



Eversana

Salesforce Development Standards and Guidelines



Generated on Mar 02, 2021

CONTENTS

1. Getting Started
2. Guiding Principles
3. Naming Standards
4. Application Configuration
 - 4.1 User Interface
 - 4.2 Business Logic
 - 4.3 Data Model
 - 4.4 Process Builders
 - 4.5 Record Types
 - 4.6 Page Layouts
 - 4.7 Multi-Currency Considerations
5. Coding
 - 5.1 Styling
 - 5.2 Commenting
 - 5.3 Apex Class
 - 5.4 Apex Trigger
 - 5.5 Test Class
 - 5.6 Apex Controller
 - 5.7 Visualforce
 - 5.8 Custom Setting & Metadata
 - 5.9 Async Processing
 - 5.9.1 Batch
 - 5.9.2 Queueable
 - 5.9.3 Future
 - 5.9.4 Schedulable
 - 5.10 Error Handling
 - 5.10.1 For Apex
 - 5.10.2 For Lightning
 - 5.10.3 Error Framework
 - 5.11 Lightning Component (Aura)
 - 5.11.1 Lightning Best Practices
 - 5.11.2 Lightning Events
 - 5.11.3 Lightning Variable Binding
 - 5.11.4 JavaScript
 - 5.11.5 CSS
 - 5.12 Lightning Web Components
 - 5.12.1 General Guidelines
 - 5.12.2 Work with Salesforce Data
 - 5.12.3 Aura/VF interoperability
 - 5.12.4 Event Standards
 - 5.12.5 JEST

- 5.12.6 Security with LWC
- 5.12.7 LWC Performance considerations
- 5.12.8 LWC Limitations/Know Issues
- 5.13 Integration
 - 5.13.1 Integration Patterns
 - 5.13.2 Authentication
 - 5.13.3 Web Services REST
 - 5.13.4 Web Services SOAP
 - 5.13.5 Integration Testing
 - 5.13.6 Platform Event/Streaming API
 - 5.13.7 Outbound Message
 - 5.13.8 Continuation
 - 5.13.9 Change Data Capture
 - 5.13.10 Integration APIs
- 6. Security Standards
- 7. Accessibility
- 8. Data Management

Deloitte.

1. GETTING STARTED

Introduction

This site is a developer's guide to designing, configuring, customizing and maintaining Salesforce Applications based on Deloitte's experience in implementing Salesforce projects.

Intended Audience

The intended audience includes anyone doing technical work with Salesforce Application such as architects, designers and developers of the Salesforce application. This would be applicable for users making configuration changes using the 'App Setup' menu as well as developers implementing new functionalities through coding.

It is assumed that the reader has technical Salesforce experience or basic Salesforce training. The information in this site is not intended as training material for Salesforce development, but as a guide to good development practices. So it is encouraged to actively use Apex Developer network, your colleagues and other avenues for gaining knowledge on technical details surrounding the standards. E.g. A standard may state that in a certain situation, a Visualforce page and a controller should be used to pre-populate an edit page, but it will not include the exact steps of how to do so as this is considered Salesforce knowledge.

Scope

This document is comprised of best practice guidelines based on Deloitte's Salesforce experience. This covers common rules based on our experience. It might not be perfect for every implementation scenario as the requirements vary based on client needs; any configuration or customization should follow required review process with peers, architects and Salesforce experts to ensure quality implementation.

2. GUIDING PRINCIPLES

Development and Design Guiding Principles

Our guiding principle for technical design and development is to implement the required functional requirements as accurately as possible, while still ensuring:

- Org Maintainability
- Upgradeability
- Performance
- Integrity and stability

An additional principle is to implement adequate error prevention and handling mechanisms, even if they are not part of the functional requirements.

Org Maintainability

If the code and configuration are difficult to understand, debug, modify or upgrade, the Salesforce org is not considered to be maintainable. The result will be more defects, slow issue resolution, inflexible staffing (if developers only understand their own implementation) and performance issues. Many of the standards detailed out in the other sections ensure repository maintainability. Some generic topics however are -

- **Commenting Standards:** Comments should be made in the script as well as the code instance to explain the intention of the modification. Comments should also provide an audit trail or history of changes made. The code comments should be done as part of the development phase before the code is migrated to the next set of environments. Please refer to Commenting Section in Coding for more details.
- **Configuration rather than Customization:** In general, configured functionality is more maintainable than customized functionality (e.g. scripting). Later chapters will detail alternatives and describe when customization should be chosen over configuration. In addition, later sections will also describe when customization should not be undertaken.
- **Intuitive Structure:** Adhere to naming conventions and structuring the configuration and script in a way that makes it easy to understand. This is relevant both for configuration and scripting.
- **Avoid Dead Code/Objects:** If script or configuration is no longer required, a backup should be taken of that particular component/object before removing/changing the component in the org. Since we will be using version control tool to maintain the repository content, we have to ensure that the component that is being removed should be put in the version control system.

Upgradeability

In general, the less configuration and customization made to the solution, the more upgradeable it will be. However, it is not realistic to be able to implement the solution out-of-the-box; some configuration and customization will always be required.

The following standards should be followed to ensure upgradeability:

- Develop a maintainable org : This will make any custom upgrade activities feasible. Please refer to the above Org Maintainability section for more details.
- Use declarative programming instead of script:
 - Use Process Builder and Workflows instead of lengthy scripted processes.
 - Use configurations such as calculated fields, validation fields, defaults, picklists, dependent picklists etc. instead of script.
- Customization should be “supplemental”:
 - Leverage as much out-of-the-box as possible and package up the required customization outside of the existing functionality instead of making substantial changes directly to the out-of-the-box objects. This allows a clean upgrade of the out-of-the-box functionality, while not disturbing the custom development.
 - Although Salesforce fully supports the API and ensures that it is backwards compatible for several releases, it may change.
- Knowing when to create or modify objects:
 - Create when there is no out-of-the-box equivalent that is a good match
 - Modify when you only need slight modifications to the out-of-the-box object. Note: The upgrade scripts may override modifications you have made to the object.

Performance

Although certain performance issues are not apparent until the org is completely built and performance tested with real data volumes (under load on a production-comparable environment), there are certain proactive steps that can be taken to ensure a solution which is scalable and has a good performance.

There are three major areas of “performance” a developer should keep in mind:

- **Data Model Performance:** Searching, Inserting or Updating of data will result in DML operations. Developers need to minimize the quantity of these by bulkifying code to avoid unnecessary DML operations and ensuring that the operations are optimized. Additional DML considerations around performance are detailed below under API Performance.
- **UI Performance:** All logic performed consumes CPU resources wherever it runs, regardless of whether it runs on Salesforce’s cloud platform (Apex code, Visualforce) or locally in the browser (e.g., javascript validations, Visualforce/HTML constructs, UI implementations). Developers need to avoid unnecessary processing logic and memory consumption, especially when scripting using javascript.
- **API Performance:** Salesforce API calls can be expensive in terms of responsiveness and can result in delays that impact the user experience. In addition to the potential delay of running through the API, there are limits governing the number of API calls that can be made by a given user over a rolling 24-hour period. Follow the best practices defined in the sections when implementing APIs and Visualforce Remoting.

Integrity

Developing with too many assumptions of what the data looks like and what the user is doing leaves the implementation open to issues, including broken functionality, performance issues, data corruption and a non-user-friendly experience. Check all your assumptions before acting on them (“defensive coding”).

The best practice is to check that you have the data which is needed (at field or row level) and also that the infrastructure is running as expected (e.g. point-to-point integrations are up, etc.). If the assumptions hold true then continue with the rest of the process. If it doesn't, make changes to handle the exceptions (error handling).

Many standards described in later sections are related to defensive coding and ensuring solution integrity.

Optimized User Experience

Implementing an optimized user experience will help prevent errors and reduce user support requests. The following are several important aspects to consider when developing the technical design (and implementation) to provide the best user experience and prevent errors:

- Help the user to do the right thing:
 - Implement intuitive UI designs
- Ensure that the user can only do the right thing. For example:
 - Only show valid values in picklists (state model, dependent picklists and record types with relevant picklist options).
 - Only allow relevant fields for a given user to be editable.
- Check that the user actually did the right thing by using:
 - Field level validation.
 - Apex triggers: Where standard field validation is not sufficient, validations can be implemented using Apex triggers. Note that such triggers should be implemented only as a last resort and in accord with the standards and best practices.
 - Approval process entry and rejection criteria.

3. NAMING STANDARDS

Naming Standards

- The following guidelines must be followed for the names:
 - Use simple and descriptive names.
 - Use whole words and avoid acronyms and abbreviations unless the maximum file name length is reached. An exception to this rule is object field names.
 - Use acronyms and abbreviations for object field names.
 - Use standardized acronyms and abbreviations.
 - Abbreviations and acronyms should not be uppercase when used as name. Example:
 - Correct usage : `exportHtmlSource()`;
 - Incorrect usage : `exportHTMLSource()`;
- Add your application namespace before the metadata name e.g. ARB
- These sections outline the general naming conventions for each type.

Configuration

Custom Object

- Custom objects have an associated name field that is defined by a Salesforce administrator during setup.
- Custom objects must have unique names within the organization. In the API, the names of custom objects are identified by a suffix of two underscores immediately followed by a lowercase “c” character.
- Add your Project namespace before the name for identification. e.g. PHX_CustomObject__c
- API names for fields/objects should be descriptive and in short as much as possible. Having the lengthy names for fields and objects might get truncated in other data integration systems. Salesforce limits the API names for fields/objects to 40 characters

Custom Fields

- Salesforce populates Field Name using the field label. This name can contain only underscores and alphanumeric characters, and must be unique in the organization. It must begin with a letter, not include spaces, not end with an underscore, and not contain two consecutive underscores.
- Use the field name for merge fields in custom links, custom s-controls, and when referencing the field from the API
- It's a good practice to maintain label and API names to be consistent as much as possible. Ex: Billing City as label and Billing_City__c as API name.
- Ensure the custom field name and label are unique for that object. Labels are determined as per business needs. e.g. Billing_City_c

Custom Labels

- Use uppercase letters for Custom Label Name.
- Separate words within the name using an underscore.
- Reflect the intent/characteristic of the label in Name.
- Avoid duplicate labels with the same value.
- Use Custom Labels instead of hardcoding any values in Apex, Process builders, formula fields and validation rules.
- In Visualforce and Lightning components, use the \$Label global variable.

Workflow Rule

- Workflow rules should be a phrase giving a high level picture of the work performed e.g. Request for Account

Workflow Action

- Workflow Actions should be verbs starting with lowercase letters, similar to methods Internal words start with capital letters
- For Update actions: setAccountStatusToOpen
- For Email Alert actions: notifyAccountOwner
- For Task actions: alertCaseOwner
- For Outbound Messages: obMsgSendStatusToARB

Validation Rule

- Validation rules should be phrases at least including the primary field on which the criteria are based on the internal name of the validation rule, with white spaces and special characters escaped for validity.
- The name can only contain characters, letters, and the underscore (_) character.
- The name must start with a letter, and cannot end with an underscore or contain two consecutive underscore characters.
- e.g. Opportunity Owner change after closure

Process builder

- Use the below combination for naming process builders <ARB>_<Object>_<Process>_<Functionality>
 - e.g. PHX_Opportunity_ADM_Notification

Email Templates

- Use the below combination for naming email templates <ARB>_<Functionality>
 - e.g. PHX_SignupEmail
- Team specific/application specific email template folders are to be created for better managing the templates.

Profile

- Only Use PascalCase for the label. Do not use any special characters. Use a common prefix for related functionality and Feature.

Permission Set

- Since the permission set are based on the specific Personas, the permission set naming convention should capture who is the user and what they intends to accomplish functionality.

Sharing Rule

- Name should specify which Object is shared with what access to whom [Object][Access][whom]
 - For e.g LeadReadAccessAccountExecutive

Description field should be populated for all of the above configuration objects. Putting proper description helps understanding their purpose and use.

Apex Class & Method

Apex Class Naming

- Must be nouns.
- Mixed case with the first letter of each internal word capitalized.
- The Name of the Class should always start with a keyword which denotes the project name.
- The following naming conventions are recommended based on the Class purpose:
 - Controller Class: PHX_<VF Page Name>Controller
 - Controller Extension Class: PHX_<VF Page Name>Extension
 - Batch Class: PHX_<Functionality>Batch
 - Scheduleable Class: PHX_<Functionality>Scheduleable
 - Queueable Class: PHX_<Functionality>Queueable
 - Utility Class: PHX_<Functionality>Utility
 - Trigger: PHX_<subject>Trigger
 - Interface Service Class : PHX_INT_<Functionality>Service
 - Interface Utility Class: PHX_INT_<Functionality>Utility
 - Test Class: PHX_<Class Name>Test
 - Exception Class: <Functionality>Exception
 - Other: PHX_<Functionality>

Method Naming

- Methods should use camel-case, similar to variables.
- They should express the use case the method is trying to solve.
- The method name should also try and include the return type variable. If void, this can be ignored.
- For example - A method to get all Accounts based on a String passed as a parameter can be named as: getAccountsByName

Apex Variables

- Variables are in mixed case with a lowercase first letter.
- The name of the variable should start with a Hungarian notation type prefix.
- Variable names should not start with underscore _ or dollar sign \$ characters, even though both are allowed.
- Variable names should be short yet meaningful. The choice of a variable name should be mnemonic- that is, designed to indicate to the casual observer the intent of its use.
- The naming convention for variables should follow camel-case.
 - Blob: binImage
 - Boolean: bStatus
 - Date: dtStartDate
 - DateTime: dtmStart
 - Decimal: dValue
 - Double: dblWidth
 - ID: accountId
 - Integer: iLoop
 - Long: lngTotal
 - String: sAccountName
 - Time: tmStartTime

Apex Data Structures

- Apex data structures i.e. sObject, List, Set, Map, Enums should start with 3 characters type of data structure i.e. lst for List, map for Map, set for Set, obj for any class, Objects or Interface.
- The name should be short yet meaningful. The choice of a variable name should be mnemonic - that is, designed to indicate to the casual observer the intent of its use.
 - List<Contact>: lstContacts
 - Map<Id,Contact>: mapIdContact
 - Set<String>: setAccountName;
 - enum enuSeason {WINTER, SPRING, SUMMER, FALL}
 - Account objAccount = new Account();
- Naming conventions for Apex data structures can also include verbs to convey the intention of using the them.
 - List<Contact>: List: lstContactsToUpdate

Naming of Constants

- 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:

- static final string SYSTEM_ADMIN = 'System Administrator'

Visual Force & Lightning Components

The rules for pages and components would be similar to Apex Classes.

- e.g.- PHX_<Functionality>

Deloitte.

4. APPLICATION CONFIGURATION

Configuration Standards

Configuration of Salesforce involves making changes to the application declaratively via the point-and-click Setup UI without the need for custom coding or scripting. This native platform functionality allows an administrator to create and administer the following components from within the Web browser application:

- Data components (standard and custom objects, fields, relationships, etc.)
- Business logic components (security and permissions, formula fields and validation rules, workflow rules, approval processes, etc.)
- UI Components (tabs, page layouts, views, reports and dashboards, etc.)

Salesforce is primarily intended for configuration and it is recommended that majority of the changes made to the application are implemented declaratively. The Salesforce infrastructure is designed to allow configuration changes to occur without any performance implications. Configuration changes are fully compatible with future upgrades. However, customizations in managed packages or in the native org might break depending on new limits violations or depreciated functions.

Commenting and User Help Standards

- Component Description Format

Provide comment on the intent of any object modification and explain what and why the modification is actually being made. Keep a history of comments by adding the latest comment above the older comments (even if they no longer are true). Pre-release, however, a description of the object is sufficient. Comment configured and created objects (first letter of first name + full last name, date and comments), and keep a history of comments. Example:

Custom Object Definition Edit
Save Save & New Cancel

Custom Object Information

The singular and plural labels are used in tabs, page layouts, and reports.
Be careful when changing the name or label as it may affect existing integrations and merge templates.

Label Example: Account
Plural Label Example: Accounts
Starts with vowel sound ☐

The Object Name is used when referencing the object via the API.

Object Name Example: Account
Description

mmurali 15/01/2019 : Replacing the Master Detail Relationship with Lookup as the Events object can be related to Contacts and Leads as well.
mmurali 09/09/2018 : Added new object to capture events tied to account. Master Detail of Account as its visibility is controlled by parent Account

Field-Level Help Text

Field-level help allows you to provide help text detailing the purpose and function of any standard or custom field. You can define custom help text for your organization's fields to provide users with a helpful description for any field on all detail and edit pages where that field displays. Users can view the field-level help text by hovering over the Info icon next to the field.

Help Text is required in Salesforce for all fields.

- The following guidelines should be followed for Help text in Salesforce:
 - Should be provided by the Business Process Owner and should provide clear definition to end users for the purpose of the field
 - Should be concise but should not contain acronyms.
 - Should not apply to a single region's business process if the field is visible/shared globally.
 - The character limit for Help Text is 255 characters
- Implementation Tips
 - Field-level help is not available for some standard fields, including fields on the User object, system read only fields, auto-number fields, multi-currency fields, Ideas fields, and Community fields.
 - The help text for a field is automatically added to a package when you add the associated field to any Force.com AppExchange package.
 - In a managed package, the help text is locked to the developer, giving installers full capabilities to change it.
- Best Practices
 - Because custom help text displays on both edit and detail pages, avoid instructions for entering data. Instead, construct help text that defines the field's purpose, such as: "The maximum discount allowed for this account."

- Provide information in your help text about the attributes of the field, such as a detailed description of the purpose for the expense report. Up to 32 KB of data are allowed. Only the first 255 characters display in reports.
- Provide examples in your help text that help users understand the field's meaning clearly, such as: "The four-digit promotional code used to determine the amount charged to the customer, for example, 4PLT (for level-four platinum pricing)."
- If your organization uses more than one language, provide translations for your Help Text using the Translation Workbench.

Salesforce Governor and Usage Limits

The Salesforce platform leverages a multi-tenant architecture to deliver services to their clients. As a result, custom development must consider the limits that are enforced. As with most things, certain limits are absolute, while others can be adjusted by Salesforce support based on documented use cases. Additional fees and contractual agreements may be required when limits are raised.

To view the current platform limits, refer to the Salesforce Limits Quick Reference Guide, which is updated with each release.

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

4.1 User Interface

Best Practices

- Unless requirements state otherwise, order your tabs by process sequence.
 - For large screens consider alphabetizing the sequence. It is hard to navigate large screens if the view tab sequence is not logical.
- Clone or create a new default page layout.
 - Each new Salesforce org comes with a default page layout for each object. Create a new page layout or clone this default page layout or modify the existing page layout as per business requirements.
- Use page layouts control visibility, not security.
 - Page layouts can be assigned for each record type and profile. It is important to note that removing a field from a page layout only restricts whether the user can see the field on the screen; it does not prevent the user from reporting on the data or potentially exporting the data.
- Implement a system admin page layout for each object to contain all fields.
 - This will allow certain profiles (like System Administrators) to select the system admin view to see all fields. This minimizes page layout changes and support requests.
- Group fields into logical sections.
 - Using Sections on page layouts, related information (fields) can be grouped together in logical sections. In addition to allowing the user to find related information close together, it allows a user to expand and collapse sections to enhance their UI experience. In the following example, all contact information is grouped together:

▼ Address Information

Mailing Address

2335 N. Michigan Avenue, Suite 1500
Chicago, IL 60601
USA

Other Address

2335, Jawaharlal Nehru Road
Hyderabad, Telangana 500001
India

▼ Contact Information

Email

barr_tim@grandhotels.com

Phone

(312) 596-1000

Mobile

(312) 596-1230

Other Phone

(312) 596-1500

Home Phone

(312) 596-1200

Asst. Phone

(312) 596-1200

- Page layouts should remain as consistent as possible across layouts/regions. Button sequence, field sequence and layout, etc., should be consistent unless strategically decided otherwise at a field or section level for individual business groups.
- Place dependent fields close to the Controlling field.
 - When using dependent fields, place the controlling field and the dependent field as close to one another as possible. If you decide to place the depending field next to the controlling field (i.e. you

are using a two columned section on your page layout), ensure that all subsequent depending fields are placed to the right as well for consistency.

- Place controlling fields exist on any page layout that contains their associated dependent picklist.
 - When the controlling field is not on the same page layout, the dependent picklist shows no available values. Note: If a dependent picklist is required and no values are available for it based on the controlling field value, users can save the record without entering a value. The record is saved with no value for that field.
- Ensure standard button consistency.
 - Using the same set of buttons on each screen in the same location allows the user to build up a level of familiarity while navigating the application. Try to achieve a level of consistency with the buttons displayed across objects and page layouts.
- Ensure related list field consistency.
 - It is important for users to have a consistent look and feel. When representing an object (e.g. account, quote, contact etc.) via a related list, make it look and feel the same no matter where it is referenced in the application. If possible, reuse the same display fields and order across the application.
- Only activate required functionality.
 - Out-of-the-box Objects include a wide range of functionality and fields that can be accessed through buttons, shortcuts, and via regular page layouts. Some of these functions are very powerful and should be inactivated unless explicitly required to avoid unexpected operations that could have performance, data or functionality impacts. Following are some considerations that should be accounted for:
 - Remove fields and buttons from page layouts if not required
 - Limit user functionality via the user's profile
- Use pick lists over text fields whenever possible.
 - Picklists are a useful field type to obtain greater data integrity. They are also favorable when reporting off data. Unless otherwise stated in the requirements, order pick lists in alphabetical order, or with the most used options at the top.
- Create generic custom views for users and allow them to build their own.
 - Custom views allow users to filter the records they see for a particular object based on criteria. The administrator should define a small number of generic views for all users (business can typically provide a sense of what these should be). Users should then have the ability to create their own views from scratch or modify existing ones.
- Error Message
 - Provided by the Business Process owner
 - Should be concise and tell the end user exactly what needs to be done to correct the error
- Apps
 - We have one main Salesforce App that controls tab visibility for all users. Any additional Apps require Architecture Review Board approval
- Custom Fields
 - Field Label/Name used for the custom field should be informative, simple, and distinct.
 - Help text is defined when creating a field and can be added or edited through the setup menu after a field is created. It is recommended to provide Help text for all the fields.

- Search Layouts
 - Update search layouts so that they provide enough information for the user. Search layouts should be consistent with the fields shown in list views.
- Mini Page Layouts
 - You can also customize mini page layouts so that they show more relevant information. Click on "Edit layout" then click on "Mini page layout", add the relevant fields, and then click "save".
 - Use the Salesforce Mini-Page layout for the Console, Hover Details, and Event Overlays.
- Thumb rule for tabs.
 - Tabs are only needed if a user needs direct access to the object or needs to be able to search and display the results of the object

Naming Conventions

Custom Objects

Custom object names should be unique, beginning with an uppercase letter. Whole words should be used and use of acronyms and abbreviations should be limited. Application-specific objects should begin with the application code. The object name should be singular (e.g. Review vs. Reviews), and NOT duplicate the name of an existing object. With Exception that widely used acronyms and abbreviations can be used instead of the long form. For example, HTTP or URL.

Example:

- Incorrect Usage: Terr Def

Here abbreviations have made this object name hard to understand.

- Correct Usage: Notification Type Application specific object with clear name

Custom Field

Custom field names should be unique, beginning with an uppercase letter. Whole words should be used and use of acronyms and abbreviations should be limited.

Field Labels should:

- Provided by the Business process owner, approved by the Product Owner
- Clearly state the intent of the field.
- Be concise and not contain filler words like "and" or "with"
- Acronyms should be avoided.
- Consider global visibility/consistency when deciding on field label
- The character limit for Field Labels is 40 characters. Lengthy labels that exceed Salesforce maximum character limit, must use predefined shortened/abbreviated words.
- Always be analyzed to see if an existing similar field exists.
- Field labels must be mixed case with the first letter of each internal word capitalized.

- Example: First Name
- Standard abbreviation must be in all caps,
 - Examples: Provider ID, SSN
- Labels should not be numbered.
 - Avoid following label names:
 - Q1. Is the child in danger?
 - Q2. Does the child know the abuser?
 - Instead, use as follows:
 - Is the child in danger?
 - Does the child know the abuser?
- Do not use long labels. Use short and clear labels.
- Labels must be right aligned to the field.
- Use "Help Text" for the fields to provide additional hints to the users.

Any field which is created ONLY for a back-end logic should be identified differently in Label or API name.

- For instance, in this example, we append 'Apex' in a back-end field:
 - "Store_Ext_Id_Apex__c"
- Each of such field should have Description as to why is this created and where it is to be used so that any developer/admin when comes across this field knows the relevance of it.

For Picklist Fields:

- Values must follow field label standards i.e., mixed case with the first letter of each internal word capitalized.
- Consider global visibility/consistency, avoid slight variations for the same value
- Picklists with Yes and No options should be only used if the user has to mandatorily select a value. If not, instead of Yes/No picklist, use Checkbox.
- If there is need to have reporting based on a field value, picklist with values as 'Yes/No' is more recommended,
- When updating a picklist value, consider the difference between adding new and updating/renaming existing values.
- The business value for adding multi-select picklist (vs single select picklist) field needs to be evaluated thoroughly.
- By default, picklist values must be sorted alphabetically. If the business requirement calls out a specific order, then it must be followed.
- Values such as, "NA", "Other" must be displayed as last option.
- If same set of values are used in more than one field, then use Global Picklist Value Sets. Following are the possible candidates for Global Picklist:
 - State
 - County
 - Country
 - Salutation
 - Suffix

- Frequency
- If both State & County picklist are used in the page, then County must be dependent on State.

Formula Fields

- Validation Rules & Formula fields should adhere to the use of AND and OR instead of || and &&
- Avoid formulas that return a text string that needs to be translated (a “true” or “false” is acceptable if it is used in a decision process)
- Avoid using LEN, LEFT and RIGHT in multi-lingual implementations as it will return different results.
- Formula field with a Currency return type will be displayed the result in the currency of the record.

When creating custom objects, fields and other components, the description field should always be populated. Though it's an optional field but putting proper description with requirement# or story number helps understand and track them in future. Also it is very convenient for new developers to know the purposes of components.

For details on component description format Refer Commenting and User Help Standards under Application Configuration.

Error Messages

An error message is text that is displayed to describe a problem that has occurred that is preventing the user or the system from completing a task. The problem could result in data corruption or loss. Other message types include confirmations, warnings, and notifications.

Poorly written error messages can be a source of frustration for users. Effective error messages inform users that a problem occurred, explain why it happened, and provide a solution so users can fix the problem.

Focus on the problem, not the user action that led to the problem, using the passive voice as necessary.

Incorrect Message: You have entered an invalid End date that is prior to Start Date.

Correct message: End date cannot be prior to Start Date

Good error messages have:

- A problem. States that a problem occurred.
- A cause. Explains why the problem occurred.
- A solution. Provides a solution so that users can fix the problem.

Additionally, good error messages are presented in a way that is:

- Relevant - The message presents a problem that users care about.
- Actionable - Users should either perform an action or change their behavior as the result of the message.
- User-centered - The message describes the problem in terms of target user actions or goals, not in terms of what the code is unhappy with.

- Brief - The message is as short as possible, but no shorter.
- Clear - The message uses plain language so that the target users can easily understand problem and solution.
- Specific - The message describes the problem using specific language, giving specific names, locations, and values of the objects involved.
- Courteous - Users shouldn't be blamed or made to feel stupid.
- Rare - Displayed infrequently. Frequently displayed error messages are a sign of bad design.

Following standards must be used for the error messages:

- All the error messages must be reviewed and approved by SMEs.
- Error messages should not end with full stop “.”.
- If the message is specific to the field, then it must be displayed below the field. The erred field must be highlighted with red color border using Salesforce standard styles. All other messages should be displayed at the top of the page.
- If the message is displayed at the top and refers fields in the page, then the fields must match with field labels.
- Do not display system error messages or code failure messages. Instead, display non-technical and easy to understand message for the end users.
- Avoid the word "please," except in situations in which the user is asked to do something inconvenient (such as waiting) or the system is to blame for the situation.
- Do not hard error messages in the Apex code. Create Custom Labels and refer them in the Apex for efficient maintenance.
- Do not create duplicate error messages for similar/identical problems. Instead, reuse Custom Labels. If needed, use label parameters to substitute field names dynamically based on the context.
 - Example: Custom Label: % is invalid
 - Actual messages after replacing the parameter:
 - End date is invalid
 - Start date is invalid
 - Message font color should be red and follow Salesforce standard styles.

Standard error messages:

Cause	Message
Field is empty	Enter a value for this field
Multiple values are left blank	Enter all the required values
Date is invalid	Invalid date

Number is invalid	Invalid number
Future date is not allowed for example Date of Birth.	<field> cannot be in the future
Date comparison – One date cannot be greater than the other, for example, End date cannot be prior to Start date.	<field1> should be later than <field2>

List Views

- Salesforce OOB List View displays records from an object in a tabular format. Create customized List View as per business requirements.
- List View name must be unique, clear and easy to understand.
- If multiple views are created and the view names need to be sorted then prefix the name with number, for example "01 Accounts with Pending Status", "02 Accounts Having No Address".
- Enter appropriate filters to narrow down the number of records.
- Keep the number of columns to the minimum number which allows the table to be displayed on one screen – without the need for horizontal scrolling.
- Make suggestions to Clients/End Users to leverage Out of the Box "Chart" and Kanban features. End Users are usually not aware of native salesforce features.
- Users can also use "Pin this list view" to make a list view default for them. These features are very handy for quick access of records. Suggestions like these will create Value add with Clients. If possible, educate the end user's via training documents or demos.
- Displaying relevant fields will keep users from lots of additional clicking since relevant information is already visible

<div> <div> Leads </div> <div> All Open Leads </div> <div> New Import Change Status </div> </div>								
5 items • Sorted by Name • Filtered by all leads - Lead Status • Updated a minute ago								
	NAME ↑	COMPANY	STATE/PROVINCE	PHONE	EMAIL	LEAD STATUS	OWNER ...	CREATED DATE
1	<input type="checkbox"/> Andy Smith	Universal Technologies	Connecticut	(555) 555-1212	info@salesforce.com	Open	SKONA	1/5/2016 6:48 AM
2	<input type="checkbox"/> Jim Steele	BigLife Inc.	Connecticut	(555) 555-1212	info@salesforce.com	Open	SKONA	1/5/2016 6:48 AM
3	<input type="checkbox"/> John Gardner	3C Systems	Massachusetts	(555) 555-1212	info@salesforce.com	Open	SKONA	1/5/2016 6:48 AM
4	<input type="checkbox"/> Sarah Loehr	MedLife, Inc.	New York	(555) 555-1212	info@salesforce.com	Contacted	SKONA	1/5/2016 6:48 AM
5	<input type="checkbox"/> Shah Rukh Khan	Deloitte				Open	rbuss	3/24/2018 3:45 PM

Page Layout – Detail Page

- The page layouts must be simple, familiar and logical to the user. The page layouts must share the same basic layout grids, themes, conventions, and hierarchies. The goal is to be consistent and predictable, so that users will feel comfortable exploring the application, and navigate confidently.

- Lengthy text inputs require 250 or more characters to be entered, use a wide, single column view for easy text entry and readability.
- Record status, when applicable, should be shown in the top right corner of the detail area.
- Keep the most used fields closer to the top to eliminate scrolling by your users.
- Use Sections to group the fields as per logical functions.
- System Information must be the last section and should contain only the audit information.

The page layouts should be in the following format:

Header Section

Contains Name of the page, action/navigation buttons

Current Object Section(s)

Contains one or more sections grouped logically as per the business needs.

Navigation Links Section

Contains parent/related entities values.

Footer Section

Contains the system/audit information

Related Lists

Contains one or more related lists, if any.

Page Layout – Related List

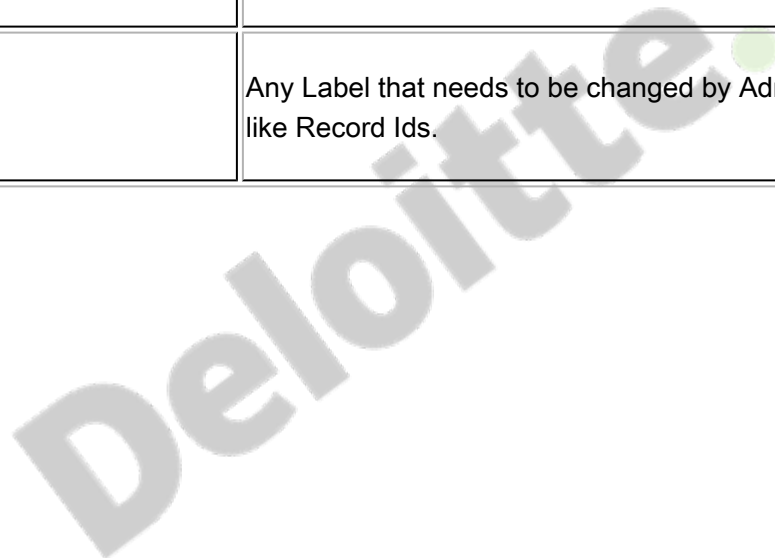
- The order of related lists must follow logical functional grouping.
- Notes & Attachments list must always be displayed as the last item.
- Keep the number of columns to the minimum number which allows the table to be displayed on one screen – without the need for horizontal scrolling.
- Display the action buttons as per the business requirements.
- The list must be sorted based on the business requirements. If the sorting order is not provided, then sort the record name in ascending order.
- Place custom related lists above the standard related lists in your Salesforce page layouts. Anything custom is likely to be used more often and should be easily accessible.

Custom Labels

- Custom Labels should be used in VF Page, Lightning Component for displaying any labels like Section titles, Buttons, Error Messages etc. Hardcoding should not be done.
- Use different categories to group the Custom Labels.

- Before creating any Custom Label, please check the existing labels to ensure that Labels are not duplicated.

Categories	Description
Message	Any message shown to User like Error or Warning or Information Message.
Button	Labels used for Button
Page	Labels used in VF page like Object Name, Section Header, Title etc.
Hardcoded	Any Label that needs to be changed by Admin during deployment like Record Ids.



4.2 Business Logic

Validation Rule Conventions and Standards

Validation rule names should be unique, beginning with an uppercase letter. Whole words should be used and the use of acronyms and abbreviations should be limited.

For example, if we need to include the region at the beginning, if applies to all, indicate Global.

<Region><Optional-Application> <Field> <Rule> <Optional Dependency>

E.g. CPM Post Code Required (Suburb)

The above example shows that Post Code is required depending on values of Suburb field. The exact rule definition is in the detail, but keeping this format helps keep the naming as intuitive as possible without being too restrictive.

Acronyms and abbreviations can be used in case when there is an upper limit for the number of characters in the name field the use of abbreviations will be permitted if by including them the name becomes easier to read.

Validation rules acts on database levels, so do verify that the fields exists on the page layout for specific record type.

Configurable flag(such as custom label or field) should be set to bypass a validation rule. Doing so is very convenient to deal with specific scenarios such as bulk data migration or to skip it for particular users/profiles

Some conventions which can be followed are -

1. Don't use hardcoded IDs.
2. Put a bypass system in the logic in order to make this logic skip for any users/profiles.
3. When adding a Validation Rule, migrate existing records that don't match this rule.
4. Define the error message at the field location instead of at the top of the page.
5. Remove inactive Validation Rules

Approval Processes & Steps Conventions and Standards

Approval process names will follow the convention:

Naming standard for overall Approval process needs to include the area it identifies e.g. <Region> first.

Process Name: <Event that fired the approval>

Note that this is not the action or actions being performed, but the original event that will trigger the entry criteria. This allows the approval actions to change over time. Including the Salesforce object in the rule name is unnecessary as there is a standard field in the list view that can show this and filter on it. Whole words should be used and the use of acronyms and abbreviations should be limited.

Description: A full description of the rule and what actions it performs.

Example:

Approval Process Name: Conflict Completed

Description: Brief description of the entry criteria to indicate a clear intention of when the process will be used.

Approval process step names will follow the convention:

Step Name: <Decision Outcome> - <User Friendly Description>

Description: A full description of the step including what it does and the intended actions that will be performed (e.g. who will it be assigned to)

Example:

Step Name: Auto Approved – Value within personal limit

Description: The approval step becomes self-documenting showing the administrator and user the result of the approval step.

Note - If you are working on Approval process that involves Currency fields and your org has multi-currency enabled, make sure you are validating the process thoroughly. Approval process consider the corporate currency and not the currency on the record which results in unexpected behavior. For e.g: Approval Steps based on Opportunity Amount ranges.

Workflow Rule Conventions and Standards

Workflow rule names will follow the convention:

Need to include the region, if applies to all use “Global”

Workflow Rule Name: <Region><Event that fired the rule>

Note that this is not the action or actions being performed but the original event that fired the workflow rule. This allows the workflow actions to change over time. Including the Salesforce object in the rule name is unnecessary as there is a standard field in the list view that can show this and filter on it. Whole words should be used and use of acronyms and abbreviations should be limited, expect for Application abbreviations.

Description: A full description of the rule and what actions it performs. Example:

Workflow Rule Name: Date of Birth Changed

Description: Sends an email to the Customer public group and updates the inactive flag of the contact for batch processing.

Replacing multiple workflows on the same object with object's trigger gives significant performance advantages. Though workflow is more configurable and doesn't require test coverage but for big enterprise application where performance is of utmost concern, replacing workflow with trigger is a better practice.

When certain functionality is built using more than three separate declarative components. For example consider a scenario where 'Case escalation' needs 5 workflow rules because each product and region has separate case-escalation logic and rules.

In this scenario prefer writing a trigger with a service class with different methods. Since all the logic is in one class, the functionality is easier to understand, test and debug.

Some conventions which can be followed while writing workflow are -

1. Don't use hardcoded ID's.
2. Put a bypass system in the logic in cases where the rules need to be skipped. i.e. Batch update through API users.
3. Don't create multiple workflows with the same rule.
4. Remove Workflow actions that are not used by workflow rules.
5. Remove rules that have no actions.
6. Remove inactive workflows.
7. Don't update fields if the current value is already correct.
8. Consider the logic of workflow and check that it should not get invoked redundant on any Apex Trigger call.

Field Update Conventions and Standards

Field update names will follow the convention:

Field Update Name: Set <Name of Field> - <Value>

Whole words should be used and the use of acronyms and abbreviations should be limited. By using this convention, the field update can be reused across workflow rules and approval processes and it is clear to the reviewer what action, field and value will be used.

Description: A full description of the field update including if other processes are dependent on the value.

Example:

Name: Set Customer Inactive Flag-False

Description: Updates the inactive flag on the customer record which will be used by batch apex for processing

Salesforce workflow field update action shouldn't check checkbox called "Re-evaluate Workflow Rules after Field Change" unless absolutely needed. If this checkbox is checked- it will rerun all other workflows and will slow down the system performance.

Email Alert Conventions and Standards

Email alert names will follow the convention:

Email Alert Description: Application - <Email Template Used>

Whole words should be used and the use of acronyms and abbreviations should be limited. By using this convention, the email alert can be reused across workflow rules and approval processes and it is clear to the reviewer who the email will be sent to.

Example:

Name: Email Alert Description

Description: NCR – New Service Program Notification Request

Roles Conventions and Standards

Since Client X- support many different applications on the force platform, a relatively flat role hierarchy will be implemented. Each business unit will have a 'root' Role – which will have no users assigned. Within the BU, role hierarchies may be used for sharing as appropriate. Ambiguous roles and the roles that inadvertently roll up to other roles should be avoided.

Profile Conventions and Standards

Salesforce Profiles should be unique and easily identify the access needs. The standard should be that the profile name starts with Region or "Global" and then describes the business function. The Description field should be required for all profiles.

- <"Global"/Region Name><Business Function>_User
- <"Global"/Region Name><Business Function>_User

These user profiles will grant the user basic login privileges to the system and access to some of the read only data that belongs to the specific business unit. The profile might provide a specific landing page or object layouts that are customized for the user of the specific business unit.

- <"Global"/Region Name><Business Function>_Admin

The admin profile will allow delegated administration if a user is a member of multiple business units, this standard breaks down. It is therefore recommended that permission sets be preferred over profiles, and more generic profiles be created for example:

- Standard_Upstream_User
- Standard_Upstream_Admin

Public Group Conventions and Standards

Public group names should be meaningful, unique, and complete. Use application specific references where reasonable. Avoid generic names. Group names can start with the group it identifies like Region.

Example: ECO – Leadership Team

Sharing Model Standards

Security – Develop a closed public sharing model with sharing rules as necessary

For larger implementations, it is better to develop a closed public sharing model to each object. For each role that needs to access and share data, sharing rules should be developed to accommodate accordingly. This helps ensure that only access that has been directly configured is granted, reducing the number of accidental access rights.

While sharing rules configuration for large volume of data, conditions like "ownership data skew" and "Parent-Child Data skew" should be explicitly be take care of.

Normally Salesforce does parallel sharing recalculation when a record is created or updated. But in case of large volume data loads, deferred sharing rule is recommended. Deferred sharing rule can be enabled by Salesforce support.

Principle of Least Privilege:

Each component of the application should only be granted access to the profiles and permission sets requiring access. Therefore all field level security should be turned off for all profiles and specifically granted back to the profiles/permission sets requiring the field visibility. This concept should be used throughout the application, including Apex and Visualforce access and other permissions.

Field-Level Security Standards

Use field-level security (FLS) to restrict access to fields

There are multiple ways to keep a user from seeing data at the UI level (removing from a page layout, using FLS, restricting access to an object), however the only way to ensure that the user does not have access to

the data in a specific field is to use FLS. This will ensure that the data is not reportable or exportable by a restricted user.

Translation Workbench Guidelines

- Overview of Translation Workbench

The Translation Workbench is used to specify languages you want to translate, assign translators to languages, and create translations for customizations done to the Salesforce organization, and override labels and translations from managed packages.

Please note that approval is needed from the architecture governance team before implementing translation workbench.

- Enabling Translation Workbench

Enable the translation workbench with the languages you require. Create Translations for objects and fields
Best Practices for the Translation Workbench

- To make the Translation Workbench most effective, following points should be adhered to:
 - Have some internal translators “on-call”
 - Let translators know which languages they are responsible for translating.
 - Notify all translators when new translated components are added to your organization.
 - Advise users when customizing reports or list views to use filter criteria values in their personal language.
 - However, if they use the Starts With or Contains operators, advise them to choose the language of The filter criteria values they entered.
 - Double check your translations, whether done in-house or professionally
 - Establish a process to change a translation – especially across same languages (Spanish, French, English, etc.)
 - Utilize local language fields for entry specific to regions Example: Local Company Name
 - If you expose multiple languages only in portals, review each and every language in the portal
 - Translation Workbench is not available for single-language organizations. If you aren’t sure whether you have a single-language or multi-language organization, contact Salesforce.
- What Can be Translated:
 - https://help.salesforce.com/articleView?id=translatable_customizations.htm&type=5
- What Cannot be Translated (see above link for updated list)
 - List Views
 - Home Page Components
 - Dashboard Components
 - Folder Names
 - Public Calendars

4.3 Data Model

Data Model Standards

Custom Objects Conventions and Standards

Custom object names should be unique, beginning with an uppercase letter. Whole words should be used and the use of acronyms and abbreviations should be limited. Application-specific objects should begin with the application code. The object name should be singular (e.g. Review vs. Reviews), and NOT duplicate the name of an existing object. With Exception that widely used acronyms and abbreviations can be used instead of the long form. For example, HTTP or URL.

Example:

- Incorrect Usage: Terr Def

Here abbreviations have made this object name hard to understand.

- Correct Usage: Notification Type Application specific object with clear name

Custom Field Conventions and Standards

Custom field names should be unique, beginning with an uppercase letter. Whole words should be used and the use of acronyms and abbreviations should be limited.

Example:

- Incorrect Usage: code

Here Ambiguous names will cause confusion

- Correct Usage: Country Code

A succinct description of the field using whole words.

Document purpose of the field in the Description field, this helps you and other's help understand why the field was created. Specially when you are looking for cleanup of obsolete fields.

Install "Field Trip" to analyze how many records are populated with data and chose which ones to remove. Field Trip is a free AppExchange app which generates reports on standard and custom field usage for insight into data quality.

Field Labels should:

- Provided by the Business process owner, approved by the Product Owner
- Clearly state the intent of the field.

- Be concise and not contain filler words like “and” or “with”
- Acronyms should be avoided.
- Consider global visibility/consistency when deciding on field label
- The character limit for Field Labels is 40 characters.
- Always be analyzed to see if an existing similar field exists.

For Picklist Fields:

- Provided by the Business process owner, approved by the Product Owner
- Acronyms should be avoided
- Consider global visibility/consistency, avoid slight variations for the same value
- When updating a picklist value, consider the difference between adding new and updating/renaming existing values.
- There are limits on picklist entries and total characters for each entry on standard picklist fields, custom picklist fields, and multi-select picklists. Our standard is to limit LOVs to 20 per picklist. If a very large number of LOVs are needed, Architecture Review Board approval is required or a custom object should be considered.
- Always be analyzed to see if an existing similar LOV exists and can be used.
- If there is need to have reporting based on a field value, picklist with values as ‘Yes/No’ is more recommended,
- The business value for adding multi-select picklist (vs single select picklist) field needs to be evaluated thoroughly.
- Multi-picklist fields are not very reporting friendly. So, before you suggest the multi-picklist fields check if you have requirement's for reporting.

Schema Extension Standards

If there is a standard object where additional fields need to be added, extend the standard objects instead of creating a custom object. This will simplify reporting as well as improve performance with large data volumes (Salesforce has native indexes on the standard objects that are not available by default on custom objects; they must be requested).

Object relationship types should be precisely defined between look-up and Master-detail. Both have their own pros and cons and shouldn't be replaced with each other.

Create custom objects with master-detail relationships to take custom object data offline. In order to bring data to the offline edition, the data must belong to a valid standard object (accounts, leads, contacts, opportunities, activities) or to a properly configured custom object. To bring this custom object offline, a master-detail relationship must exist to one of the valid standard objects.

4.4 Process Builders

Naming Convention

1. Naming a process builder can be categorized based on the trigger point or the invocation point.
 1. Record Change happens like Create or Update: PHX_<sObject>_<Action>
 2. A platform event occurs: PHX_ Process _<EventName>
 3. Invocation from another process: PHX_<sObject>_<InvokedProcessName>_SubProc
2. Name criteria node by summarizing the criteria only since Actions have their own name which is displayed on the canvas. Example: Is Account Active? Or Is Active?

Why Process Builder?

1. Process Builder is a newer tool for admins which is even more powerful. In addition to everything a workflow can do (except for sending outbound messages), you can:
 1. Create a record (not just Tasks!)
 2. Update related records
 3. Launch a Quick Action
 4. Post to Chatter
 5. Launch a Flow
 6. Call Apex code
 7. Submit for approval
 8. Invoke another process
 9. Send custom notifications
2. Use Process builder to update related records:
 1. For updating related records, Process Builder can update any field on any related record, where Workflow can only update some fields on a parent record of a Master-Detail relationship.
 2. Process Builder can also update multiple related records in a situation when all of a record's child records need the same update.
3. Using Process Builder, exact order of execution can be set.
4. Process Builder can be versioned i.e. it can retain deactivated Processes. This will be helpful in activating versions as and when needed or also looking back at what was implemented earlier.

Best Practices

General Best Practices:

1. Always use one record-change process per object. Spawn sub process from master process to modularize the processes.
2. Avoid generating infinite loops that will hit the organization limits.
3. Delete Workflow or Apex Triggers once equivalent Process Builder is created.
4. Processes that update owners don't also transfer associated items. To ensure transfer, use one "Update Records" action for each type of child record that you want to transfer.

5. Currency field updates are based on record currency. Make sure record currency is updated to the specific needs.
6. If there is change in custom field label, the process needs to be updated to use the new label since retains the old label of the field.
7. Immediate Update Records actions obey validation rules.
8. Avoid creating duplicate records while creating or updating of records.
9. If a formula field is being used during record creation or update, the process evaluate the formula value based on the field updates until the process is complete.
10. If a formula in the process use any of the following functions, the formula returns NULL.
 1. GETRECORDIDS
 2. IMAGE
 3. INCLUDE
 4. PARENTGROUPVAL
 5. PREVGROUPVAL
 6. REQUIRE SCRIPT
 7. VLOOKUP

Process Builder Actions Best Practices:

1. Combine actions when possible so that governor limits are not hit.
2. Build reusable actions like email alerts, quick actions, processes, flows, and Apex.
3. Avoid having or be careful when multiple action groups update the same field.
4. Don't start the message with a field reference, such as {[Account].Name}. Otherwise, the action fails to save. To work around this issue, add a space at the beginning of the message.
5. Make sure that immediate actions don't cancel scheduled actions.
6. To access external data after changing Salesforce data, use scheduled actions.
7. Design a process with a scheduled action so that it doesn't execute mixed DML operations. A single transaction can't mix DML operations on data objects (such as Account), Setup objects (such as User Role), and external objects. For example, you can't update an account and a user role in a single transaction.
8. To improve performance further and help avoid Apex governor limits, design scheduled actions to take advantage of bulkification.
9. Schedule based on user time like 3 days from now. The schedule is evaluated based on the time zone of the user who created the process.
10. Scheduled Update Records actions do not obey validation rules.

Event based Process Best Practices:

1. While associating a record, Use the process's matching conditions to find exactly one record.
2. When creating similar event message from the process make sure the new event being created doesn't meet the filter criteria of the process to avoid creating endless loop.
3. When event process is packaged, do add the associated object to the package.
4. Before you uninstall a package that includes a platform event, deactivate all processes that reference the platform event.

Governor Limits

1. Process Builder Limits
 1. Total characters in a process name is 255
 2. Total characters in a process's API name is 79
 3. Total versions of a process per object is 50
 4. Total criteria nodes in a process is 200
2. Shared Limits
 1. Active record change processes and rules per object is 50
 2. Criteria nodes that are evaluated and actions that are executed at runtime per process is 2000
 3. Groups of scheduled actions that are executed or flow interviews that are resumed per hour is 1000
 4. Schedules based on a field value or relative time alarms defined in flow versions is 20,000
 5. Total Processes can have 5per process type in Essentials or Professional edition and 4000 per process type in Enterprise, Unlimited, Performance or Developer edition.
 6. Active Processes can have 5per process type in Essentials or Professional edition and 2000 per process type in Enterprise, Unlimited, Performance or Developer edition.
3. Processes are governed by the per-transaction limits that are enforced by Apex. If the process causes the transaction to exceed governor limits, the system rolls back the entire transaction.
4. All formulas that are used in a criteria node must:
 1. Return true or false. If the formula returns true, the associated actions are executed.
 2. Not contain more than 3,000 characters.
 3. Not contain an unsupported function.
 4. Reference the process trigger object for that process.
 5. Use the correct capitalization when referring to the process trigger object.

References

https://help.salesforce.com/articleView?id=process_considerations.htm&type=5

<https://www.salesforceben.com/workflow-rules-vs-process-builder-feat-apex/>

https://help.salesforce.com/articleView?id=process_which_tool.htm&type=5

4.5 Record Types

When to/not to use Record Types

WHEN NOT TO USE SALESFORCE RECORD TYPES!

If users on a given profile only need to see one object page layout, then you do not need to add record types to the object. Simply assign each profile its respective page layout.

WHEN TO USE SALESFORCE RECORD TYPES!

Here are some situations that would call for custom Salesforce record types:

1. A user with a given profile may need to use >1 page layout per object. For example, opportunities may be one-time sales or service contracts, with different information captured for each type.
2. There are different sales processes for different kinds of opportunities, and some users may need to work with multiple types.
3. One or more picklist field needs to have segmented value sets.
4. Users clearly understand the different types and understand which record type to select when they create new records. The difference should be very clear to any users who might have to make a choice.

Best Practices

1. Before creating record types, include all of the possible record type values in your master list of picklists. The master picklist is a complete list of picklist values that can be used in any record type.
2. The master picklist is independent of all record types and business processes. If you add a picklist value to the master picklist, you must manually include the new value in the appropriate record types. If you remove a picklist value from the master, it is no longer available when creating new records, but records assigned to that value are unchanged.
3. When you create a new record type without cloning an existing one, the new record type automatically includes the master picklist values for both standard and custom picklists. You can then customize the picklist values for the record type.
4. Don't name your record type "Master". "Master" is a reserved name for record types.
5. If each profile is associated with a single record type, users will never be prompted to select a record type when creating new records.
6. A user can be associated with several record types. For example, a user who creates marketing campaigns for both U.S. and European divisions can have both U.S. and European campaign record types available when creating campaigns.
7. When creating and editing record types for accounts, opportunities, cases, contacts, or custom objects, check for criteria-based sharing rules that use existing record types as criteria. A record type change may affect the number of records that the rule shares. For example, let's say you have a record type

named “Service,” and you created a criteria-based sharing rule that shares all Service record types with your service team. If you create another record type named “Support” and you want these records shared with your service team, update the sharing rule to include Support record types in the criteria.

8. Renaming a record type doesn’t change the list of values included in it.
9. Deactivating a record type doesn’t remove it from any user profiles or permission sets.
10. Deactivating a record type means that no new records can be created with the record type. However, any records that were previously created with the record type are still associated with it and with its associated page layout.
11. To deactivate all the record types from an object, remove all the record types from all the profiles and then deactivate the record types. In order for existing records that used the deactivated record types to display properly, the object must have at least one record type, even if it’s not used by any profiles. Create a record type, don’t add it to any profiles, but activate it.
12. Deleting a record type also deletes the related path.
13. Business and person accounts require at least one active record type.
14. Deleting campaign member record types updates the Campaign Member Type field on campaign and campaign member records.
15. Person accounts are account records to which a special kind of record type has been assigned. These record types are called person account record types. Person account record types allow contact fields to be available on the account and allow the account to be used as if it were a contact. A default person account record type named “Person Account” is automatically created when person accounts are enabled for your org. You can change the name of this record type, and you can create additional person account record types.
16. From the UI, you can change an account’s record type from a business account to a business account or from a person account to a person account. However, to change an account’s record type from a business account to a person account, or vice versa, you must use the API.
17. When users convert, clone, or create records, these special considerations apply.
 1. When a user converts a lead, the new account, contact, and opportunity records use the default record type for the owner of the new records. The user can choose a different record type during conversion.
 1. When a user clones a record, the new record has the record type of the cloned record. If the user’s profile doesn’t have access to the record type of the cloned record, the new record adopts the user’s default record type.
 2. When a user creates a case or lead and applies assignment rules, the new record can keep the creator’s default record type or take the record type of the assignee, depending on the case and lead settings specified by the administrator.
18. Salesforce recommend creating no more than 200 record types. While there is no limit, orgs may have difficulty managing their record types if they exceed 200.
19. Record type IDs vary between production and sandbox orgs.
 1. Make use of lookups wherever they are available. Sadly, Process Builder does not support record type lookups directly.
 2. Be cautious about record type dependencies during deployments.
20. Each profile needs a default record type on objects having more than one type available.

Record Type Assignment

○ Assigning Record Type in Apex(Code)

Best way to get record type id of any object using Schema which does not count in Salesforce SOQL governors limit.

Use `getRecordTypeInfoByDeveloperName()` instead of `getRecordTypeInfoByName()` from `Schema.DescribeSObjectResult` class.

○ Code

```
public class GetRecordTypeId
{
    public GetRecordTypeId()
    {}

    // You need to pass two parameters SObject Type and RecordType Label not the Record Type Developer
    Name
    public static ID GetRecordTypeIdDynamically(String ObjectName, String RecordTypeLabel)
    {
        SObject ObjSObject;
        // Declaring Schema
        Schema.SObjectType objSchemaSObjectType = Schema.getGlobalDescribe().get(ObjectName);
        if (objSchemaSObjectType != null)
        {
            ObjSObject = objSchemaSObjectType.newSObject();
            // Declaring Object which Contains methods for describing sObjects.
            Schema.DescribeSObjectResult ObjDesSObjectRes = ObjSObject.getSObjectType().getDescribe();
            if (ObjDesSObjectRes != null)
            {
                Map mapRecordType = ObjDesSObjectRes.getRecordTypeInfoByDeveloperName();
                if (mapRecordType != null)
                {
                    Schema.RecordTypeInfo objRecordTypeRes = mapRecordType.get(RecordTypeLabel);
                    if (objRecordTypeRes != null)
                    {
                        return objRecordTypeRes.getRecordTypeId();
                    }
                }
            }
        }
        return null;
    }
}
```

Limitations

1. These special picklist fields aren't available for record types because they are used exclusively for sales processes, lead processes, support processes, and solution processes:
 1. Opportunity Stage
 2. Case Status
 3. Solution Status
 4. Lead Status
2. You can use these fields to provide different picklist values for different record types by assigning a different process to each record type.
3. These campaign member picklists aren't available for record types:
 1. Status
 2. Salutation
 3. Lead Source
4. You can't deactivate a record type if it is in use by an email routing address for Email-to-Case or On-Demand Email-to-Case.
5. You can't edit or delete a record type for an object if the object is referenced in Apex.

References

https://help.salesforce.com/articleView?id=customize_recordtype_considerations.htm&type=5

<http://suscosolutions.com/tips-using-salesforce-record-types/>

https://help.salesforce.com/articleView?id=recordtypes_picklists_limitations.htm&type=5

4.6 Page Layouts

Naming Convention

1. Page Layout for Record Pages: <ObjectName>_Layout
2. Page Layout for Record Pages based on Record Type: <ObjectName>_(<RecordType>)_Layout

Best Practices:

Page Layout Best Practices:

1. Always use page layouts to organize detail and edit pages within tabs.
2. Remove unnecessary fields from page layout. Keep the number of required fields added on page layout to a minimum.
3. Group similar fields with sections.
4. Large text areas should get a section all to themselves. Allowing 6 or more visible lines improves readability.
5. Keep Salesforce page layouts as consistent as possible across pages. This will help reduce the learning curve for the user.
6. Use field-level security to restrict users' access to fields. Field-level security settings override the visible and read-only settings on the page layout if the field-level security has a more restrictive setting than the page layout.
7. Assign page layout to combination of the user's profile and the record type, so as to guarantee on which fields are visible to user.
8. Use record types to provide unique layouts for different records. This will help minimize the number of layouts for an object.
9. Keep the number of required fields to a minimum. Setting a field to required means it must appear on the detail page of all page layouts, so consider whether each field is truly required.
10. To reduce the number of fields on a screen, consider using default values for new records instead of having the user enter the data.
11. You can't rename a page layout if you're using Salesforce Professional Edition.
12. Always check your layouts in Read and Edit modes.
13. Optimize related lists—adjust their overall order, the sorting of the records, and display of relevant columns and buttons.
14. If a dependent lookup is above its controlling field on a page layout, make its lookup filter optional or redesign the page layout. Placing a required dependent lookup above its controlling field on a page layout could confuse users who typically start from the top of the page when entering data.
15. A background process periodically runs that cleans up metadata associated with deleted custom fields. This process will affect the 'Last Modified Date' and 'Last Modified By' fields on page layouts, record types, and custom objects.
16. Salesforce recommends creating no more than 200 page layouts. Although there is no limit, it can be difficult to manage your page layouts if you have more than 200.
17. You can also customize mini page layouts so that they show more relevant information.

Page Layouts in Lightning Experience:

1. When you customize your page layouts in Salesforce Classic, those changes can affect the content of object record pages in Lightning Experience. However, in Lightning Experience, the page elements display differently, and some aren't supported.
2. These page layout elements aren't supported in Lightning Experience.
 1. Expanded lookups
 2. Mobile cards
 3. S-controls
 4. Section header visibility for Edit Page
 5. Tags

Page Layouts in Mobile App:

1. You don't need to keep fields in one column, as the page will render dynamically based on the device that's viewing it. A phone will reorder the fields into a single column, and a tablet or desktop will show two columns.
2. Put the most important fields into the compact layout which drives record highlights and record preview cards in the mobile app.

Home Tab Page Layout:

1. When editing the standard Messages & Alerts component, enter the text that you want to display to users. If entering HTML code for your message, make sure that it's self-contained, well-formed HTML. Standard Messages & Alerts home page components don't support JavaScript, CSS, iframes, and some other advanced markup.
2. When designing home page layouts for your Customer Portal, we recommend adding the following components: Search, Solution Search, Recent Items, Customer Portal Welcome, and a custom HTML Area component that includes your corporate branding in the wide column.

Compact Layout:

1. Don't make the primary field a lookup field. Doing this could result in navigation issues in Lightning Experience and the Salesforce app.
2. Removing a field from a page layout doesn't remove it from the object's compact layout. The two layout types are independent.
3. If you change a field on a compact layout to an unsupported type, the field is removed from the compact layout.
4. Fields that aren't available in the SOAP API don't show up on compact layouts in the Salesforce app.

Search Layout:

1. Update search layouts so that they provide enough information for the user. Search layouts should be consistent with the fields shown in list views.
2. If Search Layouts isn't available, the object isn't searchable, or you can't customize the search layout.

3. The search layout doesn't control which fields are searched for keyword matches. The list of fields searched is the same across Salesforce.
4. You can't remove unique identifying fields, such as Account Name or Opportunity Name, from the search layouts. These fields must be listed first in the order of fields in the search layout.
5. You can't add long text fields such as Description or Solution Details to search layouts.
6. For Professional, Enterprise, Unlimited, Performance, and Developer Edition organizations, search layouts don't override field-level security. If a field is included in the search layout but hidden for some users via field-level security, those users do not see that field in their search results.
7. For Personal, Contact Manager, Group, and Professional Edition organizations, search layouts override page layout settings. If a field is included in the search layout but hidden in the page layout, that field will be visible in search results.

Limitations

1. Page layouts in Lightning with more than 250 fields can also cause a QUERY_TOO_COMPLICATED error.
2. The mobile application treats rich text area fields like long-text area fields, which don't support formatted HTML content or images. The mobile application truncates rich text area fields at 1,000 characters, which includes HTML markup. Administrators should consider removing rich text area fields from mobile page layouts.

