Coding Guidelines

- Typical JavaScript Developer Mistakes
  - "this" context
  - Arrow vs 'regular' function
  - Asynchronous JavaScript
- Code structure
  - LWC Skeleton
    - Installing and using the snippet in VS-code
    - Installing the snippet in IntelliJ
  - Lightning Web Component Naming conventions
  - Modularity
- Error handling
  - Differentiating error types
    - User-invoked errors
    - Application errors
  - Handling server-side errors
  - Handling and displaying errors in the LWC
    - ReduceErrors
    - ShowToastEvent
- Debugging
  - Debugging approaches
  - Which debug settings to activate
    - Code debugging
    - Responsiveness debugging
    - Performance debugging
- Data CRUD & Server side operations
  - Lightning Data Service
    - Specific Lightning HTML tags
    - Lightning/ui*Api Wire adapters and functions
  - Apex Method Execution
    - Wire an Apex method to a function or to a property
    - Call an Apex method imperatively
  - Important Considerations
    - Use Dynamic parameters for Wiring
    - Wired JavaScript methods are invoked at LWC initialization
    - User Experience
    - Boxcarring
- Communication between components
  - Intercomponent communication
  - Intracomponent communication
    - Communicate from parent to child
    - Getting updates from child components
      - Publishing events from child to parent
      - Leverage an @api annotated getter or method

- Protect your application against vulnerabilities
  - Cross Site Scripting
    - Example
    - Common XSS mitigations
      - Input filtering
      - Output Encoding
    - How to prevent XSS vulnerabilities via Output Encoding?
      - Automatic HTML Encoding (Pre-Built within SF)
      - Explicit HTML encoding
      - Explicit JavaScript escaping
  - SOQL Injection
    - Example
    - Prevent against SOQL injection
      - Static Queries and Bind variables
      - Escape Single Quotes
      - Type Casting
      - Replace Characters
      - Whitelisting
      - Leverage the QueryBuilder library
  - Open Redirect
    - Example
      - An example of attack could be a redirection to an attacker web page that looks like to the SF Login page. End user without noticing the malicious redirection could enter his credentials which will be stolen by the attacker.
    - Standard SF Redirections
      - Standard SF URL redirection are fully protected against malicious redirections on all the standard SF pages and partially when used through VF pages and Apex. Therefore we strongly recommend to not use the standard SF URL re-directions within your custom pages without applying specific measures.
    - Prevent Open Redirect Vulnerabilities
      - Hardcode Redirects
      - Force Local Redirects Only
      - Whitelist URL Domains
      - In case you need to support redirection to a non-local URL coming from a input parameter, be sure to whitelist the allowed domains and reject any URLs not matching those domains.
  - Cross Site Request Forgery (CSRF)
      - CSRF is a vulnerability where a malicious application causes a user's client to perform an unwanted action on a trusted site for which the user is currently authenticated.

Introduction and Scope

This document entails general development guidelines and aims at serving as a reference guide for future development. The standards in this guide should be applied to all new development work and should be introduced to existing applications as part of the ongoing continuous improvement process. More specifically, the following subjects are covered in this document:

- A standard guideline for Apex, Visualforce and Lightning Component development
- Tips and best practices for development
- Easier maintenance/enhancement through consistent standards
- High level guideline regarding integration topics
- A reference to tool and accelerator frameworks we recommend to use

Document Conventions

The different elements and rules presented in this document will be categorized according to their level of importance. See Table 3.1 for an overview of the different levels and their description.

Must-Have	Emphasizes that this rule should be enforced at all times.
Recommended	Emphasizes that this rule should be applied whenever possible and appropriate.
Nice-to-have	Emphasizes that this rule is not strictly necessary, but can enhance your development work.

Table 3.1: Overview of the different levels of importance.

General Best Practices

Coding Pitfalls

Apex code is the Force.com programming language and allows to create custom and robust business logic. As with any programming language, key coding principles and best practices will help you to write efficient and scalable code. Here, we will give an overview of some of the most important best practices to keep in mind while writing and designing Apex Code solutions on the Force.com platform. For a more detailed explanation and some examples, please refer to the online documentation ({+}https://developer.salesforce.com/page/Apex_Code_Best_Practices+)

Best Practice #1: Bulkify your Code **(Must-Have)**
Bulkifying Apex code refers to the concept of making sure the code properly handles more than one record at a time. When a batch of records initiates Apex, a single instance of that Apex code is executed, but it needs to handle all of the records in that given batch. For example, a trigger could be invoked by a Force.com SOAP API call that inserted a batch of records. So if a batch of records invokes the same Apex code, all of those records need to be processed as a bulk, in order to write scalable code and avoid hitting governor limits.
Here, we see example code of a class invoked by a beforeInsert trigger that has been bulkified, it is capable of handling all incoming records instead of only accounting for one record. A for loop is used iterate over the entire Trigger.new collection. Now, if the trigger is invoked with a single Account or 200 Accounts, all records are properly processed.

```
/**************************************************************************************
* @author           Deloitte
* @description      Class including concatenation utilities
* @date             2015-08-14
* @group            Utilities
**************************************************************************************/
public class Concatenation {
	public static void accountConcatenation(List<Account> trigger_new){
		for(Account a : trigger_new){
			a.Description = a.Name + ' - ' + a.BillingState;
		}
	}
}
```

Best Practice #2: Avoid SOQL Queries or DML statements inside For Loops, Streamline Queries and use Collections **(Must-Have)**
For loops allow you to, for example, iterate over a collection of records or give you the possibility to loop certain code fragments for a specified amount of times. On the Force.com platform, one of the governor limits enforces a maximum number of SOQL queries while another enforces a maximum number of DML statements (insert, update, delete, undelete). When these operations are placed inside a for-loop, database operations are invoked once per iteration of the loop making it very easy to reach these governor limits.
Therefore, you should move any database operations outside of for-loops. If you need to query, query once, retrieve all the necessary data in a single query, then iterate over the results. If you need to modify the data, batch up data into a list and invoke your DML once on that list of data.
Furthermore, it is important to use Apex Collections to efficiently query data and store the data in memory. A combination of using collections together with streamlining SOQL queries can substantially help writing more efficient Apex code and avoid hitting governor limits.
Here, we see example code that illustrates these concepts.

```
public class CodingExcellence {
	public static void efficientMethod(Set accountIds) {
		// Query all related Closed Lost and Closed Won opptys in a single query
		// Method is being invoked by a trigger on Account which passes
		// the set of account Ids
		List<Account> accountWithOpptys
			= [SELECT id, Name,
				(SELECT id, Name, CloseDate, Stagename
				FROM Opportunities
				WHERE accountId IN :accountIds
				AND (StageName = 'Closed - Lost'
				OR StageName = 'Closed - Won'))
			   FROM Account WHERE Id IN :accountIds];

		List<Opportunity> opptysToUpdate = new List<Opportunity>();
		for(Account a : accountWithOpptys) {
			for(Opportunity o : a.opportunities) {
				// Do some magic here..
				opptysToUpdate.add(o);
			}
		}

		// Outside the for loop, perform a single update DML
		// statement for all records
		update opptysToUpdate;
	}
}
```

Best Practice #3: Streamlining Multiple Triggers on the Same Object **(Must-Have)**
It is important to 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 possibly have redundant for statements. First of all, you do not have any explicit control over which trigger gets initiated first. Secondly, each trigger that is invoked does not get its own governor limits. Instead, all code that is processed, *including the additional triggers,* share those available resources.
So instead of only the one trigger getting a maximum of 100 queries, all triggers on that same object will share those 100 queries. That is why it is critical to ensure that the multiple triggers are efficient and no redundancies exist.
Deloitte has implemented a ***Trigger Handler Framework*** providing a ***Selector*** concept which allows to avoid query deduplication. (See section 13.2.1)

Best Practice #4: Use of the Limits Apex Methods to Avoid Hitting Governor Limits **(Nice-to-have)**
Apex has a System class called *Limits* that lets you output debug messages for each governor limit. There are two versions of every method: the first returns the amount of the resource that has been used in the current context, while the second version contains the word limit and returns the total amount of the resource that is available for that context.

Best Practice #5: Writing Test Methods to Verify Large Datasets **(Nice-to-have)**
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. While this approach would make sense theoretically, the drawback is the increased time of execution and more complex test methods. For those reasons test classes should not necessarily test bulkification as such. However the code must be obviously robust against bulkification, especially in triggers.

Best Practice #6: Avoid Hardcoding IDs **(Must-Have)**
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.

Best Practice #7: Use Custom Labels **(Must-Have)**
Custom labels are custom text values that can be accessed from Apex classes or Visualforce pages. The values can be translated into any language Salesforce supports. Custom labels enable developers to create multilingual applications by automatically presenting information (for example, help text or error messages) in a user's native language. You can create up to 5,000 custom labels for your organization, and they can be up to 1,000 characters in length.

Naming Convention

CLASSES (Must-Have)

While leveraging naming convention within your project in a consistent manner is a must have, the below is a proposition of convention but should not be considered as the only option. Each project team can decide on their own conventions considering the client context.
The following conventions must be applied:

- Must be nouns.
- Camel case with the first letter of each internal word capitalized starting with an upper case.
- The Name of the Class should always start with the project prefix e.g. ***[Prefix]_MyClass***
- The following naming conventions are recommended based on the Class purpose:
    - Controller Class: ***[Prefix]*****[VF Page/LC Name]*****Ctrl***
    - Batch Class: ***[Prefix]*****[Functionality]*****Batch***
    - Schedulable Class: ***[Prefix]*****[Functionality]*****Schedulable***
    - Queueable Class: ***[Prefix]*****[Functionality]Queueable**_*
    - Utility Class: ***[Pefix]*****[Functionality]Utility**_*
    - Trigger: ***[Prefix]*****[sObject]Trigger**_*
    - Interface Class : ***[Prefix]_I[Functionality]***
    - Test Class: ***[Prefix]*****[Class Name]*****Test***
    - Exception Class: ***[Prefix]*****[Functionality]Exception**_*
    - Command Class: ***[Prefix]*****[Functionality]Cmd**_*
    - Factory Class: ***[Prefix]*****[Functionality]Factory**_*
    - Other: ***[Prefix]*****[Functionality]**_*
- In case of Inner classes, the [Prefix] is optional and should not be added
- Should you implement a Domain Driven Design Approach leveraging on the TriggerHandler Framework (See section 13.2.1) the following convention should be used:
    - Domain class: ***[Prefix]*****[sObjecName]*****Dom***
    - Service class: ***[Prefix]*****[Service Name]*****Srv***
    - Cross Domain Selector class: ***[Prefix]*****[sObjectName or Service Name]*****XDomSel***
    - Selector class: ***[Prefix]*****[sObjecName]*****Sel***

Please refer to the agreed prefix list in our Salesforce Development Standards

METHODS

The following conventions must be applied:

- Methods should use camel-case staring with lower case.
- They should express the use case the method is trying to solve ***starting with a verb*** describing the action of the method.
- For example - A method to get all Accounts based on a String passed as a parameter can be named as: *getAccountsByName*

VARIABLES/PROPERTIES

The following conventions must be applied:

- Variables and properties should use camel-case staring with lower case
- Variable/Property names should not start with underscore _ or dollar sign $ characters, even though both are allowed.
- Variable names should be meaningful. Better to have longer variable meaningful name instead of shorter name difficult to understand.

CONSTANTS

The following conventions must be applied:

- Constants should always be capitalized.
- Use '_' to separate two words while declaring a constant.
- It should indicate nature of the value the constant holds.
- Constants should be static and final, so that they can be used in more than one place and cannot be changed.
- For example - A string holding the name of Admin profile can be declared and instantiated as:

Public static final string SYSTEM_ADMIN = 'System Administrator'

VF PAGES & LIGHTNING COMPONENTS (AURA)

Any VF page or Lighting Component should follow the following naming convention:

- Must be nouns.
- Camel case with the first letter of each internal word capitalized starting with an upper case.

- The Name of the VF Page/Lightning Component should always start with the project prefix e.g. ***[Prefix]*****[VF/LC Name]**_

LIGHTNING WEB COMPONENTS

Any Lightning Web Component should follow the following naming conventions:

- **LWC Name:** For Lightning Web Component naming convention, the prefix should be lower case and the underscore should be removed, otherwise this will cause issues when referring the LWC. e.g. *PRJ{}*MyLightningWebComponent_ should be named ***prjMyLightningWebComponent*** instead. However the class name inside the LWC JavaScript can use the full Camel Case naming convention.
- **Wired Property**: The naming convention for wired variables states that the variable should be prefixed with '***wired'***. E.g. when information about a Contact is being wired, the variable should be named ***'wiredContact'***.
- **Wired Method:** The aforementioned naming convention applies for wired method. e.g. ***'wiredGetContact'***.
- **Event Name:** Event name must be only lowercase and, as a convention, should not start with ***'on'*** and should not end with ***'event'***. A general best practice is to name an event as descriptive as possible. E.g. when a contact detail is changed, the event should be called '***contactdetailchange***'. The handler for this event should also be named accordingly. Typically a handler for the aforementioned event should be a function called '***handleContactDetailChange'***.

Constants and Literals (Recommended)

Using *hard coded* constants should be avoided whenever possible. Application (public static final) constants should be used instead. This will have the following benefits:

- In case you need to change the constant value at a later stage, you will have to apply it only once
- This will also reduce the risk of typos in case the constant value is often used in different places.

Pay attention to not store all the constants in one big class. Keep in mind the segregation of duties and apply the following principles:

- Constants related to a given object (e.g. Record Type Name, Picklist values, …) belongs to the Domain Class
- Constants relevant to only a specific feature/business process should be stored only in the Service they belongs to.
- Other global constants can be stored in one or several constant classes grouping them logically.

Do not use:
Constants should not be used to store any Ids or Keys (SF or external) that could change or that are sensitive and subject to security breach if exposed.
Constants VS Custom Labels
In case the constant should not be translated because it is containing a picklist value, a record type name, … be sure to use constants and avoid custom labels which will lead to breaking the code if the custom labels are translated by mistaken.
In case, you need to display a text in the User Interface, always use custom label which will support translation.
An example of working with constants within a single class:

```
public static final START_DAY = 2; // Weekdays begin on Monday

if (weekDay(currentDate) >= START_DAY) {...}
```

is much easier to understand and more easy to maintain than

```
if (weekDay(currentDate) >= 2) {...}
```

You can find an example of a constant class allowing maintainance global of constants more easy and centralized:

```
/*************************************************************************************
* @author               Deloitte
* @description          Global Constants class
* @date                 2015-08-14
* @group                Common Libraries
*************************************************************************************/
public with sharing class COM_CST {

    public enum triggerAction {beforeInsert, beforeUpdate, beforeDelete, afterInsert,
afterUpdate, afterDelete, afterUndelete}

    /*******GENERAL CONSTANTS*******/
    public static final int MAX_NUMBER_OF_RETRIEVED_RECORDS = 10; // Limits
        the number of records presented to the user

    /*******ERROR MESSAGES*******/
    public static final String GENERAL_ERROR_MESSAGE
      = 'An error has occured, please try again'; // General Error Message
     public static final String ERROR_NO_RECORDS_FOUND
      = 'No records could be retrieved.' // No records could be found

}
```

Code Indentation (Must-Have)

Indentation

The code should be correctly aligned throughout for improved readability. Wherever required, please use tabs to provide the correct amount of gaps as indentation. Tabs should be used as the unit of indentation and should be set to ***4 spaces***. Tabs should be consistently used in all source files in a project. If your IDE/editor allows (see Development Tools below), you should set Tab to automatically expand to 4 spaces. Do not commit code that contains Tab characters.
Shortcuts for Indentation:

- ***Developer Console***: Use Ctrl+a then shift+tab to indent code.
- ***InteliJ*** Ctrl + Alt + L
- ***Eclipse*** Ctrl + I
- ***VSCode*** Shift + Alt + F

Wrapping Lines

When a statement will not t on a single line, break it according to these general principles:

- Break after a comma.
- Break before an operator.

