Salesforce Coding Standards

	Prepared By / Last Updated By	Reviewed By	Approved By
Name	SFDC Team	SFDC COE	SFDC BU
Role	Velumani Angappan (482043) Basusri Majumdar (136830) Sourav Hazra (130878)	Saibal Dutta (355375) Debopom Pathak (136966)	Brotin Kumar Chakraborty (103490)
Signature			
Date	23/03/2023	29/03/2023	30/03/2023

Change Log	Date
Updates • Revamped Governor Limit section as per latest SFDC standards and included API limits as well • Face-lift of General naming convention section and make it more readable and understandable with examples • Update of Apex Unit test section to include latest SFDC recommendations related to Assert class and Test Factories • Update of Coding standards including Apex best practices, Bulk data processing, Code structure and organization etc. based on latest SFDC updates • Removal of Obsolete & irrelevant sections like workflows etc. based on latest SFDC updates • Team has gone through the QRST file & audit report, and ensure to incorporate the findings in the document **New** • Automated code quality checking using various tools like Cognizant Code Sense & SFDC Optimizer • Most Common Audit and code scanner errors • LWC best coding practices • Integration • Bulk data processing using bulk API 2.0	SFDC COE

Table of Contents

1.0 Executive Summary

Consistent and well-written code is critical to ensuring high quality standards across all applications. The declarative standards will allow the platform to be self-documented and allow 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 team should always know and follow them.

- 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 **must not be deployed**.

Must read for all Developers :

The document also provides a repository of **Most Common Coding Errors** observed in multiple projects across the globe. Project Manager/Tech leads should refer to section 12 to ensure improved code quality. This will resolve majority of the code quality issues being reported in multiple projects.

If there are any questions/queries/suggestions about the standards and their applicability, then please contact EAS SFDC CoE (Cognizant) EASSFDCCoE@cognizant.com.

Disclaimer :

This document is intended for internal use only and shall not be distributed outside of Cognizant network without the prior written consent of Salesforce COE Team. This document also does not suffice for any external audit

2.0 High Level Principles

The Platform Owner retains ultimate control of the standards and may overrule this document on request in specific cases. This can be requested by projects through Customer Salesforce IT team for the Salesforce platform.

- Code compiles with no errors or warnings.
- All code is committed to approve source control. Commits are atomic and frequent, also 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 code Salesforce IDs or URLs or profile/permission names in code.
- UI Screen should be configurable. The number of columns to display should be configurable and not hard coded.
- All code must have 90% or above code coverage using an approved test framework. Best practice for this is 95% but this has been slightly relaxed on case-to-case based on mutual agreement with customer IT team.
- Always use the latest generally available version of the Salesforce API.
- Follow the code commenting guidelines.
- Code must provide proper Error and Execution Handling as advised below
- Security audits should be performed with appropriate tools (Code Sense or CheckMarx).
- Code quality should be checked with appropriate tools like Code Sense, CheckMarx
- All triggers should be bulkified.

2.1 Choose the right tools to Customize and Extend Salesforce CRM

There are several ways to adapt your application, so that it meets your business needs exactly. Each approach requires different technologies—and different skills. Here are the three main approaches:

1. Out-of-the-box configuration
2. Pre-developed enhancements and applications
3. Custom enhancements and applications

Before going to detail around these three approaches, it is very important to understand the difference between declarative and programmatic customization in Salesforce

What is Declarative and Programmatic Customization in Salesforce?

- Declarative customizations can be made point and click in a browser.
- Programmatic customizations are made with code.

Trade-Off between Declarative and Programmatic Customization

Declarative	Programmatic
Pros: ▪ It's Simple ▪ It's Speedy ▪ Ease to Development ▪ Ease to Upgrade ▪ Ease to Maintain ▪ Does not require Programmatic Skills ▪ Can take advantage of new features introduced in 3 major releases per year	Pros: ▪ More Control over the functionality ▪ More Flexibility ▪ Can extend the capability of an application beyond standard functionality
Cons: ▪ Less flexibility – Cannot extend the capabilities of an application beyond standard functionality. ▪ Less Control	Cons: ▪ Required Maintenance effort compare to declarative customization ▪ More Development effort compare to declarative customization ▪ Required Programmatic Skills ▪ Cannot take advantage of 3 major salesforce release per year

2.1.1 Approach # 1: Use out-of-the-box configuration

Even non-technical users (with the appropriate permissions) can personalize the Salesforce CRM application, either for their own use or for everyone in the organization. We call this approach “clicks, not code” because you can use it to configure various application building blocks—including the data model, the user interface, and the underlying business logic—without programming. For example, you can customize user interface components or tools to automate your business processes. And our security architecture already includes profiles with permissions, field-level security, sharing rules, and a role hierarchy.

With these tools, you can quickly respond to changing business requirements without technical

skills that may be in short supply. In general, we recommend using the configuration options before creating custom code. Even experienced programmers can work faster than when coding from scratch.

Below are some tips to keep your configuration manageable over time:

1. Out-of-the-box configuration – Tips to keep your configuration manageable over time

- ✓ Limit the number of profiles – By setting up different profiles for various user groups such as end users, support users, and administrators, customer can efficiently control user access. However, supplier recommends customer limit the number of profiles that must be maintained.
- ✓ Streamline your role Hierarchy - Keep nesting below 10 levels. Simplify sales branch if using territory management and avoid skewed data configurations
- ✓ Limit the number of sharing rules – Use sharing rules to make automatic access exceptions for specific groups defined by customer organization-wide defaults. Supplier recommends keeping sharing rules to a minimum to avoid ongoing maintenance.
- ✓ Limit the number of custom fields – Use custom fields to capture key information for your business. Supplier highly recommends customer carefully evaluate the need for any given field and its benefits to various groups.
- ✓ Limit the number of page layouts – We can have upto 15 page layouts per object in Salesforce . Page layouts can quickly grow out of control if not managed properly. Page layouts are useful for standardizing business across business units, but the more layouts, the more work it is to respond to change requests.

2.1.2 Approach #2: Use pre-developed enhancements and applications

In addition to configuring Salesforce CRM, you can extend the application with hundreds of tools and Custom applications available from the AppExchange, SFDC's online marketplace for cloud-based applications. For example, Force.com Labs created free dashboards—for sales, marketing, adoption, and service & support—that provide real-time snapshots of your business.

These apps were built by salesforce.com, individual developers, and salesforce.com partners. All are pre-integrated with the Force.com platform and certified to be secure. Some of these apps are free; others are available for a fee. Users can use the AppExchange to browse for, test-drive, and install applications with no or minimal effort. In essence, it's an easy way to get features to enhance your application or get new applications without having to build them. Partners can use the AppExchange to make the apps they develop available to thousands of salesforce.com customers.

Advantages of using the AppExchange

- ➢ Get missing functionality – Get features to enhance your application—or get new applications without having to build them. Most apps are ready to go, although some might need a little tweaking.

- Extend the business benefits of Salesforce CRM to more of your departments and employees – Take advantage of specialized applications developed by the “long tail” of the software development world to address your business needs beyond sales and customer service.
- Leverage the integration power of native Internet technologies – Partners can use Web services and service-oriented architectures (SOAs) to deliver virtual application suites for various businesses requirements across industries. Such suites offer better modularity and easier integration than traditional suites.

2.1.3 Approach #3: Develop enhancements and applications

Developers can use the Force.com platform to develop, package, and instantly deploy applications, without any infrastructure. If you need an application that isn’t already on the AppExchange—or if you want to customize beyond the capabilities of the configuration tools—the development tools on the Force.com platform are right for you.

2.2 General Audit Report Issues

An audit report related to coding standards will typically include findings related to compliance with established coding practices, procedures, and guidelines. These findings may be based on a review of source code, documentation, and discussion with software development personnel.

The audit report identifies specific issues related to coding standards, such as inconsistent naming conventions, improper use of variables or data types, lack of proper documentation, or failure to adhere to established coding practices. The report also includes recommendations for improving compliance with coding standards, such as providing additional training for developers, implementing automated code analysis tools, or developing a more comprehensive coding standards document that is aligned with industry best practices and meet the needs of the organization.

Refer to the section Issues mentioned in audit reports to know the most common audit findings

3.0 Governor Limits

Salesforce Governor Limits are usage caps enforced by Salesforce to ensure efficient processing. They allow for multiple users of the platform without impeding performance.

Governor Limits are a concept that all Salesforce Developers (and Salesforce Admins to a lesser extent) must grasp. They will fundamentally shape the way that you architect Salesforce solutions, as well as how the code is written.

If a Salesforce Governor Limit is hit, the associated governor issues a runtime exception that cannot be handled. In other words, your code breaks and won't work.

Types of Salesforce Governor Limits

- **Per-Transaction Apex Limits:** These limits count for each Apex transaction. For Batch Apex, these limits are reset for each execution of a batch of records in the execute method.
- **Per Transaction Certified Managed Package Limits:** If a managed package developed by a Salesforce ISV has passed security review, they are provided with generally higher per-transaction limits.
- **Lightning Platform Apex Limit:** These limits aren't specific to an Apex Transaction and are enforced by the Lightning platform.
- **Static Apex Limit:** Apex Limits that are applied across all transactions.
- **Size-Specific Apex Limit:** Apex Limits related to the size of code.
- **API Limits:** These limits and allocations apply to Salesforce Platform SOAP and REST APIs and any other API built on those frameworks, unless noted otherwise.

To maintain optimum performance and ensure that the Lightning Platform API is available to all our customers, Salesforce balances transaction loads by imposing three types of limits:

- Concurrent API Request Limits
- API Timeout Limits
- Total API Request Allocations
- When a call exceeds a request limit, an error is returned.

For more details, please visit

1. Salesforce documentation for API Limits
2. Salesforce documentation for Governor Limits

4.0 Declarative Standards

4.1 Enterprise Objects vs. Project Objects

In general, metadata entities will either be of Enterprise interest or of Project interest. The Data Architect will determine which entities are of which type during early engagement by the project and by subsequent design reviews.

- Creation and/or changes to Project Objects are permitted by the project team.
- The Data Architect may promote Project Objects to Enterprise Objects
- Changes may not be made to Enterprise Objects without the agreement of the Data Architect.
- All Standard Objects are Enterprise Objects
- Project Objects may not duplicate existing Enterprise Objects
- Projects may not duplicate classes/approval processes/pages etc that function upon Enterprise Objects
- Projects that need to create/update functionality that interacts with Enterprise Objects, are responsible for engaging with the Design Authority for review and risk management.

4.2 Format Conventions

Apex is designed mostly based on the Java language conventions and hence we should adhere the formatting of the source to its conventions. Following list documents each rule and at the end of the rule, there is an elaborated example covering most of these rules.

4.2.1 Indentation

Code must be indented using a tab character and where applicable Tab must be set as 4 space width.

4.2.2 Bracing

Brace is an either {or }.

Start Brace must be in the same line as the statement and end brace must be on its own line, completing the indentation

Correct: if (setABCBudgetId.Size() > 0) {

CPABC_BudgetEntryUtility.updateBudgetEntriesBalance(setAPACAMERBud

```
getId);

}

Incorrect: if

(setABCBudgetI

d.Size() > 0)

{


CPABC_BudgetEntryUtility.updateBudgetEntriesBalance(setAPACAMERBud
getId);

}
```

- All if blocks must be braced even if it contains single statement.

Correct:

```
if (setABCBudgetId.Size() > 0) {


CPABC_BudgetEntryUtility.updateBudgetEntriesBalance(setAPACAMERBu
dgetId);

}
```

Incorrect:

```
if (setABCBudgetId.Size() > 0)


CPABC_BudgetEntryUtility.updateBudgetEntriesBalance(setAPACAMERBu
dgetId);
```

4.2.3 Spacing

Along with indenting, spacing is the most important visual aspect that enhances the code readability. And hence correct spacing conventions must be followed as outlined below.

- Before opening a brace, there must be a space

Correct:

```
if (setAPACAMERBudgetId.Size() > 0) {


CPABC_BudgetEntryUtility.updateBudgetEntriesBalance(setAPACAMERBu
dgetId);
```

}

Incorrect:

if

(setAPACAMERB

udgetId**.**Size**() >**

0**){**

CPABC_BudgetEntryUtility**.**updateBudgetEntriesBalance**(**setAPACAMERBu
dgetId**);**

}

- There should NOT be any space before a comma and there must be a space after a comma (except if that comma is part of a String literal.

Correct:

String address**,** street**,** city**;**

Incorrect:

  - String address **,** street **,** city**;**
  - String address**,** street**,**city**;**

- There should a space after every end-parenthesis (**(**), and there should NOT be a space before start parenthesis (**)**).
- There must be a space after every binary operator (if it is unary operator, space must not be used).

