salesforce

THE CUSTOMER SUCCESS PLATFORM
SALES SERVICE MARKETING COMMUNITY ANALYTICS APPS

Coding Standards

July 2016

Version 1.0

Table of Contents

Whitepaper Series

This document is one of a series of Whitepaper on the Salesforce Software Development Lifecycle. The complete series is:

1. Introduction to Salesforce Software Development Lifecycle
2. Agile development – how to carry out agile development with Salesforce technology, including the basics of scrum
3. Environmental Management – This cover roadmap planning and Sandbox architecture
4. Best Practices – Development, Configuration and coding
5. Release Management – the processes and tools to migrate from Development, Test, UAT, Production, including source control and Continuous delivery
6. Testing Strategy and Tooling – the processes and tools need to carryout comprehensive testing on the Salesforce Platform
7. Citizen Development – discusses the processes to enable Citizen developers within your company
8. Coding Standards – comprehensive set of Salesforce coding standards (this document)

Scope

This document defines best practices for developing "Classic" Salesforce technology, see associated documents for other technology. ("Classic" refers to Salesforce technology that runs on the Force.com platform) for an introduction to the topic please review Development Lifecycle Guide

Executive Summary

Consistent and well-written code is critical to ensuring high quality standards across all applications. The declarative standards will allow for the platform to be self-documented and allow for future applications to build on the existing functionality without the need for extensive rework.
Following the standards outlined in this document is fundamental to the development process, and every member of the technical staff should know and follow them at all times.

- Developers need to read, understand and execute to these standards.
- Project Managers/Team Leaders need to verify that their developers have followed these standards.
- Code that is not delivered to these standards should not be deployed.

Enforcing standards have the following benefits:

- Easily maintainable code.
- Delivery consistency.
- Adherence to best practices that results in improved code performance.

Lightning components are not currently included within this standards document but will be for a later version.

High Level Principles

- Code compiles with no errors or warnings.
- All code is committed to the approved source control. Commits are atomic and frequent, are accompanied by a short description of the change, and are pushed to the remote repository at least daily.
- Naming Conventions are adhered to.
- Never store a username and/or password in code or source control.
- Never hard coded Salesforce IDs or URLs in code.
- All code must have 90% or above code coverage, target higher and by TA exception approval lower code coverage can be accepted
- Always use the latest generally available version of the Salesforce API and if a current class is updated it should be saved in the current API version.
- Follow the code commenting guidelines.
- Code must provide proper Error and Execution Handling.
- Security audits must be performed with appropriate tools.
- All triggers should be bulkified.

Declarative Standards

General Naming Convention

- All descriptions and comments must be written in English.
- We want to ensure that the Org is self-documented. Where description fields are available in the standard configuration it will be required to be completed.
- Aside from loop iterators such as i, j, and k, variable names, object names, class names, and method names should always be descriptive. Single-letter names are not acceptable.
- Class names should use CapitalizedCamelCase. Method or Function names should use lowerCamelCase. Constants should be CAPITALIZED_WITH_UNDERSCORES.
- Underscores should not be used for any variable name, object name, class name or method name except as an application prefix.
- Abbreviations should be avoided for all names due to risk of misinterpretation.
- Overriding standard tabs, objects and standard names is not allowed without CoE Core Management team approval.

- Description fields for all objects, fields, and other metadata should be populated where provided.
- Name must be meaningful. Abbreviations in names must be avoided. Like:

```
computeAverage(); // NOT: compAvg();
```

- Negated variable names must be avoided:

```
boolean isError; // NOT: isNoError
```

- Avoid the use of Static variables wherever possible.
- SFDC Keywords cannot be used as Class variables.
- Project prefixes are required for all Project logical components, e.g. Apex code, Visualforce page, project-specific static resources, Process Builder flows, workflow, etc. A project prefix is a three-character indicator of the project delivering the component followed by underscore. For this document, project prefix is denoted as “lne_”. Project prefixes should be unique to a project, and should be agreed with the overall Org owner. For the initial Live Nation implementation, we are using “lne_”. Subsequent projects and development efforts will need to select other prefixes.

Visualforce Pages

Names should be unique CamelCase and meaningful. The Visualforce description field must be completed. (Note this has to be done via the developer console or setup. Tools like eclipse does not complete the description field.
Requires project prefix:

```
lne_NewsView
lne_NewsCreate
lne_NewsApp
```

Apex Classes

Classes should be unique CamelCase

1. Apex classes that are custom controller classes should be named the same as the Visualforce page suffixed with _CC.
2. Apex classes that are controller extensions should be named the same as the Visualforce page suffixed with _CX.
3. Apex classes that are batch classes should be suffixed with _Batch.
4. Apex classes that are schedulable batch interfaces should be suffixed with _BatchSchedule.