- Prefer higher-level breaks to lower-level breaks.
- Align the new line with the beginning of the expression at the same level on the previous line
- If the above rules lead to confusing code or to code that's squished up against the right margin, just indent 8 spaces instead.

SOQL Formatting

Follow the below best practice:

- SOQL keywords (For e.g. SELECT, FROM, WHERE, TODAY) should always be written in ALL CAPS

inside Apex classes.

- Break SOQL Queries across multiple lines in case too long.
- Put each clause on its own line, including SELECT, WHERE and all its sub-clauses, FROM , GROUP
- BY, ORDER BY, LIMIT, etc.
- Generally put as many fields as possible on one line without overrunning 80 line length
- List the fields in the ascending alphabetical order
- Put each sub-query on its own line

In order to be consistent and always format SOQL correctly, feel free to use formatting tools such as:
{+}http://codegen.keweme.com/SOQL+

Allow Triggers to be muted by design

(Must-Have)
One common request on implementations is to turn off triggers for certain users usually during data loads to help improve performance or to disable functionality that is not needed. Usually this would be done by inactivating the trigger itself before running the load. However, for large customers who are already live running 24x7, inactivating a trigger for all users may be undesirable or unrealistic. Also inactivating a trigger in a PROD environment will requires a deployment, which is not best practice.
Deloitte offers a ***Trigger Handler framework*** offering this feature out of the box. (See section 13.2.1)

Apply Domain Driven Design to your implementation

(Recommended)
Applying the ***Domain Driven Design*** concepts will help you to better apply object oriented programming concepts and will help you to:

- ***Avoid building Blob Anti Pattern*** by defining a methodology where to build which logic.

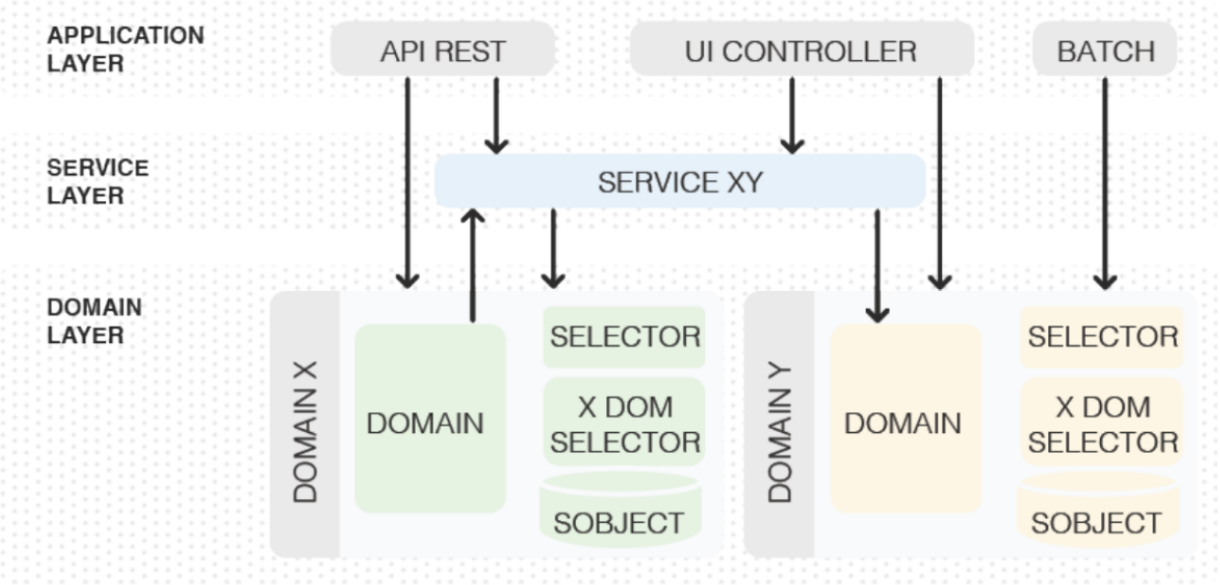
The Blob Anti Pattern is very common in Apex Trigger since it is very easy to build per Trigger one big helper class containing the whole logic of the Trigger.

- ***Avoid duplicate queries*** which will lead to reach governance limits and to performance issues by leveraging the *Selector (Repository)* concept.

Domain Driven Design recommend to build your code according **3 layers**:

- ***Application Layer:*** This layer is the entry point of any apex logic: from the UI (Controller), from Asynchronous processes (Batch), from the API (exposed Apex APIs) or from the Trigger. This layer should be minimal (no business logic) and should delegate work to collaboration of services and domains.
- ***Domain Layer:*** Each entity (typically sObject) should have its own domain in which business logic specific to this entity should be defined. Only core logic of the entity should be produced directly in the Domain Layer. If logic is complex, integration related or cross entities related, the domain layer should use a Service in order to perform the business logic. Each domain layer is using a ***Cross Domain Selector*** used to perform the needed queries on related entities and cache it in order to avoid queries deduplication. Each domain also exposes a ***Selector*** which is the class containing all the different queries performed for this Domain. This selector class is exposing ***Methods*** performing the SOQL queries and are invoked from the different ***Cross Domain Selector*** classes, which need to retrieve and cache the related entities of a given Domain (Trigger Context).
- ***Service Layer:*** Any operation that does not belong to a specific domain or that involves multiple domains should be encapsulated in a Service.

Important Remark
The ***Cross Domain Selector*** is not a concept existing in the Domain Driven Design but has been introduced for the specific SF needs in order to allow caching and ensure performances.



How can I apply those concepts on my project?
In order to help you to apply those principles, Deloitte offers a ***Trigger Handler framework*** enforcing every trigger to apply those concepts by providing the following code:

- A base abstract class for the Domain
- A base abstract class for the Selector
- A base abstract class for the XDomSelector
- A set of examples on how to use this framework.

Please refer to the framework detailed documentation (See section 13.2.1) for more information.

Apply the unit of work concept

(Recommended)
When building complex and extended business logic in a Trigger, you might quickly face the below issues:

- How to avoid duplicate DML transactions performed on the same object by different features implementation?
- How to keep your code clear and clean when you need to bulkify all your DML transactions? You will end up using Set, List and Map and make your code less readable.

The solution is the use of the ***Unit Of Work*** library. This library is collecting all the DML transactions to be performed (*Insert, Update, Delete and Upsert*). Only at the end of the logic processing, the actual DML transactions are executed.
Read more about Unit Of Work Concept here
This concept is implemented in the well-known framework fflib
However, this whole fflib framework is very extensive and not always easy to use. It also comes with some limitations. Deloitte has created its **own version of the Unit Of Work** as part of the ***Common Library*** framework (See section 13.2.2) which is an improved version of the fflib one:

- Support of Parent relationships
- Support of Circular relationships
- Full support of Upsert functionality using external ids.

Please refer to the **Common Library** documentation for full details on **when** and **how** to use the **Unit Of Work.**

Caching attributes to avoid hitting governor limits**(Must-Have)**

Standard metada information

When accessing data which are mostly static (e.g. RecodTypes, Active Territory, Profiles, …) or accessing dynamic apex data through ***globalDescribe*** method, it is key to ensure to **cache** the information retrieved in order to avoid performance issues and risk of reaching governance limits if several implemented features need an access to the same set of data.
We recommend the use of the ***Common Library*** framework which provides already a full set of methods allowing access to RecordTypes, Profiles, … information and performing Dynamic apex operation in an optimal way. The framework is already managing this caching for you out of the box.
Please refer to the Common Library framework section (See section 13.2.2) for detailed list of available methods.

Custom static information

In case of static information specific to your implementation, you should apply also this caching mechanism in order to avoid any risk. In order to do so, be sure to encapsulate any generic query done in **properties** defined in a **custom library method**.

Example of caching Implementation:

```
public static List<MyRule__c> assignmentRules
{
   get
   {
      if (assignmentRules == null)
      {
              assignmentRules = [SELECT field1, field2 FROM MyRule__c];
      }
      return assignmentRules;
   }
   private set;
}
```

Entity Specific related information

In case you need to perform a SOQL query which is related to a given trigger context, you should ensure the caching of those SOQL query by leveraging the ***Selector*** and ***Cross Domain Selector*** pattern from the ***Domain Driven Design***. (See section 3.6)

Access Modifier and sharing**(Must-Have)**

Access Modifier:

The following access modifiers are available for ***Classes***, ***Methods*** and ***Properties***:

- private
- protected
- public
- global

As a rule, you should be always as restrictive as possible and open the model only when needed:
VARIABLES/PROPERTIES/METHODS

- Class variables and properties that defines the class internal state model are usually **private**
- **Do not define** class variables used in only one method. In this case, it define it as a method variable.
- Class variable and properties that must be accessible outside of the class should be defined as **public**
- Properties can have a different access modifier for the getter and for the setter. If the property is ready only, be sure to define a **public** getter and a **private** setter.
- ***Test methods*** should always be **private**.
- **global** access modifier should be used only when the class should be accessible outside of the SF org where it is defined (variables defined as part of a message returned by a custom Apex REST/SOAP API or variables part of class developed in a manage package which needs to be accessible from the org where the package is installed.)
- Methods used only within the class should be set as **private**
- In case of inheritance, define as **protected** any methods/properties/variable that must be accessible to the class extending the base class.

CLASSES

- Class are usually defined as **public**
- Inner Classes only used within the main class are flagged as **private**
- ***Test classes*** should always be **private**
- Batch apex, Schedulable Apex, Queueable Apex are flagged as **public**
- **global** access modifier should only be used when the class exposes a custom REST/SOAP service or when the class is part of a manage package and must be accessible from the org where the package is installed.

Sharing Model

A class can have one of the 3 following sharing model defined:

- with sharing
- without sharing
- inherited sharing

Apply the following rules to decide which sharing model must be applied:

- A VF or Lightning Component controller must always be flagged as ***with sharing***
- Any other class, not being a controller should be flagged as ***inherited sharing***
- You should try to avoid using the ***without sharing*** keyword. In case you need some logic to always run in a ***without sharing*** context, encapsulate that logic in a ***without sharing*** class and limit the logic of this class to only the feature that needs to run in the without sharing context. Provide the appropriate commenting in order to explain why the without sharing is needed.

- Batch, Schedulable, Queueable Apex class should be declared with the ***inherited sharing*** keyword.

Enforcing FLS & Object Level Access

SF mechanism to enforce CRUD and FLS is not always well understood by the developers. The sections below describe the standard SF behavior and provide recommendations and best practices in order to keep the security enforced.

What are CRUD, FLS and Sharing

CRUD
CRUD is the abbreviation for Create, Read, Update, Delete and drives the fact a given user can perform those operations on a given record or not. An important attention point is that when SF refers to CRUD in the literature, they mean the **CRUD access defined in the Objects Permissions in the Profile** which sometime brings confusion since this is not because you have the CRUD access that you really have access to a given record since access is controlled by the Profile CRUD and Sharing Accesses.
FLS
FLS is the abbreviation of Field Level Security and drive the access to a given field (read or read/write access) from a record.
Sharing
Sharing allow to manage who can do which operation on a given record through standard mechanisms such as:

- OWD
- Sharing Rules
- Role Hierarchy
- Manual Sharing
- Modify All/View All Permission on Profile

With/Without Sharing functionality

With Sharing:

- *Sharing enforced:*
  - Queries will return only record accessible to the users (defined by the SF sharing model)
  - DML operation will fail in case the appropriate right is not granted to the user through the sharing model (OWD, Sharing Rules, Role, …)

- *CRUD not enforced (Profile Object Permissions)*:
  - CRUD defined at profile permission level are not enforced, this means that DML/Query operation on a record accessible through the sharing model but for which the profile does not grant the access will still succeed.

- *FLS not enforced*:

Field Level Security are not enforced and query/dml will not cause any exception even if the field is not accessible or read only.
Without Sharing:

- *Sharing and CRUD not enforced*:

Running under a system admin context which will allow to retrieve any data and perform any DML transaction.
Attention Point: Some limits still exist if the user license does not grant access to some functionalities.

- *FLS not enforced*:

Field Level Security are not enforced and query/dml will not cause any exception even if the field is not accessible or read only.

Trigger Behavior

Trigger are running in a System Admin context meaning that it will have the Without Sharing behavior unless you invoke a class explicitly flagged with the With Sharing keyword.

Enforcing FLS/CRUD directly through the VF page or Lightning Component

While the ***With Sharing*** does not enforce ***FLS and CRUD (Profile Object Permissions)***, Visualforce page and Lightning Components offers capabilities to **enforce it**. So using this functionality in combination of the With Sharing keyword is a good way to enforce the security.
VF pages tags enforcing FLS:

- *apex:inputField:* field value will not be displayed if user has no read access to the field
- *apex:outputField:* field value will not be displayed or will be read only if user has no access or read only access only.

Lightning Components tags enforcing FLS:

- *lightning:inputField:* field value will not be displayed if user has no read access to the field
- *lightning:outputField:* field value will not be displayed or will be read only if user has no access or read only access only.

CRUD Enforcement:

- When using standard SF methods to retrieve and save data within a VF or Lightning Component (e.g. leveraging on standard controller methods for VF or retrieving and saving data via the Lightning Data Service for the Lightning Components), the CRUD and the Sharing will be fully enforced. However as soon as you perform explicitly a query or a DML in Apex, CRUD (Profile Object Permission) will not be enforced and ***Sharing enforcement*** will depend on the ***With or Without sharing*** defined at the controller level.

Guidelines - Performing an Explicit CRUD/FLS check

Taking into account the above explanations, we can draw the following recommendations:
Logic within Trigger:
Usually, when executing logic within a trigger, you usually ***do not want to enforce FLS, CRUD and Sharing*** since triggers are executing business logic that must be enforced for any user role. Besides those triggers usually manage technical fields for process logic which are usually not accessible to normal users.
Within Triggers:

- Do not perform CRRUD and FLS checks
- Exception can be done for specific use cases

Logic within VF page and Lightning Component Apex Controller:
In a controller, the developer is responsible to ensure that information displayed in the User Interface or that DML operations are not allowed in case the user does not have the right access rights.
Within Apex Controller:

- Always use the With Sharing keyword in order to enforce the Sharing
- Perform FLS explicit check if you **query** fields which are **displayed in the UI** not using VF/Lightning Component tags enforcing FLS
- Perform FLS explicit check if you perform a **DML** (insert/update/upsert) providing field value **encoded in the UI** through VF/Lightning Component tags not enforcing FLS.
- In case, you perform a DML on fields not provided as input from the UI but part of your logic, this is up to the developer to decide if FLS should be enforced or not depending on the use case.
- Query and DML operations **will not enforce CRUD** (Profile Object Permissions). Therefore, unless you are using the Lightning Data Service or the Standard VF controller to perform the query or the DML, you must explicitly check the CRUD access before performing the operation and throw an exception in case it is not respected.

How to explicitly enforce CRUD and FLS?

Via Dynamic Apex
FLS:
Perform a getDescribe() for the field you would like to check the FLS:
e.g. *Schema.DescribeFieldResult dfr = Account.Description.getDescribe();*
Use the following methods

- *isAccessible()*
- *isCreateable() & isUpdateable()*

CRUD:
Perform a getDescribe() on the object you would like to check the CRUD:
e.g. *Schema.DescribeSObjectResult dsr = Account.sObjectType.getDescribe();*
Use the following methods

- *isAccessible()*

- *isCreateable()*
- *isDeletable()*
- *isUpdateable()*

Important Remark:
The CRUD check is checking the Profile Object Permission access but not the Sharing access for a given record. Therefore, this is not because a user has the right CRUD access that he has access to a given specific record.
Via SOQL
Using the ***WITH SECURITY_ENFORCED*** keyword in a SOQL query will enforce FLS and CRUD.
e.g. *SELECT Id, Name, Description FROM Account WITH SECURITY_ENFORCED*
In case, a field or record is not accessible, an exception will be thrown.
This feature has been made generally available as of Spring 20 (48.0). However in order to use it safely be sure the Apex Class using it is running at least under the api version 48.0.
Leveraging the Deloitte Common Utility Framework
The Common Utility framework offers out of the box capabilities to easily enforce CRUD and FLS (See section 13.2.2 for detailed information):

1. Using the ***UTL_DynamicApex*** class

This library offers a set of methods ***validateCRUD*** and ***validateFLS*** that can be used to easily enforce the security. An exception is thrown in case security is not respected.