Correct:

Integer a, b, c; c = a + b; c++;

Incorrect:

Integer a, b, c; c = a+b; c ++;

4.2.4 Blank Lines

- Blank line must be added before starting and after ending a code block (block of code is code enclosed within **{}** braces), unless it is a **} else if {** condition, and in such case, there must not be any blank line.
- There must be maximum one or two blank lines.

4.2.5 Line Wrapping

All code lines must be wrapped to around 100 characters. This is a guideline so even if it exceeds by +/- 10 characters, it must be fine.

4.3 General Naming Convention

Naming convention in Salesforce is a rule to follow as you decide what to name your identifiers like class, variable, constant, method, etc. But it is not forced to follow. So, it is known as convention, not rule. Naming conventions make the application easier to read and maintain. A well formatted code increases readability, understanding and ultimately maintainability of the code base. Before starting with salesforce naming conventions let's talk about What is PascalCase, camelCase, SNAKE_CASE?

camelCase: Each word in the middle of the respective phrase begins with a capital letter. for example

```
String firstName;
```

PascalCase: It is same like Camel Case where first letter always is capitalized. for example

```
Class UserController{}
```

Kebab-case: Respective phrase will be transferred to all lowercase with hyphen (-) separating words. For example

```
<c-hello-world-form></c-hello-world-form>
```

SNAKE_CASE: Each word should be in capital with _ like .

```
private static final Integer MY_INT;
```

Functional Type	Name Suffix	Examples
Trigger	Trigger	UserTrigger
Trigger Handler	TriggerHandler	UserTriggerHandler
Trigger Action	TriggerAction	UserTriggerAction
VF Controller	Controller	UserController
VF Controller Extension	Ext	UserExt
Service Class	Service	UserService
Model / Wrapper Class	Wrapper	UserWrapper
Web Service (SOAP)	Ws	UserToolsWs
Web Service (REST)	Rest	UserCreateRest
Email Service	EmlSvc	UserCreateEmlSvc
Asynchronous (Future)	Async	UserCreateAsync
Asynchronous (Batch)	Batch	UserCreateBatch
Scheduled Apex	Job	UserCleanupJob
Test Class	Test	UserCreateTest
Queueable Apex	Que	UserSyncingQue

Visualforce Page	-none-	UserClone
Visualforce Component	Cmp	UserCloneCmp

Component Type	Sub Type	Convention Type	Example
Apex/Javascript (wherever applicable)	Class Name	PascalCase	ClassNamePOSTFIX
	Variable Name	camelCase	List<Account> accountList;
	Method Name	camelCase	showAccountDetail();
	Constants	SNAKE_CASE	private static final String ACCOUNT_LIMIT =’10’;
	Trigger (Not used in javascript)	PascalCase	UserTrigger
Visualforce page		PascalCase	AccountClone
Lightning web component OR Aura	HTML file	camelCase	helloWorld.html
	Javascript file	PascalCase	export default class HelloWorld extends LightningElement{ }
	CSS file	camelCase	testStyle.css
	CSS Class		CSS classes should be named based on the component that is being addressed Any name that is longer than one word, needs to be in this format : class-name Multi word name should be separated by a ” – “

4.4 Project Specific Naming Convention

Naming convention is expected to be followed for Salesforce project. If developer comes across any new types other than listed below, naming convention has to be first agreed within the team and document should be updated and circulated.

4.4.1 Modules

One more important feature Force.com platform development lacks is namespaceing. All classes created belong to same (default) namespace. This becomes difficult to organize the code base, especially with large enterprises.

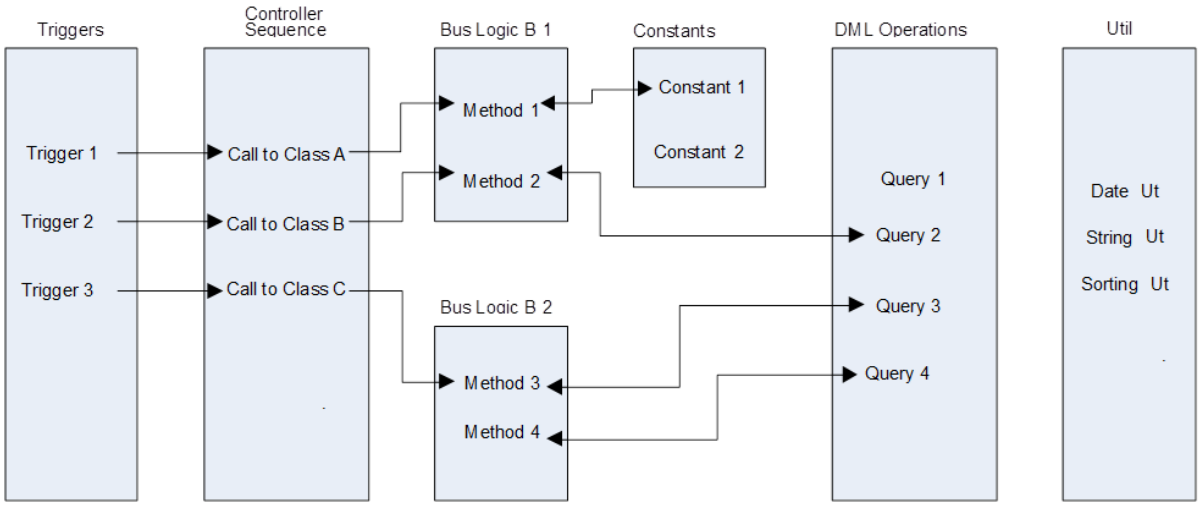
- Object /Field API : for any object or field creation, please discuss within the team and decide whether its required
- Page Layout
  - The page layout name should clearly call out the functional requirement like
    - Example : Partner Verified Account : Page layout for Partner after the account has been verified.
- Validation Rule
  - Validation rule name should clearly mention the validation criteria in plain English.
    - Example: PricingTypeRequiredPartnerAccounts
- Custom Setting
  - If Custom setting is for specific module than the module name should be prefixed.
    - Example : SFA_DealMaker_Integration_User_Id _CS
  - If we are using custom setting for admin process like inactive all the triggers during data migration then it should start with SFA_ADM_InactiveTriggers_CS
  - If we are using custom setting that can be used across modules for examples static values to store state and country, then name it as SFA_ALL_CountryStateList_CS
- Visual force page
  - Visual force page should clearly call out the functional intent of the page.
    - Example :SFA_ ParnterAccountDetails_Page
- Email Alert
  - Email alert name should be as per the functional requirement like send email to Rep when Partner adds product to opportunity
    - Example : PartnerAddsNewProductSendToRep
- Outbound Messaging
  - The name of the outbound message should clearly call out the function requirement and end point
    - Example : MatchAccountwithMDMonCreate_Siebel

5.0 Coding Standards

Coding standards are intended for developers who will be developing code into the Force.com org. It must be noted that these standards and project resulting code are subject to adhoc audit and remediation if significant issues are identified in code. These standards are a means to provide code consistency and adherence to Salesforce best practices and sustainable delivery on the Customer Org. Enforcing standards have the following benefits:

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

5.1 Code Structure and Organization



- You should have only 1 trigger, basically create one trigger per object (before insert, before update, before delete, after insert, after update, after delete & after undelete) and then call your apex class to do the processing. The trigger should not store any processing logic.

- **Sequence controller** – Is a single class which contains call to all classes related to an object or functionality. These classes are called by triggers as required.
- **Business Logic** – These are individual classes with particular methods containing the functionality specific business logic.
- **Constants** – this contains all the constants being used for a particular functionality. Constants in schedulable classes should be kept in separate classes.
- **DML operations** – contains all the queries and DML operation to be used by methods in business classes. Remember to set the query result to NULL to reduce the possibility of heap size error.
- **Utils** – Util classes are to manage the reusable methods for a particular functionality.

5.1.1 Separation of Concerns - Apex Enterprise patterns

Complex code gets out of hand when it is not partitioned properly. Heavily intermixed code is error prone, less maintainable, and hard to learn for new developers.

Design patterns are helpful for such complex projects. Design patterns are class structures that solve many of the commonly occurring issues in software. If algorithms are like baking instructions, then design patterns are like blueprints. They allow us to share a similar vocabulary of abstraction above that of implementation.

At a high level, applications have three aspects: storage, logic, and a means to interact with them (either through humans or through other applications).

Once these are separated, it becomes possible to define the layers within the application. Each layer has its own set of concerns and responsibilities for other layers, and for the application as a whole. Separation of concerns enables each layer to be reusable and maintainable without impacting other layers.

Salesforce recommends developers to follow Apex Enterprise Patterns which is explained in detail below.

- **Service Layer:** Defines the application's boundary with a layer of services that establishes a set of available operations and coordinates the application's response in each operation
- **Domain Layer:** A set of common requirements, terminology, and functionalities for any software program constructed to solve a problem in that field
- **Selector Layer:** A layer of mappers that moves data between objects and a database while keeping them independent of each other and the mapper itself

You can follow Trailhead module - Apex Enterprise Patterns Trailmix | Salesforce Trailhead to understand more on each layer and implement the same design pattern for your complex projects.

5.2 Configuration Standards

Object

- Before creating an object you must check that if a standard object or an existing custom object can be used for the same requirement or not.
- Object name should be Self- Explanatory, why this is created for.
- You must always provide the detail description while creating the object.
- Sharing model should be default private for all objects, make use of public group sharing and manual sharing to create data visibility, unless explicit business needs to make it public available for all roles across the org.

Field

- Before creating a field, check that if an existing custom field can be used for the same requirement or not.
- Field name should be less than 40 characters and Self- Explanatory, why this is created for.
- Field description should follow following pattern

<Module Name>#<GEO>#<FIELD USAGE >#<Functional Description>

- GEO = APAC,EMEA, AMER or GLOBAL
- FIELD USAGE

- If the field is used by integration points then mention the integration service as well , for example : INTG_APRIMO . If the field is used by more than one integration points than you can expand the definition as INTG_SYS1_INGT_SYS2
- If the field is used for data migration from other system then tag the field as DATAMG Example
- MJA#EMEA#INTG_SIEBEL#DATAMG# The field captures whether the account is major account or not and the value flows from Siebel to SFDC for EMEA.

- Consult with object owners before creating the fields.

Page Layouts

- To reduce the number of page layouts to maintain, use the same page layout for all profiles for a specific record type.
- Use field-level security to restrict users' field access; use page layouts primarily to organize pages.

Validation Rules

- When creating validation rules, consider all the settings in your organization that can make a record fail validation such as assignment rules, field updates, field-level security, or fields hidden on a page layout.
- Be careful not to create two contradicting validation rules for the same field; otherwise, users will not be able to save the record.
- A poorly designed validation rule can prevent users from saving valid data. Make sure you thoroughly test a validation rule before activating it. Users will never be able to save a record if your formula always returns a "True" value.

Approval Process

- Avoid associating more than one field update with a rule or approval process that applies different values to the same field.
- After creating approval processes, use the graphical Process Visualizer to visualize and understand the defined flow and decisions.
- Use one flow for each object/event as a best practice

Flows

- Utilize Before-Save Flows for Same Record Updates
- Use After-Save Flow Instead of Process Builder/Workflow
- Plan before you build – Outline the purpose of the flow.
- Ensure consistent naming across elements and variables
- Document every step
- One 'Record triggered flow' per object
- Check for Nulls/Empty Collections
- Don't loop over large data volumes (It will trigger flow element limit)
- Always Test Your Flows
- Consider Using Subflows
- Never Perform DML Statements In Loops
- Never Hard Code Ids (Use Constants IF You Must)
- Plan For Faults (And Handle Them)

Views, Reports and Dashboard

- Report name or View name should clearly indicate what its contents.
- Reports must be segmented and stored in separate folder based on their purpose and module.
- Reports to be used organization wide should be kept in unified public folder. As reports in personal folder will not be available for public usages.
- Report name should postfix with “-Dashboard” if its part of dashboard.

5.3 Apex Standards and Best Practices

5.3.1 General Apex coding best practices

- Use "with sharing" setting on all apex classes. If you need to bypass sharing rules, call into another class that is defined "without sharing". Ensure to explain reason for bypass sharing rules in the code when using "without sharing" setting.
- In order to support readability and maintainability of the code use inner classes wherever possible. Ensure to use inner sharing in inner classes to ensure sharing settings of the outer class are inherited.
- When creating code, try to limit the size of code to <45 lines of code. If exceeding this limit, validate the code and verify if there is potential to break it down into multiple, reusable methods.
- Never copy & paste code - if you need the same code in a different place encapsulate it in a method and re-use the method.
- Don’t leave behind orphaned (or obsolete) classes, properties and methods. Make sure you comment or clear the unnecessary classes and methods.
- Only use indexed fields in where clause in SOQL queries if the record count for the queried object might exceed 100k at some point.
- Always check for Null values for a variable before using it inside a logic. This will help to handle exceptions upfront and to help to inform user/system about missing data.
- All the commented code should not be moved to production. We should move only the relevant code to production.
- Try to reduce the number of properties in Apex classes/Lightning Components to ensure readability. If a large set of properties is required try to group them in an inner class.
- We can reference the business logic and actions implemented as apex classes in the custom trigger.
- Other apex standards and best practices like avoiding SOQL inside for loops, using bind variables to prevent SOQL injections, exceptions etc., are explained in detail in upcoming sections.