References

https://help.salesforce.com/articleView?id=layouts_tips.htm&type=5

https://help.salesforce.com/articleView?id=customize_layout.htm&type=5

<https://www.redargyle.com/blog/salesforce-page-layouts-best-practices/>

http://resources.docs.salesforce.com/204/17/en-us/sfdc/pdf/salesforce_app_limits_cheatsheet.pdf

4.7 Multi-Currency Considerations

Multi-currency Considerations

- Dated exchange rates are used for opportunities, opportunity products, opportunity product schedules, campaign opportunity fields, opportunity splits, and reports related to these objects and fields. Dated exchange rates are not used in forecasting, currency fields in other objects, or currency fields in other types of reports.
- Organizations with advanced currency management support roll-up summary fields between two advanced currency management objects. For example, roll-up summary fields are supported from an opportunity line object to its opportunity object, because both are advanced currency management enabled. However, if you enable advanced currency management, you can't create roll-up summary fields that calculate currency on the opportunity object rolling up to the account object, and you can't filter on the opportunity currency field on the account object. All existing currency-related roll-up summary fields on the opportunity object are disabled and their values are no longer calculated. If your organization enables advanced currency management, you should delete any currency roll-up summary fields using opportunities and accounts or opportunities and custom objects.
- Campaign opportunity fields use dated exchange rates when calculating the amount in the campaign currency, but are not used when converting those amounts to the user currency.
- Cross-object formulas always use the static conversion rate for currency conversion.
- If advanced currency management is enabled, you can't bind Visualforce pages that use `<apex:inputField>` or `<apex:outputField>` components to currency fields that support advanced currency management.
- Advanced Currency management feature enables weekly updates to currency rates based on a default currency value. More frequent Conversion rate updates can be handled through REST/SOAP API calls by integrating the rates from a source vendor like XE

5. CODING

Apex Code

Apex code is Salesforce's hosted scripting language. It is used primarily for transaction statements and has a Java-like syntax for completing stored procedural commands that work on the databases (queries, inserts, updates, and deletes). Apex code allows you to add business logic to applications in a more integrated manner than using the API. This is due to the fact the code is running on Salesforce's servers. While running the code on the servers provide performance and functionality boosts, it is also governed and limited by being a part of a multi-tenant environment. The three constructs that invoke Apex code are Classes, Triggers, and Anonymous Blocks.

- **Class:** A class is a template or blueprint from which Apex objects are created. Once successfully created and saved, class methods or variables can be invoked by other Apex scripts or via the Salesforce APIs.
- **Trigger:** A trigger is an Apex script that executes before or after specific data manipulation language (DML) events occur (e.g. before records are inserted into the database). It is best practice to develop trigger logic into a class, and then have the trigger invoke the class rather than embedding the business logic into the trigger code itself.
- **Anonymous Blocks:** An anonymous block is an Apex script that does not get stored in the metadata, but can be compiled and executed through the use of `executeAnonymous()` API call or the equivalent in the AJAX toolkit.

Base Guidelines

- **Coding Guidelines**
 - Code compiles with no errors or warnings.
 - All code is committed to the approved source control. Commits are atomic and frequent, are accompanied short description of the change, and are pushed to the remote repository at least daily.
 - Naming Conventions are adhered to.
 - Never store a username and/or password in code or source control.
 - Never hard coded Salesforce IDs or URLs in code.
 - All code must have 85% or above code coverage with proper Assert statements.
 - Always use the latest generally available version of the Salesforce API and if a current class is updated it should be saved in the current API version.
 - Follow the code commenting guidelines.
 - Code must provide proper Error and Execution Handling.
 - Security audits must be performed with appropriate tools.
 - All triggers should be bulkified.
 - Prefer Configuration over Customization.
 - Reduce cyclomatic complexity in code.

○ Cyclomatic Complexity

//Noncompliant Code Example:-

```
class Example {  
    void bar() {  
        if (baz)  
        {  
            buz.doSomething();  
        }  
        if (baz)  
        {  
            buz.doSomething();  
        }  
        while (baz)  
        {  
            buz.doSomething();  
        }  
        if (baz)  
        {  
            buz.doSomething();  
        }  
        if (baz)  
        {  
            buz.doSomething();  
        }  
        while (baz)  
        {  
            buz.doSomething();  
        }  
        if (baz)  
        {  
            buz.doSomething();  
        }  
        if (baz)  
        {  
            buz.doSomething();  
        }  
        while (baz)  
        {  
            buz.doSomething();  
        }  
        if (baz)  
        {  
            buz.doSomething();  
        }  
    }  
}
```

```
}
    if (baz)
    {
        buz.doSomething();
    }
    while (baz)
    {
        buz.doSomething();
    }
    for(String x: data)
    {
        buz.doSomething();
    }
}

void bar2() {
    if (baz)
    {
        buz.doSomething();
    }
    if (baz)
    {
        buz.doSomething();
    }
    while (baz)
    {
        buz.doSomething();
    }
}
}
```

```
//Compliant Solution:-
class Example {
    void bar() {
        for(String x: data)
        {
            buz.doSomething();
        }
    }
}
```

```
void bar2() {
    if (baz)
```

```
{  
  buz.doSomething();  
}  
}  
}
```

Deloitte.

5.1 Styling

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.
- Eclipse → Ctrl + I ,
- VSCode → Shift + Alt + F

Spaces

- A single space should be used to separate all operators (except increment / decrement), variables, literals, keywords, commas and semi-colons. Spaces should not be used after open parenthesis and before close parenthesis.
- The parenthetical clause in if, while, do, catch, etc., statements should be preceded and followed by a single space.
 - For e.g. if ((condition1 && condition2) || (condition3 && condition4) || !(condition5 && condition6))
- In method calls and definitions, there should not be whitespace between the name of the method and the open parenthesis.
- A single space should separate binary operators from the surrounding elements (e.g., +, ||, =, >=).
- A colon inside a for each loop (e.g., for (Contact con : contacts) {}) should have one space on either side.
- There should be no whitespace before commas, and one space after. For e.g.
`System.debug(LoggingLevel.INFO, 'test');`

Wrapping Lines

Avoid lines longer than 80 characters, since they are not handled well by many terminals and tools.

Note: Examples for use in documentation should have a shorter line length – generally no more than 70 characters.

When an expression will not fit on a single line, insert a line break 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.

SOQL Queries

- 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.
- 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

Wrapping Example

//Here is an example of breaking method calls:

```
someMethod(longExpression1, longExpression2, longExpression3,
           longExpression4, longExpression5);
```

//Following are two examples of breaking an arithmetic expression. The first is preferred, since the break occurs outside the parenthesized expression, which is at a higher level.

```
longName1 = longName2 * (longName3 + longName4 – longName5)
+ 4 * longname6; // PREFER
```

```
longName1 = longName2 * (longName3 + longName4
                        - longName5) + 4 * longname6; // AVOID
```

//Line wrapping for 'If' statements should generally use the double tab rule, since conventional //single tab indentation makes seeing the body difficult. For example:

// DON'T USE THIS INDENTATION

```
if((condition1 && condition2)
   || (condition3 && condition4)
   ||!(condition5 && condition6)) { // BAD WRAPS
    doSomethingAboutIt();          // MAKE THIS LINE EASY TO MISS
}
```

// USE THIS INDENTATION INSTEAD

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

```
}  
// OR USE THIS  
if((condition1 && condition2) || (condition3 && condition4)  
    || !(condition5 && condition6))  
{  
    doSomethingAboutIt ();  
}  
  
// SOQL Format Example  
for (ObjectPermissions tempObjectPermissions : [  
    SELECT  
        ParentId,  
        PermissionsCreate,  
        PermissionsDelete,  
        PermissionsEdit,  
        PermissionsRead,  
        PermissionsModifyAllRecords,  
        PermissionsViewAllRecords,  
        SObjectType  
    FROM  
        ObjectPermissions  
    WHERE  
        Parent.Name = :permissionSetName  
    ]) {  
    objectPermissionMap.put(tempObjectPermissions.SObjectType,  
        tempObjectPermissions);  
}
```

5.2 Commenting

Guidelines

Commenting involves placing human readable descriptions inside of computer programs detailing what the code is doing. Proper use of commenting can make code maintenance much easier, as well as helping make finding bugs faster. Further, commenting is very important when writing functions that other people will use. Remember, well documented code is as important as correctly working code.

All programs should be commented in such a manner as to easily describe the purpose of the code and any algorithms used to accomplish the purpose. A user should be able to utilize a previously written program without ever having to look at the code, simply by reading the comments. Comments should be used to give an overview of code and provide additional information that is not readily available in the code itself. Comments should contain only information that is relevant to reading and understanding the program.

- Comments should not be enclosed in large boxes drawn with asterisks or other characters.
- Comments should never include special characters such as form-feed and backspace.

Code can have following styles of implementation comments: documentation, block, single-line, trailing, end-of-line and HTML comments.

Comments should occur in the following places:

- The top of any program file. This is called the "Header Comment". It should include all the defining information about who wrote the code, and why, and when, and what it should do. Block comments style must be used for this. Apex class and Trigger must use Documentation comments. VisualForce Page, VisualForce Component must use HTML comments.
- Above every method. This is called the method header and provides information about the purpose of this "sub-component" of the program. Documentation comments style must be used for this. Always recommended to provide details of input parameters passed to the method if there are any. Mention what inputs are expected to the method.
- In line. Any "tricky" code where it is not immediately obvious what you are trying to accomplish, should have comments right above it or on the same line with it. Block and Single line comments are the preferred styles for the inline comments. Trailing and End-of-line comments styles can also be used.

Documentation Comments

Documentation comments describe program files such as classes, interfaces, triggers, constructors, methods, and fields. Each documentation comment is set inside the comment delimiters `/** ... */`, with one comment per API. This comment should appear just before the code declaration.

Documentation Comments Example (Class)

```
/**
 * VisualForce page controller. Using the same controller will ensure
 * that state is maintained during page navigations.
 *
 * @author John Smith
 * @date 01/20/2016
 */
```

Block Comments

Block comments are used to provide descriptions of files, methods, data structures and algorithms. A block comment should be preceded by a blank line to set it apart from the rest of the code. Block comments have an asterisk “*” at the beginning of each line except the first.

Block Comments Example (Class)

```
/*
 * Invoked before loading a page.
 * Right now, only language is set.
 * Do not perform any sensitive/user based actions here.
 */
```

Single Line Comments

Short comments can appear on a single line indented to the level of the code that follows. If a comment can't be written in a single line, it should follow the block comment format. A single-line comment should be preceded by a blank line.

Every class variable should have a comment associated denoting what the variable is used for.

Single Line Comments Example

```
/* Parses(deserializes) the JSON string into respective Apex class. */

//[DeveloperName DD-MMM-YY]: Variable to avoid recursion.
public static boolean blsFirstInvoke=true;
```

Trailing Comments

Very short comments can appear on the same line as the code they describe, but should be shifted far enough to separate them from the statements. If more than one short comment appears in a chunk of code, they should all be indented to the same tab setting.

Trailing Comments Example

```
If (iStatusCode == 2){
    bResult = true; /* status code 2 is always considered success */
}
```

End-Of-Line Comments

The // comment delimiter begins a comment that continues to the newline. It can comment out a complete line or only a partial line. It shouldn't be used on consecutive multiple lines for text comments; however, it can be used in consecutive multiple lines for commenting out sections of code.

End-Of-Line Comments Example

```
// initialize the object since the web service may return null.
```

HTML Comments

Comment tags <!-- and --> are used to insert comments in VisualForce page. HTML comments are ignored by the web browsers and hence will not be displayed in the screen. HTML comments should be used for documenting the VisualForce pages, VisualForce components and Lightning components.

HTML Comments Example

```
<!-- *****
* NAME      : <COMPONENT NAME>
* DESCRIPTION: WRITE DESCRIPTION HERE
*
* @AUTHOR <DEVELOPER NAME>
* @DATE MM/DD/YYYY
*
*
* MODIFICATION LOG:
* DEVELOPER      DATE      DESCRIPTION
* -----
* <DEVELOPERNAME> MM/DD/YYYY  WRITE DESCRIPTION HERE
* <DEVELOPERNAME> MM/DD/YYYY  WRITE DESCRIPTION HERE
-->
```

5.3 Apex Class

Do Not Hardcode IDs

When deploying Apex code between sandbox and production environments, or installing Force.com AppExchange packages, it is essential to avoid hard coding IDs in the Apex code. As the record IDs change between environments, the logic should dynamically identify the proper data to operate against and not fail.

The following example queries for the record types in the code, stores the dataset in a map collection for easy retrieval, and ultimately avoids any hard coding. By ensuring no IDs are stored in the Apex code, you are making the code much more dynamic and flexible – and ensuring that it can be deployed safely in different environments.

Prevent SOQL Injection by Escaping

```
// If dynamic SOQL is used, the SOQL should be escaped
// using String.escapeSingleQuotes() method. For example:
String strTaskQueryAll = 'SELECT id, ownerid, subject, status, description, activitydate FROM Task';
strTaskQueryAll = strTaskQueryAll + 'WHERE isclosed = false ' + filterAssign + filterStatus + filterSubject ;
strTaskQueryAll = strTaskQueryAll + ' AND CampaignID_c = \'' + string.escapeSingleQuotes(strCampaignID)
+ '\' LIMIT 400';
Database.Query(strTaskQueryAll);
```

Hardcoding Example

○ Right Way

```
//Create a map between the Record Type Name and RecordTypeInfo for easy retrieval
Map<String, Schema.RecordTypeInfo> accountRecordTypes =
Schema.SObjectType.Account.getRecordTypeInfosByDeveloperName();

for (Account a: Trigger.new) {
    //Use the Map collection to dynamically retrieve the Record Type Id
    //Avoid hardcoding Ids in the Apex code
    If (accountRecordTypes.containsKey('Healthcare') && a.RecordTypeId ==
accountRecordTypes.get('Healthcare').getRecordTypeId()) {
        //do some logic here
    } else if (accountRecordTypes.containsKey('High_Tech') && a.RecordTypeId ==
accountRecordTypes.get('High_Tech').getRecordTypeId()) {
        //do some logic here for a different record type...
    }
}
```

○ Avoid

```
for (Account a:Trigger.new) {  
    //Error – hardcoded the recordtype id  
    If (a.RecordTypeId == '032310232213War') {  
        //do some logic here  
    } else if (a.RecordTypeId=='0123434433454Km') {  
        //do some logic ...  
    }  
}
```

Query Large Data Sets in For Loop

SOQL queries that return multiple records can only be used if the query results do not exceed 50,000 records. If the query returns more than 50,000 records, then a SOQL query for loop must be used instead.

Large Data Sets Example

```
// A runtime exception is thrown if this query returns 50,001 or more records. Account[] accts = [SELECT Id  
FROM Account];  
for (List<Account> accts : [Select id, name FROM Account WHERE Name LIKE 'Acme']) {  
    //Your code here  
}
```

Avoid SOQL & DML Operations inside Loop

There is a governor limit that enforces a maximum number of SOQL queries. When queries are placed inside a 'for' or 'while' loop, a query is executed on each iteration and the governor limit is easily reached. Instead, move the SOQL query outside of the 'for' or 'while' loop and retrieve all the necessary data in a single query.

Following points can be considered for avoiding Heap size:

- Query only what you need
- Avoid querying for long-text and rich-text fields
- Load records in SOQL loop instead of loading all records
- See example below

Query within Loop Example

○ Right Way

```
trigger Account_UpdateContacts on Account (before insert, before update) {  
    //This queries all Contacts related to the incoming Account records in a single SOQL  
    //This is also an example of how to use child relationships in SOQL  
    List<Account> accountsWithContacts = [SELECT Id, Name, (SELECT Id, Salutation, Description,  
    FirstName,  
                                LastName, Email FROM Contact)
```

```

        FROM Account WHERE Id =: accountId];
    List<Contact> contactsToUpdate = new List<Contact>{};
    //For loop to iterate through all the queries Account records for
    (Account a: accountsWithContacts) {
        //Use the child relationships dot syntax to access the related Contacts for
        (Contact c: a.Contacts) {
            c.Description = c.salutation + ' ' + c.firstname + ' ' + c.lastname;
            contactsToUpdate.add(c);
        }
    }

    //Now outside the FOR loop, perform a single Update DML statement.
    update contactsToUpdate;
}

```

○ Avoid

```

trigger Account_UpdateContacts Account (before insert, before update) {
    //For loop to iterate through all the incoming Account records
    for (Account a : Trigger.new) {
        /* THIS FOLLOWING QUERY IS INEFFICIENT AND DOESN'T SCALE
        * Since the SOQL Query for related Contacts is within the FOR loop, if this trigger is
        * executed with more than 100 records, the trigger will exceed the trigger governor limit
        * of maximum 100 SOQL Queries issued.
        */

        List<Contact> contacts = [SELECT Id, Salutation, FirstName,
                                LastName, Email FROM Contact where accountId =: a.Id];
        for (Contact c: contacts) {

            /* THE FOLLOWING DML STATEMENT IS INEFFICIENT AND DOESN'T SCALE
            * Since the UPDATE dml operation is within the FOR loop, if this
            * trigger with more than 150 records, the trigger will exceed the
            * trigger governor of maximum 150 DML Operations
            */

            update c;
        }
    }
}

```

○ Avoiding Heap Size

```

//In below example, Apex maintains a query locator so that the operation still counts as a single query
for (List leads : [Select ID, LastName from Lead] ) {
    //SOQL loop loads 200 records at time

```



```

    for (Lead Id : leads) {
        // perform operation here
    }
}

//Avoid following as it can result in heap size limitations
List leads = [Select ID, LastName from Lead];
for(Lead lead : leads) {
    // perform operation here
}

```

Use Collections to Optimize

It is important to use Apex collections to efficiently query data and store the data in memory. A combination of using collections and streamlining SOQL queries can substantially help writing efficient Apex code and avoid governor limits.

In the below example, there is unnecessary querying of the opportunity records in two separate queries. Use of two inner 'for' loops that redundantly loop through the list of opportunity records just trying to find the ones related to a specific account.

Users sometimes might face "Aggregate query has too many rows for direct assignment" error when they are when accessing a large set of child records (200 or more) of a retrieved sObject inside the loop. Check below example to see how to avoid such errors.

Few Important points:

- Collections does not have limits in Salesforce. Limit is on the data that you are querying and storing.
- Use collections with caution as they contribute more in view state. So, clear the collections once they are no longer required.
- Maps can store NULL as a key, so make sure to perform null check. Else you may get unexpected comparison results.
- Avoid using Lists when you frequently need to check if element you are inserting is in list or not. Instead use Sets or Maps to optimize the performance.

Collections Example

○ Right Way

```

List<Account> accountsWithOpptys = [SELECT Id, Name, (SELECT Id, Name, ClosedDate, StageName
FROM Opportunity
                                WHERE AccountId IN : trigger.newMap.keySet()
                                AND (StageName = 'Closed - Lost' OR
                                    StageName = 'Closed - Won' ))

```

```

        FROM Account WHERE Id IN : trigger.newMap.keySet();
    for (Account eachAccWithOpp : accountsWithOpptys) {
        for (Opportunity eachOpp: eachAccWithOpp.Opportunities) {
            if (eachOpp.StageName == 'Closed – Won') {
                System.debug('Do more logic here...');
            }
            else if (eachOpp.StageName == 'Closed – Lost') {
                System.debug('Do more logic here...');
            }
        }
    }
}

```

○ Avoid

```

trigger Account_UpdateOpptys on Account (before delete, before insert, before update) {
    //This code inefficiently queries the Opportunity object in two separate queries
    List<Opportunity> opptysClosedLost = [SELECT Id, Name, ClosedDate, StageName

        FROM Opportunity WHERE
        AccountId IN :Trigger.newMap.keySet()
        AND StageName = 'Closed – Lost'];

    List<Opportunity> opptysClosedWon = [SELECT Id, Name, ClosedDate, StageName
        FROM Opportunity WHERE
        AccountId IN :Trigger.newMap.keySet()
        AND StageName='Closed – Won'];

    for (Account a : Trigger.new) {
        //This code inefficiently has two inner FOR loops
        //Redundantly processes the List of Opportunity Lost
        for (Opportunity o: opptysClosedLost) {
            if (o.accountid == a.id)
                System.debug('Do more logic here...');
        }

        for (Opportunity o: opptysClosedWon){
            if (o.accountid == a.id)
                System.debug('Do more logic here...');
        }
    }
}

// Avoid querying into single object
// Non compliant Code Example:-
Contact c = [SELECT Status__c FROM Contact WHERE Id=:ID];

```

```
c.Status__c = status;
update c;
```

○ Aggregate Query

```
// Aggregate query has too many rows for direct assignment
for (Account eachAccount: [SELECT Id, Name, (SELECT Id, Name FROM Contacts)
    FROM Account WHERE Id IN(")]) {
    List<Contact> contactList = eachAccount.Contacts; // Causes an error
    Integer count = eachAccount.Contacts.size(); // Causes an error
}

// To avoid such errors, use a for loop to iterate over the child records, as follows:
for (Account eachAccount: [SELECT Id, Name, (SELECT Id, Name FROM Contacts)
    FROM Account WHERE Id IN(")]) {
    Integer count = 0;
    for (Contact eachContact: eachAccount.Contacts) {
        //logic goes here
    }
}

// Avoid calling sizeof on a soql statement.
integer iSize = [ select id from table ].size();
```

Ensure Code is 'Bulk Safe'

Ensuring Apex code is 'bulk safe' 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. All of the records need to be processed as a bulk, in order to write scalable code and avoid hitting governor limits.

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

The correct code iterates across the entire Trigger.new collection with a for loop. Now if this trigger is invoked with a single Account or up to 200 Accounts, all records are properly processed.

For Apex or SOAP API, it is possible to use a list of sObjects of different types in same DML statement. The chunking of the input array for an insert or update or delete DML operation has two possible causes:

- The existence of multiple object types
- The default chunk size of 200.

If chunking in the input array occurs because of both of these reasons, each chunk is counted toward the limit of 10 chunks. If the input array contains only one type of sObject, you won't hit this limit. However, if the input array contains at least two sObject types and contains a high number of objects that are chunked into groups of 200, you might hit this limit.

For example, if you have an array that contains 1,001 consecutive leads followed by 1,001 consecutive contacts, the array will be chunked into 12 groups: Two groups are due to the different sObject types of Lead and Contact, and the remaining are due to the default chunking size of 200 objects. In this case, the insert or update operation returns an error because you reached the limit of 10 chunks in hybrid arrays. The workaround is to call the DML operation for each object type separately.

Reference :

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

Bulk Code Example

○ Right Way

```
Trigger Account_Update on {Account (before insert, before update) {
    List<String> accountNames = new List<String>();
    // Loop through all records in the Trigger.new collection
    for (Account a: Trigger.new) {
        // concatenate the Name and billingState into the Description field
        a.Description = a.Name + ':' + a.BillingState
    }
}
```

○ Avoid

```
trigger Account_Update on Account (before insert, before update) {
    // This only handles the first record in the Trigger.new collection
    // But if more than one Account initiated this trigger, those additional records
    // will not be processed
    Account acc = trigger.new[0];
    List<Contact> contacts = [SELECT Id, Salutation, FirstName, LastName, Email
                             FROM Contact WHERE AccountId = :acc.Id];
}
```

Constant Declaration

○ Need for Constant

Using hard coded constants should be avoided whenever possible. Preferably, global (global static final) and application (public static final) constants should be externalized into constant classes. Global static constants should be used for things that never change like days of the week, months of the year, etc. By making them global, this reduces the chance of duplication between applications.

○ Constant Declaration Example

```
// Constants need to be initialized as Static Final
public class TerritoryOutletAssignmentStatus {
    public static final string WAITING_REMOVAL = 'Waiting Removal';
    public static final string REMOVED = 'Removed';
}

// In no case should literals (especially string literals) be used in code logic. For example, the following code:
if (to.Active c == 'Waiting Removal')
    assignmentStatus = to.Territory r.Name;
else if (to.Active c == 'Removed')
    assignmentStatus = to.Territory r.Name;
// Should be rewritten as:
If (to.Active c == TerritoryOutletAssignmentStatus.WAITING_REMOVAL)
    assignmentStatus = to.Territory r.Name;
else if (to.Active c == TerritoryOutletAssignmentStatus.REMOVED)
    assignmentStatus = to.Territory r.Name;

// When comparing 2 String literals use String.equals() instead of == operator.

// == operator compares string ignoring character case.
if(to.Active c == 'Waiting Removal')

// equals() compares case of String literals as well and then only return true.
if('Waiting Removal'.equals(to.Active c))
```

Other Best Practices

DML

1. Always perform bulk DML operations. This would in turn help in reducing the number of SOQLs.
2. Use Database.<dml> for partial processing of records.
3. Do not perform DML in loops.
4. Use try-catch blocks. Have an error logging mechanism to log errors which get caught in the catch block.
5. It is recommended to use transaction controls like database.setsavepoint and database.rollback in case if

DML and Callouts

DML cannot be allowed in between two callouts. In other words a callout cannot be made after a DML within the result in pending uncommitted work that prevents callouts from executing.

SOQL

1. Use LIMIT while writing a SOQL.
2. Make sure that at least one indexed field is present in the WHERE clause.
3. Do not check for a field's value to be NULL in the SOQL. Instead, write a SOQL and then do the NULL check.
4. Do not write SOQL in loops.
5. It is recommended to use 'string.escapeSingleQuote' while building the Dynamic Query to avoid the Cross SQL Injection.
6. Leverage Aggregate Queries to fetch the counts/sum or similar functionality.
7. When using Dynamic SOQL, you can try to see if you can use FieldSets to configure fields on VF as well as SOQL.
8. Avoid calling SOQL with negative expressions in SOQL where clause i.e., use '= somevalue' instead of '!= somevalue'.

SOQL versus SOSL

Use SOQL when

1. You know in which objects or fields the data resides.
2. You want to retrieve data from a single object or from multiple objects that are related to one another.
3. You want to count the number of records that meet specified criteria.
4. You want to sort results as part of the query.
5. You want to retrieve data from number, date, or checkbox fields.

Use SOSL when

1. You don't know in which object or field the data resides and you want to find it in the most efficient way possible.
2. You want to retrieve multiple objects and fields efficiently, and the objects may or may not be related to one another.
3. You want to retrieve data for a particular division in an organization using the divisions feature, and you want to retrieve data from multiple objects.

Apex Class

1. Avoid code duplication. Create separate functions (preferably in Utils classes) and invoke the same from throughout the class. Also, nullify the data structures holding huge data when they are no longer needed.
2. While comparing a static string to a field value, check against the static string not against the field value:
 1. For e.g: Approval.status.Equals('Submitted') - if status is blank, null pointer error
 2. Best Practice: 'Submitted'.equals(Approval.status)
3. While using boolean expressions, avoid using unnecessary comparisons.
4. All empty methods or code block of a class should be commented.
5. Code written inside loops(for or while) should be included in braces.
6. Common mistake while performing check on the lists:
 1. if(acctList.size() > 0) - in order to check the size system iterates through all the n items. So, if you find the element after iterating to last element
 2. if(!acctList.isEmpty()) - fast and efficient as it returns "True" immediately after 1st element.

System.Debug

1. Usage of System.Debug should be kept to a minimum as will affect APEX trigger or Visualforce page performance.
2. Avoid debug statements in "For" loop will slow down the page performance. Keep log statements to a minimum and use them before deployment to UAT/Production.

1. A system debug within a for loop takes almost 10 times longer processing time. The impact exists e for further details:

<https://salesforce.stackexchange.com/questions/41063/writing-too-much-to-debug-log-would-affect->

View State

Developer's should also monitor view state of the pages developed by enabling Development Mode and View St: structured breakup for View State for page.

Best Practices (Code)

○ Best Practices in DML (Code)

```
/**
 * The AccountTriggerHandler implements an application that
 * runs logic for after insert trigger on Account.
 * @author Deloitte
 * @version 1.0
 * @since 2014-03-31
 */
public class AccountTriggerHandler{
    /**
    * This method is used to run a DML on Account after insert.
    * @param none
    * @return void.
    */
    public static void isAfterInsert(){
        List<Account> accountList = new Account();
        for(i =0;i< 10;i++){
            Account a = createAccount();
            accountList.add(a);
        }
        if(!accountList.isEmpty() && accountList.size() > 0){
            Database.SaveResult[] saveResultList = Database.insert(accountList, false);
            for(Database.SaveResult dbs: saveResultList){
                if(dbs.isSuccess()){
                    //code here
                }
            }
        }
        else{
            for(Database.Error db: dbs.getErrors()){
                System.Debug(db.getFields()); //Returns an array of one or more field names.
                System.Debug(db.getMessage()); //Returns the error message text.
                System.Debug(db.getStatusCode()); //Returns the error code.
                //log exception here in a list of sObject
            }
        }
    }
}
```

```

    }
    }//end for
    //insert error list
    }
}
public static Account createAccount(){
    return new Account(Name='PHX_Example');
} //end method
} //end class

```

○ Best Practices in SOQL (Code)

```

/*****code in trigger*****/
Set<Id> setAcclDs = new Set<Id>();
for(Account acc: Trigger.New){
    setAcclDs.add(acc.Id);
}
//list declaration and instantiation
List<Account> lstAccount = [SELECT
    Id, Name, Email, BillingCountry
    FROM
    Account
    WHERE
    Id IN: setAcclDs AND
    Name LIKE '%ARB%'
    LIMIT 10000
    ORDER BY Name];

//Map declaration and Instantiation
Map<Id, Account> = new Map<Id, Account>([SELECT
    Id, Name, Email, BillingCountry
    FROM
    Account
    WHERE
    Id IN: setAcclDs AND
    Name LIKE '%ARB%'
    LIMIT 10000 ORDER BY Name]);

```

○ Avoid Empty Blocks

```

//Non-Compliant Code
// For e.g. Empty If Statement finds instances where a condition is checked but nothing is done about it.
if(i == 0) {
}

```


Access Modifiers and Sharing

○ Access Modifiers and Sharing

Access Modifiers - Private, Protected, Public, Global

Private should be used for:

- Test class declaration: Tests should only work within the scope of the class.
- Variables which need not be exposed to a page or any other class.

Protected should be used for:

- To expose variables to the inner classes and extended ones.
- It offers more access than private.

Public should be used for:

- Apex Controllers, Apex Classes being used in Trigger and Batch Classes etc.

Global should be used for:

- Apex Controllers, Classes and Batches that performs web service callouts (if advised).
- Rest resources.

Sharing

The 'with sharing' keyword is used to respect the record access of a user.

1. Decide on when to use 'with sharing' vs 'without sharing'.
 1. Tip: Apex Controllers should ideally be 'with sharing' to give proper record access.
2. When using with sharing, always add a check before performing a DML(isCreateable(), etc).
3. A class inherits the sharing of the class where it is being called from. For example - If Class A calls Class B and Class B is declared as with sharing, then Class A will also enforce record access/sharing rules governing the record irrespective of how it is declared.
4. FLS (Field Level Security) should be enforced within the class using the Schema that is provided by Apex. This is because the apex by default runs in System mode. Refer to the below link for more details - https://developer.salesforce.com/page/Enforcing_CRUD_and_FLS

With Winter-19 release, Salesforce has introduced a new sharing keyword "inherited sharing". This has significant advantage over With Sharing and Without Sharing keyword whenever sharing is accidentally omitted due to execution context. Apex without a sharing declaration is insecure by default.

If the class is used as the entry point to an Apex transaction, an omitted sharing declaration runs as without sharing. However, inherited sharing ensures that the default is to run as with sharing. A class declared as inherited sharing runs only as without sharing when explicitly called from an already established without sharing context.

Reference : https://releasenotes.docs.salesforce.com/en-us/winter19/release-notes/rn_apex_inherited_sharing.html

○ Apex Sharing Best Practice in SOQL(Code)

```
//Noncompliant Code Example:-  
public without sharing class Foo {  
    // SOQL here  
}
```

```
//Compliant Solution:-  
public with sharing class Foo {  
    // SOQL here  
}
```

Keywords in Apex Classes

Final

- Constants that have as a non-changing value within the transaction should be declared as final.

Super

- Only classes that are extending from virtual or abstract classes can use super.
- You can only use super in methods that are designated with the override keyword.
- It is used to access methods defined in the parent abstract/virtual class.

Transient

- Variables that were private initially and are being used as an intermediate in an Apex Controller to finally arrive at the value to be displayed on the page, can be changed to transient to reduce the View State of the page.

Extends

- Virtual and abstract classes are extended.
- Abstract/Virtual classes include reusable functions for a specific functionality that can be overridden and used in the child class.

Implements

- Interfaces are implemented by other classes.
- Interfaces provide a skeleton for a class to follow.

Governor Limits (Sync)

- In a SOQL query with parent-child relationship subqueries, each parent-child relationship counts as an extra query. These types of queries have a limit of three times the number for top-level queries. The row counts from these relationship queries contribute to the row counts of the overall code execution. In addition to static SOQL statements, calls to the following methods count against the number of SOQL statements issued in a request.
 - Database.countQuery
 - Database.getQueryLocator
 - Database.query
- Calls to the following methods count against the number of DML queries issued in a request.
 - Approval.process
 - Database.convertLead
 - Database.emptyRecycleBin
 - Database.rollback
 - Database.setSavePoint
 - delete and Database.delete
 - insert and Database.insert
 - merge and Database.merge
 - undelete and Database.undelete
 - update and Database.update
 - upsert and Database.upsert
 - System.runAs
- Recursive Apex that does not fire any triggers with insert, update, or delete statements exists in a single invocation, with a single stack.
- CPU time is calculated for all executions on the Salesforce application servers occurring in one Apex transaction. CPU time is calculated for the executing Apex code, and for any processes that are called from this code, such as package code and workflows. CPU time is private for a transaction and is isolated from other transactions. Operations that don't consume application server CPU time aren't counted toward CPU time. For example, the portion of execution time spent in the database for DML, SOQL, and SOSL isn't counted, nor is waiting time for Apex callouts.
- In order to reduce the Viewstate:
 - Query only the fields which you need to process
 - Use Maps efficiently. All collections should be flushed out when you no longer need them.
 - Manage to reduce the viewstate. Too many forms may hamper performance.
 - See if you can use Readonly=true on VF based on requirement.
- For Callouts
 - 100 callouts for within a synchronous transaction in a single transaction.
 - 10 seconds is the default timeout for callouts in a single transaction.
 - Maximum cumulative timeout for all callouts (HTTP requests/Web services calls) in a transaction is 120 seconds
- SOQL and SOSL
 - 100 SOQL within a synchronous transaction.
 - 20 SOSL within a transaction.
 - 50000 maximum number of records retrieved across all SOQL

- 2000 maximum number of records retrieved by a single SOSL query
- **DML**
 - 150 DML within a transaction.
 - 10000 total number of records processed as a result of DML statements
- 10 sendEmail methods allowed in a transaction
- Email services heap size is 36 MB.
- Maximum execution time for each Apex transaction is 10 minutes
- Heap Size
 - 6 MB of space is provided for a transaction.
- View State
 - View state involves data stored for the working of the VF Page. The limit is 135KB.
- How to avoid Limit Exceptions?
 - Always bulkify your code and use LIMIT in queries.
 - Use transient to reduce view state.
 - Use future/queueable methods for callouts from triggers.
 - Use Limits class to keep your SOQL and DML limits in check.
- An important consideration while discussing Governor limits is the ability to cache attributes to avoid hitting governor limits. One common request on implementations is to ensure that does lookup to metadata do not inadvertently expose the developer to governor limit exceptions when writing code for triggers and related helper classes. Common issues are
 - Looking up a profile name from a given profile id.
 - Looking up record type names from a given record type id.

To avoid executing multiple DML statements to query for the above metadata it is recommended that an utility class is implemented to cache these values per Apex invocation. Other methods can be added by the developers to build upon the framework where necessary. By caching the attribute required in a private static variable each request for data can be controlled to ensure only one SOQL or DML statement is executed.

- For more details on Limits refer to
 - https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_gov_limits.htm

References

Apex Coding Best Practices - https://developer.salesforce.com/page/Apex_Code_Best_Practices

Best Practices for Controllers and Controller Extensions -

https://developer.salesforce.com/docs/atlas.en-us.pages.meta/pages/pages_compref_additional_controller.htm

Platform Cache Best Practices -

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

5.4 Apex Trigger

Best Practices

- One Trigger per Object.
 - 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.
 - Each trigger that is invoked does not get its own governor limits. Instead, the first trigger defines the entry point and therefore defines the context of the governor limit calculations. So instead of only the one trigger getting a maximum of 20 queries, all triggers on that same object will share those 20 queries.
- Triggers are executed for a batch of maximum 200 records. Logic executed in the trigger needs to be optimized to handle bulk data in the context of Governor limits. For example, if the user does a bulk update on 500 records then the code on the trigger is executed 3 times for batches of 200, 200 and 100 records. The Apex limits used by the trigger logic is thrice so if the logic has 10 SOQL, then the trigger uses 30 SOQL.
- Any logic that needs to be performed should be called via classes through the single trigger. This will make the lifecycle of the record update predictable and controlled.
- Include only necessary trigger contexts and have context specific handler methods.
- Delegate trigger logic in utility/helper classes.
- Use collections (Map and Set data structures) to bulkify trigger logic and 'SOQL For loop' when querying large datasets.
- Avoid Future methods, SOQL and DML statements inside iterations.
- Minimize the number of SOQL statements by pre-processing records and generating sets, which can be placed in single SOQL statement used with the IN clause.
- Always add field validation in the before triggers.
- Use static variables in-order to persist data across before and after triggers.
- Ensure triggers can be turned off/on using custom setting.

Best Practices Example

○ Trigger Example

```
/**
Trigger Name   : PHX_AccountTrigger
Description    : This is a Trigger for the Account object.
Created By     : <developer name>
Created Date   : <name>
Modification Log:
```

Developer	Date	Description
-----------	------	-------------

```

<developer name>      <date>      <changes done description>
*/

trigger PHX_AccountTrigger on Account (before Insert,before Update, after Insert,after Update) {
    /** Invoke the corresponding object's handler class through trigger factory */
    new PHX_AccountTriggerHandler().run();
}

/** SOQL for loops retrieve all sObjects, using efficient chunking with calls
to the query and queryMore methods of the SOAP API. Developers should
always use a SOQL for loop to process query results that return many records,
to avoid the limit on heap size
**/
Account[] accts = new Account[];
for (List<Account> acct : [SELECT id, name FROM account WHERE name LIKE 'Acme']){
    /**Logic here - ensure we don't have any DML statements inside for loops**/
    accts.add(acct);
}
update accts ;

```

○ Bulkification Example

```

/**Set and map data structures are vital for successful coding of bulk triggers**/
// When a new line item is added to an opportunity, this trigger copies the value of the
// associated product's color to the new record.
trigger oppLineTrigger on OpportunityLineItem(before insert) {
    // For every OpportunityLineItem record, add its associated pricebook entry
    // to a set so there are no duplicates.
    Set < Id > pbelds = new Set < Id > ();
    for (OpportunityLineItem oli: Trigger.new) {
        pbelds.add(oli.pricebookentryid);
    }
    // Query the PricebookEntries for their associated product color and place the results
    // in a map.
    Map < Id, PricebookEntry > entries = new Map < Id, PricebookEntry > ([select product2.color__c from
pricebookentry
                                where id in: pbelds]);
    // Now use the map to set the appropriate color on every OpportunityLineItem processed
    // by the trigger.
    for (OpportunityLineItem oli: Trigger.new) {
        oli.color__c = entries.get(oli.pricebookEntryId).product2.color__c;
    }
}

```

○ Limits Methods

```

/** Trigger code snippet which invokes a method with sample limit methods**/
trigger PHX_AccountAsyncTrigger on Account(after insert, after update) {
    //invoke the class where the trigger logic is placed.
    PHX_AccountHandler.processAccounts(Trigger.newMap.keySet());
}

/** Apex class with future implementaion **/
global class PHX_AccountHandler {
    public static void processAccount(Set < Id > accountIds) {
        //query list of contacts associated to account
        List < Contact > contacts = [select id, salutation, firstname, lastname, email from Contact where
accountId IN: accountIds];
        System.debug('1. Number of Queries used in this Apex code so far: ' + Limits.getQueries());
        System.debug('2. Number of rows queried in this Apex code so far: ' + Limits.getDmlRows());
        System.debug('3. Number of DML statements used so far: ' + Limits.getDmlStatements());
        System.debug('4. Amount of CPU time (in ms) used so far: ' + Limits.getCpuTime());
        /**NOTE:Proactively determine if there are too many contacts to update and avoid governor limits**/
        if (contacts.size() + Limits.getDMLRows() > Limits.getLimitDMLRows()) {
            System.debug('Need to stop processing to avoid hitting a governor limit. Too many contacts to
update in this trigger');
            for (Contact c: contacts) {
                System.debug('Contact Id[' + c.Id + '], FirstName[' + c.firstname + '],LastName[' + c.lastname + ']');
                c.Description = c.salutation + ' ' + c.firstName + ' ' + c.lastname;
            }
            update contacts;
        }
    }
}

```

Async methods in Trigger

Queueable/@future in Trigger:

- It is important to make sure that the asynchronous methods are invoked in an efficient manner and that the code in the methods is efficient.
- Additionally, no more than 50 @future methods can be invoked within a single Apex transaction.
- Ensure that the @future methods are not in iteration, instead they should be invoked with a batch of records, so that they are called only once for all records that are needed to processed.
- There can be 50 jobs added to the queue with System.enqueueJob in a single transaction.

- The maximum stack depth for chained jobs is 5, which means that you can chain jobs four times and the maximum number of jobs in the chain is 5, including the initial parent queueable job.
- Consider a scenario where there is a trigger on the Account object and invoking a future method. If a batch update happens of trigger, we get an exception. This exception can be avoided by enclosing the code in `System.isFuture() && System.isBatch()`.

Batch apex in Trigger:

- Batch apex can be invoked from trigger, but with serious considerations on governor limits.
- Ensure trigger invoking the batch is bulkified (may be a max batch size of 200).
- When the job is triggered, the system queues the batch job for processing. If Apex flex queue is enabled in the org, the batch job is added at the end of the flex queue.

System Class Methods:

Salesforce provides static methods on the System Class that allow us to work on system operations, such as writing debug messages and scheduling jobs. Below are some methods that allows the developer to check if the code is being executed on the user context or the system async context.

- `isFuture()` - Returns true if the currently executing code is invoked by code contained in a method annotated with future; false otherwise. Since a future method can't be invoked from another future method, use this method to check if the current code is executing within the context of a future method before you invoke a future method.
- `isBatch()` - Returns true if a batch Apex job invoked the executing code, or false if not. A batch Apex job can't invoke a future method. Before invoking a future method, use `isBatch()` to check whether the executing code is a batch Apex job.
- `isQueueable()` - Returns true if a queueable Apex job invoked the executing code. Returns false if not, including if a batch Apex job or a future method invoked the code. Used to check if the code is being executed in a Queueable context. For example if a Queueable Job updates an object and the trigger in turn calls another Queueable method then the execution fails. This method can be used to skip the execution.

Async Operation Example

○ @future in Trigger

```
/** Trigger code snippet which invokes a future method */
trigger PHX_AccountAsyncTrigger on Account(after insert, after update) {
    //By passing the @future method a set of Ids, it only needs to be
    //invoked once to handle all of the data.
    PHX_AsyncApex.processAccount(Trigger.newMap.keySet());
}

/** Apex class with future implementation */
global class PHX_AsyncApex {
```



```
//cannot call future from future or batch
public static void processAccount(Set < Id > accountIds) {
    if (!System.isFuture() && !System.isBatch()) {
        processAccountFuture(accountIds);
    }
}
@future(callout = true)
public static void processAccountFuture(Set < Id > accountIds) {
    // Future Callout Logic
}
}
```

○ Batch from Trigger

```
/** Trigger code snippet which invokes a future method */
trigger PHX_AccountAsyncTrigger on Account(after insert, after update) {
    /**trigger handler will invoke the batch apex*/
    if (!System.isBatch()) { // If the batch should not be invoked again from batch apex
        PHX_AsyncApex.invokeBatchApex(Trigger.new);
    }
}

/** Apex class which invokes batch apex */
global class PHX_AsyncApex {
    public static void invokeBatchApex(List < Account > accList) {
        /** invoke a batch apex */
        Database.executeBatch(new UpdateAccountArea(accList));
        //Exception handling to be done at function level on batch apex.
    }
}
```

○ Queueable from Trigger

```
/** Trigger code snippet which invokes a future method */
trigger PHX_AccountAsyncTrigger on Account(after insert, after update) {
    //Invoke Queueable class from trigger handler
    // Keep a note on chaining limitations as described above
    try {
        //if the snippet should not be executed again from Queueable context
        if (!System.isQueueable()) {
            PHX_AsyncApex.queueSample();
        }
    } catch (Exception ex) {
        System.debug('Queueable Exception' + ex.getMessage());
    }
}
```

```

    }
}

/**Trigger Handler**/
public class PHX_AsyncApex {
    public static void queueSample() {
        List < Account > acctList = Trigger.new; // Get this from a trigger call
        // Assemble List of IDs for queueable method
        List < Id > acctIdList = new List < Id > ();
        for (Account a: acctList) {
            acctIdList.add(a.Id);
        }
        if (acctIdList != null && acctIdList.size() > 0) {
            Id jobId = System.enqueueJob(new PHX_ServiceQueue(acctIdList));
        }
    }
}

/**Queueable class which will be invoked from trigger**/
public class PHX_ServiceQueue implements Queueable, Database.AllowsCallouts {
    public final List < Id > idList;
    public PHX_ServiceQueue(List < Id > idList) {
        this.idList = idList;
    }
    public void execute(QueueableContext context) {
        // This method will make a REST API call
        PHX_Service.accountPaymentService(idList);
    }
}

```

Trigger Framework

Why use a Framework ?

- Trigger patterns enforce a logical sequence to the trigger code and in turn help to keep code tidy and more maintainable.
- This pattern involves delegating work from the trigger to a structured Trigger Handler class.
- Each object will have its own trigger handler. The trigger itself has almost no code in it.
- Enforces consistent implementation of Trigger logic.
- Implement tools, utilities, and abstractions to make your handler logic as lightweight as possible.

What Can Frameworks Do?

Routing Abstractions:

- In the example code for the “Context-Specific Handler Methods”, you probably noticed that I still needed to implement routing logic in the trigger. This logic looked at the context that the trigger was running in and dispatched to the correct handler method. This is something that an underlying Trigger framework could easily handle for you.

Recursion Detection and Prevention:

- A Trigger framework can also act like a watchdog over all Trigger executions. A common mistake that developers have is building their Trigger logic so that Triggers execute recursively. When this happens, unexpected things may occur and even governor limits can be hit. Your Trigger framework can detect this and figure out how to handle the situation properly.

Centralize Enable/Disable of Triggers:

- As someone who knows the pain of deploying a Trigger to production that contains bugs, I can tell you that giving yourself the ability to easily disable a trigger that isn’t functioning properly is a huge win. Your Trigger framework can easily be wired up to a Custom Setting in your org to give you on/off control of your triggers. This is a great thing to have when you need to buy some time to patch broken code and don’t want your users to be interrupted by Apex exceptions on their pages!

Trigger Framework (Code)

○ Trigger Code

```
/**
 * @author : Author name
 * @date : 01/09/2019
 * @description : Trigger on Account Object
 **/
trigger PHX_AccountTrigger on Account (after insert, after update) {
    new PHX_AccountTriggerHandler().run();
}
```

○ Account Trigger Handler

```
/**
Trigger Name   : PHX_AccountTriggerHandler
Description    : This is a Trigger Handler for the Account object.
Created By     : <developer name>
Created Date   : <name>
Modification Log:
```

Developer	Date	Description

```

<developer name>      <date>      <changes done description>
*/
public class PHX_AccountTriggerHandler extends TriggerHandler {
    private Map<Id, Account> newAccDet;
    //constructor for variable assignment
    public PHX_AccountTriggerHandler (){
        this.newAccDet = (Map<Id, Account>) Trigger.newMap;
        // sets the number of times account handler has to be executed, can be used to prevent recursion.
        this.setMaxLoopCount(1);

    }
    //overriding the trigger handler
    public override void afterUpdate() {
        //logic to be added for after update context
        List<Account> accList = [SELECT Id,Name,Type FROM Account WHERE Id IN
:Trigger.newMap.keySet()];
        // ensure the handler is run once by setting the max loop count.
        update accList ;
        //Specify the handler to be bypassed
        TriggerHandler.bypass('PHX_OpportunityTriggerHandler');
        List<Opportunity> oppList = [SELECT Id,AccountId from Opportunity where AccountId IN
:Trigger.newMap.keySet()];
        //Above handler will ensure, opportunity trigger handler is not executed.
        update oppList;

    }

}

```

○ Trigger Handler

```

/**
 * @author : Author name
 * @date : 19/02/2019
 * @description : Generic TriggerHandler
 */
public virtual class TriggerHandler {

    // static map of handlername, times run() was invoked
    private static Map < String, LoopCount > loopCountMap;
    private static Set < String > bypassedHandlers;

    // the current context of the trigger, overridable in tests
    @TestVisible

```

```

private TriggerContext context;

// the current context of the trigger, overridable in tests
@TestVisible
private Boolean isTriggerExecuting;

// static initialization
static {
    loopCountMap = new Map < String, LoopCount > ();
    bypassedHandlers = new Set < String > ();
}

// constructor
public TriggerHandler() {
    this.setTriggerContext();
}

/*****
 * public instance methods
 *****/

// main method that will be called during execution
public void run() {

    if (!validateRun()) return;

    addToLoopCount();

    // dispatch to the correct handler method
    if (this.context == TriggerContext.BEFORE_INSERT) {
        this.beforeInsert();
    } else if (this.context == TriggerContext.BEFORE_UPDATE) {
        this.beforeUpdate();
    } else if (this.context == TriggerContext.BEFORE_DELETE) {
        this.beforeDelete();
    } else if (this.context == TriggerContext.AFTER_INSERT) {
        this.afterInsert();
    } else if (this.context == TriggerContext.AFTER_UPDATE) {
        this.afterUpdate();
    } else if (this.context == TriggerContext.AFTER_DELETE) {
        this.afterDelete();
    } else if (this.context == TriggerContext.AFTER_UNDELETE) {
        this.afterUndelete();
    }
}

```

```

    }

}

public void setMaxLoopCount(Integer max) {
    String handlerName = getHandlerName();
    if (!TriggerHandler.loopCountMap.containsKey(handlerName)) {
        TriggerHandler.loopCountMap.put(handlerName, new LoopCount(max));
    } else {
        TriggerHandler.loopCountMap.get(handlerName).setMax(max);
    }
}

public void clearMaxLoopCount() {
    this.setMaxLoopCount(-1);
}

/*****
 * public static methods
 *****/

public static void bypass(String handlerName) {
    TriggerHandler.bypassedHandlers.add(handlerName);
}

public static void clearBypass(String handlerName) {
    TriggerHandler.bypassedHandlers.remove(handlerName);
}

public static Boolean isBypassed(String handlerName) {
    return TriggerHandler.bypassedHandlers.contains(handlerName);
}

public static void clearAllBypasses() {
    TriggerHandler.bypassedHandlers.clear();
}

/*****
 * private instancemethods
 *****/

@TestVisible
private void setTriggerContext() {

```

```

        this.setTriggerContext(null, false);
    }

    @TestVisible
    private void setTriggerContext(String ctx, Boolean testMode) {
        if (!Trigger.isExecuting && !testMode) {
            this.isTriggerExecuting = false;
            return;
        } else {
            this.isTriggerExecuting = true;
        }

        if ((Trigger.isExecuting && Trigger.isBefore && Trigger.isInsert) ||
            (ctx != null && ctx == 'before insert')) {
            this.context = TriggerContext.BEFORE_INSERT;
        } else if ((Trigger.isExecuting && Trigger.isBefore && Trigger.isUpdate) ||
            (ctx != null && ctx == 'before update')) {
            this.context = TriggerContext.BEFORE_UPDATE;
        } else if ((Trigger.isExecuting && Trigger.isBefore && Trigger.isDelete) ||
            (ctx != null && ctx == 'before delete')) {
            this.context = TriggerContext.BEFORE_DELETE;
        } else if ((Trigger.isExecuting && Trigger.isAfter && Trigger.isInsert) ||
            (ctx != null && ctx == 'after insert')) {
            this.context = TriggerContext.AFTER_INSERT;
        } else if ((Trigger.isExecuting && Trigger.isAfter && Trigger.isUpdate) ||
            (ctx != null && ctx == 'after update')) {
            this.context = TriggerContext.AFTER_UPDATE;
        } else if ((Trigger.isExecuting && Trigger.isAfter && Trigger.isDelete) ||
            (ctx != null && ctx == 'after delete')) {
            this.context = TriggerContext.AFTER_DELETE;
        } else if ((Trigger.isExecuting && Trigger.isAfter && Trigger.isUndelete) ||
            (ctx != null && ctx == 'after undelete')) {
            this.context = TriggerContext.AFTER_UNDELETE;
        }
    }

    // increment the loop count
    @TestVisible
    private void addToLoopCount() {
        String handlerName = getHandlerName();
        if (TriggerHandler.loopCountMap.containsKey(handlerName)) {
            Boolean exceeded = TriggerHandler.loopCountMap.get(handlerName).increment();
            if (exceeded) {

```

```

        Integer max = TriggerHandler.loopCountMap.get(handlerName).max;
        throw new TriggerHandlerException('Maximum loop count of ' + String.valueOf(max) + ' reached in
' + handlerName);
    }
}

// make sure this trigger should continue to run
@TestVisible
private Boolean validateRun() {
    if (!this.isTriggerExecuting || this.context == null) {
        throw new TriggerHandlerException('Trigger handler called outside of Trigger execution');
    }
    if (TriggerHandler.bypassedHandlers.contains(getHandlerName())) {
        return false;
    }
    return true;
}

@TestVisible
private String getHandlerName() {
    return String.valueOf(this).substring(0, String.valueOf(this).indexOf(':'));
}

/*****
 * context methods
 *****/

// context-specific methods for override
@TestVisible
protected virtual void beforeInsert() {}
@TestVisible
protected virtual void beforeUpdate() {}
@TestVisible
protected virtual void beforeDelete() {}
@TestVisible
protected virtual void afterInsert() {}
@TestVisible
protected virtual void afterUpdate() {}
@TestVisible
protected virtual void afterDelete() {}
@TestVisible
protected virtual void afterUndelete() {}

```



```

/*****
 * inner classes
 *****/

// inner class for managing the loop count per handler
@TestVisible
private class LoopCount {
    private Integer max;
    private Integer count;

    public LoopCount() {
        this.max = 5;
        this.count = 0;
    }

    public LoopCount(Integer max) {
        this.max = max;
        this.count = 0;
    }

    public Boolean increment() {
        this.count++;
        return this.exceeded();
    }

    public Boolean exceeded() {
        if (this.max < 0) return false;
        if (this.count > this.max) {
            return true;
        }
        return false;
    }

    public Integer getMax() {
        return this.max;
    }

    public Integer getCount() {
        return this.count;
    }

    public void setMax(Integer max) {

```

```

        this.max = max;
    }
}

// possible trigger contexts
@TestVisible
private enum TriggerContext {
    BEFORE_INSERT,
    BEFORE_UPDATE,
    BEFORE_DELETE,
    AFTER_INSERT,
    AFTER_UPDATE,
    AFTER_DELETE,
    AFTER_UNDELETE
}

// exception class
public class TriggerHandlerException extends Exception {}
}

```

○ Trigger Handler Test Class

```

@Test
private class TriggerHandler_Test {

    private static final String TRIGGER_CONTEXT_ERROR = 'Trigger handler called outside of Trigger
    execution';

    private static String lastMethodCalled;

    private static TriggerHandler_Test.TestHandler handler;

    static {
        handler = new TriggerHandler_Test.TestHandler();
        // override its internal trigger detection
        handler.isTriggerExecuting = true;
    }

    /*****
    * unit tests
    *****/

    // contexts tests
}

```

```
@isTest
static void testBeforeInsert() {
    beforeInsertMode();
    handler.run();
    System.assertEquals('beforeInsert', lastMethodCalled, 'last method should be beforeInsert');
}

@isTest
static void testBeforeUpdate() {
    beforeUpdateMode();
    handler.run();
    System.assertEquals('beforeUpdate', lastMethodCalled, 'last method should be beforeUpdate');
}

@isTest
static void testBeforeDelete() {
    beforeDeleteMode();
    handler.run();
    System.assertEquals('beforeDelete', lastMethodCalled, 'last method should be beforeDelete');
}

@isTest
static void testAfterInsert() {
    afterInsertMode();
    handler.run();
    System.assertEquals('afterInsert', lastMethodCalled, 'last method should be afterInsert');
}

@isTest
static void testAfterUpdate() {
    afterUpdateMode();
    handler.run();
    System.assertEquals('afterUpdate', lastMethodCalled, 'last method should be afterUpdate');
}

@isTest
static void testAfterDelete() {
    afterDeleteMode();
    handler.run();
    System.assertEquals('afterDelete', lastMethodCalled, 'last method should be afterDelete');
}
```

```
@isTest
static void testAfterUndelete() {
    afterUndeleteMode();
    handler.run();
    System.assertEquals('afterUndelete', lastMethodCalled, 'last method should be afterUndelete');
}
```

```
@isTest
static void testNonTriggerContext() {
    try{
        handler.run();
        System.assert(false, 'the handler ran but should have thrown');
    } catch(TriggerHandler.TriggerHandlerException te) {
        System.assertEquals(TRIGGER_CONTEXT_ERROR, te.getMessage(), 'the exception message should match');
    } catch(Exception e) {
        System.assert(false, 'the exception thrown was not expected: ' + e.getTypeName() + ': ' + e.getMessage());
    }
}
```

```
// test bypass api
```

```
@isTest
static void testBypassAPI() {
    afterUpdateMode();

    // test a bypass and run handler
    TriggerHandler.bypass('TestHandler');
    handler.run();
    System.assertEquals(null, lastMethodCalled, 'last method should be null when bypassed');
    System.assertEquals(true, TriggerHandler.isBypassed('TestHandler'), 'test handler should be bypassed');
    resetTest();
}
```

```
// clear that bypass and run handler
TriggerHandler.clearBypass('TestHandler');
handler.run();
System.assertEquals('afterUpdate', lastMethodCalled, 'last method called should be afterUpdate');
System.assertEquals(false, TriggerHandler.isBypassed('TestHandler'), 'test handler should be bypassed');
resetTest();
}
```

```
// test a re-bypass and run handler
TriggerHandler.bypass('TestHandler');
```

```

    handler.run();
    System.assertEquals(null, lastMethodCalled, 'last method should be null when bypassed');
    System.assertEquals(true, TriggerHandler.isBypassed('TestHandler'), 'test handler should be bypassed');
    resetTest();

    // clear all bypasses and run handler
    TriggerHandler.clearAllBypasses();
    handler.run();
    System.assertEquals('afterUpdate', lastMethodCalled, 'last method called should be afterUpdate');
    System.assertEquals(false, TriggerHandler.isBypassed('TestHandler'), 'test handler should be bypassed');
    resetTest();
}

// instance method tests

@Test
static void testLoopCount() {
    beforeInsertMode();

    // set the max loops to 2
    handler.setMaxLoopCount(2);

    // run the handler twice
    handler.run();
    handler.run();

    // clear the tests
    resetTest();

    try {
        // try running it. This should exceed the limit.
        handler.run();
        System.assert(false, 'the handler should throw on the 3rd run when maxloopcount is 3');
    } catch(TriggerHandler.TriggerHandlerException te) {
        // we're expecting to get here
        System.assertEquals(null, lastMethodCalled, 'last method should be null');
    } catch(Exception e) {
        System.assert(false, 'the exception thrown was not expected: ' + e.getTypeName() + ': ' +
e.getMessage());
    }

    // clear the tests
    resetTest();
}

```

```

// now clear the loop count
handler.clearMaxLoopCount();

try {
    // re-run the handler. We shouldn't throw now.
    handler.run();
    System.assertEquals('beforeInsert', lastMethodCalled, 'last method should be beforeInsert');
} catch (TriggerHandler.TriggerHandlerException te) {
    System.assert(false, 'running the handler after clearing the loop count should not throw');
} catch (Exception e) {
    System.assert(false, 'the exception thrown was not expected: ' + e.getTypeName() + ': ' +
e.getMessage());
}
}

@isTest
static void testLoopCountClass() {
    TriggerHandler.LoopCount lc = new TriggerHandler.LoopCount();
    System.assertEquals(5, lc.getMax(), 'max should be five on init');
    System.assertEquals(0, lc.getCount(), 'count should be zero on init');

    lc.increment();
    System.assertEquals(1, lc.getCount(), 'count should be 1');
    System.assertEquals(false, lc.exceeded(), 'should not be exceeded with count of 1');

    lc.increment();
    lc.increment();
    lc.increment();
    lc.increment();
    System.assertEquals(5, lc.getCount(), 'count should be 5');
    System.assertEquals(false, lc.exceeded(), 'should not be exceeded with count of 5');

    lc.increment();
    System.assertEquals(6, lc.getCount(), 'count should be 6');
    System.assertEquals(true, lc.exceeded(), 'should not be exceeded with count of 6');
}

// private method tests

@isTest
static void testGetHandlerName() {
    System.assertEquals('TestHandler', handler.getHandlerName(), 'handler name should match class

```

```

name');
}

// test virtual methods

@isTest
static void testVirtualMethods() {
    TriggerHandler h = new TriggerHandler();
    h.beforeInsert();
    h.beforeUpdate();
    h.beforeDelete();
    h.afterInsert();
    h.afterUpdate();
    h.afterDelete();
    h.afterUndelete();
}

/*****
 * testing utilities
 *****/

private static void resetTest() {
    lastMethodCalled = null;
}

// modes for testing

private static void beforeInsertMode() {
    handler.setTriggerContext('before insert', true);
}

private static void beforeUpdateMode() {
    handler.setTriggerContext('before update', true);
}

private static void beforeDeleteMode() {
    handler.setTriggerContext('before delete', true);
}

private static void afterInsertMode() {
    handler.setTriggerContext('after insert', true);
}

```

```
private static void afterUpdateMode() {
    handler.setTriggerContext('after update', true);
}

private static void afterDeleteMode() {
    handler.setTriggerContext('after delete', true);
}

private static void afterUndeleteMode() {
    handler.setTriggerContext('after undelete', true);
}

// test implementation of the TriggerHandler

private class TestHandler extends TriggerHandler {

    public override void beforeInsert() {
        TriggerHandler_Test.lastMethodCalled = 'beforeInsert';
    }

    public override void beforeUpdate() {
        TriggerHandler_Test.lastMethodCalled = 'beforeUpdate';
    }

    public override void beforeDelete() {
        TriggerHandler_Test.lastMethodCalled = 'beforeDelete';
    }

    public override void afterInsert() {
        TriggerHandler_Test.lastMethodCalled = 'afterInsert';
    }

    public override void afterUpdate() {
        TriggerHandler_Test.lastMethodCalled = 'afterUpdate';
    }

    public override void afterDelete() {
        TriggerHandler_Test.lastMethodCalled = 'afterDelete';
    }

    public override void afterUndelete() {
        TriggerHandler_Test.lastMethodCalled = 'afterUndelete';
    }
}
```



```
}  
  
}
```

References

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

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

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

<https://github.com/kevinohara80/sfdc-trigger-framework>

Deloitte.

