experience Home. As a replacement for the sidebar, you can create utility bars in your Lightning Apps.
6. Do not further use “Case.Email” quick action on case layouts. It isn’t supported in Lightning Experience. Instead, create a Case.SendEmail quick action to send and reply to emails about cases in Lightning Experience.
7. When you create new Visualforce overrides for actions on Salesforce Classic pages, test them in Salesforce Lightning Experience. First, test your existing Visualforce pages in Lightning Experience, and update actions and overrides if they don’t work correctly. For the View action, you can use a Lightning page for the override in Lightning Experience, while retaining the Visualforce page for the override in Salesforce Classic. If your overrides don’t require custom logic, this approach is straightforward, and often very quick. For the List action override, see “Changes With Action Overrides” in the Visualforce and Lightning Experience Trail (https://trailhead.salesforce.com/modules/lex_dev_visualforce/units/lex_dev_visualforce_known_issues). If you need to use Visualforce pages for overrides in both Salesforce Classic and Lightning Experience, consider restyling the pages so that they look like record view and record edit pages in Lightning Experience. Keep in mind that restyling pages can be time consuming. As an alternative, educate your users about styling differences in Lightning Experience.
8. When you decide to use Sharing button on page layout, develop new custom sharing button in Lightning Experience using SOAP API sharing objects like AccountShare or customObject_Share in a Visualforce page, and by creating a Lightning action. This button will be reused in Lightning Experience.
9. Try to use 'Component' type of events rather than 'Application' type.

11 Guidelines for ABB annual organization readiness:

Global picklists will replace local picklists duplicated on several objects. Whenever possible, instead of local picklists use Picklist Value Sets (global picklists). Introduce it especially for masterdata related picklists. If Picklist Value Set does not exist on production, deploy it and reuse it for your object. If similar picklist (local one) is already in use on other objects, there should be separate change request opened to migrate local picklist to global one.

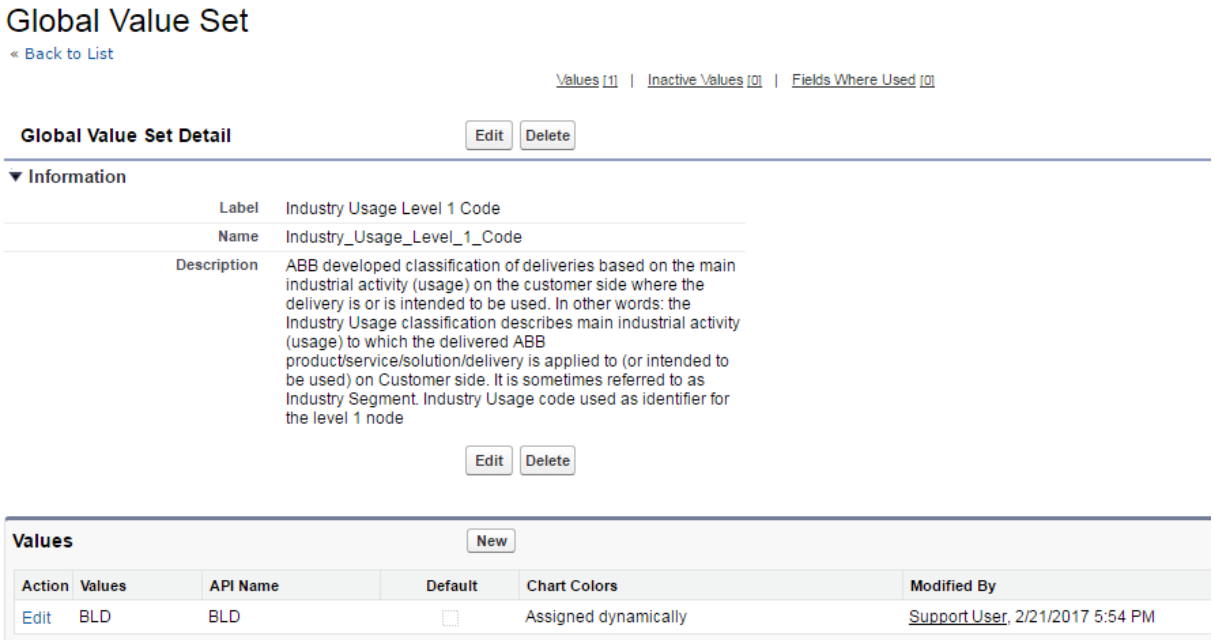
Populate Picklist Value Sets with current values (e.g. taken from active period from masterdata on the date of value set implementation). Those values will be then maintained via automated integration (planned for 2017) or using support channel.