5. Apex classes that are test classes should be named the same as their primary test target suffixed with _Test after all other suffixes.
6. All classes should have a project prefix.

Example:

```
lne_Customer
lne_Customer_CX
lne_Customer_CC
lne_LicenseExpiry_Batch
lne_LicenseExpiry_BatchSchedule
lne_Customer_Test
lne_LicenseExpiry_Batch_Test
```

Apex Methods

Method names should be lowerCamelCase: verb followed by nouns. The name should be without underscores or spaces:

```
lne_handleExpiry
deleteCustomer
getLicense
```

Apex Variables

Variables should be lowerCamelCase. They should be descriptive in name – apart from iterators/counters which can be single letters:

```
Integer x //Only for iterators
Date licenseExpiryDate
Integer customerWarrantyCount
```

Apex Constants

Variables should be uppercase and underscore delimited. They should be descriptive in name:

```
MAX_CHARACTERS
```

Approval Processes

Names should be CamelCase – including the object noun and the triggering event:

```
Ine_ArticlePublishRequest
```

Approval Process Steps

Guidance only - Names should be Capitalised Clear Text with the step outcome and a description:

```
Auto Approved – small amount
Auto Rejected – past expiry date
Approval – send to manager
```

Workflow rules

Names should be Capitalised Clear Text with the event clearly identifiable:

```
Ine_Customer Verified
Ine_Account Authorized
```

Validation Rules

Names should be CamelCase with the Field and Rule clearly identifiable:

```
Ine_DateInFuture
Ine_FirstNameRequired
```

The validation description field must be completed. When the validation rule is updated then the description field needs to be updated.

Page Layout

Best practice to limit number of page layouts and use field level security for accessing fields.

Record Types

Record type label will be business focused and description must add to business value.

Buttons and Links

Standard buttons can be overridden – they will allow for additional users to use the standard functionality. For new custom buttons the labels will be business focused. The description field will be completed.

Objects and Fields

Object and field API names will not be prefixed and will not contain spaces or underscores.

Coding Standards

Coding standards are intended for developers who will be developing code into their Org. It must be noted that these standards are not strictly enforced by the system, rather a means to provide code consistency and adherence to Salesforce best practices and will be verified during the regular project code reviews.
Enforcing standards have the following benefits:

- Easily maintainable code.
- Delivery consistency.
- Adherence to best practices that results in improved code performance.

Apex Classes

```
/*****************************************************************
 Name:  ConfigureOpportunity()
 Copyright © 2016  ITC
============================================================
Purpose:
============================================================
History
-------
VERSION  AUTHOR        DATE          DETAIL          Description
   1.0         Name        dd/mm/yyyy       Created          CSR:
*****************************************************************/
<access> class <name>
{
    //Static Block
     {
          //Static variable
          //Static Methods
     }
    //Non-Static Variables
    ..
    // Constructors
    ..
    // getter & setter methods
    ..
    // Action Methods
    ..
    // Logical Methods

    // Inner Classes and Functions
}
```

Methods

Methods must have a try-catch-finally block to handle possible exceptions. The try-catch block must appear at logical places where you are sure of Exception nature and its handling. Methods must have Debug statements in the beginning and at the end or before the return. This must also include Input and Output parameters to help debugging the code (as shown in the example below). Methods must have block comments to capture the details in format shown:

```
public class ConfigureProducts
{
..
..
/*******************************************************************
   Purpose:  Detailed description of the method
   Parameters: [optional]
   Returns: [optional]
   Throws [Exceptions]: [optional]

History
--------
VERSION  AUTHOR        DATE          DETAIL        Description
   1.0       Name        dd/mm/yyyy     Created           CSR:
 ********************************************************************/
  public String addUDAC(param1, param2)
  {
    system.debug(‘Entering <Method Name>: ‘+ <I/P parameters>);
    …
    system.debug(‘Exiting <Method Name>: ‘+ <return value if any>);
    return ..;
  }
      }
```

Naming convention for the getter and setter methods of Class variables is "get"/"set" followed by the variable name, with the first letter of the name capitalized.
For example, the instance variable "`customerID`", can have the following formats for the getter and setter methods:

```
public string customerID{get; set;}

public string customerID;    // Variable declaration section
public string getCustomerID()
{  //Get method definition
       return <string>;
   }
public string setCustomerID()
{  //set method definition
       return <string>;
   }
```

Static Variable

Never hard code an Id, link etc. in the code directly. Use Custom Labels or static Apex Classes as a mechanism to drive values for a variable.