1. Using the ***UTL_QueryBuilder*** class (for SOQL Queries)

When building a query dynamically with this class, you can easily enforce the CRUD and FLS. Building queries in such a way provides an easy way to enforce CRUD and FLS in a consistent way (when needed) while performing SOQL queries.

1. Using the ***UTL_SObjectUnitOfWork*** class (for DML operations)

The Deloitte implementation of the Unit Of Work offers out of the box capabilities in order to enforce CRUD and FLS in a consistent way (when needed) while performing DML operations.

Commenting

All procedures should begin with a brief comment describing the functional characteristics of the method (what it does). This description should not describe the implementation details (how it does it) because these often change over time, resulting in unnecessary comment maintenance work, or worse yet erroneous comments. The code itself and any necessary inline or local comments will describe the implementation.
Parameters passed to a method should be described when their functions are not obvious and when the method expects the parameters to be in a specific range. Function return values and public variables that are changed by the routine must also be described at the beginning of each appropriate routine.

Class and Trigger Header

Author	Name of author of original code
Description	What the class does (not how).
Date	Date of creation of the class
Group	A logical grouping of the class

```
/*************************************************************************
* @author Deloitte
* @description Utility class for common functions
* @date 2015-04-27
* @group Common Libraries
*************************************************************************/
```

Table 5.1: Class Header

Method Header

Author	Name of author of original code
Date	The date when the method/function was created

Description	What the function does (not how). Including an outline of all methods executed
Params	A description of the attributes passed to the method
Return	A description of the returned value

```
/*************************************************************************************
* @authorKarolinski Stephane
* @date2015-04-27
* @description The method format a structured address in a string
* @paramstreet (String): the address's street
* @paramcity (String): the address's city
* @parampostalCode (String): the address's postal code
* @paramcountry (String): the address's country
* @returnString: the formatted address to be displayed

*************************************************************************************/
```

Table 5.3: Function Header

Property Header

Author	Name of author of original code
Description	What the class does (not how).
Date	Date of creation of the class

```
/*************************************************************************************
* @authorKarolinski Stephane
* @date2015-07-28
* @descriptionProperty returning a boolean indicating if the SFDC is a
* sandbox or prod environment
*************************************************************************************/
```

Table 5.1: Property Header

Automatic Documentation Generation (ApexDoc)

Following the above commenting style allows to easily generate code documentation using the ***ApexDoc*** open source Java program.
ApexDoc is a java app that you can use to document your Salesforce Apex classes. You tell ApexDoc where your class files are, and it will generate a set of static HTML pages that fully document each class, including its properties and methods. Each static HTML page will include an expandable menu on its left hand side, that shows a 2-level tree structure of all of your classes. Command line parameters allow you to control many aspects of ApexDoc, such as providing your own banner HTML for the pages to use.
Download the program here:
{+}https://github.com/SalesforceFoundation/ApexDoc+

Asynchronous Jobs guidelines

Overview

SF offers capabilities to process some apex in an asynchronous transaction running outside of the Trigger or VF/LC Controller transaction. There are several use case where this is would be required:

- Jobs that needs to be schedule are at given frequency and not initiated manually
- Processes that are resources intensive which cannot afford to block the UI
- Processes that cannot be executed within a Trigger (e.g. apex callouts)

4 types of asynchronous Jobs are available within SF:

Schedulable Apex	Allows to schedule an Apex class to be executed at a given frequency (**e.g.** once a day). Frequency is defined by a CRON syntax.	**No**
Queueable Apex	Allows to initiate an asynchronous job providing any type of parameters as input	**No**
Batch Apex	Allows initializing an asynchronous jobs processing the outcome of a SOQL query or processing a List<sObject>. A batch size is defined and SF splits the list to be processed in smaller lists of the size of the batch. Each sub list is processed in a dedicated transaction which allows to better control the governance limit than a Queueable Apex or Schedulable Apex.	**Yes**
@future	Allows to initiate an asynchronous job providing a limited set of input parameters (only primitive data type and array or collection of primitive data types are supported)	**No**

Important Governance Limits

There are a few important governance limits that any developer should be aware of in order to design correctly their asynchronous processes.

Flex Queues Limit *Parallel Jobs Processing*	There is a limit of **max 5 Flex Queues executed at the same** time within a SF Org	Using Flex Queues does not allow to build a fully scalable application and even worst if the jobs are being cumulated it will lead to a complete failure of the system.
Flex Queues Limit *Max Jobs Queued*	A maximum of **100 jobs** using Flex Queues can be queued. In case this limit is reached, an exception will be thrown when adding an additional job.	
Long Running Synchronous Transactions	Maximum **10** synchronous concurrent long-running transactions that last **longer than 5 seconds** for each org. Any transaction, involving directly or indirectly apex are subject to this limit. In case you reach this limit an exception will be thrown.	While this limit does not include callout duration since the Winter 20 release, it still means that if you are trying to process synchronously some logic, which takes time to execute, it will lead to scalability issues. Depending on how many agents are performing this action at the same time the limit can be quickly reached without any way to scale this up.
@Future *Parallel Jobs Processing*	**Up to 2000 jobs in parallel**. If the limit is reached, jobs are simply queued and no exception is thrown.	Using @Future is a good way to allow to process asynchronous jobs in a scalable way if you can live with the @Future limitations.
@Future *Invocation Context*	A @Future job cannot be invoked from another @Future or Batch Apex job.	This limit makes difficult to invoke a @Future from a Trigger in case this one can be initiated from a synchronous or asynchronous context. In this case a check should be done to avoid calling the @Future method if already in an asynchronous jobs context. However this would mean that you will execute your logic within the same transaction without resetting the governance limits.
@Future	**Maximum 50 @Future** jobs can be invoked within a given transaction.	This means that if you plan to invoke an @Future job from a Trigger, you will not be able to bulkify it. It means that the @Future job should have the capacity to process the bulk in one go.

Invocation #		
Batch Apex *Invocation Context*	A batch Apex **cannot be invoked** from the ***execute*** method of another Batch Apex	This means that if you plan to design complex jobs that needs to invoke another batch apex (e.g. in case you get closed from the governance limits), you will have to design as following: Batch Apex Invoking Queueable Apex Invoking Batch Apex
Queueable Apex *Invocation #*	**Maximum 50 Queueable Apex** jobs can be invoked within a given **synchronous** transaction. **Only 1 Queueable Apex** job can be invoked within a given **asynchronous** transaction.	Limit are different depending on the context (synchronous or asynchronous) which makes designs difficult given that jobs designed to be invoked in Triggers can be invoke in cascade by other Triggers and can run in some case synchronously and in some other case asynchronously.
Increased Limits in Asynchronous Jobs	Most of the key governance limits are doubled for asynchronous transactions (e.g. DML, Queries, …)	This highlight the fact that performing heavy processing is safer to be executed in an asynchronous transaction.

Guidelines and recommendations

Taking into account the above limits and in the mindset to design an application, which is fully scalable, the following recommendations can be made when invoking asynchronous processes from a Trigger:

- Never design synchronous processing which can takes more than 5 seconds to execute, in this case always consider asynchronous options.
- Always prefer the use of @Future or Queueable Apex within a Trigger at the condition you can live with the limitations.
- In case you need to invoke a Batch Apex within a Trigger due to large volume of data to process, it will be safer to invoke a Queueable Apex invoking the Batch instead of involving directly the Batch to avoid the risk of reaching some of the limits.
- Batch Apex can be invoked from a Trigger at the only condition frequency is not too high. Keep in mind the Batch Apex are consuming Flex Queues and are therefore subject to the Flex Queues limits.
- Never call @Future in a loop (e.g. within a Trigger which can lead to reaching the governance limits)
- Always keep in mind that there is no SLA on the time SF will take to pick a job from the async stack whatever async capability you are using. While in most of the case, it is taking 2 to 5 seconds, if the tenant is heavily being used by other companies, degradation can be observed. In this case contact SF to ask them to perform a server re-balancing to solve this issue.

Error Handling

Error handling on the Force.com platform exists in two forms:

- **Do nothing at all:** Exception that will occur will be raised and the technical error will displayed in the UI. In case of trigger, the full transaction of the batch will be rolled-back. This is, however, not recommended. In any realistic scenario, one should always incorporate some error handling logic into their code in order to design robust applications.
- **Use Try-Catch statements and special processing for unexpected errors:** Try-catch statements should be used as frequent as possible, because this allows for full control on how errors are handled in the application. It allows you to process errors in a clean way, instead of them being showed to the user through the UI. It is recommended to incorporate a global error framework method to consistently log errors, debug, info and warning messages. This is a must for asynchronous code execution where errors will not be displayed to any UI (@future, batch, email servicing,..). The LOG Message framework offers that capability (See section 13.2.3)

The recommended error handling is different depending on the context:

- Apex Triggers
- Apex Controllers
- Asynchronous processes

For further details on Error Handling in Apex, please refer to: {+}https://developer.salesforce.com/page/An_Introduction_to_Exception_Handling+

General exception handling considerations

(Must Have)
In complex projects, the code is usually made of hundreds of classes with many dependencies. A controller could potentially make the use of a service class, which is also used in a context of a trigger. In order to keep your application manageable a strict segregation of duty is needed:

- An unexpected exception occurring in a given context should not be processed in the class where it occurred but should **bubble up until the top class** (Trigger, Controller, Main Asynchronous class). In that way depending on the context the error occurred, a different error management can be applied.

- This means that all the classes **should not have try catch** systematically. Only the top level class (Trigger Handler class, VF/LC Controller class and Main Asynchronous class) **must have it**.

- In case a service class expects that some functional exceptions could occur as part of the normal application behavior, it is recommended to also make use of try catch in this case in order to manage correctly the expected behavior. In this case this is up to the developer to decide, depending on the business requirement, if an exception should be thrown or if a given business logic should be applied.

Apex Trigger Exceptions

(Must Have)
Custom Apex Trigger execute on both standard and custom objects in response to all data modifications performed, either through the application or the API. As such, it is imperative for triggers to adhere to optimal performance and high maintainability, to guard against recursive behavior and most critically to protect data integrity. Triggers should be used judiciously and an assessment should be done to verify whether the logic is indeed time critical or could also be executed using e.g. Batch Apex.
Error handling in Apex Triggers can be done by using a combination of the ***Try-Catch*** mechanism and the ***addError*** method. The ***addError*** message can be used on the object or on the field level and allows to only rollback the affected records, instead of the full batch.
How these techniques should be used depends on the specific scenario, there are 3 use cases:

1. **We do not expect any custom Apex validation or errors**: No Try-Catch, No addError.
2. **If a custom validation is needed:** No Try-Catch, use addError.
3. **If expected errors might happen for which a specific handling or error message is needed and you want to avoid the whole batch being in error**: Use Try-Catch, optionally use addError on the record for which the transaction must be rolled back.

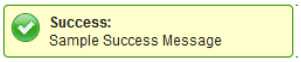
Apex VF Controllers

(Must Have)
In case of VF page controllers, you should always catch any unexpected error through the use try-catch statement.
Errors can be displayed nicely on VisualForce pages. To do so, a specific VisualForce tag must be used: ***<apex:pageMessages>.*** This component is used to display all the messages generated from the components on the current page. If this component is not included in the page then most of the warnings and error messages are only shown in debug log.
If you use a custom controller or extension, you must use the ***ApexPages*** class for collecting errors. An example of it being used, see Figure 2. We recommend to leverage on the ***hasMessage*** method before adding systematically add the Exception. For DML exception SF is adding automatically the Exception into the messages and adding it again in the try catch might duplicate the messages displayed.
There are two methods that can be used to display error messages:

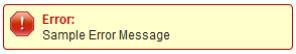
- **addMessage**: display a custom message with a custom severity.
- **addMessages:** is used to display an Exception message that would have occurred. It can contain several message in case of DML Exceptions.

When using the ***addMessage*** method, there are four different severity levels which all have a different look and feel:

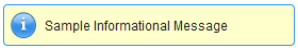
- **Confirm/Success** *ApexPages.addMessage(new ApexPages.Message(ApexPages.Severity**.Confirm** , '**Sample Success Message**'));*



- **Error** *ApexPages.addMessage(new ApexPages.Message (ApexPages.Severity.Error , 'Sample Error Message'));*



- **Info** *ApexPages.addMessage(new ApexPages.Message(ApexPages.Severity.**Info** , '**Sample Informational Message**'));*



- **Warning** *ApexPages.addMessage(new ApexPages.Message(ApexPages.Severity.**Warning** , **'Sample Warning Message'**));*

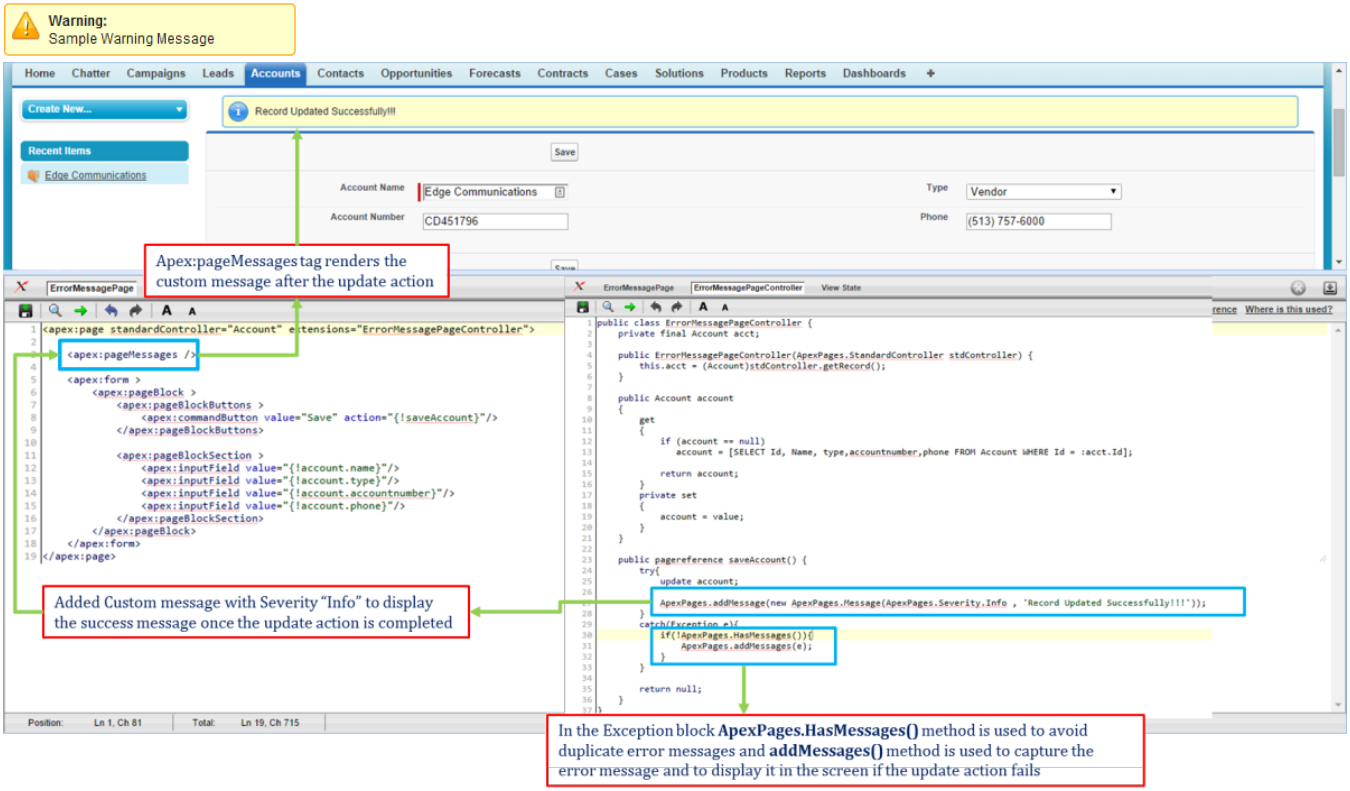


Figure 2 Handling Exception in an VF Context

Lightning Component (aura) server side controller

While processing server side a method invoked from a Lightning Component, a raw exception thrown server side will not be catchable client side and the Lightning Component will crash. In order to manage this correctly there are two possible approaches:

- Throw the exception as an ***AuraEnabledException***

This should be the default approach and will allow the Client Side Controller to catch the exception and process it accordingly (e.g. Displaying a Toast)

- Do not throw any exception and build a wrapper class including the response message expected by the Lightning Component in case no exception is thrown and some additional info providing a full context if an error occurred:

-
  - ***isSuccess***: to indicate if an error occurred or not
  - ***errorMessage***: the error message in case an error occurred
  - ***errorType***: the type of the error in case an error occurred
  - ***stackTrace***: the stackTrace of the error in case an error occurred

The Common Library Framework offers an abstract class that can be extended which provides the above variables out of the box and an easy method to set those in case of error. (***UTL_ExceptionMgt.MessageResponseBaseForLC***). See section 13.2.2
This approach has the advantage to not have the transaction rolled backed and is typically used in the context of integration where, even if an exception occurred, you would like to be able to still log the integration payload for debugging.
In this case, this is the Client Side Controller parsing the wrapper, which deduce if an error occurred or not.
This approach is essentially used in the context of Integration callout where you want to log the payload for debugging purpose, even in case of exception. However if you leverage the **latest Log Message framework** (see 13.2.3) which is using Platform Events, you could in this case throw directly the exception since message logged will not be rolled back.