5.3.2 Apex Classes

It is not legal to define a class and interface with the same name in the same class. It is also not legal for an inner class to have the same name as its outer class. However, methods and variables have their own namespaces within the class so these three types of names do not clash with each other. In particular it is legal for a variable, method, and a class within a class

to have the same name.

```
/**************************************************************************
Name: ConfigureOpportunity()
Copyright © 2015  Salesforce
Purpose: ..
History: ..
VERSION        AUTHOR        DATE            DETAIL
        DESCRIPITOPN
1.0             Name          dd/mm/yyyy      INITIAL DEVELOPMENT
        CSR
**************************************************************************/
<access> class <name>
{
        //Non-Static Variables
          ..
        // Constructors
        ..
        // getter & setter methods
          ..
         //logical methods
        ..
        //inner classes
        ..
       //action methods
       ..
}
```

5.3.3 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 in Block comments

```
public class ConfigureProducts
{
…
….
/************************************************************
```

```
Description: ….
Parameter: [Optional]
Return: [Optional]
Throws (Exception): [Optional]
**********************************************************/
  public string addUDAC (param1, param2){
  …….
  }
}
```

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 customeID { get; set; }

public string customerID; //variable declaration section

public string getCustomerID () {  //get method definition

    return <string>;

}

public string setCustomerID () {  //set method definition

    customerID = value;

}
```

5.3.4 Static Variable

Never hard code an Id, link etc. in the code directly. Use Custom Labels or static Apex Use Custom labels for storing Error messages.

Classes as a mechanism to drive values for a variable.

5.3.5 Code Blocks

Always use braces to in the code blocks, given below a quick example of what should be the appropriate way of writing code blocks

Never use the pattern:

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

Never leave empty code blocks:

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

Always use:

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

5.3.6 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
}
```

- Do not throw NullPointerException – developers may misunderstand the source – instead consider throwing an InvalidParameterValueException.

- Avoid rethrowing 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 {
      //do something
}
catch  (Exception e){
    //handling exception
}
finally {
throw new Exception (); //avoid doing this
}
```

- 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){
    ApexPage.addMessage(se);
}
```

5.3.7 Exception Handler Basics

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.

The Exception Logging Helper class should be used to log exceptions.
Exception handlers should trap typed exceptions where possible:

```
try {
      //do something
}
catch  (DMLException e){
    //log then do something
}
catch(Exception e){
    //log then do something
}
finally {
   //Cleanup
}
```

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

```
try {
      throw new Exception ();
}
catch(Exception e){
    throw e;
}
finally {
   return “A. O. K”; // not recommended
}
```

1. Encapsulate error handling in custom exception classes to ensure errors are logged in the same way and error messages are displayed consistently across the platform.
2. Always wrap SOQL queries and DML statements in a try/catch block in order to ensure that any possible exception is fetched and can be logged and displayed properly to the end user.
3. Exceptions should never be suppressed without a comment in code explaining why
4. Empty exception handlers are never acceptable.
5. Exceptions should be logged to a logging object: ApplicationLog__c
6. Bulk log messages should be collated and passed to the Log in one call to prevent excessive DML calls
7. Page based exceptions should be added to ApexPages.Messages
8. In triggers, exceptions should use addError() method to prevent committing
9. Do not catch NullPointerExceptions – identify and resolve the core issue.

5.3.8 Exception Logging Helper

The Exception Logging Helper is used as follows:

ExceptionLoggingHelper.createErrorLog(User, Class Name, Method Name, Message, EXECEPTION OBJECT, DebugLevel, IntegrationPayload, ReferenceInfo, Timer);

Message	String	System message or custom message
Exception	String	Exception Object (to be kept for future changes)
DebugLevel	String	This will be set to ERROR by default; developer can set it either of Error, Info, Warning or Debug
Integration Payload	String	This will be set to NULL by default; developer needs to set the value in case of integration exception
User	Id	Current User **Stateful**
Class Name	String	Class name where the exception occurred.
Method Name	String	Method name where the exception occurred
ReferenceInfo	String	This will be set to NULL by default; developer needs to set the value batch job id in case of any batch jobs
Timer	Number	This will be set to NULL by default; developer needs to set by calculating execution end time – execution start time

5.3.9 Avoid SOQL Queries/DML inside FOR Loops

There is a Governor limit to the number of SOQL queries and DML Statements in an execution context. Ensure that SOQL/DML 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> accList = [SELECT Id, Name FROM Account WHERE Name LIKE 'Acme%'];
for (Account a : accList){
    //your code without DML statements here
    a.Name = 'New Name';
}
update accList;
```

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

```
trigger contactTrigger on Contact (before insert, before update){
    for (Contact c : trigger.new){
        Account acc = [SELECT Id, Name, BillingState FROM Account WHERE Id = :c.AccountId];
        If (acc.BillingState == 'CA'){
            //do something
        }
    }
}
```

Instead do:

```
trigger contactTrigger on Contact (before insert, before update){
    Set<Id> accIdSet = new Set<Id>();
    for (Contact c : trigger.new){
        accIdSet.add(c.AccountId);
    }
    // Do SOQL query outside the iterator
    Map<Id, Account> accountMap = new Map<Id, Account>( SELECT Id, Name, BillingState FROM Account WHERE Id in : accIdSet);
    for (contact c: trigger.new){
        if (accountMap.get(c.AccountId).BillingState == 'CA'){
            //do something
        }
    }
}
```

5.3.10 Use bind variable to prevent SOQL Injections

SQL injection is a common application security flaw that results from insecure construction of database queries with user-supplied data. SQL injection flaws are extremely serious. A single flaw anywhere in your application may allow an attacker to read, modify or delete your entire database.

Do not:

```
    public PageReference query() {
        String qryString = 'SELECT Id FROM Contact WHERE ' +
            '(IsDeleted = false and Name like \'%' + name + '%\')';
        queryResult = Database.query(qryString);
        return null;
    }
}
```

Rather do:

```
    public PageReference query() {
        String queryName = '%' + name + '%';
        queryResult = [SELECT Id FROM Contact WHERE
          (IsDeleted = false and Name like :queryName)];
        return null;
    }
}
```

5.4 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, Name, BillingState FROM Account];
for (Account acct: accts){
        //do something
}
```

Use:

```
for (List<Account> acct: [SELECT Id, Name, BillingState FROM Account]){
        //do something
}
```

5.4.1 Using the Limits Apex Methods to Avoid Hitting Governor Limits

Apex has a System class called Limits that lets output debug messages for each governor limit.

There are two versions of every method:

- The first version returns the amount of the resource that has been used in the current context.
- The second version contains the word limit and returns the total amount of the resource that is available for that context.

Embed these types of statements in code and determine if or when any governor limits are about to be exceeded, When writing test methods check for limits usage (for example when running queries) to ensure that testing is not exceeding governor limits, as limitations might be significantly lower in sandboxes. Using either the System Log or Debug Logs, it is possible to evaluate the output to see how the specific code is performing against the governor limits.

Additionally, enable Apex governor Limit warning emails.

When an end-user invokes Apex code that surpasses more than 50% of any governor limit, specify a user in organization to receive an email notification of the event with additional details.

5.5 Use of @future method

It is critical to write your Apex code to efficiently handle bulk or many records at a time. This is also true for asynchronous Apex methods (those annotated with the @future keyword). 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 50@future methods can be invoked within a single Apex transaction.

The parameters specified must be primitive data types, arrays of primitive data types, or collections of primitive data types.

Methods with the future annotation cannot take sObjects or objects as arguments. Methods with the future annotation cannot be used in Visual force controllers in either getMethodName or setMethodName methods, nor in the constructor.

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

1. No more than 50 method calls per Apex invocation

Future methods can't be invoked in loops e.g.:-

```
{
  for(Account a: Trigger.new){
    // Invoke the @future method for each Account
    // This is inefficient and will easily exceed the governor limit of      // at most 10 @future invocation
per Apex transaction     asyncApex.processAccount((String)a.id);
   }
 }
```

Here is the Apex class that defines the @future method:

```
global class asyncApex {

  @future
  public static void processAccount(Id accountId) {
      List<Contact> contacts = [select id, salutation, firstname, lastname, email                 from
Contact where accountId = :accountId];

        for(Contact c: contacts){
              System.debug('Contact Id[' + c.Id + '], FirstName[' + c.firstname + '], LastName[' +
c.lastname +']');                                    c.Description=c.salutation + ' ' + c.firstName + ' ' +
c.lastname;
       }
       update contacts;
  }
}
```

Since the @future method is invoked within for loop, it will be called N-times (depending on the number of accounts being processed). So if there are more than ten accounts, this code will throw an exception for exceeding a governor limit of only ten @future invocations per Apex transaction.

@future method should be invoked with a batch of records so that it is only invoked once for all records it needs to process:

```
{
   //By passing the @future method a set of Ids, it only needs to be     //invoked once to handle all of the data.
   asyncApex.processAccount(Trigger.newMap.keySet());
}
```

And now the @future method is designed to receive a set of records:

```
global class asyncApex {

  @future
  public static void processAccount(Set<Id> accountIds) {
      List<Contact> contacts = [select id, salutation, firstname, lastname, email from Contact where accountId IN
:accountIds];
      for(Contact c: contacts){
            System.debug('Contact Id[' + c.Id + '], FirstName[' + c.firstname + '], LastName[' + c.lastname +']');                                  c.Description=c.salutation + ' ' + c.firstName + ' ' + c.lastname;
       }
       update contacts;
  }
}
```

Notice the minor changes to the code to handle a batch of records. It doesn't take a whole lot of code to handle a set of records as compared to a single record, but it's a critical design principle that should persist across all of your Apex code - regardless if it's executing synchronously or asynchronously.

5.6 Writing Test Methods to Verify Large Datasets

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

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

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

```
{
  for(Contact ct: Trigger.new){
            Account acct = [select id, name from Account where Id=:ct.AccountId];
if(acct.BillingState=='CA'){
              System.debug('found a contact related to an account in california...');
ct.email = 'test_email@testing.com';          //Apply more logic here....
              }
  }
}
```

Here is the test method that tests if this trigger properly handles volume datasets:

```
public class sampleTestMethodCls {

        static testMethod void testAccountTrigger(){

              //First, prepare 200 contacts for the test data
      Account acct = new Account(name='test account');
      insert acct;

                  Contact[] contactsToCreate = new Contact[]{};
                   for(Integer x=0; x<200;x++){
                     Contact ct = new Contact(AccountId=acct.Id,lastname='test');
                       contactsToCreate.add(ct);
                     }

                  //Now insert data causing an contact trigger to fire.
                   Test.startTest();
                         Insert contactsToCreate;
                         Test.StopTest();
              }
}
```

This test method creates an array of 200 contacts and inserts them. The insert will, in turn, cause the trigger to fire. When this test method is executed, a System.Exception will be thrown when it hits a governor limit. Since the trigger shown above executes a SOQL query for each contact in the batch, this test method throws the exception 'Too many SOQL queries: 101'. A trigger can only execute at most 20 queries.

Now let's correct the trigger to properly handle bulk operations. The key to fixing this trigger is to get the SOQL query outside the for loop and only do one SOQL Query:

```
{
   Set<Id> accountIds = new Set<Id>();
for(Contact ct: Trigger.new)
                accountIds.add(ct.AccountId);
   //Do SOQL Query
   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('found a contact related to an account in
california...');                    ct.email = 'test_email@testing.com';
//Apply more logic here....
                 }
   }
}
```

Note how the SOQL query retrieving the accounts is now done once only. If you re-run the test method shown above, it will now execute successfully with no errors and 100% code coverage.

5.7 Avoid Hard Coding IDs

When deploying Apex code between sandbox and production environments, or installing Force.com AppExchange packages, it is essential to avoid hardcoding IDs in the Apex code. By doing so, if the record IDs change between environments, the logic can dynamically identify the proper data to operate against and not fail.