5.5 Test Class

Best Practices

General:

- Test classes should be defined in separate files and use the `@isTest` annotation since an apex class defined with `@isTest` does not count towards code size limits.
- While only 75% of your Apex code must be covered by tests, ensure every use case of your application is covered, including Positive and Negative cases, as well as Bulk and Single records.
- Exercise bulk trigger functionality—use at least 20 records in your tests.
- Write comments stating not only what is supposed to be tested, but the assumptions the tester made about the data, the expected outcome, and so on.
- Use the `System.runAs` method to test your application in different user contexts.
- Test the classes in your application individually. Never test your entire application in a single test.
- Avoid hard coding of Id's anywhere in test Class.
- A test method should not be declared or implemented in a class that is not annotated with `@isTest`.

Testing Patterns / What to Test ?

- Test for Positive scenarios.
- Test for Negative scenarios.
- Check for right Permissions.
- Make sure testing covers catch blocks.

Test Data:

- Create the necessary data in test classes, so the tests do not have to rely on data in a particular organization. `TestSetup` or `TestDataFactory` class can be used for data creation.
- Create all test data before calling the `Test.startTest` method.

`Test.startTest` & `Test.stopTest`:

- Use the `startTest` and `stopTest` methods to validate how close the code is to reaching governor limits. Only the code that is targeted to test should be included in the `Test.startTest()` and `Test.stopTest()`.
- Any code that executes after the call to `startTest` and before `stopTest` is assigned a new set of governor limits.
- All asynchronous calls made after the `startTest` method are collected by the system. When `stopTest` is executed, all asynchronous processes are run synchronously.

Assertions:

- Have meaningful assertions using `System.assert()`, `System.assertEquals()` and `System.assertNotEquals()` methods.

- Make an assertion before the call to the method being tested is made.
- Assert to check if the output from the method being tested matches the expected result.

Best Practices Example

○ RunAs

```
@isTest
private class TestRunAs {
    public static testMethod void testRunAs() {
        // Setup test data
        // Create a unique UserName
        String uniqueUserName = 'standarduser' + DateTime.now().getTime() + '@testorg.com';
        // This code runs as the system user
        Profile p = [SELECT Id FROM Profile WHERE Name='Standard User'];
        User u = new User(Alias = 'standt', Email='standarduser@testorg.com',
            EmailEncodingKey='UTF-8', LastName='Testing', LanguageLocaleKey='en_US',
            LocaleSidKey='en_US', ProfileId = p.Id,
            TimeZoneSidKey='America/Los_Angeles',
            UserName=uniqueUserName);

        System.runAs(u) {
            // The following code runs as user 'u'
            System.debug('Current User: ' + UserInfo.getUserName());
            System.debug('Current Profile: ' + UserInfo.getProfileId());
        }
    }
}
```

○ Assertions

```
//Sample test class to indicate use of assertions
@isTest
private class HelloWorldTestClass {
    static testMethod void validateHelloWorld() {
        Book__c b = new Book__c(Name = 'Behind the Cloud', Price__c = 100);
        System.debug('Price before inserting new book: ' + b.Price__c);

        // Insert book
        insert b;

        // Retrieve the new book
        b = [SELECT Price__c FROM Book__c WHERE Id =: b.Id];
        System.debug('Price after trigger fired: ' + b.Price__c);
    }
}
```

```

        // Test that the trigger correctly updated the price
        System.assertEquals(90, b.Price__c);
    }
}

```

○ Load Test Data from Static Resources

```

/*****Load Test Data from Static Resources*****/
@isTest
private class DataUtil {
    static testmethod void testLoadData() {
        // Load the test accounts from the static resource
        List<sObject> ls = Test.loadData(Account.sObjectType, 'testAccounts');
        // Verify that all 3 test accounts were created
        System.assert(ls.size() == 3);

        // Get first test account
        Account a1 = (Account)ls[0];
        String acctName = a1.Name;
        System.debug(acctName);

        // Perform some testing using the test records
    }
}

```

Testing Pattern

- A test method must ensure that an exception occurs to cover the try-catch blocks in the actual code as given in the below example.
- Test Classes which test for emails sent from apex might fail if the Deliverability is set to “No Access” or “System Email only”. To avoid this, surround the lines which call the method within try-catch block and use assert statements to check the exception being thrown.
- Criteria Based Sharing rules do not get executed in the Test context. Share Records would have to be created to share with a group or user.
 - This can be done in a Trigger with the usage of Test.isRunningTest() where we provide the condition/Criteria and create Sharing Records According to the Rule.

Testing Pattern Example

○ Test Class for Outbound Email

```

/**Email Utility Class**/
public with sharing class EmailUtility
{

```

```

public static void sendPlainTextEmail(List<String> toAddresses, String subject, String body)
{
    try {
        Messaging.SingleEmailMessage mail = new Messaging.SingleEmailMessage();
        mail.setToAddresses(toAddresses);
        mail.setSubject(subject);
        mail.setPlainTextBody(body);
        Messaging.sendEmail(new Messaging.SingleEmailMessage[] { mail });
    } catch(Exception e) {
        throw e;
    }
}

/**Test Class**/
@Test
private class EmailUtilityTest
{
    @isTest
    static void testEmailUtility()
    {
        Test.StartTest();
        EmailUtility.sendPlainTextEmail(new List<String>{'test@test.com'}, 'Unit Test X', 'Unit Test');
        Integer invocations = Limits.getEmailInvocations();
        Test.stopTest();
        System.assertEquals(1, invocations, 'An email has not been sent');
    }
}

```

○ Testing Try Catch

```

/**Testing try catch**/
public class LeadCreation {
    public Lead objLead;
    public String lastName;
    public LeadCreation() {

    }
    public PageReference newLead() {
        objLead = new Lead(Company = 'Test', LastName = lastName, Status = 'Open - Not Contacted');
        try {
            insert objLead;
            PageReference pg = new PageReference('/') + objLead.Id);
            pg.setRedirect(true);
            return pg;
        }
    }
}

```

```

        } catch(DMLEException e) {
            throw e;
        }
    }
}
/**Test class to try catch***/
@isTest
private class LeadCreationTest {

    @isTest static void leadTest() {
        LeadCreation obj = new LeadCreation();
        try {
            obj.newLead();
        } catch(DMLEException e) {
            system.assertEquals(e.getMessage().contains("Last Name"));
        }
        obj.lastName = 'Testing';
        obj.newLead();
    }

}

```

○ Testing VF Controller

```

/**Test class sample for VF controller***/
@isTest
private class MyControllerTest {
    static testMethod void myUnitTest() {
        // create a new page reference, set it as the current page,
        // and use the ApexPages.currentPage() method to put the values
        PageReference PageRef = Page.MyPage;
        Test.setCurrentPage(PageRef);
        ApexPages.currentPage().getParameters().put('urlparm', 'theValue');

        // load the controller and proceed with testing
        MyController controller = new MyController();
    }
}

```

○ Testing Batch Apex

```

/**Batch Apex Class**/
global class AccountUpdate implements Database.Batchable {

```

```

global Database.QueryLocator start(Database.BatchableContext BC){
    return Database.getQueryLocator('Select Id, Name From Account');
}

global void execute(Database.BatchableContext BC, List scope){
    for(Account a: scope){
        a.Name += ' Updated';
    }
    update scope;
}

global void finish(Database.BatchableContext BC){}
}

/**Test Class for Batch Apex**/
private class accListountUpdate {

    static testmethod void test() {
        // Create test accounts to be updated
        // by the batch job.
        Account[] accList = new List < Account > ();
        for (Integer i = 0; i < 200; i++) {
            Account m = new Account(Name = 'Account ' + i);
            accList.add(m);
        }
        insert accList;
        Test.startTest();
        AccountUpdate c = new AccountUpdate();
        Database.executeBatch(c);
        Test.stopTest();
        // Verify accounts updated
        Account[] accUpdatedList = [SELECT Id, Name FROM Account];
        System.assert(accUpdatedList[0].Name.Contains('Updated'));
    }
}

```

Annotations

@isTest :

- This define classes and methods that only contain code used for testing your application.
- Classes defined with the @isTest annotation don't count against your organization limit of 6 MB for all Apex code.

@testSetup:

- Annotation used for creating common test records that are available for all test methods in the class.
- Records that are created in a test setup method are available to all test methods in the test class and are rolled back at the end of test class execution.
- There can be only one test setup method per test class.

@TestVisible:

- To allow test methods to access private or protected members of another class outside the test class.
- This annotation doesn't change the visibility of members if accessed by non-test classes.

@IsTest(SeeAllData=true) Not recommended :

- Grant test classes and individual test methods access to all data in the organization. The access includes pre-existing data that the test didn't create.
- @IsTest(SeeAllData=true) and @IsTest(isParallel=true) annotations cannot be used together on the same Apex method.

@IsTest(OnInstall=true) :

- To specify which Apex tests are executed during package installation.
- This annotation is used for tests in managed or unmanaged packages.
- Only test methods with this annotation, or methods that are part of a test class that has this annotation, are executed during package installation.

@IsTest(isParallel=true) :

- To indicate test classes that can run in parallel.
- This annotation makes the execution of test classes more efficient because more tests can be run in parallel.

Annotations Example

○ @IsTest

```
@IsTest(SeeAllData=true) /** Here it fetches data from org, not a recommended approach, just to show the use **/
```

```
public class TestDataAccessClass {
```

```
    // This test accesses an existing account.
```

```
    // It also creates and accesses a new test account.
```

```
    static testmethod void myTestMethod1() {
```

```
        // Query an existing account in the organization.
```

```
        Account a = [SELECT Id, Name FROM Account WHERE Name='Acme' LIMIT 1];
```

```
        System.assert(a != null);
```

```
    }

    // Create a test account based on the queried account.
```



```

    Account testAccount = a.clone();
    testAccount.Name = 'Acme Test';
    insert testAccount;

    // Query the test account that was inserted.
    Account testAccount2 = [SELECT Id, Name FROM Account
                           WHERE Name='Acme Test' LIMIT 1];
    System.assert(testAccount2 != null);
}

// Like the previous method, this test method can also access all data
// because the containing class is annotated with @isTest(SeeAllData=true).
@isTest static void myTestMethod2() {
    // Can access all data in the organization.
}

}

//Noncompliant Code Example:-
// A test class not declared with @isTest annotation
public with sharing SomeClass { // class without test class annotation
    ...
    @isTest
    public void someTestMethod() {...} // this isn't a test class, so it shouldn't have a test method
}

```

○ @testSetup

```

@isTest
private class CommonTestSetup {

    @testSetup
    static void setup() {
        // Create common test accounts
        List < Account > testAccts = new List < Account > ();
        for (Integer i = 0; i < 2; i++) {
            testAccts.add(new Account(Name = 'TestAcct' + i));
        }
        insert testAccts;
    }

    @isTest

```

```

static void testMethod1() {
    // Get the first test account by using a SOQL query
    Account acct = [SELECT Id FROM Account WHERE Name = 'TestAcct0' LIMIT 1];
    // Modify first account
    acct.Phone = '555-1212';
    // This update is local to this test method only.
    update acct;
    // Delete second account
    Account acct2 = [SELECT Id FROM Account WHERE Name = 'TestAcct1' LIMIT 1];
    // This deletion is local to this test method only.
    delete acct2;
    // Perform some testing
}

@Test
static void testMethod2() {
    // The changes made by testMethod1() are rolled back and
    // are not visible to this test method.
    // Get the first account by using a SOQL query
    Account acct = [SELECT Phone FROM Account WHERE Name = 'TestAcct0' LIMIT 1];
    // Verify that test account created by test setup method is unaltered.
    System.assertEquals(null, acct.Phone);
    // Get the second account by using a SOQL query
    Account acct2 = [SELECT Id FROM Account WHERE Name = 'TestAcct1' LIMIT 1];
    // Verify test account created by test setup method is unaltered.
    System.assertNotEquals(null, acct2);
    // Perform some testing
}
}

```

○ @TestVisible

```

public class TestVisibleExample {
    // Private member variable
    @TestVisible
    private static Integer recordNumber = 1;

    // Private method
    @TestVisible
    private static void updateRecord(String name) {
        // Do something
    }
}

```

○ Test Utility

```

/**Common utility class**/
@isTest
public class TestDataFactory {
    public static void createTestRecords(Integer numAccts, Integer numContactsPerAcct) {
        List<Account> accts = new List<Account>();
        for(Integer i=0;i<numAccts;i++) {
            Account a = new Account(Name='TestAccount' + i);
            accts.add(a);
        }
        insert accts;
        List<Contact> cons = new List<Contact>();
        for (Integer j=0;j<numAccts;j++) {
            Account acct = accts[j];
            // For each account just inserted, add contacts
            for (Integer k=numContactsPerAcct*j;k<numContactsPerAcct*(j+1);k++) {
                cons.add(new Contact(firstname='Test'+k,
                                    lastname='Test'+k,
                                    AccountId=acct.Id));
            }
        }
        // Insert all contacts for all accounts
        insert cons;
    }
}

/**Test class implementing common utility**/
@isTest
private class MyTestClass {
    static testmethod void test1() {
        TestDataFactory.createTestRecords(5,3);
        // Run some tests
    }
}

```

Testing Callouts

HTTP Callouts:

- HTTP callouts can be tested by instructing Apex to generate mock responses in tests, using `Test.setMock`.
- Provide an implementation for the `HttpCalloutMock` interface to specify the response sent in the `respond` method, which the Apex runtime calls to send a response for a callout.
- The class that implements the `HttpCalloutMock` interface can be either global or public.

- This class can be annotated with `@isTest` since it will be used only in test context.

Webservice:

- Apex provides the built-in `WebServiceMock` interface and the `Test.setMock` method. Use `WebServiceMock` and `Test.setMock` to receive fake responses in the test methods.
- Provide an implementation for the `WebServiceMock` interface to specify the response sent in the `respond` method, which the Apex runtime calls to send a response every time a `WebServiceCallout.invoke` is called.
- The class implementing the `WebServiceMock` interface can be either global or public.

StaticResourceCalloutMock:

- `StaticResourceCalloutMock` class allows you to store the response of a Callout in a Static resource.
- Store the response JSON in a text file.
- Create an instance of `StaticResourceCalloutMock` and set the static resource to use for the response body, along with other response properties, like the status code and content type.

MultiStaticResourceCalloutMock:

- `MultiStaticResourceCalloutMock` class is used to test scenarios where multiple callouts need to be tested in a single transaction.
- Store all of the JSON responses in separate text files.
- Upload the files as static resources.

Testing Callouts (Code)

○ HTTP Callout

```
/**Actual class implementing callout**/
public class CalloutClass {
    public static HttpResponse getInfoFromExternalService() {
        HttpRequest req = new HttpRequest();
        req.setEndpoint('http://example.com/example/test');
        req.setMethod('GET');
        Http h = new Http();
        HttpResponse res = h.send(req);
        return res;
    }
}

/**Class Implementing HttpCalloutMock interface**/
@isTest
global class MockHttpResponseGenerator implements HttpCalloutMock {
    // Implement this interface method
    global HTTPResponse respond(HTTPRequest req) {
        // Optionally, only send a mock response for a specific endpoint
    }
}
```

```

        // and method.
        System.assertEquals('http://example.com/example/test', req.getEndpoint());
        System.assertEquals('GET', req.getMethod());

        // Create a fake response
        HttpResponse res = new HttpResponse();
        res.setHeader('Content-Type', 'application/json');
        res.setBody('{"example":"test"}');
        res.setStatusCode(200);
        return res;
    }
}

/**Test class**/
@isTest
private class CalloutClassTest {
    @isTest static void testCallout() {
        // Set mock callout class
        Test.setMock(HttpCalloutMock.class, new MockHttpResponseGenerator());

        // Call method to test.
        // This causes a fake response to be sent
        // from the class that implements HttpCalloutMock.
        HttpResponse res = CalloutClass.getInfoFromExternalService();

        // Verify response received contains fake values
        String contentType = res.getHeader('Content-Type');
        System.assert(contentType == 'application/json');
        String actualValue = res.getBody();
        String expectedValue = '{"example":"test"}';
        System.assertEquals(actualValue, expectedValue);
        System.assertEquals(200, res.getStatusCode());
    }
}

```

○ SOAP Callout

```

/**Webservice Mock**/
@isTest
global class WebServiceMockImpl implements WebServiceMock {
    global void doInvoke(
        Object stub,
        Object request,
        Map<String, Object> response,
        String endpoint,

```

```

        String soapAction,
        String requestName,
        String responseNS,
        String responseName,
        String responseType) {
    docSample.EchoStringResponse_element respElement =
        new docSample.EchoStringResponse_element();
    respElement.EchoStringResult = 'Mock response';
    response.put('response_x', respElement);
}
}
/**Actual Apex Class**/
public class WebSvcCallout {
    public static String callEchoString(String input) {
        docSample.DocSamplePort sample = new docSample.DocSamplePort();
        sample.endpoint_x = 'http://example.com/example/test';

        // This invokes the EchoString method in the generated class
        String echo = sample.EchoString(input);

        return echo;
    }
}
/**Corresponding test class**/
@isTest
private class WebSvcCalloutTest {
    @isTest static void testEchoString() {
        // This causes a fake response to be generated
        Test.setMock(WebServiceMock.class, new WebServiceMockImpl());

        // Call the method that invokes a callout
        String output = WebSvcCallout.callEchoString('Hello World!');

        // Verify that a fake result is returned
        System.assertEquals('Mock response', output);
    }
}

```

○ StaticResourceCalloutMock

```

/** Test class example for StaticResourceCalloutMock**/
@isTest
private class CalloutStaticClassTest {
    @isTest static void testCalloutWithStaticResources() {

```

```

// Use StaticResourceCalloutMock built-in class to
// specify fake response and include response body
// in a static resource.
StaticResourceCalloutMock mock = new StaticResourceCalloutMock();
mock.setStaticResource('mockResponse');
mock.setStatusCode(200);
mock.setHeader('Content-Type', 'application/json');

// Set the mock callout mode
Test.setMock(HttpCalloutMock.class, mock);

// Call the method that performs the callout
HttpResponse res = CalloutStaticClass.getInfoFromExternalService(
    'http://example.com/example/test');

// Verify response received contains values returned by
// the mock response.
// This is the content of the static resource.
System.assertEquals('{\"hah\":\"this is a mock class\"}', res.getBody());
System.assertEquals(200, res.getStatusCode());
System.assertEquals('application/json', res.getHeader('Content-Type'));
}
}

```

○ MultiStaticResourceCalloutMock

```

@isTest
private class CalloutMultiStaticClassTest {
    @isTest static void testCalloutWithMultipleStaticResources() {
        // Use MultiStaticResourceCalloutMock to
        // specify fake response for a certain endpoint and
        // include response body in a static resource.
        MultiStaticResourceCalloutMock multimock = new MultiStaticResourceCalloutMock();
        multimock.setStaticResource(
            'http://example.com/example/test', 'mockResponse');
        multimock.setStaticResource(
            'http://example.com/example/sfdc', 'mockResponse2');
        multimock.setStatusCode(200);
        multimock.setHeader('Content-Type', 'application/json');

        // Set the mock callout mode
        Test.setMock(HttpCalloutMock.class, multimock);

        // Call the method for the first endpoint
    }
}

```

```

    HTTPResponse res = CalloutMultiStaticClass.getInfoFromExternalService(
        'http://example.com/example/test');
    // Verify response received
    System.assertEquals('{"hah":"this is a mock class"}', res.getBody());

    // Call the method for the second endpoint
    HTTPResponse res2 = CalloutMultiStaticClass.getInfoFromExternalService(
        'http://example.com/example/sfdc');
    // Verify response received
    System.assertEquals('{"hah":"this is a mock class"}', res2.getBody());
}
}

```

Stub API

- Stubbing is the process of telling Salesforce instead of executing a piece of business logic when unit tests are run, run another piece of logic instead.
- A mocking framework created by a stubbing API has the following advantages.
- Complex classes which call other layers in the application requires lots of test data setup. You end up having to test other areas of the application as part of your tests, so they aren't really unit tests.
- Different scenarios to be tested are difficult to recreate either due to complex test data setup or very difficult to produce.
- Speed. The stubbing API allows you to avoid creating test data in the database, which ultimately is way faster.
- Avoid having to use Test.isRunningTest() statements within your production code.

Mock Implementation With Stub API

○ Actual Apex Code

```

// Code to be tested
public class DateFormatter {
    // Method to test
    public String getFormattedDate(DateHelper helper) {
        return 'Today's date is ' + helper.getTodaysDate();
    }
}

//Helper class
public class DateHelper {
    // Method to stub
    public String getTodaysDate() {
        return Date.today().format();
    }
}

```



```

}

// Code to invoke the method
DateFormatter df = new DateFormatter();
DateHelper dh = new DateHelper();
String dateStr = df.getFormattedDate(dh);

```

○ Stub Provider Interface

```

@Test
public class MockProvider implements System.StubProvider {

    public Object handleMethodCall(Object stubbedObject, String stubbedMethodName,
        Type returnType, List<Type> listOfParamTypes, List<String> listOfParamNames,
        List<Object> listOfArgs) {

        // The following debug statements show an example of logging
        // the invocation of a mocked method.

        // You can use the method name and return type to determine which method was called.
        System.debug('Name of stubbed method: ' + stubbedMethodName);
        System.debug('Return type of stubbed method: ' + returnType.getName());

        // You can also use the parameter names and types to determine which method
        // was called.
        for (integer i = 0; i < listOfParamNames.size(); i++) {
            System.debug('parameter name: ' + listOfParamNames.get(i));
            System.debug(' parameter type: ' + listOfParamTypes.get(i).getName());
        }

        // This is a very simple mock provider that returns a hard-coded value
        // based on the return type of the invoked.
        if (returnType.getName() == 'String')
            return '8/8/2016';
        else
            return null;
    }
}

```

○ Stub Util

```

// The following utility class returns a stub object that you can use as a mock.
public class MockUtil {
    private MockUtil(){}
}

```

```
public static MockProvider getInstance() {  
    return new MockProvider();  
}  
  
public static Object createMock(Type typeToMock) {  
    // Invoke the stub API and pass it our mock provider to create a  
    // mock class of typeToMock.  
    return Test.createStub(typeToMock, MockUtil.getInstance());  
}  
}
```

○ Test Class using Mock

```
@isTest  
public class DateFormatterTest {  
    @isTest  
    public static void testGetFormattedDate() {  
        // Create a mock version of the DateHelper class.  
        DateHelper mockDH = (DateHelper)MockUtil.createMock(DateHelper.class);  
        DateFormatter df = new DateFormatter();  
  
        // Use the mocked object in the test.  
        System.assertEquals('Today's date is 8/8/2016', df.getFormattedDate(mockDH));  
    }  
}
```

References

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

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

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

5.6 Apex Controller

Best Practices in VF Context

Unless you are working on a classic/custom platform application move from VF to Lightning. Use Lightning components for developing the same functionality.

Enforce Sharing Rules in Controllers

Like other Apex classes, all controllers and controller extensions run in system mode. Typically, we want a controller or controller extension to respect a user's organization-wide defaults, role hierarchy, and sharing rules. We can do that by using the 'with' sharing keywords in the class definition.

Note: If a controller extension extends a standard controller, the logic from the standard controller does not execute in system mode. Instead, it executes in user mode, in which the profile-based permissions, field-level security, and sharing rules of the current user apply.

Evaluate Controller Constructors before Setter Methods

Do not depend on a setter method being evaluated before a constructor. For example, in the following component, the component's controller depends on the setter for `selectedValue` being called before the constructor method.

Sharing

Refer Sharing under Apex Classes.

Picklist Values

Avoid hardcoding picklist values. Use field describe result, `getPicklistValues` method to get the `Schema.PicklistEntry` object.

View State Limits

Refer Limits under Apex Classes.

Wrapper Classes

- Use wrapper classes if data associated from different objects is to be displayed on the page.
- These classes should be written at the end of the controller class.
- Give proper getter and setter for variables and add a constructor to give default values to the variables.
- Implement Comparable interface if sorting is required (This is optional).

Constructor

- Set defaults for variables in the constructor, if setter methods are not being used.

- Try to avoid multiple SOQL in the constructor, since these will be run in every page load.
- No DMLs are allowed within a constructor. It's not recommended to execute DML statements on page load. If it's a must have and requirements cannot be modified, try using the action VF page attribute.

URL Parameters

- Always use PageReference in the controller instead of using JavaScript for page redirection.
- getParameters returns a map of the query string parameters for the PageReference; both POST and GET parameters are included. The key string contains the name of the parameter, while the value string contains the value of the parameter.
- Avoid fetching redirection url from returnUrl parameter in page Url.
 - String returnUrl = ApexPages.currentPage().getParameters().get('returnUrl');
 - PageReference pageRef = new PageReference(returnUrl);
- Parameter keys are case-insensitive.
- This map can be modified and remains in scope for the PageReference object. For instance, you could do:
 - PageReference.getParameters().put('id', myID);
 - ApexPages.currentPage().getParameters().get('myParamName')

Best Practices in VF Context (Code)

○ Best Practices in VF Context (Code)

// For example, in the following component, the component's controller depends on the setter
// for selectedValue being called before the constructor method.

```
<apex:component controller="CustCompController">
  <apex:attribute name="value" description=""
    type="String" required="true"
    assignTo="{!selectedValue}">
  </apex:attribute>
  //...
  //...
</apex:component>
```

// Since the constructor is called before the setter, selectedValue will always be null
// when the constructor is called. Thus, EditMode will never be set to true.

```
public class CustCompController{
  // Constructor method
  public CustCompController() {
    if (selectedValue != null) {
      EditMode = true;
    }
  }
  private Boolean EditMode = false;
  // Setter method
```

```
public String selectedValue { get;set; }
}
```

○ Avoid fetching redirection URL from Page URL parameter

// Non-Compliant Code:-

```
public class DraftArticle {
    String returnUrlxyz ='dfdfsad';
    public void someMethod() {
        String returnUrl = Apexpages.currentpage().getparameters().get('returnUrl');
        PageReference pageRef = new PageReference(returnUrl);
    }
    public void method() {
        String startUrl = Site.login(userName, password,
        ApexPages.currentPage().getParameters().get('startURL'));
        PageReference pageRef = new PageReference(startUrl);
    }
}
```

// Compliant Code:-

```
public class DraftArticle {
    public void someMethod() {
        if ( unsafeLocation == 'something ) { //Good: this is one possibility for a safer redirect.
            return new PageReference('/somewhere')
        } else {
            return new pageReference('/somewhere-else')
        }
    }
}
```

5.7 Visualforce

Components

The four basic components of Visualforce are Pages, Components, Controllers and Resources.

- **Pages:** Pages are the basic creative building blocks for application designers. Similar to a standard Web page, a Visualforce page uses HTML to specify the appearance of the application's interface, with the option of using other Web technologies such as CSS, AJAX, and Adobe Flex for additional flexibility.
- **Components:** Components are the key for reusing common interface elements and for binding both standard and custom elements to data.
- **Controllers:** Controllers are the basic functional building blocks that control the application's logic. Implemented in Apex code, controllers provide the underlying business rules for the interface, as well as the "connective tissue" between the application's page presentation and the underlying data.
- **Resources:** JavaScript and CSS should be included as Static Resources allowing the browser to cache them.

Best Practices

View State:

- To maintain the state of Visualforce pages across requests, salesforce includes the state for the component, field values and controller state in a hidden encrypted form element. This is called view state. You can have upto 135KB of view state (Governor limit). The lesser the view state, the better will be the performance.
- By reducing your view state size, your pages can load quicker and stall less often.
- You can monitor view state performance through the View State tab in the development mode footer
- Use the transient keyword in your Apex controllers for variables that aren't essential for maintaining state and aren't necessary during page refreshes.
- If you notice that a large percentage of your view state comes from objects used in controllers or controller extensions, consider refining your SOQL calls to return only the data that's relevant to the Visualforce page.
- If your view state is affected by a large component tree, try reducing the number of components your page depends on.
- If you are returning large volumes of data to be displayed. Consider pagination and filters with a small set of records per each page. avoid querying additional fields that aren't required.

Controlling the Data Size:

- Visualforce pages can't exceed the 15 MB standard response limit, but even smaller page sizes affect load time.
- To minimize load times,
 - Use filters in the query to return small volumes of data to be displayed.

- use 'with sharing' keyword to retrieve only records the user can access
- Use pagination to display 20-30 records at a time.

Lazy Loading:

- Lazy Loading is a technique where the page loads its essentials first and delays the rest.
- To lazy load parts of a Visualforce page:
- Use the rerender attribute on Visualforce components to update the component without updating the entire page.
- Use JavaScript Remoting to call functions in your controller through JavaScript, and to retrieve ancillary or static data.
- Create a custom component to show and hide data according to user actions.

Optimize Component Hierarchies:

- Limit the nesting of Custom Components. Vast hierarchies increase server-side management and processing time because Visualforce maintains context throughout the entire request, and traversing component hierarchies consumes time and resources.
- Deeply nested components incur the highest processing costs. Vast hierarchies also put pages at risk of hitting hit heap size governor limits.

Optimizing CSS and JavaScript:

- Use static resources to serve JavaScript files, as well as images, CSS, and other non-changing files.
- JavaScript and other assets served this way benefit from the caching and content distribution network (CDN) built into Salesforce.
- Use minified javascript/CSS files in Static Resource.
- Put scripts at the bottom of the page to load them after the HTML
- Consider moving any JavaScript outside of the `<apex:includeScript>` tag and placing it into a `<script>` tag right before your closing `<apex:page>` tag. The `<apex:includeScript>` tag places JavaScript right before the closing `<head>` element; thus, Visualforce attempts to load the JavaScript before any other content on the page. However, you should only move JavaScript to the bottom of the page if you're certain it doesn't have any adverse effects to your page. For example, JavaScript code snippets requiring `document.write` or event handlers should remain in the `<head>` element.

Multiple Concurrent Requests:

Concurrent requests are long-running tasks that could block other pending tasks. To reduce these delays:

- Action methods used by `<apex:actionPoller>` should be lightweight. It's a best practice to avoid performing DML, external service calls, and other resource-intensive operations in action methods called by an `<apex:actionPoller>`. Carefully consider the effect of your action method being called repeatedly by an `<apex:actionPoller>` at the interval you specify, especially if it's used on a page that will be widely distributed, or open continuously.

- Increase the time interval for calling Apex from your Visualforce page. For example, when using the `<apex:actionPoller>` component, you could adjust the `interval` attribute to 30 seconds instead of 15.
- Move non-essential logic to an asynchronous code block using Ajax.
- Use the `immediate` Attribute where necessary. Visualforce components configured, with the `immediate="true"` attribute do not bypass the normal component cycle of processing or validation steps. In this case, the values are submitted as requests. Functional problems occur when the component behavior includes more than basic navigation functionality. Use of this attribute is recommended only for processes where an immediate cancel of the action is required.
 - e.g. `<apex:CommandLink action="{!cancelApplication}" value="Cancel" styleClass="btn" id="btnCancel" immediate="true">`

Accessing DOM Ids:

- Use `$Component` global variable to reference generated DOM ID.
- e.g. `{!$Component.itemId}` to access DOM ID at the same level in the page.
- Specify complete path while accessing in the hierarchy -> `$Component grandparentId.parentId.itemId`

Static Resource in VF page

- To reference a stand-alone file, use `$Resource.<resource_name>` as a merge field, where `<resource_name>` is the name you specified when you uploaded the resource. For example:
 - `<apex:image url="{!$Resource.TestImage}" width="50" height="50"/>` OR
 - `<apex:includeScript value="{!$Resource.MyJavascriptFile}"/>`
 - `<script type="text/javascript" src="{!URLFOR ($Resource.StreamingAPI, '/js/jquery-3.3.1.min.js')}"/></script>`
- To reference a file in an archive, use the `URLFOR` function. Specify the static resource name that you provided when you uploaded the archive with the first parameter and the path to the desired file within the archive with the second. For example:
 - `<apex:image url="{!URLFOR($Resource.TestZip, 'images/Bluehills.jpg')}" width="50" height="50"/>` OR
 - `<apex:includeScript value="{!URLFOR($Resource.LibraryJS, '/base/subdir/file.js')}"/>`
- You can use relative paths in files in static resource archives to refer to other content within the archive. For example, in your CSS file, named `styles.css`, you have the following style:
 - `table { background-image: url('img/testimage.gif')}`
- Displaying the Content of a Static Resource with the Action Attribute On `<Apex: Page>`
 - We can use action attribute on a `<apex: page>` component to redirect from a Visualforce page to a static resource. This functionality allows us to add rich, custom help to your Visualforce pages. The static resource reference is wrapped in a `URLFOR` function. Without that, the page does not redirect properly.
 - This redirect is not limited to PDF files. You can also redirect a page to the content of any static resource. For example, you can create a static resource that includes an entire help system composed of many HTML files mixed with JavaScript, images, and other multimedia files. As long as there is a single entry point, the redirect works.
 - For example:

- Create a zip file that includes your help content.
- Upload the zip file as a static resource named customhelpsystem.

```
<apex: page sidebar="false" showHeader="false" standardStylesheets="false"
action="!URLFOR($Resource.customhelpsystem, 'index.htm')"></apex: page>
```

VF remoting

JavaScript remoting is a tool that front-end developers can use to make an AJAX request from a Visualforce page directly to an Apex controller. JavaScript remoting allows you to run asynchronous actions by decoupling the page from the controller and to perform tasks on the page without having to reload the entire page.

Why go for Javascript Remoting?

- To perform tasks without reloading the entire page.
- It will help reduce view state issues.
- Pass data to the controller only which is needed.
- Can help alleviate view state issues while still executing in the context of the user viewing the page.
- Most efficient way of calling the controller and passing data in from the page, because you can ensure that you're passing only the data that you need each time that you make a call.
- To make a VF page work in mobile page and use 3rd party Javascript library.

apex:actionFunction

- The <apex:actionFunction> component also lets you call controller action methods through JavaScript.
- In general, <apex:actionFunction> is easier to use and requires less code, while JavaScript remoting offers more flexibility. Here are some specific differences between the two.
- The <apex:actionFunction> tag:
 - lets you specify rerender targets
 - submits the form
 - doesn't require you to write any JavaScript
- JavaScript remoting:
 - lets you pass parameters
 - provides a callback
 - requires you to write some JavaScript

Parameters { buffer: true, escape: true, timeout: 30000 }

Name	Data Type	Description
		Whether to group requests executed close to each other in time into a single request. The default is <code>true</code> .

buffer Boolean

JavaScript remoting optimizes requests that are executed close to each other in time and groups the calls into a single request. This buffering improve the efficiency of the overall request-and-response cycle, but sometimes it's useful to ensure all requests execute independently.

escape Boolean Whether to escape the Apex method's response. The default is `true`.

timeout Integer The timeout for the request, in milliseconds. The default is 30,000 (30 seconds). The maximum is 120,000 (120 seconds, or 2 minutes).

Limitations

- By default, the response of the remote call must return within 30 seconds, after which the call will time out. If your request needs longer to complete, configure a longer timeout, up to 120 seconds.
- The request, including headers, has a maximum size of 4 MB.
- The response of the remote call has a maximum size of 15 MB.

VF Remoting (Code)

○ VF Remoting (Code)

//Here's a basic sample demonstrating how to use JavaScript remoting in your Visualforce pages.
global with sharing class PHX_accountRemoter {

```
public String accountName { get; set; }
public static Account account { get; set; }
public PHX_accountRemoter() { } // empty constructor

@RemoteAction
global static Account getAccount(String accountName) {
    account = [SELECT Id, Name, Phone, Type, NumberOfEmployees
                FROM Account WHERE Name = :accountName];
    return account;
}
}
```

○ VF Remoting (VF page)

```
<!-- To make use of the remote method, create a Visualforce page that looks like this -->
<apex:page controller="PHX_accountRemoter">
    <script type="text/javascript">
        function getRemoteAccount() {
            var accountName = document.getElementById('acctSearch').value;
            // Another format - AccountRemoter.getAccount(accountName, function(result, event) {
            Visualforce.remoting.Manager.invokeAction(
                '{$RemoteAction.PHX_accountRemoter.getAccount}',
```

```

accountName,
function(result, event){
    if (event.status) {
        // Get DOM IDs for HTML and Visualforce elements like this
        document.getElementById('remoteAcctId').innerHTML = result.Id
        document.getElementById(
            "{$Component.block.blockSection.secondItem.acctNumEmployees}"
        ).innerHTML = result.NumberOfEmployees;
    } else if (event.type === 'exception') {
        document.getElementById("responseErrors").innerHTML =
            event.message + "<br/>\n<pre>" + event.where + "</pre>";
    } else {
        document.getElementById("responseErrors").innerHTML = event.message;
    }
}, // refer above for the param details
{ buffer: true, escape: true, timeout: 30000 }
);
}
</script>

```

```

<input id="acctSearch" type="text"/>
<button onclick="getRemoteAccount()">Get Account</button>
<div id="responseErrors"></div>

```

```

<apex:pageBlock id="block">
    <apex:pageBlockSection id="blockSection" columns="2">
        <apex:pageBlockSectionItem id="firstItem">
            <span id="remoteAcctId"/>
        </apex:pageBlockSectionItem>
        <apex:pageBlockSectionItem id="secondItem">
            <apex:outputText id="acctNumEmployees"/>
        </apex:pageBlockSectionItem>
    </apex:pageBlockSection>
</apex:pageBlock>
</apex:page>

```

○ apex:actionFunction (VF page)

```

<!-- actionFunction example -->
<apex:page controller="PHX_sampleController">
    <apex:form>
        <!-- Define the JavaScript function actionSampleMethod-->
        <apex:actionFunction name="actionSampleMethod" action="{!actionSampleMethod}" rerender="out"
            status="myStatus"/>
    </apex:form>
</apex:page>

```

```

</apex:form>

<apex:outputPanel id="out">
  <apex:outputText value="Hello " />
  <apex:actionStatus startText="requesting..." id="myStatus">
    <apex:facet name="stop">{!username}</apex:facet>
  </apex:actionStatus>
</apex:outputPanel>

<!-- Call the sayHello JavaScript function using a script element-->
<script>window.setTimeout(actionSampleMethod,2000)</script>

<p><apex:outputText value="Clicked? {!state}" id="showstate" /></p>

<!-- Add the onclick event listener to a panel. When clicked, the panel triggers
the methodOneInJavascript actionFunction with a param -->
<apex:outputPanel onclick="methodOneInJavascript('Yes!')" styleClass="btn">
  Click Me
</apex:outputPanel>
<apex:form>

<apex:actionFunction action="{!methodOne}" name="methodOneInJavascript" rerender="showstate">
  <apex:param name="firstParam" assignTo="{!state}" value="" />
</apex:actionFunction>
</apex:form>
</apex:page>

```

VF Page References

- `$CurrentPage` - A global merge field type to use when referencing the current Visualforce page or page request.
- Use this global variable in a Visualforce page to reference the current page name (`$CurrentPage.Name`) or the URL of the current page (`$CurrentPage.URL`). Use `$CurrentPage.parameters.parameterName` to reference page request parameters and values, where `parameterName` is the request parameter being referenced. `parameterName` is case sensitive.

VF- Current Page References (Code)

```

<apex:page standardController="Account">
  <apex:pageBlock title="Hello {!$User.FirstName}!">
    You belong to the {!account.name} account.<br/>
    You're also a nice person.
  </apex:pageBlock>

```

```
<apex:detail subject="{!account}" relatedList="false"/>
<apex:relatedList list="OpenActivities"
  subject="{!$CurrentPage.parameters.relatedId}"/>
</apex:page>
```

Render, Re-render and RenderAs

- **Rendered**
 - A Visualforce component in a Visualforce page can be displayed or hidden by using the rendered attribute. Rendered is bound to a boolean variable in the controller which can be switched between true and false making the Visualforce component displayed or hidden depending on boolean value.
- **ReRender**
 - ReRender is used to refresh a particular section of the Visualforce page. We have to just mention the id of the page section (in the ReRender attribute) that needs to be refreshed.
- **RenderAs**
 - This is used with page component and renders the page in the specified format. Currently, only pdf format is supported.

References

Trailheads related to VF page:

https://trailhead.salesforce.com/en/content/learn/modules/visualforce_fundamentals

<https://trailhead.salesforce.com/en/content/learn/projects/quickstart-visualforce/vf-qs-1>

https://trailhead.salesforce.com/en/content/learn/modules/visualforce_fundamentals/visualforce_standard_contro

https://trailhead.salesforce.com/en/content/learn/modules/visualforce_fundamentals/visualforce_standard_list_co

https://trailhead.salesforce.com/en/content/learn/modules/visualforce_fundamentals/visualforce_forms

Visualforce page best practices:-

https://developer.salesforce.com/docs/atlas.en-us.pages.meta/pages/pages_best_practices_performance.htm

Communicating b/w lightning component and vf page

<https://developer.salesforce.com/blogs/developer-relations/2017/01/lightning-visualforce-communication.html>

5.8 Custom Setting & Metadata

Using Custom Settings

Custom settings are similar to custom objects and enable application developers to create custom sets of data, as well as create and associate custom data for an organization, profile, or a specific user. All custom settings data is exposed in the application cache, which enables efficient access without the cost of repeated queries to the database. This data can then be used in formula fields, validation rules, Apex, and the SOAP API. The following examples illustrate how you can use custom settings:

- A shipping application requires users to fill in the country codes for international deliveries. By creating a list setting of all country codes, users have quick access to this data without needing to query the database.
- An application calculates and tracks compensation for its sales reps, but commission percentages are based on seniority. By creating a hierarchy setting, the administrator can associate a different commission percentage for each profile in the sales organization. Within the application, one formula field can then be used to correctly calculate compensation for all users; the personalized settings at the profile level insert the correct commission percentage.
- An application displays a map of account locations, the best route to take, and traffic conditions. This information is useful for sales reps, but account executives only want to see account locations. By creating a hierarchy setting with custom checkbox fields for route and traffic, you can enable this data for just the “Sales Rep” profile.

Custom Metadata Types

Custom metadata types are similar to custom settings they are also used to store metadata. Your reusable functionality reads your custom metadata and uses it to produce customized application behavior.

Custom metadata types enable you to create your own setup objects whose records are metadata rather than data. These are typically used to define application configurations that need to be migrated from one environment to another, or packaged and installed.

Rather than building apps from data records in custom objects or custom settings, you can create custom metadata types and add metadata records, with all the manageability that comes with metadata: package, deploy and upgrade. Querying custom metadata records don't count against SOQL limits. For example, you can use custom metadata types for the following.

- Mappings—Create associations between different objects, such as a custom metadata type that assigns cities, states, or provinces to particular regions in a country.
- Business rules—Combine configuration records with custom functionality. Use custom metadata types along with some Apex code to route payments to the correct endpoint.

- Master data—Let's say that your org uses a standard accounting app. Create a custom metadata type that defines custom charges, like duties and VAT rates. If you include this type as part of an extension package, subscriber orgs can reference the master data.
- Whitelists—Manage lists, such as approved donors and pre-approved vendors.
- Secrets—Store information, like API keys, in your protected custom metadata types within a package.

Deloitte.

5.9 Async Processing

Asynchronous Apex

Asynchronous Apex is used to run processes in a separate thread, at a later time. An asynchronous process is a task "in the background" without the user having to wait for the task to finish.

You'll typically use Asynchronous Apex for callouts to external systems, operations that require higher limits, and certain time. The key benefits of asynchronous processing include:

User efficiency

- With asynchronous processing the user can get on with their work, the processing can be done in the background and results at their convenience.
- Controlled frequency: Schedule the process to run daily or at a specific time of interval

Scalability

- By allowing some features of the platform to execute when resources become available at some point in time, they can be managed and scaled quickly. This allows the platform to handle more jobs using parallel processing.

Higher Limits

- Asynchronous processes are started in a new thread, with higher governor and execution limits.

Type	Overview	Comments
Future Methods	Run in their own thread, and do not start execution until resources are available.	Web services
Batch Apex	Run large jobs that would exceed normal processing limits and do not start execution until resources are available.	Data cleanup
Queueable Apex	Similar to future methods, but provide additional job chaining and allow more complex data types to be used. It does not start execution until resources are available.	Performance optimization
Scheduled Apex	Schedule Apex to run at a specified time and do not start execution until resources are available.	Daily cleanup

Best Practice & Tips

- Avoid future method calls inside a for loop.
- Try avoiding multiple future method call for same set of data as it may lead to record locking.
- Try keeping batch size of Batch Apex configurable via custom setting, labels etc.
- Always enclose future method call within 'if(!System.isBatch() && !System.isFuture())' to avoid calling future method from async processes.

- Use extreme care if you are planning to invoke a batch job from a trigger. You must be able to guarantee that the trigger does not add more batch jobs than the limit. To mitigate this problem refer to below link: https://help.salesforce.com/articleView?id=000182449&language=en_US&type=1

Capture Async Job details in a common location.

It is recommended to maintain a common location to track all the async jobs in an organization to assess the loads on the system. For example in a common ORG track all the async job details on an excel sheet with basic details as below. This is critical for planning long-running batch executions so that the system resources are used in an optimum manner.

- Name
- Project
- Type
- What is it used for
- Expected Frequency of Invocation
- Data Volume
- Scheduled Time
- Requirement Number
- Average Execution Duration

Testing Async

- To test async methods call the class containing the method in a startTest(), stopTest() code block. All asynchronous calls made after the startTest method are collected by the system. When stopTest is executed, all asynchronous processes are run synchronously.
- Asynchronous calls, such as @future or executeBatch, called in a startTest, stopTest block, do not count against your limits for the number of queued jobs.
- While testing batch apex make sure test data count is always less than the batch size so that batch executes only a single time.
- Assert statements should be put after test.stop() block to ensure batch execution is finished.
- Use CRON expression to test scheduled apex class.

Limitations of Async

Governor Limits

- You're limited to 50 calls per Apex invocation, and there's an additional limit on the number of calls in a 24-hour period.
- The maximum number of invocations per a 24-hour period is 250,000 or the number of user licenses in your organization multiplied by 200, whichever is greater.
- Maximum number of batch Apex jobs in the Apex flex queue that are in Holding status: 100
- Maximum number of batch Apex jobs queued or active concurrently: 5
- Maximum scheduled Apex jobs at one time: 100

- Maximum number of records returned for a Batch Apex query in Database.QueryLocator: 50 million
- Maximum size for Batch Apex: 2000
- Total number of callouts allowed from the start, execute, and finish: 10 callouts each.
- Maximum number of jobs to the queue with System.enqueueJob in a single transaction: 50
- Below are the key differences between Sync and Async Limits. For more details refer to
 - https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_gov_limits.htm

Description	Synchronous Limit	Asynchronous Limit
Total number of SOQL queries issued	100	200
Total heap size	6 MB	12 MB
Maximum CPU time on the Salesforce servers	10,000 milliseconds	60,000 milliseconds

Future method Limitations

- The specified parameters must be primitive data types, arrays of primitive data types, or collections of primitive data types; future methods can't take objects as arguments.
- Future methods won't necessarily execute in the same order they are called.
- Cannot call another future method from a future method.

Batch Apex Limitations

- Cannot call Future method or another batch apex from execute method.

Scheduled Apex Limitations

- Synchronous Web service callouts are not supported in schedulable Apex.

Async Job Object (AsyncApexJob)

This is the common object that holds the details of the async jobs running in the ORG. Below is a sample SOQL to reach the async job details -

```
[SELECT Id, MethodName, JobItemsProcessed, ApexClassId, CompletedDate, CreatedById, CreatedDate,
NumberOfErrors, JobType, Status, ExtendedStatus, TotalJobItems FROM AsyncApexJob WHERE
CreatedDate=today AND Status in ('Processing','Holding','Preparing','Queued') AND ApexClassId in
:ApexClassMap.keySet()];
```

Below are the high-level details of the different jobs logged in the table -

1. Future method represents an asynchronous apex method annotated as an @future method. Future methods are used to execute long-running operations such as callouts to external Web services or any operation requiring its own thread and to run in its own time
2. Queueable job represents an asynchronous Apex class using the Queueable interface via implements Queueable. Using the Queueable interface, allows users to manage and monitor asynchronous Apex jobs

3. ScheduledApex job represents an asynchronous Apex class using the Schedulable interface via implements Schedulable. Using Scheduled Apex allows for the invocation of Apex classes to run at specific times when resources are available
4. BatchApex job represents an asynchronous Apex class using the Batchable interface via implements Database.Batchable. Using Batch Apex allows for the asynchronous processing of a long running process on a large data volume by breaking the job into manageable chunks to be processed separately
5. SharingRecalculation job represents a Batch Apex class (class that implements Batchable interface via implements Database.Batchable) used to recalculate the Apex managed sharing for a specific custom object. 'SharingRecalculation' can be associated with a custom object via the custom object detail page and initiated via 'Recalculate Apex Sharing'
6. BatchApexWorker Used internally by Salesforce. For each 'AsyncApexJob' record of type 'BatchApex', Apex creates for internal use an 'AsyncApexJob' record of type 'BatchApexWorker' per 10,000 records to be processed. When querying for all 'AsyncApexJob' records, we recommend that you filter out records of type 'BatchApexWorker' using the 'JobType' field to avoid returning more than one record for each 'AsyncApexJob' record of type 'BatchApex.'
7. TestRequest Every time Apex test(s) are run asynchronously from the UI or from Apex code, the system creates a parent 'AsyncApexJob' with 'JobType' set as 'TestRequest'. Customers can also execute a SOQL query to check the status of an overall test request
8. TestWorker job is used only internally by Salesforce and is not saved in the database
9. ApexToken job is used for Flex Queue enabled organizations. From Winter '16 onwards, the flex queue is enabled by default for all organizations. If an Apex code block invokes Database.executeBatch(), instead of accepting the job directly into Salesforce's queueing system, an 'ApexToken' Job is submitted to the queue. For any batch job that a user submits, a token job is created internally. However, if the maximum number of queued batch jobs in the batch job queue has been reached, the new batch job is placed in a 'Holding' state in the flex queue. 'ApexToken' is internal mechanism for managing system resources. When a token job gets picked up by the system for execution, it's an indication that the system can take in more jobs from the flex queue. Note: Every organization has only a limited number of 'ApexToken' jobs. This token mechanism allows organizations to submit more batch jobs (maximum is a 100), however only some of them could be running in parallel on the Salesforce platform.

5.9.1 Batch

Apex Batch Best Practices

Limits in Batch Classes

1. Up to five queued or active batch jobs are allowed for Apex and maximum number of batch executions is 250,000 per 24 hours
2. A maximum of 50 million records can be returned in the Database.QueryLocator object. If more than 50 million records are returned, the batch job is immediately terminated and marked as Failed
3. The start, execute, and finish methods can implement up to 100 callouts each
4. Only one batch Apex job's start method can run at a time in an organization. Batch jobs that haven't started yet remain in the queue until they're started. Note that this limit doesn't cause any batch job to fail and execute methods of batch Apex jobs still run in parallel if more than one job is running.
5. Limitations related to batch size are -
 1. Minimum Batch size - 1
 2. Maximum Batch size - 2000
 3. Default Batch size - 200
6. Total number of "SOQL queries issued" limit in one batch execution is 200.

Database.QueryLocator/Iterable

Start method takes the input of Database.QueryLocator object or an iterable that contains the records or objects passed to the job.

1. When you're using a simple query (SELECT) to generate the scope of objects in the batch job, use the Database.QueryLocator object. If you use a QueryLocator object, the governor limit for the total number of records retrieved by SOQL queries is bypassed (up to 50 million). Scope parameter of Database.executeBatch can have a maximum value of 2,000 for Database.QueryLocator. If set to a higher value, Salesforce chunks the records returned by the QueryLocator into smaller batches of up to 2,000 records.
2. Use the iterable to create a complex scope for the batch job. You can also use the iterable to create your own custom process for iterating through the list. The governor limit for the total number of records retrieved by SOQL queries is still enforced for this. If the start method returns an iterable, the scope parameter value has no upper limit; however, if you use a very high number, you may run into other limits.
3. If no size is specified with the optional scope parameter of Database.executeBatch, Salesforce chunks the records returned by the start method into batches of 200 and then passes each batch to the execute method. Apex governor limits are reset for each execution of execute.

Future and Batch Class

1. Methods declared as future aren't allowed in classes that implement the Database.Batchable interface.
2. Methods declared as future can't be called from a batch Apex class.

3. 1 Queueable invocation is allowed within the execute method of a Batch Class.

Database.ExecuteBatch

1. Use extreme care if you are planning to invoke a batch job from a trigger. You must be able to guarantee that the trigger will not add more batch jobs than the five that are allowed. In particular, consider API bulk updates, import wizards, mass record changes through the user interface, and all cases where more than one record can be updated at a time.
2. When you call Database.executeBatch, Salesforce only places the job in the queue at the scheduled time. Actual execution may be delayed based on service availability.
3. When testing your batch Apex, you can test only one execution of the execute method. You can use the scope parameter of the executeBatch method to limit the number of records passed into the execute method to ensure that you aren't running into governor limits.
4. The executeBatch method starts an asynchronous process.
5. You cannot call the Database.executeBatch method from within any batch Apex method.

Database.Stateful and Chaining

Use Database.Stateful with the class definition if you want to share variables or data across job transactions. Otherwise, all instance variables are reset to their initial state at the start of each transaction. Example you can use Stateful to get all the account Ids inserted in the batch to send as input to a second batch process that calls an external system to create the related contacts in Salesforce. The second batch would be called from the Finally block.

Chain a batch job to start a job after another one finishes and when your job requires batch processing, such as when processing large data volumes. Otherwise, if batch processing isn't needed, consider using Queueable Apex.

AsyncApexJob Record

1. Each batch Apex invocation creates an AsyncApexJob record. Use the ID of this record to construct a SOQL query to retrieve the job's status, number of errors, progress, and submitter. For more information about the AsyncApexJob object, see AsyncApexJob in the Web Services API Developer's Guide.
2. For each 10,000 AsyncApexJob records, Apex creates one additional AsyncApexJob record of type BatchApexWorker for internal use. When querying for all AsyncApexJob records, we recommend that you filter out records of type BatchApexWorker using the JobType field. Otherwise, the query will return one more record for every 10,000 AsyncApexJob records. For more information about the AsyncApexJob object, see AsyncApexJob in the Web Services API Developer's Guide.

Other Additional Points

1. All methods in the class must be defined as global.
2. To use a callout in batch Apex, specify Database.AllowsCallouts in the class definition.
3. For a sharing recalculation, we recommend that the execute method delete and then re-create all Apex managed sharing for the records in the batch. This ensures the sharing is accurate and complete.

4. When a batch Apex job is run, email notifications are sent either to the user who submitted the batch job, or, if the code is included in a managed package and the subscribing organization is running the batch job, the email is sent to the recipient listed in the ApexException Notification Recipient field.
5. Each method execution uses the standard governor limits anonymous block, Visualforce controller, or WSDL method.
6. You cannot use the getContent and getContentAsPDF PageReference methods in a batch job.
7. In the event of a catastrophic failure such as a service outage, any operations in progress are marked as Failed. You should run the batch job again to correct any errors.

Apex Batch Best Practices (Code)

○ QueryLocator Example

```
/**
Simple Batch logic that takes input of query string, object, field and value to set
Query Locator will by pass governor limits to get upto 50 million records
*/
global class UpdateObjectFields implements Database.Batchable < sObject >, Database.Stateful,
Database.AllowsCallouts{
    global final String strQuery;
    global final String strEntity;
    global final String strField;
    global final String strValue;
    // constructor to initialize variables
    global UpdateObjectFields(String q, String e, String f, String v) {
        strQuery = q;
        strEntity = e;
        strField = f;
        strValue = v;
    }
    // start method to query
    global Database.QueryLocator start(Database.BatchableContext BC) {
        return Database.getQueryLocator(query);
    }

    global void execute(Database.BatchableContext BC, List < sObject > scope) {
        // loop through the records to set the field value for update
        for (Sobject s: scope) {
            s.put(strField , strValue );
        }
        update scope;
    }

    global void finish(Database.BatchableContext BC) {
```

```

    }

}

```

○ Custom Iterable Example

```

/**
Simple batch logic that implements the iterable interface
Batch class calls the iterable interface in start to get the list of accounts to process
*/

```

```

global class batchClass implements Database.batchable<Account>{
    global Iterable<Account> start(Database.batchableContext info){
        return new example();
    }
    global void execute(Database.batchableContext info, List<Account> scope){
        List<Account> accsToUpdate = new List<Account>();
        for(Account a : scope){
            a.Name = 'true';
            a.NumberOfEmployees = 69;
            accsToUpdate.add(a);
        }
        update accsToUpdate;
    }
    global void finish(Database.batchableContext info){
    }
}

```

```

global class example implements iterable<Account>{
    global Iterator<Account> iterator(){
        return new CustomIterable();
    }
}

```

```

/**
If you do not want to use a custom iterator with a list, but instead want to create your own data structure,
you can use the Iterable interface to generate the data structure. The iterator method must be declared as
global or public. It creates a reference to the iterator that you can then use to traverse the data structure.

```

Iterable Methods

hasNext Boolean Returns true if there is another item in the collection being traversed, false otherwise

next Any type Returns the next item in the collection.

```

*/

```

```
global class CustomIterable implements Iterator<Account>{
    List<Account> accs {get; set;}
    Integer i {get; set;}

    public CustomIterable(){
        accs = [SELECT Id, Name, NumberOfEmployees FROM Account WHERE Name = 'false'];
        i = 0;
    }

    global boolean hasNext(){
        if(i >= accs.size()) {
            return false;
        } else {
            return true;
        }
    }

    global Account next(){
        // 8 is an arbitrary
        // constant in this example
        // that represents the
        // maximum size of the list.
        if(i == 8){return null;}
        i++;
        return accs[i-1];
    }
}
```


5.9.2 Queueable

Queueable Apex Best Practices

1. Use Queueable interface over Future methods as it allows you to pass custom inputs, object instance instead of only primitive data types. It also returns the Job Id for tracking
2. Queueable supports chaining. No limit is enforced on the depth of chained jobs, which means that you can chain one job to another job and repeat this.
3. You can add only one job from an executing job with System.enqueueJob, means that only child job can exist for parent queueable job.
4. The execution of a queued job counts once against the shared limit for asynchronous Apex method executions.
5. You can add up to 50 jobs to the queue with System.enqueueJob in a single transaction.
6. Limits.getQueueableJobs() helps to check how many queueable jobs have been added in one transaction.
7. For Developer Edition and Trial organizations, the maximum stack depth for chained jobs is 5.

Queueable Apex Best Practices (Code)

```
// to call the queueable class , create instance and enqueue it
ID jobId = System.enqueueJob(new AsyncExecutionExample());

// to search the async job object
AsyncApexJob jobInfo = [SELECT Status,NumberOfErrors FROM AsyncApexJob WHERE Id=:jobId];

public class AsyncExecutionExample implements Queueable {
    public void execute(QueueableContext context) {
        // Your processing logic here

        // Chain this job to next job by submitting the next job
        System.enqueueJob(new SecondJob());
    }
}
```

5.9.3 Future

Future method Best Practices

Use Queueable Interface over Future methods where ever possible. Refer to Queueable section for more details.

Important Points

1. Methods with the future annotation must be static methods
2. They can only return a void type
3. The specified parameters must be primitive data types, arrays of primitive data types, or collections of primitive data types
4. Methods with the future annotation cannot take sObjects or objects as arguments.

Best Practices

1. If using Web service callouts, try to bundle all callouts together from the same future method, rather than using a separate future method for each callout.
2. Conduct thorough testing at scale. Test that a trigger enqueueing the @future calls is able to handle a trigger collection of 200 records. This helps determine if delays may occur given the design at current and future volumes.
3. Consider using Batch Apex instead of future methods to process a large number of records asynchronously. This is more efficient than creating a future request for each record.
4. Methods with the future annotation must be static methods, and can only return a void type.
5. The specified parameters must be primitive data types, arrays of primitive data types, or collections of primitive data types; future methods can't take objects as arguments.
6. Future methods won't necessarily execute in the same order they are called. In addition, it's possible that two future methods could run concurrently, which could result in record locking if the two methods were updating the same record.
7. Future methods can't be used in Visualforce controllers in getMethodName(), setMethodName(), nor in the constructor.
8. You can't call a future method from a future method.
9. The getContent() and getContentAsPDF() methods can't be used in methods with the future annotation.

Future method Best Practices (Code)

```
global class FutureMethodExample
{
    @future(callout=true)
    public static void getStockQuotes(String acctName)
    {
        // Perform a callout to an external service
    }
}
```

```
}
```

```
}
```

Deloitte.