Lightning Component (aura) client side controller

When the apex server side controller throws an ***AuraEnabledException***, it can be caught and displayed via a toast as described below. We recommend encapsulating this logic in a helper method, which can be reused cross remote calls.
Example:
The below example displays the technical error message that occurred as a Toast. In case a generic message should be displayed instead, it can be easily adapted.

```
/**************************************************************************************************************
* @author         Tanguy Charlier
* @date           2018-09-27
* @description    Generic function for server-side controller calls
*                 In case of success, the response return value is provided to the callback method that is executed
*                 In case of (any) error, a toast is shown with the error message received from the server
*
* @param          Object response: A response object returned by ther server
* @param          Function successCallback: A function called in case of success
**************************************************************************************************************/
handleControllerResponse : function(cmp, event, helper, response, successCallback) {
    if (response.getState() === "SUCCESS") {
        successCallback(cmp, event, helper, response.getReturnValue());
    }

    else {
        var errorMessage = '';
        var errors = response.getError();
        if (errors) {
            if (errors[0] && errors[0].message) {
                errorMessage = errors[0].message;
            }
        }
        else {
            errorMessage = $A.get("$Label.c.ATT_Unknown_Error");
        }

        helper.displayToast($A.get("$Label.c.ATT_Unexpected_Error"), errorMessage , "error");
    }
},
```

This generic helper method can be then invoked easily as following when performing a remote call to the server:
uploadAction.setCallback(**this**, **function**(response) { helper.handleControllerResponse(cmp, event, helper, response, **function**(cmp, event, helper, responseValue) {
//Code to Process the response
});});$A.enqueueAction(uploadAction);

Lightning Web Components

Error management for Lightning Web Components is discussed in section 10.4

Asynchronous Processes Error Management

In case of asynchronous process, **always catch the error** and apply the right error management strategy. In case you do apply this principle the whole job will go in error and processing of next batch (in case of Batch Apex) will be stopped. Store the outcome either in a custom error object or by sending an email:

- **Email Sending:** The approach consists to send an email to a dedicated person in case of issue. Downside here is that we need to avoid spamming in case too many errors are raised. This is also not ideal in order to keep an overview of the errors in a central place.
- **Custom Error Table (Recommended):** Mostly for errors that **will not be displayed in any UI**, it can be interesting to build a Custom Object and to store the unexpected errors in it. (A **Deloitte Nugget** exists already and can be re-used: ***LOG Messages***) See section 13.2.3 more info.

The **Custom Error Table** approach is not only **useful** for Asynchronous jobs, error management but also in different **other cases**:

- Integration Logs (Allowing to Debug the payload and to track errors)
- Complexes processes for which you would like to persistently log debug messages

IMPORTANT REMARK:
Logging a Debug Log is a persistent table is consuming DML transaction. Be sure that the mechanism in place to log messages offers bulk capabilities (e. g. through Unit Of Work or Queueing principle)

Trigger Handler Pattern

Trigger Development

It is common in a multi-developer environment that trigger logic grows unmanaged and companies end up with multiple triggers on a single object. This is problematic for the following reasons:

- Trigger classes are not executed in any guaranteed order
- Repetitive SOQL queries being executed
- Governor limits being exceeded
- Objects with multiple triggers can be very hard to debug

It is a best practice to have a single trigger handler class per object and then use a framework with factory pattern to invoke the trigger logic. Developers write their trigger logic according to the trigger framework and inject their logic into the right ***Domain*** class. This provides a clear point of execution for the trigger logic, it allows to control the order of execution of the logic, and provides the ability to execute upfront SOQL logic to cache data to be shared across triggers (if needed) (***Selector*** and ***XDom Selector*** principles).

Trigger Handler Framework(Must-Have)

Take into account the following simple framework rules when developing triggers and classes:

- No code logic in Trigger
- Encapsulate the business logic as much as possible (***Domain Driven Design*** is recommended)
- Allow Trigger or Trigger features to be muted
- Avoid duplicating queries

Based on project experience and best practices, Deloitte has implemented a Trigger Handler Framework supporting the above guidelines out of the box and helping the developer to apply the ***Domain Driven Design*** principles. (See section 3.6)
In order to get more information about the Trigger Handler framework, please refer to Section 13.2.1

Testing

Key Guidelines

When unit testing Apex code, the following principles should be applied:

1. **(Must-Have) Cover as many lines of code as possible**

   You must have at least 75% of your Apex scripts covered by unit tests to deploy your scripts to production environments. In addition, all triggers should have some test coverage. Salesforce recommends that you have 100% of your scripts covered by unit tests, where possible. However, one should aim to get an *as high as possible* Test Code Coverage. 90% Test Code Coverage is a realistic goal, since it is not always possible cover every line of code with unit tests.

   a. (**Recommended)** In case of conditional logic (including ternary operators), execute each branch of code logic.
   b. (**Recommended)** Make calls to methods using both valid and invalid inputs.
   c. **(Must-Have)** Complete successfully without throwing any exceptions, unless those errors are expected and caught in a try…catch block.
   d. **(Must-Have)** Use ***System.assert*** methods to verify whether the code behaves properly.
   e. (**Recommended)** Use the runAs method to test your application in different user contexts, whenever possible and appropriate.
   f. **(Must-Have)** Use the isTest method. Classes defined with the isTest annotation do not count against your organization limit for all Apex code.
   g. **(Nice-to-have)** Write comments stating not only what is supposed to be tested, but the assumptions the tester made about the data, and the expected outcome whenever appropriate. In straightforward unit tests, a high level explanation suffices.

   (Must-Have) Test the classes in your application individually. Never test your entire application in a single test. Create one Test Class for every Class in your environment. (The naming convention to follow is the following: *[ClassName]_Test*.)

   a. **(Must-Have)** WebService/HTTP callouts cannot be executed in a Test Class. In order to be able to test the callout correctly:

   - implement HttpCalloutMock interface for HTTP callout shttps://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_classes_restful_http_testing.htm

   For Http Callout the Integration Framework offers a generic class (***INT_SwaggerResponseMock***) implementing the ***HttpCalloutMock*** interface that can be used for mocking any of your service. (See Section 13.2.4)

   - implement WebServiceMock interface for SOAP callouts

   https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_callouts_wsdl2apex_testing.htm

   a. (**Must-Have)** Avoid using @seeAllData =true. Accessing data specific to a sandbox can result in deployments failing.
   b. **(Recommend)** Leverage on an existing Data Factory framework (e.g. UTT Unit Testing Framework) to create test data.

   (Must-Have) In case a method/property/variable must be publicly accessible from a test method, do not change the access modifier to public but use the ***@TestVisible*** annotation.

TIP Debug statements and Comments are not included in the Code coverage

Testing pattern

(Must Have)
In order to build test classes which are readable and maintainable, the below pattern should be applied when building a test class:

- Use a **Data Factory utility class/framework** in order to have a central place to generate raw test data. This factory class is a central place to create any type of data in order to ensure, data creation logic is applied in the same way everywhere in the application (e.g. providing default values for records). There is a ***Unit Testing Framework*** offers a nice approach to generate test data (See Section 13.2.4).

- Test data generation for a given test method must be encapsulated either:

  - Creating a method generating test data using the **@testSetup** annotation
    - Annotation used for creating common test records that are available for all test methods in the class.
    - More efficient since SF is creating the test data only once and restore it with a rollback mechanism at the start of each test method.
    - Down side is that you cannot store any record reference as static variables of your test class

  - Creating a method generating test data without using the @testSetup annotation
    - In this case, the method should be invoked at the beginning of each test method
    - The method should not be flagged as test method
    - It allows to store record reference as static variable of the test class

- Use ***Test.startTest()*** and ***Test.stopTest()*** methods an an appropriate way:

  - ***startTest*** can be invoked only once within a test method and is resetting the governance limits. It must be invoked once the test data have been generated and before starting the execution of the test as such.

  - ***stopTest*** must be invoked once the test has been performed and before the assertion checking the test result is done. In fact, all the asynchronous calls made are collected by the system and not executed. When ***stopTest*** is invoked, all the asynchronous processes are run synchronously.

- Apply the System assertion after the ***stopTest*** invocation in order to check the correct behavior of the functionality being tested.

Example:

1. Initialize the test data with a ***@TestSetup*** method and generate the data using a data factory framework (e.g. Unit Testing Framework) to generate raw test data.

```
/**************************************************************************************************************
* @author           Karolinski Stephane
* @date             2018-07-05
* @description      This method is a generating the test class test data
* @return           void
**************************************************************************************************************/
@TestSetup
private static void initializeTestData()
{
    Map<String, Object> fields = new Map<String,Object>
    {
        'Name' => 'My Mapping',
        'TRS_Mapping_Code__c' => 'myCode'
    };

    UTL_DynamicApex.createRecord( sObjectName: 'TRS_Mapping_Definition__c', fields, executeDML: true);
}
```

1. Create a test Method

```apex
/**************************************************************************************************
* @author           Karolinski Stephane
* @date             2018-07-05
* @description      This is the test method testing the Query engine happy flow
* @return           void
**************************************************************************************************/
@isTest
private static void  test_QueryEngine()
{
    Test.startTest();

    //Query the data with the Query Engine
    List<TRS_Mapping_Definition__c> queryResult1 =  (List<TRS_Mapping_Definition__c>) TRS_QueryEngine.executeQuery('SELECT Name, TRS_Mapping_Code__c FROM TRS_Mapping_Definition__c LIMIT 1');
    List<TRS_Mapping_Definition__c> queryResult2 =  (List<TRS_Mapping_Definition__c>) TRS_QueryEngine.executeQuery('SELECT Name, TRS_Mapping_Code__c FROM TRS_Mapping_Definition__c LIMIT 1');

    Test.stopTest();

    System.assertEquals(true, queryResult1 === queryResult2); //Same memory item
    System.assertEquals(1, queryResult1.size());
    System.assertEquals('My Mapping', queryResult1[0].Name);
    System.assertEquals(1, TRS_QueryEngine.queryCache.size());
    System.assertEquals(1, TRS_QueryEngine.queriesExecuted.size());
    System.assertEquals('SELECT Name, TRS_Mapping_Code__c FROM TRS_Mapping_Definition__c LIMIT 1', TRS_QueryEngine.queriesExecuted[0]);
}
```

Use a Mocking Framework

(Nice To Have)

Use Case

SF is offering the capability to mock easily a class behavior. This can be useful is several cases:

- Testing of features dependent on classes, which are difficult to run in a testing context due to some complexity such as the generation of test data or dependencies with other system.

- Building Unit Testing on the top of end 2 end testing which should run quickly and should avoid creating test data slowing down the process. In this case mocking the right classes might help simulating the end 2 end behavior.

While SF is offering out of the box the Apex Stub API, the capabilities are not so easy to be used and we recommend the user a Mocking framework sur as **ApexMocks** built on the top of the Apex Stub API. See: {+}https://github.com/apex-enterprise-patterns/fflib-apex-mocks+
The Unit Testing Framework is also providing a Light version of the fflib ApexMocks (***UTT_LightMock***) allowing to easily mock your classes without having the complexity of the full fflib framework. See **13.2.4 – Unit Testing Framework documentation** for more details.

How to create the Mock using the Apex Stub API?

In order to mock a class the below steps should be followed:

1. Create a class implementing the ***System.StubProvider*** interface. This class will stub the behavior of the given class you want to mock.
2. Instantiate a stub object by using the System.Test.createStub() method.
3. Invoke the relevant method of the stub object from within a test class.

See reference: {+}https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_testing_stub_api.htm+

Design Consideration when mocking

Given that this will be the responsibility of the test method to instantiate the class to Mock, it will requires some design consideration if you plan to use this feature for your test classes otherwise it will just not be possible to invoke the mock. A possible design approach could be to use a factory pattern responsible to instantiate the class. This factory will check if running in a test class context and will instantiate the mock instead of the real class in this case.
This pattern is implemented and exposed in the Trigger Handler framework (Service, Selector, Domain and Cross Domain Selector) and in the Common Library Framework (Unit Of Work) which allows to easily design your code in order to support mocking. (**See 13.2.4 – Unit Testing Framework documentation** (Leveraging a Mocking framework) for more details).

Lightning Component (Aura) considerations

This section does not have the purpose to provide a detailed guidance on building Lightning Component but is providing high-level recommendations:

Attributes:

Aura attributes are defining the state model of your client side controller in the same way as the properties/variable in a class. Always specify the ***access*** attribute (public, private, global) by being as restrictive as possible:

- Attributes which are used only within the component should be **private**
- Attributes that should be provided as input/output of the component should be **public**
- Attributes that should be used cross managed package should be defined as **global**

Attention point:
When a component is placed in a Lightning Page, SF forces to have all the attributes defined as public even if some are not input/output parameters.

Retrieving Data & Performing CRUDS (SOQL & DML operations)

When you Lightning Component should consider the use of **the Lightning Data Service** before writing your own custom controller logic to perform CRUDs (SOQL & DML operations) on record.
Key Benefits of using the Lightning Data Service (LDS)

- - CRUD & FLS are enforced out of the box
  - Data retrieved is shared cross components on the same page
    - Data retrieved only once
    - Change done by one component is automatically refreshed on the other components
  - When using ***lightning:xxx*** data service tags, UI is automatically rendered in Lightning without having to use the SLDS.

The below tags can be used:

Form Function	
Display, create, or edit records	`lightning:recordForm`
Display records only	`lightning:recordViewForm` (with `lightning:outputField`)
Create or edit records only	`lightning:recordEditForm` (with `lightning:inputField`)
Display, create, edit, or delete records with granular customization	`force:recordData`

Lightning Component Events

The Aura framework offers two types of events:

- *Component events:*

A component event is fired from an instance of a component. A component event can be handled by the component that fired the event or by a component in the containment hierarchy that receives the event.

- *Application events:*

Application events follow a traditional publish-subscribe model. An application event is fired from an instance of a component. All components that provide a handler for the event are notified.
As a guideline, always prefer ***Component*** events and use ***Application*** events only when absolutely needed (e.g. two fully independent component must exchange information).