Here is a hard coded sample of the record type IDs that are used in an conditional statement. This will work fine in the specific environment in which the code was developed, but if this code were to be installed in a separate org (ie. as part of an AppExchange package), there is no guarantee that the record type identifiers will be the same.

```
for(Account a: Trigger.new){
  //Error - hardcoded the record type id
  if(a.RecordTypeId=='012500000009WAr'){
    //do some logic here.....
  }else if(a.RecordTypeId=='0123000000095Km'){       //do some logic here for a different record
type...
  }
}
```

Now, to properly handle the dynamic nature of the record type IDs, the following example queries for the record types in the code, stores the dataset in a map collection for easy retrieval, and ultimately avoids any hard coding.

```
//Query for the Account record types
    List<RecordType> rtypes = [Select Name, Id From
RecordType                  where sObjectType='Account' and
isActive=true];

    //Create a map between the Record Type Name and Id for easy retrieval
Map<String,String> accountRecordTypes = new Map<String,String>{};
for(RecordType rt: rtypes)
       accountRecordTypes.put(rt.Name,rt.Id);

     for(Account a: Trigger.new){

            //Use the Map collection to dynamically retrieve the Record Type Id
             //Avoid hardcoding Ids in the Apex code
if(a.RecordTypeId==accountRecordTypes.get('Healthcare')){
           //do some logic here.....
            }else if(a.RecordTypeId==accountRecordTypes.get('High
Tech')){                    //do some logic here for a different record type...
              }
    }
```

By ensuring no IDs are stored in the Apex code, you are making the code much more dynamic and flexible - and ensuring that it can be deployed safely to different environments.

5.8 Best Practices for Designing Bulk Programs

The following are the best practices for designing bulk programs:

- Minimize the number of data manipulation language (DML) operations by adding records to collections and performing DML operations against these collections.
- Minimize the number of SOQL statements by preprocessing records and generating sets, which can be placed in single SOQL statement used with the IN clause.

5.9 Apex Triggers best practices:

Lightning flows can be a best alternative solution instead of triggers for actions on insert, update and delete events. Still there are some situations like actions on undelete event, validations require triggers to be used.

- Streamlining Multiple Triggers on the Same Object - Avoid redundancies and inefficiencies when deploying multiple triggers on the same object. If developed independently, it is possible to have redundant queries that query the same dataset, or have redundant FOR statements.
- Before-Triggers - This event is ideal for performing data validation, setting default values, or performing additional logic and/or calculations, which doesn't require DML.
- After – Triggers - This particular event is ideal for working with data that is external to the record itself (such as referenced objects) or for creating records based on information from the triggered object.

  Example – Creation of an Opportunity after an Account is created since Account ID is required to perform this.

As a best practice, avoid performing a DML operation on the record that initiated the trigger event for an after-trigger event, even if it is possible.

- Trigger Context – Trigger,new holds the information about the record(s) that was(were) just inserted or updated whereas Trigger.old can be used to search for the data that was deleted. When Salesforce compiles triggers, it looks at the fields that are being referenced, and ensures that data is available at runtime. So there is no need to query the information in the same record. But query is required to get data related to that record, otherwise null is returned without producing an error. Trigger.newMap and Trigger.oldMap contains a key value pair of ID to SObject and contains current information and the one exists before the record was changed.

- Bulk Mode Triggers - Always use the trigger.new collection regardless of whether one record or multiple records are being handled. This is because all triggers execute in batch mode. The trigger should be prepared to be able to handle a batch of records. Additionally, the number of queries and loops can be reduced by using Apex Maps and sub select queries efficiently.

- If multiple triggers are executed on an object, the object cannot have any dependencies on a field that another trigger is expected to update. Salesforce.com does not guarantee the order in which each of these triggers fire. It is recommended not to assume anything.

- The next question is: how many triggers should be there per object? Most people are of the opinion that there should be only one trigger per object. However, before-trigger events and after-trigger events should have their own triggers, since the before-trigger is a unique event in

comparison with an after-trigger event. It is possible to have a single method that can divide the work between before and after events.

- Include the code's logic into a class and then use triggers as hooks for calling the methods within this class. This leads to reusability of classes across organization as code within trigger can’t be reused.

- Use the webService keyword in a trigger only when it is in a method defined as asynchronous, that is, when the method is defined with the @future keyword.

- A trigger invoked by an insert, delete, or update of a recurring event or recurring task results in a runtime error when the trigger is called in bulk from the Force.com API.

- Use static variables to pass data between triggers.

5.10 Apex Scheduling Best Practices

- Salesforce.com only adds the process to the queue at the scheduled time. Actual execution may be delayed based on service availability.

- Use extreme care if you are planning to schedule a class from a trigger. You must be able to guarantee that the trigger will not add more scheduled classes than the ten that are allowed. In particular, consider API bulk updates, import wizards, mass record changes through the user interface, and all cases where more than one record can be updated at a time.

- Though it's possible to do additional processing in the execute method, Salesforce.com recommends that all processing take place in a separate class.

- There is a limit of 100 classes that can be scheduled.

5.11 Batch Apex Best Practices

Use extreme care if you are planning to invoke a batch job from a trigger. You must be able to guarantee that the trigger will not add more batch jobs than the five that are allowed. In particular, consider API bulk updates, import wizards, mass record changes through the user interface, and all cases where more than one record can be updated at a time.

- When you call Database.executeBatch, Salesforce.com only places the job in the queue at the scheduled time. Actual execution may be delayed based on service availability.
- When testing your batch Apex, you can test only one execution of the execute method. You can use the scope parameter of the executeBatch method to limit the
- number of records passed into the execute method to ensure that you aren't running into governor limits.
- The executeBatch method starts an asynchronous process. This means that when you test batch Apex, you must make certain that the batch job is finished before testing against the results. Use the Test methods startTest and stopTest around the executeBatch method to ensure it finishes before continuing your test.
- Use Database.Stateful with the class definition if you want to share variables or data across job transactions. Otherwise, all instance variables are reset to their initial state at the start of each transaction.
- Database.AllowsCallouts - To allow callouts from Batch apex need to extend AllowCallouts interface. Use Apex callouts to tightly integrate your Apex with an external service by making a call to an external Web service or sending a HTTP request from Apex code and then receiving the response. Apex provides integration with Web services that utilize SOAP and WSDL, or HTTP services (RESTful services).
- Before any Apex callout can call an external site, that site must be registered in the Remote Site Settings page, or the callout fails. Salesforce prevents calls to unauthorized network addresses.
- Methods declared as future cannot be called from a batch Apex class.
- You cannot call the Database.executeBatch method from start and excute method while finish can call.
- Each batch Apex invocation creates an AsyncApexJob record. Use the ID of this record to construct a SOQL query to retrieve the job’s status, number of errors, progress, and submitter.
- For a sharing recalculation, Salesforce.com recommends that the execute method delete and then re-create all Apex managed sharing for the records in the batch. This ensures the sharing is accurate and complete.

Visualforce Best Practices

Salesforce provides good guidelines for improving visualforce pages performance and design considerations.Refer below links for detailed documentation

https://developer.salesforce.com/docs/atlas.en-us.salesforce_visualforce_best_practices.meta/salesforce_visualforce_best_practices/pages_best_practices_perf_optimizations.htm https://developer.salesforce.com/docs/atlas.en-us.pages.meta/pages/pages_intro.htm

These documentation covers best practices for

- Improving Visualforce Performance
- Accessing Component IDs
- Static Resources
- Controllers and Controller Extensions
- Using Component Facets
- Page Block Components
- Rendering PDFs
- <apex:panelbar>

5.12 Considerations Querying Large Data Set

Use of Limit Clause - For example, if the results are too large, the syntax below causes a runtime exception:

```
//A runtime exception is thrown if this query returns enough records to exceed your heap limit.

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

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

// Use this format for efficiency if you are executing DML statements  // within the for loop.  Be

careful not to exceed the 150 DML statement limit.

for (List<Account> accts&amp;nbsp;: [SELECT id, name FROM account

                        WHERE name LIKE 'Acme']) {

    // Your code here

    update accts;

}
```

5.13 Bulk Data Processing:

For optimal processing time, consider these tips when planning your data loads and data processing in bulk. Always test your data loads in a sandbox organization first. Note that the processing times can be different in a production organization.

There are several ways to process bulk data:

Use Parallel Mode Whenever Possible

You get the most benefit from the Bulk API by processing batches in parallel, which is the default mode and enables faster loading of data. However, sometimes parallel processing can cause lock contention on records. The alternative is to process using serial mode. Don't process data in serial mode unless you know this would otherwise result in lock timeouts and you can't reorganize your batches to avoid the locks.

You set the processing mode at the job level. All batches in a job are processed in parallel or serial mode.

For more details on Bulk API please refer to below salesforce documentation:

https://developer.salesforce.com/docs/atlas.en-us.224.0.api_asynch.meta/api_asynch/asynch_api_intro.htm#:~:text=Bulk%20API%20is%20based%20on,processes%20batches%20in%20the%20background.

Organize Batches to Minimize Lock Contention

For example, when an AccountTeamMember record is created or updated, the account for this record is locked during the transaction. If you load many batches of AccountTeamMember records and they all contain references to the same account, they all try to lock the same account and it's likely that you'll experience a lock timeout. Sometimes, lock timeouts can be avoided by organizing data in batches. If you organize AccountTeamMember records by AccountId so that all records referencing the same account are in a single batch, you minimize the risk of lock contention by multiple batches.

Be Aware of Operations that Increase Lock Contention

These operations are likely to cause lock contention and necessitate using serial mode:

- Creating new users
- Updating ownership for records with private sharing
- Updating user roles
- Updating territory hierarchies

If you encounter errors related to these operations, create a separate job to process the data in serial mode.

Minimize Number of Fields

Processing time is faster if there are fewer fields loaded for each record. Foreign key, lookup relationship, and roll-up summary fields are more likely to increase processing time. It's not always possible to reduce the number of fields in your records, but if it is, loading times improve.

Minimize Number of Triggers

You can use parallel mode with objects that have associated triggers if the triggers don't cause side-effects that interfere with other parallel transactions. However, Salesforce doesn't recommend loading large batches for objects with complex triggers. Instead, rewrite the trigger logic as a batch Apex job that is executed after all the data has loaded.

Minimize Number of Batches in the Asynchronous Queue

Salesforce uses a queue-based framework to handle asynchronous processes from such sources as future and batch Apex, and Bulk API batches. This queue is used to balance request workload across organizations. If more than 2,000 unprocessed requests from a single organization are in the queue, any more requests from the same organization will be delayed while the queue handles requests from other organizations. Minimize the number of batches submitted at one time to ensure that your batches are not delayed in the queue.

Optimize Batch Size

Salesforce shares processing resources among all its customers. To ensure that each organization doesn't wait too long to process its batches, any batch that takes more than 10 minutes is suspended and returned to the queue for later processing. The best course of action is to submit batches that process in less than 10 minutes.

Adjust batch sizes based on processing times. Start with 5000 records and adjust the batch size based on processing time. If it takes more than 5 minutes to process a batch, it can be beneficial to reduce the batch size. If it takes a few seconds, increase the batch size. If you get a timeout error when processing a batch, split your batch into smaller batches, and try again.

6.0 Code Convention

6.1 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.

6.2 Comments

Comments in code are important because they allow 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.

Also, the comments added to each method/code blocks inside the classes will be helpful to understand the purpose of those. It's better to provide comments for change in logic inside code block and at the top of the class that who, when modified along with ticket/CR number for which it is modified.

Various components comment with example: -

6.2.1 Apex Comments

```
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*/
```

6.2.2 VisualForce Comments

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

6.2.3 Javascript Comments

```
/**
Return an object as a JSON string

* @method toJSON
* @return {Object} copy of…
*/
toJSON : function (){
}
```

6.2.4 CSS Comments

```
p{
    Color: red;
    /* This is a single line comment */
}
/* This is a multiline comment */
```

6.3 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 {
    //methods for testing
    static testMethod void test1(){
            //implement test code
    }
}
```

The following need to be adhered to when writing test code

- Use a consistent naming convention including “Test” and the name of the class being tested (e.g., AccountTriggerTest)

- To make your tests simpler and easier to follow, you can move your data creation into a reusable class. Called data factories, these reusable classes contain methods that model the creation of one or more objects. For more details please refer to salesforce documentation.

- Cover as many lines of code as possible. You must have 80% of your Apex scripts covered by unit tests to deploy your scripts to pre-prod and production environments. In addition, all triggers must have unit test coverage.

- When testing for governor limits, use Test.startTest and Test.stopTest and the Limit class instead of hard-coding governor limits.

- 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.

- Use Assert class and methods to verify the test results.

- Use System.runAs() to execute code as a specific user to test for sharing rules (but not CRUD or FLS permissions)

- 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 Apex code coverage.

- Do not use try catch statements in test methods

- Write Mock Callout classes for API testing. For more details please visit salesforce documentation.

- User records are not covered by test methods

- Avoid using @SeeAllData unless required via the API (eg ConnectAPI)

For more information, please refer to https://developer.salesforce.com/docs/atlas.en-us.apexref.meta/apexref/apex_methods_system_test.htm

6.4 Additional Rules

- Make sure that each Page items(all VisualForce tags) has ID and the IDs is 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 sSFAe and does not include usersupplied 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

6.5 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(".cu
stomer id").innerText().toLowerCase()
```

Avoid inlining javascript – use binding handlers eg

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

7.0 Code security and Secure Coding

Salesforce has incorporated several security defenses into the Force.com platform itself.