Workflow Criteria, Validation Rules, and Triggers

All forms of workflow (Process Builder, Flow, workflow rules), validation rules, and triggers must be able to be prevented from running by changing a hierarchical custom settings value (GeneralSettings__c.BypassAutomation__c). This should not be implemented by modifying the User object. The object specific trigger dispatcher will be augmented to check to see if the specified trigger should be bypassed for the currently logged in user. The execute method will not be called if a match is found.

For instance, a validation rule that fires when the Account Name is changed would be written like:

```
!$Setup.GeneralSettings__c.BypassAutomation__c && ISCHANGED(Name)
```

All, or nearly all, validation rule criteria and workflow (Process Builder) criteria will begin with the !$Setup.GeneralSettings__c.BypassAutomation__c or NOT($Setup.GeneralSettings__c.BypassAutomation__c) entry.

Code Blocks

Always use braces to surround code blocks (eg if…else, while, for) and add a description of the functionality of the block.
Never use the pattern:

```
If ([condition])
//do something
else
//do something else
```

Always use:

```
If ([condition])
{
        //do something
}
else
{
        //do something else
}
```

Never leave empty code blocks:

```
If([condition]) {
} else {
//do something
}
```

Class length

Excessive class file lengths are usually indications that the class may be burdened with excessive responsibilities that could be provided by external classes or functions. In breaking these methods apart the code becomes more manageable and ripe for reuse.
Try to reduce the method length by creating helper methods and removing any copy/pasted code.

Method length

Avoid long methods and classes – if necessary refactor the code.
When methods are excessively long this usually indicates that the method is doing more than its name/signature might suggest. They also become challenging for others to digest since excessive scrolling causes readers to lose focus. Best practice is to keep the length to less than 2 screens approximate 60 lines, this enables clear readability and hence maintainability.

Parameter lists

Methods with numerous parameters are a challenge to maintain, especially if most of them share the same datatype. These situations usually denote the need for new objects to wrap the numerous parameters.

Excessive Fields

Classes that have too many fields can become unwieldy and could be redesigned to have fewer fields, possibly through grouping related fields in new objects.
For example, consider:

```
public class Person {  // many separate fields
   int birthYear;
   int birthMonth;
   int birthDate;
   float height;
   float weight;
}
```

Vs:

```
public class Person {  // more manageable
   Date birthDate;
   BodyMeasurements measurements;
}
```

String Literals

Code containing string literals can usually be improved by declaring the String as a constant for ease of maintenance – particularly where it is duplicated. The exception here is for SOQL statements.

String Conversion

Do not convert literals to strings by concatenating them with strings. Use one of the type-specific toString() methods instead:

```
String s = "" + 123;                              // inefficient
String t = Integer.toString(456);                 // preferred approach
```

If Statements

Avoid creating deeply nested if-then statements since they are harder to read and error-prone to maintain:

```
if (x>y) {
      if (y>z) {
        if (z==x) {
         // !! too deep
        }
      }
    }
  }
```

Exceptions

Please refer to “Salesforce Apex Language reference” for details on Exception types, methods and classes as listed below:

- Exception Class.
- Common Exception Methods.
- DMLException and EmailException Methods.
- Bulk DML Exception handling.
- Exception Statements:
    - Throw Statements.
    - Try-Catch-Finally Statements.
- Trigger Exceptions.
- Handling Uncaught Exceptions.

Do not throw raw exceptions. Throw either standard typed or custom exceptions:

```
//define custom exception
public class MyException extends Exception{}

try
{
            throw new MyException();
}
catch ( MyException e )
{
     //MyException handling code here
}
```

Do not throw NullPointerException – developers may misunderstand the source – instead consider throwing an InvalidParameterValueException. Avoid re-throwing an existing exception, or wrapping it in another exception – either don’t catch it, or handle it correctly:

```
try
{
    // do something
}
catch (SomeException se)
{
       throw se;  //Do not do this
                throw new SomeException(se); //nor this
}
```

Avoid throwing exceptions in a finally block as it can hide other exceptions:

```
try
{
            // Here do some stuff
}
catch( Exception e)
{
           // Handling the issue
}
finally
{
            //Avoid doing this
            throw new Exception();
}
```

Don’t needlessly access Exception values unless you’re going to use them. Don’t do this:

```
try {
                        // do something
     } catch (SomeException se) {
                        se.getMessage();
            }
```

Instead use the exception:

```
            try {
                        // do something
            } catch (SomeException se) {
                        ApexPages.addMessages(se);
            }
```

Exception Handlers

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.

Uncaught Exceptions:
There is no standard way to store exception logs in Salesforce.com. By default all uncaught Exceptions will be flagged to the UI and an email with a brief error description will be sent to the developer/user specified in the *LastModifiedBy* field. This way we lose valuable exception log details, which will be required later for the debugging.