Lightning Web Component Considerations

This section does not have the purpose to provide a detailed guidance on building Lightning Component but is providing high-level recommendations.

General Considerations

When (not) to use LWC

Although the LWC framework is a great tool for building apps on the Salesforce platform, we should always keep the application's requirements in mind. Whenever possible, choose LWC over Aura as technology as it follows the latest web standards and is the Salesforce-preferred development tool for Lightning.

Standard vs Custom functionality

When deciding to create a LWC or not, always check if the requirement can be achieved with a standard Salesforce functionality.

When a custom solution is needed, always choose base components over a fully customized UI. For example, when a form is needed for user input, use a record(-edit/view)-form instead of designing a custom form component. In fact, never build custom forms unless there is a strong use case for it (e.g. when fields of different objects need to be displayed mixed together or when custom branding is required).

In any case, prefer using built-in functionality over custom Apex-controller methods. Examples are the [get/create/update/delete]Record methods of the lightning/uiRecordApi standard module.

Base Components

As mentioned above, whenever possible use base components. These components are created by salesforce and look the same as standard salesforce components. They can be found in the lightning web components library. The component logic is already implemented by Salesforce, including responsiveness and accessibility. Salesforce maintains and updates these components.
Due to shadow DOM restrictions, it is not possible to extensively style the contents of a base component. However, limited styling can be done through styling hooks.

Lightning Design System Framework

If existing components don't have the required functionalities, look into the blueprints of the Lightning Design System. These blueprints are accessible HTML and CSS files used to create components in conjunction with Salesforce implementation guidelines. They don't contain any logic so it is required to build the logic.
Part of the provided Salesforce blueprints are Utilities. These can help to style a component without a CSS class.

Custom Components

If it is not possible to use a component from the component library or from the lightning design system, use a custom HTML and CSS file. In this case, every part of the design like the responsiveness of the solution need to be handled.
Regarding the CSS management, there are two options possible. These options can be used together.

Custom CSS theme component

To avoid duplicating code, create a custom CSS theme component. This component will act as a theme-css file containing all the branding definitions and the styling hooks. When styling is required, the custom developed component should import this css file.

Design tokens

Consider using the built-in design tokens, especially when designing for communities. That way, the look and feel of the component will resemble the overall design (e.g. when the color palette of the community is changed in the experience builder the design tokens are automatically updated).

Typical JavaScript Developer Mistakes

"this" context

In JavaScript the keyword 'this' is referring to the object it belongs to.
The behavior of 'this' can be difficult to understand and can lead to some unexpected behavior (i.e. 'this' in a function, in an event handler,…). Please refer to {+}https://www.w3schools.com/js/js_this.asp+ to fully understand the behavior of 'this'.

Arrow vs 'regular' function

In the different examples provided by Salesforce for the use of LWC, arrow functions are used. Arrow functions differ from the regular functions in several points. In order to understand these differences and avoid unexpected behavior, please refer to {+}https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions+.

Asynchronous JavaScript

There are two patterns to execute asynchronous JavaScript:

1. The *Promise* pattern, with *.then()*, *.catch()* and *.finally()*
2. The *async/await* pattern

Although async/await works **exactly the same** as promises (as it is just a syntactic sugar layer around promises), it is advised to stick to one of both patterns within one component. This will increase code readability and decrease code complexity and entanglement. However, it is extremely important to understand how these patterns work.
Before discussing some pitfalls, the understanding of two LWC core methods is required, explained in their respective order of execution:

1. ***connectedCallback***: a method that is fired when the component is inserted into the DOM
2. ***renderedCallback***: a method that is fired when the component has finished the rendering phase

A common pitfall in LWC is to initialize a component in the *connectedCallback* using async/await. As the *connectedCallback* is then annotated with the *async* keyword, it makes this method asynchronous, whereas it normally would be synchronous! This can highly complicate component logic as there is no longer a guarantee that the normal component lifecycle is executed in a synchronous way.
In practice, this might imply that *connectedCallback* is not finished (e.g. the asynchronous call for fetching data via Apex has not yet returned) before the component is being rendered and thus *renderedCallback* is being called. This in turn can lead to variables not being initialized when attempting to display

them in the HTML-template or when processing them e.g. in renderedCallback. In this case, consider using the Promise pattern over async/await. The following demonstrates this by putting the *Promise* pattern next to the *async/await* pattern.

```
export default class asyncAwaitTest extends LightningElement {
    @track data;
    @track otherData;

    connectedCallback(){
        console.log('connectedCallback init');

        this.doLongrunningCalculation().then(result =>{
            this.data = result;
            console.log('connectedCallback', this.data);
        });
        this.otherData = 42;

        console.log('connectedCallback end');
    }

    renderedCallback(){
        console.log('renderedCallback init');
        console.log('renderedCallback', this.data) // will be undefined
        console.log('renderedCallback', this.otherData) // will be 42
        console.log('renderedCallback end');
    }

    doLongrunningCalculation(){ ···
    }

    // PRINTS:
    // connectedCallback init
    // connectedCallback end
    // renderedCallback init
    // renderedCallback undefined
    // renderedCallback 42
    // renderedCallback end
    // connectedCallback -49999955000000
}
```

```
export default class asyncAwaitTest extends LightningElement {
    @track data;
    @track otherData;

    async connectedCallback(){
        console.log('connectedCallback init');

        let result = await this.doLongrunningCalculation();
        this.data = result;
        console.log('connectedCallback', this.data);
        this.otherData = 42;

        console.log('connectedCallback end');
    }

    renderedCallback(){
        console.log('renderedCallback init');
        console.log('renderedCallback', this.data) // will be undefined
        console.log('renderedCallback', this.otherData) // will be undefined
        console.log('renderedCallback end');
    }

    doLongrunningCalculation(){ ···
    }

    // PRINTS:
    // connectedCallback init
    // renderedCallback init
    // renderedCallback undefined
    // renderedCallback undefined
    // renderedCallback end
    // connectedCallback -49999955000000
    // connectedCallback end
}
```

Code structure

LWC Skeleton

To improve uniformity and readability of components, a generic template has been created to generate the boilerplate code of a LWC. This template contains a predefined code structure that the developer is advised to adhere to. The general component structure should follow this order:

1. Imports
    a. Salesforce metadata and standard functionality (labels, LWC built-in functions, ...)
    b. Apex methods
    c. (Custom) JavaScript modules
2. Variables
    a. Public reactive variables (@api)
    b. Private reactive variables (@track)
    c. Wired variables and functions (@wire)
    d. Non-annotated variables
3. Getters and setters
4. Lifecycle hooks
5. Event handlers
6. Business logic functions

A VS-code snippet exists to apply this structure. The importable IntelliJ live templates can be found here.

Installing and using the snippet in VS-code

To install the code snippet in vs-code, follow the following steps:

1. Select **User Snippets** under **File** > **Preferences** (**Code** > **Preferences** on macOS).
2. Select the language for which the snippet should appear (in this case, JavaScript).
3. Paste the code from the nugget into this file.

To use the code snipped in vs-code, follow the following steps:

1. Create your LWC
2. Delete prefilled content
3. Type ***dl-lwc***
4. While typing, two possible option will appear:
    - *dl-lwc-skeleton:* Select this one if you build a LWC with a UI
    - *dl-lwc-module-skeleton:* Select this one if you build a JavaScript only LWC

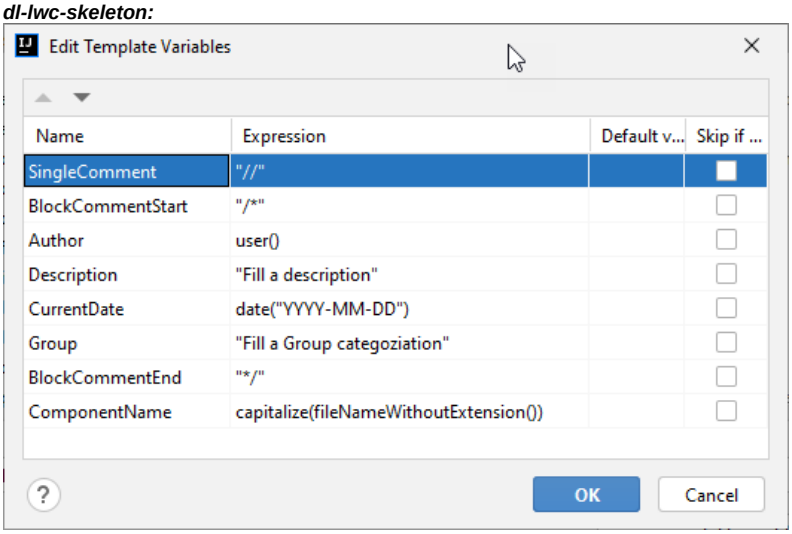
Installing the snippet in IntelliJ

To install the code snippet in IntelliJ, follow the following steps.

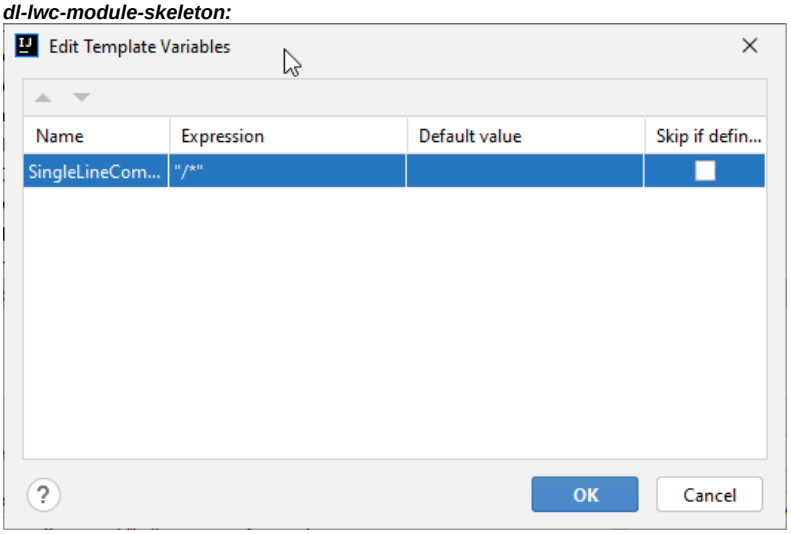
1. Open the snippets *Skeleton_live_template.txt* and *Module_live_template.txt* in **InteliJ**
2. For each snipet apply the following:
    a. Select all the snipet text **Cmd/Ctrl+A**

b. Select it and press **Cmd**/**Ctrl**__**Shift**__**A** and type *Save as Live Template*.
c. Add the abbreviation (*dl-lwc-skeleton* or *dl-lwc-module-skeleton*) and relevant description.
d. Change the *Applicable In* and select JavaScript (or EcmaScript)
e. Edit Variables and do the following setup:

dl-lwc-skeleton:



dl-lwc-module-skeleton:



1. a. Save the code snippet.

To use the code snipped in vs-code, follow the following steps:

1. Create your LWC
2. Delete prefilled content
3. Type **Cmd/Ctrl+J**
4. Select:
   - *dl-lwc-skeleton:* Select this one if you build a LWC with a UI
   - *dl-lwc-module-skeleton:* Select this one if you build a JavaScript only LWC

Lightning Web Component Naming conventions

For naming conventions, we refer to section 3.2.

Modularity

At all times, try to architect code in such a way that functionality can be abstracted for reuse. For complex components, it is often a good idea to split up logic in separate parts. This abstraction is orchestrated by creating JavaScript modules. A good example of a module is the reduceError.js module, which abstracts error handling and formatting functionality.
When the logic is separated into several modules, exposing functionality from these modules can be done in two ways:

1. Define one default export

2. Define (several) named exports

A best practice for modules is to define named exports instead of one default export object, because this enforces developers who use a module to reference the exported functionality by the same names.
Default export:
// myModule.js
export default { func1, func2 }
// myComponent.js
import myPreferredModuleName from "c/myModule";
Named export:
// myModule.js
export {func1, func2}
// myComponent.js
import {func1, func2} from "c/myModule";

Error handling

Differentiating error types

When performing a call to an Apex controller or using the Lightning Data Service directly, sometimes something goes wrong. Therefore, it is important to address each scenario according to the type of error that can occur.
There are some best practices in how errors are displayed. Two types of errors can be distinguished:

1. User-invoked errors
2. Application errors

Remember to apply a try-catch(-finally) block around code when it makes sense, i.e. when a certain piece of code might result in an error (e.g. a validation rule being triggered).
Be aware that there are some limits to try-catch in regards with asynchronous operations. Putting a global try-catch around your whole code won't catch all exceptions in this case.
To summarize:

- Using async / await enables the use of ordinary try / catch blocks around asynchronous code.
- Errors in a promise should be handled by using ".catch".
- If you chain promises, using one .catch will catch errors that occurred in any promises in the chain.
- When you raise an exception outside the promise, you must catch it with try/catch

User-invoked errors

A user-invoked error is an error which results from a user performing a certain action. It is a best practice to display user-invoked errors as a toast. The ***UI Libraries framework*** provides a way to display a toast without having to build it from scratch (See section 13.2.7).

Application errors

Application errors are errors that occur when something goes wrong during the initialization of the app. When this occurs, a more permanent error message needs to be displayed. Salesforce SLDS provides a template for a permanent inline error. It is recommended to build a dedicated, re-usable LWC to display those type of error which can then be re-used everywhere. See below an example on how errors could be managed using the ***static error message*** component.

```
<c-static-error-message if:true={hasErrorMessages} messages=
{errorMessages} />
```

```
@track errorMessages = [];

get hasErrorMessages(){
    return this.errorMessages.length > 0;
}

connectedCallback(){
    controllerMethod.then(...).catch
(exception){
      this.errorMessages.push(reduceErrors
(exception));
    }
}
```

Handling server-side errors

Server-side error handling happens very much in the same way as for Aura Components. There are two common approaches:

- Return a wrapper class to the LWC component to avoid DML rollbacks in case of an exception. See section 6.4.
- Throwing **any exception** in the controller class. The main difference here is that we no longer need to wrap the exception in an **AuraHandledException**, as the LWC framework can accept *any* exception coming from the controller. This makes for a much richer exception context client-side as we no longer only have a String containing the exception, but rather have access to the entire exception.

`@AuraEnabled`
`public static Decimal divisionByZero(Decimal numerator){`
`  Decimal dec;`
`  try {`
`  dec = numerator/0;`
`  } catch (Exception ex) {`
`  throw new AuraHandledException(ex.getMessage());`
`  }`
`  return dec;`
`}`
	`@AuraEnabled`
	`public static Decimal divisionByZero(Decimal numerator){`
	`  Decimal dec;`
	`  try {`
	`  dec = numerator/0;`
	`  } catch (Exception ex) {`
	`  throw ex;`
	`  }`
	`  return dec;`
	`}`

Handling and displaying errors in the LWC

ReduceErrors

Part of the ***UI Libraries framework***, the ***reduceErrors*** helper method can be used to reduce the received error object. This helper returns an array of all the error messages that have occurred. (See section 13.2.7)

ShowToastEvent

Part of the ***UI Libraries framework***, the ***showToast*** helper method can be used to display a toast. (See section 13.2.7)

Debugging

Debugging is an important part of the development process. The following debugging guidelines assume **Google Chrome** is being used as browser in order to leverage Google DevTools.

Debugging approaches

An commonly used debugging strategy is to leverage on the *console.log()* statements in the JavaScript code, and then inspect the values of the printed expressions in the console. While this is a valid approach for simple use cases, it comes with two important downsides:

1. Code clutter
2. No flexibility in what information is displayed at runtime

In contrast, a more robust debugging approach can be taken by leveraging the ***Google DevTools*** functionalities, which come prepackaged with the Google Chrome browser. More info on the debugging toolset can be found here: Javascript debugging reference and Debug Javascript

Which debug settings to activate

While developing a component, it is recommended to:

1. Enable **debug mode** for the users that will be working on that component (*Settings* > *Custom Code* > *Lightning Components* > *Debug Mode*). This setting will prevent the framework from minifying JavaScript code, allowing for easier debugging. When enabled, the DevTools console also displays additional LWC engine runtime warnings indicating when a bad (LWC-related) coding practice is applied.