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

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

Show and Tell session Conducted after every sprint/release as applicable to get early feedback and acceptance.

Below is the list of most repetitive issues found in Checkmarx tool scan:

- DML statements inside loops
- Hardcoding (IDs, List[0], etc)
- SOQL / SOSL inside loops
- Queries without WHERE clause or LIMIT clause
- Non-bulk apex methods
- Use the future method in a loop
- Multiple triggers on the same object
- Test method without using assert
- SOQL / SOSL Injection
- Cross-Site Scripting. This vulnerability can occur if an attacker inserts unauthorized JavaScript, Visual Basic Script, HTML, or other content into a web page that another user is viewing.
- The CRUD issue. It occurs when a user tries to find, insert, update, update, or delete any record without confirming whether the user has permission to operate or not.
- Frame Spoofing vulnerability. It occurs when data comes in software through an untrusted source and source URL of iframe without being validated
- Checkmarx will look for variables with names like 'password', 'credentials', 'Authentication' etc.. and when it sees that you are assigning them a value, it warns you that you might be storing sensitive information in the code (hardcoding it). In the case that you mentioned it looks like a false positive because this is not sensitive information.

7.1 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 make sense to include anti- XSS filters on commands that are intentionally added to a page

7.2 Cross-Site Request Forgery (CSRF)

Cross-Site Request Forgery (CSRF) flaws are less of a programming mistake as they are a lack of a defense. 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 trSFAfic 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 Force.com 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 defenses, either deliberately or inadvertently.

Developers should be cautious about writing pages that take action based upon a user-supplied parameter.

7.3 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.

7.4 Data Access Control

The Force.com 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 behavior 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 con = [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 an 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
}
```

8.0 Development Processes

- All code must be held within the Customer Source code management system.
- All code must be auditably peer reviewed by a senior developer.
- All commits need comments including User Story ID and description of the work performed.
- The source control repo follows the Git Flow branching model, as follows:

1) Development work starts by creating and committing to Feature branches from the Develop branch.

2) When Feature branch code is ready and peer reviewed, a Pull Request is requested to merge it into the Develop branch.

3) Once merged into the Develop branch the Code will be quality checked and build verified and unit tested in a Continuous Integration Sandbox.

   Following successful automated quality checks: -
   a Pull Request is used to merge the Develop branch into a Master branch.

   This branch will be used as the deploy version into QATest, PreProd and Production.

   Some of the best practices can be followed while moving the code to branches and higher environments.
   - Remove debug statements though it was required during development.

9.0 Automated Code Quality Checking

The Cognizant or Customer environment will provide automatic code quality checking capability through the use of tools, such as Code Sense (based on PMD), and a number of rules preconfigured on the platform to work with the Customer source control system. The objective of which is to drive up code quality and ensure sustainability on the Customer platform as multiple projects look to deliver into a single ORG. The use of these tools is **supplemental to peer reviews and quality checks in projects**.

However, within the coding standards we highlight a few tools & few exception use cases to developers to ensure transparency and clarity in this area or advise where we have made changes to the default settings in the quality tool rules for information.

9.1 Cognizant® Code Sense for Salesforce

PMD stands for Programming Mistake Detector. It is an open-source static source code analyzer that reports on issues found within the application code. This finds common programming flaws like unused variables and empty catch blocks. It will allow us to have better quality and avoid maintenance, performance, and bug problems in our Apex code.

Cognizant Salesforce CoE has developed a tool called **Code Sense** which is based on PMD. These tools have a wider coverage then the OOB rule sets from PMD and has close to 200 rule sets compared to the 75 odd rule rules provided by PMD. The step for using the tools is available in the below link. Login to this link via Single Sign-on and search for Code Sense and download the user guide.

https://eas.lightning.force.com/lightning/r/CollaborationGroup/0F9C0000000LgfxKAC/view

To get access to the tool please raise a HelpXchange ticket in

https://eas.lightning.force.com/lightning/r/CollaborationGroup/0F9C0000000LgfxKAC/view

and CoE team will address the request.

Disclaimer: This tool is intended for Cognizant associates use only and should be removed/uninstalled when the associate is moving out of the said engagement. For any additional clarification please reach out to EAS SFDC CoE (Cognizant) EASSFDCCoE@cognizant.com.

9.2 Salesforce Optimizer

Consider running Salesforce Optimizer as part of your monthly maintenance, before installing a new app, before each Salesforce release, or at least once a quarter. You can run the report as often as you want to keep on top of maintenance activities. You can set the App to run automatically on a monthly basis.

- From Setup, enter Optimizer in the Quick Find box, then select Optimizer.
- Enable Optimizer by allowing access, if you haven't done so already.
- Decide if the app should automatically run and update.
- Click Open Optimizer.

Salesforce sends an in-app notification when results are ready. Access Optimizer results by clicking the notification. Return to the Optimizer in Setup to review your results.

In the App, Org Metric History is shown with graphs for file storage limits, data storage limits, and static resource limits. These graphs give a high-level visual overview for how these limits have impacted your org.

Salesforce also saves an .xls file in Salesforce Files. The file includes some of the information from the report:

- Feature section and subsection per the report's table of contents
- Type of feature analyzed, along with the number of items found
- Severity of observation

Use the .xls file to load the data into Salesforce to analyze it for trending and historical analysis. By uploading data into a Salesforce custom object, you can create triggers and alerts when various thresholds are reached.

For more details please visit salesforce documentation.

9.3 Confusing Ternary

For long term maintainability and viability code must be readable and clear for parties external to a project to understand. Initial set of rules regarding confusing ternary were set so that errors of this type are deemed major. This has been revised in the case of direct examples using IF statements of which there are many ways to write them.

Ideally in an **if** expression with an else clause, avoid negation in the test. For example, rephrase: if (x != y) diff(); else same(); as: if (x == y) same(); else diff(); Most if (x != y) cases without an else are often return cases, so consistent use of this rule makes the code easier to read.

Given the above the prevailing view is that this is a MINOR rule infringement rather than a MAJOR.

9.4 Unused local variable/private field

Where false positives are generated by the code scanning tool these will be ignored.

However on a case by case basis for other scenarios of this an exception review is required between Customer Salesforce IT team and the project in question.

Where false positives are generated by the code scanning tool these will be ignored.

However on a case by case basis for other scenarios of this an exception review is required between Customer Salesforce IT team and the project in question.

9.5 No Magic Number

A magic number is a number that comes out of nowhere, and is directly used in a statement. Magic numbers are often used, for instance to limit the number of iterations of a loop, to test the value of a property, etc.

That is why magic numbers must be demystified by first being assigned to clearly named variables before being used.

-1, 0 and 1 are not considered magic numbers.

Split this 263 characters long line (which is greater than 240 authorized).

Having to scroll horizontally makes it harder to get a quick overview and understanding of any piece of code.

Add the missing "else" clause.

This rule applies whenever an if statement is followed by one or more else if statements; the final else if should be followed by an else statement.

The requirement for a final else statement is defensive programming.

The else statement should either take appropriate action or contain a suitable comment as to why no action is taken. This is consistent with the requirement to have a final default clause in a switch statement.

Remove the unused function parameter "event"

Unused parameters are misleading. Whatever the values passed to such parameters, the behavior will be the same.

Unexpected empty method 'helperMethod'.

There are several reasons for a function not to have a function body:

- It is an unintentional omission, and should be fixed to prevent an unexpected behavior in production.
- It is not yet, or never will be, supported. In this case an exception should be thrown in languages where that mechanism is available.
- The method is an intentionally-blank override. In this case a nested comment should explain the reason for the blank override.

10.0 LWC Best Practices

Lightning Web Components (LWC) are **a user interface (UI) framework that Salesforce Developers use to create customized pages and functions on the Salesforce platform**. LWCs use a standardized JavaScript framework, HTML, and CSS, without a third-party framework.

Following are the best practices for Lightning Web Components:

- **Use Cache data:**

Lightning web components offer two ways to cache data. The first is to use Lightning Data Service, which automatically handles security for you and you do not have to write any Apex classes – especially platform required test classes.

If you must use Apex, then you can simply mark your methods as cacheable using the AuraEnabled annotation.

Example :

```
@AuraEnabled(cacheable=true)

public static myCacheableMethod() {}
```

- **Use Conditional Rendering:**

Conditional rendering means that portions of a component will not be rendered until a condition is met.

For example, let's assume you had a component that displayed a list of widget data. You would not want the list to be built unless there was data available. If there was no data, then the user would see a message telling them there are no widgets.

Example :

```
<div if:true{widgets}>

  <template for:each={widgets} for:item="widget">

     <li key={widget.Id}>{widget.Name}</li>

  </template>

</div>

<div if:false{widgets.length}>

  There are no widgets available

</div>
```

- **Use Pagination with List:**

Pagination is used to split a huge content in the tables into smaller parts. By default, Pagination provides Previous button, Next button with page numbers and total records.

Many Salesforce orgs have custom objects that contain hundreds, thousands, if not millions of records. And rendering a huge list of data has the potential for causing lots of performance problems. To prevent these lists from getting out of control, introduce a pagination component.

- **Use Base Lightning Web Components:**

There are now **94** base Lightning web components to choose from. They cover everything from a simple input box to a complex record form.

These components not only offer the CSS from the Salesforce Lightning Design System (SLDS), but they offer a performance advantage. These components are already rendered at the client-side, so they do not require additional download processing.

- **Use SLDS Icons and Styling:**

In Salesforce Lightning Design System (SLDS), there are 100's of optimized icons. Using your own customized icons can result in low render quality and resolution, so take advantage to these readily available icons in Lightning Design System Website.

Example :

<lightning-icon icon-name="utility:warning" alternative-text="Warning!" title="Warning"></lightning-icon>

<lightning-icon icon-name="utility:error" alternative-text="Error!" title="Error"></lightning-icon>

Some other best practices to consider :

- **Naming Convention for Lightning Web Components :**
  - Html file : Use **camel case** (Eg : helloWorld) to name your component and use **kebab-case** (Eg : c-hello-world-form) to reference a component in the markup
  - JavaScript File : Java Script Class name should be in **PascalCase** (Eg : HelloWorld)
  - Bundle Component : Use **camelCase** (Eg : helloWorld).

- **Calling Apex from LWC :**

There are 2 ways to call Apex class :

  - Imperatively
  - Wire - Wire a Property, Wire a function

As per Lightning component best practices use @wire over imperative method invocation. @wire fits nicely in the overall Lightning Web Component reactive architecture. Salesforce is building some performance enhancement features that are only available with @wire. But there are a few use cases, that require you to use imperative Apex.

- **Lightning Data Service:**

As per LWC best practice use Lightning Data Service functions to create Record and delete a record over invoking Apex methods. Yes there are some use cases where you need to multiple records then we can use Apex methods.

Lightning Data Service is built on top of User Interface API. UI API is designed to make it easy to build Salesforce UI. UI API gives you data and metadata in a single response

Give preference to user interface form-type in below order.

- lightning-record-form : It is the fastest/most productive way to build a form.
- lightning-record-view-form : If you need more control over the layout, want to handle events on individual input fields, or need to execute pre-submission.
- @wire(getRecord) : If you need even more control over the UI, or if you need to access data without a UI.

- **Event in LWC:**

There are typically 3 approaches for communication between the components using events.

- Communication using Method in LWC (Parent to Child)
- Custom Event Communication in Lightning Web Component (Child to Parent)
- Publish Subscriber model in Lightning Web Component Or Lightning Message Service (Two components which don't have a direct relation)

Here is some recommendation for DOM Event.

- No uppercase letters
- No Spaces
- use underscores to separate words
- Don't prefix your event name with the string "on".

- **How to Debug LWC :**

Use Chrome's pretty JS setting to see unminified JavaScript and Debug Proxy Values for Data.

Enable Debug mode

- It gives unminified Javascript
- Console warnings
- Pretty data structure

Caution – it reduces the performance of Salesforce, make sure it's disabled in production

- **Be careful while naming variables:**

Do not start a property name with 'on', 'aria', 'data'. Also do not use reserved keywords like 'slot', 'part', 'is'.

- **LWC deals with Case Sensitivity:**

While writing codes we need to be careful regarding case sensitivity.

For example, ‘message’ and ‘MESSAGE’ are two different properties.

Similarly, while accessing Salesforce object fields, we need to mention the exact api name. For example, {account.Name} will return data but {account.name} won’t.

- **Always use dynamic import in LWC:**

If you are importing any custom field in LWC, always use dynamic import. Because static imports are not referenced during field deletion, but dynamic imports are referenced.

Example:

import CUSTOM_FIELD1 from '@salesforce/schema/Account.CustomField1__c';

- **Use Getters instead of Expressions:**

To compute a value for a property, use a JavaScript getter. Getters are more powerful than expressions because these are JavaScript functions and enable unit testing, which reduces bugs and increases fun.

Example:

This sample <c-todo-item> component converts a string to uppercase.