5.9.4 Schedulable

Schedulable Apex Best Practices

1. Salesforce schedules the class for execution at the specified time. Actual execution may be delayed based on service availability.
2. Use extreme care if you're planning to schedule a class from a trigger. You must be able to guarantee that the trigger won't add more scheduled classes than the limit. In particular, consider API bulk updates, import wizards, mass record changes through the user interface, and all cases where more than one record can be updated at a time.
3. Though it's possible to do additional processing in the execute method, we recommend that all processing take place in a separate class.
4. Synchronous Web service callouts are not supported from scheduled Apex. To be able to make callouts, make an asynchronous callout by placing the callout in a method annotated with `@future(callout=true)` and call this method from scheduled Apex. However, if your scheduled Apex executes a batch job, callouts are supported by the batch class. See Using Batch Apex.
5. Apex jobs scheduled to run during a Salesforce service maintenance downtime will be scheduled to run after the service comes back up, when system resources become available. If a scheduled Apex job was running when downtime occurred, the job is rolled back and scheduled again after the service comes back up. Note that after major service upgrades, there might be longer delays than usual for starting scheduled Apex jobs because of system usage spikes.

Scheduling the Job

Jobs can be scheduled by using Setup Menu/Apex

1. From Setup, enter Apex Classes in the Quick Find box, select Apex Classes, and then click Schedule Apex. For more details refer to https://help.salesforce.com/articleView?id=code_schedule_batch_apex.htm&type=5
2. Using the System.Schedule method. The method takes three arguments: a name for the job, an expression used to represent the time and date the job is scheduled to run, and the name of the class.
 1. Cron Expression Format - Seconds Minutes Hours Day_of_month Month Day_of_week
Optional_year
 2. For more details refer to https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_scheduler.htm
 3. The following are some examples of how to use the expression.

Expression	Description
0 0 13 * * ?	Class runs every day at 1 PM.
0 0 22 ? * 6L	Class runs the last Friday of every month at 10 PM.
0 0 10 ? * MON-FRI	Class runs Monday through Friday at 10 AM.

0 0 20 * * ? 2010	Class runs every day at 8 PM during the year 2010.
-------------------	--

Schedulable Apex Best Practices (Code)

```
//In the following example, the class implements the Schedulable interface.
//The class is scheduled to run at 8 AM, on the 13th of February.
DailyLeadProcessor dlp = new DailyLeadProcessor();
String sch = '0 0 8 13 2 ?';
String jobId = system.schedule('Lead Job', sch, dlp);

// Schedulable Class example to process leads
global class DailyLeadProcessor implements Schedulable {
    global void execute(SchedulableContext ctx) {
        List<Lead> leads = [Select Id, LeadSource from Lead where LeadSource=null LIMIT 200];
        for(lead Id: leads){
            Id.LeadSource = 'Dreamforce';
        }
        update leads;
    }
}
```

5.10 Error Handling

Exceptions in Apex

An exception denotes an error that disrupts the normal flow of code execution. You can use Apex built-in exceptions or create custom exceptions. All exceptions have common methods. Use throw statements to generate exceptions while try, catch and finally statements are used to gracefully recover from exceptions.

All exceptions support built-in methods for returning the error message and exception type. Salesforce has about twenty different exceptions that can be thrown when there is a problem. Also, there are many ways to handle errors in your code for preventing exceptions, including using assertions like System.assert calls, or returning error codes or Boolean values.

The advantage of using exceptions is that they simplify error handling. Exceptions bubble up from the called method to the caller, as many levels as necessary, until a catch statement is found to handle the error. This bubbling up relieves you from writing error handling code in each of your methods. Also, by using finally statements, you can recover from exceptions, like resetting variables and deleting data. Finally blocks will be executed each time even in case of exception or no exception.

What Happens When an Exception Occurs?

When an exception occurs (and not handled), code execution halts. Any DML operations that were processed before the exception are rolled back and aren't committed to the database. Exceptions get logged in debug logs. For unhandled exceptions, that is, exceptions that the code doesn't catch, Salesforce sends an email that includes the exception information. The end user sees an error message in the Salesforce user interface if there is an exception while performing an operation on the UI.

Unhandled Exception Emails

When unhandled Apex exceptions occur, emails are sent that include the Apex stack trace, exception message, and the customer's org and user ID. No other data is returned with the report. Unhandled exception emails are sent by default to the developer specified in the LastModifiedBy field on the failing class or trigger. In addition, you can have emails sent to users of your Salesforce org and to arbitrary email addresses. These email recipients can also receive process or flow error emails. To set up these email notifications, from Setup, enter Apex Exception Email in the Quick Find box, then select Apex Exception Email. The entered email addresses then apply to all managed packages in the customer's org. You can also configure Apex exception emails using the Tooling API object ApexEmailNotification.

Unhandled Exceptions in the User Interface

If an end user runs into an exception that occurred in Apex code while using the standard user interface, an error message appears. The error message includes text similar to the notification shown here. So it is important to handle exceptions, so that users do not see such errors on the screen. We can also handle

exceptions to display a friendly message which is mostly done in case of showing a custom validation message which cannot be taken care in a validation rule.

The screenshot shows a Salesforce 'New Merchandise' form. At the top, there's a 'Merchandise Edit' header with a 'New Merchandise' title and a 'Help for this Page' link. Below the header, there are three buttons: 'Save', 'Save & New', and 'Cancel'. A red error message is displayed in the center: 'Error: Invalid Data. Review all error messages below to correct your data. Apex trigger myMerchandiseTrigger caused an unexpected exception, contact your administrator: myMerchandiseTrigger: execution of BeforeInsert caused by: System.NullPointerException: Attempt to de-reference a null object: Trigger.myMerchandiseTrigger: line 3, column 1'. Below the error message, there's an 'Information' section with a legend indicating that a red vertical bar represents 'Required Information'. The form fields are: 'Merchandise Name' (Erasers), 'Description' (White erasers), 'Price' (1.50), and 'Total Inventory' (120). The 'Owner' field is set to 'Test User'.

Exceptions that Can't be Handled

Some special types of built-in exceptions can't be caught. Those exceptions are associated with critical situations in the Lightning Platform. These situations require the abortion of code execution and don't allow for execution to resume through exception handling. When exceptions are uncatchable, catch blocks, as well as finally blocks if any, aren't executed. Below are few examples,

- **System.LimitException:** One such exception is the limit exception that the runtime throws if a governor limit has been exceeded, such as when the maximum number of SOQL queries issued has been exceeded, or when Apex CPU Time Limit is exceeded.
- **Failure of System.assert:** Other examples are exceptions thrown when assertion statements fail or license exceptions.

Handle Exceptions

Try, catch and finally method should be used in all Apex classes and trigger to catch exceptions when executing apex code.

There should never be any unhandled exceptions. Exceptions in apex and Visualforce should be reformatted in a relevant error message and no stack traces should ever be presented to an end user. The central logging object can be used to capture the stack trace on an exception.

Since you can't throw built-in Apex exceptions but can only catch them, you can create custom exceptions to throw in your methods. That way, you can also specify detailed error messages and have more custom error handling in your catch blocks.

To create your custom exception class, extend the built-in Exception class and make sure your class name ends with the word Exception. Append extends Exception after your class declaration as follows.

Custom Exception

With custom exceptions, you can throw and catch built-in exceptions in your methods. Please see Error Handling For Apex section for more details.

Deloitte.

5.10.1 For Apex

Apex Error Handling

- Every good code has error handling mechanisms. As a bare minimum have try-catch blocks.
- Keep the try-catch block as small as possible. As larger the block, it is more likely to suppress errors.
- Whenever possible try to catch a specific type of exception instead of a generic exception.
- Check if value for field is changed using trigger.new & Old, only then perform the business logic.
- In Salesforce, one can have error handling in trigger/class.
- Exception Handling Behavior:
 - Exception handling behavior will be different based on what apex context that you use.
 - Triggers: you would want to report the errors to the users if it is a breaking exception.
 - Future Methods and batch jobs: you would want to log the errors in a custom object and email if it is blocker or critical.
 - Custom Controllers: you would want to report the errors to the users if it is a breaking exception
- Trigger Errors:
 - Triggers should be written in such a manner that they either fire or do not fire due to the business rule. Any action that could result in an error, such as DML statements, should be handled in a class method.
 - Triggers can be used to prevent DML operations from occurring by calling the addError() method on a record or field. When used on Trigger.new records in insert and update triggers, and on Trigger.old records in delete triggers, the custom error message is displayed in the application interface and logged.
 - Be careful when you add error message in triggers using addError method. The business logic or code written after the addError is still executed.
- Class Errors:
 - Using the guide documentation, any exception can be handled in a try/catch/finally block. These blocks should be used to avoid any hard system failures. For try/catch blocks, either specifically, place only one triggering error condition in each block or ensure multiple catch blocks are present to catch each error that is thrown. Note that this must happen in order of the most specific to the least specific error, otherwise the error may be trapped too early. Avoid leaving finally block empty.
- Error Display: Errors can be displayed 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 message class for collecting errors. An example of it being used 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 Exception.

- SOQL queries, DML statements, collection sizes or uninitialized variables left unchecked can lead to exceptions. These exceptions or checks have to be handled carefully. The scenarios that can throw exceptions are as follows:
 - Iterating over or referencing an element of empty collection
 - SOQL queries: Any issue with SOQL queries, such as assigning a query that returns no records or more than one record to a single sObject variable.
- Have an error logging framework for your application. Have a custom object called 'Error Log' which will have relevant fields to capture the error/exception details (stack trace, etc). Create Error logs for any exception that occurs so that system admins can monitor and take care of them. Ensure to preserve stack trace.
- Use the System.SavePoint() and Database.rollback() functions. The key advantage of doing a rollback is that we can set the database back to a known state and not necessarily worry about whether an admin adds a validation rule or not.
- Async processing vs Synchronous processing: Try catch can be used both in synchronous and asynchronous operations. But for synchronous transactions, try catch should throw error message on the UI to let users know what went wrong whereas for asynchronous transaction (batch, future) it should be used to catch exceptions and save/log error message in an object to be monitored by Admin later.

Handling DML Exceptions

Any issue related to DML statements can lead to exceptions. For example, an 'insert' statement missing can throw exception. 'DmlException' is the in-built support method used for handling these kinds of exceptions.

Use the methods provided as part of Database Class to handle these exceptions. For details on the methods provided refer to

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

A suitable example for DML exception handling is given below.

Other additional methods available for DmlException are:

Name	Arguments	Return Type	Description
getDmlFieldNames	Integer i	String []	Returns the names of the field or fields that caused the error described by the ith failed row.
getDmlFields	Integer i	Schema.sObjectField []	Returns the field token or tokens for the field or fields that caused the error described by the ith failed row. For more information on field tokens, see Dynamic Apex.
getDmlId	Integer i	String	Returns the ID of the failed record that caused the error described by the ith failed row.

getDmlIndex	Integer i	Integer	Returns the original row position ith failed row.
getDmlMessage	Integer i	String	Returns the user message for the failed row.
getDmlStatusCode	Integer i	String	Deprecated. Use getDmlType ins Returns the Apex failure code for ith failed row.
getDmlType	Integer i	System.StatusCode	Returns the value of the System.StatusCode enum.
getNumDml		Integer	Returns the number of failed row: DML exceptions

Avoiding catching NullPointerException's in catch blocks used for error handling.

Handling Exception Example

○ VF Page addMessage

```
/**
In this scenario we observe three error messages
1. Field level error message from trigger
2. Record level error message from trigger
3. Error message from the controller.
**/
trigger PHX_accountTrigger on Account (after update) {
    for (Account acct: Trigger.new) {
        // This will throw error and display it on the field Name at UI
        acct.Name.addError('Error added to Name on trigger');
        // This error will be shown on the top of the record as it is at record level.
        acct.addError('Error added to account on trigger');
    }
}

// If UI is a VF Page with Controller.
public with sharing class AccountController {
    private final Account account;
    public AccountController() {
        account = [SELECT Id, Name, AccountNumber FROM Account][0];
    }

    public Account getAccount() {
        return account;
    }
}
```

```

public PageReference save() {
    try {
        update account;
    } catch (Exception e) {
        ApexPages.addMessage(new ApexPages.Message(ApexPages.severity.ERROR,
            'Error added with ApexPages.addMessages'));
    }
    return null;
}
}

```

○ DML Exception

```

Account[] accts = new Account[]{new Account(billingcity = 'San Jose')};
try {
    database.insert(accts);
} catch (System.DmlException e) {
    for (Integer i = 0; i < e.getNumDml(); i++) {
        // Process exception here
        System.debug(e.getDmlMessage(i));
    }
}
}

```

Custom Exception

There might be a situation that needs to use an exception class which is not normally provided by the platform they are working on. Salesforce platform provides inheritance features and hence we have the ability to inherit the standard exception class. APEX which has a standard exception class can be extended to create our own exceptions.

With custom exceptions, you can throw and catch built-in exceptions in your methods. Custom exceptions enable you to specify detailed error messages and have more custom error handling in your catch blocks. Exceptions can be top-level classes, that is, they can have member variables, methods and constructors, they can implement interfaces, and so on.

To create your custom exception class, extend the built-in Exception class and make sure your class name ends with the word Exception, such as “MyException” or “PurchaseException”. All exception classes extend the system-defined base class Exception, and therefore, inherits all common Exception methods.

Exception Class:-

The exception class is easy to extend and does not have any abstract methods, hence there is no need to create any method in the extended class. However, it is always a good practice to override at least the `getMessage()` function so that when we use those classes, we will have an idea about why the exception is occurring.

Exception Class gives these following methods to use, by overriding them:

`getLineNumber()` – Returns the line number of the origination of exception.

`getStackTraceString()` – Returns the trace string of the origination of exception.

`getTypeName()` – Returns the type of the exception like, DML exception, list exception or Math exception.

Custom Exception (Code)

○ Custom Exception Class

```
public class PHX_CustomException extends Exception{
    public Decimal d;

    public PHX_CustomException (String message, Decimal d)
    {
        this(message);
        this.d = d;
    }
}
```

○ Custom Exception Class Usage

```
public class PHX_CustomExceptinCls{
    public static void validateMethod(){
        try
        {
            throw new PHX_CustomException('this is my message',3.1415);
        }
        catch(Exception e)
        {
            system.debug(e.getMessage());
            PHX_CustomException me = (PHX_CustomException) e;
            system.debug(me.getMessage());
            system.debug(me.d);
        }
    }
}
```

Transaction Control

All requests are delimited by the trigger, class method, Web Service, Visualforce page or anonymous block that executes the Apex code. If the entire request completes successfully, all changes are committed to the database. For example, suppose a Visualforce page called an Apex controller, which in turn called an additional Apex class. Only when all the Apex code has finished running and the Visualforce page has finished running, are the changes committed to the database. If the request does not complete successfully, all database changes are rolled back.

Sometimes during the processing of records, your business rules require that partial work (already executed DML statements) be “rolled back” so that the processing can continue in another direction. Apex gives you the ability to generate a savepoint, that is, a point in the request that specifies the state of the database at that time. Any DML statement that occurs after the savepoint can be discarded, and the database can be restored to the same condition it was in at the time you generated the savepoint.

As an extension to exception handling or designing some particular use cases, we can make use of Database.savepoint and Rollback. Consider a scenario where you had to insert both parent and child records and its mandatory to insert both else nothing. If this is happening in a try block, then the lines of code and DMLs will be executed till the error comes. The error causing line would not execute. In that case, imagine, if parents are inserted, but child insertion gave a problem, we will either right a code block in catch block to delete parents or simply make use of a rollback like given in the example.

Transaction Control (Save Point Code)

```
/**
 * sample code to show use of Save point and roll back
 */
Savepoint sp = Database.setSavepoint();
try{
    insert parent;
    insert child;
}
catch(DmlException ex){
    // We catch DMLException as we have DML statements in try block
    Database.rollback(sp);
    // this will undo the insert of parent as well if exception is while insert of child.
    // Returning the transaction to to a known state.
}
catch(Exception e) {
    //This block will catch if any other unknown exceptions.
    System.debug('Exception caught: ' + e.getMessage());
    Database.rollback(sp);
}
```

5.10.2 For Lightning

Lightning Error Handling

1. Lightning Component send a request for server side action. This can cause a server side error.
2. The controller sends a response to the Lightning component.
3. The Lightning component processes the response in a callback function. This can trigger a client-side error (unexpected response).
4. Server side action methods should be static and @AuraEnabled. Use try catch block to return a readable error message to the Component.
5. Throw an AuraHandledException in the catch block. This allows you to provide a custom user-friendly error message.
6. Only 'AuraHandledException' type should be thrown from lightning context if the error message needs to be shown in user interface
7. The easiest way to do this if you're working on a page that's rendered in Lightning Experience is to display a Lightning Toast notification.
8. You cannot extend AuraHandledException to create a custom exception because it is not an extensible Apex class. However, there is a trick that can help you bypass this limitation. You can add custom data to your AuraHandledException by following these steps.

8.1. Create a simple wrapper class that can hold the data.

// Wrapper class for my custom exception data

```
public class CustomExceptionData {  
  
    public String name;  
  
    public String message;  
  
    public Integer code;  
  
    public CustomExceptionData(String name, String message, Integer code) {  
  
        this.name = name;  
  
        this.message = message;  
  
        this.code = code;  
  
    }  
  
}
```

8.2. Instantiate your custom class, serialize it as JSON, then pass it to the AuraHandledException.

```
// Throw an AuraHandledException with custom data
```

```
CustomExceptionData data = new CustomExceptionData('MyCustomServerError', 'Some message about the error', 123);
```

```
throw new AuraHandledException(JSON.serialize(data));
```

8.3. Finally, on the client-side (Lightning controller or helper), parse the error message string as JSON, and access your custom error data.

```
// Parse custom error data & report it
```

```
let errorData = JSON.parse(error.message);
```

```
console.error(errorData.name + " (code "+ errorData.code +)": "+ errorData.message);
```

Lightning Error Handling (Code)

○ Code Snippet

```
// Best practice: user-friendly error message provided by an AuraHandledException
// Controller of the lightning component
@AuraEnabled
public static void triggerBasicAuraHandledError() {
    try {
        integer a = 1 / 0; // Division by zero causes exception
    }
    catch (Exception e) {
        // "Convert" the exception into an AuraHandledException
        throw new AuraHandledException('Darn it! Something went wrong: '
            + e.getMessage());
    }
    finally {
        // Something executed whether there was an error or not
    }
}

//Client Component Controller
// Server-side action callback
function(response) {
    // Checking the server response state
    let state = response.getState();
    if (state === "SUCCESS") {
        // Process server success response
    }
}
```



```

        let returnValue = response.getReturnValue();
    }
    else if (state === "ERROR") {
        // Process error returned by server
        let errors = response.getError();
        let message = 'Unknown error'; // Default error message
        // Retrieve the error message sent by the server
        if (errors && Array.isArray(errors) && errors.length > 0) {
            message = errors[0].message;
        }
        // Display the message
        console.error(message);
    }
    else {
        // Handle other reponse states
    }
}

```

○ Toast Notification Snippet

```

// a helper method that can handler server processing errors to show a toast to user
handleErrors : function(errors) {
    // Configure error toast
    let toastParams = {
        title: "Error",
        message: "Unknown error", // Default error message
        type: "error"
    };
    // Pass the error message if any
    if (errors && Array.isArray(errors) && errors.length > 0) {
        toastParams.message = errors[0].message;
    }
    // Fire error toast
    let toastEvent = $A.get("e.force:showToast");
    toastEvent.setParams(toastParams);
    toastEvent.fire();
}

```

5.10.3 Error Framework

Error Log Framework

Below is the sample code for a simple Error Log Framework. Listing the features,

- The framework logs the error messages on a custom object. Email alerts can be setup on the error object to notify admins in case of exceptions to take the appropriate action.
- The framework provides methods to log exceptions in different scenarios
 - `businessException` : This method can be used to log business validation exceptions in case it needs to be captured for analysis like reporting for better user training
 - `genericException` : System exceptions would be handled by this method. It can accept exception object as input to capture its details on the error log object
 - `logDMLerrors` : This method can be used to log DML exceptions. In case of partial record failures, this method can capture the error message against each record/field
 - `insertAttachment` : Method to capture DML, integration request, response messages against the error record as an attachment.
- Object Details (PHX_Error_Log__c)

FIELD	FIELD NAME	DATA TYPE
Error Type	Error_Type__c	Picklist
Exception Type	Exception_Type__c	Text(40)
Class Method	Class_Method__c	Text(100)
Class Name	Class_Name__c	Text(100)
Error Log Name	Name	Text(80)
Error Message	Error_Message__c	Text(255)
Object Id	Object_Id__c	Text(100)
Message	Message__c	Long Text Area(32768)
Source URL	Source_URL__c	URL(255)
Username	Username__c	Lookup(User)

Error Log Framework(Code)

○ Framework Class

```
/**
 * PHX_Error_Log --- This is a generic class created to handle exception in case of
 * Integration and application scenarios. This provides different methods to handle the error logging.
 * Create error record/log attachments against it
 * @author Deloitte
```

```

*/

public with sharing class PHX_Error_Log {

    /**
     * This method is used to capture business exceptions or DML exceptions. It accepts the error and the error
     * description as input
     * @param
     *   Parameter 1 : Error Name - unique name for the record
     *   Parameter 2 : Error Type - type of the error (inbound,outbound,javascript,component etc.).
     * This is a picklist field
     *   Parameter 3 : Error Message - Message details in short
     *   Parameter 4 : Error Description - Detailed stack trace of the message
     *   Parameter 5 : ARB Class Name - Class in which the error occurred
     *   Parameter 6 : ARB Method Name - Method in which the error occurred
     *   Parameter 7 : Object Id - Record id for which the error occurred
     *   Parameter 8 : URL, for integrations
     * @return error object id to further add attachments under it
     */

    public static String businessException(String sErrorName, String sErrorType, String sErrorMessage,
    String sErrorDesc,
                                     String sClassName, String sClassMethod,
                                     String sObjectId, String sURL) {
        if(sErrorType != 'Javascript'){
            PHX_Error_Log__c objError = new PHX_Error_Log__c();
            objError.Class_Name__c = sClassName;
            objError.Class_Method__c = sClassMethod;

            objError.Error_Type__c = sErrorType;

            if (String.isNotBlank(sErrorMessage) && sErrorMessage.length() > 254)
                objError.Error_Message__c = sErrorMessage.substring(0, 254);
            else
                objError.Error_Message__c = sErrorMessage;

            //updates the object id. This can be a salesforce id or external id
            if (String.isNotBlank(sObjectId) && sObjectId.length() > 99)
                objError.Object_Id__c = sObjectId.substring(0, 99);
            else
                objError.Object_Id__c = sObjectId;

            if(String.isNotBlank(sErrorDesc) && sErrorDesc.length() > 32767)

```

```

        objError.Message__c = sErrorDesc.substring(0,32767);
    else
        objError.Message__c = sErrorDesc;

    if (String.isNotBlank(sErrorType) && sErrorType.length() > 39)
        objError.Exception_Type__c = sErrorType.substring(0, 39);
    else
        objError.Exception_Type__c = sErrorType;

    //Stores the url. This is could be a callout endpoint url or the page url for javascript exceptions
    objError.Source_URL__c = sURL;

    if (String.isNotBlank(sErrorName) && sErrorName.length() > 79)
        objError.Name = sErrorName.substring(0, 79);
    else
        objError.Name = sErrorName;

    insert objError;
    //System.debug('objError==' + objError);
    return objError.Id;
}
return null;
}

/**
 * This method is used to capture generic exceptions. It accepts the exception object as input
 * @param
 *   Parameter 1 : Error Name - unique name for the record
 *   Parameter 2 : Error Type - type of the error (inbound,outbound,javascript,component etc.).
 * This is a picklist field
 *   Parameter 3 : Exception - Exception object obtained
 *   Parameter 4 : ARB Class Name - Class in which the error occurred
 *   Parameter 5 : ARB Method Name - Method in which the error occurred
 *   Parameter 6 : Object Id - Record id for which the error occurred
 *   Parameter 7 : URL, for integrations
 * @return error object id to further add attachments under it
 */
public static String genericException(String sErrorName, String sErrorType, Exception e,
                                     String sClassName, String sClassMethod,
                                     String sObjectId, String sURL) {
    PHX_Error_Log__c objError = new PHX_Error_Log__c();
    objError.Class_Name__c = sClassName;

```

```

objError.Class_Method__c = sClassMethod;

objError.Error_Type__c = sErrorType;

if (e == null) {
    objError.Error_Message__c = 'No Exception';
    objError.Message__c = 'No Exception';
    objError.Exception_Type__c = 'No Exception';
}
else {
    //gets the exception details and puts it on the error log record
    String sErrMsg = e.getTypeName() + '++Message: ' + e.getMessage();
    if (sErrMsg.length() > 254)
        objError.Error_Message__c = sErrMsg.substring(0, 254);
    else
        objError.Error_Message__c = sErrMsg;
    objError.Message__c = e.getStackTraceString();
    if (e.getTypeName().length() > 39)
        objError.Exception_Type__c = e.getTypeName().substring(0, 39);
    else
        objError.Exception_Type__c = e.getTypeName();
}
//updates the object id. This can be a salesforce id or external id
if (String.isNotBlank(sObjectId) && sObjectId.length() > 99)
    objError.Object_Id__c = sObjectId.substring(0, 99);
else
    objError.Object_Id__c = sObjectId;
objError.Source_URL__c = sURL;

if (String.isNotBlank(sErrorName) && sErrorName.length() > 79)
    objError.Name = sErrorName.substring(0, 79);
else
    objError.Name = sErrorName;
insert objError;
//System.debug('objError==' + objError);
return objError.Id;
}

/**
 * This method is used to capture business exceptions or DML exceptions. It accepts the error and
 * the error description as input
 * @param

```

```

* Parameter 1 : Error Id - id of the error log record created for the exception
* Parameter 2 : Database Result - list of Database Result records for DML operations
* @return no return value
*/

public static void logDMLerrors(String sErrorId, List<Database.SaveResult> lstSaveResult,
    List<Database.UpsertResult> lstUpsertResult) {
    string sErrorMsg = "";

    //looks out for any insert results and updates the DML Exception details on error log record
    if (lstSaveResult != null) {
        for (Integer i = 0; i < lstSaveResult.size(); i++) {
            Database.SaveResult objResult = lstSaveResult[i];
            //system.debug('result===== ' + objResult);
            //system.debug('result===== ' + objResult.getErrors());
            //system.debug('result===== ' + objResult.isSuccess());
            if (!objResult.isSuccess()) {
                for (Database.Error objError : objResult.getErrors()) {
                    sErrorMsg += sErrorMsg;
                    //System.debug('The following error has occurred. ');
                    //System.debug(objError.getStatusCode() + ': ' + objError.getMessage());
                    //System.debug('Fields that affected this error: ' + objError.getFields());
                    sErrorMsg = i + ' ' + objError.getMessage();
                    sErrorMsg = sErrorMsg + '++++++\n Error Fields that caused exception:\n' +
                        objError.getFields() + '\n+++++++';
                }
            }
        }
    }

    //looks out for any upsert results and updates the DML Exception details on error log record
    if (lstUpsertResult != null) {
        for (Integer i = 0; i < lstUpsertResult.size(); i++) {

            Database.UpsertResult objResult = lstUpsertResult[i];
            //system.debug('result===== ' + objResult);
            //system.debug('result===== ' + objResult.getErrors());
            //system.debug('result===== ' + objResult.isSuccess());
            if (!objResult.isSuccess()) {
                for (Database.Error objError : objResult.getErrors()) {
                    sErrorMsg += sErrorMsg;
                    //System.debug('The following error has occurred. ');

```

```

        //System.debug(objError.getStatusCode() + ': ' + objError.getMessage());
        //System.debug('Fields that affected this error: ' + objError.getFields());
        sErrorMsg = i + ' ' + objError.getMessage();
        sErrorMsg = sErrorMsg + '++++++\n Error Fields that caused exception:\n' +
objError.getFields() +
        '\n++++++';
    }
}

}
PHX_Error_Log__c objError = new PHX_Error_Log__c(id = sErrorId, Message__c = sErrorMsg);
update objError;
//Creates an attachment to hold the complete error information to avoid data loss
InsertAttachment( sErrorId, 'LogDMLerrors', sErrorMsg);

}
}

/**
 * This method is used to capture generic exceptions. It accepts the exception object as input
 * @param
 *   Parameter 1 : Error Id - id of the error log record created for the exception
 *   Parameter 2 : FileName - file which contains the error details
 *   Parameter 3 : Error Text - short description of the error
 * @return  no return value
 */
public static void insertAttachment(String sErrorId, String sFileName, String sErrorText) {
    if(String.isNotBlank(sErrorText) && String.isNotBlank(sErrorId)){
        Attachment objAtt = new Attachment();
        objAtt.Body = Blob.valueOf(sErrorText);
        objAtt.Name = sFileName + '.txt';
        objAtt.parentId = sErrorId;
        insert objAtt;
    }
}
}
}

```

○ Test Class

```

/**
 * PHX_Error_Log_Test --- Test class for generic exception handling class
 * @author Deloitte
 */

```

```

@istest
public class PHX_Error_Log_Test {

    /**
     * Test data setup method to create user
     */
    @testsetup
    static void createusers()
    {
        // Setup test data
        // Create a unique UserName
        String uniqueUserName = 'standarduser' + DateTime.now().getTime() + '@testorg.com';
        // This code runs as the system user
        Profile standardUser = [SELECT Id FROM Profile WHERE Name='Standard User'];
        User userRec = new User(Alias = 'standt', Email='abc_xyz@deloitte.com',
        EmailEncodingKey='UTF-8', LastName='Testing', LanguageLocaleKey='en_US',
        LocaleSidKey='en_US', ProfileId = standardUser.Id,
        TimeZoneSidKey='America/Los_Angeles',UserName=uniqueUserName);
        insert userRec;
    }

    /**
     * Test method for error log business exception
     */
    @istest
    static void testbusinessException()
    {
        User testUser = [SELECT Id FROM User where Email = 'abc_xyz@deloitte.com'];
        System.runAs(testUser)
        {
            String sErrorName = 'Error Name' ;
            String sErrorType = 'Error Type' ;
            String sErrorMessage = 'Error Message' ;
            String sErrorDesc = 'Error Desc' ;
            String sClassName = 'Class Name' ;
            String sClassMethod = 'Class Method' ;
            String sObjectId = 'Object ID' ;
            String sURL = 'URL' ;
            test.startTest();

            String errorLogId = PHX_Error_Log.businessException(sErrorName, sErrorType, sErrorMessage,
sErrorDesc,
                        sClassName, sClassMethod, sObjectId, sURL);

```



```

    PHX_Error_Log__c objError = [select Id,Name,Error_Type__c,Exception_Type__c,Class_Method__c,
                                   Class_Name__c,Error_Message__c,Object_Id__c,Message__c,Source_URL__c,
                                   Username__c from PHX_Error_Log__c where id = :errorLogId];

    system.assertEquals(sErrorName, objError.Name);
    system.assertEquals(sErrorType, objError.Error_Type__c);
    system.assertEquals(sErrorMessage, objError.Error_Message__c);
    system.assertEquals(sErrorDesc, objError.Message__c);
    system.assertEquals(sClassMethod, objError.Class_Method__c);
    system.assertEquals(sObjectId, objError.Object_Id__c);
    system.assertEquals(sURL, objError.Source_URL__c);
    system.assertEquals(sClassName, objError.Class_Name__c);

    // testing length validation
    sErrorName =
'greaterthan79charactersgreaterthan79charactersgreaterthan79charactersgreaterthan79characters';
    String errorLogId1 = PHX_Error_Log.businessException(sErrorName, sErrorType, sErrorMessage,
sErrorDesc,
                sClassName, sClassMethod, sObjectId, sURL);
    PHX_Error_Log__c objError1 = [select Id,Name from PHX_Error_Log__c where id = :errorLogId1];

    system.assertEquals('greaterthan79charactersgreaterthan79charactersgreaterthan79charactersgreatertha',
        objError1.Name);
    test.stopTest();
}
}

/**
 * Test method for error log generic exception
 */
@istest
static void testgenericException()
{
    User testUser = [SELECT Id FROM User where Email = 'abc_xyz@deloitte.com'];
    System.runAs(testUser)
    {
        String sErrorName = 'Error Name' ;
        String sErrorType = 'inbound' ;
        String sClassName = 'Class Name' ;
        String sClassMethod = 'Class Method' ;
        String sObjectId = 'Object ID' ;
        String sURL = 'URL' ;
        System.DmlException e = null;
    }
}

```

```

test.startTest();
Account[] accts = new Account[]{new Account(billingcity = 'San Jose')};

try { insert accts;}
catch (System.DmlException exc) { e=exc;}

String errorLogId = PHX_Error_Log.genericException(sErrorName, sErrorType, e,
    sClassName, sClassMethod, sObjectId, sURL);
PHX_Error_Log__c objError = [select Id,Name,Error_Type__c,Class_Method__c,Class_Name__c,
    Object_Id__c,Source_URL__c,Username__c from PHX_Error_Log__c where id = :errorLogId];

system.assertEquals(sErrorName, objError.Name);
system.assertEquals(sErrorType, objError.Error_Type__c);
system.assertEquals(sClassMethod, objError.Class_Method__c);
system.assertEquals(sObjectId, objError.Object_Id__c);
system.assertEquals(sURL, objError.Source_URL__c);
system.assertEquals(sClassName, objError.Class_Name__c);

// text size limit of field
sErrorName =
'greaterthan79charactersgreaterthan79charactersgreaterthan79charactersgreaterthan79characters';
sErrorType = 'inboundinboundinboundinboundinboundinbound' ;// greater than 39

// test no exception where e is null
String errorLogId1 = PHX_Error_Log.genericException(sErrorName, sErrorType, null,
    sClassName, sClassMethod, sObjectId, sURL);
PHX_Error_Log__c objError1 = [select Id,Name,Exception_Type__c,Error_Message__c,
    Message__c,Source_URL__c from PHX_Error_Log__c where id = :errorLogId1];

system.assertEquals('No Exception', objError1.Error_Message__c);
system.assertEquals('No Exception', objError1.Message__c);
system.assertEquals('No Exception', objError1.Exception_Type__c);

test.stopTest();
}
}

/**
 * Test method for error log DML exception using Database insert with partial processing
 */
@istest
static void testlogDMLerrors()

```

```

{
    User testUser = [SELECT Id FROM User where Email = 'abc_xyz@deloitte.com'];
    System.runAs(testUser)
    {
        PHX_Error_Log__c errorLog = new PHX_Error_Log__c();
        insert errorLog;
        Account acc = new Account(name = 'name');
        insert acc;
        String sErrorId = errorLog.id;
        List<Database.SaveResult> lstSaveResult = null;
        List<Database.UpsertResult> lstUpsertResult = null;
        List<Database.UpsertResult> lstUpsertResult1 = null;
        List<Database.UpsertResult> lstUpsertResult3 = null;

        Account[] accts = new List<Account>{new Account(Name='Account1', billingcity = 'Bombay'),
                                             new Account (Name = 'Account2', billingcity = 'Bombay'),
                                             new Account (billingcity = 'Bombay')};
        lstSaveResult = Database.insert(accts, false);
        system.debug('lstSaveResult---' + lstSaveResult);

        Account[] acc2 = [select id from Account where Name = 'Account1'];
        try{
            acc2[0].billingcity = 'Mumbai';
            lstUpsertResult1 = Database.upsert(acc2, true);
            system.debug('try lstUpsertResult1---' + lstUpsertResult1);
        } catch (System.DmlException exc)
        {
            system.debug('catch lstUpsertResult1---' + exc);
        }

        PHX_Error_Log.logDMLerrors(sErrorId, lstSaveResult, lstUpsertResult1);
    }
}

/**
 * Test method for error log attachment creation
 */
@istest
static void testinsertAttachment()
{
    User testUser = [SELECT Id FROM User where Email = 'abc_xyz@deloitte.com'];
    System.runAs(testUser)
    {

```

```

    PHX_Error_Log__c errorLog = new PHX_Error_Log__c();
    insert errorLog;
    Account acc = new Account(name = 'name');
    insert acc;
    String sErrorId = errorLog.id;
    system.debug('sErrorId before passing -- '+sErrorId);
    String sFileName = 'filename';
    String sErrorText = 'errortext';
    PHX_Error_Log.insertAttachment(sErrorId, sFileName, sErrorText);
    Attachment objAtt = [select Id, Body, Name, parentId from Attachment where name = 'filename.txt'];
    system.assertEquals(Blob.valueOf(sErrorText), objAtt.Body);
    system.assertEquals(sFileName + '.txt', objAtt.Name);
    system.assertEquals(Blob.valueOf(sErrorText), objAtt.Body);
}
}
}

```

○ Error Object (ARB_Error_Log__c.object)

```

<?xml version="1.0" encoding="UTF-8"?>
<CustomObject xmlns="http://soap.sforce.com/2006/04/metadata">
  <actionOverrides>
    <actionName>Accept</actionName>
    <type>Default</type>
  </actionOverrides>
  <actionOverrides>
    <actionName>CancelEdit</actionName>
    <type>Default</type>
  </actionOverrides>
  <actionOverrides>
    <actionName>Clone</actionName>
    <type>Default</type>
  </actionOverrides>
  <actionOverrides>
    <actionName>Delete</actionName>
    <type>Default</type>
  </actionOverrides>
  <actionOverrides>
    <actionName>Edit</actionName>
    <type>Default</type>
  </actionOverrides>
  <actionOverrides>
    <actionName>List</actionName>
    <type>Default</type>
  </actionOverrides>

```

```

</actionOverrides>
<actionOverrides>
  <actionName>New</actionName>
  <type>Default</type>
</actionOverrides>
<actionOverrides>
  <actionName>SaveEdit</actionName>
  <type>Default</type>
</actionOverrides>
<actionOverrides>
  <actionName>Tab</actionName>
  <type>Default</type>
</actionOverrides>
<actionOverrides>
  <actionName>View</actionName>
  <type>Default</type>
</actionOverrides>
<allowInChatterGroups>false</allowInChatterGroups>
<compactLayoutAssignment>SYSTEM</compactLayoutAssignment>
<deploymentStatus>Deployed</deploymentStatus>
<description>Used to capture the exceptions encountered by users;</description>
<enableActivities>true</enableActivities>
<enableBulkApi>true</enableBulkApi>
<enableChangeDataCapture>false</enableChangeDataCapture>
<enableFeeds>false</enableFeeds>
<enableHistory>true</enableHistory>
<enableReports>true</enableReports>
<enableSearch>true</enableSearch>
<enableSharing>true</enableSharing>
<enableStreamingApi>true</enableStreamingApi>
<fields>
  <fullName>Error_Type__c</fullName>
  <description>used to determine the type of error that occurred;</description>
  <externalId>false</externalId>
  <label>Error Type</label>
  <required>false</required>
  <trackHistory>false</trackHistory>
  <trackTrending>false</trackTrending>
  <type>Picklist</type>
  <valueSet>
    <valueSetDefinition>
      <sorted>true</sorted>
      <value>

```

```

        <fullName>Apex Trigger</fullName>
        <default>false</default>
        <label>Apex Trigger</label>
    </value>
    <value>
        <fullName>Business Validation</fullName>
        <default>false</default>
        <label>Business Validation</label>
    </value>
    <value>
        <fullName>Component</fullName>
        <default>false</default>
        <label>Component</label>
    </value>
    <value>
        <fullName>DML Errors</fullName>
        <default>false</default>
        <label>DML Errors</label>
    </value>
    <value>
        <fullName>Integration Batch Job</fullName>
        <default>false</default>
        <label>Integration Batch Job</label>
    </value>
    <value>
        <fullName>Integration Inbound</fullName>
        <default>false</default>
        <label>Integration Inbound</label>
    </value>
    <value>
        <fullName>Integration Outbound</fullName>
        <default>false</default>
        <label>Integration Outbound</label>
    </value>
    <value>
        <fullName>VF Controller</fullName>
        <default>false</default>
        <label>VF Controller</label>
    </value>
</valueSetDefinition>
</valueSet>
</fields>
<fields>

```

```

    <fullName>Exception_Type__c</fullName>
    <description>Used to represent the exception type;</description>
    <externalId>>false</externalId>
    <label>Exception Type</label>
    <length>40</length>
    <required>>false</required>
    <trackHistory>>false</trackHistory>
    <trackTrending>>false</trackTrending>
    <type>Text</type>
    <unique>>false</unique>
</fields>
<fields>
    <fullName>Class_Method__c</fullName>
    <description>used to represent the method from which exception occurred;</description>
    <externalId>>false</externalId>
    <label>Class Method</label>
    <length>100</length>
    <required>>false</required>
    <trackHistory>>false</trackHistory>
    <trackTrending>>false</trackTrending>
    <type>Text</type>
    <unique>>false</unique>
</fields>
<fields>
    <fullName>Class_Name__c</fullName>
    <description>Used to represent the class where exception occurred;</description>
    <externalId>>false</externalId>
    <label>Class Name</label>
    <length>100</length>
    <required>>false</required>
    <trackHistory>>false</trackHistory>
    <trackTrending>>false</trackTrending>
    <type>Text</type>
    <unique>>false</unique>
</fields>
<fields>
    <fullName>Error_Message__c</fullName>
    <description>Used to hold the error message;</description>
    <externalId>>false</externalId>
    <label>Error Message</label>
    <length>255</length>
    <required>>false</required>
    <trackHistory>>false</trackHistory>

```

```

    <trackTrending>>false</trackTrending>
    <type>Text</type>
    <unique>>false</unique>
</fields>
<fields>
    <fullName>Object_Id__c</fullName>
    <description>Used to hold the Object details (SFDC/External id);</description>
    <externalId>>false</externalId>
    <label>Object Id</label>
    <length>100</length>
    <required>>false</required>
    <trackHistory>>false</trackHistory>
    <trackTrending>>false</trackTrending>
    <type>Text</type>
    <unique>>false</unique>
</fields>
<fields>
    <fullName>Message__c</fullName>
    <description>Used to hold the stack trace of the error;</description>
    <externalId>>false</externalId>
    <label>Message</label>
    <length>32768</length>
    <trackHistory>>false</trackHistory>
    <trackTrending>>false</trackTrending>
    <type>LongTextArea</type>
    <visibleLines>3</visibleLines>
</fields>
<fields>
    <fullName>Source_URL__c</fullName>
    <description>Holds the url details;</description>
    <externalId>>false</externalId>
    <label>Source URL</label>
    <required>>false</required>
    <trackHistory>>false</trackHistory>
    <trackTrending>>false</trackTrending>
    <type>Url</type>
</fields>
<fields>
    <fullName>Username__c</fullName>
    <deleteConstraint>SetNull</deleteConstraint>
    <description>lookup to user who triggered the transaction;</description>
    <externalId>>false</externalId>
    <label>Username</label>

```

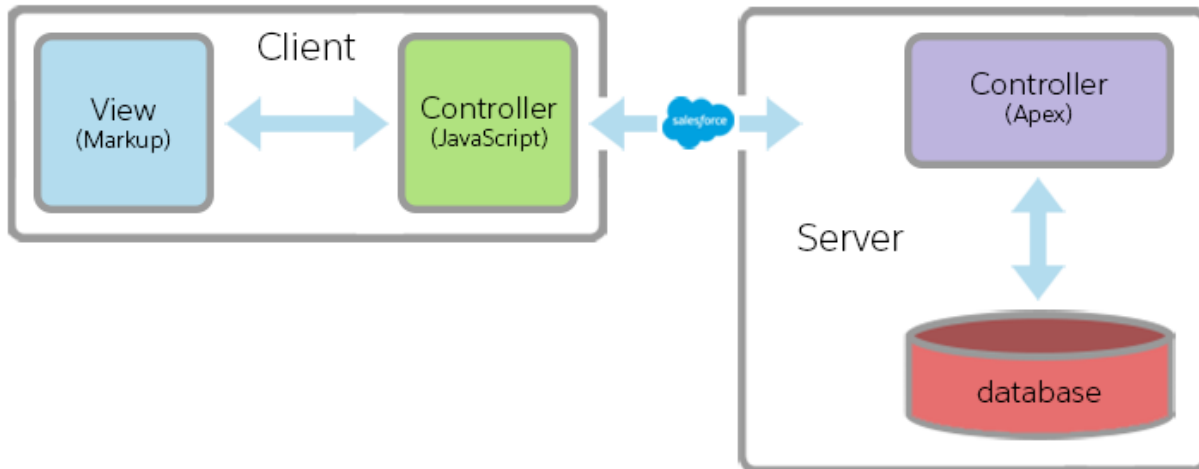


```
<referenceTo>User</referenceTo>
<relationshipName>PHX_Error_logs</relationshipName>
<required>>false</required>
<trackHistory>>false</trackHistory>
<trackTrending>>false</trackTrending>
<type>Lookup</type>
</fields>
<label>ARB Error Log</label>
<listViews>
  <fullName>All</fullName>
  <filterScope>Everything</filterScope>
  <label>All</label>
</listViews>
<nameField>
  <label>ARB Error Log Name</label>
  <trackHistory>>false</trackHistory>
  <type>Text</type>
</nameField>
<pluralLabel>ARB Error logs</pluralLabel>
<searchLayouts/>
<sharingModel>ReadWrite</sharingModel>
<visibility>Public</visibility>
</CustomObject>
```

5.11 Lightning Component (Aura)

Lightning Components

The Lightning Component framework is a UI framework for developing web apps for mobile and desktop devices. It's a modern framework for building single-page applications with dynamic, responsive user interfaces for Lightning Platform apps. It uses JavaScript on the client side and Apex on the server side.



Component Bundle

A component bundle contains a component or an app and all its related resources. All resources in the component bundle follow the naming convention and are auto-wired. For example, a controller `<componentName>Controller.js` is auto-wired to its component, which means that you can use the controller within the scope of that component.

Resource	Resource Name	Usage	See Also
Component or Application	sample.cmp or sample.app	The only required resource in a bundle. Contains markup for the component or app. Each bundle contains only one component or app resource.	Creating Components aura:application
CSS Styles	sample.css	Contains styles for the component.	CSS in Components
Controller	sampleController.js	Contains client-side controller methods to handle events in the component.	Handling Events with Client-Side Controllers
Design	sample.design	Used for passing attributes to component from Lightning App Builder, Lightning pages, Community Builder, or Flow Builder.	Aura Component Bundle Design Resources
		A description, sample code, and one or multiple	Providing Component

Documentation	sample.auradoc	references to example components	Documentation
Renderer	sampleRenderer.js	Client-side renderer to override default rendering for a component.	Create a Custom Renderer
Helper	sampleHelper.js	JavaScript functions that can be called from any JavaScript code in a component's bundle	Sharing JavaScript Code in a Component Bundle
SVG File	sample.svg	Custom icon resource for components used in the Lightning App Builder or Community Builder.	Configure Components for Lightning Pages and the Lightning App Builder

For more details please refer -

https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/components_bundle.htm

Lightning Component Interfaces

Interfaces determine a component's shape by defining its attributes. Implement an interface to allow a component to be used in different contexts, such as on a record page or in Lightning App Builder. The lightning components should have proper description and should implement just the right interface. Unwanted interfaces should not be implemented.

Implement these platform interfaces to allow a component to be used in different contexts, or to enable your component to receive extra context data. A component can implement multiple interfaces. Some interfaces are intended to be implemented together, while others are mutually exclusive. Some interfaces have an effect only in Lightning Experience and the Salesforce app. Below are some common interfaces. For the entire list refer to -

<https://developer.salesforce.com/docs/component-library/overview/interfaces>

- flexipage:availableForAllPageTypes
 - A global interface that makes a component available in the Lightning App Builder, and for any type of Lightning page.
 - To appear in the utility bar, a component must implement the flexipage:availableForAllPageTypes interface.
- flexipage:availableForRecordHome
 - If the component is designed only for record pages, implement flexipage:availableForRecordHome interface instead of flexipage:availableForAllPageTypes.
- forceCommunity:availableForAllPageTypes
 - To appear in Community Builder, a component must implement the forceCommunity:availableForAllPageTypes interface.
- force:appHostable
 - Allows a component to be used as a custom tab in Lightning Experience or the Salesforce app.

- `force:lightningQuickAction`
 - Allows a component to display in a panel with standard action controls, such as a Cancel button. These components can also display and implement their own controls but should handle events from the standard controls. If you implement `force:lightningQuickAction`, you can't implement `force:lightningQuickActionWithoutHeader` within the same component.
- `force:lightningQuickActionWithoutHeader`
 - Allows a component to display in a panel without additional controls. The component should provide a complete user interface for the action. If you implement `force:lightningQuickActionWithoutHeader`, you can't implement `force:lightningQuickAction` within the same component.
- `ltng:allowGuestAccess`
 - Add the `ltng:allowGuestAccess` interface to your Lightning Out dependency app to make it available to users without requiring them to authenticate with Salesforce. This interface lets you build your app with Lightning components, and deploy it anywhere and to anyone.
 - Refer: https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/lightning_out.htm

Lightning Attributes Considerations

- Use the design file to control which attributes are exposed to the Lightning App Builder
- Make your attributes easy to use and understandable to an administrator. Don't expose SOQL queries, JSON objects, or Apex class names
- Create inner wrapper classes as separate classes as inner wrapper classes are not supported from v42 onwards.
 - Instead of the following:
 - `<aura:attribute name="Release" type="ProjectWrapper.Release" />`
 - Use the following: Remove the inner class "Release" from the ProjectWrapper class and create it as a separate class.
 - `<aura:attribute name="Release" type="Release" />`
- Give your required attributes default values. When a component that has required attributes with no default values is added to the App Builder, it appears invalid, which is a poor user experience
- Use basic supported types (string, integer, boolean) for any exposed attributes
- Specify a min and max attribute for integer attributes in the `<design:attribute>` element to control the range of accepted values
- String attributes can provide a data source with a set of predefined values allowing the attribute to expose its configuration as a pick-list
- Give all attributes a label with a friendly display name
- Provide descriptions to explain the expected data and any guidelines, such as data format or expected range of values. Description text appears as a tooltip in the Property Editor
- To delete a design attribute for a component that implements the `flexipage:availableForAllPageTypes` or `forceCommunity:availableForAllPageTypes` interface, first remove the interface from the component before deleting the design attribute. Then re-implement the interface. If the component is referenced in a Lightning page, you must remove the component from the page before you can change it

Base Components

- Base Lightning components are the building blocks that handle the details of HTML and CSS for you i.e. lightning:button
- There is a wide variety of base components available in different namespace i.e. ui, lightning, force. Use these base components to have consistent and modern Lightning Experience, Salesforce app, and Lightning Communities user interfaces.
- It is recommended to use the lightning namespace components. The lightning namespace components are optimized for common use cases. Beyond being equipped with the Lightning Design System styling, they handle accessibility, real-time interaction, and enhanced error messages.
- With LockerService enforced, you can't traverse the DOM for components you don't own. Instead of accessing the DOM tree, take advantage of value binding with component attributes and use component methods that are available to you.
- For details on the Base Components available refer to the below link -
 - <https://developer.salesforce.com/docs/component-library/overview/components>

Deloitte.

5.11.1 Lightning Best Practices

Best Practices - Components

- Use SLDS (Lightning Design System) as much as possible since Salesforce has created a lot of base components with SLDS styling embedded into these components which reduces the need for custom CSS effort. Bootstrap and other UI libraries should be used only with careful consideration.
- Use Lightning Data Service (LDS) for loading and performing CRUD operations instead of performing CRUD in Apex.
 - LDS is built on highly efficient local storage. Records loaded are cached and shared across all components. As records are shared, if one component modifies others are notified. Record changes in the back-end via Salesforce via integrations would also trigger notifications.
 - LDS enforces object and Field Level security and enforces sharing rules
 - LDS in detail - https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/data_service
 - Considerations: https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/data_service_considerations
- Use `<aura:if>` instead of toggling visibility using CSS Toggling visibility. Using the `<aura:if>` tag is the preferred approach to conditionally display markup but there are alternatives. Consider the performance cost and code maintainability when you design components.
- Avoid Third party java-script libraries - It is recommended to avoid third-party DOM Manipulation libraries (jQuery), UI Libraries (e.g. Bootstrap) and MVC Frameworks. In case if it is not avoidable, use minified version of these library and style sheet.
- Avoid Inheritance in Lightning Components to the extent possible. Composition is the fundamental model for reusing code. Refer: https://trailhead.salesforce.com/en/content/learn/modules/lex_dev_lc_vf_tips/lex_dev_lc_vf_tips
- Lightning Service Components can be used as an alternative to Inheritance in Lightning Components to resolve performance issues. Refer: <https://developer.salesforce.com/blogs/2018/08/implement-and-use-lightning-service-components.html>
- Use unbound expression for variable binding instead of bound expression. Refer to variable binding expression for more details.
- Use lightning: layout and layout items to build responsive pages instead of hard-coding slds size classes and widths.
- Use Base Lightning Components (`<lightning:*>` instead of `<ui:*>` namespace elements.
 - E.g. Use `<lightning:button>` instead of `<ui:button>`
 - Styles: Base Lightning Components are styled with the native Lightning look and feel.
 - Performance: Base Lightning Components are already loaded at the client-side and don't require additional download or processing. Performance optimization efforts are also focused on components in the lightning namespace.
 - Innovation: The Lightning namespace is where components are being actively developed. This is what you can expect to see new and improved components moving forward.
 - Accessibility: Base Lightning Components are built for accessibility.
 - Client-side validation: Base Lightning Components include client-side validation when applicable.

- Use Community page optimizer extension in Chrome to analyze performance in community pages. through the following link for details :
https://help.salesforce.com/articleView?id=community_builder_page_optimizer.htm&type=5
- Use "\$Browser" global value provider instead of the slds classes to define the markup to be visible only in Refer: https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/expr_browser_value_provider
- Usage of custom labels wherever applicable (label, placeholder, static text, ADA- related attributes or assets, error messages, etc.).
- Avoid hard coding values like field labels, localized strings, help text, error messages etc. in lightning component markup and JavaScript controller
 - Instead create custom label for custom text values and access using \$Label as below
 - Label in a markup expression using the default namespace
 - {\$Label.c.labelName}
 - Label in JavaScript code if your org has a namespace
 - \$.get("\$Label.namespace.labelName")
 - Label expressions in markup are supported in .cmp and .app resources only.
- If the number of {\$Label.c.labelName} instances causes performance degradation in the lightning page, s can be to retrieve all the labels used in the component through apex controller in init method as Map object use this Map object instead of "\$Label.c.XXX"
- Parameterize the application/component events and have a single handler for an event per component instead of multiple handlers (with different names) in the same component.
- Ensure the markup elements are written in the following sequence: (1). attributes declaration, (2). registers/handlers for events ("init" handler first) and then (3). Remaining markup elements.
- DO NOT hardcode salesforce environment URLs in lightning component or apex class. Look into alternative making it generic.
- Reuse controller methods in actions by using label, name or data-* attributes to differentiate between the two elements triggering the events.
- Avoid unnecessary multiple sizing for same unit: e.g.: can be changed to since bigger sizes will take immediate smaller size units by default.
- Use variant attribute (variant = "label-hidden") in lightning:input if the label needs to be hidden. DO NOT use slds-assistive-text class or any other workaround for the same.
- Most of the hardcodings in the Lightning Markup is observed when the developer needs to build a custom edit/create screen for an object. Since Lightning Component does not provide an ability to dynamically access field labels (like in VisualForce pages), developers end up hard coding the labels in the lightning base component.
- One alternative to this is using Schema Class. In the init method, developer can fetch a map of fieldApiName to fieldLabel. It can be used to dynamically reference the map to fetch the label value. This functionality can be extended to fetch help text etc. This will also prevent code updates if the field label is changed in future.
 - For e.g. code:

```
/*Apex Controller*/
```

```
/*get api name vs label. Will be used for displaying field Label on front end*/
```

```
Map schemaMap = Schema.getGlobalDescribe().get('customObject__c').getDescribe().fields.getMap();
```



```
for(String fieldName: schemaMap.keySet()){

    initWrap.mapFieldAPIVsLabel.put(fieldName,schemaMap.get(fieldName).getDescribe().getLabel());

}

/*Lightning Component markup*/

<lightning:input aura:id="field" label="{!v.pcmIssueWrapper.mapFieldAPIVsLabel.name}"
value="{!v.pcmIssueRecord.Name}" readonly="true"/>
```

Best Practices - Helper

- All JavaScript logic for a controller should reside in Helper and only event handlers should be used in controller. <https://developer.salesforce.com/blogs/developer-relations/2015/06/understanding-javascript-controllers-v>
- Have all the client side business logic and server calls in Helper and invoke it from Lightning Controller
- Whenever there is server action in progress, there should be a Spinner indicating processing is underway.
- Performance
 - Optimize server round-trips. Consider combining several requests (actions) in a single composite request.
 - Before making a call to the server, make sure there's no other option to obtain the data.
 - When appropriate, consider passing data between components (using attributes, events, or method calls in different components).
 - When making a call to the server, limit the columns and rows of the result set. Only SELECT the columns you need and provide a paging mechanism if needed. Don't return huge numbers of rows at once.
 - Lazy load occasionally accessed data. Don't preload data that the user may never ask for (For example, in a list view the user may not click, or in a combo-box the user may not open).
 - Client-side filtering and sorting: Don't make a call to the server to filter or sort data you already have.
- Cache data when possible
 - Most server requests are read-only and idempotent, which means that a request can be repeated without causing data changes. The responses to idempotent actions can be cached and quickly reused for subsequent requests.
 - To cache data returned from an Apex method for any component with an API version of 44.0 or higher, use `@AuraEnabled(cacheable=true)`.
 - Scenarios:
 - Lightning Data Service <-- Single record
 - Storable actions <-- Collections of records, composite responses, custom data structures, third-party data
 - Custom cache <-- Complete control over caching implementation
- Use Storable Action:

Lightning component shows cached result instead of making immediate server trip.

On client side controller of the Lightning component, mark action as storable.

- Throw `AuraHandledException` in Apex to handle errors/exceptions gracefully and display meaningful errors user.

Lightning Data Service Example

○ LDS - Load Record

Component

```
<aura:component
implements="flexipage:availableForRecordHome,force:lightningQuickActionWithoutHeader,
force:hasRecordId">

    <aura:attribute name="record" type="Object"/>
    <aura:attribute name="simpleRecord" type="Object"/>
    <aura:attribute name="recordError" type="String"/>

    <force:recordData aura:id="recordLoader"
        recordId="{!v.recordId}"
        targetFields="{!v.simpleRecord}"
        targetError="{!v.recordError}"
        recordUpdated="{!c.handleRecordUpdated}"
    />

    <!-- Display a lightning card with details about the record -->
    <div class="Record Details">
        <lightning:card iconName="standard:account" title="{!v.simpleRecord.Name}" >
            <div class="slds-p-horizontal--small">
                <p class="slds-text-heading--small">
                    <lightning:formattedText title="Billing City" value="{!v.simpleRecord.BillingCity}" /></p>
                <p class="slds-text-heading--small">
                    <lightning:formattedText title="Billing State" value="{!v.simpleRecord.BillingState}" /></p>
            </div>
        </lightning:card>
    </div>

    <!-- Display Lightning Data Service errors, if any -->
    <aura:if isTrue="{!not(empty(v.recordError))}">
        <div class="recordError">
            {!v.recordError}</div>
    </aura:if>
</aura:component>

Js Controller
({
    handleRecordUpdated: function(component, event, helper) {
```

```

var eventParams = event.getParams();
if(eventParams.changeType === "LOADED") {
    // record is loaded (render other component which needs record data value)
    console.log("Record is loaded successfully.");
    console.log("You loaded a record in " +
        component.get("v.simpleRecord.Industry"));
} else if(eventParams.changeType === "CHANGED") {
    // record is changed
} else if(eventParams.changeType === "REMOVED") {
    // record is deleted
} else if(eventParams.changeType === "ERROR") {
    // there's an error while loading, saving, or deleting the record
}
}
})

```

○ LDS - Create Record

Component

```
<aura:component implements="flexipage:availableForRecordHome, force:hasRecordId">
```

```
<aura:attribute name="newContact" type="Object"/>
```

```
<aura:attribute name="simpleNewContact" type="Object"/>
```

```
<aura:attribute name="newContactError" type="String"/>
```

```
<aura:handler name="init" value="{!this}" action="{!c.doInit}"/>
```

```
<force:recordData aura:id="contactRecordCreator"
```

```
    layoutType="FULL"
```

```
    targetRecord="{!v.newContact}"
```

```
    targetFields="{!v.simpleNewContact}"
```

```
    targetError="{!v.newContactError}" />
```

```
<!-- Display the new contact form -->
```

```
<div class="Create Contact">
```

```
    <lightning:card iconName="action:new_contact" title="Create Contact">
```

```
        <div class="slds-p-horizontal--small">
```

```
            <lightning:input aura:id="contactField" label="First Name"
```

```
value="{!v.simpleNewContact.FirstName}"/>
```

```
            <lightning:input aura:id="contactField" label="Last Name"
```

```
value="{!v.simpleNewContact.LastName}"/>
```

```
            <lightning:input aura:id="contactField" label="Title" value="{!v.simpleNewContact.Title}"/>
```

```
        <br/>
```

```
        <lightning:button label="Save Contact" variant="brand" onclick="{!c.handleSaveContact}"/>
```

```

        </div>
    </lightning:card>
</div>

<!-- Display Lightning Data Service errors -->
<aura:if isTrue="{!not(empty(v.newContactError))}">
    <div class="recordError">
        {!v.newContactError}</div>
    </aura:if>

```

```

</aura:component>

```

JS Controller

```

({
    doInit: function(component, event, helper) {
        // Prepare a new record from template
        component.find("contactRecordCreator").getNewRecord(
            "Contact", // sObject type (objectApiName)
            null,      // recordTypeId
            false,     // skip cache?
            $A.getCallback(function() {
                var rec = component.get("v.newContact");
                var error = component.get("v.newContactError");
                if(error || (rec === null)) {
                    console.log("Error initializing record template: " + error);
                    return;
                }
                console.log("Record template initialized: " + rec.apiName);
            })
        );
    },

    handleSaveContact: function(component, event, helper) {
        if(helper.validateContactForm(component)) {
            component.set("v.simpleNewContact.AccountId", component.get("v.recordId"));
            component.find("contactRecordCreator").saveRecord(function(saveResult) {
                if (saveResult.state === "SUCCESS" || saveResult.state === "DRAFT") {
                    // record is saved successfully
                    var resultsToast = $A.get("e.force:showToast");
                    resultsToast.setParams({
                        "title": "Saved",
                        "message": "The record was saved."
                    });
                }
            });
        }
    }
});

```

```

        resultsToast.fire();

    } else if (saveResult.state === "INCOMPLETE") {
        // handle the incomplete state
        console.log("User is offline, device doesn't support drafts.");
    } else if (saveResult.state === "ERROR") {
        // handle the error state
        console.log("Problem saving contact, error: ' + JSON.stringify(saveResult.error));
    } else {
        console.log("Unknown problem, state: ' + saveResult.state + ', error: ' +
JSON.stringify(saveResult.error));
    }
    });
}
}
})

```

Concurrency Issues in LDS and Solutions

Using LDS, without a doubt, is a best practice. However, we tend to overlook concurrency issues in such cases.