Note: ensure that debug mode is disabled when development is done, as it has a negative impact on page performance.

1. Enable **custom formatters** in Google DevTools (*Settings* > *Preferences* > *Console*)LWC comes with a prepackaged custom JavaScript formatter. Enable this DevTools setting in order to leverage this formatter. By default, the LWC framework wraps @api variables in a proxy, which can be considered as a membrane around a variable making it read-only. The formatter unwraps the @api variable and allows the developer to inspect the true value of variable, instead of needing to drill down in the proxy-cluttered object.

We can distinguish three types of debugging:

- code debugging
- responsiveness debugging

- performance debugging

Code debugging

Code debugging relates to detecting and investigating any defects in a component's code using the *sources* tab of Google DevTools. Useful features that any developer should be aware of include:

- Step-by-step debugging: allows the developer to go through each line of code and inspect the state of the application at any point in time (i.e. variable values, call stack, …)
- Blackboxing scripts: allows you to skip stepping into specified scripts during the debugging process
- The 'Pause on caught exception' setting: pauses execution of a script when an exception is thrown to further investigate the current state of the application, instead of just showing the exception in the console.
- Watches: allows to configure expressions that are automatically re-calculated during the debugging process (e.g. a watch expression could be **typ eof a** or **a + b**, given that variables **a** and **b** exist in the application context)

Responsiveness debugging

It is frequent that a component needs to adjust to the screen size of the context in which it is rendered (e.g. a mobile device, a tablet or a desktop). This behavior needs to be tested to ensure no UI bugs are present in the component. To this extent, DevTools provides functionalities to test a component against any screen aspect ratio. Toggle the device toolbar to leverage these functionalities.
Note that this can be used to test different **screen sizes**, not different **device types:** As of summer '20, it is no longer possible to emulate the salesforce mobile app in a desktop browser, therefore toggling the device to a mobile device or tablet in DevTools will no longer launch the mobile app, but will instead open the current page in Salesforce Classic. If a component needs to be tested or previewed on an actual mobile device or tablet, it should be done on a physical device.

Performance debugging

Additionally, the toolbar also allows for two types of throttling: CPU throttling and network throttling. This makes it possible to emulate the application on a lower-tier device or even on a lower-tier network, enabling the developer to detect performance problems.

Data CRUD & Server side operations

There are two ways to Create, Read, Update and/or Delete the data (CRUD) from the front-end.

1. Lightning Data Service
2. Apex Method Execution

This section discusses the possible approaches and provides general considerations to keep in mind when it comes to asynchronous operations to the Server.
This section does not describe the different Apex access modifier and sharing models and their respective impact. This point is covered in section 3.9.

Lightning Data Service

As explained for the Lightning Components in the section 9.2 of this document, the Lightning Data Service (LDS) enforces the Sharing, the CRUD permissions and the FLS permissions automatically. From a high-level perspective, the Lightning Data Service can be compared to the Visual Force standard controller.
There are two ways to use the Lightning Data Service in a Lightning Web Component:

- Use some specific ***Lightning HTML tags***
- Use the ***lightning/ui*Api*** Wire adapters and functions

The LDS is the perfect approach in order to perform CRUD on records for simple cases.

Specific Lightning HTML tags

Some of the Lightning HTML tags are built on top of the LDS. As the LDS enforces the Sharing, CRUD permissions and FLS permissions automatically, there is no need for extra code when these tags are used. Whenever possible, prefer using those tags to read and modify Salesforce data in your components. An example of such a tag is '*lightning-record-form*'.

Lightning/ui*Api Wire adapters and functions

A second way of using the LDS is to leverage on the wire adapters and functions from the 'lightning/ui*Api' that Salesforce provides. Refer to the link below in order to get a detailed view of the available functions and their use:
{+}https://developer.salesforce.com/docs/component-library/documentation/en/lwc/lwc.reference_ui_api+.

Apex Method Execution

The Apex approach should be considered when the standard LDS approaches reaches its limits:

- Multiple records processing
- Complex business logic to be applied in a consistent way
- Any other limitation from the LDS
- Business logic not involving CRUD needs to be executed (e.g. Integration callout)

There are two possible approaches in order to invoke a server side Apex method from your LWC:

1. **Wiring**: Wire an Apex method to a function or to a property
2. **Immediate Apex**: Call an Apex method imperatively

Wire an Apex method to a function or to a property

In order to retrieve data, wiring an Apex method can be done to:

1. A JavaScript function
2. A JavaScript property

Wiring Apex method to a property allows less control management processing. Thus, this solution is **not recommended**. Always favor wiring an Apex method to a function.
Wiring a function or a method will require to flag your Apex method as ***@AuraEnabled(Cacheable = true)*** which has the key benefit to provide an out of the box caching mechanism. However the drawback is that it cannot be used to perform **DMLs**. This is definitely the preferred approach when retrieving data if standard LDS cannot be used.
In case the cache must be explicitly refreshed, use the ***refreshApex*** JavaScript method to force this refresh.

Call an Apex method imperatively

When neither LDS, neither Wiring are suitable for your use case, you should consider invoking an Apex method imperatively.
When?

- A DML action must be performed
- Requires a clear control on when the invocation should occur

In this case invocation is made asynchronously using a promise and the *then().catch()* syntax
e.g.

```
handleLoad() {
        getContactList()
                .then(result => {
                        this.contacts = result;
                })
                .catch(error => {
                        this.error = error;
                });
}
```

Important Considerations

Use Dynamic parameters for Wiring

When invoking a method (LDS or Apex) using the wiring technique, it is recommended to use ***dynamic parameter(s)*** if you know that this parameter might change over the life cycle of the LWC. The wired server side method will be re-invoked automatically when the dynamic parameter is changed.

Wired JavaScript methods are invoked at LWC initialization

When you wire an Apex method to a JavaScript method, it is important to know that the method will be invoked at the LWC initialization (with only null parameters) before the server side call is done. It will then be invoked a second time when the server feedback is received. This means that you should be sure that the JavaScript is not doing any action if the ***data*** and the ***error*** parameter returned are null.

User Experience

In order to ensure a good user experience, it is important to display a spinner when the JavaScript is waiting for an answer (from the back-end but also from the front-end). Depending of the use cases, the spinner could:

- be displayed on the whole screen in order to prevent any action to be performed by the user.
- be displayed only in a component in order to show that some asynchronous process are running without preventing other actions to be performed by the user.

Boxcarring

When a lot of components need to load at the same time, or a lot of data need to be fetched in parallel, be mindful of a phenomenon which is called ***boxcar ring***. When more than six[1] *near-simultaneous* calls to the server occur, only the first five are processed in parallel. All additional calls will be placed in a *box car* and are sent as a bundle to the server. For example, this means that when calling 10 apex methods in parallel only the first five will be executed separately. The other 5 requests will be boxcarred and will only return a result once all 5 requests have completed.
Be aware of this behavior when splitting calls to the server. How boxcarring on the Lightning Platform works is also explained in full by the salesforce team here.
[1] six is the amount of requests the platform can separately execute in parallel at the time of writing.

Communication between components

Intercomponent communication

Communication between components that are unrelated was previously enabled by the pubsub.js library that Salesforce provided. However, this library has been replaced with the ***Lightning Message Service (LMS)***. It is strongly advised to no longer use the pubsub.js library, unless when facing limitations: LMS Limitations. Use Lightning Message Service to seamlessly communicate between LWC, Aura and Visualforce.

Intracomponent communication

Communicate from parent to child

Parents should only interact with children through ***@api*** annotated variables (with or without setters/getters) and functions. **Do not leverage LMS** for this type of communication, as it will impact the performance of the components.
If you need to handle the variables value in the child component, you can either use @api functions or use @api setters/getters. This allows to decouple the child from the parent as you can modify the values handled in the child without having to change the parent. When using an @api setter, a corresponding getter should be used. Only one of the two should be annotated with @api. If both are annotated with @api, an error will be thrown. The usual convention is to annotate the getter.

Getting updates from child components

Aura components support direct mutation of a variable passed in via the parent (called 'bidirectional binding'). In Contrast, LWC enforces that properties passed from parent to child via an @api property are immutable in the child component. This is called 'unidirectional binding'. Attempting to update such a property will result in an error. Therefore, there are two approaches to get data from a child component:

- Publishing events from child to parent
- Leverage an @api annotated getter or function

Publishing events from child to parent

Children can communicate with their parent through events, and these events can be configured in several ways:

- Event.bubbles: a Boolean value indicating whether the event bubbles up through the DOM or not. Defaults to false.
- Event.composed: a Boolean value indicating whether the event can pass through the shadow boundary. Defaults to false.

Always consider the propagation of events that are being dispatched! It is almost never a good idea to dispatch a composed event, as this type of event crosses the shadow DOM and thus exposes the event to the entire DOM. Instead, choose for bubbled events when the event needs to bubble up through the components.
Make sure to understand the nuances between the different types of events, as to apply them correctly and efficiently. When unsure, read up on this article.

Leverage an @api annotated getter or method

When a parent component is not interested in continuous updates from a child component, an option is to only go and request data when it is needed, instead of subscribing to child component events. To this extent, an @api annotated getter or method can be used.

Protect your application against vulnerabilities

This section is describing the main vulnerabilities that could lead a hacker to perform unauthorized actions within the implemented application.
This section is covering the following vulnerabilities:

- Cross Site Scripting (XSS)
- SOQL Injection
- Open Redirect
- Cross Site Request Forgery (CSRF)
- Clickjacking

Cross Site Scripting

Cross Site Scripting (XSS) is an injection vulnerability that occurs when an attacker can insert unauthorized JavaScript, HTML, or other active content into a web page. When other users view the page, the malicious code executes and affects or attacks the user.

Example

Let's illustrate a XSS attack with the **following example**:
Apex Controller initialization

```
basicText = apexpages.currentpage().getparameters().get('text');
outputText = basicText.replace('\r\n','<br/>');
```

VF Page

```
<apex:page controller="XSS_Basics_Demo" sidebar="false" tabStyle="XSS_Basics_Demo__tab">


    [...]


    <script>
        document.getElementById('{!$Component.textOutput}').innerHTML = '<p>{!outputText}</p>';


    </script>
[...]


</apex:page>
```

This code is subject to vulnerabilities. It takes a parameter provided by a user and then display it in a specific place using JavaScript. What will happen is now the parameter provided is the following:

```
<img src=x onerror="alert(\\\'I said, HEAR YE, HEAR YE, COME ONE, COME ALL!!\\\');"> </img>
```

Suddenly, some java script will be executed when displaying the page. In the above example a simple alert box is displayed but imagine what a hacker could do in case he would like to damage the application.

Common XSS mitigations

Input filtering

Input filtering works on the idea that malicious attacks are best caught at the point of user input. If the user inputs ***duck*** and the page strips or blocks the code, then no unauthorized code runs.
There are two types of input filtering.

- ***Blacklisting*** — Specific "bad" characters or combinations of characters are banned, meaning they can't be entered or stored. The developer creates a list of known bad characters (such as HTML or script tags) and throws an error if any bad characters are in the input.
- ***Whitelisting*** — Only characters or words from a known list of entries are permitted, preventing malicious input. For example, if the user enters anything besides numbers in a phone number field, the application throws an error.

Output Encoding

While input filtering techniques work by preventing malicious data from entering the system, output encoding techniques take an opposite approach: They prevent malicious payloads already in the system from executing. In fact, output encoding is often considered more necessary than input encoding because it doesn't rely on any upstream or downstream protections, and it can't be bypassed by alternative input pathways.

How to prevent XSS vulnerabilities via Output Encoding?

Automatic HTML Encoding (Pre-Built within SF)

Salesforce automatically HTML encodes any values and merge fields placed in HTML context. This includes all standard functionality, as well as Visualforce pages and Lightning components.
Example in a VF page

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

Imagine that someone tries to pass some html tags in the username parameter such as: ***<script>.*** It will be automatically escaped by SF to ***<script>*** so there is no risk browser can start rendenring the HTML tags.
Remark:
HTML encoding is automatically applied only when binded variables are located in a *HTML context*. In case the binded variable are used in a *Script Context* no automatic HTML escape is performed.

Explicit HTML encoding

In case, you explicitly do not escape HTML encoding:

- Using the ***escape="false"*** attribute of the ***apex:outputText*** tag.
- Building the HTML dynamically via javascript

In this case, be sure to explicitly HTML escape the parameters received as input leveraging the following methods:

- Apex String method: *escapeHtml4()*
- VF Page: *HTMLENCODE*

Explicit JavaScript escaping

In case, you are using input parameters directly within your JavaScript or with a JavaScript execution context embedded with an HTML context, you are exposing your code to a possible script injection.
e.g. Script used in the HTML context

```
<div onclick="console.log('{!$CurrentPage.parameters.userInput}')">Click me!</div>
```

In order to avoid any risk of injection, in this case you need to escape some of the JavaScript characters. It can be achieved in the following way:

- Apex: String method: *escapeEcmaScript()*
- VF Page: *JSENCODE()*

SOQL Injection

SOQL is an injection vulnerability that occurs when an attacker can modify a SOQL query in an unauthorized way by injection addition SOQL query clauses. In this way the hacker could potentially access information he should not have access to.

Example

Imagine that a developer is building a dynamic query in Apex where some of the information used to apply a filter on the records to be retrieved has been input from the user interface. In our example the ***textualTitle*** variable.

```
whereClause = 'Title__c LIKE \'%'+textualTitle'%\' ';
records = database.query(query+' WHERE ' + whereClause);
```

Imagine now that, instead of providing the title to query on, someone provides the following input:

```
%' AND Performance_rating__c<2 AND name LIKE '%
```

This will result in executing the following query:

```
SELECT …. FROM … WHERE Title_c LIKE '%%' AND Performance_rating_c<2 AND name LIKE '%%'
```

This will allow the hacker to retrieve information he normally should not have access to which obviously represents a security breach.

Prevent against SOQL injection

In order to avoid SOQL injection, different options are available to developers:

- Static Queries and Bind variables
- Escape Single Quotes
- Type Casting
- Replace Characters
- Whitelisting
- Leverage the *QueryBuilder* library

Static Queries and Bind variables

Static queries using bind variable is a functionality offered out of the box by SF and is robust against SOQL Injection. You should always prefer this approach when possible.

Example of static Query using bind variable to manage query parameters

```
String var = ‘Amanda’;

queryResult = [SELECT id FROM Contact WHERE firstname =:var];
```

This approach is easy but is subject to some limits in term of capabilities. In case you reach those limits, you will probably have to perform Dynamic Queries using the *Database.query* method which is not protected against SOQL injection. In this case apply the below techniques in order to secure your application.

Escape Single Quotes

When input parameters are of type *String* and are used as part of the WHERE clause of the SOQL in order to filter out data, an easy to protect your code against SOQL injection is simply to escape single quotes from the received parameter.
In order to escape single quote from the parameter, simply use the method: ***String.escapeSingleQuotes(String)***
Looking back to our example of injection and applying the escapeSingleQuotes method:

```
whereClause = 'Title__c LIKE \'%'+String.escapeSingleQuotes(textualTitle)'%\' ';
records = database.query(query+' WHERE ' + whereClause);
```

This will result in executing the following query:

```
SELECT …. FROM … WHERE Title_c LIKE '%%\' AND Performance_rating_c<2 AND name LIKE \'%%'
```

This query will now be secured and will probably return no data since it will search for records having the *Titlec* field containing the string: *' AND Performance_ratingc<2 AND name LIKE '*

Type Casting

The previous mechanism of escaping single quotes is working only in case the value in the WHERE clause is of type String and therefore encapsulated between single quotes. For non-String type this is key to always explicitly cast it to its real type (converting the string input received) in order to avoid any risk of injection.