1. Picklist Value Sets sourcing from ABB masterdata should follow masterdata naming convention (see https://metadata.ch.abb.com/ for reference). For example when you are implementing Industry Usage data, please create 3 value sets (we use level 1, 2 and 3 in Salesforce): “Industry Usage Level 1 Code”, “Industry Usage Level 2 Code” and “Industry Usage Level 3 Code”:

2. Every Picklist Value Set should have filled in **description**. For masterdata picklist - description should be taken from https://metadata.ch.abb.com/ site. Concatenate description from main hierarchy level (this description is more meaningful) and lower level. For example when you create „Industry Usage Level 1” value set in Salesforce, use description from Industry Usage (https://metadata.ch.abb.com/DataClass/Details/176):

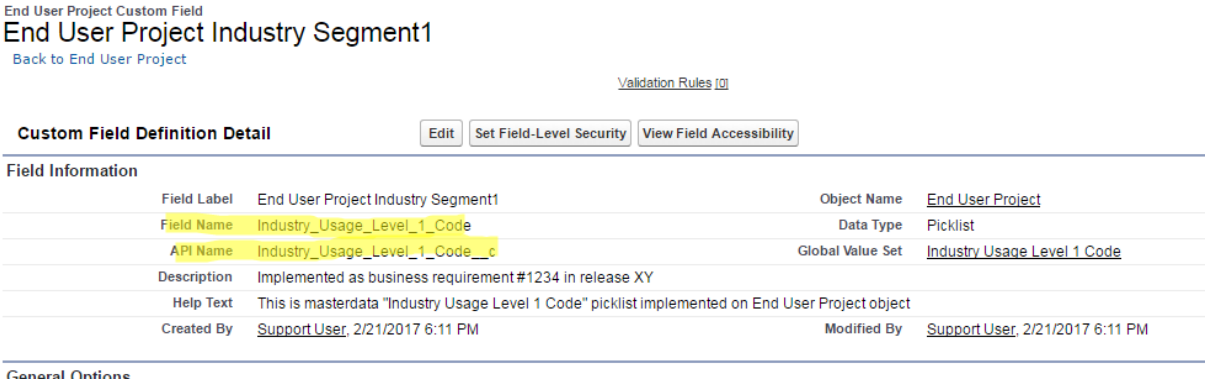
and „Industry Usage Level 1” Code (https://metadata.ch.abb.com/DataElement/Details/177):

Here is result in Salesforce:



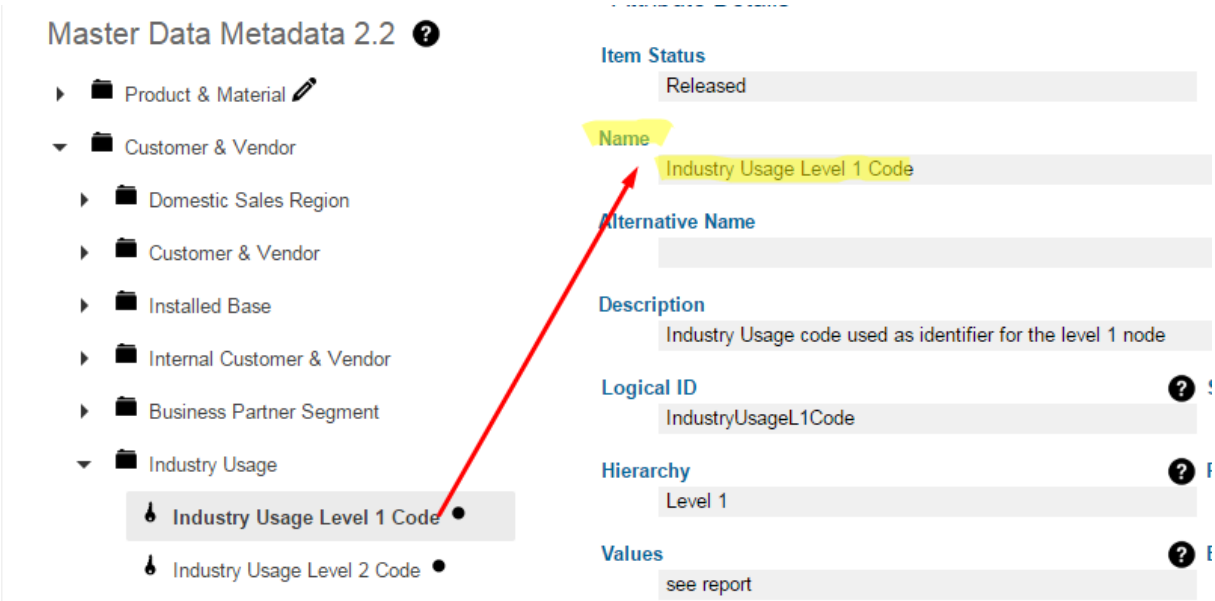
3. Field name (and API name) of picklist created on object level should be same as Name value of picklist from Global Value Set Detail. For example when we have Name = Industry_Usage_Level_1_Code for „Industry Usage Level 1 Code" Global Value Set:

Then Field name and API name on object picklist should follow Global Value Set name:



4. Every picklist implemented on object should have **description** and **help text** filled in. In description please provide reference to requirement id and release in which this field is introduced. In help text - business meaning of new field (this should be provided by business during design session).

5. In case of picklist related to masterdata, Field Label of picklist implemented on object may differ from masterdata (if this is clearly stated in business requirement). If this is not captured in design session, then use Name from https://metadata.ch.abb.com/:



6. Please remember to deliver all language translations for picklist values. If you move from local picklist on an object to picklist sourcing from Global Value Set, then migrate all translations as well.

7. As we will be moving old local picklists to Global Value Sets, every new development should be done based on Global Value Sets and picklists sourcing from those Global Value Sets. Do not create unnecessary technical debt!

12 Naming Convention

The below set of rules is put in place to make the code consistent and easy to maintain. Labels (wherever applicable) should be understandable for end-users (if visible) and as unique as possible (i.e. 'External System Connection' is less probable to be duplicated in the Subscriber Org than 'Connection'). Rules refer to the main part of the component name, excluding the prefix.

1) All names will be as descriptive and as short as possible.
2) Apex Method names will start with a lower-case letter
3) All other components' names will start with a capital letter
   Rule applies to: Custom Objects (including Custom Settings), Custom Fields, Global Value Sets, Picklist Values, Apex Classes, Apex Pages, Triggers, Record Types and Permission Sets
4) Unit Test Classes will have the same name as a Class they cover with 'Test' appended at the end.
5) No spaces, underscores or dashes will be used in the name of any component, unless specified by other rule or enforced by the system (i.e. underscores between the namespaces or double underscore at the end of Custom Object name).
   Spaces in the Labels will be removed, not replaced with other characters (i.e. Long Object Name → LongObjectName)
   Note: Rule does not apply to Validation Rules and Picklist Values label. It applies to Picklist Values developer names.
6) If the name consists of multiple words, each subsequent word will start with a capital letter

All of the created objects in Salesforce have two fields specifying the name - Label and API name. The label is used for display purposes, while the former is referenced in code and formulas.

The rest of this chapter will focus on creating meaningful API names, and words API Name and Name will be used interchangeably.

12.1 Common rules

12.1.1 Start name with [App Name]

Usually, when you code something, it is part of some application, some logic - maybe some integration, maybe a service for a customer, maybe a product. To easily differentiate between the components, it's useful to start all of the objects names with acronym of the application owning the object, e.g. INT_Name__c, SL_Name__c and so on.

12.1.2 Always use English

Even if you are not a native English speaker, use English names for all the objects. The reasons for such behavior our straightforward and simple - it is easier to later outsource the project to your international coworkers or another company if it is in English and also don't lead to confusion about using or not special (specific for a language) signs in the names.

12.2 The convention

12.2.1 Object, Custom Field, Custom Metadata

Convention: [App]_[Object name]__c or [Object name]__c

If the object is not directly connected to any app/project or it will be reused in future projects, you may omit the "App" prefix. ABBSTS is example for Strategic Sales stream.