For instance, let's say we edit a Case record using Lightning Component (Standard Edit button of Case is overridden with Lightning Component). User 1 edits Case 1 record. While he is still on the Edit screen, User 2 also edits Case 1 record. In such situations, data is overridden and User is not aware of that. LDS does not handle such scenarios.

Solutions:

Preferred:

Use of Platform Event and Lightning EMPApi

Anytime a record is updated, and the Edit button is overridden by Lightning Component, in the After Update trigger, insert a Platform Event record which stores the ID of the current record.

In the Lightning Component which is being invoked on Edit, we use EMPApi to handle the Platform Event and call a method in the controller/helper. If the ID matches that of current record, we display a prompt to User that the current record has been updated by someone.

User can then choose to refresh/reload the record for fresh edit.

Alternate Solution:

Every time we load a record, we record the last modified date. While saving, if the last modified date is different than that of when recorded on load, an update was made. User can be notified and he can then act accordingly

Deloitte.

5.11.2 Lightning Events

Events

Lightning Events are useful for communication between different lightning components in DOM. The framework uses event-driven programming. Handlers will be implemented that respond to interface events as they occur. A component registers that it may fire an event in its markup. Events are fired from JavaScript controller actions that are typically triggered by a user interacting with the user interface.

Events are declared by the `aura:event` tag in a `.evt` resource, and they can have one of two types: component or application.

- 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.

Events System

- A notification by the browser regarding an action (If event is fired due to user interaction!). Browser events are handled by client-side JavaScript controllers as below where `c.handleClick` is client-side JavaScript controllers action.
 - `<lightning:button label = "Click Me" onclick = "{!c.handleClick}" />`
- A browser event is not the same as a framework component event or application event.
- Another type of event, known as a system event, is fired automatically by the framework during its lifecycle, such as during component initialization, change of an attribute value, and rendering. Components can handle a system event by registering the event in the component markup.

While implementing interactions between components, evaluate the following approaches (mentioned in the “preferred” order).

1. Component Event → child communicating with parent.
2. `aura:method` → parent communicating with child
3. Service (or) Child component with `aura:method` → for implementing common functions invoked by multiple components.
4. Application Event → siblings communicating with each other in “same container”.

Best Practices - Events

- It is recommended to use a component event instead of an application event, if possible. Component events can only be handled by components above them in the containment hierarchy so their usage is more localized to the components that need to know about them
- Application events allow communication between components that are in separate parts of the application and have no direct containment relationship.
- If there are a large number of handler component instances listening for an event, it may be better to identify a dispatcher component to listen for the event. The dispatcher component can perform some logic to decide which component instances should receive further information and fire another component or application event targeted at those component instances.
- When working with lists, letting events bubble, and registering a single event listener on a parent element instead of a separate event listener on every list item can significantly reduce the number of event listeners in your application, which can have a positive impact on performance.
- Use a single handler to handle multiple scenarios of the same event instead of creating multiple handlers of the same event.

Events Anti-Patterns

- Don't Fire an Event in a Renderer
 - Firing an event in a renderer can cause an infinite rendering loop. For example

```
afterRender: function(cmp, helper) {  
  
this.superAfterRender();  
  
$A.get("e.myns:mycmp").fire();  
  
}
```

- Instead, use the init hook to run a controller action after component construction but before rendering. This is to execute logic on component initialization.
- Don't Use onclick and ontouchend Events
 - You can't use different actions for onclick and ontouchend events in a component. The framework translates touch-tap events into clicks and activates any onclick handlers that are present.
- For more details refer to below link-
 - https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/events_best_practices.htm

Event Propagation

DOM handling patterns

- Bubble: When an event happens on an element, it first runs the handlers on it, then on its parent, then all the way up on other ancestors.
- Capture: Capture is opposite to bubble. First the top most element's handler runs and all the way down to the child which actually triggered the event

Component Events

- Lightning components support Capture and Bubble patterns. Default is bubble phase if you don't specify it when handling the event. Examples provided in the below section.
- Event propagation might be required in scenarios where we have to perform actions one by one.

Application Events

- Application event supports another phase called default phase in addition to capture and bubble.
- The default phase preserves the framework's original handling behavior.

Stopping Event Propagation:

Many times it is not necessary to propagate the events, and needs to be stopped. E.g. Onclick on Menu button, onclick event of Menu Items should not be triggered through capture phase. And Onclick of Menu Item Onclick event of Menu button should not be triggered through Bubble propagation.

In such cases use `stopPropagation()` method in the Event Object.

Pausing Event Propagation for Asynchronous Code Execution:

Use `event.pause()` to pause event handling and propagation until `event.resume()` is called. This flow-control mechanism is useful for any decision that depends on the response from the execution of asynchronous code. For example, you might make a decision about event propagation based on the response from an asynchronous call to native mobile code.

You can call `pause()` or `resume()` in the capture or bubble phases.

Application Event Example

○ Container Component

```
<!--You can test this code by adding <c:aeContainer> to a sample aeWrapper.app application and
navigating to the application.
https://<myDomain>.lightning.force.com/c/aeWrapper.app, where <myDomain> is the name of your custom
Salesforce domain.-->
<!--*****Container Component*****-->
<!--c:aeContainer-->
<aura:component>
    <c:aeNotifier/>
    <c:aeHandler/>
</aura:component>

<!--*****Application Event*****-->
<!--c:aeEvent-->
<aura:event type="APPLICATION">
```

```
<aura:attribute name="message" type="String"/>
</aura:event>
```

○ Notifier Component

```
<!--c:aeNotifier Registers and triggers the Event-->
<aura:component>
  <aura:registerEvent name="appEvent" type="c:aeEvent"/>

  <h1>Simple Application Event Sample</h1>
  <p><lightning:button
    label="Click here to fire an application event"
    onclick="{!c.fireApplicationEvent}" />
  </p>
</aura:component>
<!-- *****-->
/* aeNotifierController.js */
{
  fireApplicationEvent : function(cmp, event) {
    // Get the application event by using the
    // e.<namespace>.<event> syntax
    var appEvent = $A.get("e.c:aeEvent");
    appEvent.setParams({
      "message" : "An application event fired me. " +
        "It all happened so fast. Now, I'm everywhere!" });
    appEvent.fire();
  }
}
```

○ Handler Component

```
<!-- *****Handler Component*****-->
<!--c:aeHandler-->
<aura:component>
  <aura:attribute name="messageFromEvent" type="String"/>
  <aura:attribute name="numEvents" type="Integer" default="0"/>

  <aura:handler event="c:aeEvent" action="{!c.handleApplicationEvent}"/>

  <p>{!v.messageFromEvent}</p>
  <p>Number of events: {!v.numEvents}</p>
</aura:component>
<!-- *****-->
/* aeHandlerController.js */
```

```

{
  handleApplicationEvent : function(cmp, event) {
    var message = event.getParam("message");

    // set the handler attributes based on event data
    cmp.set("v.messageFromEvent", message);
    var numEventsHandled = parseInt(cmp.get("v.numEvents")) + 1;
    cmp.set("v.numEvents", numEventsHandled);
  }
}

```

Component Event Example

```

<!--*****Component Event*****-->
<!--c:compEvent-->
<aura:event type="COMPONENT">
  <!-- Add aura:attribute tags to define event shape.
  One sample attribute here. -->
  <aura:attribute name="message" type="String"/>
</aura:event>
<!--c:ceEvent-->

<!--*****Notifier Component that fires the event*****-->
<aura:registerEvent name="sampleComponentEvent" type="c:compEvent"/>

<!-- controller JS to fire the event -->
var compEvent = cmp.getEvent("sampleComponentEvent");
// Optional: set some data for the event (also known as event shape)
// A parameter's name must match the name attribute
// of one of the event's <aura:attribute> tags
compEvent.setParams({"message" : "myValue" });
compEvent.fire();

<!-- Handler Component that receives the event --->
<aura:handler name="sampleComponentEvent" event="c:compEvent"
  action="{!c.handleComponentEvent}"/>

<!--controller JS to receive the event-->
/* ceHandlerController.js */
{
  handleComponentEvent : function(cmp, event) {

```

```
        var message = event.getParam("message");
    }
}
```

Bubble vs Capture Phase Example

<!--Bubble Event

A component that fires a component event registers that it fires the event by using the <aura:registerEvent> tag.-->

```
<aura:component>
```

```
    <aura:registerEvent name="compEvent" type="c:compEvent" />
```

```
</aura:component>
```

<!--A component handling the event in the bubble phase uses the <aura:handler> tag to assign a handling action in its client-side controller.-->

```
<aura:component>
```

```
    <aura:handler name="compEvent" event="c:compEvent" action="{!c.handleBubbling}"/>
```

```
</aura:component>
```

<!--Capture Event

A component handling the event in the capture phase uses the <aura:handler> tag to assign a handling action in its client-side controller.-->

```
<aura:component>
```

```
    <aura:handler name="compEvent" event="c:compEvent" action="{!c.handleCapture}"
        phase="capture" />
```

```
</aura:component>
```

<!-- If we don't mention the phase attribute value, it will be defaulted to "bubble" -->

5.11.3 Lightning Variable Binding

Types of Data Binding

`{!expression}` (Bound Expressions) : Bound Expression is represented as `{!v.boundStr}`. Whenever the value of the string is changed, this expression will reflect the change and also affect the components where it is used, we can say the value change dynamically through this expression.

Data updates in either component are reflected through bidirectional data binding in both components. Similarly, change handlers are triggered in both the parent and child components.

Bi-directional data binding is expensive for performance and it can create hard-to-debug errors due to the propagation of data changes through nested components. Salesforce recommends using the `{#expression}` syntax instead when you pass an expression from a parent component to a child component unless you require bi-directional data binding.

`{#expression}` (Unbound Expressions) : Unbound Expression is represented as `{#v.unboundStr}`. Whenever the value of the string is changed, this expression will not reflect the change, we can say the value remains static through this expression.

Data updates behave as you would expect in JavaScript. Primitives, such as `String`, are passed by value, and data updates for the expression in the parent and child are decoupled.

Objects, such as `Array` or `Map`, are passed by reference, so changes to the data in the child propagate to the parent. However, change handlers in the parent aren't notified. The same behavior applies for changes in the parent propagating to the child.

Data binding between components

```
<!-- Bound Expression Example {!v.var}-->
<!--In this example if the value of "parentAttr" is changed, this will be reflected in Parent-->
<!--c:parent-->
<aura:component>
    <aura:attribute name="parentAttr" type="String" default="parent attribute"/>
    <!-- Instantiate the child component -->
    <c:child childAttr="{!v.parentAttr}" />
</aura:component>

<!-- Unbound Expression Example {#v.var}-->
<!-- In this example, if the value of "parentAttr" is changed by child component, that will not be reflected in
parent-->
<!--c:parentExpr-->
<aura:component>
```

```
<aura:attribute name="parentAttr" type="String" default="parent attribute"/>

<!-- Instantiate the child component -->
<c:childExpr childAttr="{#v.parentAttr}" />

<p>parentExpr parentAttr: {!v.parentAttr}</p>
<p><lightning:button label="Update parentAttr"
    onclick="{!c.updateParentAttr}" /></p>
</aura:component>
```

Deloitte.

5.11.4 JavaScript

Best Practices

- There are two different kinds of equality operators ('===' and '=='). It is a good practice to use '===' instead of '==' , because
 - the == operator will compare for equality after doing any necessary type conversions.
 - The ===operator will not do the conversion, so if two values are not the same type === will simply return false
- All variables used in a function should be declared as local variables.
 - Local variable must be declared with 'var' keyword; otherwise they will become global variables.
- JavaScript has additional type which is called "undefined".
 - Undefined means a variable has been declared but has not yet been assigned a value.
 - On the other hand, null is an assignment value. It can be assigned to a variable as a representation of no value.
- All JavaScript logic should be present in Helper.js and only event handlers should in Controller.js.
- Manipulate UI through "attributes" always and avoid DOM hacking as much as possible.
- Pay more attention to Execution Sequence while updating attribute values. For example, if parent component's attribute (A) is sent to child component as bound expression and the visibility of child component is controlled by another parent component's attribute (B), then following should be the execution sequence: (1). Update the attribute B, (2). Update the attribute A to make the child component visible.
- Lazy load the data from server used by the components. For example, the data used by hidden tab, component hidden > 50% of the times the page is accessed.
- Combine multiple server calls into single call (wherever possible).
- Try catch block in ALL helper methods (irrespective of the no. of lines in the method).
- Use &A.getCallBack to wrap any code written within functions (window functions like setTimeout, setInterval, etc..) that modify component outside the component's rendering cycle. Refer this link for more information.
- Cache the length of array variables (wherever applicable) instead of evaluating the array length every time (in for loop condition). Same approach can be exercised for accessing the individual elements of a large array multiple times in the same loop/block.
- Usage of the following: localStorage, sessionStorage and browser cookies should be after careful consideration of the behavior across devices.
- Avoid the usage of third-party libraries and use plain JavaScript for any browser side manipulation. In unavoidable circumstances, use "ltngr:require" to load such scripts and "afterScriptsLoaded" attribute to run any script post loading of the library.
- Load all script resources (e.g. jQuery) from the application origin as cached Static Resources.

5.11.5 CSS

Cascading Style Sheet Best Practices

- Naming conventions: In general, use class-names-like-this, id-names-like-this.
- Avoid using inline styling elements and always prefer class attribute.
- Use SLDS for styling lightning components.
- Usage of default tokens instead of hard-coded color values if the color is common across components.
- Avoid using unit! Values for 0. Usual: margin-left: 0px; top: 0rem; Preferred: margin-left: 0; top: 0;
- Avoid using !important. Use specificity instead.
- Use single quotation marks for attribute selectors and property values.
- Don't use margin-top. Vertical margins [collapse](#). Always prefer padding-top or margin-bottom on preceding elements.
- Avoid using units when value of an attribute is 0. i.e.: Don't use 0px / 0em / 0%. Use 0.
- Check out common-style.css for commonly used styling classes across the portal to avoid unnecessary repetition of CSS.
- Refer [Google's style guide](#).
- Refer [this article](#) for CSS styling for PDFs.
- Use design token when necessary in lightning component. Refer: https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/tokens_intro.htm
- Components must be set to 100% width because they can be moved to different locations on a Lightning page, components must not have a specific width nor a left or right margin. Components should take up 100% of whatever container they display in. Adding a left or right margin changes the width of a component and can break the layout of the page.
- Don't remove HTML elements from the flow of the document. Some CSS rules remove the HTML element from the flow of the document. For example: float: left; float: right; position: absolute; position: fixed; Because they can be moved to different locations on the page as well as used on different pages entirely, components must rely on the normal document flow. Using floats and absolute or fixed positions breaks the layout of the page the component is on. Even if they don't break the layout of the page you're looking at, they will break the layout of somepage the component can be put on.
- Child elements shouldn't be styled to be larger than the root element: The Lightning page maintains consistent spacing between components, and can't do that if child elements are larger than the root element.
- Vendor prefixes, such as —moz- and —webkit- among many others, are automatically added in Lightning. For example: there is no need to write border radius for -webkit and -moz
 - .class {

-webkit-border-radius: 2px;

-moz-border-radius: 2px;

border-radius: 2px;


```
}
```

- Salesforce recommends that you name the Lightning Design System archive static resource using the name format SLDS####, where #### is the Lightning Design System version number (for example, SLDS252). This lets you have multiple versions of the Lightning Design System installed, and manage version usage in your components. To reference an external CSS resource that you've uploaded as a static resource, use a tag in your .cmp or .app markup.

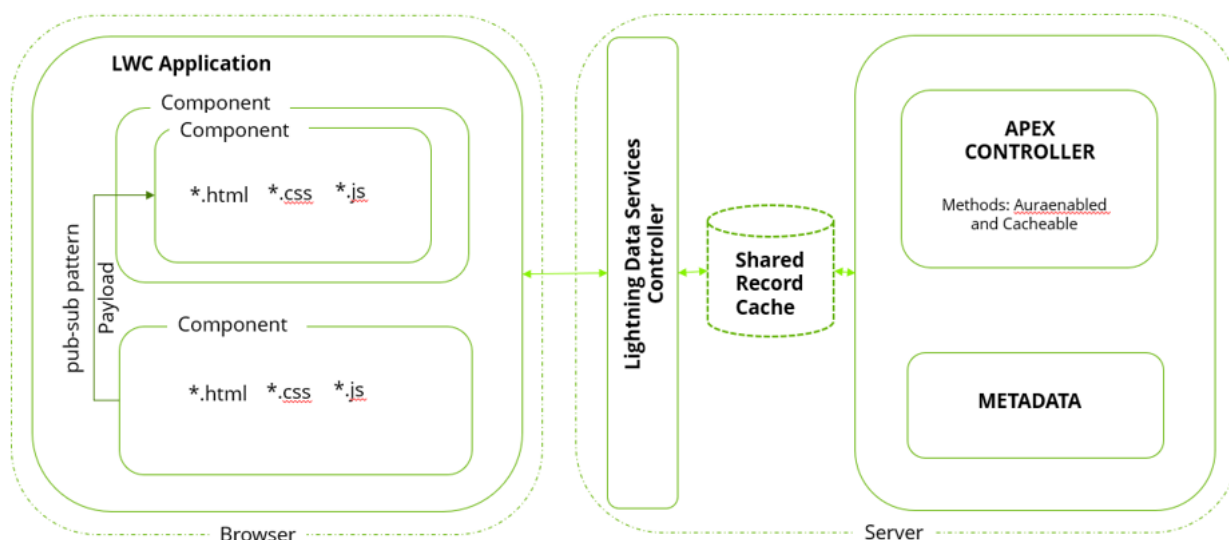
Deloitte.

5.12 Lightning Web Components

Lightning Web Components

Lightning Web Components is a new programming model for building Lightning components. These are custom HTML elements built using HTML and modern JavaScript.

Lightning Web Components uses core Web Components standards and provides only what's necessary to perform well in browsers supported by Salesforce. Because it's built on code that runs natively in browsers. Lightning Web Component is lightweight and delivers exceptional performance. Standard JavaScript and HTML is used for building the component.



Lightning Web Components Architecture

Aura Programming Model	Lightning Web Components Programming Model
<p>Custom Component Model</p> <p>(Definition of properties that components must satisfy, methods and mechanisms for the composition of components)</p>	<p>Web Components</p> <p>(Reusable custom elements, with their functionality encapsulated away from the rest of your code)</p>

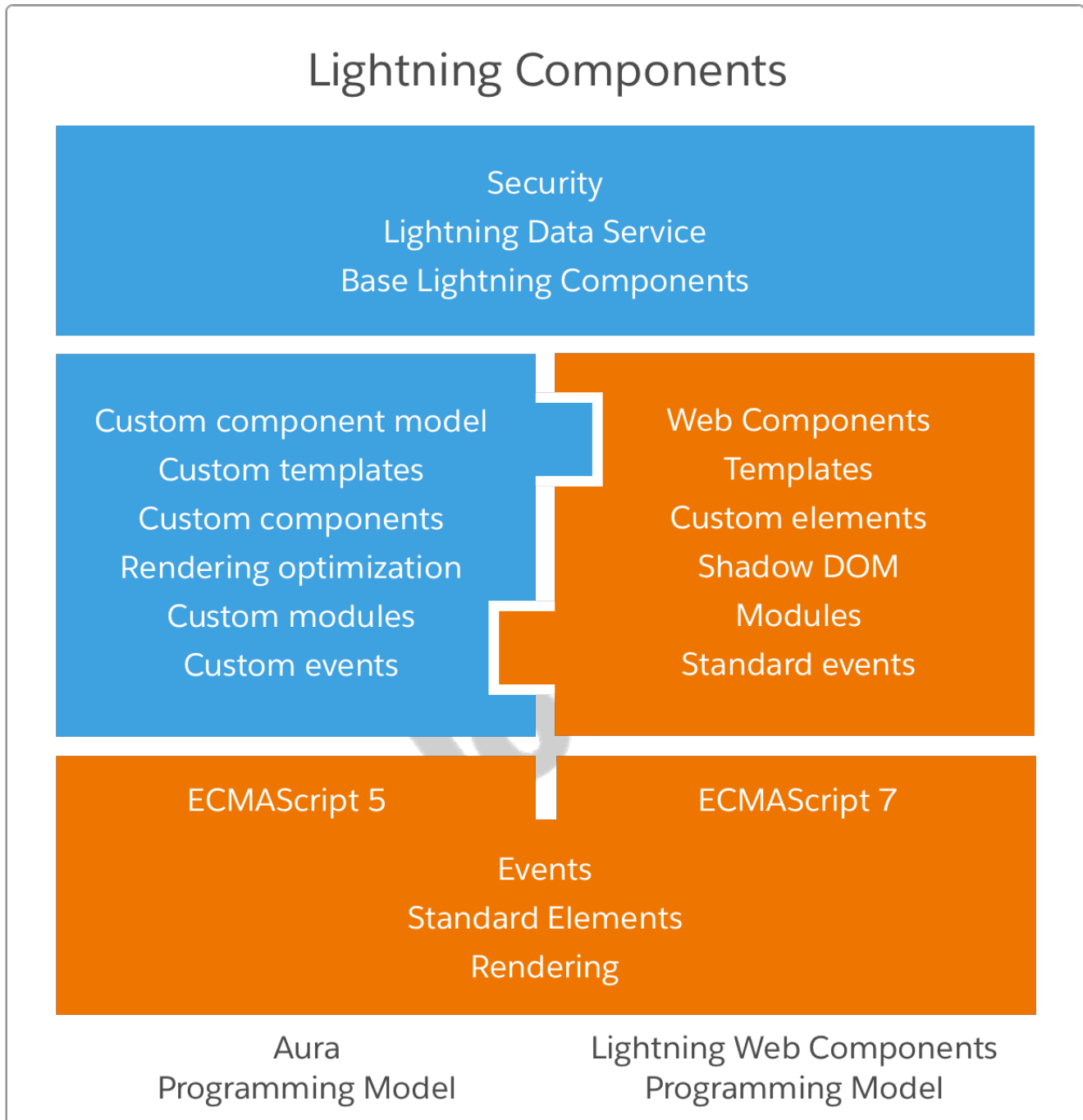
Facets (Facet is defined as an attribute of the component that has a type of Aura.Component[])	Slots (A slot is a placeholder for markup that a parent component passes into a component's body)
Custom Template	Templates and Modules (Template holds HTML code which is rendered by javascript while Module is javascript file which is imported in another js file to use)
Custom Components	Custom DOM Elements
Rendering Optimization	Shadow DOM (Encapsulation of DOM)
Custom Modules	Modules
ECMAScript 5 (Released 2009)	ECMAScript 7 (Released 2016)
Two-way Binding	Restricted Parent to Child Binding
Single Apex Controller	Methods from multiple Classes can be imported
Supported in Salesforce Developer Console	Not Supported in Salesforce Developer Console
Custom Events	Custom Events

(Component and Application Event)	(Standard CustomEvent DOM Object and PubSub Module)
Naming Convention (Components and Attributes are referred as it is in the markup)	Naming Convention (Camel case components and attributes map to kebab case in the markup)
Rendering (init, render, destroy)	Rendering (connectedCallback, renderedCallback, disconnectedCallback)
Standard Events	
Standard Elements	
Security	
Lightning Data Services	
Base Lightning Components	

Coexistence and interoperability

With the addition of Lightning Web Components, there are now two ways to build Lightning components:

- Aura Components, leveraging its own component model, templates, and modular development programming model.
- Lightning Web Components, built on top of the web standards breakthroughs of the last five years: web components, custom elements, Shadow DOM, etc.



Aura components and Lightning web components can coexist and interoperate, and they share the same high level services:

- Aura components and Lightning web components can coexist on the same page
- Aura components can include Lightning web components
- Aura components and Lightning web components share the same base Lightning components. Base Lightning components were already implemented as Lightning web components.

- Aura components and Lightning web components share the same underlying services (Lightning Data Service, User Interface API, etc.).

If you are already developing Lightning components with the Aura programming model, you can continue to do so. Your Aura components will continue to work as before. You can build new components with Aura or Lightning Web Components. Your Aura and Lightning Web Components can coexist and interoperate. Over time, you can consider migrating your Aura Components to Lightning Web Components, starting with the components that would benefit the most from the performance benefits of Lightning Web Components.

If you are new to developing on Lightning or if you are starting a new project, we recommend using the Lightning Web Components programming model.

LWC and Aura – What’s changing and What’s remaining

Aura Programming Model	Lightning Web Components Programming Model
Custom Component Model (Definition of properties that components must satisfy, methods and mechanisms for the composition of components)	Web Components (Reusable custom elements, with their functionality encapsulated away from the rest of your code)
Facets (Facet is defined as an attribute of the component that has a type of <code>Aura.Component[]</code>)	Slots (A slot is a placeholder for markup that a parent component passes into a component’s body)
Custom Template	Templates and Modules (Template holds HTML code which is rendered by javascript while Module is javascript file which is imported in another js file to use)

Custom Components	Custom DOM Elements
Rendering Optimization	Shadow DOM (Encapsulation of DOM)
Custom Modules	Modules
ECMAScript 5 (Released 2009)	ECMAScript 7 (Released 2016)
Two-way Binding	One-Way Binding
Single Apex Controller	Methods from multiple Classes can be imported
Supported in Salesforce Developer Console	Not Supported in Salesforce Developer Console
Custom Events (Component and Application Event)	Custom Events (Standard CustomEvent DOM Object and PubSub Module)
Naming Convention (Components and Attributes are referred as it is in the markup)	Naming Convention (Camel case components and attributes map to kebab case in the markup)
Rendering (init, render, destroy)	Rendering (connectedCallback, renderedCallback, disconnectedCallback)

Standard Events
Standard Elements
Security
Lightning Data Services

LWC and Aura – File structure differences

Resource	Aura File	LWC File	Useful Links
Markup	sample.cmp	sample.html	LWC HTML File
Controller	sampleController.js	sample.js	LWC JavaScript File
Helper	sampleHelper.js		Lightning Component Helpers
Renderer	sampleRenderer.js		Lightning Component Renderers
CSS	sample.css	sample.css	LWC CSS File
Documentation	sample.auradoc	Not currently available	Lightning Component Documentation
Design	sample.design	sample.js-meta.xml	LWC Config File

SVG	sample.svg	Not currently available	Lightning Component Custom Icons
-----	------------	-------------------------	--

LWC and Aura – Syntax differences of common building blocks

Building Blocks	Aura Components	Lighting Web Components
Attributes and properties	<aura:attribute name="myproperty">	@api myproperty @track properties
Expression Syntax	{!v.myattribute}	{myattribute}
Event Handling	Markup tags like <aura:handler> and <aura:register> required	Const customEvt = new CustomEvent(); this.dispatchEvent(customEvt);
CSS Syntax	.THIS.myclass {}	.myclass()
Facets	aura:facet	slot
Base Components Syntax	Components are referred as it is in the markup and namespace is separated by colon : For ex - lightning:layoutItem	Camel case components map to kebab-case and namespace is separated by hyphen - For Ex -lightning-layout-item

LWC Supported Experiences and Tools

Name	Definition	LWC Support

Lightning Experience	Lightning Experience is a modern user interface that helps your sales reps sell faster and your service reps support customers more productively	Lightning Experience supports Lightning Web Components
Salesforce App	A salesforce application is a group of tabs that work as a unit to provide functionality	Lightning Web Components can be accessible through Salesforce Apps
Lightning Communities	It gives a way to leverage the power CRM, enabling customers, partners, and employees to access your Salesforce data and business processes, in an engaging, branded experience	Lightning Communities support Lightning Web Components to be added via drag-drop or as a stand-alone
Lightning App Builder	The Lightning App Builder is a one-stop shop for configuring Lightning apps	Lightning Web Components can be accessed through Apps built leveraging Lightning App Builder
Community Builder	Community Builder lets you quickly create and style your custom community to match your organization's branding	Lightning Communities support Lightning Web Components to be added via drag-drop
First-Generation Managed Packages	A managed package is a bundle of components that make up an application or piece of functionality	

Second-Generation Managed Packages	It allows customers and system integrators to create packages in a source-driven development environment	
Unlocked Packages	Unlocked packages give customers and system integrators a means to organize their metadata into a package and then deploy the metadata (via packages) to different orgs	It can be added to packages to be used in other orgs
Unmanaged Packages	Unmanaged packages are typically used to distribute open-source projects or application templates to provide developers with the basic building blocks for an application	
Change Sets	Change sets are used to send customizations from one Salesforce org to another	It can be deployed through change-set to other orgs
Metadata API - LightningComponentBundle	Represents a Lightning web component bundle. A bundle contains Lightning web component resources	Metadata API supports deployments of Lightning Web Components using LightningComponentBundle

Tooling API - LightningComponentBundle, LightningComponentResource	<p>Represents a Lightning web component bundle. A bundle contains a Lightning web component and its related resources</p> <p>Represents a Lightning web component resource, such as HTML markup, JavaScript code, a CSS file, an SVG resource, or an XML configuration file</p>	Metadata API supports deployments of Lightning Web Components using LightningComponentBundle or LightningComponentResource
--	---	--

LWC Unsupported Experiences and Tools

Name	Description
Lightning Out	Lightning Out is a feature that extends Lightning Apps and acts as a bridge to surface Lightning Components in any remote web container
Lightning Components for Visualforce	Lightning Components for Visualforce is based on Lightning Out, a powerful and flexible feature that lets you embed Aura components into almost any web page
Standalone Apps	An app is a special top-level component whose markup is in a .app resource
Salesforce Console (Navigation Item API, Workspace API, UtilityBar API)	Salesforce Console apps are a tab-based workspace suited for fast-paced work environments. Manage multiple records on a single screen and reduce time spent clicking and scrolling to quickly find, update, and create records

Utility Bars	A utility is broadly defined as a single-column Lightning page. Salesforce provides you with several ready-to-use utilities, such as Recent Items, History, and Notes
URL Addressable Tabs	This interface is used to indicate that a component can be directly navigated to through a URL
Flows	A flow is an application that can execute logic, interact with the Salesforce database, call Apex classes, and collect data from users. You can build flows by using the Cloud Flow Designer.
Snap-ins Chat	The Snap-ins Chat component allows users to request a chat with a support agent.
Lightning for Gmail, Outlook Integration	One can work directly from Microsoft or Google applications like Microsoft Outlook®, Gmail™, or Google Calendar™
EMP API, Conversation Toolkit API, Omni Toolkit API, Quick Action API	Some APIs provided by Salesforce for interaction like Streaming, Console integration APIs for Live Agent etc.
Standard Action Overrides, Custom Actions, Global Actions, List View Actions, Related List View Actions	Action overrides are used when your business model requires a more customized user experience than the Salesforce standard page provides
Chatter Extensions	Represents the metadata used to describe a Rich Publisher App that's integrated with the Chatter publisher

5.12.1 General Guidelines

Naming Convention

- Name components as <ProjectInitials> MyComponent so that they can be used as <c-<ProjectInitials>-my-component>.
- The folder and its files must have the same name, including capitalization and underscore

Component Folder

myComponent

- myComponent.html
- myComponent.js
- myComponent.js-meta.xml
- myComponent.css
- myComponent.svg

- Must begin with a lowercase letter
- Must contain only alphanumeric or underscore characters
- Must be unique in the namespace
- Can't include whitespace
- Can't end with an underscore
- Can't contain two consecutive underscores
- Can't contain a hyphen (dash)

- Prefer property names in JavaScript in camel case while HTML attribute names are in kebab case (dash-separated) to match HTML standards
- Don't start a property name with these characters like

1. on (for example, onClick)
2. aria (for example, ariaDescribedby)
3. data (for example, dataProperty)
4. Don't use the reserved keywords like Slot, Part & Is as a variable name

HTML Templates & JS

- Always use out of the box UI components provided by lightning web components when possible.
- Always use Lightning Design system when possible and overwrite it when needed.
- Never use !important in your CSS. There will always be another use case where you may want to override your CSS with another rule. Using !important makes it difficult for one rule to be overwritten by another. Use specificity to override CSS styles instead.
- If you want nested template renders, change the value of isTrueTemplate to true in the component's JavaScript class.
- Prefer wiring to a property. This best practice applies to @wire in general (not just to wiring Apex methods). It makes the code less verbose and defines a predictable response pattern to work with myProperty.data and myProperty.error.
- Composition is useful to compose apps and components from a set of smaller components to make the code more reusable and maintainable.

- To ensure CSS encapsulation, each element has an extra attribute, which also increases rendering time.
- Use `this.querySelector()` and `this.querySelectorAll()` instead of `this.template.querySelector()` or `this.template.querySelectorAll()`. Template is used for the access to the elements rendered by the component. The elements rendered by the DOM is not accessible. However `this.querySelector()` / `this.querySelectorAll()` has access to elements rendered by DOM.
- Send only primitives, or to copy data to a new object before adding it to the detail property. Copying the data to a new object ensures that you're sending only the data you want, and that the receiver can't mutate your data.
- If an event uses Bubbling and composed events configuration, the event type becomes part of the component's public API. It also forces the consuming component and all of its ancestors to include the event as part of their APIs.
- Use events configured with `bubbles: false` and `composed: false` because they're the least disruptive. These events don't bubble up through the DOM and don't cross the shadow boundary.
- Browser support - Ensure your web page works & functions fine as per requirement on the latest versions of Google Chrome, Mozilla Firefox & Microsoft Edge.
- Accessibility - Ensure your web page is ADA Compliant by running tools like AInspector on it after it is developed. Ensure there are no issues within the custom Lightning components on your page. Any ADA issues highlighted by the extension on any out-of-the-box Salesforce elements can be ignored.
- Expressions are not allowed in Lightning Web Component template. Refer to HTML Template Directives documentation for details. The engine doesn't allow computed expressions. To compute the value use a getter in the JavaScript class of expression.

```

30 | Example :
31
32 <template if:true={index % 5 == 0}><br></template> <!--Not Allowed -->
33
34 <!-- Instead use getter -->
35
36 <!-- HTML: -->
37
38 <template if:true={ Mod5 }><br></template>
39
40
41 <!-- JS: -->
42
43 import {LightningElement, api} from 'lwc';
44
45 export default class AccountListItem extends LightningElement {
46
47     @api index;
48     @api item;
49
50     get isMod5() {
51         return this.index % 5 == 0;
52     }
53
54 }

```

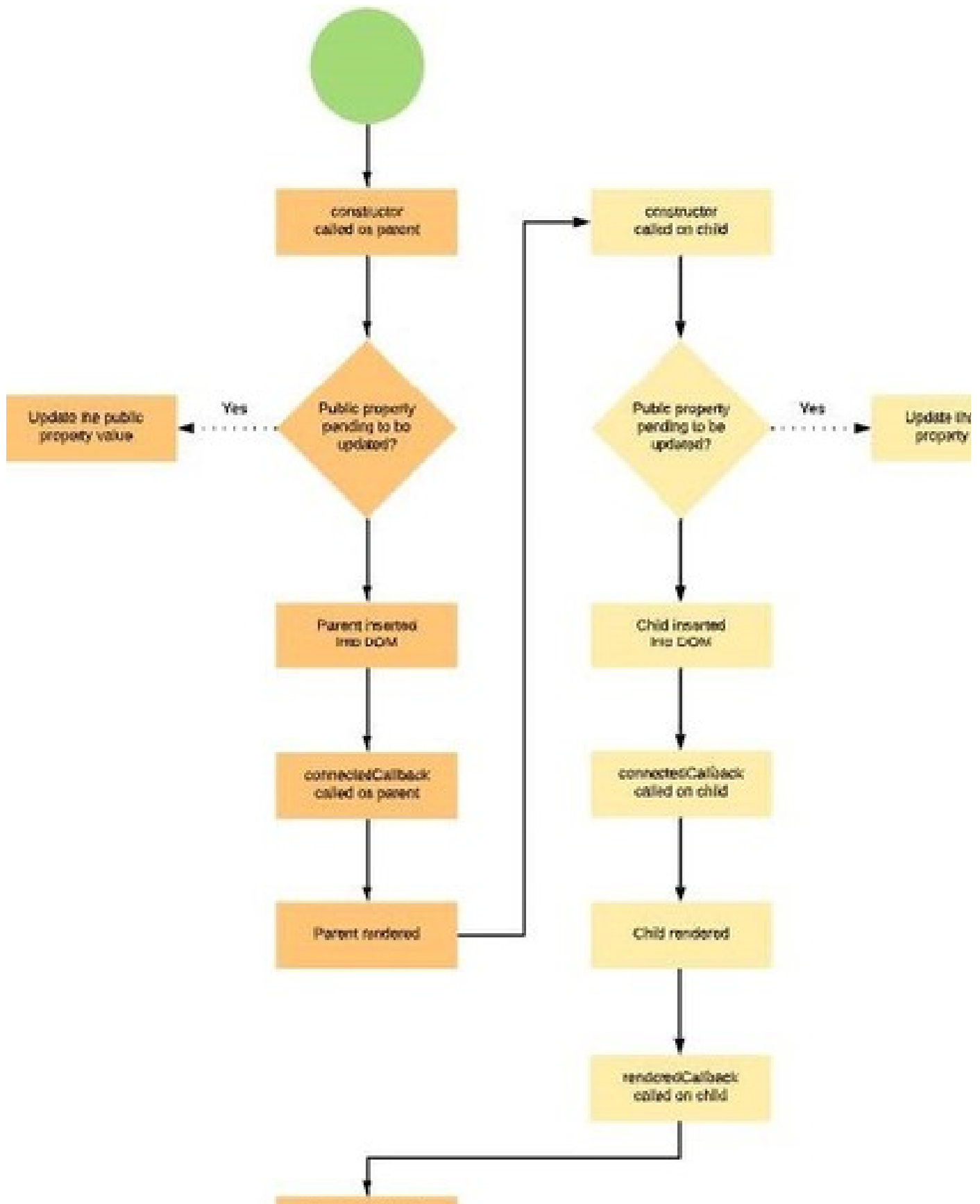
- Prefer standard promises over async. More here:

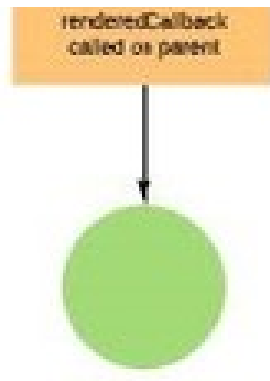
https://developer.salesforce.com/docs/componentlibrary/documentation/lwc/get_started_supported_javascript

- Ensure that, this.hasRendered convention for code that should only be executed once within that hook.
- Avoid misuse of @track properties. Generally, a private property would suffice if the property never changes after component init.
- Do not attempt to write the @api property in child component as it is read only.
- Enforce Single Responsibility Principle when composing component controller code. This ensures that code stays lean, efficient, maintainable, scalable, and easy to debug.

https://en.wikipedia.org/wiki/Single_responsibility_principle

LWC Life Cycle





Similar to Aura components, Lightning web components have a lifecycle managed by the framework.

- In the creation phase, create the components, insert them into the DOM, and finally render them.
- In the destroy phase, remove them from the DOM.

constructor()

- Hook that fires when a component instance is created.
- `super()` must be used in the first line.
- Component Properties are not available to be accessed in `constructor()` hook.
- It flows from parent to child.
- Reference:
https://developer.salesforce.com/docs/componentlibrary/documentation/en/lwc/lwc.create_lifecycle_hooks

Best Practices related to implementing Constructor() hooks

- It is possible to set properties of component in constructor but it is recommended to use getter setter.
- Don't use a return statement inside the constructor body, unless it is a simple `earlyreturn`.

connectedCallback()

- Fires when a component is inserted into the DOM.
- The component properties will be ready, but the child elements won't be yet.
- It can fire more than once.
- It flows from parent to child.

Best practices in connectedCallback()

- Use `ConnectedCallback` hook instead of `Constructor` for `dispatchEvent` / `registerListener` / `navigation service` / `show toast`.
- Child elements cannot be accessed the callbacks as the child is not loaded on DOM yet.
- We recommend use of `this.template.element` to access the host elements.
- The `connectedCallback()` hook can fire more than once. To prevent the code to run one time we recommend make use of a Boolean in code.

- Use `connectedCallback()` to interact with a component's environment. For Example – Establish communication with the current document or container and coordinate behavior with the environment or Perform initialization tasks, such as fetch data, set up caches, or listen for events (such as publish-subscribe events)

`disconnectedCallback()`

- Fires when a component is removed from the DOM.
- Rest practices are same as `connectedCallback()`

Example:

```
1  // test.js
2  import { LightningElement } from 'lwc';
3  export default class Test extends LightningElement
4  connectedCallback() {
5    window.addEventListener('test', this.handleTest);
6  }
7  disconnectedCallback() {
8    window.removeEventListener('test', this.handleTest);
9  }
10 handleTest = () => {};
11 }
```

`renderedCallback()`

- Fires when a component rendering is done.
- It can fire more than once.
- It flows from child to parent.

Best practices in `renderedCallback()`

- Usage of `RenderedCallback` with `Wire` and `track` is not recommended as it will result in an infinite loop.
- `renderedCallback()` updates an `@wire` config change and the `@wire` provisioning triggers a render.
- `renderedCallback()` updates an `@track` property, which triggers a render.
- Use `renderedCallback()` to interact with a component's UI. For example, use it to:

- Compute node sizing. Perform tasks not covered by our template declarative syntax, such as add a listener to a non-standard event from a component's child.
- `renderedCallback` is invoked everytime the component renders. So if we use to set properties the callback is invoked multiple times.

However we can smartly make use of a Boolean to avoid multiple call.

Example:

```

1  import { LightningElement, track } from 'lwc';
2  export default class RenderedCallbackInLWC extends LightningElement {
3    @track properties;
4    @track hasRendered = true;
5    renderedCallback() {
6      //guarding code inside the renderedCallback using boolean property
7      if (hasRendered) {
8        this.properties = 'set by renderedCallback';
9        console.log('properties ' + this.properties);
10       hasRendered = false;
11     }
12   }
13   handleClick() {
14     this.properties = 'set by buttonClick';
15   }
16 }

```

`errorCallback(error, stack)`

- Captures errors that may happen in all the descendent components lifecycle hooks.
- This is used to create a boundary error template that can capture the error and can be used to handle rendering errors in the descendent components

Example:

HTML

```
<template>
  <template if:true={error}>
    <p>Error: {error}</p>
    <p>Info: {stack}</p>
  </template>
  <template if:false={error}>
    <c-child></c-child>
  </template>
</template>
```

JS

```
import { LightningElement, track } from 'lwc';
export default class ErrorBoundary extends LightningElement {
  @track error;
  @track stack;
  errorCallback(error, stack) {
    this.error = error;
    this.stack = stack;
  }
}
```

Cross Origin Custom Elements

The cross-origin custom element manages APIs on the outside prototype object using the following new methods of the custom element global scope:

1. `setProperty(name, property_descriptor)` - mirroring `Object.defineProperty` for property creation
2. `setProperties(multiple_property_descriptors)` - mirroring `Object.defineProperties` for convenience
3. `deleteProperty(name)` - delete name;
4. `getPropertyDescriptor(name)` - `Object.getOwnPropertyDescriptor` to test for existence or get the descriptor

Create a interface for event handling lifecycle management instead of repeating the code every LWC


```

1 interface CustomElementPrototypeGlobalScope: EventTarget {
2   readonly attribute CustomElementPrototypeGlobalScope self; // self-reference (like in Workers)
3   readonly attribute WorkerLocation location; // This is defined in workers (gives href, etc.)
4   attribute onerror; // For catching general script problems
5   // Events for handling instance lifecycle management
6   attribute oninstancecreated; // see custom elements: createdCallback (target is a CustomElementInstance)
7   attribute oninstanceattached;
8   attribute oninstancedetached;
9 };
10
11 // To represent the element instances...
12 interface CustomElementInstance: EventTarget {
13   // Attribute change handling
14   readonly attribute AttrReadOnly[] attributes;
15   attribute onattrset; // Event who's target is the AttrReadOnly...
16   attribute onattrremoved; // ditto
17   attribute onattrchanged; // ditto
18   // Borrowed from ShadowDOM (extensions to Element interface):
19   ShadowRoot createShadowRoot();
20   readonly attribute ShadowRoot ? shadowRoot;
21 };
22
23 Copied from DOM4, but without a read / write 'value':
24 interface AttrReadOnly {
25   readonly attribute DOMString localName;
26   readonly attribute DOMString value;
27   readonly attribute DOMString name;
28   readonly attribute DOMString ? namespaceURI;
29   readonly attribute DOMString ? prefix;
30   readonly attribute boolean specified; // useless; always returns true
31 };

```

- Use ES6 Script Modules for Componentized code and for better dependency management
- HTML Modules are imported using the same Import statement

```

1 <script type="module">
2   import {content} from "import.html"
3   document.body.appendChild(content);
4 </script>

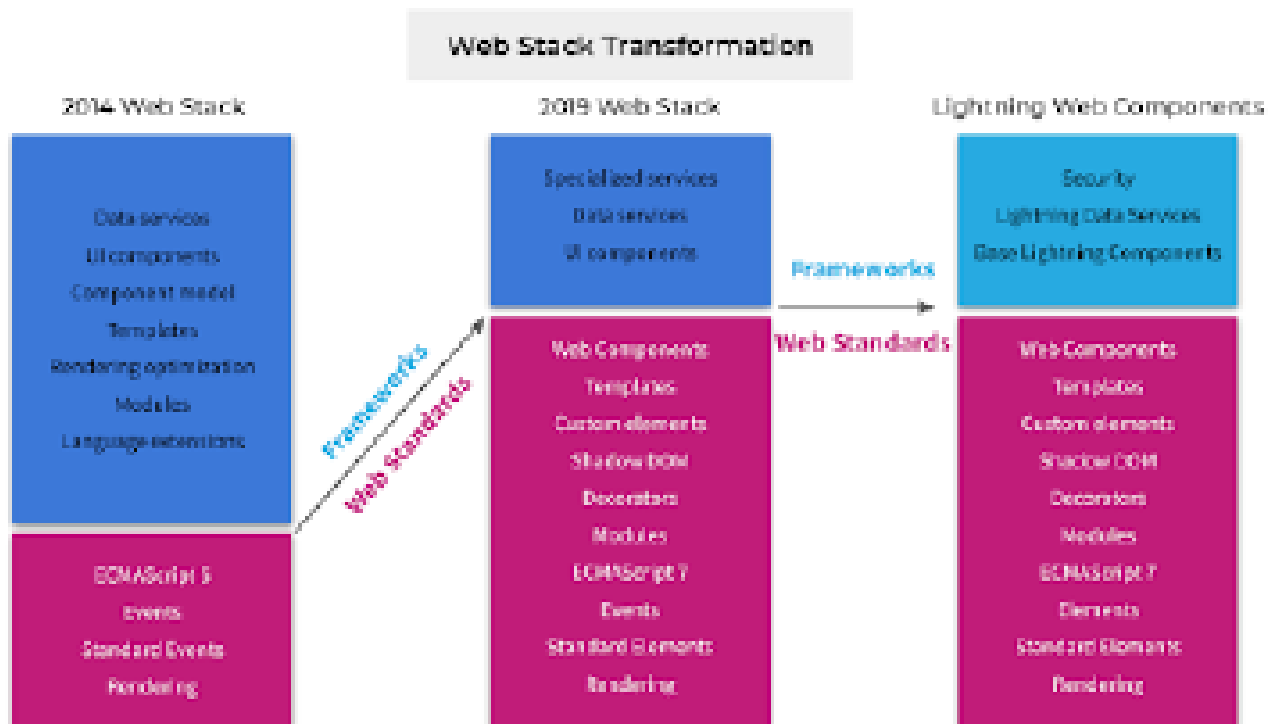
```

- Access the declarative content from within HTML Module

```
1 <template id="myCustomElementTemplate">
2   <div>Custom element shadow tree content...</div>
3 </template>
4
5 <script type="module">
6   let importDoc = import.meta.document;
7
8   class myCustomElement extends HTMLElement {
9     constructor() {
10       super();
11       let shadowRoot = this.attachShadow({ mode: "open" });
12       let template = importDoc.getElementById("myCustomElementTemplate");
13       shadowRoot.appendChild(template.content.cloneNode(true));
14     }
15   }
16
17   window.customElements.define("myCustomElement", myCustomElement);
18 </script>
```

Advantages Of LWC

- Improved performance of the component as most of the code is recognized by the native web browser engine and web stack
- Ability to compose applications using smaller chunks of code since the crucial elements that are required to create a component is part of the native web browser engine and web stack
- Increase in the robustness of the applications built using LWCs as they are inclusive of the said modern web standards.
- Parallel interoperability and feasibility to use both Lightning Web Components and Aura components together in the applications with no visible differentiation to the end-users
- Languages like React, Angular and the Lightning Components Framework which use JavaScript are now part of the modern Web Stack. Consequently, the Web Stack no longer requires an additional mid-layer which used to be a challenge for the developers working with Aura as it impacted the speed and performance of the applications.
- Because of the no added abstraction layer, LWC is likely to render faster than aura components since performance is important to deliverability.



- Aura-based Lightning components are built using both HTML and JavaScript, but LWC is built directly on the Web stack.
- Web components can easily interact with the Aura component and can handle the events of one another
- A developer who works on LWC is not only using coding skills on a particular framework but in other frameworks like React or Angular, which are based on the Web Stack.
- In addition to the latest features, LWC still embraces the features like the Security, Lightning Data Service, and Base lightning components from Aura.
- Creating an LWC is fast as it no longer requires the user to download the JavaScript and wait for the engine to compile it before rendering the component.
- LWC can be embedded inside Aura Components, but Aura Components cannot be embedded inside LWC.

5.12.2 Work with Salesforce Data

Lightning Data Service

The cross-origin custom element manages APIs on the outside prototype object using the following new methods: the custom element global scope:

1. `setProperty(name, property_descriptor)` - mirroring `Object.defineProperty` for property creation
2. `setProperties(multiple_property_descriptors)` - mirroring `Object.defineProperties` for convenience
3. `deleteProperty(name)` - delete name;
4. `getPropertyDescriptor(name)` - `Object.getOwnPropertyDescriptor` to test for existence or get the descriptor

Create a interface for event handling lifecycle management instead of repeating the code every LWC

```

1  interface CustomElementPrototypeGlobalScope: EventTarget {
2    readonly attribute CustomElementPrototypeGlobalScope self; // self-reference (like in Workers)
3    readonly attribute WorkerLocation location; // This is defined in workers (gives href, etc.)
4    attribute onerror; // For catching general script problems
5    // Events for handling instance lifecycle management
6    attribute oninstancecreated; // see custom elements: createdCallback (target is a CustomElementInstance)
7    attribute oninstanceattached;
8    attribute oninstancedetached;
9  };
10
11 // To represent the element instances...
12 interface CustomElementInstance: EventTarget {
13   // Attribute change handling
14   readonly attribute AttrReadOnly[] attributes;
15   attribute onattrset; // Event who's target is the AttrReadOnly...
16   attribute onattrremoved; // ditto
17   attribute onattrchanged; // ditto
18   // Borrowed from ShadowDOM (extensions to Element interface):
19   ShadowRoot createShadowRoot();
20   readonly attribute ShadowRoot ? shadowRoot;
21 };
22
23 Copied from DOM4, but without a read / write 'value':
24 interface AttrReadOnly {
25   readonly attribute DOMString localName;
26   readonly attribute DOMString value;
27   readonly attribute DOMString name;
28   readonly attribute DOMString ? namespaceURI;
29   readonly attribute DOMString ? prefix;
30   readonly attribute boolean specified; // useless; always returns true
31 };

```

- Use ES6 Script Modules for Componentized code and for better dependency management
- HTML Modules are imported using the same Import statement

```
1 <script type="module">
2   ... import {content} from "import.html"
3   ... document.body.appendChild(content);
4 </script>
```

- Access the declarative content from within HTML Module

```
1 <template id="myCustomElementTemplate">
2   <div>Custom element shadow tree content...</div>
3 </template>
4
5 <script type="module">
6   import.meta.document;
7
8   class myCustomElement extends HTMLElement {
9     constructor() {
10       super();
11       let shadowRoot = this.attachShadow({ mode: "open" });
12       let template = importDoc.getElementById("myCustomElementTemplate");
13       shadowRoot.appendChild(template.content.cloneNode(true));
14     }
15   }
16
17   window.customElements.define("myCustomElement", myCustomElement);
18 </script>
```

Always prefer the Lightning Data Service functions (`createRecord`, `updateRecord`, and `deleteRecord`) over invoking Apex methods to create, update and delete single records.

Benefits: This ensures, you do not have to write Apex code, and, most importantly, the Lightning Data Service updates the local record cache, ensuring consistency across components showing the same data.

- Always use lightning-record-form wherever possible
- If we need more control over the layout, want to handle events on individual input fields, or need to execute pre-submission logic: consider lightning-record-view-form or lightningrecord-edit-form.
- If we need even more control over the UI, or if we need to access data without a UI: Use `@wire(getRecord)`
- Use LDS as an alternative to apex methods which allows you to retrieve single record, avoid write test classes. LDS updates the local record cache, ensuring consistency across components showing the same data.

Lightning/uiRecordApi module adapters

This module includes wire adapters to record data and get default values to create records. It also includes JavaScript APIs to create, delete, update, and refresh records. Methods provided by UIRecord API eliminate the

need for implementing custom Apex classes. Explore the usage of methods provided by uiRecordApi before implementing Custom Apex classes

Example:

```
{ updateRecord } from 'lightning/uiRecordApi';

updateRecord(recordInput: Record, clientOptions: Object): Promise<Record>

import { updateRecord } from 'lightning/uiRecordApi';

updateRecord(recordInput: Record, clientOptions: Object): Promise<Record>
```

Other wire adapters can be found in the following documentation:

https://developer.salesforce.com/docs/componentlibrary/documentation/en/lwc/lwc.reference_lightning_ui_api_re

Apart from adapters for database actions and getting records, the API provides adapters to get the UI information like layout information / metadata.

Following factors need to be considered before the usage of UiRecordApi:

1. lightning/uiRecordApi does not support bulk records updates, this means you have to call it multiple times(case you want to update multiple rows of the data table or multiple objects). All and all, if it is very primitive CRUD , go for uiRecordApi, if its more complex stick with apex.
2. There is no transaction control, which implies, if you update/insert multiple from JS, there is no complete rollback.
3. You would want to sanitize your data on Server Side apex code, the reason being JS being browserbased easy to hack, checking data sanity in triggers and validation rule might be bit late. The goal is to prevent bad data as soon as possible
4. Complex requirements like Disabling trigger while inserting few records cant be performed from lightning/uiRecordApi

Conclusion – If your requirement required complex manipulation of user input from LWC component, uiRecordAF might not suffice the need. Consider using custom Apex classes in such scenarios.

Lightning Record Form

- Always consider lightning-record-form first. It is the fastest/most productive way to build a form.
- If you need more control over the layout, want to handle events on individual input fields, or need to execute pre-submission logic: consider lightning-record-view-form or lightningrecord-edit-form.
- If you need even more control over the UI, or if you need to access data without a UI: Use @wire(getRecord)
- Check the following example with best practices

```

Terminal Help  • customCaseRecordPage.js - Learning LWC - Visual Studio Code [Administrator]
JS customCaseRecordPage.js  • customCaseRecordPage.html  • lightningController.js

force-app > main > default > lwc > customCaseRecordPage > JS customCaseRecordPage.js

62 //LWC.html
63
64 <template>
65 <lightning-card title="Lightning Record Edit Form" icon-name="custom:custom14">
66 <div class="slds-m-around_medium">
67 <div>
68 <select id="myselect" class="slds-select" onchange={handleAccountSelect}>
69 <option value="null"><--Select Account--</option>
70 <template for:each={accountData} for:item="acc">
71 <option key={acc.id} value={acc.id}>{acc.Name}</option>
72 </template>
73 </select>
74 </div>
75 <lightning-record-edit-form title="Record Edit"
76 record-id={recordId} object-api-name={accountObject}
77 onsuccess={handleAccountCreated}
78 onsubmit={handleSubmit}>
79 <lightning-messages></lightning-messages>
80 <div class="slds-grid slds-wrap slds-m-around_medium">
81 <template if:false={reloadForm}>
82 <div class="slds-col slds-size_2-of-2">
83 <lightning-input-field field-name={nameField}></lightning-input-field>
84 </div>
85 <div class="slds-col slds-size_2-of-2">
86 <lightning-input-field field-name={phoneField}></lightning-input-field>
87 </div>
88 </template>
89 <template if:true={reloadForm}>
90 <div>
91 <div>
92 <div>
93 <lightning-button class="slds-m-around_medium" type="submit" variant="brand" label={btnLabel}></lightning-button>
94 </lightning-record-edit-form>
95 </lightning-card>
96 </template>
97
98
99
100
101 //LWC.js
102
103 import { LightningElement, track, wire } from 'lwc';
104 import { refreshApex } from '@salesforce/apex';
105 import ACCOUNT_OBJECT from '@salesforce/schema/Account';
106 import ACCOUNT_NAME_FIELD from '@salesforce/schema/Account.Name';
107 import ACCOUNT_PHONE_FIELD from '@salesforce/schema/Account.Phone';
108 import getAllAccounts from '@salesforce/apex/helloWorldClass.getAllAccounts';
109 import { ShowToastEvent } from 'lightning/platformShowToastEvent';
110
111 export default class LightningRecordEditForm extends LightningElement {
112 @wire getAllAccountResult; // a new variable was introduced
113 @track accountData;
114 @track error;
115 @track recordId;
116 @track reloadForm;
117
118 accountObject = ACCOUNT_OBJECT;
119 nameField = ACCOUNT_NAME_FIELD;
120 phoneField = ACCOUNT_PHONE_FIELD;
121
122 @wire(getAllAccounts)
123 imperativeWiring(result) {
124 this.wiredAllAccountResult = result;
125 if(result.data) {
126 this.accountData = result.data;
127 }else if (result.error) {
128 this.error = result.error;
129 }
130 }
131
132 handleAccountCreated(event){
133 this.reloadForm=true;
134 // Run code when account is created.
135 let message = 'Account has been created successfully with the name \''+event.detail.fields.Name.value+'\'';
136 if(this.recordId==undefined && this.recordId==null){
137 message = 'Account has been updated successfully'
138 }
139 const evt = new ShowToastEvent({
140 title: 'Successfull',
141 message: message,
142 variant: 'success',
143 });
144 this.dispatchEvent(evt);
145 refreshApex(this.wiredAllAccountResult);
146 this.recordId=null;
147 this.reloadForm=false;
148 }
149 handleSubmit(event){
150 event.preventDefault(); // stop the form from submitting
151 this.reloadForm=true;
152 const fields = event.detail.fields;
153 this.template.querySelector('lightning-record-edit-form').submit(fields);
154 }
155
156 handleAccountSelect(event) {
157 if(event.target.value!=='null'){
158 this.recordId =event.target.value;
159 }else{
160 this.recordId = null;
161 }
162 }
163 }

```

APEX

- Before you use an Apex method, make sure that there isn't an easier way to get the data. See whether a base Lightning component, like lightning-record-form, lightningrecord-view-form, or lightning-record-edit-form works for your use case. If they don't give you enough flexibility, use a wire adapter like getListUi or getRecordUi. And if you can't use a wire adapter, write an Apex method

- Prefer `@wire` while fetching the meta-data or the data whose chance of getting change at back-end is less because when data get fetch through `@wire` it will get store in Cache memory to increase the performance and if any data get changes in back-end it won't get refresh in Cache memory. They are also building some performance enhancement features that are only available with `@wire`. It makes the code less verbose and defines a predictable response pattern to work and there are two ways to wire an Apex method:
 - Wire the Apex method to a property (`apexWireMethodToProperty` recipe)
 - Wire the Apex method to a function (`apexWireMethodToFunction` recipe)
- Prefer imperative Apex when DML is required or when the return result should not be read-only by keeping `cacheable=false`. when you need to invoke an Apex method without responsive parameters in response to a specific event, like a button click in the apex `ImperativeMethod` recipe mentioned above.
- To call an Apex method imperatively, you can choose to set `cacheable=true`. This setting caches the result on the client and prevents Data Manipulation Language (DML) operations and keep the result read-only.
- In some cases, you know that the cache is stale. If the cache is stale, the component needs fresh data. To query the server for updated data and refresh the cache, call `refreshApex()`
- `@AuraEnabled(continuation=true)`-An Apex controller method that returns a continuation must be annotated with `@AuraEnabled(continuation=true)`.
- `@AuraEnabled(continuation=true cacheable=true)`-To cache the result of a continuation action, set `cacheable=true` on the annotation for the Apex callback method.
- If an Apex method is not annotated with `cacheable=true`, you must call it imperatively. To call any Apex method from an ES6 module, you must call it imperatively.
- You can't refresh data that was fetched by calling an Apex method imperatively. • Prefer the Lightning Data Service functions (`createRecord`, `updateRecord`, and `deleteRecord`) over invoking Apex methods to create, update and delete single records. Benefits: You don't have to write Apex code, and, most importantly, the Lightning Data Service updates the local record cache, ensuring consistency across components showing the same data.


```

/**
 * @description this method is used to fetch records based on object name and
 * field set name and accountId
 * @param strObjectName object name
 * @param strFieldSetName field set name
 * @param selectedAccountId selected account id
 * @return Wrapper which holds column and row for lightning data table
 */
@AuraEnabled(cacheable = true) public static String fetchRecordsByAccountId(
    String strObjectName, String strFieldSetName, String selectedAccountId, String listViewName) {
    switchedAccountId = selectedAccountId;
    if ((strObjectName == LEAD_OBJECT || strObjectName == OPPORTUNITY_OBJECT
        || strObjectName == BULK_UPLOAD_REQUEST_OBJECT
        || strObjectName == PARTNER_BENEFIT_OBJECT
        || strObjectName == PARTNER_INITIATIVE_OBJECT)
        && String.isNotEmpty(listViewName)) {
        filterCriteria = getListViewFilterCriteria(listViewName, strObjectName);
    }
    String allFetchedRecords = fetchRecords(strObjectName, strFieldSetName);
    return allFetchedRecords;
}

```

Error Handling

○ Error handling

Avoid using popup dialogs or toasts for error handling of each component. If multiple components on a page fail, user would see a succession of dialogs, which is a poor user experience. It would be hard for the user to tell which dialog is related to which component because dialogs or toasts aren't visually linked to the components that fail. Prefer in-place/inline error messages, especially for errors that occur without user interaction / when components their state.