Replace Characters

While the Type Casting and Single Quote escaping is efficient for protecting against SOQL injection for parameters within the WHERE clause, it might not be used, in case you dynamically generate, based on user input, the fields to query or to filter on or the object name from the FROM clause.
In this case, a good technique is to replace unexpected characters in order to ensure that in case of injection tentative, SOQL will go in error.
e.g. Removing white spaces from the object from the FROM clause

Whitelisting

Another way to prevent SOQL injection is whitelisting. Create a list of all "known good" values that the user is allowed to supply. If the user enters anything else, you reject the response

Leverage the QueryBuilder library

In case you need to perform Dynamic queries, we recommend you to use the ***QueryBuilder*** library part of the ***Common Utility framework***. This QueryBuilder is robust against most of the SOQL injections by applying the above described mechanism. (See Section 13.2.2)

Open Redirect

Open redirect (also known as "arbitrary redirect") is a common web application vulnerability where values that are controlled by the user determine where the app redirects. It can be exploited by a hacker in order to redirect the user to a malicious web site without have the end user noticing.

Example

Here's an example of a vulnerable application.
https://www.vulnerable-site.com?startURL=https://www.good-site.com
In this URL, the vulnerable-site.com application should automatically redirect the user to www.good-site.com as the page loads. Seems pretty reasonable, right? However, what if an attacker changes the URL to this?
https://www.vulnerable-site.com?startURL=https://www.evil-hacker.com

An example of attack could be a redirection to an attacker web page that looks like to the SF Login page. End user without noticing the malicious redirection could enter his credentials which will be stolen by the attacker.

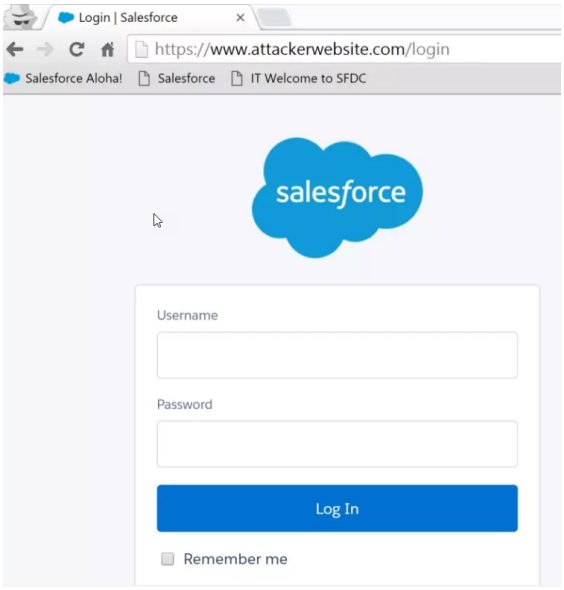


Figure 3 Attacker web site with SF Login Page Look & Feel

Standard SF Redirections

The below parameters are used as standard SF redirections:

Parameter	Usage
`startURL`	Used to redirect users to a location on page load
`retURL`	Used to redirect users to a location when they click the **Back** button
`saveURL`	Used to redirect users to a location when they click the **Save** button
`cancelURL`	Used to redirect users to a location when they click the **Cancel** button

Standard SF URL redirection are fully protected against malicious redirections on all the standard SF pages and partially when used through VF pages and Apex. Therefore we strongly recommend to not use the standard SF URL redirections within your custom pages without applying specific measures.

Prevent Open Redirect Vulnerabilities

In order to prevent malicious open redirects the following approaches can be taken:

- Hardcode Redirects
- Force Local Redirects Only
- Whitelist Redirects

Hardcode Redirects

This consists in designing your application avoiding to get the redirect URL from a user input when possible and to hardcoded in your controller logic.

Force Local Redirects Only

As long as your redirection are redirection within the SF platform, you should never accept full URL but only allow local URL (e.g. /001/o).
In order to do so the first slash of the URL must be hardcoded in order to avoid any tentative of redirection to a non-local URL:
e.g. In a VF page context

```
providedRelativeURL = ApexPages.currentPage().getParameters().get('onSave');
if(providedRelativeURL.startsWith('/'))
{
 providedRelativeURL = providedRelativeURL.replaceFirst('/','');
}
PageReference ref = new PageReference('/' + providedRelativeURL);
```

Whitelist URL Domains

In case you need to support redirection to a non-local URL coming from a input parameter, be sure to whitelist the allowed domains and reject any URLs not matching those domains.

Cross Site Request Forgery (CSRF)

CSRF is a vulnerability where a malicious application causes a user's client to perform an unwanted action on a trusted site for which the user is currently authenticated.

Example

Let's assume that an end user is currently logged in within the SF platform. Now assume that he is also accessing, without knowing, to a malicious website (e.g. by clicking on a hyperlink received in an email) and that malicious web site returns an HTML which will indirectly request access to a SF resource. Access to the resource will be granted since user is authenticated to SF. See below the example illustrating this:

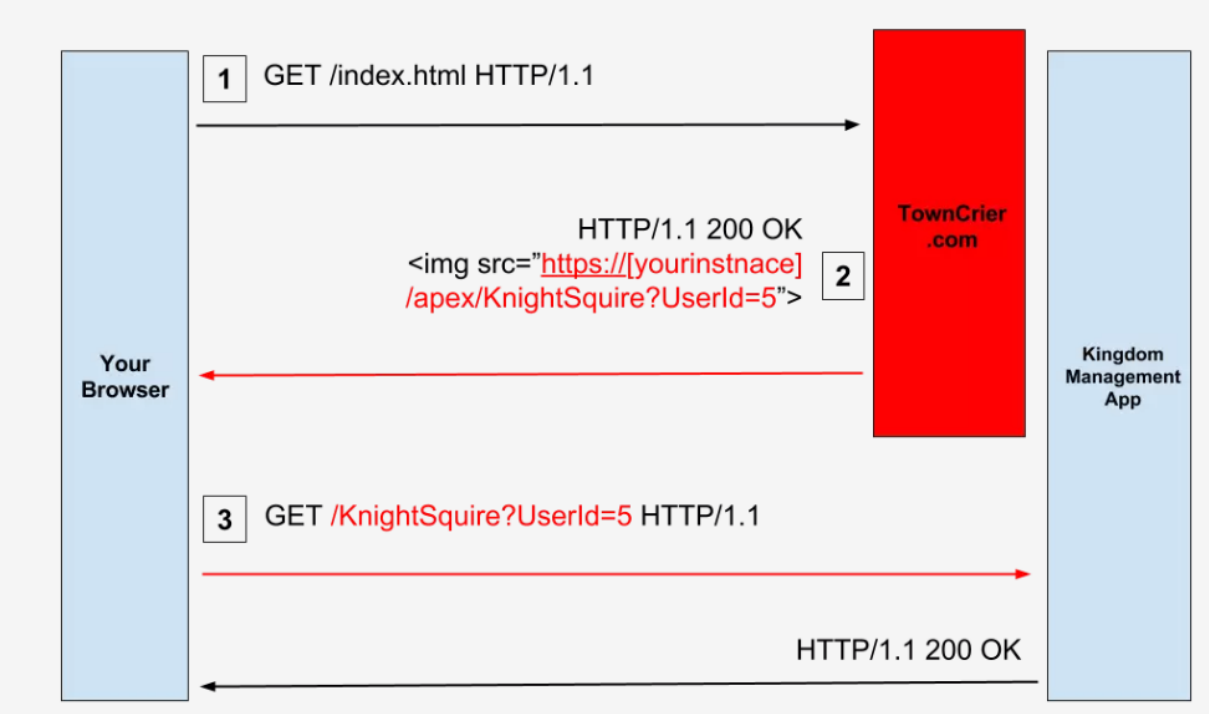


Figure 4 Vulnerable Website against CSRF

Prevent Cross Site Request Forgery

A way to prevent the unauthorized access to the SF page would consists in having the URL parameter expecting an additional token parameter which cannot be guessed by the eventual hacker and reject any request where the token has not been provided.

Figure 5 Web Site implementing a CSRF protection

Standard SF mechanism against CSRF

SF offers standard CSRF protection for **all their standard pages** by implementing the token mechanism described above.
Visualforce pages are protected against CSRF **only for POST** requests typically used when performing callback from the page to the server after the page has been initially loaded. GET requests are by default not protected against CSRF but can be enabled from the VF page settings. However enabling this protection will lead to some restriction of use described below.

POST Action - CSRF Protection mechanism

The VF page life cycle is the following:

- User access initially to the page through a GET request
- Page is initialized (Constructor and Init action are invoked)
- CSRF Token is generated
- Page is rendered on client side
- Once the user perform any action on the page that needs server interaction, the CSRF token is provided in the request to the server and this is how the protection is achieved.

This mechanism is directly highlighting a weakness. The first time you access to the page is not protected and is therefore subject to CSRF attack. **How can we mitigate this?**
There are two possible approaches:

- Be sure to not perform an DML within the Constructor/Init action in order to avoid any risk of damage in case of CSRF attack
- Enable the GET CSRF protection (with the associated consequences and limitations)

GET Action – No DML

By avoiding any DML action during the invocation of the VF page constructor or initialization action, you are mitigating the risk in case of CSRF attack and you are safe. This is the **preferred approach** whenever it is feasible.

GET Action – CSRF Protection mechanism

Enabling the CSRF protection on your VF pages for the GET action will completely secure your page. However it will have the consequence that your page will not be accessible directly having the user typing the URL directly in his internet browser or via hyperlink or client side redirection.
ATTENTION: Building an URL button within SF redirecting to the page will just not work anymore.
How do you access the page in this case?
In case CSRF protection has been activated for GET action, the only to be able to access this page will be to perform a redirection built from the Server side (Apex Controller). Concretely, this will mean that you need to build an intermediate VF page requesting a confirmation and displaying and apex ***Comm andLink.*** This will ensure that the end user confirms manually the action and the CSRF token will be automatically generated by SF when redirecting the page on server side.
Example of implementation with an apex CommandLink
The action *knightSquire* is performing the redirection server side

```
<apex:commandLink value="Knight This Squire" action="{!knightSquire}">
     <apex:param name="accId" value="{!person.id}" assignTo="{!currPerId}"/>
</apex:commandLink>
```

DO NOT: avoid implementing an output link as below since in this case redirection will happen client side and access to the page will not be granted

```
<apex:outputLink value="/apex/CSRF_Demo?UserID={!person.Id}"> Knight This Squire </apex:outputLink>
```

Clickjacking

This attack is used by malicious actors to trick users into thinking that they are interacting with one object while they are actually interacting with another. On a clickjacked page, the attacker displays some benign content to the user while it loads another page on top in a transparent layer. On the clickjacked page, the users think they are clicking buttons corresponding to the bottom layer, while they are actually performing actions on the hidden page on top.

Example

The example below illustrate and example of clickjacking where the attacker included an ***iFrame*** in this page that references a page where sensitive actions can be performed. However, as a user you don't see this form because the attacker cleverly set the transparency of the iFrame to 0, rendering it invisible. Next, the attacker modified the CSS properties of the iFrame to position it directly on top of the button.
When a user clicks the button, the user is actually interacting with the transparent iFrame above it. Resulting in the sensitive action being executed without the consent of the user.

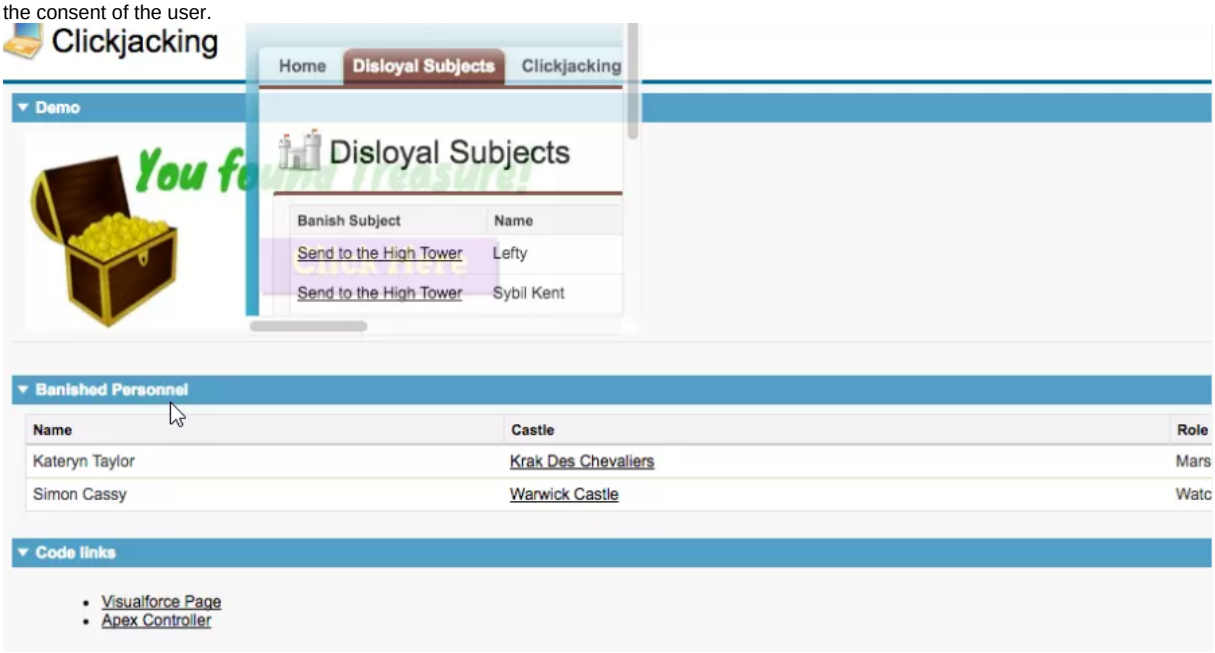


Figure 6 Example of Clickjack attack

Preventing Clickjack attacks

In order to prevent Clickjack, the web application should simply not allowed to be rendered within an iFrame. SF is providing out of the box protection for all the standard pages. By default protection for the Visualforce page is not enabled but this can be done from the ***Session Settings*** menu.
When enabling this Clickjacking protection, by default iFrames to a VF page within the Salesforce domain is allowed. **All the other domains** will **not** be **all owed** but domains can be whitelisted if needed.

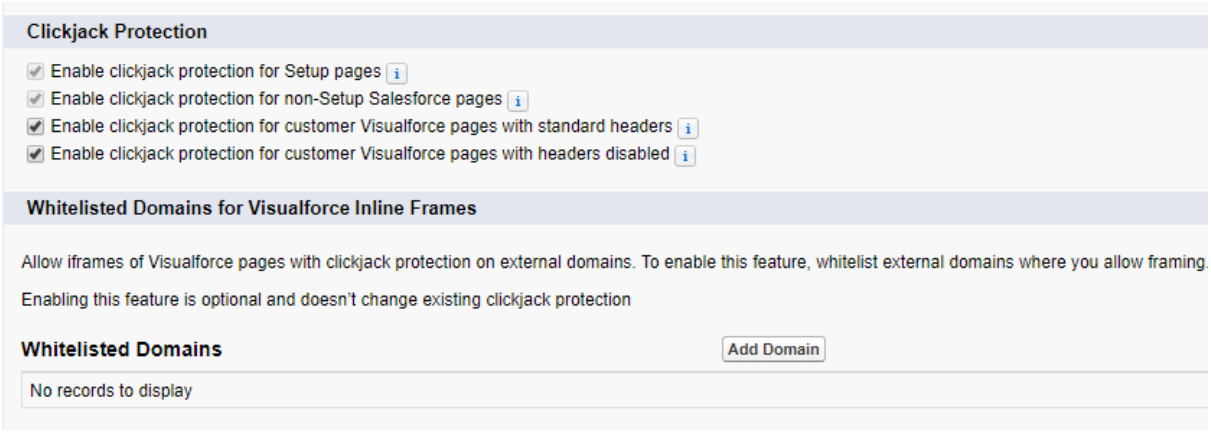


Figure 7 Setup of Clickjack Protection

Integration Consideration

This section has not the purpose to provide an exhaustive documentation on integration patterns but is providing some guidance on key recurring elements.