Caught Exceptions:
If handled in a try-catch block, the exception logs can be stored in a Custom Object and a meaningful message can be displayed on the UI to end user. This customized error logging process includes following:

- Store error messages and related information in a Custom object. The custom object will store severity, error type, user and other related details.
- Use Workflows to send email notification whenever a user encounters error based on the error severity.
- Cleanup or Archive error data have custom reports created to generate errors based on status and severity.

Errors and exceptions must be handled in a way that provides the user with information regarding what went wrong AND what they should do about it. Whenever possible, this error should also be logged, so the user can provide detailed information regarding the nature of the issue he or she encountered. Exception handlers should trap typed exceptions where possible:

```
Try
{
        //do something here that could throw exception
}
catch (DMLException e)
{
        //log then do something
}
        catch (Exception e)
{
        //log then do something else
}
finally
{
        // cleanup
}
```

Do not return from a finally block as you may lose exceptions in the process:

```
public String foo()
{
        try
        {
                throw new Exception( "My Exception" );
        }
        catch (Exception e)
        {
                throw e;
        }
        finally
        {
                return "A. O. K."; // return not recommended here
        }
}
```

- Exceptions should never be suppressed without a comment in code explaining why.
- Empty exception handlers are never acceptable.
- Exceptions should be logged to a logging object: ApplicationLog__c.
- Bulk log messages should be collated and passed to the Log in one call to prevent excessive DML calls.
- Page based exceptions should be added to ApexPages.Messages.
- In triggers, exceptions should use addError() method to prevent committing.
- Do not catch NullPointerExceptions – identify and resolve the core issue.

ApplicationLog__c

Field	Type	Required	Description
Age	Formula	N	Age of the record in days (used to purge the table)
DebugLevel	Picklist	Y	Error, Info, Warning, Debug
IntegrationPayload	Long Text Area	N	If log is integration related show payload
LogCode	Text (50)	N	Either the exception error code of custom org code for record
Message	Long Text Area	N	Message to log
ReferenceId	Text (18)	N	The related record id
ReferenceInfo	Text (255)	N	The related record info (e.g. Apex Batch Job Id, Contact etc)

Source	Text (150)	Y	The originating class (e.g. CustomerManagement)
SourceFunction	Text (200)	Y	The originating function in the class (e.g. updateDivision())
StackTrace	Long Text Area	N	Raw exception stack trace for unhandled errors
Timer	Number (18,0)	N	The time in milliseconds for the transaction (e.g. For integration/batch apex messages it might be the time taken to process)

Apex Global Variables

Using Global Variables such as $Page, $Action etc allows Salesforce to determine references and prevent deletion of referenced objects.

For example, use:

```
<apex:outputLink value="{!$Page.otherPage}">
```

Instead of:

```
<a href=”/apex/otherPage”>Go here</a>
```

SOQL inside loops

There is a Governor limit to the number of SOQL queries in an execution context. Ensure that SOQL is executed outside a loop as much as possible to prevent this limit being breached. Do not do:

```
for (Account a : [SELECT Id, Name FROM Account
                  WHERE Name LIKE 'Acme%'])
{
   // Your code without DML statements here
         a.Name=’new name’;
         update a;
}
```

Instead do:

```
List<Account> accts : [SELECT Id, Name FROM Account
                       WHERE Name LIKE 'Acme%'];
For(Account a: accts)
{
```

```
a.name='new name';
}
   update accts;
}
```

Similarly, design triggers to support bulk invocation. Do not do:

```
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('do something');

                }
  }

}
```

Instead do:

```
trigger contactTest on Contact (before insert, before update)
{

  Set<Id> accountIds = new Set<Id>();
  for(Contact ct: Trigger.new)
        {
          accountIds.add(ct.AccountId);
  }
  //Do SOQL Query outside the iterator
        Map<Id, Account> accounts = new Map<Id, Account>(
                [select id, name, billingState from Account where id in
                        :accountIds]);

        for(Contact ct: Trigger.new)
        {
        if(accounts.get(ct.AccountId).BillingState=='CA')
                {
                        System.debug('do something');
            }
    }
}
```

Querying Large Data sets

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. Instead of:

```
Account[] accts = [SELECT id FROM account];
For (Account acct : accts)
{
            //Do Something
}
```

Use:

```
for (List<Account> acct : [SELECT id, name FROM account])
{
            //Do Something
}
```

Code Convention

General Standards

- Declare all variables at the beginning of each class.
- All instance variables must be declared either private or protected.
- A class must 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. Generally, consider the scope of the variable and minimize where possible.
- All constants must have uppercase names, with logical sections of the name separated by underscores.
- Local variables must not have the same name as instance variables. In general, variables in a nested scope must 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.