We can display component specific error message instead of toast message.

Check the HTML & Javascript example

- Implement a generic error logging framework which can be used on all of the components. Some of the important fields that are often missed as part of error logging frameworks are
 - Device information
 - Browser information
 - Component which caused the error
 - Logged in User

○ HTML

```

<template>
<div if:true={error}>
    <div class="slds-notify_container slds-is-relative">

```

```
<div role="alert" class="slds-notify slds-notify_toast slds-theme_error">
  <div class="slds-notify__content">
    <center> <h2 class="slds-text-heading_small">{errorMessageDisplayed}</h2> </center>
  </div>
</div>
</div>
</div>
</template>
```

○ JS

```
@track error;
@wire(getRecord, { recordId: '$recordId', fields })
wiredRecord({error, data}) {
  if (error) {
    this.error = 'Unknown error';
    if (Array.isArray(error.body)) {
      this.error = error.body.map(e => e.message).join(', ');
    } else if (typeof error.body.message === 'string') {
      this.error = error.body.message;
    }
    this.record = undefined;
  } else if (data) {
    // Process record data
  }
}
```

5.12.3 Aura/VF interoperability

Aura/VF Interoperability

In Winter 20, Salesforce is released “Lightning Message Service” (LMS), a new feature that allows developers to communicate very easily between VF, AURA and Lightning Web components. LMS cuts down the time it takes today to communicate between these platforms by providing a quick channel to which consumers can subscribe and get both updates and convenient tags/import modules, so developers don’t have to worry about communicating resources and/or complicated Javascript.

The advantage of using a Lightning message channel over pubsub or Aura Application event is that message channels are not restricted to a single page. Any component in a Lightning Experience application that listens for events on a message channel updates when it receives a message

However, The Lightning message service doesn’t support the following experiences(yet):

- Salesforce mobile app
- Lightning Out
- Lightning Communities

In containers that don’t support Lightning Messaging Service, use the pubsub module.

Code Reusability between LWC and AURA

To share JavaScript code between Lightning web components and Aura components, put the code in an ES6 module.

- Create the ES6 module in your Lightning Web Components development environment.
- Reference the module in a Lightning web component’s JavaScript file.
- Reference the module in an Aura component.

It is recommended that developers create reusable JavaScript methods in a Utils ES6 module and expose them to other components. This promotes code reusability. An example of the same can be found [here](#).

5.12.4 Event Standards

Parent-to-child

- @api variable - is read only and is used to pass values from parent to child but can't be mutated.
- @api function - doesn't rely on data from parent component

During parent to child communication, we declare the child method as @api so that it can get called from parent LWC by template.querySelector().

<!--Child Component -->

```
<template>
  <div class="fancy-border">
    <video autoplay>
      <source src={videoUrl} type={videoType} />
    </video>
  </div>
</template>

// videoPlayer.js
export default class VideoPlayer extends LightningElement {
  @api videoUrl;
  // The method is declared as @api so that it can get called from parent
  @api
  play() {
    const player = this.template.querySelector('video');
    // the player might not be in the DOM just yet
    if (player) {
      player.play();
    }
  }
}
```

<!--Parent Component-->

```
<template>
  <div>
    <c-video-player video-url={video}></c-video-player>
    <button onclick={handlePlay}>Play</button>
  </div>
</template>

// methodCaller.js
export default class MethodCaller extends LightningElement {
  video = "https://www.w3schools.com/tags/movie.mp4";
  handlePlay() {
    // calling child component. It is not dependent on any parent data
    this.template.querySelector('c-video-player').play();
  }
}
```

- Use custom getter, setter api property - parent component changes data and mutate data

```

1  /*CHILDLWC.js*/
2  import { LightningElement, api } from "lwc";
3  export default class ChildLwc extends LightningElement {
4      @api progressValue;
5      handleChnage(event) {
6          this.progressValue = event.target.value;
7          // Creates the event with the data.
8          const selectedEvent = new CustomEvent("progressvaluechange", {
9              detail: this.progressValue
10         });
11         // Dispatches the event.
12         this.dispatchEvent(selectedEvent);
13     }
14 }
15
16 /*PARENTLWC.html*/
17 <template>
18     <lightning-card title="Getting Data From Child">
19         <p class="slds-p-horizontal_small">
20             <lightning-progress-bar
21                 value={progressValue}
22                 size="large"
23             ></lightning-progress-bar>
24             <c-child-lwc
25                 onprogressvaluechange={hanldeProgressValueChange}
26             ></c-child-lwc>
27         </p>
28     </lightning-card>
29 </template>
30
31
32 /*PARENTLWC.js*/
33 import { LightningElement, track } from "lwc";
34 export default class ParentLwc extends LightningElement {
35     @track progressValue = 0;
36     hanldeProgressValueChange(event) {
37         this.progressValue = event.detail;
38     }
39 }

```

- Avoid sending data from parent to child through events is not a good practice. Instead of events we should pass values from parent to child, by defining attributes. Here is the example.

<!--Parent Component-->

<template>

<c-child parent-record-id={parentRecordId}/>

</template>

<!--Child component-->

.js-

@api parentRecordId;

Best way is to define attributes in child which can be set by Parent.

Use of event is best when data to be passed from child to parent.

Child to Parent

- Use events to send data from child to parent.
- Prefer passing data using primitive data types in the event payload.
- If you must pass data using a non-primitive data type in the event payload, pass a copy of the object or array to avoid leaking private objects and unpredictable mutations by the event listener.
- Prefer setting bubbles to false and composed to false when dispatching events (these are the default values).
- Bubbling is especially useful when the parent uses an iteration of the child component

Deloitte.

```

1  // ChildComponentLWC.js
2  import { LightningElement, api } from 'lwc';
3
4  export default class VideoPlayer extends LightningElement {
5      @api videoUrl;
6      @api
7      play() {
8          const player = this.template.querySelector('video');
9          // the player might not be in the DOM just yet
10         if (player) {
11             player.play();
12         }
13     }
14 }
15
16 //ParentComponentLWC.html
17 <template>
18     <div>
19         <c-video-player video-url={video}></c-video-player>
20         <button onclick={handlePlay}>Play</button>
21         <button onclick={handlePause}>Pause</button>
22     </div>
23 </template>
24
25 //ParentComponentLWC.js
26 import { LightningElement } from 'lwc';
27 export default class MethodCaller extends LightningElement {
28     video = "https://www.w3schools.com/tags/movie.mp4";
29     handlePlay() {
30         this.template.querySelector('c-video-player')
31             .play();
32     }
33 }

```

DOM Events

- Use events to communicate up the component containment hierarchy
- Create and dispatch events in a component's JavaScript class. To create an event, use the CustomEvent constructor. To dispatch an event, call the EventTarget.dispatchEvent() method.
- Naming Convention recommended by salesforce :
 - No uppercase letters
 - No spaces
 - Use underscores to separate words
- To pass data up to a receiving component, set a detail property in the CustomEvent constructor
- It's a best practice either to send only primitives, or to copy data to a new object before adding it to the detail property. Copying the data to a new object ensures that you're sending only the data you want, and that it can't mutate your data.
- Don't include the non-primitive from @api or @wire in detail. These values are wrapped in a read-only member. In IE 11, when crossing the LWC to Aura bridge, the read-only membrane is lost, which means that mutation occurs.

- bubbles: True, composed: True config is not recommended because they bubble up the entire DOM tree, across Shadow DOMs, and through parent components (unless stopped).
- Bubbling is especially useful when the parent uses an iteration of the child component.
- Bubbling and composed events are not recommended because they bubble up the entire DOM tree, across boundaries, and through parent components (unless stopped). If you configure an event with bubbles: true composed: true, the event type becomes part of the component's public API. It also forces the consuming component and all of its ancestors to include the event as part of their APIs. It's a big API contract to sign if you do use this configuration, the event type should be globally unique.
- Use a function signature when attaching event listeners. Example:
 - `myElement.addEventListener('click', functionSignature);`
- Use the same function signature when detaching event listeners. Example:
 - `myElement.removeEventListener('click', functionSignature);`
- Do not use anon functions for listener handlers.
- Use debounce when using event listeners that fire too often.
- CustomEvent.detail: To communicate data to elements in the same shadow tree, don't add myProperty into event.detail. The consumer of the event can use the event.target.myProperty.
- To communicate data to elements that aren't in the same shadow tree, use event.detail. In these cases, event.target.* doesn't work because the real target isn't visible to the listener. (When an event bubbles up if it crosses the shadow boundary, the value of Event.target changes to match the scope of the listener. The retargeted so the listener can't see into the shadow tree of the component that dispatched the event.)
- If you do use detail, use primitive types. JavaScript passes all data types by reference except for primitives: if a component includes an object in its detail property, any listener can mutate that object without the component's knowledge, which is a bad thing!
- It is possible to avoid leaking internal state when using a non-primitive type. Before you add data to the detail property, copy it to a new object or array. Copying the data to a new object ensures that you're sending on what you want, and that the receiver can't mutate your data.
- To communicate between components that aren't in the same DOM tree, use Lightning message service (lightning/messageService).
https://developer.salesforce.com/docs/componentlibrary/documentation/en/lwc/lwc.use_message_channel
- In containers that don't support Lightning Messaging Service (see Aura/VF interoperability section), use the module.

5.12.5 JEST

JEST

Jest is a delightful JavaScript Testing Framework with a focus on simplicity. Use Jest to write unit tests for all of your Lightning web components. To run Jest tests, install `sfdx-lwc-jest`, which sets up all the necessary Jest configurations for a Salesforce DX project.

How to use JEST in Project Repository.

- Every component must have its test script under “__tests__” folders.
- Push the components with “__tests__” folders into repository.
- Use “`npx jest --coverage`” to get the coverage. It is advisable to have coverage more than 75%.
- Make sure coverage is enabled in your package.json jest entry and you can also specify formats you want.
- At time of deployment, All the components under “__tests__” will get ignored.
- Refer: <https://jestjs.io/docs/en/configuration>
- Write unit tests in Jest to test your component & make sure that future changes in your component by other developers don't break things that currently work
- Test file should be present in the folder with name “__tests__” inside the component.
- “`*/__tests__/*`” should be present in the “.forceignore” file in the root folder.
- Each test file shares a single instance of jsdom, and changes aren't reset between tests inside the file. Therefore, it's a best practice to clean up between tests, so that a test's output doesn't affect any other test.
- To test components with wire, __tests__ folder should contain a folder called data. Within the data folder, a file should be present with the same name as the wire adapter in component (getRecord.json).
- Check the `expect.stringContaining(string)` to test the expected result from LWC

5.12.6 Security with LWC

Security with LWC

- To share JS code among components please implement export – import framework. JS function to be exported from source and imported in the component.
- As per LWC content policy requirement please use import static resource functionality to access third party JS , Stylesheet ,Images and files.
- All variables must be declared using var, let and const as LWC implements locker service which enforces strict mode by default.
- Ensure that external libraries which are used also follow strict mode.
- Do not use script tags to include javascript in html.
- Event handlers in html cannot use inline javascript.
- LWC cannot use window or document global properties to query DOM. For example, use `this.template.querySelector()` instead of `document.querySelector()`.
- Lightning locker blocks the usage of some Salesforce Global JS objects such as `$A`, `Aura`, `sfdc` and `sforce`
- Standard DOM APIs that are supported by lightning locker service.
- Locker API Viewer

5.12.7 LWC Performance considerations

LWC Performance considerations

- Use the Lightning Data Service for Data retrieval or cache data when ever possible. Use custom APEX on service does not suffice the need(working with Lists etc)
- A Cacheable Apex Method is a server action whose response is stored in the client cache so that subsequent same server method with the same set of arguments can be accessed from that cache instead of the server.
- Lazy load occasionally accessed data. Don't preload data that the user may never need. For example, put components in a secondary tab that the user may not click.
- To communicate between sibling components within a single Lightning page or across multiple pages, you can use the Lightning Message Service. It has the advantage of working across Visualforce, Aura, LWC, utility bar components & a console app.

For more info - <https://developer.salesforce.com/blogs/2020/06/lightning-web-componentsperformance-best-practices/>

- `async await` can cause performance issues if executed too many times. Use a standard promise chain instead.

Example -

```
async function processData() {
  const data1 = await downloadFromService1();
  const data2 = await downloadFromService2();
  const data3 = await downloadFromService3();
  ...
}
```

That means the `downloadFromService2()` method wouldn't be called and the second request wouldn't be sent before the first request is completed from service 1.

Also the function `downloadFromService3()` wouldn't be called before the `downloadFromService2()` is completed.

Imagine if all the services had ~100ms response time. Then the whole `processData()` method would take more than 300ms to complete.

We would like to have requests sent in parallel and wait for all of them to finish simultaneously. To fix the issue we can use `Promise.all()`.

```
async function processData() {
  const promise1 = downloadFromService1();
  const promise2 = downloadFromService2();
  const promise3 = downloadFromService3();
  ...
}
```

```
const promise3 = downloadFromService3();

const allResults = await Promise.all([promise1, promise2, promise3]);

...

}
```

- Use Continuation - To make a long-running callout, define an Apex method that returns a Continuation object response in a callback method. An asynchronous callout made with a continuation doesn't count toward the synchronous requests that last longer than five seconds. For more info -
 - https://developer.salesforce.com/docs/componentlibrary/documentation/en/lwc/lwc.apex_continuation
 - https://developer.salesforce.com/docs/componentlibrary/documentation/en/lwc/lwc.apex_continuation

Deloitte.

5.12.8 LWC Limitations/Know Issues

General

- A custom Lightning web component can't access a Lightning web component or module in a custom namespace. It can access Lightning web components and modules only in the c and lightning namespaces.
- A custom Lightning web component and a custom Aura component in the same namespace can't have the same name.
- Application event (Pubsub is restricted to a single page)
- Lightning/uiRecordApi on LWC does not return the exact field labels. Also, the positions of the fields get changed compare to actual layout.
- Aura events like force:createRecord, force:editRecord not supported in LWC
- Quick Action is not supported in LWC.

Unsupported features of Lightning Web Components

- Overriding standard buttons like Create, Edit & View
- Custom Actions
- Global Actions
- List View Actions
- Related List View Actions
- Lightning Out [Beta]
- Rendering LWC as PDF Page
- Using LWC in Email Templates
- Console APIs (Navigation Item API, Workspace API, Utility Bar API)
- Chatter Action
- Conversation Toolkit API
- Omni Toolkit API
- Quick Action API

Apex

- If your JavaScript code invokes multiple Apex methods via @wire or imperatively, the Apex limits are applied separately to each method call
- Use the Continuation class in Apex to make a long-running request to an external Web service. Process the response in a callback method. An asynchronous callout made with a continuation doesn't count toward the Apex limit of 10 synchronous requests that last longer than five seconds.
- A single Continuation object can contain a maximum of three callouts.
- The previous continuation invocation must have completed before the next continuation invocation is made.
- Continuation object can't perform any Data Manipulation Language (DML) operations
- The transaction is rolled back if DML any DML Operations performed within the continuation method

Aura coexistence

- You can compose Aura components from Lightning web components, but not the other way around. To communicate down the hierarchy, parents set properties on children.
- Aura component can fire an Aura event to communicate with other Aura components or with the app container.
- ES6 module is mandatory to share JavaScript code between Lightning web components and Aura components

Components Restrictions

- A parameter-less call to `super()` must be the first statement in the constructor body, to establish the correct prototype chain and this value before any further code is run.
- The constructor must not use the `document.write()` or `document.open()` methods.
- A return statement must not appear anywhere inside the constructor body, unless it is a simple early-return (`return` or `return this`).

Lightning Element

- When extending `LightningElement`, we can use only the following accessor: `tagName` and `classList`

DOM Traversing

- `node.childNodes` is discouraged since it returns a `LiveNode` collection. In Lightning Web Components we force it to return an Array snapshot.
- `elm.setAttribute(name, value)` is discouraged if the element is controlled by a template, while `this.setAttribute()` in the component is allowed to add new attributes to the host element.
- `this.addEventListener(elm, type, options)` is discouraged if `options` is present because `passive` and `once` are not supported at the moment.
- `this.attachShadow()` is executed in the super during construction, and cannot be invoked more than once on the same element.
- `slot.childNodes` is discouraged since it is probably a mistake, instead we recommend using `assignedNodes` or `assignedElements`
- Lightning web components will not allow users to manually create or attach shadow roots

Style

- In LWC, the `slotted` keyword is forbidden inside the CSS
- `host-context()` selector is disallowed inside the component's css

5.13 Integration

Integration

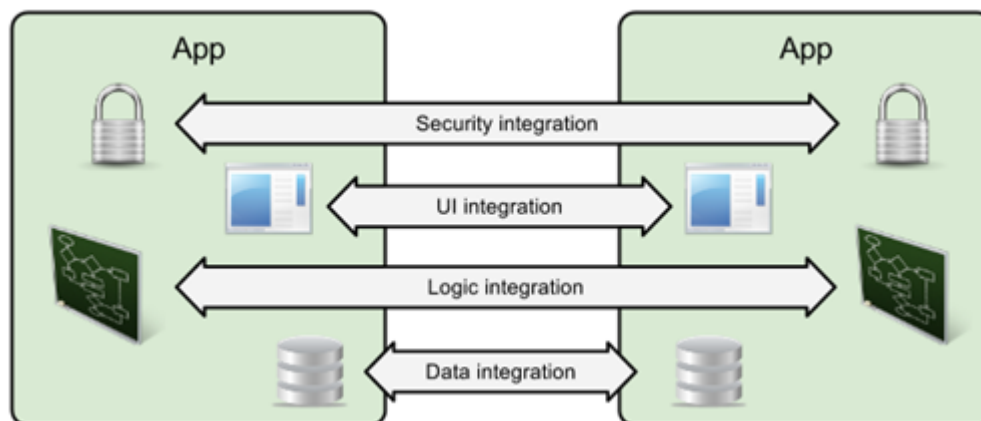
Most enterprise-level applications have a need to integrate with other applications used by the same organization. These integrations usually cater to different layers, like Data, Business Logic, Presentation and Security, depending on the requirement. This helps organizations achieve greater levels of operational consistency, efficiency and quality.

Possible Ways to Integrate with Salesforce

Speaking further on the different layers and integration features available at each level for an enterprise-level application, we have:

- User Interface
- Business Logic
- Data

User Interface Integration is one great way to surface various applications inside Salesforce with a little redesign of each individual app. It provides your users, a single point of entry into multiple applications. The most relevant example is Facebook Apps, which appears to be in the Facebook instance but in reality, the content is served from different application vendors.



Source: <https://developer.salesforce.com/page/Integration>

Business Logic Integration uses Apex Web Services for Inbound and Apex Callouts for Outbound. It typically handles a scenario where business logic is spread across several applications to implement the complete end-to-end business process. An example would be building complex logic on the data received before committing it into Salesforce.

Data Integration uses Change Data Capture, SOAP APIs and REST APIs. It typically handles data synchronization requirements, where one application in an enterprise acts as the primary source for a particular business object, like Account. It requires no coding in Salesforce, which is an advantage, but then it cannot implement any custom logic.

Inbound/Outbound Integration

Before getting onto designing the approach for Integration, as a developer we need to understand these points:

Salesforce Outbound Transaction: SFDC To Remote System

- Need to understand the structure of the third-party whether it is SOAP/REST supportive or not
- Need to understand the basic difference between SOAP and REST
- Need to understand the communication channel mechanism whether it is Synchronous or Asynchronous depending on further parameters:
 - Time taken by the remote system to respond
 - No of callouts made and channel traffic
 - Amount of data being transferred
- Need to understand Authorization Techniques whether to go with oAUTH, SSL/TLS
- HTTPS vs HTTP
- Integration Callout limitations

Salesforce Inbound Transaction: Remote Systems To SFDC

- Need to understand the difference between using Enterprise WSDL or Partner WSDL
- Different Solutions like Rest/SOAP API, APEX Web Service/Apex Rest Service, Bulk API
- Bulkified methodology should be used

Integration Best Practices

When we are implementing any Integration whether it is Inbound Or Outbound one must consider the following points

1. Integration Patterns: Always consider the patterns when we are implementing large scale adoptions and third party integrations. Refer to the Integration pattern section for more details.
2. Use External ID's: The External ID field allows you to store unique record IDs from an external system for update/insert/upsert. Its typically for integration purposes. Salesforce allows you mark up to 25 fields as External IDs and these fields must be text, number or email field types. Values in these External ID fields

must also be unique and you can also determine whether or not the values are case sensitive. If you are loading data from an external system, it is key for mapping the external system's identifier on the Salesforce record.

3. Error Handling: There are different ways to facilitate the handling of data integration errors like sending an email notification to a system admin or create an object and log the errors in the new object.
4. Help Text/Description: When new fields are created on SObject for integration updates, please provide complete description about the field which helps new resource to understand how we used the fields.
5. Avoid Synchronous Make long running callout:
 1. Once a synchronous Apex request runs longer than 5 seconds, it begins counting against limit. Each organization is allowed 10 concurrent long-running requests. If the limit is reached, any new synchronous Apex request results in a runtime exception. This behavior occurs until the organization's requests are below the limit.
<https://developer.salesforce.com/blogs/engineering/2013/05/force-com-concurrent-request-limits.htm>
 2. Use continuation OR use Asynchronous callouts to overcome this.

Key Considerations

- Evaluate the possibility of not storing data in Salesforce if it is required for transactional read only purposes in Salesforce.
- Consider mash up with source UI, if the information is read-only & UI doesn't necessitate a different user experience.
- No point to point integrations while integrating with internal Client systems.
- Evaluate the possibility of leveraging existing integrations if possible than creating new integration
- Avoid file-based transactions if possible and evaluate the possibility of connecting directly to the ODS through an ETL solution.
- Only use Client approved ETL solutions for building batch integrations.

API Callout Limits and Limitations

The limitation will apply when Apex code makes a callout to an HTTP request or a web services call. The web services call can be a SOAP API call or any external web services call.

1. A single Apex transaction can make a maximum of 100 callouts to an HTTP request or an API call.
2. The default timeout is 10 seconds and the customized timeout can be defined with a maximum of 120000 mill secs
3. The maximum cumulative timeout for callouts by a single Apex transaction is 120 seconds. This time is additive across all callouts invoked by the Apex transaction.
4. You can't make a callout when there are pending operations in the same transaction. Things that result in pending operations are DML statements, asynchronous Apex (such as future methods and batch Apex jobs), scheduled Apex, or sending an email. You can make callouts before performing these types of operations.
5. Maximum number of parallel Apex callouts in a single continuation : 3
6. Maximum number of chained Apex callouts : 3
7. Maximum timeout for a single continuation : 120 sec

8. Maximum HTTP Response Size : 1 MB

External Object

External Objects

External objects are supported in API version 32.0 and later. External objects are similar to custom objects, but external object record data is stored outside your Salesforce organization. For example, perhaps you have data that's stored on premises in an enterprise resource planning (ERP) system. Instead of copying the data into your org, you can use external objects to access the data in real time via web service callouts.

External objects are available with Salesforce Connect and Files Connect. Each external object is associated with an external data source definition in your Salesforce organization.

An external data source specifies how to access an external system. Salesforce Connect uses external data sources to access data that's stored outside your Salesforce organization. Files Connect uses external data sources to access third-party content systems. External data sources have associated external objects, which your users and the Lightning Platform use to interact with the external data and content.

By accessing record data on demand, external objects always reflect the current state of the external data. You don't have to manage a copy of that data in Salesforce, so you're not wasting storage and resources keeping data in sync.

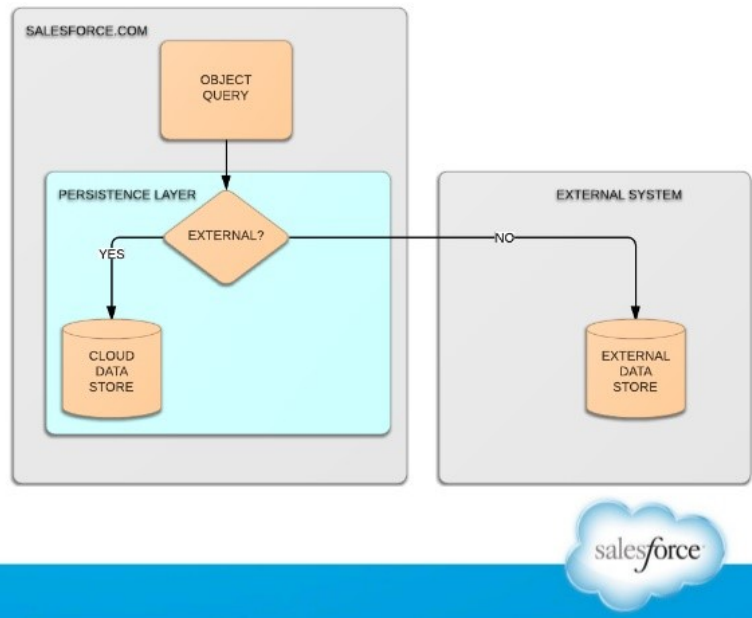
External objects are best used when you have a large amount of data that you can't or don't want to store in your Salesforce organization, and you need to use only a small amount of data at any one time.

See “Define External Objects” in the Salesforce Help for how to create and modify external objects.

An Alternative to ETL

- Extract-Transform-Load is very popular for data synchronization
- Synchronized dataset goes stale immediately

External Data Source allows data interaction without physical movement



Naming Conventions for External Objects

Object names must be unique across all standard, custom, and external objects in the org.

In the API, the names of external objects are identified by a suffix of two underscores immediately followed by a lowercase “x” character. For example, an external object named “ExtraLogInfo” in the Salesforce user interface is seen as ExtraLogInfo__x in that organization's WSDL.

We recommend that you make object labels unique across all standard, custom, and external objects in the org.

External Objects

__x extension

External ID

External Data Source

Fields have External Alias

External Object
SalesOrders

Standard Fields (1) | Custom Fields & Relationships (14) | Page Layouts (1) | Field Sets (1) | Search Layouts (1) | Custom Links (1) | Related User Licenses (1)

Help for this Page

External Object Definition Detail [Edit](#) [Delete](#)

Singular Label	SalesOrders	Description	SalesOrders
Plural Label	SalesOrders	Deployment Status	In Development
Object Name	SalesOrders		
API Name	SalesOrders__x		
External Data Source	SAP		
Table Name	SalesOrders		
Repository Name			
Created By	Test User, 7/22/2013 12:18 PM	Modified By	Test User, 8/8/2013 12:56 PM

Standard Fields

Action	Field Label	Field Name	Data Type	Controlling Field
Edit	Display URL	DisplayUrl	URL(1000)	
Edit	External ID	ExternalId	External Lookup	

Custom Fields & Relationships [New](#)

Action	Field Label	API Name	Data Type	Controlling Field	External Alias	Modified By
Edit Del	BusinessPartnerID	BusinessPartnerID__c	Text(10)		BusinessPartnerID	Test User, 8/8/2013 12:56 PM
Edit Del	ChangedAt	ChangedAt__c	Date/Time		ChangedAt	Test User, 7/22/2013 12:18 PM
Edit Del	CreatedAt	CreatedAt__c	Date/Time		CreatedAt	Test User, 7/22/2013 12:18 PM
Edit Del	Currency	Currency__c	Text(5)		Currency	Test User, 7/22/2013 12:18 PM
Edit Del	CustomerName	CustomerName__c	Text(80)		CustomerName	Test User, 7/22/2013 12:18 PM
Edit Del	NetSum	NetSum__c	Number(13, 2)		NetSum	Test User, 7/22/2013 12:18 PM



External Object Relationships

External objects support standard lookup relationships, which use the 18-character Salesforce record IDs to associate related records with each other. However, data that's stored outside your Salesforce org often doesn't contain those record IDs. Therefore, two special types of lookup relationships are available for external objects: external lookups and indirect lookups. See "External Object Relationships" in the Salesforce Help for details.

Feature Support for External Objects

Most of the Salesforce features that support custom objects also support external objects. However, there are exceptions, and some features have special limitations and considerations for external objects. See the following topics in the Salesforce Help.

- [Salesforce Compatibility Considerations for Salesforce Connect—All Adapters](#)
- [Considerations for Salesforce Connect—All Adapters](#)

Salesforce Connect Adapters

Salesforce Connect uses a protocol-specific adapter to connect to an external system and access its data. This table describes the available adapters.

	Description	Where to Find Callout Limits
--	-------------	------------------------------

Salesforce Connect Adapter		
Cross-org	Uses the Lightning Platform REST API to access data that's stored in other Salesforce orgs.	<p>No callout limits. However, each callout counts toward the API usage limits of the provider org.</p> <p>Salesforce Help: API Usage Considerations for Salesforce Connect—Cross-Org Adapter</p> <p>Salesforce Limits Quick Reference Guide: API Request Limits and Allocations</p>
OData 2.0 OData 4.0	Uses Open Data Protocol to access data that's stored outside Salesforce. The external data must be exposed via OData producers.	<p>Salesforce Help: General Limits for Salesforce Connect—OData 2.0 and 4.0 Adapters</p>
Custom adapter created via Apex	<p>You use the Apex Connector Framework to develop your own custom adapter when the other available adapters aren't suitable for your needs.</p> <p>A custom adapter can obtain data from anywhere. For example, some data can be retrieved from anywhere in the Internet via callouts, while other data can be manipulated or even generated programmatically.</p>	<p>Apex Developer Guide: Callout Limits and Limitations</p> <p>Apex Developer Guide: Execution Governors and Limits</p>

File Connection Adapters

Several Files Connect adapters are also available:

- Google Drive
- Box
- SharePoint Online
- OneDrive for Business

Resources:

- [Set Up Salesforce Connect to Access External Data with the OData 2.0 or 4.0 Adapter](#)
- [Set Up Salesforce Connect to Access Data in Another Org with the Cross-Org Adapter](#)
- [Set Up Salesforce Connect to Access External Data with a Custom Adapter](#)
- [External objects - SlideShare](#)

CORS Whitelisting

○ CORS

What is CORS -

Cross-Origin Resource Sharing (CORS) enables web browsers to request resources from origins other than their (cross-origin). For example, using CORS, JavaScript code at <https://www.example.com> could request a resource <https://www.salesforce.com>. To access supported Salesforce APIs, Apex REST resources, and Lightning Out from JavaScript code in a web browser, add the origin serving the code to a Salesforce CORS whitelist.

Where is CORS used -

CORS whitelisting is used when we are accessing salesforce using web browser then CORS does not support requests for unauthenticated resources.

How to overcome this –

1. Make the callout from server.
2. In case of custom webservice set the response header “Access-Control-Allow-Origin”
3. Whitelist the Origin in salesforce CORS whitelisting setting

Salesforce Technologies supporting CORS –

- Analytics REST API
- Bulk API
- Chatter REST API
- Salesforce IoT REST API
- Lightning Out
- REST API
- User Interface API
- Apex REST

Key Points of consideration –

1. CORS does not support requests for unauthenticated resources, including OAuth endpoints. You must pass OAuth token with requests that require it.
2. VF Pages overcome CORS by default.

3. The origin URL pattern must include the HTTPS protocol (unless you're using your localhost) and a domain name and can include a port.
4. The wildcard character (*) is supported and must be in front of a second-level domain name.
5. The origin URL pattern can be an IP address.

○ HTTPS Protocol

//Noncompliant Code Example:-

```
public without sharing class Foo {  
    void foo() {  
        HttpRequest req = new HttpRequest();  
        req.setEndpoint('http://localhost:com');  
    }  
}
```

//Compliant Solution:-

```
public without sharing class Foo {  
    void foo() {  
        HttpRequest req = new HttpRequest();  
        req.setEndpoint('https://localhost:com');  
    }  
}
```

5.13.1 Integration Patterns

Salesforce Integration Patterns

Below is the Integration patterns available in Salesforce. Architects and developers should consider these pattern design and implementation phase of a Salesforce integration project.

If implemented properly, these patterns enable you to get to production as fast as possible and have the most stable applications possible. Salesforce's own consulting architects use these patterns as reference points during architecture in maintaining and improving them.

Integration Pattern Categories:

- **Data Integration**—These patterns address the requirement to synchronize data that resides in two or more systems and contain timely and meaningful data. Data integration is often the simplest type of integration to implement and includes management techniques to make the solution sustainable and cost-effective. Such techniques often include master data management (MDM), data governance, mastering, de-duplication, data flow design, and others.
- **Process Integration**—The patterns in this category address the need for a business process to leverage two or more applications. When you implement a solution for this type of integration, the triggering application has to call across multiple applications. Usually, these patterns also include both orchestration (where one application is the central “controller” and other applications are multi-participants and there is no central “controller”). These types of integrations can often handle exception handling requirements. Also, such composite applications are typically more demanding on the system to support long-running transactions, and the ability to report on and/or manage process state.

Remote Process Invocation - Request and Reply

- Salesforce invokes a process on a remote system, waits for completion of that process, and then tracks state and returns the result to the remote system.

Remote Process Invocation—Fire and Forget

- Salesforce invokes a process in a remote system but doesn't wait for completion of the process. Instead, it acknowledges the request and then hands off control back to Salesforce.

Batch Data Synchronization

- Data stored in Lightning Platform should be created or refreshed to reflect updates from an external system. Updates in either direction are done in a batch manner.

Remote Call-In

- Data stored in Lightning Platform is created, retrieved, updated, or deleted by a remote system.

UI Update Based on Data Changes

- The Salesforce user interface must be automatically updated as a result of changes to Salesforce data.

For more details refer to the link below -

https://developer.salesforce.com/docs/atlas.en-us.integration_patterns_and_practices.meta/integration_patterns_

Pattern Selection Matrix

The following table lists the patterns, along with key aspects, to help you determine the pattern that best fits your integration requirements.

Source/Target	Type		Timing		Key Pattern(s) to Consider
	Process Integration	Data Integration	Synchronous	Asynchronous	
Salesforce → System (s)	X		X		Remote Process Invocation—Request and Reply
				X	Remote Process Invocation—Fire and Forget
		X	X		Remote Process Invocation—Request and Reply
				X	UI Update Based on Data Changes
System → Salesforce	X		X		Remote Call-In
				X	Remote Call-In
		X	X		Remote Call-In
				X	Batch Data Synchronization

Remote Process Invocation - Request and Reply

Salesforce invokes a process on a remote system, waits for completion of that process, and then tracks state based on the response from the remote system.

Consider the following forces when applying solutions based on this pattern.

- Does the call to the remote system require Salesforce to wait for a response before continuing processing? Is the call to the remote system a synchronous request-reply or an asynchronous request?
- If the call to the remote system is synchronous, does Salesforce have to process the response as part of the same transaction as the initial call?

- Is the message size small or large?
- Is the integration based on the occurrence of a specific event, such as a button click in the Salesforce user interface, or DML-based events?
- Best Fit Solutions:
 - A custom Visualforce page/Lightning Component or button initiates an Apex SOAP callout in a synchronous manner.
 - A custom Visualforce page/Lightning Component or button initiates an Apex HTTP callout in a synchronous manner.
- Sub Optimal:
 - A trigger that's invoked from Salesforce data changes performs an Apex SOAP or HTTP callout in a synchronous manner.
 - A batch Apex job performs an Apex SOAP or HTTP callout in a synchronous manner.

Remote Process Invocation—Fire and Forget

Salesforce invokes a process in a remote system but doesn't wait for completion of the process. Instead, the remote process receives and acknowledges the request and then hands off control back to Salesforce.

Consider the following forces when applying solutions based on this pattern.

- Does the call to the remote system require Salesforce to wait for a response before continuing processing? Is the call to the remote system synchronous request-reply or asynchronous?
- If the call to the remote system is synchronous, does the response need to be processed by Salesforce as part of the same transaction as the call?
- Is the message size small?
- Is the integration based on the occurrence of a specific event, such as a button click in the Salesforce user interface, or DML-based events?
- Is guaranteed message delivery from Salesforce to the remote system a requirement?
- Is the remote system able to participate in a contract-first integration in which Salesforce specifies the contract? In some solution variants (for example, outbound messaging), Salesforce specifies a contract that the remote system endpoint implements.
- Are declarative configuration methods preferred over custom Apex development? In this case, solutions such as outbound messaging are preferred over Apex callouts.
- Best Fit Solutions:
 - Workflow-driven outbound messaging
 - Outbound messaging and callbacks
- Good:
 - Custom Visualforce page/Lightning Component that initiates an Apex SOAP or HTTP asynchronous callout
 - Trigger that's invoked from Salesforce data changes performs an Apex SOAP or HTTP asynchronous callout
- Sub Optimal:

- Batch Apex job that performs an Apex SOAP or HTTP asynchronous callout

Batch Data Synchronization

Data stored in Lightning Platform should be created or refreshed to reflect updates from an external system, and when changes from Lightning Platform should be sent to an external system. Updates in either direction are done in a batch manner.

There are various forces to consider when applying solutions based on this pattern:

- Should the data be stored in Salesforce? If not, there are other integration options an architect can and should consider (mashups, for example).
- If the data should be stored in Salesforce, should the data be refreshed in response to an event in the remote system?
- Should the data be refreshed on a scheduled basis?
- Does the data support primary business processes?
- Are there analytics (reporting) requirements that are impacted by the availability of this data in Salesforce?
- Best Fit Solutions:
 - Change data capture
 - Remote system is the Data Master : Leverage a third-party ETL tool that allows you to run change data capture against source data.
 - Salesforce is the Data Master : Leverage a third-party ETL tool that allows you to run change data capture against ERP and Salesforce data sets.
- Sub Optimal:
 - Remote call-in
 - Remote system is the Data Master : It's possible for a remote system to call into Salesforce by using one of the APIs and perform updates to data as they occur. However, this causes considerable on-going traffic between the two systems.
 - Remote process invocation
 - Salesforce is the Data Master : It's possible for Salesforce to call into a remote system and perform updates to data as they occur. However, this causes considerable on-going traffic between the two systems.

Remote Call-In

Data stored in Lightning Platform is created, retrieved, updated, or deleted by a remote system.

There are various forces to consider when applying solutions based on this pattern:

- Does the call to Salesforce require the remote process to wait for a response before continuing processing? Remote calls to Salesforce are always synchronous request-reply, although the remote process can discard the response if it's not needed to simulate an asynchronous call.

- What is the format of the message (for example, SOAP or REST, or both over HTTP)?
- Is the message size relatively small or large?
- In the case of a SOAP-capable remote system, is the remote system able to participate in a contract-first approach, where Salesforce dictates the contract? This is required where our SOAP API is used, for which a predefined WSDL is supplied.
- Is transaction processing required?
- What is the extent to which you are tolerant of customization in the Salesforce application?
- Best Fit Solutions
 - SOAP API
 - REST API
 - Bulk API (Optimal for bulk operations)
- Sub Optimal:
 - Apex Web services
 - Apex REST service

UI Update Based on Data Changes

The Salesforce user interface must be automatically updated as a result of changes to Salesforce data.

There are various forces to consider when applying solutions based on this pattern:

- Does the data being acted on need to be stored in Salesforce?
- Can a custom user interface layer be built for viewing this data?
- Will the user have access for invoking the custom user interface?

The recommended solution to this integration problem is to use the Salesforce Streaming API/ Platform Events. This solution is comprised of the following components:

- A PushTopic with a query definition that allows you to:
 - Specify what events trigger an update
 - Select what data to include in the notification
- A JavaScript-based implementation of the [Bayeux](#) protocol (currently [CometD](#)) that can be used by the user interface
- A Visualforce page/Lightning Component
- A JavaScript library included as a static resource

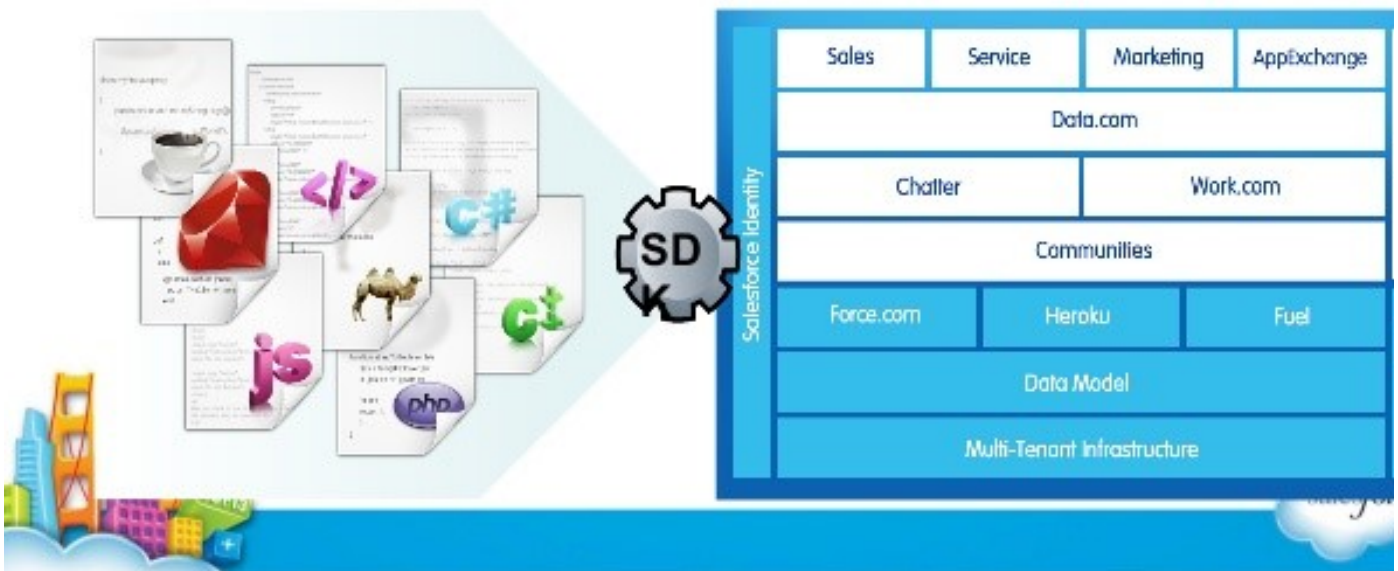
User Interface Integration

Canvas apps :

Canvas enables you to easily integrate a third-party application in Salesforce. Canvas is a set of tools and JavaScript APIs that you can use to expose an application as a canvas app. This means you can take your new or existing applications and make them available to your users as part of their Salesforce experience.

Force.com Canvas – You code it, we show it!

Regardless of your language of choice, JavaScript libraries allow your app to connect with salesforce at a UI and API layer:



Resources:

https://developer.salesforce.com/docs/atlas.en-us.platform_connect.meta/platform_connect/canvas_framework_i

5.13.2 Authentication

Understanding Authentication

Salesforce uses the OAuth protocol to allow users of applications to securely access data without having to reveal username and password credentials.

Before making REST API calls, you must authenticate the application user using [OAuth 2.0](#). To do so, you'll need to:

- [Set up your application as a connected app](#) in the Salesforce organization.
- Determine the correct Salesforce [OAuth endpoint](#) for your connected app to use.
- Authenticate the connected app user via one of several different OAuth 2.0 authentication flows. An OAuth authentication flow defines a series of steps used to coordinate the authentication process between your application and Salesforce. Supported OAuth flows include:
 - [Web server flow](#), where the server can securely protect the consumer secret.
 - [User-agent flow](#), used by applications that cannot securely store the consumer secret.
 - [Username-password flow](#), where the application has direct access to user credentials.

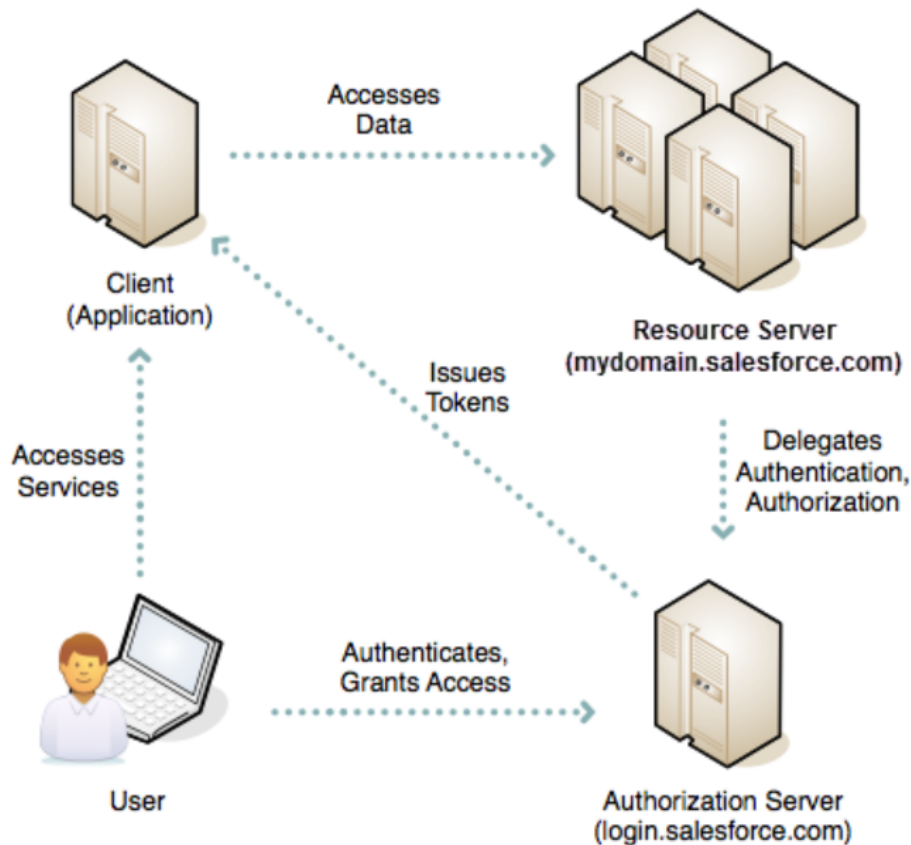
After successfully authenticating the connected app user with Salesforce, you'll receive an access token which can be used to make authenticated REST API calls.

OAuth 2.0

[Deeper into OAuth 2.0 in Salesforce](#)

OAuth can authorize access to resources without revealing user credentials to apps. Apps that use OAuth can also directly authenticate and access Salesforce resources without a user's presence and APIs, such as the Salesforce REST and SOAP web service APIs or the Chatter REST API, can use OAuth 2.0 to authorize access to Salesforce resources.

OAuth is sometimes described as a valet key for the web. A valet key restricts access to a car. A person can drive it, but can't use the key to open the trunk or glove box. In the same way, OAuth gives a client application restricted access to your data on a resource server. To allow access, an authorization server grants tokens to the client app in response to an authorization.



OAuth Tokens

As the valet key, OAuth tokens authorize access to resources. Here's a quick review of the different OAuth 2.0 token types.

- **Authorization code**—The authorization server creates this short-lived token and passes it to the client application via the browser. The client application sends the authorization code to the authorization server to obtain an access token and, optionally, a refresh token.
- **Initial access token**—After configuring an OAuth 2.0 connected app, generate an initial access token. Salesforce requires this token to authenticate the dynamic client registration request. See [Generate an Initial Access Token](#) and [OpenID Connect Dynamic Client Registration Endpoint](#).
- **Access token**—The client uses an access token to make authenticated requests on behalf of the end user. It has a longer lifetime than the authorization code, usually minutes or hours. When an access token expires, attempts to use it fail, and the app must obtain a new access token. In Salesforce terms, the access token is a session ID (SID), much like a session cookie on other systems. It must be protected against interception, for example, by Transport Layer Security (TLS, also called SSL). To use an OAuth access token, either set a SID cookie or use a "front door" URL (such as `https://mydomain.salesforce.com/secur/frontdoor.jsp?sid=<sid_value>`). In both cases, you must include "web" in the list of requested scopes. For parameter details, see [Scope Parameter Values](#).
- **Refresh token**—A refresh token can have an indefinite lifetime, persisting for an admin-configured interval or until explicitly revoked. The client application can store a refresh token, using it to periodically

obtain fresh access tokens. For this reason, the app must protect a refresh token against unauthorized access. Like a password, a refresh token can be used repeatedly to gain access to the resource server. Because a refresh token can expire or a user can revoke it outside of the client, the client must handle failures to obtain an access token. Typically, the client replays the protocol from the start.

- ID token—OpenID Connect, an authentication layer on top of OAuth 2.0, defines an ID token as a signed data structure. The data structure contains authenticated user attributes, including a unique identifier for the user. It also contains the time when the token was issued and an identifier for the requesting client. An ID token is encoded as a JSON web token (JWT).
- [Scope Parameter Values](#) - The scope parameter fine-tunes the permissions associated with the tokens that you're requesting. Scope is a subset of values that you specified when defining the connected app.

OAuth 2.0 Authentication Endpoints

1. Authorization—<https://login.salesforce.com/services/oauth2/authorize>
2. Token—<https://login.salesforce.com/services/oauth2/token>
3. Revoke—<https://login.salesforce.com/services/oauth2/revoke>

Listed below are the different types of OAuth flows in Salesforce.

- [OAuth 2.0 Web Server Authentication Flow](#)
 - Apps that are hosted on a secure server use the web server authentication flow. A critical aspect of the web server flow is that the server must be able to protect the consumer secret. You can use code challenges and verifier values in the flow to prevent authorization code interception.
- [OAuth 2.0 Username-Password Flow](#)
 - Use the username-password authentication flow to authenticate when the consumer already has the user's credentials.
- [OAuth 2.0 User-Agent Flow](#)
 - With the OAuth 2.0 user-agent authentication flow, users authorize your desktop or mobile app to access their data. Client apps that run on a device or in a browser use this flow to obtain an access token.
- [OAuth 2.0 Refresh Token Flow](#)
 - The OAuth 2.0 refresh token flow renews tokens issued by the web server or user-agent flows.
- [OAuth 2.0 SAML Bearer Assertion Flow](#)
 - The OAuth 2.0 SAML bearer assertion flow defines how a SAML assertion is used to request an OAuth access token when a client wants to use a previous authorization. Authentication of the authorized app is provided by the digital signature applied to the SAML assertion. A SAML assertion is an XML security token issued by an identity provider and consumed by a service provider. The service provider relies on its content to identify the assertion's subject for security-related purposes.
- [OAuth 2.0 JWT Bearer Token Flow](#)
 - A JSON Web Token (JWT) enables identity and security information to be shared across security domains. When a client wants to use previous authorization, the client posts an access token

request that includes a JWT to Salesforce's OAuth token endpoint. Salesforce authenticates the authorized app through a digital signature that is applied to the JWT. Use the OAuth 2.0 JWT bearer token flow to define the authentication process.

- [OAuth 2.0 Device Authentication Flow](#)
 - The OAuth 2.0 device authentication flow is typically used by applications on devices with limited input or display capabilities, such as TVs, appliances, or command-line applications. Users can connect these client applications to Salesforce by accessing a browser on a separate device that has more developed input capabilities, such as a desktop computer or smartphone.
- [OAuth 2.0 Asset Token Flow](#)
 - Client applications use the OAuth 2.0 asset token flow to request an asset token from Salesforce for connected devices. In this flow, an OAuth access token and an actor token are exchanged for an asset token. This flow combines asset token issuance and asset registration for efficient token exchange and automatic linking of devices to Service Cloud Asset data.
- [SAML Assertion Flow](#)
 - The SAML assertion flow is an alternative for orgs that are currently using SAML to access Salesforce and want to access the web services API the same way. You can use the SAML assertion flow only inside a single org. You don't have to create a connected app to use this assertion flow. Clients can use this assertion flow to federate with the API using a SAML assertion, the same way they federate with Salesforce for web single sign-on.

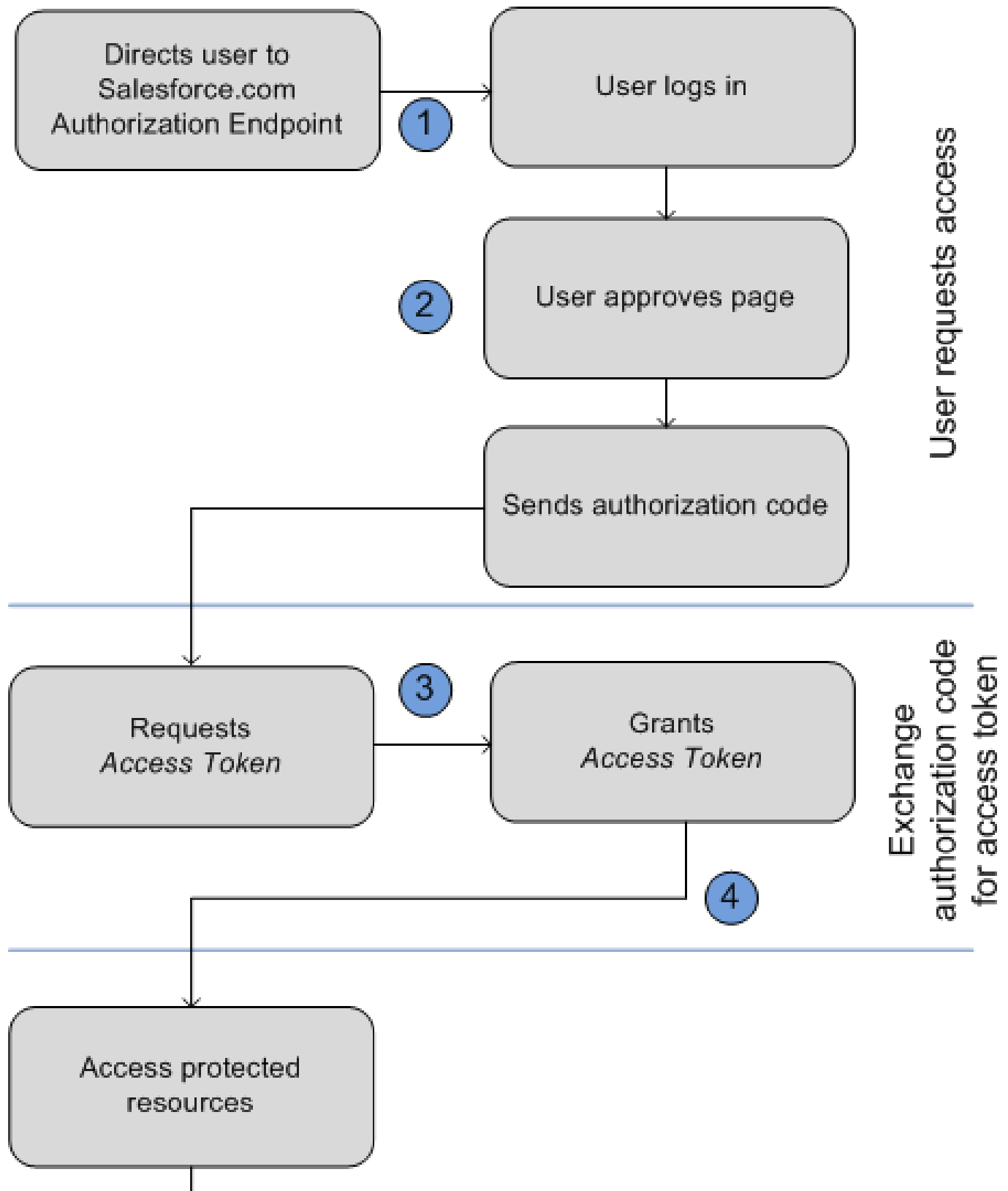
Reference : https://help.salesforce.com/articleView?id=remoteaccess_authenticate_overview.htm

Web Server Flow

Apps that are hosted on a secure server use the web server authentication flow. A critical aspect of the web server flow is that the server must be able to protect the consumer secret. You can use code challenges and verifier values in the flow to prevent authorization code interception.

Web Server Consumer

Salesforce





The following image shows the authentication flow for Web server clients

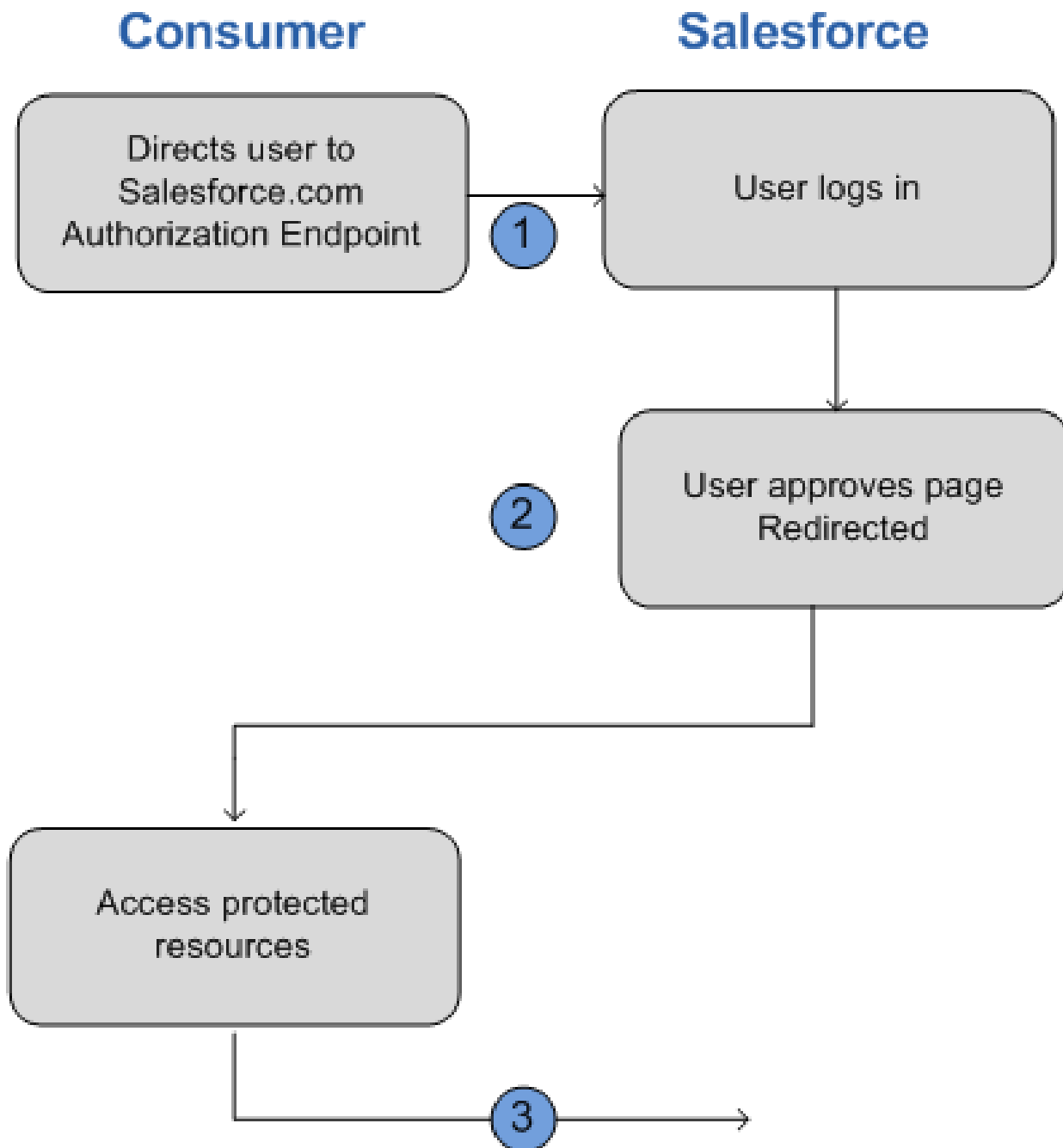
- The web server redirects the user to Salesforce, which authenticates and authorizes the server to access the data on the user's behalf.
- After the user approves access, the web server receives a callback with an authorization code. The web server passes back the authorization code to get a token response.
- After validating the authorization code, Salesforce passes back a token response. If there's no error, the token response includes an access code and additional information.
- After the token is granted, the web server accesses the user's data.

User Agent Flow

Users authorize your desktop or mobile app to access their data. Client apps that run on a device or in a browser use this flow to obtain an access token. With the user-agent authentication flow, the client app receives the access token as an HTTP redirection. The client app requests the authorization server to redirect the user-agent to another web server or to an accessible local resource. The server can extract the access token from the response and pass it to the client app. For security, the token response is provided as a hash (#) fragment on the URL. It prevents the token from being passed to the server or to any other servers in referral headers.

Because the access token is encoded into the redirection URI, it might be exposed to the user and other apps residing on the device. If you're using JavaScript to authenticate, call `window.location.replace()`; to remove the callback from the browser's history.