```
<!-- todoItem.html -->

<template>

    {itemName}

</template>
```

The property value is provided to the template via the getter.

```
// todoItem.js

import { LightningElement, api } from 'lwc';

export default class TodoItem extends LightningElement {

    _uppercaseItemName;

    @api

    get itemName() {

        return this._uppercaseItemName;
```

```
    }
    set itemName(value) {
      this._uppercaseItemName = value.toUpperCase();
    }
}
```

- **Make use of multiple templates using render():**

If you want to render different UI based on certain conditions, render() function would be helpful in that case. You can import different html files in order to show different UI and from the render function choose which one to show and render the same using template if:true.

Example:

```
<template if:true={accounts.data}>
    <template for:each={accounts.data} for:item="acc">
        <p key={acc.Id}>{acc.Name}</p>
    </template>
</template>
<template if:true={accounts.error}>
    {accounts.error}
</template>
```

- **Pass only primitive data in custom event:**

As one of the best practices, pass only primitive data types in custom events. JavaScript passes all data types by reference except for primitive data types. So, any listener can mutate the actual object which is not a good practice.

If at all we need to pass an object, we need to copy the data to a new object before sending so that it does not get changed by any of the listeners.

- **Specify fields instead of layout in lightning-record-form:**

To improve performance, specify specific fields instead of layout in lightning-record-form. If we specify layout, the component must handle receiving every field that is assigned to the layout for the context user.

- **Use getRecord over getRecordUi:**

getRecord returns record data while getRecordUi returns record data in addition to layout information and object metadata.

- **Data and Error property names are fixed in wire service:**

In case of wire service, the results of the apex method are provisioned to a function via an object with two property ‘data’ and ‘error’.

We cannot put other name for these two variables since it is as per the standard. So, we need to refer these two properties as ‘data’ and ‘error’ only. Any attempt to use some other name will fail silently and would impact the output.

- **Refresh the cache using refreshApex() and getRecordNotifyChange():**

To get the updated data on the UI after some operation, make use of the refreshApex() and getRecordNotifyChange() methods for wire and imperative calls respectively.

- **Use targetConfig for configuring input to LWC:**

To set the input in LWC from lightning app builder/flow/community make use of the targetConfig property.

- **Include error handling in the code logic:**

We should consider the error scenario as well and notify the user with a user-friendly error message. For this, we can make use of toast message to show high level detail to user on what went wrong.

- **Avoid using hardcoding, use custom labels:**

Like apex, avoid using hardcoding in LWC. Keep the text inside a custom label and configure as and when required without making any change to existing code.

The below table gives a quick summary of LWC best practices should be followed

S.no	Topic	Reference's
1	Tips & Trick for Lightning Web Components	https://www.apexhours.com/20-tips-for-lightning-web-components/
2	Lightning Web Components Performance Best Practices	https://developer.salesforce.com/blogs/2020/06/lightning-web-components-performance-best-practices

3	Lightning Web Components (LWC) Best Practice 1. LWC component Bundle naming convention 2. Calling Apex From LWC : Wire VsImperatively Wire Property Vs Wire function 3. Lightning Data Service (LDS) 4. Event in LWC 5. Streaming API, Platform Event, Change Data Capture 6. How to debug LWC 7. Use Storable Action 8. Build reusable Lightning Web component 9. Styling Reusable Components 10. LWC Best Practices	https://www.apexhours.com/lightning-web-components-lwc-part-2/#4-event-in-lwc
4	Lightning component library	https://developer.salesforce.com/docs/component-library/documentation/en/lwc/lwc.get_started_component_library
5	Secure development with LWC 1. Security with Lightning Locker 2. Lightning Web Security 3. Content Security Policy Overview	https://developer.salesforce.com/docs/component-library/documentation/en/lwc/lwc.security_intro
6	Access Salesforce Resources: 1. Access Static Resources 2. Access Content Asset Files 3. Use SVG Resources 4. Access Labels 5. Get Information about current logged-in user 6. Check Permissions	https://developer.salesforce.com/docs/component-library/documentation/en/lwc/lwc.create_global_value_providers
7	Design Patterns and Best Practices to build reusable Lightning Web Components	https://www.apexhours.com/design-patterns-and-best-practices-to-build-reusable-lightning-web-components/
8	Top 5 best practices for LWC	https://saramorgan.net/2022/09/27/top-five-best-practices-for-lightning-web-components/
9	Jest for LWC (A test class for LWC)	https://www.salesforceben.com/how-to-use-jest-for-lightning-web-component-testing/
10	Component Lifecycle	https://developer.salesforce.com/docs/component-library/documentation/en/lwc/create_lifecycle_hooks_intro
11	View Debug Information for Your Wired Properties	https://help.salesforce.com/s/articleView?id=release-notes.rn_lwc_wire_debug.htm&release=242&type=5
12	Migrate Aura Components to Lightning Web Components	https://developer.salesforce.com/docs/component-library/documentation/en/lwc/lwc.migrate_introduction

13	Latest features in LWC	https://developer.salesforce.com/blogs/2023/01/lwc-enhancements-for-developers-learn-moar-spring-23
14	Guidelines to work with Salesforce data	https://developer.salesforce.com/docs/component-library/documentation/en/lwc/lwc.data_guidelines
15	Advantages of using LWC	https://www.apexhours.com/lightning-web-components/#advantages-of-using-lwc
16	Guidelines to Call APIs from JavaScript	https://developer.salesforce.com/docs/component-library/documentation/en/lwc/lwc.js_api_calls
	Examples 1. Using Apex Callout 2. Using JavaScript	https://medium.com/tech-force/callout-from-lightning-web-component-4e848b72365a
17	LWC Cheat Sheet	https://santanuboral.blogspot.com/2019/07/salesforce-lwc-cheatsheet.html
18	Pagination using Salesforce Lightning Web Components	https://santanuboral.blogspot.com/2019/09/pagination-using-LWC.html
19	Documentation Changelog	https://developer.salesforce.com/docs/component-library/documentation/en/lwc/lwc.get_started_change_log
20	Optimize Data Retrieval	https://trailhead.salesforce.com/content/learn/modules/best-practices-in-lightning-web-components/work-with-data-lwc
21	Use Progressive Disclosure and Conditional Rendering	https://trailhead.salesforce.com/content/learn/modules/best-practices-in-lightning-web-components/use-progressive-disclosure-and-conditional-rendering
22	Events Best Practices	https://developer.salesforce.com/docs/component-library/documentation/en/lwc/events_best_practices

11.0 Integration (Web Service Callouts):

Salesforce Integration is the process of bringing two or more systems together, which allows you to streamline separate processes. Rest API is best for web and mobile applications.

These are the best practices to design REST API:

- **REST API must accept and respond with JSON:**

It is a common practice that APIs should accept JSON requests as the payload and send responses back. JSON is a open and standardized format for data transfer. It is derived from JavaScript in a way to encode and decode JSON via the Fetch API or another HTTP client. Moreover, server-side technologies have libraries that can decode JSON without any hassle.

- **Go With Error Status Codes:**

Over 100 status codes have already been built by HTTP. It is a boon for developers to use status codes in their REST API design. With the status codes, developers can instantly identify the issue, which reduces the time of writing parsers to address all the different types of errors. There's a status code for everything – from finding out the cause of a denied session to locating the missing resource. Developers can quickly implement routines for managing numerous errors based on status codes.

- **Use Nouns Instead of Verbs in Endpoints:**

When you're designing a REST API, you should not use verbs in the endpoint paths. The endpoints should use nouns, signifying what each of them does.

So, for example, an endpoint should not look like this:

https://mysite.com/getPosts or https://mysite.com/createPost

Instead, it should be something like this: **https://mysite.com/posts**

In short, you should let the HTTP verbs handle what the endpoints do. So, GET would retrieve data, POST will create data, PUT will update data, and DELETE will get rid of the data.

- **Use Plural Nouns to name a Collection:**

When you must develop the collection in REST API, just go with plural nouns. It makes it easier for humans to understand the meaning of collection without opening it. Let's go through this example:

GET /cars/123

POST /cars

GET /cars

It is clear from the example that ‘car’ is referred to as number 123 from the entire list of "cars". The usage of a plural noun is merely indicating that this is a collection of different cars. Now, look at one another example:

GET /car/123

POST /car

GET /car

This example doesn’t clearly show whether there is more than one car in the system or not. For a human reader, it might be challenging to understand, as well.

- **Well Compiled Documentation:**

Documentation is one of the important but highly ignored aspects of a REST API structure. The documentation is the first point in the hands of customers to understand the product and critical deciding factor whether to use it or not. One good documentation is neatly presented in a proper flow to make an API development process quicker.

It is a simple principle – the faster developers understand your API, the faster they start using it. Your API documentation must be compiled with precision. It must include all the relevant information such as the endpoint and compatible methods, different parameter options, numerous types of data, and so on. The documentation should be so robust that it can easily walk a new user through your API design.

- **Return Error details in the Response Body:**

It is convenient for the API endpoint to return error details in the JSON or response body to help a user with debugging. If you can explicitly include the affected field in error, this will be special kudos to you.