Integration Capabilities

Standard SF APIs

- ***Standard SOAP API***

Provides an exhaustive set of operations allowing to mostly manage data within SF. (Insert, Update, Upsert, Query, …). It must be used for ***low/medium volume*** of data.

- ***Standard REST API***

Provides an exhaustive set of operation allowing to mostly manage data with SF in the same way as the SOAP API. REST API is mostly suited for ***low volume*** of data and application not supporting the import of a WSDL easily.

- ***Bulk API***
  This API is suited for performing CRUD operations on a very large volume of record. This API is REST based and allows delegating the execution of the operation to the SF server.
  - **Pros:**
    - Allows parallel execution of a give batch (speeding up the execution)
    - Benefit of the calculation power of the SF server
  - **Cons:**
    - Parallel processing introduce the risk of record locks
    - Need to order records by its parent in order to reduce the risk of locks
    - More suitable for migration activities

- ***Metadata API***
  This SOAP based API allows managing any operation related to SF Metadata. Mostly used by IDE vendors and rarely in the context of integration.

- ***Tooling API***
  The tooling API exists in SOAP or REST and offers some basic metadata management capabilities and provides also capabilities to:
  - Run & Execute test classes
  - Execute Anonymous Apex
  - …

  This API is mostly used by IDE vendors and rarely in the context of integration.

- ***Streaming API***
  This API is allows to implement an event driven integration approach. The API is based on a long running http request protocol (Bayeux Protocol – CometD implementation) allowing an external application to subscribe to a given event (channel).
  Three type of channels are available:
  - ***Push Topics***
    Allows capturing Create/Update/Delete DML operations performed on a given SF sObject.

- ***Platform Events***
  Allows capturing custom events published from the SF platform.
- ***Change Data Capture***
  Similar to the Push Topics, the CDC allows capturing any database transaction performed on a given sObject. It provides more insights than the push topics and is intended to be used by BI systems in order to help them to perform data reconciliation.

Attention points:

- Leverage at maximum on ESB layer supporting this API out of the box. Implementing a custom implementation of CometD might be effort consuming.
- Be sure to implement the ReplayId extension in order to be able to retrieve lost events in case your application was down and not listening
- Depending on the event type, SF is currently storing the event for max 3 days. This means that you need to ensure that your ESB layer will not be down for longer period.

Outbound Messaging

SF is providing an out of the box capability in order to perform ***asynchronous SOAP callout*** to an external system. The key advantage is that it allows building easily integration within SF without any single line of code. (trigger via a workflow)
Considerations:

- Security can be enforced only via certificate signature
- Only fields available directly on the object related to the workflow can be provided in the payload. To overcome this limit, use Outbound messages as a notification pattern and have the ESB layer retrieving detailed information in SF via the standard SOAP/REST API
- SF is providing the WSDL designing the Service to be exposed.
- The service cannot return any information to SF
- The service must be be bulkified
- Cannot be invoked synchronously

In case you consider using the Outbound Message functionality, consider using the ***Integration Message nugget*** (See Section 13.2.6) which provides a custom event bus table within SF, sending outbound messages when a record is inserted and having out of the box retry mechanisms in case of failure. This approach is a good alternative to Platform Events in case those ones cannot be easily implemented due to technical constraints.

Apex REST/SOAP Callouts

SF allows to perform synchronous callout in Apex consuming a SOAP or REST Service. While considering this approach keep into account the following:

- Apex callouts cannot be performed synchronously within a Trigger
- For SOAP callouts, WSDL can be imported into SF but have some limitations:
  - No ***import*** supported: the WSDL cannot contain URL reference and should contain the whole definition
  - ***Extension*** syntax for xsd is not supported
  - Other undocumented limitation are existing
  - SOAP payload are not accessible and cannot be logged for debugging

- Prefer REST against SOAP if possible due to the above limitations
- If REST specification is provided under the form of a Swagger yaml, SF classes can be easily generated from the Swagger Editor.

Security:
In case you are performing SOAP/REST Apex callout, always use ***NamedCredential*** in order to store your credential and manage the security. Avoid using Custom Settings or Custom Metadata types to store sensitive information since they are publically available.
Framework
Leverage on the ***Integration Framework*** (See Section 13.2.4) for SOAP and REST callouts.
The framework supports out of the box:
For SOAP and REST:

- The use of NamedCredentials
- An automatic detection of the environment selecting the right integration credential to be used.

For REST only:

- Automatic logging of the payload and error in the Log Message table
- Automatic serialization/deserialization from Class to JSON and JSON to Class.
- Custom mapping can be defined
- Automatic generation of the Apex Classes from the Yaml specification using the Swagger Editor

Apex REST/SOAP Custom web services

SF allows extending the existing standard REST and SOAP API using Apex programming. This should be considered only when requirements are such that it cannot be achieved via the SF standard APIs.
Guidelines and best practices:

- Standard APIs cannot be used because advanced business logic must be exposed as part of the Service. ***e.g. Custom Search Engine exposed via API***

- Standard APIs cannot be used because there is a strict requirement the enforce data consistency cross entities and to perform a full roll back cross entities when something went wrong. (***e.g. Process creating several objects depending of each other: creation of an Order and Order Lines***)
- Service exposed should ***not have bulkification requirements***. Usually, you will go for a custom Apex web service only to cope limits of the standard APIs. This means, that most probably you will end up in one of the above use case. Having bulkification implemented as part of a custom apex web service is challenging for the following reasons:
    - SF is subject to governance limits. If your service is creating several objects and that you try to bulkify it, chances that you will reach the SF governance limits will be high.
    - Usually, when exposing an Apex Web Service managing the creation of several objects, you will ensure data consistency by rolling back all the objects created in case of error. However, if you introduce a request in Bulk, this will make it very complex and almost impossible to manage partial roll back in case only one record of your bulk is causing an error.
- When exposing a REST service, never thrown an unhandled exception. In case of exception, this one should be caught correctly in the main web service method and this one should decide to return the appropriate status code (500, 400, …) and probably provide the error info as a JSON within the Response Body.

```
/**************************************************************************************************
* @author           Karolinski Stephane
* @date             2018-08-22
* @description      This method expose a REST GET custom apex service which will download a file from the external
*                   document management system via API callouts. Using the approach will limit the file size to 4MB due
*                   to the heap size callout limit.
* @param            documentId (GET PARAM) (String):     The file unique reference from the external document management system
* @param            className  (GET PARAM) (String):     The full name of the Apex class implementing the ATT_COMLayer.IManageAttachmentsIntegration interface to be used in order
*                                                        to physically connect to the external document management system.
* @return           void
**************************************************************************************************/
@HttpGet
global static void downloadAttachment()
{
    String documentExternalId = System.RestContext.request.params.get('documentId');
    String classNameForComLayer = System.RestContext.request.params.get('className');

    try
    {

        //1) Check that the needed parameters are provided
        if (String.isEmpty(documentExternalId) || String.isEmpty(classNameForComLayer))
            throw  new InvalidURLFormatException('Invalid GET Parameters received');


        ATT_ComLayer.AttachmentFull attachment;

        //2) Invoke the COM Layer
        Type comLayerObjectType = Type.forName(classNameForComLayer);
        ATT_ComLayer.IManageAttachmentsIntegration comLayer = ( ATT_ComLayer.IManageAttachmentsIntegration) comLayerObjectType.newInstance();

        attachment = comLayer.downloadAttachment(documentExternalId);

        //3) Send the attachment as response
        System.RestContext.response.statusCode = 200;
        System.RestContext.response.responseBody = attachment.fileContentAsBlob;
        System.RestContext.response.addHeader( name: 'Content-Type', attachment.contentType);
        System.RestContext.response.addHeader( name: 'Content-Disposition', value: 'attachment; filename="' +  attachment.fileName  + '"');
        //System.RestContext.response.addHeader('Content-Disposition', 'attachment; filename="' +  EncodingUtil.urlEncode(attachment.fileName, 'UTF-8') + '"');
    }
    catch (Exception e)
    {
        System.RestContext.response.statusCode = 500;
        System.RestContext.response.addHeader( name: 'Content-Type', value: 'application/json');
        Map<String, String> errorMessage = new Map<String, String>{
                'errorStackTrace' => e.getStackTraceString(),
                'errorMessage' => e.getMessage(),
                'errorType' => e.getTypeName()
        };

        System.RestContext.response.responseBody = Blob.valueOf(JSON.serializePretty(errorMessage));
        System.debug(JSON.serializePretty(errorMessage));

    }

}
```

Figure 8- Example of error management for custom REST Apex web service

Frameworks

Introduction

This section describes the key tools & accelerator frameworks available, which we recommend to be used on our different projects. Those framework and related documentation are available in a ***private Azure Dev Ops repository*** owned by Deloitte BE.

Access to the repository here: {+}https://dev.azure.com/DTT-BE-DTC-CM-DC/_git/Nugget%20Org%20BE+

Should you not have access, please contact Karolinski Stéphane (skarolinski@deloitte.com)

Available frameworks

Trigger Handler

(Must Use)
This framework offers a full Trigger Handler framework, which **must** be used in all our new SF projects. This version of the framework introduces the concepts of ***Domain Driven Design*** and enforces developers to apply those concepts as much as possible.

Documentation Location	/Documentation/ TriggerHandler.docx
Components Prefix	***TRG***
Use case	Must be installed for all the projects and be used as framework when implementing any Trigger logic

Common Library

(Must Use)
This framework provides a set of utility classes of common methods usually needed on cross implementations.

Documentation Location	/Documentation/ CommonLibraries.docx
Components Prefix	***UTL***
Use case	Must be installed for all the projects and provided method should be used on ad hoc base instead of rebuilding it each time.

Log Message

(Must Use)
This Framework offers the capabilities to persistently log messages in a table so it can be used for monitoring and debugging. This framework is used by most of the other frameworks such as the integration ones.

Documentation Location	/Documentation/ Log_Messages.docx
Components Prefix	***LOG***
Use case	Must be installed for all the projects. It is used for debugging and monitoring purpose.

Unit Testing Framework

(Must Use)
This Framework offers a consistent approach in order to **generate test data** in a simple and efficient way. It also offer a **Light Mocking framework** inspired from the fflib ApexMocks one in order to easily mock the behavior of your Services/Domain/Selector.

Documentation Location	/Documentation/ UnitTestingFramework.docx
Components Prefix	**UTT**
Use case	Must be installed for all the projects. It is used for building test data when building test classes.

Integration for Apex Rest/SOAP callout

(Must Use if apex callout integration)
This framework is an accelerator when building Apex REST or SOAP Callouts. It is providing mostly REST callout capabilities but can be also useful for SOAP ones. It enforces the use of Named Credential and provide out of the box debugging capabilities.

Documentation Location	/Documentation/ IntegrationFramework.docx
Components Prefix	**INT**
Use case	Must be installed for all the projects performing REST or SOAP Apex callouts.

Extended Integration Framework

(Ad Hoc Use)
The integration framework is providing great features, but the following use cases will require some more advanced coding or are not covered out of the box by the basic integration framework:

- Trigger the integration asynchronously.
- Setup a retry mechanism.
- Retrieves easily message logs related to a given interface in a central place.
- Decide easily what technology is used to trigger the integration:
  - Synchronous Callout
  - Async Callout
  - Platform Event
- Generate integration payload as config only.

In this case, the ***Extended Integration Framework*** might be what you are looking for.

Documentation Location	/Documentation/ ExtendedIntegrationFramework.docx
Components Prefix	**EIF**
Use case	In case you are looking for Integration capabilities doing config only and you would like to bring the Integration Framework at the next level, this framework extends the capabilities of the standard integration framework.

Integration Messaging (Outbound Message)

(Must Use if Outbound Message integration)
This framework is acting as an event bus allowing to easily publish events to external systems via Apex or Configuration. The events are sent as Outbound Messages. This nugget is a very good alternative when platform events cannot be used for technical reasons and offers a powerful way to implement a notification pattern without having to write any single line of code.

Documentation Location	/Documentation/ Integration_Messages.docx
Components Prefix	**EVT**
Use case	Good match when planning to use Outbound Message integration or when you need to integrate systems through an asynchronous notification pattern.

UI Libraries

(Must Use if LWC Development)
This framework provides a set of common front end Libraries to be used in the context of Lightning Component & Lightning Web Component. Those libraries are defining methods and capabilities commonly used cross SF projects implementations.
The libraries available are categorized in 2 areas:

- **LWC Common:** This library is providing a set a commonly used JavaScript methods that can be used in the context of building LWC.
- **Record Creation Redirection:** This library offers the possibility to create a record with pre-defined values by a simple URL redirection. It can be easily used in any type of URL buttons and is fully supported in SF Mobile.

Documentation Location	/Documentation/ UILibraries.docx
Components Prefix	UIL

Use case	This library is providing a set a UI accelerators and libraries.

Job Engine

(Ad Hoc Use)
Have you ever had to execute some code logic but you were wondering if you should execute this logic synchronously or asynchronously? And when it comes to asynchronous process, what SF technology do you need to use? @Future, Queueable Apex, … ?
The Job Engine framework will manage this for you and will take the right decision, depending on your context and how close you are from reaching governor limits. On the top, this framework provides an easy way to retry the execution of a given job that would have previously failed till it succeeds. This is typically useful in an integration context where you would need to keep retrying your apex callout for a certain number of times if this one is failing. Of course, the framework is generic and is allowing an automated retry for any job that would need some retry.

Documentation Location	/Documentation/ JobEngine.docx
Components Prefix	JOB
Use case	A standalone job must be executed and eventually retried in case of failure.

Inverted Relationships Framework

(Ad Hoc Use)
This framework is providing an easy way to represent relationships between objects of the same type without having to display 2 related lists. The principle consists in defining for each relationship type and inverted relationship type. Each time a relationship is created/updated/deleted the associated inverted relationship is created/updated/deleted automatically by this framework which is allowing to view easily the whole information in only one related list.

Documentation Location	/Documentation/ InvertedRelationshipFramework.docx
Components Prefix	**IRE**
Use case	You need to represent Account to Account , Contact to Contact or any relationship between the same object.

Tax/VAT Validator Engine

(Ad Hoc Use)
This framework provides a generic Tax/VAT validation engine with the following **capabilities**:

- Out of the box integration with **VIES** for European countries supported by VIES.
- Out of the box format validation for European countries supported by VIES.
- Easy extension for other countries allowing to integrate with your own external validation system or creating your own formatting rules.

Existing European setup can be adapted per country to leverage an external validation service of your choice (if VIES should not be used)

Documentation Location	/Documentation/ InvertedRelationshipFramework.docx
Components Prefix	**VAT**
Use case	You are looking for a component allowing to easily perform Tax/VAT validation. This framework is what you are looking for.

Transformation Engine

(Ad Hoc Use)
This framework offers ETL capabilities to generate dynamically an XML or JSON payload, extracting data from SF by only doing configuration and not having to write any single line of code. This framework has not the pretention to replace real ETL tools and should be used only in specific good use cases:
e.g. External Document generation engine integration:
SF needs to send the data to an external system in order to generate a document. The data structure to be provided is usually not complex and new templates can be added on regular basis. This framework allows to easily build an engine which will not need custom coding each time a new template is created or if a change is needed in one of the existing template.

Documentation Location	/Documentation/TransformationEngine.docx
Components Prefix	**TRS**
Use case	Good match when you need to dynamically generate JSON or XML simple structure and you don't want to it via Apex.

External Attachments Plugin

(Ad Hoc Use)
This plugin provides a fully configurable User Interface allowing to View, Upload, Download attachments which are not stored physically in SF but in an external content management system. The plugin offers a lightning ready, fully customizable User Interface but not the integration with the content management system as such. Integration hook can be easily built by implementing an Interface defined by the plugin.

Documentation Location	/Documentation/AttachmentsPlugin.docx
Components Prefix	**ATT**
Use case	Requirements to store attachments and files related to SF records outside of SF. This plugin provides a fully customizable UI, which can be easily hooked to a custom integration layer of your choice.