The following diagram shows the authentication for web server clients



1. The client app directs the user to Salesforce to authenticate and authorize the app.
2. The user approves access for this authentication flow.
3. The app receives the callback from Salesforce.

After a consumer has an access token, it can use the access token to access Salesforce data on the user's behalf. The consumer can use a refresh token to get a new access token if it becomes invalid for any reason.

Username-Password Flow

Use the username-password authentication flow to authenticate when the consumer already has the user's credentials.

Here are the steps for the username-password authentication flow.

1. The consumer uses the user's username and password to [request an access token](#) (session ID.)
2. After the request is verified, Salesforce [sends a response](#) to the client.

After a consumer has an access token, it can [use the access token](#) to access Salesforce data on the user's behalf.

The consumer can use the user's username and password to request an access token, which can be used as a session ID. This flow does not support including scopes in the request, and the access token returned from this flow does not get scopes

The consumer makes an out-of-band POST request to the token endpoint, with the following parameters.

- grant_type—Value must be password for this flow
- client_id—Consumer key from the connected app definition
- client_secret—Consumer secret from the connected app definition
- username—User's username
- password—User's password (+ Security Token)
- format—(Optional) Expected return format. Values are:urlencoded,json,xml

Reference : https://help.salesforce.com/articleView?id=remoteaccess_authenticate_overview.htm

Username-Password Flow Example

Connected App : Test_Inbound_Integration

Consumer Key : 3MVG9bx

Consumer Secret : 7392422711018044081

URL : <https://test.salesforce.com/services/oauth2/token>

POST Request Parameter:

grant_type=password

client_id=3MVG9bx

client_secret=7392

username=USERID

password=PASSWORDSECURITYTOKEN (PASSWORD+SECURITY TOKEN)

e.g. POST

https://test.salesforce.com/services/oauth2/token?grant_type=password&client_id=3MVG9bx&client_secret=

Response:

```
{
  "access_token": "0De0000005WO66!ARgAQJXIP",
  "instance_url": "https://test--Dev.cs15.my.salesforce.com",
  "id": "https://test.salesforce.com/id/00De00000005WO66EAG/005e0000002nSyLAAU",
  "token_type": "Bearer",
  "issued_at": "1438872311460",
  "signature": "eOhwkqoGsXog4p9ESm3mH5UctuS+V1mIM+rnMPL8n7g="
}
```

The REST Invocation :

"instance_url" + "/services/apexrest/v1/Test_Integ_Response_Handler"

e.g. https://test--Dev.cs15.my.salesforce.com/services/apexrest/v1/Test_Integ_Response_Handler

Header :

Authorization

Value : "token_type" + " " + "access_token"

e.g. Authorization: Bearer 0De0000005WO66!ARgAQJXIP

e.g JSON Request -

```
{
  "EventName": "TestInboundRequest",
  "Data": [ {
    "OrderRequests": [ {
      "Status": "Submitted",
      "sOrderId": "123"
    }, {
      "Status": "Error",
      "sOrderId": "131"
    }
  ]
}]
}
```

Named Credentials

Named Credentials can help to avoid sharing the Username / Password to connect to an external system and also to hide it.

A named credential specifies the URL of a callout endpoint and its required authentication parameters in one definition. Salesforce manages all authentication for Apex callouts that specify a named credential as the callout endpoint so that your code doesn't have to. You can also skip remote site settings, which are otherwise required for callouts to external sites, for the site defined in the named credential.

By separating the endpoint URL and authentication from the callout definition, named credentials make callouts easier to maintain. For example, if an endpoint URL changes, you update only the named credential. All callouts that reference the named credential simply continue to work.

To reference a named credential from a callout definition, use the named credential URL. A named credential URL contains the scheme `callout:`, the name of the named credential, and an optional path. For example: `callout:My_Named_Credential/some_path`.

Named Credentials Example

```
HttpRequest req = new HttpRequest();  
req.setEndpoint('callout:My_Named_Credential/some_path');  
req.setMethod('GET');  
Http http = new Http();  
HttpResponse res = http.send(req);  
System.debug(res.getBody());
```

5.13.3 Web Services REST

REST

REST (Representational State Transfer): You can expose your Apex class and methods so that external applications can access your code and your application through the REST architecture. This is done by defining your Apex class with the `@RestResource` annotation to expose it as a REST resource. Similarly, add annotations to your methods to expose them through REST.

- REST is another architectural pattern, an alternative to SOAP.
- It works over with HTTP and HTTPS.
- It works with GET, POST, PUT and DELETE verbs to perform CRUD operations.
- It is based on URI.
- REST Supports both XML and JSON format.

1. REST API preferred for services that are exposed as public APIs and mobile, since JSON being Lighter the app runs smoother and faster.

REST Web Service

You can expose your Apex class and methods so that external applications can access your code and your application through the REST architecture. This is done by defining your Apex class with the `@RestResource` annotation to expose it as a REST resource. Similarly, add annotations to your methods to expose them through REST.

These are the classes containing methods and properties you can use with Apex REST.

Class	Description
RestContext Class	Contains the <code>RestRequest</code> and <code>RestResponse</code> objects.
request	Represents an object used to pass data from an HTTP request to an Apex RESTful Web service method.
response	Represents an object used to pass data from an Apex RESTful Web service method to an HTTP response.

Authentication

Apex REST supports these authentication mechanisms:

- OAuth 2.0
- Session ID

Apex REST Annotations

A total of six new annotations have been added that enables you to expose an Apex class as a RESTful Web service.

- **@RestResource(urlmapping='/yourURL/') :** Used at the class level and enables you to expose an Apex class as a REST resource. Considerations when using this annotation as follows
 - The URL mapping is relative to `https://instance.salesforce.com/services/apexrest/`
 - A wildcard character (*) may be used.
 - The URL mapping is case-sensitive
 - Apex class must be defined as global.
- **@Httpdelete :** Used at the method level and called when an HTTP DELETE request is sent and deletes the specified resource. To use this Apex method must be defined as global static.
- **@HttpGet :** Used at the method level and called when an HTTP GET request is sent and returns the specified resource. To use this Apex method must be defined as global static and Methods annotated with @HttpGet are also called if the HTTP request uses the HEAD request method.
- **@HttpPatch :** Used at the method level and called when an HTTP PATCH request is sent and updates the specified resource. To use this Apex method must be defined as global static.
- **@HttpPost :** Used at the method level and called when an HTTP POST request is sent and creates a new resource. To use this Apex method must be defined as global static.
- **@HttpPut :** Used at the method level and called when an HTTP PUT request is sent and creates or updates the specified resource. To use this Apex method must be defined as global static.

Key Considerations:

- Apex class methods that are exposed through the Apex REST API don't enforce object permissions and field-level security by default. We recommend that you make use of the appropriate object or field describe result methods to check the current user's access level on the objects and fields that the Apex REST API method is accessing.
- Also, sharing rules (record-level access) are enforced only when declaring a class with the `with sharing` keyword. This requirement applies to all Apex classes, including to classes that are exposed through Apex REST API. To enforce sharing rules for Apex REST API methods, declare the class that contains these methods with the `with sharing` keyword.
- These return and parameter types are allowed:
 - Apex primitives (excluding sObject and Blob).
 - sObjects
 - Lists or maps of Apex primitives or sObjects (only maps with String keys are supported).
 - [User-defined types](#) that contain member variables of the types listed above.
- Methods annotated with @HttpGet or @HttpDelete should have no parameters. This is because GET and DELETE requests have no request body, so there's nothing to deserialize.
- A single Apex class annotated with @RestResource can't have multiple methods annotated with the same HTTP request method. For example, the same class can't have two methods annotated with @HttpGet.

- `RestRequest` and `RestResponse` objects are available by default in your Apex methods through the static `RestContext` object.
 - `RestRequest req = RestContext.request;`
 - `RestResponse res = RestContext.response;`
- If the Apex method has no parameters, Apex REST copies the HTTP request body into the `RestRequest.requestBody` property. If the method has parameters, then Apex REST attempts to deserialize the data into those parameters and the data won't be deserialized into the `RestRequest.requestBody` property.
- Apex REST uses similar serialization logic for the response. An Apex method with a non-void return type will have the return value serialized into `RestResponse.responseBody`.
- Apex REST methods can be used in managed and unmanaged packages. When calling Apex REST methods that are contained in a managed package, you need to include the managed package namespace in the REST call URL. For example, if the class is contained in a managed package namespace called `packageNamespace` and the Apex REST methods use a URL mapping of `/MyMethod/*`, the URL used via REST to call these methods would be of the form `https://instance.salesforce.com/services/apexrest/packageNamespace/MyMethod/`.
- If a login call is made from the API for a user with an expired or temporary password, subsequent API calls to custom Apex REST Web service methods aren't supported and result in the `MUTUAL_AUTHENTICATION_FAILED` error. Reset the user's password and make a call with an unexpired password to be able to call Apex Web service methods.

Rest Web Service Example

○ Process Account

//This sample shows you how to implement a simple REST API in Apex that handles three different HTTP request methods.

```
@RestResource(urlMapping='/Account/*')
```

```
global with sharing class MyRestResource {
```

```
    @HttpDelete
```

```
    global static void doDelete() {
```

```
        RestRequest req = RestContext.request;
```

```
        RestResponse res = RestContext.response;
```

```
        String accountId = req.requestURI.substring(req.requestURI.lastIndexOf('/')+1);
```

```
        Account account = [SELECT Id FROM Account WHERE Id = :accountId];
```

```
        delete account;
```

```
    }
```

```
    @HttpGet
```

```
    global static Account doGet() {
```

```
        RestRequest req = RestContext.request;
```

```
        RestResponse res = RestContext.response;
```

```
        String accountId = req.requestURI.substring(req.requestURI.lastIndexOf('/')+1);
```



```

    Account result = [SELECT Id, Name, Phone, Website FROM Account WHERE Id = :accountId];
    return result;
}

```

```

@HttpPost
global static String doPost(String name,
    String phone, String website) {
    Account account = new Account();
    account.Name = name;
    account.phone = phone;
    account.website = website;
    insert account;
    return account.Id;
}
}

```

○ RestRequest object

//The following sample shows you how to add an attachment to a case by using the RestRequest object
 @RestResource(urlMapping='/CaseManagement/v1/*')

```

global with sharing class CaseMgmtService
{

```

```

    @HttpPost
    global static String attachPic(){
        RestRequest req = RestContext.request;
        RestResponse res = Restcontext.response;
        Id caseId = req.requestURI.substring(req.requestURI.lastIndexOf('/')+1);
        Blob picture = req.requestBody;
        Attachment a = new Attachment (ParentId = caseId,
            Body = picture,
            ContentType = 'image/jpg',
            Name = 'VehiclePicture');

        insert a;
        return a.Id;
    }
}

```

External Services via Flow

Use External Services to connect to a service of your choice, invoke methods based on the external source via a flow, and import data from the service into Salesforce—all with the help of an easy-to-use wizard.

External Services is a feature within Salesforce that leverages declarative tools to connect to an external endpoint, such as a credit service provider, and bring its logic into Salesforce.

There are many other applications for using External Services. In fact, there are probably as many applications as there are external service providers. Once you learn the basics, you can determine what your business cases are and use this feature as you need.

With External Services, you use declarative tools to import Swagger or Interagent-based API definitions right into Salesforce using a schema. Once you import the definitions, you can create a flow based on the Apex classes generated from your External Services registration...

Deloitte.



1. An external services provider, such as a bank, shares its REST-based API schema specification. Think of this specification as a type of technical contract.
2. Based on the specification, a developer creates a schema definition that describes the API.
3. A Salesforce administrator declaratively creates a named credential to authenticate to the service's endpoint using the URL provided by the external service provider. The endpoint is simply what exposes the web services resources that External Services needs to interact with.

4. A Salesforce administrator declaratively registers the service and uses both the named credential and schema definition during the registration process. External Services automatically imports the definitions into your org and generates Apex actions, which are available immediately in Lightning Flow.
5. A Salesforce administrator declaratively creates a flow using the External Services-generated Apex actions.
6. Users run the flow. During runtime, External Services sends a callout to the service's endpoint. The service returns output based on the schema definition.
7. Data is retrieved, created, updated, or deleted.

Resources:

- [External Services](#)
- [External Services: Connect to a Service Using a Wizard](#)

Deloitte.

5.13.4 Web Services SOAP

SOAP

SOAP(Simple Object Access Protocol) : SOAP (Simple Object Access Protocol) is a messaging protocol that allows programs that run on disparate operating systems (such as Windows and Linux) to communicate using Hypertext Transfer Protocol (HTTP) and its Extensible Markup Language (XML)

- SOAP is a web service architecture, which specifies the basic rules to be considered while designing web service platforms.
- It works over with HTTP, HTTPS, SMTP, XMPP.
- It works with WSDL.
- It is based on standard XML format and Supports data in the form of XML only
- SOAP API preferred for services within the enterprise in any language that supports Web services.

SOAP Web Services

Salesforce allows us to expose Apex methods as SOAP web services so that external applications can access the code and the application. This is commonly used for scenarios that involve connecting a company's front office CRM systems to their back office and accounting ERP counterparts.

Apex class methods can be exposed as custom SOAP Web service calls. This allows an external application to invoke an Apex Web service to perform an action in Salesforce. Use the webservice keyword to define these methods.

Key Considerations :

- Apex class methods that are exposed through the API with the webservice keyword don't enforce object permissions and field-level security by default. We recommend that you make use of the appropriate object or field describe result methods to check the current user's access level on the objects and fields that the webservice method is accessing.
- Also, sharing rules (record-level access) are enforced only when declaring a class with the with sharing keyword. This requirement applies to all Apex classes, including classes that contain webservice methods. To enforce sharing rules for webservice methods, declare the class that contains these methods with the with sharing keyword.
- Use the webservice keyword to define top-level methods and outer class methods. You can't use the webservice keyword to define a class or an inner class method.
- You cannot use the webservice keyword to define an interface or to define interface methods and variables.
- System-defined enums cannot be used in Web service methods.
- You cannot use the webservice keyword in a trigger.
- All classes that contain methods defined with the webservice keyword must be declared as global. If a method or inner class is declared as global, the outer, top-level class must also be defined as global.

- Methods defined with the `webservice` keyword are inherently global. Any Apex code that has access to the class can use these methods. You can consider the `webservice` keyword as a type of access modifier that enables more access than global.
- Define any method that uses the `webservice` keyword as static.
- You cannot deprecate `webservice` methods or variables in managed package code.
- Because there are no SOAP analogs for certain Apex elements, methods defined with the `webservice` keyword cannot take the following elements as parameters. While these elements can be used within the method, they also cannot be marked as return values.
 - Maps
 - Sets
 - Pattern objects
 - Matcher objects
 - Exception objects
- Use the `webservice` keyword with any member variables that you want to expose as part of a Web service. Do not mark these member variables as static.
- Considerations for calling Apex SOAP Web service methods:
 - Salesforce denies access to Web service and execute anonymous requests from an AppExchange package that has Restricted access.
 - Apex classes and triggers saved (compiled) using API version 15.0 and higher produce a runtime error if you assign a String value that is too long for the field.
 - If a login call is made from the API for a user with an expired or temporary password, subsequent API calls to custom Apex SOAP Web service methods aren't supported and result in the `INVALID_OPERATION_WITH_EXPIRED_PASSWORD` error. Reset the user's password and make a call with an unexpired password to be able to call Apex Web service methods.

SOAP Web Service Example

```
// example Class for SOAP Web Service
global class SpecialAccounts {

    global class AccountInfo {
        webservice String AcctName;
        webservice Integer AcctNumber;
    }

    webservice static Account createAccount(AccountInfo info) {
        Account acct = new Account();
        acct.Name = info.AcctName;
        acct.AccountNumber = String.valueOf(info.AcctNumber);
        insert acct;
        return acct;
    }
}
```

```

webservice static Id [] createAccounts(Account parent,
    Account child, Account grandChild) {

    insert parent;
    child.parentId = parent.Id;
    insert child;
    grandChild.parentId = child.Id;
    insert grandChild;

    Id [] results = new Id[3];
    results[0] = parent.Id;
    results[1] = child.Id;
    results[2] = grandChild.Id;
    return results;
}
}

// Test class for the previous class.
@isTest
private class SpecialAccountsTest {
    testMethod static void testAccountCreate() {
        SpecialAccounts.AccountInfo info = new SpecialAccounts.AccountInfo();
        info.AcctName = 'Manoj Cheenath';
        info.AcctNumber = 12345;
        Account acct = SpecialAccounts.createAccount(info);
        System.assert(acct != null);
    }
}

```

5.13.5 Integration Testing

Workbench

Using Workbench :

Prerequisites

- 1) Click on the link to open workbench: [Workbench](#)
- 2) Log in to your Salesforce Org, and allow access.
- 3) Go to Utilities > REST Explorer

REST Explorer Through Workbench

- 1: Go to Utilities and Rest explorer as shown below



- 2: Under "Choose an HTTP method to perform on the REST API service URI below: select Http Post Verb
- 3: In the url-box enter this URL: "/services/apexrest/v44.0/YourRestServucename/" e.g : "
/services/apexrest/v44.0/PartnerDirectorytoPostPartnerDetails/"
- 4: If it is post method, enter the request body in JSON format and it will look like below

Choose an HTTP method to perform on the REST API service URI below:

☐ GET
 ☒ POST
 ☐ PUT
 ☐ PATCH
 ☐ DELETE
 ☐ HEAD
 Headers Reset Up

/services/apexrest/v1/Account Execute

Request Body

```
{
  "Name" : "Test ARB User",
  "Address" : "Edge Communication Consulting",
  "LastName" : "ARB User",
}
```

Your Salesforce Instance

This is the custom URL created to be used by external system

5: Click on execute to see the response

6: if we need to test @httpget repeats steps from 2 and in get we are not required to provide the body.

Reference :

https://developer.salesforce.com/docs/atlas.en-us.api_rest.meta/api_rest/quickstart_using_workbench.htm

Postman

Before going to know how to use postman for testing Apex callouts, we have to know about the connected APPs.

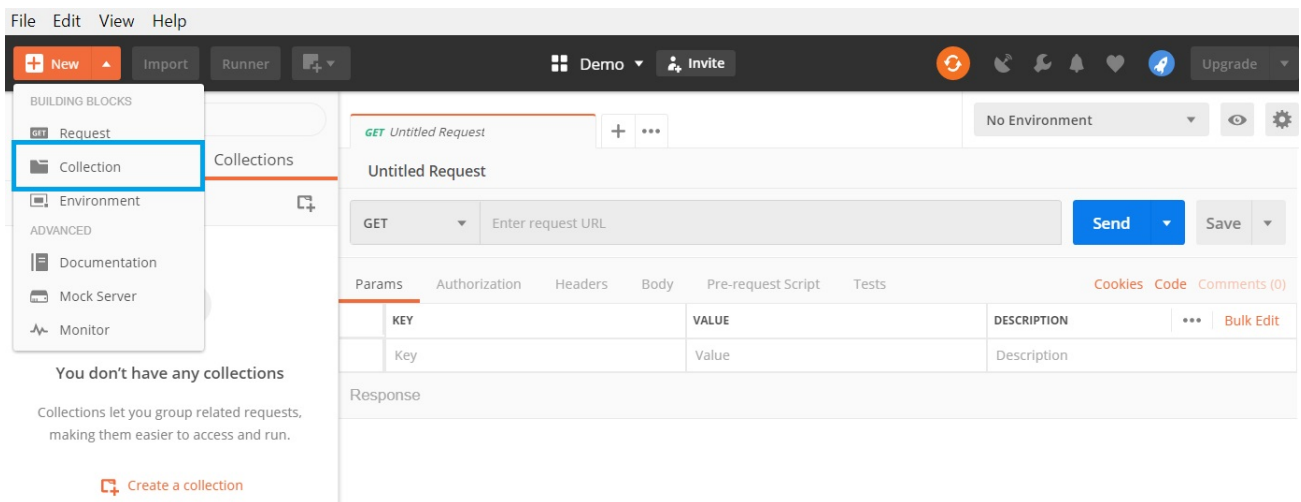
A connected app integrates an application with Salesforce using APIs. Connected apps use standard SAML and OAuth protocols to authenticate, provide single sign-on, and provide tokens for use with Salesforce APIs. In addition to standard OAuth capabilities, connected apps allow Salesforce admins to set various security policies and have explicit control over who can use the corresponding apps.

How to create a Connected APP?

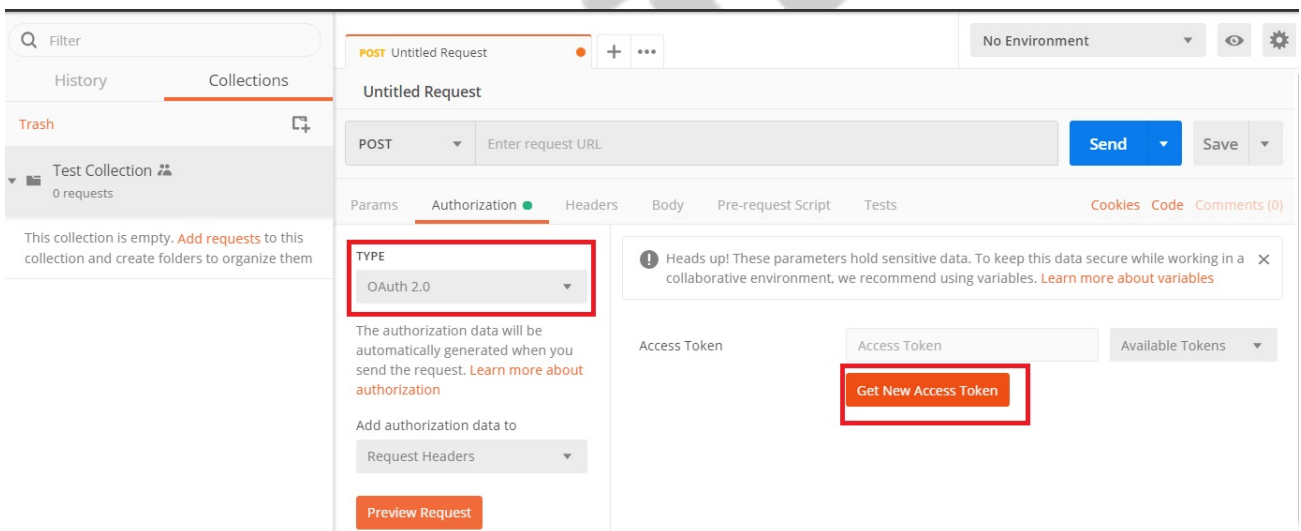
- In Lightning Experience, you use the App Manager to create connected apps. From Setup, enter App in the Quick Find box, then select App Manager. Click New Connected App.
- In Salesforce Classic, from Setup, enter Apps in the Quick Find box, then select Build > Create > Apps. Under Connected Apps, click New.
- Please provide the information and please provide the Callback URL as "<https://www.getpostman.com/oauth2/callback>"
- Please give appropriate access e.g. Choose Full Access
- Now Save and don't forget to copy the Consumer Key and Consumer Secret keys.

Steps to test with Postman:

1. Download the [Postman Client](#) or Add the [Extension from Google Play](#) and follow the steps below
2. Open the Postman Client and Click on New --> Click Collection --> Give a Name and Save.



3. Now Click on the collection we just created and then Click "Authorization" and the Type to "OAuth 2.0" as shown below



4. Now click on get access token after filling

Auth url : <https://test.salesforce.com/services/oauth2/authorize>

Access Token url : <https://test.salesforce.com/services/oauth2/token>

Username : yourSalesforceusername

Password : SalesforcePassword

GET NEW ACCESS TOKEN

Token Name

Token Name

Grant Type

Authorization Code

Callback URL ⓘ

http://your-application.com/registered/callback

Auth URL ⓘ

https://test.salesforce.com/services/oauth2/authorize

Access Token URL ⓘ

https://test.salesforce.com/services/oauth2/token

Client ID ⓘ

Ze03Pqks2

Client Secret ⓘ

3LMHU

Scope ⓘ

e.g. read:org

State ⓘ

State

Client Authentication

Send as Basic Auth header

Request Token

5. Once the token is generated please click on Use Token by selecting Add token to header

Filter

History

Collections

Trash

Test Collection 0 requests

POST https://test.salesforce.com/services/oauth2/token

POST https://test.salesforce.com/services/oauth2/token

Send

Save

Params

Authorization

Headers

Body

Pre-request Script

Tests

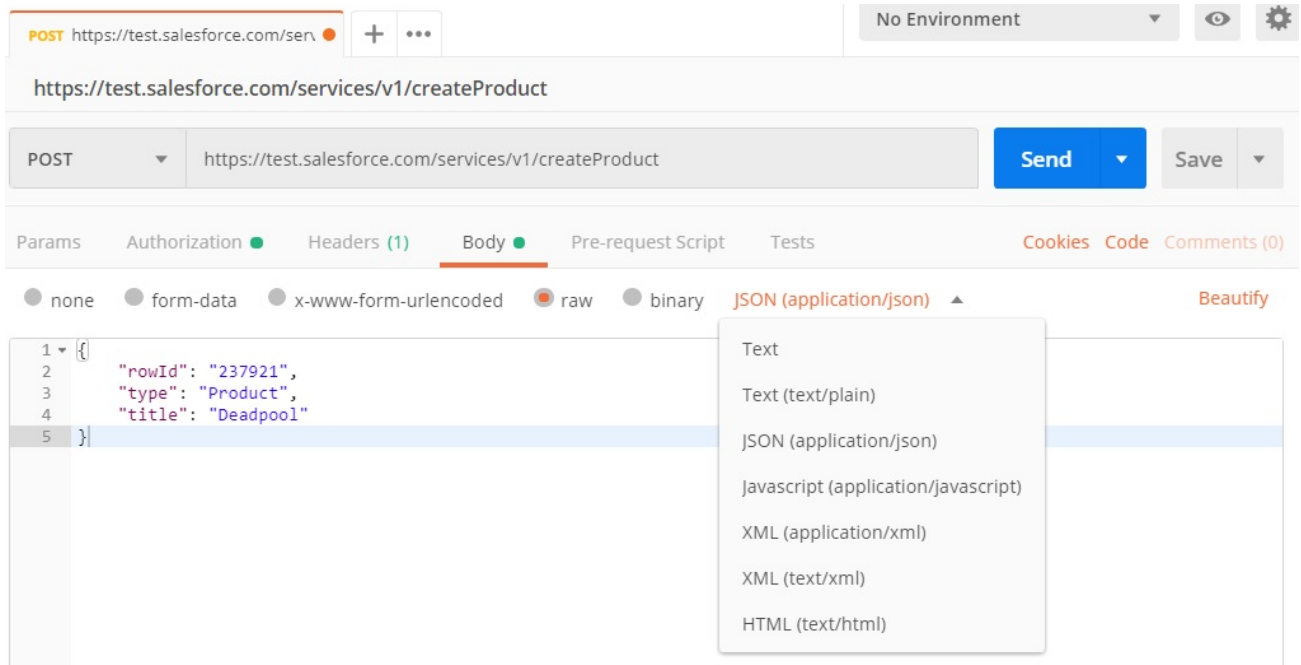
Cookies

Code

Comments (0)

6. Please select the token and Save the Request

7. Select the method you want to test . for example if you want to test Post Method select Post from drop down and Provide the complete URL of post method and provide the JSON request body as shown below



The screenshot shows a REST client interface with the following details:

- Method:** POST
- URL:** `https://test.salesforce.com/services/v1/createProduct`
- Environment:** No Environment
- Body Type:** JSON (application/json)
- Body Content:**

```
{
  "rowId": "237921",
  "type": "Product",
  "title": "Deadpool"
}
```
- Dropdown Menu Options:**
 - Text
 - Text (text/plain)
 - JSON (application/json)
 - Javascript (application/javascript)
 - XML (application/xml)
 - XML (text/xml)
 - HTML (text/html)

5.13.6 Platform Event/Streaming API

Platform Event Example

○ Event Creation

```

/*
go to setup -> develop -> Platform events -> Create new Object (Employee
Onboarding)->API->Employee_Onboarding__e
The following code used to publish platform events by using apex trigger A trigger processes platform event
notifications sequentially in the order they're received and trigger runs in its own process asynchronously and
isn't part of the transaction that published the event.
*/

trigger PlatformEventPublish on Account (after insert , after update ) {

    If(trigger.isAfter && trigger.isUpdate){
        List<Employee_On_boarding__e> publishEvents = new List<Employee_On_boarding__e>();
        for(Account a : Trigger.new){
            Employee_On_boarding__e eve = new Employee_On_boarding__e();
            eve.Name__c = a.Name ;
            eve.Phone__c = a.Phone ;
            eve.Salary__c = a.AnnualRevenue ;
            publishEvents.add(eve);
        }
        //Used to publish the events
        if(publishEvents.size()>0){
            EventBus.publish(publishEvents);
        }
    }
}

```

○ Handling Event in VF Page

<!-- Below is a sample Console component that receives a platform event to open a console component
The component handles an inbound call for the user to take action -->

```

<apex:page controller="PHX_processInboundEvent">
    <apex:slds />
    <div class="slds-grid" >
        <div class=" slds-size_1-of-12" >
            </div>

```

```

<div id="eventDetails" class="slds-p-around_small slds-text-title_bold slds-text-align_center
slds-size_10-of-12" >
    <lightning-formatted-text>Awaiting incoming call details</lightning-formatted-text>
</div>
<div class=" slds-size_1-of-12">
</div>
</div>
<script type="text/javascript" src="{!URLFOR ($Resource.StreamingAPI,
'/js/jquery-3.3.1.min.js')}"></script>
<script type="text/javascript" src="{!URLFOR ($Resource.StreamingAPI, '/js/cometd.js')}"></script>
<script type="text/javascript" src="{!URLFOR ($Resource.StreamingAPI, '/js/json2.js')}"></script>
<script type="text/javascript" src="{!URLFOR ($Resource.StreamingAPI, '/js/jquery.cometd.js')}"></script>
<apex:includeScript value="/support/console/42.0/integration.js" />
<script type="text/javascript">
console.log('loading Inbound Call Component');
$.cometd.websocketEnabled = false;
// logic to initialize the handler on the load of the page
$(document).ready(function() {
    $.cometd.init({
        url: window.location.protocol + '//' + window.location.hostname + '/cometd/41.0/',
        requestHeaders: {
            Authorization: 'OAuth {!$Api.Session_ID}'
        }
    });
    //subscribes to the box create event and calls loadData when event is fired
    $.cometd.subscribe('/event/Inbound_Call__e', function(message) {
        console.log(message);
        var data = message['data'];
        if (data != null && data['payload'] != null) {
            var conversationId = message['data']['payload'].conversationId__c;
            var recId = '0011U000004jQROQA2';
            console.log($('#eventDetails'));
            var htmlText = '<lightning-formatted-text>Receiving incoming call, conversation Id : ' +
                conversationId + '</lightning-formatted-text>';
            document.querySelector('#eventDetails').innerHTML = htmlText;
            processJacadaEvent.saveConversation(recId, conversationId, function(result, event) {
                if (event.status) {
                    console.log(result);
                    OpenPrimaryTab();
                    setCustomConsoleComponentWindowVisible();
                } else {
                    console.log('Error Processing the Request : ' + event.message);
                }
            }

```

```

        }, {
            escape: false
        });

    }
});
});

metaHandshakeListener = $.cometd.addListener('/meta/handshake', function(message) {
    if (message.successful) {
        console.log('<br><br> DEBUG: Handshake Successful: ' +
            JSON.stringify(message) + ' <br><br>');
    } else
        console.log('DEBUG: Handshake Unsuccessful: ' +
            JSON.stringify(message) + ' <br><br>');
});

function setCustomConsoleComponentWindowVisible() {
    //Make the custom console component window visible
    sforce.console.setCustomConsoleComponentVisible(true);
}
// function that opens the account after receiving the call
function OpenPrimaryTab() {
    //Open a new primary tab with the salesforce.com home page in it
    sforce.console.openPrimaryTab(null,

'https://SFDC-NAMESPACE.lightning.force.com/lightning/r/Account/0011U000004jQROQA2/view',
        true,
        ", openSuccess, 'salesforceTab');
}

var openSuccess = function openSuccess(result) {
    //Report whether opening the new tab was successful

    if (result.success == true) {
        console.log('Primary tab successfully opened');
    } else {
        console.log(result);
        console.log('Primary tab cannot be opened');
    }
};
</script>

```

```

</apex:page>

// Controller methods to fetch the account based on the external Id
Controller :
public without sharing class PHX_processInboundEvent {
    @AuraEnabled
    public static String getConversationId(String recId) {
        return [SELECT Id, conversationId__c FROM account where id=:recId][0].conversationId__c;
    }

    @AuraEnabled
    public static String saveConversationId(String recId,String conversationId) {
        account acclnst = new account();
        acclnst.Id=recId;
        acclnst.conversationId__c=conversationId;
        update acclnst;
        return 'Success';
    }

    @RemoteAction
    public static String saveConversation(String recId,String conversationId) {
        account acclnst = new account();
        acclnst.Id=recId;
        acclnst.conversationId__c=conversationId;
        update acclnst;
        return 'Success';
    }
}

```

○ Handling Event in Lightning Component

```

// component
<aura:component implements="flexipage:availableForAllPageTypes,force:hasRecordId" access="global"
controller="processInboundEvent">
    <lightning:empApi aura:id="empApi"/>
    <lightning:workspaceAPI aura:id="workspace" />
    <aura:handler name="init" value="{!this}" action="{!c.onInit}"/>
    <aura:attribute name="conversationId" type="String" default=""/>
    <aura:attribute name="channel" type="String" default="/event/Inbound_Call__e"/>
    <aura:attribute name="subscription" type="Map"/>

    <lightning:layout>

```



```

    <lightning:layoutItem size="2" class="slds-p-around_small"></lightning:layoutItem>
    <lightning:layoutItem size="8" class="slds-p-around_small slds-text-title_bold slds-text-align_center">

        <aura:if isTrue="{!v.conversationId!}">
            <lightning:formattedText value="{!'Receiving incoming call, conversation Id : '+ v.conversationId}"
        />

        <aura:set attribute="else">
            <lightning:formattedText value="{!'Awaiting incoming call details}" />
        </aura:set>
    </aura:if>
</lightning:layoutItem>
<lightning:layoutItem size="2" class="slds-p-around_small"></lightning:layoutItem></lightning:layout>

</aura:component>

// Controller
({
    // Called when the component is initialized.
    // Subscribes to the channel and displays a toast message.
    // Adds event listener for page unload to call unsubscribe().
    onInit: function (component, event, helper) {
        component.set('v.subscription', null);
        component.set('v.notifications', []);
        // Register error listener for the empApi component.
        const empApi = component.find('empApi');
        // Error handler function that prints the error to the console.
        const errorHandler = function (message) {
            console.error('Received error ', JSON.stringify(message));
        };
        // Register error listener and pass in the error handler function.
        empApi.onError($A.getCallback(errorHandler));
        helper.subscribe(component, event, helper);
        helper.displayToast(component, 'success', 'Ready to receive notifications.');
```

```

    // Helper
    ({
        // Client-side function that invokes the subscribe method on the
        // empApi component.
        subscribe: function (component, event, helper) {
```

```

// Get the empApi component.
const empApi = component.find('empApi');
// Get the channel from the attribute.
const channel = component.get('v.channel');
// Subscription option to get only new events.
const replayId = -1;
// Callback function to be passed in the subscribe call.
// After an event is received, this callback prints the event
// payload to the console. A helper method displays the message
// in the console app.
const callback = function (message) {
    console.log('Event Received : ' + JSON.stringify(message));
    helper.onReceiveNotification(component, message);
    helper.openTab(component, message);
};
// Subscribe to the channel and save the returned subscription object.
empApi.subscribe(channel, replayId, $A.getCallback(callback)).then($A.getCallback(function
(newSubscription) {
    console.log('Subscribed to channel ' + channel);
    component.set('v.subscription', newSubscription);
})));
},
// Client-side function that invokes the unsubscribe method on the
// empApi component.
unsubscribe: function (component, event, helper) {
    // Get the empApi component.
    const empApi = component.find('empApi');
    // Get the channel from the component attribute.
    const channel = component.get('v.subscription').channel;
    // Callback function to be passed in the unsubscribe call.
    const callback = function (message) {
        console.log('Unsubscribed from channel ' + message.channel);
    };
    // Unsubscribe from the channel using the subscription object.
    empApi.unsubscribe(component.get('v.subscription'), $A.getCallback(callback));
},
// Client-side function that displays the platform event message
// in the console app and displays a toast if not muted.
onReceiveNotification: function (component, message) {
    // Extract notification from platform event
    try {
        var data = message['data'];
        if(data!=null && data['payload'] !=null){

```

```

var conversationId = message['data']['payload'].conversationId__c;
component.set('v.conversationId', conversationId);
var recId = "0011U000004jQROQA2";
//component.set("v.standardDetails",null);
var action = component.get("c.saveConversationId");
action.setParams({
    "recId": recId,
    "conversationId": conversationId
});
action.setCallback(this, function(response) {
    var state = response.getState();
    if (state == "SUCCESS") {
        console.log('saveConversationId', response.getReturnValue());
    } else if (state == "INCOMPLETE") {
        // do something
    } else if (state == "ERROR") {
        var errors = response.getError();
        if (errors) {
            if (errors[0] && errors[0].message) {
                console.log("Error message: " + errors[0].message);
            }
        } else {
            console.log("Unknown error");
        }
    }
});

$A.enqueueAction(action);
}
} catch (e) {
    console.log(e.stack);
}
},
openTab: function (component, message) {
    console.log("The recordId for this tab is: ");
    var workspaceAPI = component.find("workspace");
    workspaceAPI.openTab({
        pageReference: {
            "type": "standard__recordPage",
            "attributes": {
                "recordId": "0011U000004jQROQA2",
                "actionName": "view"
            }
        },
    },

```

```
        "state": {}
    },
    focus: true
  }).then(function(response) {
    workspaceAPI.getTabInfo({
      tabId: response
    }).then(function(tabInfo) {
      console.log("The recordId for this tab is: " + tabInfo.recordId);
    });
  }).catch(function(error) {
    console.log(error);
  });
}

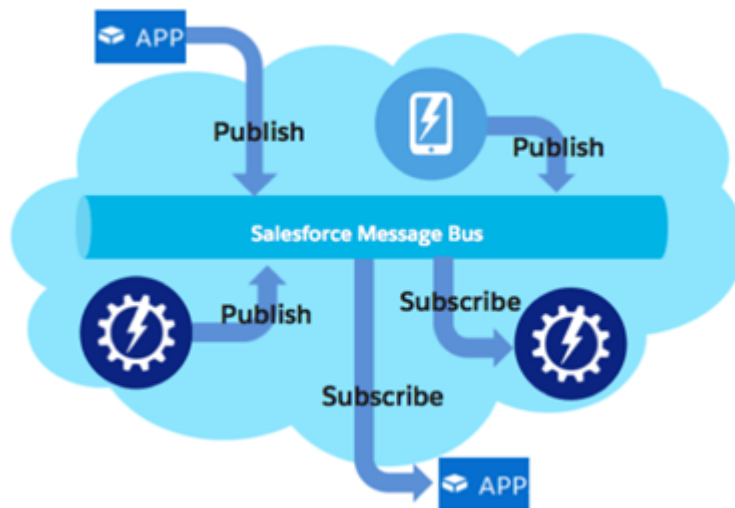
})
```

Platform Events

The Salesforce enterprise messaging platform offers the benefits of event-driven software architectures. Platform events are the event messages (or notifications) that your apps send and receive to take further action. Platform events simplify the process of communicating changes and responding to them without writing complex logic. Publishers and subscribers communicate with each other through events. One or more subscribers can listen to the same event and carry out actions.

Salesforce enterprise messaging platform provides the delivery of secure and scalable custom notifications within Salesforce and from external sources. With platform events, you can monitor your systems and communicate changes to other systems. Platform Events is event-based communication that revolves around a publisher-subscriber model—sender broadcasts a message that one or more receivers capture. They simplify the process of communicating changes and responding to them without requiring you to write complex logic. Publishers and subscribers communicate with each other through platform events. One or more subscribers can listen to the same event and carry out actions.

The following picture gives information on how new messaging system works



Publish Events subject name ends with “__e” instead of “__c”. A bigger difference is that event records are immutable—meaning that once an event has been published, it can’t be updated or deleted by a user—and it is not query-able using SOQL or SOSL. Instead, to receive events, you subscribe to a channel.

Unlike custom objects, platform events aren’t processed within database transactions in the Lightning platform. As a result, published platform events can’t be rolled back.

Things to remember while enabling the platform Events and Using :

- The allOrNoneHeader API header is ignored when publishing platform events through the API.
- The Apex setSavepoint() and rollback() Database methods aren’t supported with platform events. When publishing platform events, DML limits and other Apex governor limits apply
- When we publish event messages using process builder, make sure to add a Record Create or a Fast Create element to the appropriate flow. Where you’d usually pick an object to create, select the platform event.
- To publish event messages using Apex Class, call the EventBus.publish method instead of Publish Subject
 - e.g., results = EventBus.publish(msgEvents); // msgEvents are the list of msg_Events__e object
- We can use any Salesforce API to create platform events, such as SOAP API, REST API, or Bulk API.
- Triggers provide an auto subscription mechanism. No need to explicitly create and listen to a channel in Apex. To subscribe to event notifications, write an after insert trigger on the event object type. The after insert trigger corresponds to the time after a platform event is published. After an event message is published, the after trigger is fired.
- The publishing of high-volume platform events is asynchronous.
- Use high-volume platform events to publish and process millions of events efficiently and to scale your event-based apps.
- Event Retention - High-volume platform event messages are stored for 72 hours (3 days). Standard-volume platform event messages are stored for 24 hours (1 day). You can retrieve past event messages when using CometD clients to subscribe to a channel.

Limits :

- The synchronous limits apply to platform event triggers.
- platform event trigger runs in a separate transaction from the one that fired it, governor limits are reset, and trigger gets its own set of limits.

Cons :

- Platform Encryption is not supported for Platform Event fields
- When you delete an event definition, it's permanently removed and can't be restored
- You can't query event notifications using SOQL.
- When uninstalling a package with the option Save a copy of this package's data for 48 hours after uninstall enabled, platform events aren't exported
- Platform events aren't supported in Professional and Group Edition orgs
- No Record Page Support in Lightning App Builder

References:

https://developer.salesforce.com/docs/atlas.en-us.platform_events.meta/platform_events/platform_events_intro_6

https://developer.salesforce.com/docs/atlas.en-us.platform_events.meta/platform_events/platform_event_limits_h

Streaming API

Streaming API enables streaming of events using push technology and provides a subscription mechanism for receiving events in near real time. The Streaming API subscription mechanism supports multiple types of events, including PushTopic events, generic events, platform events, and Change Data Capture events.

Consider the following applications for Streaming API.

1. Applications That Poll Frequently : Applications that have constant polling action against the Salesforce infrastructure, consuming unnecessary API calls and processing time, would benefit from Streaming API because it reduces the number of requests that return no data.
2. General Notification : Use Streaming API for applications that require general notification of data changes in an org to reduce the number of API calls and improve performance.

How Streaming API works?

The Streaming API uses a publisher-subscriber (pub/sub) model to push data to the client. The first thing a developer must do is set up a PushTopic in Salesforce. This is basically a query that watches for changes in values of certain fields in a Salesforce Object. Then the client can subscribe to that PushTopic and receive updates whenever a change is made to the specified fields. This is done using a push protocol called the Bayeux protocol. The Salesforce implementation uses long polling connections to the server that will be able to notify the client on changes. This differs from traditional polling in that it keeps an incoming request open until an event is fired and then sends the response.

References:

https://developer.salesforce.com/docs/atlas.en-us.api_streaming.meta/api_streaming/intro_stream.htm

https://developer.salesforce.com/docs/atlas.en-us.api_streaming.meta/api_streaming/limits.htm

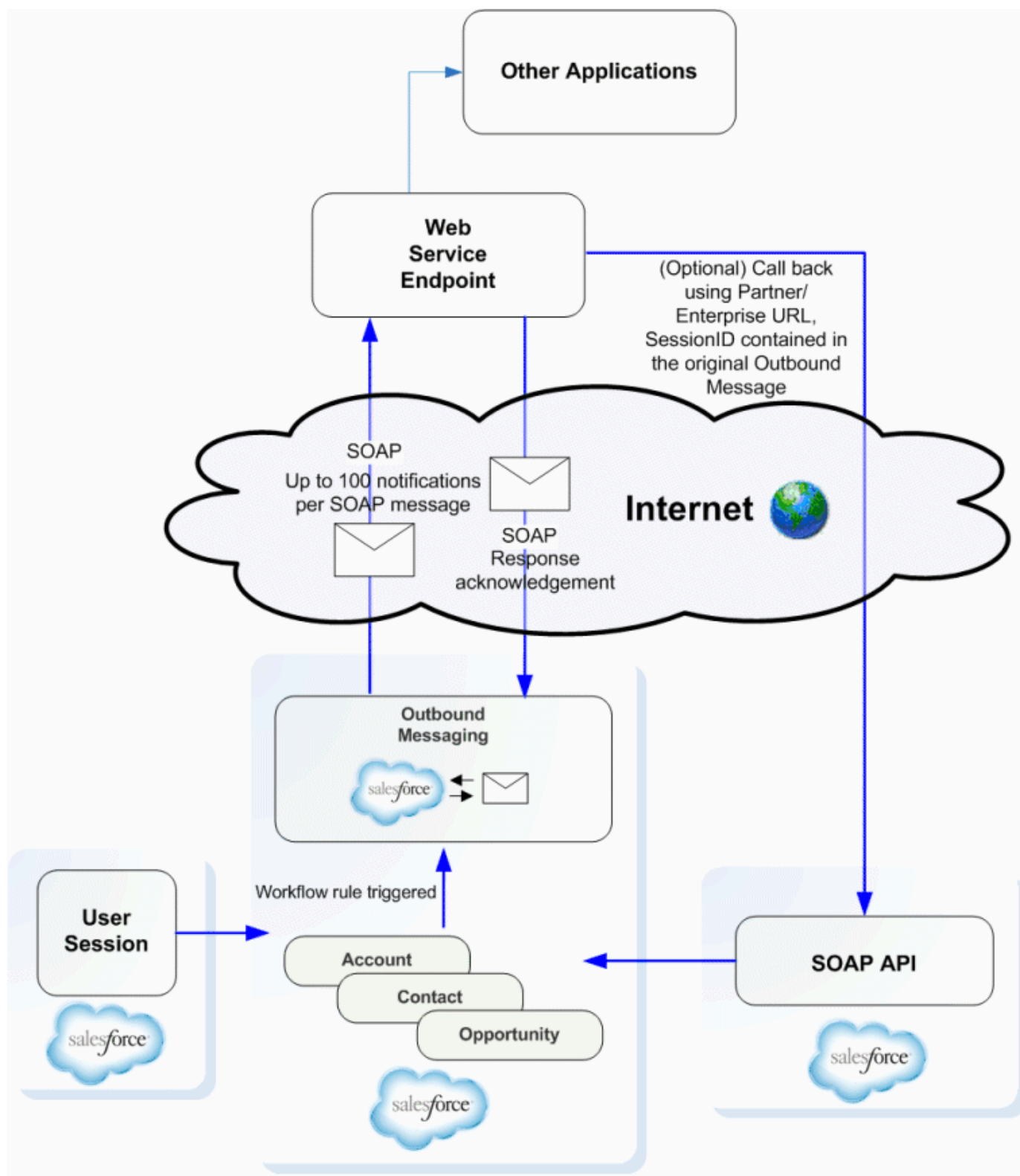
Deloitte.

5.13.7 Outbound Message

Outbound Message

Outbound messaging uses the `notifications()` call to send SOAP messages over HTTP(S) to a designated endpoint when triggered by a workflow rule.

Deloitte.



Workflow rules allow you to send outbound messages in XML format to external systems. Outbound messages can contain the field values. This is a great feature, but no support for sending JSON limits the usability.

Recommendations:

1. Follow the Naming Convention for Outbound Message Label
2. Always use "Integration User" as System Admin Profile for all the Outbound Message

When creating outbound messages for workflow rules or approval processes, consider the following:

- A single SOAP message can include up to 100 notifications. Each notification contains an ID that uniquely identifies a record, and a reference to the data in the record. Therefore, if the information in the record changes after the notification is sent, but before the notification is delivered, only the updated information is delivered.
- Messages are queued until they are sent, to preserve message reliability.
- If the endpoint is unavailable, messages will stay in the queue until sent successfully or until they are 24 hours old. After 24 hours, messages are dropped from the queue.
- If a message can't be delivered, the interval between retries increases exponentially, up to a maximum of two hours between retries.
- Messages are retried independent of their order in the queue. This may result in messages being delivered out of order.
- You can't build an audit trail using outbound messages. While each message should be delivered at least once, it may be delivered more than once. Also, it may not be delivered at all if delivery cannot be done within 24 hours. Finally, as noted above, the source object may change after a notification is sent but before it is delivered, so the endpoint will only receive the latest data, not any intermediate changes.
- No support for sending JSON
- Cannot configure the endpoint from external strings
- All field types are supported
- No support for delete operation

Considerations

- The endpoint must be capable of implementing a listener that can receive SOAP messages in predefined format sent from Salesforce.
- Each outbound message has its own predefined WSDL
- A single outbound message can contain up to 100 records. Therefore, it's important that the remote listener can handle more than a single record in its implementation

Security considerations

- Remote integration servers should white-list Salesforce server IP ranges. For an up-to-date list of Salesforce IP ranges, see [What are the Salesforce IP Addresses to whitelist?](#)
- The remote system should be appropriately protected by firewalls
- For outbound messaging, one-way SSL is enabled by default. However, two-way SSL can be used together with the Salesforce outbound messaging certificate.
- The organization Id is included in each message. Your client application should validate that messages contain your organization Id.

Error handling and Retry

- In the case of outbound messaging, Salesforce initiates retries if no positive acknowledgment is received from the outbound listener within the timeout period, for up to 24 hours. The retry interval increases exponentially over time, starting with 15-second intervals and ending with 60-minute intervals. Administrators must monitor this queue for any messages exceeding the 24 hour delivery period and retry manually, if required.
- Because this pattern is asynchronous, the remote system needs to handle any error-handling.
- Ideally, the invocation of this remote service should be via a reliable messaging system (based on JMS or MQ, for example) Outbound messaging to ensure full end-to-end guaranteed delivery. The positive acknowledgement to the Salesforce outbound message should occur after the remote listener has successfully placed its own message on its local queue.
- Outbound messaging sends a unique ID per message and this ID remains the same for any retries. The remote system can track duplicate messages based on this unique ID. The unique record ID for each record being updated is also sent, and can be used to prevent duplicate record creation

Outbound Messaging and Deletes

- Salesforce workflow rules can't track deletion of a record, it can only track the insert or update of a record. Therefore, it's not possible to directly initiate an outbound message from the deletion of a record. You can do this indirectly with the following process:
 - Create a custom object to store key information from the deleted records.
 - Create an Apex trigger, fired by the deletion of the base record, to store information such as the unique identifier in the custom object.
 - Implement a workflow rule to initiate an outbound message based on the creation of the custom object record. It's important that state tracking is enabled either by storing the remote system's unique identifier in Salesforce or storing the Salesforce unique identifier in the remote system

5.13.8 Continuation

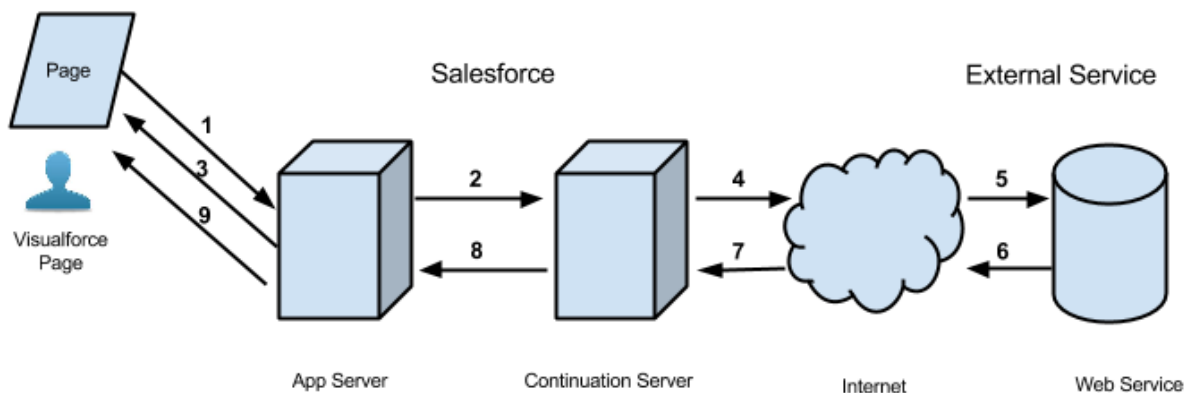
Overview

Use asynchronous callouts to make long-running requests from a Visualforce page to an external Web service and process responses in callback methods. Asynchronous callouts that are made from a Visualforce page don't count toward the Apex limit of 10 synchronous requests that last longer than five seconds. As a result, you can make more long-running callouts and you can integrate your Visualforce pages with complex back-end assets.

An asynchronous callout is a callout that is made from a Visualforce page for which the response is returned through a callback method. An asynchronous callout is also referred to as a continuation.

This diagram shows the execution path of an asynchronous callout, starting from a Visualforce page. A user invokes an action on a Visualforce page that requests information from a Web service (step 1). The app server hands the callout request to the Continuation server before returning to the Visualforce page (steps 2–3). The Continuation server sends the request to the Web service and receives the response (steps 4–7), then hands the response back to the app server (step 8). Finally, the response is returned to the Visualforce page (step 9).

Execution Flow of an Asynchronous Callout



- You can make up to three asynchronous callouts in a single continuation. Add these callout requests to the same continuation by using the `addHttpRequest` method of the Continuation class. The callouts run in parallel for this continuation and suspend the Visualforce request. Only after the external service returns all callouts, the Visualforce process resumes.
- Asynchronous callouts are supported only through a Visualforce page. Making an asynchronous callout by invoking the action method outside a Visualforce page, such as in the Developer Console, isn't supported.

Continuation-Specific Limits

The following are Apex and Visualforce limits that are specific to a continuation.

Description	Limit
Maximum number of parallel Apex callouts in a single continuation	3
Maximum number of chained Apex callouts	3
Maximum timeout for a single continuation	120 seconds
Maximum Visualforce controller-state size	80 KB
Maximum HTTP response size	1 MB
Maximum HTTP POST form size—the size of all keys and values in the form	1 MB
Maximum number of keys in the HTTP POST form	500

Continuation in Lightning

At a high level, this approach works like this:

- The Lightning Component embeds a Visualforce page (in an iframe).
- Using the standard [window.postMessage\(\)](#) API, the Lightning Component instructs the iframed Visualforce page to invoke the Apex continuation on its behalf.
- The Visualforce page invokes the continuation using Javascript Remoting.
- When the request completes, the Visualforce page uses the same [window.postMessage\(\)](#) API to pass the result back to its Lightning Component container.

For detailed implementation refer the the detailed approach in the below link -

<https://developer.salesforce.com/blogs/2017/09/invoking-apex-continuations-lightning-components.html>

Multiple Callout - Continuation Example

- Visual Force Controller

```
public with sharing class ChainedContinuationController {

    // Unique label for the initial callout request
    public String requestLabel1;
    // Unique label for the chained callout request
    public String requestLabel2;
    // Result of initial callout
    public String result1 {get;set;}
    // Result of chained callout
    public String result2 {get;set;}
    // Endpoint of long-running service
```

```
private static final String LONG_RUNNING_SERVICE_URL1 =
    '<Insert your first service URL>';
private static final String LONG_RUNNING_SERVICE_URL2 =
    '<Insert your second service URL>';

// Action method
public Object invokeInitialRequest() {
    // Create continuation with a timeout
    Continuation con = new Continuation(60);
    // Set callback method
    con.continuationMethod='processInitialResponse';

    // Create first callout request
    HttpRequest req = new HttpRequest();
    req.setMethod('GET');
    req.setEndpoint(LONG_RUNNING_SERVICE_URL1);

    // Add initial callout request to continuation
    this.requestLabel1 = con.addHttpRequest(req);

    // Return the continuation
    return con;
}

// Callback method for initial request
public Object processInitialResponse() {
    // Get the response by using the unique label
    HttpResponse response = Continuation.getResponse(this.requestLabel1);
    // Set the result variable that is displayed on the Visualforce page
    this.result1 = response.getBody();

    Continuation chainedContinuation = null;
    // Chain continuation if some condition is met
    if (response.getBody().toLowerCase().contains('expired')) {
        // Create a second continuation
        chainedContinuation = new Continuation(60);
        // Set callback method
        chainedContinuation.continuationMethod='processChainedResponse';

        // Create callout request
        HttpRequest req = new HttpRequest();
        req.setMethod('GET');
        req.setEndpoint(LONG_RUNNING_SERVICE_URL2);
```

```

        // Add callout request to continuation
        this.requestLabel2 = chainedContinuation.addHttpRequest(req);
    }

    // Start another continuation
    return chainedContinuation;
}

// Callback method for chained request
public Object processChainedResponse() {
    // Get the response for the chained request
    HttpResponse response = Continuation.getResponse(this.requestLabel2);
    // Set the result variable that is displayed on the Visualforce page
    this.result2 = response.getBody();

    // Return null to re-render the original Visualforce page
    return null;
}
}

```

○ Visual Force Page

```

<apex:page controller="ChainedContinuationController" showChat="false" showHeader="false">
    <apex:form >
        <!-- Invokes the action method when the user clicks this button. -->
        <apex:commandButton action="{!invokeInitialRequest}" value="Start Request" reRender="panel"/>
    </apex:form>

    <apex:outputPanel id="panel">
        <!-- Displays the response body of the initial callout. -->
        <apex:outputText value="{!result1}" />

        <br/>
        <!-- Displays the response body of the chained callout. -->
        <apex:outputText value="{!result2}" />
    </apex:outputPanel>
</apex:page>

```


5.13.9 Change Data Capture

Change Event

Change Data Capture is a streaming product on the Lightning Platform that enables you to efficiently integrate your Salesforce data with external systems. With Change Data Capture, you can receive changes of Salesforce records in real time and synchronize corresponding records in an external data store.

A Change Data Capture event, or change event, is a notification that Salesforce sends when a change to a Salesforce record occurs as part of a create, update, delete, or undelete operation. The notification includes all new and changed fields, and header fields that contain information about the change. For example, header fields indicate the type of change that triggered the event and the origin of the change. Change events support all custom objects and a subset of standard objects.

Use Change Data Capture to update data in an external system instead of doing periodic exports or API polling. Capturing changes with Change Data Capture event notifications ensures that your external data can be updated in real time and stays fresh.

When to use Change Data Capture

Use change events to:

- Receive notifications of Salesforce record changes, including create, update, delete, and undelete operations.
- Capture field changes for all records.
- Get broad access to all data regardless of sharing rules.
- Get information about the change in the event header, such as the origin of the change, which allows ignoring changes that your client generates.
- Perform data updates using transaction boundaries.
- Use a versioned event schema.
- Subscribe to mass changes in a scalable way.
- Get access to retained events for up to three days.

Considerations

- Change Data Capture ignores sharing settings and sends change events for all records of a Salesforce object. To receive change events, the subscribed user must have one or more permissions depending on the channel that is subscribed to. The following table outlines the required permissions per channel.
- Change Data Capture respects your org's field-level security settings. Delivered events contain only the fields that a subscribed user is allowed to view. Before delivering a change event for an object, the subscribed user's field permissions are checked. If a subscribed user has no access to a field, the field isn't included in the change event message that the subscriber receives.

- If Salesforce record fields are encrypted with Shield Platform Encryption, changes in encrypted field values generate change events.
- The following actions don't generate change events for affected records.
- Any action related to state and country picklists that you perform in Setup on the State and Country/Territory Picklists page.
 - Changing the type of an opportunity stage picklist value.
 - When a custom picklist field is defined on Contact in a person account org, the field is present on Account with the __pc suffix. Replacing or renaming a value of the custom picklist doesn't generate account change events but only contact change events for the affected records. However, if the custom picklist field is defined on Account, the field is not present on Contact and only account change events are generated, as expected.

References

- [Change Data Capture Developer Guide](#)
- [Streaming API Developer Guide](#)
- [Change Data Capture Trailhead Link](#)

Deloitte.

5.13.10 Integration APIs

SOAP API

Use SOAP API to create, retrieve, update or delete records, such as accounts, leads, and custom objects. With 20 different calls, SOAP API also allows you to maintain passwords, perform searches, and much more. Use SOAP in any language that supports Web services.

The WSDL file defines the Web service that is available to you. Your development platform uses this WSDL to generate the API to access the Lightning Platform Web service it defines. You can either obtain the WSDL file from your organization's Salesforce administrator or you can generate it yourself if you have access to the WSDL download page in the Salesforce user interface. You can navigate to the most recent WSDL for your organization from Setup by entering API in the search box, then selecting API.

Salesforce provides a WSDL (Web Service Description Language) files. They are called 'Enterprise WSDL' and 'Partner WSDL'. A WSDL is an XML-document which contains a standardized description on how to communicate using a web service (the Salesforce API is exposed as a web service). The WSDL is used by developers to aid in the creation of Salesforce integration pieces. A typical process involves using the Development Environment (eg, Eclipse for Java or IntelliJ for .Net) to consume the WSDL and generate classes which are then referenced in the integration.

Enterprise WSDL - Enterprise WSDL is intended primarily for Customers

1. The Enterprise WSDL is strongly typed.
2. The Enterprise WSDL is tied (bound) to a specific configuration of Salesforce (ie. a specific organization's configuration).
3. The Enterprise WSDL changes if modifications (e.g custom fields or custom objects) are made to an organization's Salesforce configuration.