Example: ABBSTS_DataSource__c, ABBSTS_SalesTypes__mdt, CustomProduct__c

12.2.1.1 Reference fields

Convention: [App]_[Field name]_ref__c
Reference fields (both lookup and master-detail) should be in addition suffixed with "ref".
Example: ABBSTS_Event_Participant_ref__c

12.2.2 Classes

12.2.2.1 Controllers

Convention: [App]_[Name]Controller

Example: AccountExtensionController, ABBSTS_ProjectSpecificCaseController

12.2.2.2 Batches

Convention: [App]_[Name]Batch

Example: ABBSTS_FixAccountTypesBatch

12.2.2.3 Schedules

Convention: [App]_[Name]Scheduler

Example: ABBSTS_FixAccountTypesBatchScheduler

12.2.2.4 Utils

Convention: [Name]Util

Example: DateUtil, StringUtil, CaseUtil

12.2.2.5 Triggers

Convention: [Object]Trigger

Example: AccountTrigger, ContactTrigger

12.2.2.6 Test Classes

Convention: [App]_[ClassName]Test

Example: ABBSTS_ProjectSpecificCaseControllerTest

12.2.3 Page Layout

Convention: [App]_[Object name] [Function name] Layout
Examples: ABBSTS_PricingCaseLayout__c, TSK_PricingCaseCloseLayout__c, INT_PricingCaseForSalesUserLayout__c

12.2.4 Custom Setting

Custom Setting Label should be pluralized in all cases with meaningful name (e.g. Data Sources, not Settings).

12.2.4.1 List Settings

Convention: [App]_[Data entity that each list entry represents]ListSetting__c
Each record represents an individual entry and as such singular naming is applied, as per objects.
Examples: ABBSTS_AnalyticViewListSetting__c, ABBSTS_DataSourceListSetting__c

12.2.4.2 Hierarchy Settings

Convention: [App]_[Function of the settings]Settings__c
Each record represents the same set of settings applied at different levels. In concept this differs from objects and list settings, the plural naming reflects this.
Examples: ABBSTS_OrgBehaviourSettings__c, ABBSTS_MyApplicationSettings__c

12.2.5 Validation Rule

12.2.5.1 Single field

Convention: [App]_[Field Label] [Rule Applied]
Examples: ABBSTS_MailingCityIsRequired__c, TSK_StartDateMustBeaWeekday__c

12.2.5.2 Multiple fields

Convention: [App]_[Field grouping term] [Rule Applied]
Examples: ABBSTS_BillingAddressMustBeComplete__c

12.2.5.3 Cross object

Convention: [App]_[Object Name] [Field Label] [rule applied]
Examples: ABBSTS_OpportunityStageIsClosedNoEditOfOpportunityProducts__c

12.2.6 Action, Button or Link

Convention: [App]_[Verb] [Noun]
Examples: ABBSTS_NewInvoice__c, ABBSTS_UpdateOrder__c, ABBSTS_AlertExecutiveTeam__c

12.2.7 User Profile

Convention: [App]_[Department] [User | Sysadmin]
Examples: ABBSTS_AccountsPayableUser__c, ABBSTS_MarketingExecutiveUser__c, ABBSTS_AcmeSysadmin__c

12.2.8 Permission Set

Convention: [App]_[Feature Area Descriptor] [User Type]
Examples: ABBSTS_Work.comAdministrator__c, ABBSTS_CloudInvoicesUser__c, TSKKnowledge-Contributor__c

Business permission sets should always begin with “Business: “.

Integration permission sets should always begin with “Integration: “ + name of system we are integrating with. E.g.: “Integration: Orbis”.

12.2.9 Public Group/Queue

Convention: [App]_[Grouping term] [[Users] | [Members]]
Examples: ABBSTS_EUUsers__c, ABBSTS_SalesUsers__c, ABBSTS_HRUsers__c, ABBSTS_AMembers__c

13 Resources

For more references, please review following resources:

1) Apex Code Best Practices: https://developer.salesforce.com/page/Apex_Code_Best_Practices.
2) Execution Governors and Limits: https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_gov_limits.htm
3) Developer Best Practices Checklist: https://developer.salesforce.com/blogs/engineering/2015/05/developer-practices-checklist.html
4) Secure Coding Guidelines: https://developer.salesforce.com/page/Secure_Coding_Guideline
5) Salesforce Mobile App Developer Guide: https://developer.salesforce.com/docs/atlas.en-us.Salesforce mobile.meta/Salesforce mobile/dev_best_practices.htm

6) Lightning Components Performance Best Practices: https://developer.salesforce.com/blogs/developer-relations/2017/04/lightning-components-performance-best-practices.html

7) Developer Documentation library: https://developer.salesforce.com/docs/

8) EventLogFile object: https://developer.salesforce.com/docs/atlas.en-us.208.0.object_reference.meta/object_reference/sforce_api_objects_eventlogfile.htm