```
{

  "error": "Invalid payload.",

  "detail": {  "surname": "This field is required."  }

}
```

- **Use Resource Nesting:**

Resource objectives always contain some sort of functional hierarchy or are interlinked to one another. However, it is still ideal to limit the nesting to one level in the REST API. Too many nested levels can lose their elegant appeal. If you take a case of the online store into consideration, we can see “users” and “orders” are part of stores. Orders belong to some user; therefore, the endpoint structure looks like:

/users - // list all users

/users/123 - // specific user

/users/123/orders - // list of orders that belong to a specific user

/users/123/orders/0001 - // specific order of a specific users order list

- **Use SSL for Security:**

SSL stands for secure socket layer. It is crucial for security in REST API design. This will secure your API and make it less vulnerable to malicious attacks.

Other security measures you should take into consideration include: making the communication between server and client private and ensuring that anyone consuming the API doesn't get more than what they request.

The clear difference between the URL of a REST API that runs over SSL and the one which does not is the "s" in HTTP:

https://mysite.com/posts runs on SSL.

http://mysite.com/posts does not run on SSL.

- **Secure Your API:**

It is a favorite pastime for hackers to use automated scripts to attack your API server. Thus, your API needs to follow proactive security measures to run operations while safeguarding your sensitive data smoothly. Foremost, your API must have an HTTP Strict Transport Security (HSTS) policy. Up next, you should secure your network from middleman attacks, protocol downgrade attacks, session hijacking, etc., Just use all the relevant security standards to the security of your API.

Perfectly designed REST API stays on the positive side of technical constraints along with taking user experience-based solutions. API is a part of the business strategy; it is a marketing tool for the organization; thus, it is essential to execute APIs in the right manner. That's because unstructured API is a liability rather than an asset.

- **Use Filtering, Sorting, and Pagination to Retrieve the Data Requested:**

Sometimes, an API's database can get incredibly large. If this happens, retrieving data from such a database could be very slow.

Filtering, sorting, and pagination are all actions that can be performed on the collection of a REST API. This lets it only retrieve, sort, and arrange the necessary data into pages so the server doesn't get too occupied with requests.

An example of a filtered endpoint is the one below:

https://mysite.com/posts?tags=javascript

This endpoint will fetch any post that has a tag of JavaScript.

- **Be Clear with Versioning:**

REST APIs should have different versions, so you don’t force clients (users) to migrate to new versions. This might even break the application if you're not careful.

One of the commonest versioning systems in web development is semantic versioning.

An example of semantic versioning is 1.0.0, 2.1.2, and 3.3.4. The first number represents the major version, the second number represents the minor version, and the third represents the patch version.

Many RESTful APIs from tech giants and individuals usually comes like this:

https://mysite.com/v1/ for version 1

https://mysite.com/v2 for version 2

12.0 Most Common Error Reference Documentation

12.1 Most Common QRST Audit Report Findings and Solutions

- As per pureforce methodology SFDC recommended best practices needs to be followed, however it is observed that SFDC recommended best practices not followed in Test Class preparation

  1. Test classes are generated but not reviewed for Assert Statements

  2. Write mock callout classes.

  3. Avoid try catch block in test classes.

  4. Avoid standard exception test in system assert method in test class

  5. Add dummy values for test class at the time of initializing o variable

  6. Observed that Assert Statements missing in test classes

  7. Method description missing in sampled test class

  8. There should be a proper annotation using in some classes however it is observed that proper annotation using in few classes is missing (Note - In few classes @testMethod keyword is used, now it is deprecated. Use the @isTest annotation on classes and methods instead.)

  9. Assert equals missing in test class

  For all the above issues refer **Unit test coverage section or Comments** for guidance on the best practice

- As per pureforce methodology, mock test factory classes not used for preparing the test data for unit testing.

  As per SFDC documentation test factories are recommended. It is also mentioned in Unit test Coverage section.

- SFDC recommended best practices not followed in Apex classes/triggers/VF pages etc. Few examples,

  1. All the commented code should not be moved to production.
     Refer **General Apex coding best practices section** for guidance on the best practice

  2. Add Descriptions in custom fields or any custom element
     Refer **configuration standard section** for guidance on the best practice

  3. System.debug statements not removed before deploying to QA and other higher sandbox
     Refer **General Apex coding best practices** for guidance on the best practice

  4. Not using 'Custom Labels' for displaying error messages
     Refer **Static Variable** for guidance on the best practice

5. Mod log not available for the legacy code with respect to top code commenting
   Refer **Comments section** for guidance on the best practice

6. Change history block should be present in Apex Classes however it is observed that In few apex classes, Change history block is missing
   Refer **Comments section** for guidance on the best practice

7. No exception logger leveraged
   Refer **Exception Handler Basics and ExceptionLoggingHelper** for guidance on the best practice

8. Missing Error logging framework in Apex class
9. Hard coded text are used in Apex batch classes
10. Missing Exception handling in flow
11. Custom metadata to control flow trigger
    Refer **Apex Standards and Best Practices** for item # 9 through #12

12. Dynamic apex query not leveraged rather than applying same query in if & else block
    Refer **SOQL Injection** for guidance on the best practice

13. It is advice that hard coding should not be used however it is observed that hard code is used in JavaScript
    Mentioned in multiple areas in **coding standard document**

- History (who created, when with comments) not tracked in the class

  Refer to **Apex Classes sub section of Apex Standards and Best Practices section** to know the best practice to be followed

- DML operations not validated using isempty() / != null method in trigger handler which will improve the performance and computing time

  Refer to **Apex Standards and Best Practices section** to know the best practice to be followed

- One record-change process per object - Critical and Old objects have more than 1 process per object

  Should be thought through properly during design of the application

- Coding standard not followed- team follows specific trigger framework for trigger handler code, but there is some existing code which are outside of their framework.

  Should be thought through properly during design of the application

- SOQL maintained in the same class, which is not the recommended best practice.

  SOQL should be moved to selector classes for the projects which implements separation of concerns /Apex Enterprise Patterns, refer **Separation of Concerns - Apex Enterprise patterns**

- Project team should regularly perform PMD checks in VSCode prior to pushing to pipeline however it is observed that they don't perform PMD checks in VSCode prior to pushing to pipeline

Refer section **Cognizant Code Sense** for guidance

- Cognizant defined coding standard not – Multiple Flows used for same object along trigger Refer section **Configuration Standards -> Workflows and Approval Processes for guidance**

12.2 Most common Code Quality Errors

The below table provides a list of most common errors of historical scans and accounts for **80% of the code scan anomaly and requires corrective effort and results in rework and unwanted client attention.** So care should be taken during development cycle to avoid such known errors. This is a representative and output for Checkmarx scans.

Note: Code Sense, Checkmarx, CodeScan Shield, SonarQube are the recommended tools from CoE for Salesforce Apex & JS coding quality, however Code Sense is preferred tool over the other tools because of the features it provides. The rules and error patterns remain valid for all the scanning tools and the table below will help tech leads/PMs to manage code quality in a more efficient manner

Error Code	Description	Link
FLS_Create	FLS is defined as the Field Level Permission that defines access to objects and fields for users. FLS can be controlled in several ways, e.g., through sharing settings in the organization, where we can set the default access to public or private. We can organize sharing by the user, role, etc. The Lightning component and Visualforce page data depend on the Apex class controller, and the Apex class does not require standard sharing. Apex class enforcement permissions are required so that no one can access these records and data that the user cannot access. Therefore, we need to use the keyword “share” in the class definition. Example: Insecure: public class ClassName{ .. } Secure: public with secure class ClassName{ .. } Risk What might happen A malicious user could access other users’ information. By requesting information directly, such as by an account number, authorization may be bypassed and the attacker could steal confidential or restricted information (for example, a bank account balance), using a direct object reference. Cause How does it happen The application provides user information without filtering by user ID. For example, it may provide information solely by a submitted account ID. The application	https://blog.cloudanalogy.com/checkmarx-salesforce-code-review-tool/

	concatenates the user input directly into the SQL query string, without any additional filtering. The application also does not perform any validation on the input, nor constrain it to a pre-computed list of acceptable values. General Recommendations How to avoid it Generic Guidance: · Enforce authorization checks before providing any access to sensitive data, including the specific object reference. · Explicitly block access to any unauthorized data, especially to other users' data. · If possible, avoid allowing the user to request arbitrary data by simply sending a record ID. For example, instead of having the user send an account ID, the application should look up the account ID for the current authenticated user session. Specific Mitigation: · Do not concatenate user input directly into SQL queries. · Include a user-specific identifier as a filter in the WHERE clause of the SQL query. · Map the user input to an indirect reference, e.g. via a prepared list of allowable values.	
FLS_Update	FLS is defined as the Field Level Permission that defines access to objects and fields for users. FLS can be controlled in several ways, e.g., through sharing settings in the organization, where we can set the default access to public or private. We can organize sharing by the user, role, etc. The Lightning component and Visualforce page data depend on the Apex class controller, and the Apex class does not require standard sharing. Apex class enforcement permissions are required so that no one can access these records and data that the user cannot access. Therefore, we need to use the keyword "share" in the class definition. Example: Insecure: public class ClassName{ .. } Secure: public with secure class ClassName{ .. } Risk What might happen A malicious user could access other users' information. By requesting information directly, such as by an account number, authorization may be bypassed and the attacker could steal confidential or restricted information (for example, a bank account balance), using a direct object reference. Cause How does it happen The application provides user information without filtering by user ID. For example, it may provide information solely by a submitted account ID. The application concatenates the user input directly into the	https://blog.cloudanalogy.com/checkmarx-salesforce-code-review-tool/

	SQL query string, without any additional filtering. The application also does not perform any validation on the input, nor constrain it to a pre-computed list of acceptable values. General Recommendations How to avoid it Generic Guidance: · Enforce authorization checks before providing any access to sensitive data, including the specific object reference. · Explicitly block access to any unauthorized data, especially to other users' data. · If possible, avoid allowing the user to request arbitrary data by simply sending a record ID. For example, instead of having the user send an account ID, the application should look up the account ID for the current authenticated user session. Specific Mitigation: · Do not concatenate user input directly into SQL queries. · Include a user-specific identifier as a filter in the WHERE clause of the SQL query. · Map the user input to an indirect reference, e.g. via a prepared list of allowable values.	
Second_Order_SOQL_SOSL_Injection	SOQL/SOSL injection is a common application security bug that results from insecure database queries against user-supplied data. This can happen when an unexpected value is given as input, and that input value is used in every SOQL/SOSL query, after which the value or purpose of the whole query changes. Solution : Use input string with String.escapeSingleQuotes() Example: Insecure: Contact conObj = [SELECT Name,Id FROM Contact WHERE LastName =: stringInput LIMIT 1]; Secure: Contact conObj = [SELECT Name,Id FROM Contact WHERE LastName =: String.escapeSingleQuotes(stringInput) LIMIT 1]; Risk What might happen An attacker could directly access all of the system's data. Using simple tools and text editing, the attacker would be able to steal any sensitive information stored by the system (such as personal user details or credit cards), and possibly change or erase existing data. Cause How does it happen The application communicates with its database by sending a textual SOQL query. The application creates the query by simply concatenating strings including the user's input.	https://blog.cloudanalogy.com/checkmarx-salesforce-code-review-tool/

	Since the user input is neither checked for data type validity nor subsequently sanitized, the input could contain SOQL commands that would be interpreted as such by the database. General Recommendations How to avoid it 1. Validate all input, regardless of source. Validation should be based on a whitelist: accept only data fitting a specified structure, rather than reject bad patterns. Check for: o Data type o Size o Range o Format o Expected values.	
FLS_Update_Partial	FLS is defined as the Field Level Permission that defines access to objects and fields for users. FLS can be controlled in several ways, e.g., through sharing settings in the organization, where we can set the default access to public or private. We can organize sharing by the user, role, etc. The Lightning component and Visualforce page data depend on the Apex class controller, and the Apex class does not require standard sharing. Apex class enforcement permissions are required so that no one can access these records and data that the user cannot access. Therefore, we need to use the keyword “share” in the class definition. Example: Insecure: public class ClassName{ .. } Secure: public with secure class ClassName{ .. } Risk What might happen A malicious user could access other users’ information. By requesting information directly, such as by an account number, authorization may be bypassed and the attacker could steal confidential or restricted information (for example, a bank account balance), using a direct object reference. Cause How does it happen The application provides user information without filtering by user ID. For example, it may provide information solely by a submitted account ID. The application concatenates the user input directly into the SQL query string, without any additional filtering. The application also does not perform any validation on the input, nor constrain it to a pre-computed list of acceptable values. General Recommendations How to avoid it Generic Guidance:	https://blog.cloudanalogy.com/checkmarx-salesforce-code-review-tool/

	· Enforce authorization checks before providing any access to sensitive data, including the specific object reference. · Explicitly block access to any unauthorized data, especially to other users' data. · If possible, avoid allowing the user to request arbitrary data by simply sending a record ID. For example, instead of having the user send an account ID, the application should look up the account ID for the current authenticated user session. Specific Mitigation: · Do not concatenate user input directly into SQL queries. · Include a user-specific identifier as a filter in the WHERE clause of the SQL query. · Map the user input to an indirect reference, e.g. via a prepared list of allowable values.	
Parameter_Tampering	Parameter tampering is a simple attack targeting the application business logic. This attack takes advantage of the fact that many programmers rely on hidden or fixed fields (such as a hidden tag in a form or a parameter in a URL) as the only security measure for certain operations. Risk What might happen A malicious user could access other users' information. By requesting information directly, such as by an account number, authorization may be bypassed and the attacker could steal confidential or restricted information (for example, a bank account balance), using a direct object reference. Cause How does it happen The application provides user information without filtering by user ID. For example, it may provide information solely by a submitted account ID. The application concatenates the user input directly into the SQL query string, without any additional filtering. The application also does not perform any validation on the input, nor constrain it to a pre-computed list of acceptable values. General Recommendations How to avoid it Generic Guidance: · Enforce authorization checks before providing any access to sensitive data, including the specific object reference. · Explicitly block access to any unauthorized data, especially to other users' data. · If possible, avoid allowing the user to request arbitrary data by simply sending a record ID. For example, instead of having the user send an account ID, the application should look up the account ID for the current authenticated user session. Specific Mitigation: · Do not concatenate user input directly into SQL queries. · Include a user-specific identifier as a filter in the WHERE clause of the SQL query.	