Comments

Apex Classes and Visualforce pages should have a comment block at the top to specify high level information about the asset. This includes: brief description of its intention, author and development chain in ApexDoc format.

For example:

```
<!-- /**
 * @author Paul Roth <proth@salesforce.com>
 * @created 2016-09-01
 * @group VisualReporting (this is used to group assets together)
 * @description: Base container page that provides block level reporting - US121
**/ -->
```

Internal Apex Class members (such as methods, properties, etc) should also have apexDoc style comments including: @description, @param, @returns, etc.

What is ApexDoc: Model after JavaDoc documentation generator, ApexDoc generates document for Salesforce Apex classes. You tell ApexDoc where your class files are, and it will generate a set of static HTML pages that fully document each class, including its properties and methods. For details:

https://github.com/SalesforceFoundation/ApexDoc/blob/master/README.md

Benefits: spend 15 mins each day and a nice set of documents for all Apex classes will be produced painlessly at the end.

Comments in code are important because they enable other developers to quickly comprehend the logic and intent of unfamiliar code. Avoid sentence fragments. Start sentences with a properly capitalized word, and end them with punctuation.

All source code files (of all languages and/or resource types) should have a comment block at the top that specifies the name of the class (or interface, etc.), what it does, when it was created, and who created it. If you make significant changes to a file that someone else originally wrote, add yourself to the author line.

Apex Doc follows JavaDoc synatx, here are a few suggestion/tips:
On class level, we should have a comment block that include: @author, @date, @group (see suggestion in next slide), @description (we can skip @group-content).

On property level, we should have @description attribute. Special tokens are called out with @token.

All methods should have comments explaining what they do, including information about the input parameters and return data (if applicable). Comments should be added within methods whenever the logic or functionality of a section of code is not immediately obvious. On method level, we should have @description, @param, @return (@example is optional)

Block comments are used to provide descriptions of files, methods, data structures and algorithms. Block comments inside a function or method must be indented to the same level as the code they describe. Remember all code, Apex, Visualforce, JavaScript, CSS should have comments.

Block comment for data structures and algorithms:

```
/* Start Version: X.X
  * Block comments with details of changes
  * ..
  * ..
  */
..
..
/* End Version: X.X */
```

Single line short comments can appear on a single line indented to the level of the code that follows:

```
if (condition) {
    // Handle the condition.
    ...
}
```

Very short comments can appear on the same line as the code they describe:

```
if (a == 2) {
    return TRUE;  // special case
} else {
    return isPrime(a); // works only for odd a
}
```

Apex

```
Integer i = 1; // This comment is ignored by the parser
Integer i = 1; /* This comment can wrap over multiple
                  lines without getting interpreted by the
                  parser. */
```

VisualForce

```
<apex:page standardController="Contact">
<!-- Here is a comment -->
  <apex:sectionHeader title="Contact Edit" subtitle="New SFDC99 Member" />
```

Javascript

```
/**
 * Returns an object as a JSON string
 *
 * @method toJSON
 * @return {Object} Copy of ...
 */
toJSON: function () {
```

CSS

```
p {
    color: red;
    /* This is a single-line comment */
    text-align: center;
}

/* This is a multi-line
Comment */
```

Indentation

- Use Tabs (4 spaces) for indentation. Set your editor tab to 4 spaces.
- Avoid lines longer than 80 characters, since they're not handled well by many terminals and tools.
- Split lines occurs when a statement exceed the 80 column limit given above. It is difficult to give rigid rules for how lines should be split, but the examples above should give a general hint.
- In general:
    - Break after a comma.
    - Break after an operator.
    - Align the new line with the beginning of the expression on the previous line.

```
//DON'T USE THIS INDENTATION
longName1 = longName2 * (longName3 + longName4
            -longName5) + 4 * longname6;

//USE THIS INDENTATION INSTEAD
longName1 = longName2 * (longName3 + longName4 - longName5)
            + 4 * longname6;
```

```
//DON'T USE THIS INDENTATION
if ((condition1 && condition2)
    || (condition3 && condition4)
    ||!(condition5 && condition6)) { //BAD WRAPS doSomethingAboutIt();        }
//MAKE THIS LINE EASY TO MISS

//USE THIS INDENTATION INSTEAD
if ((condition1 && condition2)
        || (condition3 && condition4)
        ||!(condition5 && condition6)) {
    doSomethingAboutIt();
}
```

```
//OR USE THIS
if ((condition1 && condition2) || (condition3 && condition4)
        ||!(condition5 && condition6)) {
    doSomethingAboutIt();
}
```

Unit Test Coverage

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 either the @isTest annotation on the class, or the testMethod keyword in the method. Either:

```
@isTest
Public class myClass_Test{
   // Methods for testing
   @isTest static void test1() {
      // Implement test code
   }
}
```

Or:

```
public class myClass_Test {
   static testMethod void test1 () {
         // Implement test code
   }
}
```

The following need to be adhered to when writing test code:

- Cover as many lines of code as possible.
- You must have 90% of your Apex scripts covered by unit tests to deploy your scripts to pre prod and production environments. Exception to be approved by TA. In addition, all triggers must have unit test coverage.
- Calls to System.debug are not counted as part of Apex code coverage in unit tests.
- In the case of conditional logic (including ternary operators), execute each branch of code logic.
- Make calls to methods using both valid and invalid inputs.
- Complete successfully without throwing any exceptions, unless those errors are expected and caught in a try…catch block.
- Always handle all exceptions that are caught, instead of merely catching the exceptions.

- Use System.assert methods to prove that code behaves properly.
- Use the runAs method to test your application in different user contexts.
- Use the isTest annotation. Classes defined with the isTest annotation do not count against your organization limit of 1 MB for all Apex scripts.
- Exercise bulk trigger functionality—use at least 20 records in your tests.
- Write comments stating not only what is supposed to be tested, but the assumptions the tester made about the data, the expected outcome, and so on.
- Test the classes in your application individually. Never test your entire application in a single test.
- Debug statements and Comments are not included in the Code coverage.
- User records are not covered by test methods.
- Test methods and business methods must not be included inside the same class
- There must always be separate methods for parsing the response of a webservice call. This way it will help in writing test methods for code coverage and debugging the code when required.
- Avoid using @SeeAllData unless required via the API (eg ConnectAPI).

JavaScript

Use of third party JavaScript frameworks such as jQuery are acceptable however their use should be considered an architecturally significant inclusion for agreement:

- Load libraries from Salesforce static resources not from external CDNs.
- Use NoConflict modes as available to avoid overriding $.
- Use shared static resources as libraries for repeated code.
- If possible, load JavaScript libraries at end of page.
- User-Agent sniffing should be avoided – instead consider using Modernizr for feature detection.
- Avoid excessive chaining of methods as they make reading code overly complex.

```
$($(event.target).parents('.wrapper')[0]).find('.record').find(".customerid").innerT
ext().toLowerCase()
```

Avoid inlining javascript – use binding handlers eg:

```
jQuery.bind();
document.addEventListener()
```

Triggers

Name of a Trigger can consist of alphanumeric characters. It should be unique, beginning with an uppercase letter and start with the <Object Name>. It should not contain underscores and spaces apart from the Application Prefix. The words should be concatenated with Initial uppercase.
One trigger per object
Logic-less Triggers
Trigger Framework with

https://developer.salesforce.com/page/Trigger_Frameworks_and_Apex_Trigger_Best_Practices

http://developer.force.com/cookbook/recipe/trigger-pattern-for-tidy-streamlined-bulkified-triggers

• Re-entrant ready, controlled recursive
• API bypass
• Global on/off Switch

There should only be one trigger per action on an object. The trigger name must be simply the object name and the action. Use trigger handlers to link multiple logical triggers to an event:

- Create a single trigger per object. Since Trigger execution order is impossible to predict or control, the best practice is that one trigger per object is best practice. The single Trigger will use Trigger Context Variables to execute delegate class utility methods that contain the desired logic. This also create must better maintainable code.
- All triggers must be bulkified and test classes must test the bulkification – regardless of a project's expectation for bulk invocation.
- In addition it must be possible to turn off the complete trigger via a custom setting and static variable.
- It must be possible to stop component parts of project functionality in a trigger via a custom setting and static variable.
- User-defined methods in a trigger cannot be declared as static.
- 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.
- You can only set five savepoints in all contexts.

- 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.
- If a Trigger is required to make an external call it must be done asynchronously using *@future* .
- For complex triggers suggest you develop a trigger handler framework, see SFDC trigger framework as an example.

@future

To run Apex asynchronously, use the @future annotation when defining the Apex method. By default, Apex runs synchronously in a single transaction. The key benefit of running the Apex asynchronously is to remove the business logic from the synchronous transaction and therefore the user won't have to wait for the processing. In other words, offloading the business logic processing to the async Apex, there is less logic to process synchronously and the user experience is more efficient.

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 10 @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.

Cyclomatic complexity[1]

Cyclomatic complexity is a measure of the potential code routes through a piece of code. A high value makes testing and maintenance much harder. Use a tool (such as SonarQube) to measure cyclomatic complexity and refactor if the measure approaches or exceeds 200.