	· Map the user input to an indirect reference, e.g. via a prepared list of allowable values.	
FLS_Create_Partial	FLS is defined as the Field Level Permission that defines access to objects and fields for users. FLS can be controlled in several ways, e.g., through sharing settings in the organization, where we can set the default access to public or private. We can organize sharing by the user, role, etc. The Lightning component and Visualforce page data depend on the Apex class controller, and the Apex class does not require standard sharing. Apex class enforcement permissions are required so that no one can access these records and data that the user cannot access. Therefore, we need to use the keyword "share" in the class definition. Example: Insecure: public class ClassName{ .. } Secure: public with secure class ClassName{ .. } Risk What might happen A malicious user could access other users' information. By requesting information directly, such as by an account number, authorization may be bypassed and the attacker could steal confidential or restricted information (for example, a bank account balance), using a direct object reference. Cause How does it happen The application provides user information without filtering by user ID. For example, it may provide information solely by a submitted account ID. The application concatenates the user input directly into the SQL query string, without any additional filtering. The application also does not perform any validation on the input, nor constrain it to a pre-computed list of acceptable values. General Recommendations How to avoid it Generic Guidance: · Enforce authorization checks before providing any access to sensitive data, including the specific object reference. · Explicitly block access to any unauthorized data, especially to other users' data. · If possible, avoid allowing the user to request arbitrary data by simply sending a record ID. For example, instead of having the user send an account ID, the application should look up the account ID for the current authenticated user session. Specific Mitigation: · Do not concatenate user input directly into SQL queries.	https://blog.cloudanalogy.com/checkmarx-salesforce-code-review-tool/

	· Include a user-specific identifier as a filter in the WHERE clause of the SQL query. · Map the user input to an indirect reference, e.g. via a prepared list of allowable values.	
Unsafe_Use_Of_Target_blank	Developers have been frequently using this attribute to open a new webpage. But this attribute, though looks pretty simple, can create a major security threat to your application. The threat associated is called Reverse Tabnabbing. The issue is the webpage that we are linking our existing page to gains a partial access to the linking page, or in other words, the target page or url gains a partial access to our parent page from where the user is redirected to a new url. This happens through the window.opener object of Javascript. The attacker can change the window.opener.location to some malicious page and also the parent page. In case the parent page has the same look and feel of the user intended page, he might end up sharing credentials or secure information assuming that the webpage is secure. To prevent pages from misusing window.opener property, we use rel="noopener". This ensures that window.opener is assigned to NULL. This works in Chrome 49 and above, Opera 36 and above, Firefox 52 and above, Desktop Safari 10.1+ and iOS Safari 10.3+. Though for older browsers, rel="noreferrer" works pretty fine. So, as a combination, we can use rel="noopener noreferrer" attribute. Also, remember, whenever we open a new window using window.open() property, we're also vulnerable to this, so, its always advisable to reset the "opener" property.	https://hackernoon.com/unsafe-use-of-target_blank-39413ycf
Sharing_With_Controller	controller is not declared using the with sharing keywords, and therefore isn't using the system's sharing model. Risk What might happen Without proper restrictions it is possible to leak data and expose restricted information to a user without the proper clearance. Cause How does it happen The force plafform makes extensive use of data sharing rules. Each object can have unique permissions for which users and profiles can read, create, edit and delete. These restrictions are encforced when using all standard controllers. When using a custom Apex class, the built-in profile permissions and field-level security restrictions are not respected during execution.	https://salesforce.stackexchange.com/questions/132613/sharing-with-controller-vulnerability

	The default behavior is that an Apex class has the ability to read and update all data with 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 profile-based permissions, field-level security or organization-wide defaults. This is particulary true for Visualforce pages. General Recommendations How to avoid it 1. Use the with sharing keywords when declaring a class to enforce the sharing rules that apply to the current user. 2. Use the without sharing keywords when declaring a class to ensure that the sharing rules for the current user are not enforced.	
Bulkify_Apex_Methods_Using_Collections_In_Methods	This is probably a coding practice/style warning which is not relevant to a security review (but that is worth addressing to make your code perform better); Checkmarx never flags all instances of the same issue (if you have many of them) because the report you get from Salesforce Checkmarx execution is truncated and only shows the first n instances of any given problem - you need to either run Checkmarx locally with your own license or use PMD. Checkmarx is a static analysis tool and cannot know what volumes of data get passed to any given method when those methods receive collections or query collections of data. It can only look for patterns in the code itself and really has no understanding of data. Risk What might happen If code exceeds a governor limit, the associated governor issues a runtime exception that cannot be handled, which terminates the request. Cause How does it happen The issue occurs when a class structured to receive a single object in the method signature, and since the method performs a DML statement, it raises the risk of hitting a governor exception if that apex method is called within a loop from other apex code. General Recommendations How to avoid it Make sure any utility of helper methods are efficiently written to handle collections of records. This will avoid unnecessarily executing inefficient queries and DML operations.	https://salesforce.stackexchange.com/questions/349113/checkmarx-bulkify-apex-methods-using-collections-in-methods?rq=1

Sharing	FLS is defined as the Field Level Permission that defines access to objects and fields for users. FLS can be controlled in several ways, e.g., through sharing settings in the organization, where we can set the default access to public or private. We can organize sharing by the user, role, etc. The Lightning component and Visualforce page data depend on the Apex class controller, and the Apex class does not require standard sharing. Apex class enforcement permissions are required so that no one can access these records and data that the user cannot access. Therefore, we need to use the keyword “share” in the class definition. Example: Insecure: public class ClassName{ .. } Secure: public with secure class ClassName{ .. } Risk What might happen Without proper restrictions it is possible to leak data and expose restricted information to a user without the proper clearance. Cause How does it happen The force plafform makes extensive use of data sharing rules. Each object can have unique permissions for which users and profiles can read, create, edit and delete. These restrictions are encforced when using all standard controllers. When using a custom Apex class, the built-in profile permissions and field-level security restrictions are not respected during execution. The default behavior is that an Apex class has the ability to read and update all data with 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 profile-based permissions, field-level security or organization-wide defaults. This is particulary true for Visualforce pages. General Recommendations How to avoid it 1. Use the with sharing keywords when declaring a class to enforce the sharing rules that apply to the current user. 2. Use the without sharing keywords when declaring a class to ensure that the sharing rules for the current user are not enforced.	https://blog.cloudanalogy.com/checkmarx-salesforce-code-review-tool/

SOQL_SOSL_Injection	SOQL/SOSL injection is a common application security bug that results from insecure database queries against user-supplied data. This can happen when an unexpected value is given as input, and that input value is used in every SOQL/SOSL query, after which the value or purpose of the whole query changes. Solution : Use input string with String.escapeSingleQuotes() Example: Insecure: Contact conObj = [SELECT Name,Id FROM Contact WHERE LastName =: stringInput LIMIT 1]; Secure: Contact conObj = [SELECT Name,Id FROM Contact WHERE LastName =: String.escapeSingleQuotes(stringInput) LIMIT 1];	https://blog.cloudanalogy.com/checkmarx-salesforce-code-review-tool/
Queries_With_No_Where_Or_Limit_Clause	SOQL Query Optimization Always limit the number of records that to be returning by the query by using the LIMIT keyword. Never use fields that are not indexed in the WHERE clause. The syntax for LIMIT is: SELECT fieldList FROM objectType [WHERE conditionExpression] [LIMIT numberOfRows] SELECT Name FROM Account WHERE Industry = 'Media' LIMIT 125. SELECT MAX(CreatedDate) FROM Account LIMIT 1.	

CRUD_Delete	The CRUD issue occurs when a user tries to find, insert, update, update, or delete any record without confirming whether the user has permission to operate or not.
	
	Solution :
	
	We can use any of the different tests available in Apex, such as B. isAccessible(), isUpdateable(), isQueryable(), isCreateable(), and so on. on fields as well as on objects.
	
	Example :
	
	Query:
	
	If(Schema.sObjectType.Contact.isQueryable()){
	
	Contact conObj = [SELECT Name,Id FROM Contact WHERE LastName =: String.escapeSingleQuotes(stringInput) LIMIT 1];
	
	}
	
	Insertion:
	
	Contact conObj = new Contact();
	
	if (Schema.sObjectType.Contact.fields.LastName.isCreateable()){
		https://blog.cloudanalogy.com/checkmarx-salesforce-code-review-tool/

	conObj.Title = chainDetail[0].Name;
	
	}
	
	if (Schema.sObjectType. Contact.fields.Email.isCreateable()){
	
	conObj.ParentId = chainDetail[0].Agenthub__Deal_Agreed__c;
	
	}
	
	if (Schema.sObjectType.Contact.isCreateable() && conObj !=null){
	
	Insert conObj;
	
	}
	
	Deletion:
	
	if (Schema.sObjectType.Contact.isDeletable() && conObj !=null){
	
	Delete conObj;
	
	}
	
	Update:
	
	Contact conObj = new Contact();
	
	conObj.Id = id;
	
	if (Schema.sObjectType. Contact.fields.Email.isUpdateable()){
	
	conObj.Email = 'example@gmail.com';
	
	}
	
	if (Schema.sObjectType.Contact. isUpdateable () && conObj !=null){
	
	Update conObj;
	
	}
	
	Risk
	What might happen
	A malicious user could access other users' information. By requesting information directly, such as by an account number, authorization may be bypassed and the attacker could steal confidential or restricted information (for example, a bank account balance), using a direct object reference.
	Cause
	How does it happen
	The application provides user information without filtering by user	

	ID. For example, it may provide information solely by a submitted account ID. The application concatenates the user input directly into the SQL query string, without any additional filtering. The application also does not perform any validation on the input, nor constrain it to a pre-computed list of acceptable values. General Recommendations How to avoid it Generic Guidance: · Enforce authorization checks before providing any access to sensitive data, including the specific object reference. · Explicitly block access to any unauthorized data, especially to other users' data. · If possible, avoid allowing the user to request arbitrary data by simply sending a record ID. For example, instead of having the user send an account ID, the application should look up the account ID for the current authenticated user session. Specific Mitigation: · Do not concatenate user input directly into SQL queries. · Include a user-specific identifier as a filter in the WHERE clause of the SQL query. · Map the user input to an indirect reference, e.g. via a prepared list of allowable values.	
Privacy_Violation	Checkmarx is looking for variables with names like 'password', 'credentials', 'Authentication' etc.. and when it sees that you are assigning them a value, it warns you that you might be storing sensitive information in the code (hardcoding it). In the case that you mentioned it looks like a false positive because this is not sensitive information.	https://stackoverflow.com/questions/33671245/privacy-violation-checkmarx#:~:text=Checkmarx%20is%20looking%20for%20variables,the%20code%20(hardcoding%20it).
Use_Of_Hardcoded_Password	Hardcoded Passwords, also often referred to as Embedded Credentials, are plain text passwords or other secrets in source code. Password hardcoding refers to the practice of embedding plain text (non-encrypted) passwords and other secrets (SSH Keys, DevOps secrets, etc.) into the source code. Default, hardcoded passwords may be used across many of the same devices, applications, systems, which helps simplify set up at scale, but at the same time, poses considerable cybersecurity risk.	https://www.beyondtrust.com/resources/glossary/hardcoded-embedded-passwords#:~:text=Hardcoded%20Passwords%2C%20also%20often%20referred,into%20the%20source

		%20code.
Client_Hardcoded_Domain	Fortify's reasoning for this "Hardcoded Domain in HTML" warning is linking to an external domain will compromise the security of your site because the file you are linking can be changed. The following is from "Fortify Taxonomy: Software Security Errors": Abstract Including a script from another domain means that the security of this web page is dependent on the security of the other domain. Explanation Including executable content from another web site is a risky proposition. It ties the security of your site to the security of the other site. Example: Consider the following <script> tag. <script src="http://www.example.com/js/fancyWidget.js"/> If this tag appears on a web site other than www.example.com, then the site is dependent upon www.example.com to serve up correct and non-malicious code. If attackers can compromise www.example.com, then they can alter the contents of fancyWidget.js to subvert the security of the site. They could, for example, add code to fancyWidget.js to steal a user's confidential data.	https://stackoverflow.com/questions/50632726/best-option-to-store-hardcoded-domain-names-in-web-application

Hardcoding_Ids	First, the component you need to reference does not exist in Production — you just created it for the solution you're building. That component will have a new ID that only exists in that sandbox. That same component will need to be created in each sandbox leading up to Production. Each time that component is created, a different Salesforce ID is created and, hence, a different ID will be referenced. As a result, you'll have different versions of your formula, validation rule, process, flow, and so on in your sandboxes and Production. What a deployment and maintenance nightmare! Second, let's say you hard coded messaging that's displayed in a Flow screen and other components shown to your community or internal users. Marketing decides to update the message. Guess what? Since that same message is hard-coded everywhere it's shown, you'll now need to maintain that text in all the places — and that assumes you remember what all those places were. And third, if you need to troubleshoot or enhance an existing solution and it references a hard-coded ID, you can't easily tell what that component is without looking up the ID. The Salesforce ID is not self-describing.	https://admin.salesforce.com/blog/2021/why-you-should-avoid-hard-coding-and-three-alternative-solutions

12 .3 Informational Links

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

http://wiki.developerforce.com/page/Secure_Coding_Guideline
http://wiki.developerforce.com/page/Wiki https://www.owasp.org/index.php/Main_Page

http://www.salesforce.com/docs/en/cce/salesforce_visualforce_best_practices/salesforce

visualforce_best_practices.pdf https://developer.salesforce.com/page/Apex_Code_Best_Practices
https://developers.google.com/speed/pagespeed/insights/
https://developer.salesforce.com/page/Apex_Code_Best_Practices
https://developer.salesforce.com/docs/atlas.enhttps://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_testing_best_practices.htmus.apexcode.meta/apexcode/apex_testing_best_practices.htm https://trailhead.salesforce.com/en/content/learn/modules/apex_testing
https://developer.salesforce.com/docs/atlas.enhttps://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_testing_utility_classes.htmus.apexcode.meta/apexcode/apex_testing_utility_classes.htm
https://developer.salesforce.com/docs/atlas.en-us.secure_coding_guide.meta/secure_coding_guide/secure_coding_sql_injection.htm

https://developer.salesforce.com/docs/atlas.enus.pages.meta/pages/pages_best_practices_performance.htm https://developer.salesforce.com/page/An_Introduction_to_Exception_Handling
https://developer.salesforce.com/page/Apex_Design_Patterns