[1] https://en.wikipedia.org/wiki/Cyclomatic_complexity

Visualforce Pages / Components

- **Label and Name**: The Label and Name should be identical. They should start with the project prefix, underscore, and then a meaningful name. The words should be concatenated with Initial uppercase.
    - o Example: “Ine_ConfigureOpportunity”.
- **Description**: Provide a meaningful description of the Visualforce page.

Page Layout

```
<apex:page controller="DependentObjects" id="UseCaseDisplay"
label="FeatureCategoryReport" >
<!--=========================================================-->
<!--Name:  ConfigureOpportunity()                           -->
<!-- Copyright © 2016  ITC                                  -->
<!--=========================================================-->
<!--=========================================================-->
<!-- Purpose:                                                -->
<!---------                                                  -->
<!--=========================================================-->
<!--=========================================================-->
<!-- History                                                 -->
<!-- -------                                                 -->
<!-- VERSION AUTHOR   DATE       DETAIL  RELEASE/CSR         -->
<!--   1.0 -        Name  dd/mm/yyyy    Created              -->
<!--                                                         -->
<!--=========================================================-->
<apex:form>
    <apex:pageBlock title="Feature Selection" mode="edit" id="thePageBlock">
        <apex:pageBlockSection columns="1">
            <apex:pageblockSectionItem>
                <apex:outputLabel value="Feature Category:" for="category"/>
                    <apex:selectList value="{!category}" size="1" id="category">
                        <apex:selectOptions value="{!categories}"/>
                    </apex:selectList>
            </apex:pageblockSectionItem>
        </apex:pageBlockSection>
    </apex:pageBlock>
  </apex:form>
</apex:page>
```

General Convention

- Make sure that each Page items(all visualforce tags) has ID and the IDs should be unique.
- Pay attention to the comments and remarks and put them in legible form before every Page item.
- You can use global variables to reference general information about the current user and your organization on a Visualforce page. All global variables must be included in expression syntax, for example, *{!$User.Name}.*
- The *<apex: includeScript>* Visualforce component allows you to include a custom script on the page. In these cases, careful measures need to be taken to validate that the content is safe and does not include user-supplied data.
- The alignments have to be properly maintained with a tab of 4 spaces.
- It is recommended to have partial rendering of a page item then a full page.
- By default all pages will use standard Salesforce/Lightning look and feel. Any customization in this must be discussed across the scrum to maintain uniform page layouts in all modules.
- Note: If performance of the page load is an issue, complex code documentation may be an issue, however it is important to have documentation.

Code security and Safety

Salesforce has incorporated several security defences into the platform itself. However, careless developers can still bypass the built-in defences in many cases and expose their applications and customers to security risks. Many of the coding mistakes a developer can make on the platform are similar to general Web application security vulnerabilities, while others are unique to Apex.

Security audits for Apex code should be done with the Checkmarx tool at http://security.force.com/. Running Checkmarx regularly should be an integral part of every sprint to identify issues.

Standard Visualforce components

All standard Visualforce components, which start with `<apex>`, have anti-XSS filters in place. For example, the following code is normally vulnerable to an XSS attack because it takes user-supplied input and outputs it directly back to the user, but the `<apex:outputText>` tag is XSS-safe. All characters that appear to be HTML tags are converted to their literal form. For example, the < character is converted to `&lt;` so that a literal < displays on the user's screen.

```
<apex:outputText>
   {!$CurrentPage.parameters.userInput}
```

```
</apex:outputText>
```

By default, virtually all Visualforce tags escape the XSS-vulnerable characters. It is possible to disable this behaviour by setting the optional attribute `escape="false"` . This default behaviour must not be disabled without ITC Core Management Team and Security Approval.

Programming Items Not Protected from XSS

The following items do not have built-in XSS protections, so take extra care when using these tags and objects. This is because these items were intended to allow the developer to customize the page by inserting script commands. It does not makes sense to include anti-XSS filters on commands that are intentionally added to a page.

Custom JavaScript

If you write your own JavaScript, the platform has no way to protect you. For example, the following code is vulnerable to XSS if used in JavaScript. Care should be taken when using JavaScript, and code must be subject to rigorous QA where such methods are used.

```
<script>
var foo = location.search;
   document.write(foo);
</script>
```

<apex:includeScript>

The `<apex:includeScript>` Visualforce component allows you to include a custom script on the page. In these cases, ensure the content is safe and does not include user-supplied data. For example, the following snippet is extremely vulnerable because it includes user-supplied input as the value of the script text. The value provided by the tag is a URL to the JavaScript to include. If an attacker can supply arbitrary data to this parameter, they can potentially direct the victim to include any JavaScript file from any other website.

```
<apex:includeScript value="{!$CurrentPage.parameters.userInput}" />
```

Formula Tags

The general syntax of these tags is: `{!FUNCTION()}` or `{!$OBJECT.ATTRIBUTE}`. Formula expressions can be function calls or include information about platform objects, a user's environment, system environment, and the request environment. An important feature of these expressions is that data is not escaped during rendering. Since expressions are rendered on the server, it is not possible to escape rendered data on the client using JavaScript or other client-side technology. This can lead to potentially dangerous situations if the formula expression references non-system data (that is potentially hostile or editable data) and the expression itself is not wrapped in a function to escape the output during rendering.

The standard mechanism to do server-side escaping is through the use of the `SUBSTITUTE()` formula tag.

Formula tags can also be used to include platform object data. Although the data is taken directly from the user's organization, it must still be escaped before use to prevent users from executing code in the context of other users (potentially those with higher privilege levels). While these types of attacks must be performed by users within the same organization, they undermine the organization's user roles and reduce the integrity of auditing records. Additionally, many organizations contain data which has been imported from external sources and might not have been screened for malicious content.

Cross-Site Request Forgery (CSRF)

Cross-Site Request Forgery (CSRF) flaws are less of a programming mistake as they are a lack of a defence. The easiest way to describe CSRF is to provide a very simple example. An attacker has a Web page at www.attacker.com. This could be any Web page, including one that provides valuable services or information that drives traffic to that site.
If the user is still logged into your Web page when they visit the attacker's Web page, the URL is retrieved and the actions performed. This attack succeeds because the user is still authenticated to your Web page.

Within the Salesforce platform, Salesforce has implemented an anti-CSRF token to prevent this attack. Every page includes a random string of characters as a hidden form field. Upon the next page load, the application checks the validity of this string of characters and does not execute the command unless the value matches the expected value. This feature protects you when using all of the standard controllers and methods.

Developers must not bypass these built-in defences, either deliberately or inadvertently. Developers should be cautious about writing pages that take action based upon a user-supplied parameter.

SOQL Injection

In other programming languages, the previous flaw is known as SQL injection. Apex does not use SQL, but uses its own database query language, SOQL. 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, dynamic SOQL queries should not be used. Instead, use static queries and binding variables.

If you must use dynamic SOQL, you must 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.

Data Access Control

The Salesforce platform makes extensive use of data sharing rules. Each object has permissions and may have sharing settings for which users can read, create, edit, and delete. These settings are enforced when using all standard controllers.

When using an Apex class, the built-in user permissions and field-level security restrictions are not respected during execution. The default behaviour is that an Apex class has the ability to read and update all data within the organization. Because these rules are not enforced, developers who use Apex must take care that they do not inadvertently expose sensitive data that would normally be hidden from users by user permissions, field-level security, or organization-wide defaults. This is particularly true for Visualforce pages.

```
public class customController {
    public void read() {
        Contact contact = [SELECT id FROM Contact WHERE Name = :value];
    }
}
```

In this case, all contact records are searched, even if the user currently logged in would not normally have permission to view these records. The solution must use the qualifying keywords `with sharing` when declaring the class:

```
public with sharing class customController {
    . . .
}
```

The `with sharing` keyword directs the platform to use the security sharing permissions of the user currently logged in, rather than granting full access to all records.

Alternatively, you may use the isAccessible, isCreateable, or isUpdateable methods of Schema.DescribeSObjectResult to verify whether the current user has read, create, or update access to a sObject, respectively. Similarly, Schema.DescribeFieldResult exposes these access control methods that you can call to check the current user's read, create, or update access for a field. In addition, you can call the isDeletable method provided by Schema.DescribeSObjectResult to check if the current user has permission to delete a specific sObject.

```
if (Schema.sObjectType.Contact.fields.Email.isUpdateable()) {
   // Update contact phone number
}
```

Development Processes

- All code should be held within a Source code management system such as GIT.
- All code should be peer reviewed by a senior developer.
- All commits need comments including User Story ID and description of the work performed.

Reference Documentation

The following is a list of links where additional information on the subjects discussed in this document can be found:

- Secure Coding Guideline.
- Open Web Application Security Project.
- Visualforce Performance: Best Practices.
- Apex Code Best Practices.
- Web Page Speed Insights.
- Developer Best Practices Checklist.
- Force.com Apex Code Developer's Guide.

About the Author

James Burns is a Senior Director – Solution Architect at Salesforce and is the Global SME (Subject Matter Expert) on Salesforce Governance and works with customers on building their governance frameworks globally. (This document was based on work by Suchin Rengan and Felix Lindsay).