Partner WSDL - Partner WSDL is intended primarily for Partners

1. The Partner WSDL is loosely typed.
2. The Partner WSDL can be used to reflect against/interrogate any configuration of Salesforce (ie. any organization's Salesforce configuration).
3. The Partner WSDL is static, and hence does not change if modifications are made to an organization's Salesforce configuration.
4. This API is for developers who are creating client applications for multiple organizations, or for developers who develop more flexible integrations.

Download a WSDL file when logged into Salesforce

1. Click Setup | Develop | API
2. Click the link to download the appropriate WSDL.

3. Save the file locally, giving the file a ".wsdl" extension.

For more details on the SOAP API refer to the developer guide -

https://developer.salesforce.com/docs/atlas.en-us.api.meta/api/sforce_api_quickstart_intro.htm

For Limits and considerations refer to the below link -

https://developer.salesforce.com/docs/atlas.en-us.218.0.api.meta/api/implementation_considerations.htm?Search

Rest API

REST API provides a powerful, convenient, and simple REST-based web services interface for interacting with Salesforce. Its advantages include ease of integration and development, and it's an excellent choice of technology for use with mobile applications and web projects. For certain projects, you may want to use REST API with other Salesforce REST APIs. To build UI for creating, reading, updating, and deleting records, including building UI for list views, actions, and dependent picklists use User Interface API. To build UI for Chatter, communities, or recommendations, use Chatter REST API. If you have many records to process, consider using Bulk API, which is based on REST principles and optimized for large sets of data.

REST API uses the same underlying data model and standard objects as those in SOAP API. See the SOAP API Developer Guide for details. REST API also follows the same limits as SOAP API. See the Limits section in the SOAP API Developer Guide.

Before making REST API calls, you must authenticate the application user using OAuth 2.0. To do so, you'll need to:

- [Set up your application as a connected app](#) in the Salesforce organization.
- Determine the correct Salesforce [OAuth endpoint](#) for your connected app to use.
- Authenticate the connected app user via one of several different OAuth 2.0 authentication flows. An OAuth authentication flow defines a series of steps used to coordinate the authentication process between your application and Salesforce. Supported OAuth flows include:
 - [Web server flow](#), where the server can securely protect the consumer secret.
 - [User-agent flow](#), used by applications that cannot securely store the consumer secret.
 - [Username-password flow](#), where the application has direct access to user credentials.

Composite Rest API :

Use composite resources to improve your application's performance by minimizing the number of round-trips between the client and server. It combines multiple operations in the same request. Below are some scenarios where it can be used :

- [Execute Dependent Requests in a Single API Call](#)
 - The following example uses the Composite resource to execute several dependent requests all in a single call. First, it creates an account and retrieves its information. Next, it uses the account data and the Composite resource's reference ID functionality to create a contact and populate its

fields based on the account data. Then it retrieves specific information about the account's owner by using query parameters in the request string. Finally, if the metadata has been modified since a certain date, it retrieves account metadata. The `composite.json` file contains the composite request and subrequest data.

- [Update an Account, Create a Contact, and Link Them with a Junction Object](#)
 - The following example uses the Composite resource to update some fields on an account, create a contact, and link the two records with a junction object called `AccountContactJunction`. All these requests are executed in a single call. The `composite.json` file contains the composite request and subrequest data.
- [Update a Record and Get Its Field Values in a Single Request](#)
 - Use the Batch resource to execute multiple requests in a single API call.
- [Create Nested Records](#)
 - Use the SObject Tree resource to create nested records that share a root record type. For example, in a single request, you can create an account along with its child contacts, and a second account along with its child accounts and contacts. Once the request is processed, the records are created and parents and children are automatically linked by ID. In the request data, you supply the record hierarchies, required and optional field values, each record's type, and a reference ID for each record, and then use the POST method of the resource. The response body will contain the IDs of the created records if the request is successful. Otherwise, the response contains only the reference ID of the record that caused the error and the error information.
- [Create Multiple Records](#)
 - While the SObject Tree resource can be used to create nested records, you can also create multiple, unrelated records of the same type. In a single request, you can create up to two hundred records. In the request data, you supply the required and optional field values for each record, each record's type, and a reference ID for each record, and then use the POST method of the resource. The response body will contain the IDs of the created records if the request is successful. Otherwise, the response contains only the reference ID of the record that caused the error and the error information.

Bulk API

The Bulk API provides a programmatic option to quickly load your org's data into Salesforce. Bulk API is based on REST principles and is optimized for loading or deleting large sets of data. You can use it to query, queryAll, insert, update, or delete many records asynchronously by submitting batches. Salesforce processes batches in the background.

SOAP API, in contrast, is optimized for real-time client applications that update a few records at a time. You can use SOAP API for processing many records, but when the data sets contain hundreds of thousands of records, SOAP API is not practical. Bulk API is designed to make it simple to process data from a few thousand to millions of records.

The easiest way to use Bulk API is to enable it for processing records in Data Loader using CSV files. Using Data Loader avoids the need to write your own client application.

The REST Bulk API lets you query, queryAll, insert, update, upsert, or delete a large number of records asynchronously. Bulk API records can include binary attachments, such as Attachment objects or Salesforce CRM Content. You first send

batches to the server using an HTTP POST call and then the server processes the batches in the background. While batches are being processed, you can track progress by checking the status of the job using an HTTP GET call. All operations use HTTP GET or POST methods to send and receive CSV, XML, or JSON data.

How Bulk API Works

You process a set of records by creating a job that contains one or more batches. The job specifies which object is processed and what type of operation is being used. A batch is a set of records sent to the server in an HTTP POST call. Each batch is processed independently by the server, not necessarily in the order it is received. Batches may be processed in parallel. It's up to the client to decide how to divide the entire data set into a suitable number of batches.

A job is represented by the JobInfo resource. This resource is used to create a new job, get status for an existing job, change status for a job. A batch is created by submitting a CSV, XML, or JSON representation of a set of records with references to binary attachments in an HTTP POST request. When created, the status of a batch is represented by the BatchInfo resource. When a batch is complete, the result for each record is available in a result set resource.

Processing data typically consists of the following steps.

1. Create a new job that specifies the object and action.
2. Send data to the server in a number of batches.
3. Once all data has been submitted, close the job. Once closed, no more batches can be sent as part of the job.
4. Check status of all batches at a reasonable interval. Each status check returns the state of each batch.
5. When all batches have either completed or failed, retrieve the result for each batch.
6. Match the result sets with the original data set to determine which records failed and succeeded, and take action.

At any point in this process, you can abort the job. Aborting a job has the effect of preventing any unprocessed batches from being processed. It doesn't undo the effects of batches already processed.

Guidelines for Data Loads

These are some tips for planning your data loads for optimal processing time. Always test your data loads in a sandbox organization first. Note that the processing times may be different in a production organization.

- **Use Parallel Mode Whenever Possible:** You get the most benefit from the Bulk API by processing batches in parallel mode, which is the default mode and enables faster loading of data. However, sometimes parallel processing can cause contention on records. The alternative is to process using serial mode. Don't process data in serial mode unless you know this would otherwise result in lock timeouts and you can't reorganize your batches to avoid the locks. The Bulk API supports parallel processing mode at the job level. All batches in a job are processed in parallel or serial mode.
- **Be Aware of Operations that Increase Lock Contention:** For example, when an AccountTeamMember record is created or updated, the account for this record is locked during the transaction. If you load many batches of AccountTeamMember records and they all contain references to the same account, they all try to lock the account and it's likely that you'll experience a lock timeout. Sometimes, lock timeouts can be avoided by organizing data in batches. If you organize AccountTeamMember records by AccountId so that all records referencing the same account are in a single batch, you minimize the risk of lock contention by multiple batches.

- **Minimize Number of Fields:** Processing time is faster if there are fewer fields loaded for each record. Foreign lookup relationship, and roll-up summary fields are more likely to increase processing time. It's not always reduce the number of fields in your records, but, if it is possible, loading times will improve.
- **Minimize Number of Workflow Actions:** Workflow actions increase processing time.
- **Minimize Number of Triggers:** You can use parallel mode with objects that have associated triggers if they don't cause side-effects that interfere with other parallel transactions. However, Salesforce doesn't recommend large batches for objects with complex triggers. Instead, you should rewrite the trigger logic as a batch Apex class executed after all the data has loaded.
- **Optimize Batch Size:** Salesforce shares processing resources among all its customers. To ensure that each organization doesn't have to wait too long to process its batches, any batch that takes more than 10 minutes is suspended and returned to the queue for later processing. Batch sizes should be adjusted based on processing time. Start with 5000 records and adjust the batch size based on processing time.
- **Minimize Number of Batches in the Asynchronous Queue:** Salesforce uses a queue-based framework to handle asynchronous processes from such sources as future and batch Apex, as well as Bulk API batches. This is used to balance request workload across organizations. If more than 2,000 unprocessed requests from a single organization are in the queue, any additional requests from the same organization will be delayed while the queue handles requests from other organizations. Minimize the number of batches submitted at one time to ensure batches are not delayed in the queue.

Configure the Data Loader to Use the Bulk API

The Bulk API is optimized to load or delete a large number of records asynchronously. It is faster than the SOAP API due to parallel processing and fewer network round-trips. By default, Data Loader uses the SOAP-based API to process records.

To configure Data Loader to use the Bulk API for inserting, updating, upserting, deleting, and hard deleting records:

1. Open the Data Loader.
 2. Choose Settings | Settings.
 3. Select the Use Bulk API option.
 4. Click OK.
- To use Bulk API to import data that was exported directly from Microsoft Outlook, Google Contacts, and other third-party sources, map data fields in any CSV import file to Salesforce data fields. The CSV import file must be Bulk API-compatible. Add a transformation spec (spec.csv) file to the Job that provides the instructions to map the data in your import file to Salesforce data.

Bulk Query

Use bulk query to efficiently query large data sets and reduce the number of API requests. A bulk query can retrieve up to 1 GB of data, divided into 15 1-GB files. The data formats supported are CSV, XML, and JSON.

If the query succeeds, Salesforce attempts to retrieve the results. If the results exceed the 1-GB file size limit or take more than 10 minutes to retrieve, the completed results are cached and another attempt is made. After 15 attempts, the error message “Retried more than fifteen times” is returned. In this case, consider using the PK Chunking to split the query results into smaller chunks. If the attempts succeed, the results are returned and stored for seven days.

PK Chunking Header

Salesforce recommends that you enable PK chunking when querying tables with more than 10 million records or when a query consistently times out. However, the effectiveness of PK chunking depends on the specifics of the query and the queried data.

Use the PK Chunking request header to enable automatic primary key (PK) chunking for a bulk query job. PK chunking splits bulk queries on very large tables into chunks based on the record IDs, or primary keys, of the queried records. Each chunk is processed as a separate batch that counts toward your daily batch limit, and you must download each batch's results separately. PK chunking works only with queries that don't include SELECT clauses or conditions other than WHERE.

PK chunking is supported for the following objects: Account, Campaign, CampaignMember, Case, CaseHistory, Contact, Event, EventRelation, Lead, LoginHistory, Opportunity, Task, User, and custom objects.

PK chunking works by adding record ID boundaries to the query with a WHERE clause, limiting the query results to a chunk of the total results. The remaining results are fetched with extra queries that contain successive boundaries. The number of records within the ID boundaries of each chunk is referred to as the chunk size. The first query retrieves records between a specified starting ID and the starting ID plus the chunk size. The next query retrieves the next chunk, and so on. For more details refer to

https://developer.salesforce.com/docs/atlas.en-us.api_async.meta/api_async/async_api_headers_enable_pk_chunking.htm

Limitations:

- API usage limits : Each HTTP request counts as one call for the purposes of calculating usage limits.
- Batch content : Each batch must contain exactly one CSV, XML, or JSON file containing records for a single object. If the batch is not processed and state Message is updated. Use the enterprise WSDL for the correct format for each object's records.
- Batch Allocation : You can submit up to 10,000 batches per rolling 24-hour period. You can't create batches associated with a job that is more than 24 hours old. If a batch is submitted to a closed job, the batch will not be created, however it will still count against the batch allocation as a submitted batch.
- Batch lifespan : Batches and jobs that are older than seven days are removed from the queue regardless of their status.
- Batch size : 10,000 Batches per rolling 24 hours.
- Bulk API query doesn't support the following SOQL:
 - COUNT
 - ROLLUP
 - SUM
 - GROUP BY CUBE
 - OFFSET
 - Nested SOQL queries

- For more limits refer to link
https://developer.salesforce.com/docs/atlas.en-us.api_async.meta/api_async/async_api_concepts_limitations.htm

Cons:

- The Client application is responsible for splitting the batches
- Doesn't support Roll Backs
- Must be tuned to avoid lock Errors (use Sort By MasterId)

Reference : https://developer.salesforce.com/docs/atlas.en-us.api_async.meta/api_async/async_api_intro.htm

Bulk API Example

```
/*
1) Create a Job :
Create a job by sending a POST request to the following URI. The request body identifies the type of object
processed in
all associated batches.
*/
URI: https://instance_name—api.salesforce.com/services/async/APIversion/job
```

Json Request :

```
{
  "operation" : "insert",
  "object" : "Account",
  "contentType" : "CSV"
}
```

JSON Response :

```
{
  "apexProcessingTime" : 0,
  "apiActiveProcessingTime" : 0,
  "apiVersion" : 36.0,
  "concurrencyMode" : "Parallel",
  "contentType" : "JSON",
  "createdById" : "005D00000001b0fFIAQ",
  "createdDate" : "2015-12-15T20:45:25.000+0000",
  "id" : "750D000000004SkGIAU",
  "numberBatchesCompleted" : 0,
  "numberBatchesFailed" : 0,
  "numberBatchesInProgress" : 0,
  "numberBatchesQueued" : 0,
  "numberBatchesTotal" : 0,
  "numberRecordsFailed" : 0,
```



```
"numberRecordsProcessed" : 0,
"numberRetries" : 0,
"object" : "Account",
"operation" : "insert",
"state" : "Open",
"systemModstamp" : "2015-12-15T20:45:25.000+0000",
"totalProcessingTime" : 0
}
/*
```

2) Add a Batch to a Job :

Add a new batch to a job by sending a POST request to the following URI. The request body contains a list of records for processing.

*/

URI

https://instance_name—api.salesforce.com/services/async/APIversion/job/jobid/batch

/*

3) Monitor Job :

You can monitor a Bulk API job in Salesforce. The monitoring page tracks jobs and batches created by any client application, including Data Loader or any client application that you write.

To track the status of bulk data load jobs that are in progress or recently completed, enter Bulk Data Load Jobs in the Quick Find box, then select Bulk Data Load Jobs. This page allows you to monitor the progress of current jobs and the results of recent jobs.

4) Close a Job :

Close a job by sending a POST request to the following URI. The request URI identifies the job to close.

When a job is closed, no more batches can be added.

*/

URI : https://instance_name—api.salesforce.com/services/async/APIversion/job/jobId

Example JSON Request :

```
{
  "state" : "Closed"
}
```

Example JSON Response :

```
{
  "apexProcessingTime" : 0,
  "apiActiveProcessingTime" : 5059,
  "apiVersion" : 36.0,
```

```

    "concurrencyMode" : "Parallel",
    "contentType" : "JSON",
    "createdById" : "005xx000001SyhGAAS",
    "createdDate" : "2015-11-19T01:45:03.000+0000",
    "id" : "750xx000000000GAAQ",
    "numberBatchesCompleted" : 10,
    "numberBatchesFailed" : 0,
    "numberBatchesInProgress" : 0,
    "numberBatchesQueued" : 0,
    "numberBatchesTotal" : 10,
    "numberRecordsFailed" : 0,
    "numberRecordsProcessed" : 100,
    "numberRetries" : 0,
    "object" : "Account",
    "operation" : "insert",
    "state" : "Closed",
    "systemModstamp" : "2015-11-19T01:45:03.000+0000",
    "totalProcessingTime" : 5759
  }
  /*

```

5) Get Job Details :

Get all details for an existing job by sending a GET request to the following URI.

*/

URI : https://instance_name—api.salesforce.com/services/async/APIVersion/job/jobId

Example JSON Response :

```

{
  "apexProcessingTime" : 0,
  "apiActiveProcessingTime" : 0,
  "apiVersion" : 36.0,
  "concurrencyMode" : "Parallel",
  "contentType" : "JSON",
  "createdById" : "005D0000001b0fFIAQ",
  "createdDate" : "2015-12-15T20:45:25.000+0000",
  "id" : "750D000000004SkGIAU",
  "numberBatchesCompleted" : 0,
  "numberBatchesFailed" : 0,
  "numberBatchesInProgress" : 0,
  "numberBatchesQueued" : 0,
  "numberBatchesTotal" : 0,
  "numberRecordsFailed" : 0,
  "numberRecordsProcessed" : 0,

```

```

    "numberRetries" : 0,
    "object" : "Account",
    "operation" : "insert",
    "state" : "Open",
    "systemModstamp" : "2015-12-15T20:45:25.000+0000",
    "totalProcessingTime" : 0
  }
  /*

```

6) Abort a Job : Abort an existing job by sending a POST request to the following URI. The request URI identifies the job to abort. When a job is aborted, no more records are processed. If changes to data have already been committed, they aren't rolled back.

```

  */

```

URI : https://instance_name—api.salesforce.com/services/async/APIVersion/job/jobId

```

{
  "state" : "Aborted"
}

```

Example JSON response body

```

{
  "apexProcessingTime" : 0,
  "apiActiveProcessingTime" : 2166,
  "apiVersion" : 36.0,
  "concurrencyMode" : "Parallel",
  "contentType" : "JSON",
  "createdById" : "005D00000001b0fFIAQ",
  "createdDate" : "2015-12-15T21:54:29.000+0000",
  "id" : "750D00000004SkVIAU",
  "numberBatchesCompleted" : 2,
  "numberBatchesFailed" : 0,
  "numberBatchesInProgress" : 0,
  "numberBatchesQueued" : 0,
  "numberBatchesTotal" : 2,
  "numberRecordsFailed" : 0,
  "numberRecordsProcessed" : 2,
  "numberRetries" : 0,
  "object" : "Account",
  "operation" : "insert",
  "state" : "Aborted",
  "systemModstamp" : "2015-12-15T21:54:29.000+0000",
  "totalProcessingTime" : 2870
}

```

```

}
/*
7) Job and Batch Lifespan : All jobs and batches older than seven days are deleted.
It may take up to 24 hours for jobs and batches to be deleted once they are older than seven days.
If a job is more than seven days old, but contains a batch that is less than seven days old, then all of the
batches
associated with that job, and the job itself, aren't deleted until the youngest batch is more than seven days
old.
Jobs and batches are deleted regardless of status.
Once deleted, jobs and batches can't be retrieved from the platform.
*/

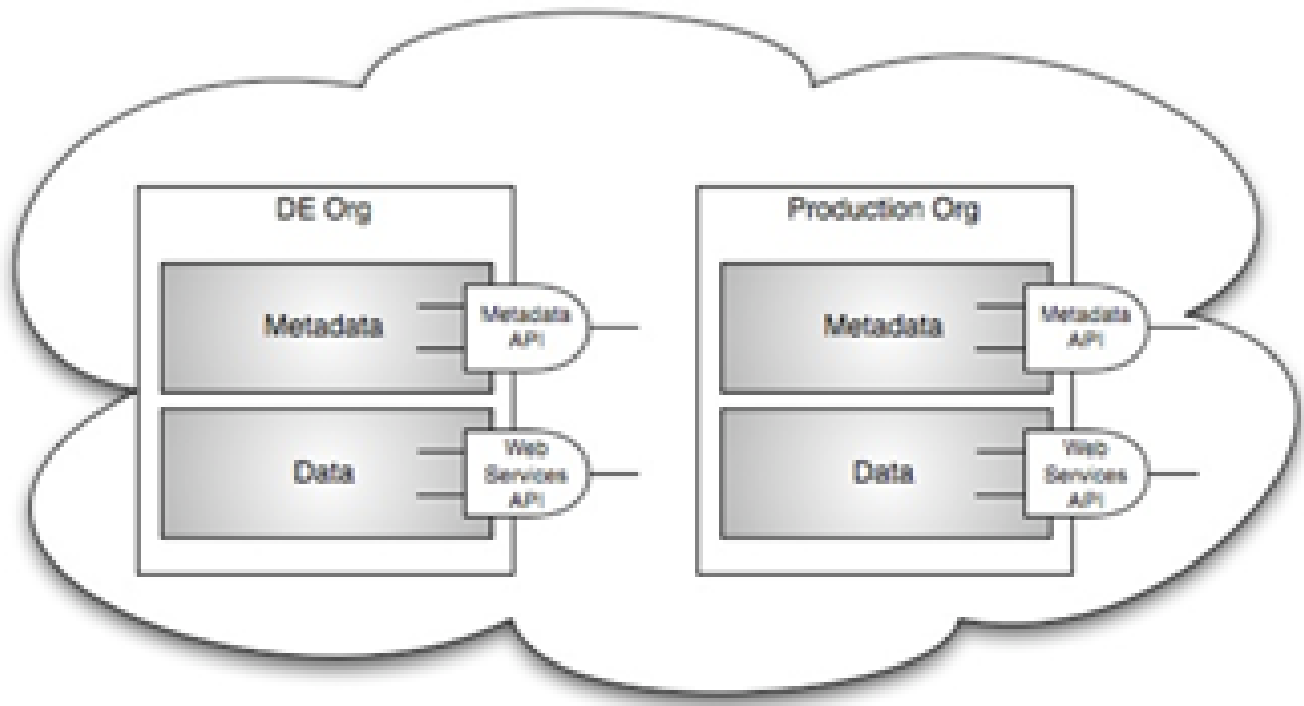
```

Metadata API

Use Metadata API to retrieve, deploy, create, update, or delete customizations for your org. Using Metadata API we can retrieve, deploy, create and update the Custom Objects definitions and Page Layouts of the Org. It allows us to get XML Version of your Org. It is SOAP based API and supports both synchronous and asynchronous invocations. The most common use is to migrate changes from a sandbox or testing org to your production environment. Metadata API is intended for managing customizations and for building tools that can manage the metadata model, not the data itself.

The easiest way to access the functionality in Metadata API is to use the Force.com IDE or Ant Migration Tool. Both tools are built on top of Metadata API and use the standard Eclipse and Ant tools, respectively, to simplify working with Metadata API.

- Force.com IDE is built on the Eclipse platform, for programmers familiar with integrated development environments. Code, compile, test, and deploy from within the IDE.
- The Ant Migration Tool is ideal if you use a script or the command line for moving metadata between a local directory and a Salesforce org.



References : https://developer.salesforce.com/docs/atlas.en-us.api_meta.meta/api_meta/meta_intro.htm

Tooling API

Tooling API exposes metadata used in developer tooling that you can access through REST or SOAP. Use Tooling API when you need fine-grained access to an org's metadata. Tooling API's SOQL capabilities for many metadata types allow you to retrieve smaller pieces of metadata. Smaller retrieves improve performance, which makes Tooling API a better fit for developing interactive applications.

We can use this to develop an interactive application or tools for developers. This can be used to fetch the metadata such as Apex classes, Apex triggers, custom objects, custom fields, Visualforce Pages, Users, Apex Component etc. Tooling API provides both REST and SOAP interfaces.

For example, you can:

- Add features and functionality to your existing Lightning Platform tools.
- Build dynamic modules for Lightning Platform development into your enterprise integration tools.
- Build specialized development tools for a specific application or service.

Because Tooling API allows you to change just one element within a complex type, it can be easier to use than Metadata API. Other use cases include:

- Source control integration
- Continuous integration
- Apex classes or trigger deployment
- Access Apex Code Coverage

- SOQL : SELECT Coverage FROM ApexCodeCoverage WHERE ApexClassOrTrigger = 'TriggerId' AND ApexTestClass = 'TestClassId'
- Code coverage percentage is a simple calculation of the number of covered lines divided by the sum of the number of covered lines and the number of uncovered lines.

References : https://developer.salesforce.com/docs/atlas.en-us.api_tooling.meta/api_tooling/

Tooling API Example

```
// Class variable for Tooling API base URL
private static String baseUrl = URL.getSalesforceBaseUrl().toExternalForm()
    + '/services/data/v45.0/tooling/';

public static void createClass() {
    // Note the escaping on newlines and quotes
    String classBody = 'public class MyNewClass {\n'
        + ' public string SayHello() {\n'
        + ' return \'Hello\';\n'
        + ' }\n'
        + '};

    HTTPRequest req = new HTTPRequest();
    req.setEndpoint(baseUrl + 'subjects/ApexClass');
    req.setMethod('POST');
    // OAuth header
    req.setHeader('Authorization', 'Bearer ' + UserInfo.getSessionId());
    req.setHeader('Content-Type', 'application/json');
    req.setBody('{'+
        ""Body" : "" + classBody +""+
    '}');

    Http h = new Http();
    HttpResponse res = h.send(req);
    // Response to a create should be 201
    if (res.getStatusCode() != 201) {
        System.debug(res.getBody());
        throw new MyException(res.getStatus());
    }
}
```

6. SECURITY STANDARDS

Security Standards

The Lightning platform offers robust protection against most security concerns. However, there are areas where code can introduce vulnerabilities in the platform. This is particularly true when Salesforce is used to integrate Web applications with Salesforce, either through web service calls or with Visualforce and iFrame “Mash-ups”.

General information on Web Application security can be found here:

https://www.owasp.org/index.php/OWASP_Secure_Coding_Practices_-_Quick_Reference_Guide

- When writing Salesforce applications, the following design principles should be followed as best practices for security.
- To protect sensitive data, use the “transient” keyword to declare instance variables within Visualforce controllers so they are not transmitted as part of the view state.
- Explicitly initialize all your variables and other data stores, either during declaration or just before the first use.
- In cases where the application must run with elevated privileges, raise privileges as late as possible, and close them as soon as possible.
- Connection strings should not be hard coded within the application. Connection strings should be stored in a protected custom setting.
- Do not include sensitive information in HTTP GET request parameters to protect against CSRF attacks (not always a GET request).
- Disable escaping on VisualForce tags unless it's absolutely necessary.
- Conduct all data validation inside the Apex Class, do not validate data in the client-side script. All validation failures should result in input rejection.
- Specify proper character sets, such as UTF-8, for all sources of input. If any potentially hazardous characters are allowed as input, be sure that you implement additional controls like output encoding, secure task specific accounting for the utilization of that data throughout the application. Examples of common hazardous characters are: `> " ' % () & + \ ' \`.
- Avoid dynamic SOQL wherever possible to avoid SOQL injection. If it's necessary, then use the `escapeSingleQuotes` method to sanitize user-supplied input or use static queries with binding variables instead.
- HTML markup in the custom error message should be escaped. Unescaped strings displayed in the Salesforce interface can represent a vulnerability in the system.
- Do not hard-code keys when using the Crypto classes. Store them in protected custom settings. For e.g.
 - `Blob hardCodedKey = Blob.valueOf('0000000000000000');`
- VisualForce will automatically enforce CRUD and FLS when direct references to SObjects and SObject field objects or field values are referenced as generic data types or data is copied to other elements, make sure you implement the appropriate access control checks (Use `Describe` methods).
- Run the Salesforce Security Source Scanner to test your Org for a number of security and code quality issues (e.g., Site Scripting, Access Control Issues, Frame Spoofing). It can be accessed at <http://security.force.com/security/tools/forcecom/scanner>
- Leverage security scanner by Checkmarx to a complete code review. It can be accessed at <http://lp.checkmarx.com>

- Winter 13 has introduced pass-through attributes. Use these to disable auto-complete features on forms e sensitive information
- Leverage encoding functions like HTMLENCODE, JSENCODE, JSINHTMLENCODE, URLENCODE to av scripting vulnerabilities for user inputs reflected on the page.
- Never try to use visualforce attribute of escape="false" to bypass the capability of escaping the XSS- vulne When used, this tag exposes the page to XSS attacks.
- Never try to directly use user-provided parameters through the VisualForce URL through Javascript.
- Leverage escape characters or encoding functions in java script on visualforce to avoid cross site scripting
- Never read user-provided values from the URL into apex:includescript, style parameters, help parameters, VisualForce does not encode those.
- Use 'escapeHtml4' method to escape the characters in a String using HTML 4.0 entities.
- In addition to the programming guidelines, the following Security best practices should apply to any develo
- Engage the Client's internal Information Security team to assure compliance with their security practices. V use the more restrictive guidelines.
- Implement a software change control system to manage and record changes to the code both in developr production
- Restrict users from generating new code or altering existing code
- Avoid passing user supplied data to any dynamic execution function. If this is required, sanitize the user in it.
- Do not disclose sensitive information in error responses, including system details, session identifiers or ac
- Restrict access to logs to only authorized individuals
- Remove test code or any functionality not intended for production, prior to deployment. Remove comments accessible production code that may reveal external system or other sensitive information
- Review all secondary applications, third party code and libraries to determine business necessity and valid functionality, as these can introduce new vulnerabilities
- Connected Apps and below OAuth Polices attributes should be used judiciously to ensure Salesforce not all calls/requests into the system:
 - Permitted Users
 - IP Relaxation
 - Refresh Token Policy.
- Lightning Locker Service: Lightning Locker is a powerful security architecture for Lightning components. Li enhances security by isolating Lightning components that belong to one namespace from components in a namespace. Refer: https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/security_coc
- Use the Salesforce Burp Suite to evaluate the security of any web application that exists outside of Salesf (http://security.force.com/security/tools/webapp/burpabout)
- Security Health Check: Health Check report should be generated periodically to identify & fix potential vuln org security settings. Refer: https://help.salesforce.com/articleView?id=security_health_check.htm&type=5
- ESAPI (The OWASP Enterprise Security API) is a free, open source, web application security control libra easier for programmers to write lower-risk applications. The ESAPI libraries are designed to make it easier to retrofit security into existing applications or build a solid foundation for new development. Link: <https://github.com/forcedotcom/force-dot-com-esapi>

Named Credentials

Refer Named Credentials under Authentication.

Few key security points from Identity & Access Management point of view:

- Turn on IP restriction for user logins to minimize the risk of unauthorized access in case of compromised a
- Turn on multi-factor authentication for all users to further reduce the risk of unauthorized access.
- Make organization-wide sharing rules as restrictive as possible while allowing normal business functions a hierarchies, sharing rules, permission sets, etc., to extend access beyond the organization-wide sharing ru
- Set a maximum incorrect login attempt to between 3 and 5 times.
- Enable obscured secret answers for password resets.
- Force re-login upon session timeout but enable session time out warning popup.
- Keep the session timeout timeframe as low as possible without annoying your Salesforce user base.
- Disable caching and autocomplete on the login page.
- Expire user passwords within 90 days of creating it.
- Enforce password history so the same password isn't used until at least 5 new passwords have been usec time the given password was used.
- Passwords should not contain the word 'password'.
- If using platform encryption, regularly generate a new tenant secret, which will generate a new encryption
- When destroying encryption keys, make sure all data encrypted with that key is decrypted first.
- Re-encrypt already encrypted data with the latest key if they're using old keys, even if the old key is archiv destroyed.

For more information on security guidelines please refer:

https://developer.salesforce.com/docs/atlas.en-us.secure_coding_guide.meta/secure_coding_guide/secure_codir

Sharing Model Standards

Few important points on sharing model standards using Salesforce OOTB sharing settings:

- Closed public sharing model: Develop a closed public sharing model with sharing rules as necessary. For larger implementations, it is better to develop a closed public sharing model to each object. For each role that needs to access and share data, sharing rules should be developed to accommodate accordingly. This helps ensure that only access that has been directly configured is granted, reducing the number of accidental access rights.
- Principle of Least Privilege: Each component of the application should only be granted access to the profiles and permission sets requiring access. Therefore all field level security should be turned off for all profiles and specifically granted back to the profiles/permission sets requiring the field visibility. This concept should be used throughout the application including apex and Visualforce access and other permissions.
- Field-Level Security Standards: Use field-level security (FLS) to restrict access to fields. There are multiple ways to keep a user from seeing data at the UI level (removing from a page layout, using FLS, restricting access to an object), however the only way to ensure that the user does not have access to the data in a specific field is to use FLS. This will ensure that the data is not reportable or exportable by a restricted user.

- Recommendations for global implementation of project: For global implementation of project, it is recommended not to rely on a public sharing model for development and data visibility. If the application is rolled out to new countries or groups of users, requirements for limitations to sharing of data could arise whether because of business or legal compliance.

Differentiation between 4 types of sharing:

- Public Read Write (★☆☆☆☆)
 - In general not preferred.
 - Should only be used if the object is restricted in some other way (e.g. via Profile object permissions)
- Public Read (★★☆☆☆)
 - In some cases applicable, based on requirement
- Private (★★★★☆)
 - Most used for independent objects
 - Should be used for top level objects like Accounts, Campaigns, Opportunities, Cases. Or in case certain records should be hidden for specific users.
- Controlled by parent (★★★★☆)
 - Preferred solution
 - Controlled by parent (account) allows us to leverage territory sharing
- Rule: Sharing for new objects
 - Avoid public sharing for records
 - Try to leverage “controlled by parent”
 - In other cases keep the object private

7. ACCESSIBILITY

508 Compliance

The Section 508 Standards are part of the Federal Acquisition Regulation (FAR) and address access for people with physical, sensory, or cognitive disabilities. They contain technical criteria specific to various types of technologies and performance-based requirements which focus on functional capabilities of covered products. The purpose of this part is to implement section 508 of the Rehabilitation Act of 1973, as amended (29 U.S.C. 794d). Section 508 also requires that individuals with disabilities, who are members of the public seeking information or services from a Federal agency, have access to and use of information and data that is comparable to that provided to the public who are not individuals with disabilities unless an undue burden would be imposed on the agency.

1. Use [accessible labels](#) for all input fields
2. Use heading tags for all individual pages. At least one heading tag should be present. (h1 tag is mandatory.)
3. Use proper semantic tags like nav, header, button that represents the content.
4. Use ARIA attributes for appropriate tags and alt attribute for img tags.
5. Use a color contrast analyzer to check for the colors used for styling elements

Guidelines

- Always use lightning components in preference to HTML tags. When customizing the components from lightning design, be careful in preserving code that ensures accessibility, such as the aria attributes. If you are using the code snippets from the lightning design system:
- This is a sample Breadcrumb component is supplied by lightning design system. The content highlighted below ARIA “role” and ARIA attribute “aria-label” are provided by within the component syntax. Don’t change the values unless you are sure about the consequences.

```
<nav role="navigation" aria-label="Breadcrumbs">
```

```
<ol class="slds-breadcrumb slds-list--horizontal">
```

```
<li class="slds-breadcrumb__item slds-text-title--caps"><a href="javascript:void(0);">Parent Entity</a></li>
```

```
<li class="slds-breadcrumb__item slds-text-title--caps"><a href="javascript:void(0);">Parent Record  
Name</a></li>
```

```
</ol>
```

```
</nav>
```

- Provide value for "label" attribute on form elements such as input, text area, radio, checkboxes, pick lists, buttons etc. Use tool tip to convey proper message for input. Every input form element should have a corresponding label element. Ensure that label is not empty.
 - UI components
 - `<ui:button label="Find" class="img" />`
 - Lightning Component
 - `<lightning:button variant="brand" label="Download" iconName="utility:download" iconPosition="left" onclick="{! c.handleClick }" />`
 - HTML
 - `<label for="firstName">First Name< label/>`
 - `<input id="firstName" name="fName" value="{!c.firstName}" />`
- Set "alternativeText" or "alt" to a proper description for images, icons, container, buttonIcons etc. The description should indicate what happens when you click the button. Eg. 'Upload file' not what the button looks like. If the icons are only for decorative purposes and need not convey any message ignore this rule.
 - UI Component
 - `<ui:outputCheckbox value="{!c.eligible}" altChecked="You are eligible" altUnchecked="You are not eligible."/>`
 - Lightning Component
 - `<lightning:buttonIcon iconName="utility:close" variant="bare" onclick="{! c.handleClick}" alternativeText="Close window." />`
 - HTML
 - ``
- Use h1, h2 for headings instead of styling regular `<p>` by changing font-size and always order from h1 to h6. Avoid `<p>` tag to display headings.
 - `<h1 class="slds-text-heading--medium slds-truncate" title="My Expenses">My Expenses</h1>`
Instead of
 - `<p class="slds-text-heading--medium slds-truncate">My Expenses</p>`
- `<table>` tag - headers in the table should be represented by `<th>` related to the cells. If no headings are used those can be ignored. Strictly use `<table>` instead of `<div>` tags and styling to display data in tabular format. `<table>` tag must be immediately followed by `<caption>` tag which stands as table header.

Shelly's Daughters

Name	Age	Birthday
Jackie	5	April 5
Beth	8	January 14

`<table summary="The table describing the birth date details of Shelly's daughters" >`

`<caption>Shelly's Daughters</caption>`

`<tr>`

```
<th scope="col">Name</th>
```

```
<th scope="col">Age</th>
```

```
<th scope="col">Birthday</th>
```

```
</tr>
```

```
<tr>
```

```
<th scope="row">Beth</th>
```

```
<td>8</td>
```

```
<td>January 14</td>
```

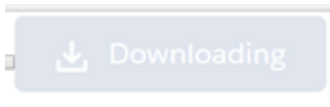
```
</tr>
```

```
</table>
```

- To inform screen readers that a button is disabled set disabled attribute to disabled.

```
<lightning:button variant="brand" label="Downloading" iconName="utility:download" iconPosition="left"
onclick="{!c.handleClick}" disabled="true" />
```

Renders the image as and will be read by screen readers as disabled Downloading button.



- <iframe> should have non-blank text for title. Headings and titles should be included where appropriate on sections, tabs etc.
- Have proper contrast between back ground color and text. The contrast ratio between text and a text's background should be at least 4.5 to 1 for large texts and 3:1 for small texts (below 24px).

Bad:

Click here to proceed.

Good:

Click here to proceed.

- When an operation is taking time appropriate message should be displayed and give appropriate/more time for accessibility users. I.e. During session timeouts.
- Focused component should be highlighted while traversing through key board tab. Example CSS:

```
:focus {
```

```
outline: blue dotted solid;
```

```
}
```

- Color should not be the only criteria to distinguish content. Required fields having labels in red should be avoided instead use asterisk or any visible description. Or if a graph is showing bar charts with colors, the bars should also be distinguished with text.

```
<div class="slds-form-element slds-is-required">
```

```
<div class="slds-form-element__control">
```

```
<ui:inputText class="slds-input" label="Name" aura:id="recordName" value="{!v.record.Name}" required="true"/>
```

```
</div>
```

```
</div>
```

Renders as below. “*” indicates the field is required instead of red color:

Name*

- User aria-describedby to associate form fields (input, checkbox, radio..) beyond labels such as error messages, help texts.
- Use <fieldset> to group elements such as radio button or related fields.
- Use class="assistive-text" if you want the content be hidden but should read by assistive tools.

```
<a href="http://www.facebook.com">
```


```
<span class="icon-facebook">
```

```
<span class="assistive-text">Facebook</span>
```

```
</span>
```

```
</a>
```



- Displays as  but the assistive tools read it as Facebook
- Proper navigation should be provided and facility to skip the repeating the steps say Skip to main content or Go to top.

```
<a href="#maincontent">Skip to main content</a>
```

```
<h1><a name="maincontent" id="maincontent"></a>Heading</h1>
```

- Apply onclick event to elements that are actionable in HTML by default, such as <a>, <button>, or <input> tags.
- User proper aria roles and attributes if you are developing your own components with just html tags.
- Ensure the ID attributes on the components are unique and are referred appropriately by the aria attributes.

Exceptions

The following are some exceptions which are not accessibility compliant in Lightning:

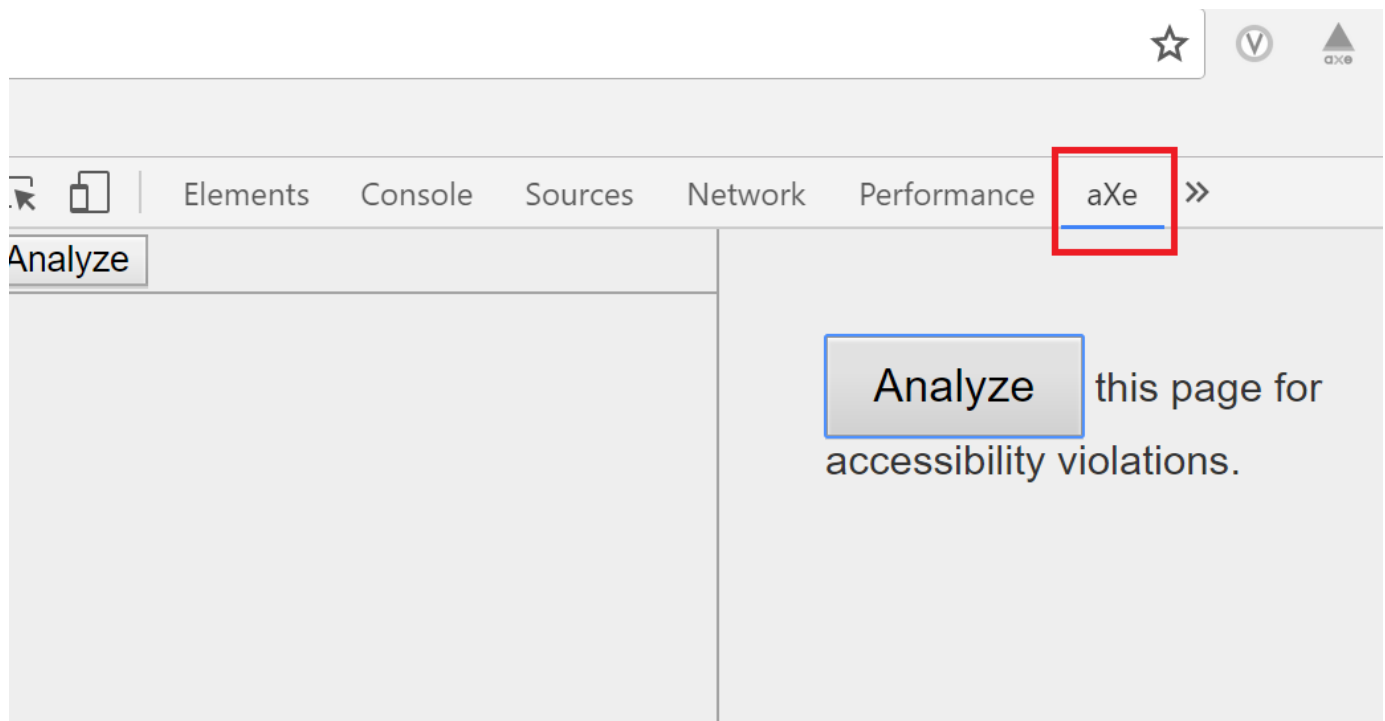
1. Modal window using Iframe to display “Help and Training” videos is not reachable by keyboard alone.
2. Popover information available within “Get More Accounts” modal window and “Kanban” grid display view are not reachable by keyboard.
3. The menu option located in “My Calendars” view list is not reachable using the keyboard alone and/or available to assistive technology.
4. “Help and Training” video player embedded in the modal window is not fully accessible with the use of keyboard and assistive technology. Users cannot select video-selectable controls easily to enable video captions and select other options

Testing

aXe extension for Chrome:

Once installed (<https://chrome.google.com/webstore/detail/axe/lhdoppojpmngadmndnejejpokejbdd>).

Navigate to the page which needs to be checked for accessibility and open Developer Tools (F12) and click on the aXe tab.



Click Analyze, the results will be shown up. Click on each result to know the accessibility violation: The portion of the page responsible along with the code and the reason will be displayed.

Analyze

2 violations found.

< Violation 1 of 1

Critical

RESULTS

Elements must have sufficient color contrast

1

<html> element must have a lang attribute

1

Ensures the contrast between foreground and background colors meets WCAG 2 AA contrast ratio thresholds

wcag2aa

wcag143

Inspect

(More Info)

Target: html > .null > .error.uiMessage > .uiBlock > .bRight > .close

HTML:

```
<a class="close" href="javascript:void(0);" data-aura-rendered-by="52:0">x</a>
```

Summary:

Fix any of the following:

- Element has insufficient color contrast of 1.60 (foreground color: #cabebb, background color: #fdede, font size:

Reference

https://www.salesforce.com/company/legal/508_accessibility.jsp - Please refer to this link to know more about Salesforce Accessibility Status.

<https://www.access-board.gov/guidelines-and-standards/communications-and-it/about-the-section-508-standards>

<https://www.slideshare.net/ShannonHale2/accessible-salesforce>

<http://a11yproject.com/checklist.html>

<https://www.ssa.gov/accessibility/bpl/default.htm>

8. DATA MANAGEMENT

Determine Scope of Data Migration

Every effort should start with determining and documenting the scope of the effort. A high-level object model with objects/tables in scope with an overview of entities to be migrated should be recorded.

Identify if there is a need for data cleansing/conversion beforehand. This analysis is key for estimating data migration efforts.

This effort should include all key stakeholders and SMEs. A detailed document listing the object model, expected data conversion and data manipulation, entities to be migrated signed off by the stakeholders is an expected outcome of this step. In addition to these details, its recommended to consider the following during this effort:

- Sources of data
- End users
- Data relevant for this effort
- Data Transformations needed

Analyze Source and Target Systems

Determine and analyze your source and target systems to help determine the right tool and strategy. This analysis in addition to the scope will help determine the acceptable list of tools.

Profile and audit all source data in the scope before proceeding with mapping specifications. This step should produce a clear data model of both the source and the target system. This may be a challenge for new target implementations, but it is recommended to have a data model before proceeding with the next steps. The outcome of this step will serve as an input for data mapping exercise.

This analysis should also focus on uncovering issues such as licenses, data storage limits, platform limits if any.

It is advised to have a detailed data profile exercise performed at this point to analyze following:

- Required fields in source and target have data in the source system for all rows.
- All validations in the target system are respected in source system and if not, define the transformation rules or address gaps with business stakeholders

- Locale and currencies in scope
- Determine encoding to be used for multi-lingual data
- For ongoing loads, it is important to identify an external ID. If an unique identifier is present in source system it can be used. In case source system has no unique identifier for that entity, then it is important to have an unique identifier to be maintained in both the systems(ex: SFDC ID).

Discrepancies identified in this step can serve as an input to the detailed data mapping exercise.

Analyze Data and Types of Data Loads

For each object in scope, clearly identify the following details:

- One-time vs ongoing feed
- Volume of the data
- Data quality according to rules/object model in target systems

Depending on the data analysis, it's always good practice to standardize, clean, de-dupe (or merge if necessary) and validate source data prior to migration.

Detailed Data Mapping Exercise

Create and follow a data migration workbook throughout the scope of migration. A consolidated workbook that holds the data mapping for each object involved in the process should be maintained. A single template with multiple tabs (one each for each mapping object) including a DM checklist and storage requirements. The workbook can be personalized based on business requirements. Every baseline data mapping document should hold following columns:

- SFDC Field Name
- SFDC API Name
- SFDC Field Type (data type, length etc.)
- Field Values
- Required/Optional
- Data Load needed?
- Load Comments
- Source Field Name
- Source Field Type (data type, length etc.)
- Source Field Values
- Transformation Rule (if any)

As you initiate the process of mapping of the data from one system into another, one of the first things to perform is the user mapping. We need to have a clear understanding of how user ids of existing system map to the new user ids on the Salesforce platform. Carefully evaluate record ownership and analyze business impact of each transformation rule for ownership. As owner field grants full access to a record, data sensitivity should be considered. Defaulting the ownership to a single user can also prove costly if the default owner's role

changes in future (due to sharing recalculation). This is also a good time to revisit which all users have what kind of licenses.

This effort should clearly state the field conversion/data transformation rules if any. It is advised to thoroughly analyze each entity and field to ensure the field types are compatible and supported in the target system. If the field types don't match or are unsupported, a clear conversion rule should be defined that assists the data migration effort. The transformation logic should result in an acceptable format for the target system.

Use External ID's to Import data:

When importing custom objects, solutions, or person accounts, you can use external IDs to prevent the import from creating duplicate records.

An external ID is a custom field that has the External ID attribute, meaning that it contains unique record identifiers from a system outside of Salesforce. When you select this option, the Data Import Wizard detects existing records in Salesforce with external IDs that match those values in the import file.

Using the Data Import Wizard, you can choose from multiple external IDs to match to lookup and master-detail records.

Validating Data Load/Template files:

It is important to prepare load template files for the client to ensure data integrity is maintained. This file should list all fields that are in scope. It is also recommended to:

- Highlight required fields to ensure the stakeholder (or the respective actor) doesn't miss this in load template/mapping
- Highlight field length (of text, number etc.) to match target system
- Custom validation rules should be documented and considered in the mapping
- Restrict data type as per target system (such as email, number, percent, date etc.)
- Allow only picklist values available in the system (for restricted picklists)
- Restrict ID fields (such as owner, record type IDs) to allowed values
- Respect field dependencies and record type – picklist mapping
- Validate lookup fields to ensure they respect lookup filter restrictions
- It is also recommended to consider date and date time fields for global projects. The default locale might not always match the source system. The format of the value and time zone should be mapped appropriately
- If the mapping is for an object type that supports queues, and queues own records, ensure queue IDs (or the corresponding group IDs) are available in target system.

It is also nice to:

- Avoid mapping fields which will be updated by automated processes (such as workflows or triggers)

- Provide an example for each load template file if you're trying to receive inputs from multiple parties. This could particularly prove useful for lookups (to ensure the value mapped to a lookup field is an external ID from the load template that's already loaded)
- Track purpose of fields created specifically for data migration in the data mapping document.

Determine Load Sequence

Determine the order of migration: In Salesforce, relationships that exist between objects and dependencies dictate the order of migration. For example, all accounts have owners, and opportunities are associated with an account. In this case, the order would be to

- Load users
- Load accounts
- Load opportunities

Relationships are expressed through related lists and lookups in a Salesforce application while IDs(foreign key) create relationships in a database.

For larger data models involving self lookups, multiple iterations of the same entity would be needed. This step should focus on uncovering all such relations and detailing the right sequence.

For complex entities such as attachments, it is a good idea to also plan a strategy to optimize the load time by determining a load sequence. For instance, all smaller attachments which add up to the maximum load size of a batch can be grouped together followed by larger attachments.

It is also a good idea to validate the legacy ID mapping while performing this exercise. Mapping legacy system IDs to a field in target system can help with troubleshooting and validating data loads.

Document Pre-requisites

The outcome of this step will help identify any pre-requisites or steps that need to be executed in the target system. It should also include the setup for the data migration user. For ex., for service cloud projects, ensure the data migration user is service cloud and a knowledge user. Following steps are some key considerations before migrating data –

- Create and set up a user with a system administrator profile for data migration.
- Complete system configuration.
- Set up roles/profiles.
- Confirm that record types and picklist values are defined.
- Setup currencies
- Set up every single product/currency combination in the pricebooks if it will be used in Salesforce.
- Enable audit fields if necessary
- Turning off emails if necessary

- While inserting date fields, Salesforce data loader uses the time zone of the user who is performing data load and this might adversely impact the field values post data load.
- Lookup filters must be considered and handled while data load operation.
- Check if you need to wipe out data from certain fields. If you chose Data Loader, you must set the flag “Insert Null Values” box in Settings as true. Otherwise you will not be able to blank out the data.
- For Large Data Volume Loads, consider deferring Sharing Calculation to run once data load has been completed.
- While dealing with PII/PHI data, consider if we need to enable Shield Encryption in Salesforce so that the data is encrypted when inserted/updated.
- While migrating users in sandbox - make sure the email id's are masked to avoid unnecessary trigger of emails.
- If territory account & user association are migrated then we need to enable territory hierarchy.
- Verify if global country & state picklist are used & enabled to capture standardized address format in the org.
- Data Filter Criteria for each entity. Ex: Only Open Opportunities need to be migrated. Opportunities which are beyond certain time frame, shouldn't be considered for migration.
- In some cases it becomes important to create data migration flow as multiple systems might be involved to review and prepare the final load ready file. This helps in conveying the dependencies we have on other systems & highlight the pre-requisites before loading the entity.
- For future delta loads it is important to identify an ExternalID. If we have a unique identifier for entity from the existing data then it should suffice. Else SFDC ID can also be shared & used for future communication if existing entity data has no unique identifier.
- Identify if existing data for a record for an entity is derived from single source or multiple source. For multiple source need to understand the logic to tie the data to a single record.
- Determine the need to have a data archival strategy in place. Data that is not frequently used can be archived after a period of 180 days or later using ETL or SQL tools such as DBAmp

Engage Business People

Business people are the only ones who truly understand the data and who can therefore decide what data can be thrown away and what data to keep.

It is important to have somebody from the business team involved during the mapping exercise, and for future backtracking, it is useful to record mapping decisions and the reasons for them.

Since a picture is worth more than a thousand words, load a test batch into the new system, and let the business team play with it.

Even if data migration mapping is reviewed and approved by the business team, surprises can appear once the data shows up in the new system's UI.

“Oh, now I see, we have to change it a bit,” becomes a common phrase.

Failing to engage subject matter experts, who are usually very busy people, is the most common cause of problems after a new system goes live.

Keep a Track of Target Metadata Changes

It is not uncommon to have a data migration project in progress during an ongoing metadata implementation. In such scenarios, it is mandatory to keep a track of target metadata changes. As having the development team track every change might sound easier, it is also risky to solely rely on community. It's a good practice to periodically review metadata changes using inhouse tools such as WSDLs, Schema Builders, Salesforce Utility etc.

Some metadata changes (such as following) could affect the data mapping and migration effort depending on the volume of the data:

- New required fields
- New validations
- New business logic that manipulates data

It is important track data model changes as well. As it may impact data loaded in previous release.

All this loaded data in previous release must be matched with new data model. The new data model changes might include:

- Introduction of some new fields to get data.
- Few required fields & validations may have been added.
- Transformation of existing data into new model.
- During the transformation the most important field will be the external Id. This will help us map the changes as per new data model.

Other Impacting factors:

- Removal of temporary fields.
- New Duplicate rules.
- Picklist values addition or removal. Restrictd picklist value fields will impact existing and future loading data.

Identify Data Manipulation Rules in Target System

It is crucial to identify all the metadata rules that affect the data in the system. Every DML operation in Force.com can invoke one or more rules listed below:

- Duplicate Rules

- Validation Rules
- Triggers
- Assignment Rules
- Process Builders (or flows)
- Workflow rules
- Auto response and escalation rules

It is advised to leave duplicate and validation rules turned on to maintain data quality as per the target system rules. For other rules it is a good idea to disable/defer what's unnecessary during the migration efforts to improve the performance of data loads. However, there could be scenarios where certain business rules should be allowed to run for rest of the data setup or child entity creation. In such cases, it is recommended to clearly document the rules that should be turned off and on for every entity. It is a good idea to include these details in the same format as data mapping document.

Once the list of business rules are clearly documented, utilize the list to identify a tool that can clearly assist in switching the rules on and off. Its not always practical to manually turn the rules on. In certain cases, some of these rules might not even be editable via Force.com UI (such as triggers in production environments). If the rule count is low and all are accessible easily via Force.com UI, a manual change or a tool such as Salesforce Switch can be utilized. For complex projects where certain rules need to be turned off and the rest should be let on, a bypass logic can be built as part of metadata development and utilized.

If the target data model uses any standard objects that have inbuilt business logic such as fields like revenue on Opportunity etc. or de-duplication logic on Team Member objects, it is a good idea to document those and consider it during validation.

Common Challenges and Workarounds

Support Objects

- Email Messages
 - Is Client Managed field:
 - Use this field to create email messages during data loads in any status (by default, email messages cannot be created with sent status i.e., Status Code = 3)
 - HasAttachment field in Email Messages is not evaluated during data loads as Email Message and Content/Attachment loads happen in multiple iterations. Use Is Client Managed to fix this issue.
 - Another alternative to create email messages in sent status is to enable email drafts in support settings. However, with this approach, you will need two iterations to achieve this. The first load will only allow loads with Status = 5. The second iteration should update status to 3.

- Reply Id field:
 - Reply Id on Email Message holds the ID of the Email Message on which this reply was sent. It can be any number of levels deep depending on the number of replies sent on Email Messages.
 - Considering it's a self-lookup, mapping this field will be difficult as you'll need the (maximum depth + 1) number of iterations to complete this load.
 - This field also doesn't check for integrity constraints and lets you map an ID that's not a valid Email Message ID for the org you're operating in (known issue)
 - As the field doesn't affect any email message behavior, you can either choose to load this with nulls or decide an iteration number that best suits your business needs.

User, Sharing AND Permission Objects

- Loading users is like any other standard object. However, there are few things different from other objects:
 - When creating new users in bulk, remember to turn off the email deliverability if you don't want to notify users.
 - Same applies when you're trying to update the email address of the user to a new email address.
 - If you want to change an email address without notifying the new user, either request salesforce or use "reset password and notify user immediately" option. The latter option only applies for manual updates as its not supported via API
 - Deactivating users may also be a common requirement. However, this shouldn't be an alternate solution to temporarily block users instead. Consider using freeze/unfreeze or login hours option. Deactivation should be avoided as:
 - All manual share records and team membership is lost and must be reinstated if you activate them back
 - Users higher in the hierarchy lose access to the deactivated user's records
 - They're removed from followers and following list (if both users follow each other are deactivated)
 - They're removed from default account and opportunity teams of others
 - Even if they had RW access to accounts via teams, the access defaults to read on reactivation.

Product And Schedule Objects

- Things to note when loading price book data:
 - A standard price book always exists in Salesforce by default
 - If you're creating a new entry in a custom price book, you might see field integrity exception issues if:
 - You're creating an entry for an already existing price book + product + currency combination. Always ensure that this combination (Price book + product + currency) is unique in your org. If you try uploading a duplicate, you'll see "This price definition already exists in this price book" error"

- You're setting "Use Standard Price" to true and setting a unit price value for a custom price book entry. Use standard price internally copies the price from standard price book. Setting price again causes an integrity exception

Choose your Tool Wisely

Salesforce.com facilitates data manipulations in the application through 2 major channels: Salesforce UI (user interface) and API

Salesforce.com UI – Standard Import wizard, Mass Updates, Mass Deletes, Mass Transfer Records and Replace Picklist Values.

1. Standard Import Wizard

1. Salesforce allows importing certain standard objects and custom objects using Salesforce UI.

List of objects which can be imported are:

1. Contacts and Business Accounts
2. Person Accounts
3. Leads
4. Solutions
5. Custom Objects

2. It is recommended to use Standard Import Wizard for smaller data sets with count less than 50,000
3. This method has an advantage of validating the data for duplicates according to account name and site, contact email address, or lead email address.
4. For more details on how to effectively use this wizard, search for "Import Overview" on Salesforce Help and Training site.
5. While using Data Import Wizard pay attention to the Mailing Country & Mailing City when you have state & country picklist enabled. In case user does not have "Modify All" permission, Data Import has a known issues around importing these fields. They are either imported as blank or incorrectly imported in different field.

2. Mass Updates: There is no explicit method of mass updating the data from UI but there are few techniques which can be used to update certain data:

1. Mass Update Address: This will help in standardizing the Country and State data throughout the address fields in Salesforce. To do this, user should have "Modify All Data" permission. Go to: Your Name | Setup | Data Management | Mass Update Addresses. To learn more about mass updating address, search for "Mass Updating Addresses" on Salesforce Help and Training site.
2. Mass Transfer Records: A system administrator with "Modify All Data" permission can transfer ownership of records from one user to another user. To mass transfer records, go to: Your Name | Setup | Data Management | Mass Transfer Records. To learn more about how to mass transfer the records, search "Mass Transferring Records" on Salesforce Help and Training site.

3. Replace Picklist Values: A system administrator with "Customize Application" permission can replace picklist values with more relevant alternatives as business needs evolve. To replace global picklist value, go to: Your Name | Setup | Create | Picklist Value Sets. To replace picklist value on an object, go to Your Name | Setup | Create | Objects. To learn more about replacing picklist values, search for "Replace Picklist Values" on Salesforce Help and Training site.

3. Mass Deletes

1. To delete records from Salesforce UI, user must have "Modify All Data" permission. Go to: Your Name | Setup | Data Management | Mass Delete Records
2. This method of deleting gives an advantage of choosing to delete selected records like deleting related Closed/Open opportunities when deleting accounts.
3. For more details on how to use mass delete, searching "Deleting Mass Data" on Salesforce Help and Training site.

Force.com API – Various Force.com API based tools can be used for loading data into Salesforce. Examples of such tools are detailed in the table below. The API based tools give enough flexibility to data administrators to massage large sets of data after it is extracted from disparate source systems and load into Salesforce either manually or through scheduled automated loads. Most API based applications incorporate Bulk API technique.



1. Force.com Data Loader

1. Data Loader is an application for bulk import or export of data. It can be used to insert, update, delete and export records from all the Salesforce.com objects. Data Loader users CSV format to import and export the data.
2. Use Data Loader when you want to:
 1. Load larger sets of data: from 50,000 records to 5,000,000 records.

2. Manually load data from csv files or automate/schedule data loads from different data sources.
3. Schedule data backups and store data from Salesforce into a fixed location on data servers.
4. Load data and the transformation/conversion logic of source data is not complex.
3. Note: Use MS Access and Excel (Formulas and vlookups) to manipulate and clean the data and prepare the final loadable csv files in right format.
4. While using Data Loader through an VPN, make sure you have the port detail setting present in the Data loader. This is an issue which throws an soap connectivity error while trying to connect via data loader.
2. Partner ETL Tools
 1. ETL tools with Salesforce connectors are most widely used for extracting, transforming and loading the data from various data sources into Salesforce.com application. Examples of partner ETL tools: Informatica on demand, Pervasive, Scribe, CRM Fusion etc.
 2. It is always recommended to use ETL tools when:
 1. There is a need to clean large sets of data.
 2. The data extraction and transformation logic is complex and has to be done on regular basis.
 3. There is a need to do the batch loads and schedule timely loads (Daily, Hourly, Real Time) into Salesforce.
3. Custom Java/.Net based applications
 1. Custom Java/.Net based applications can be developed to transform and load data from source systems to Salesforce.com. These applications use Salesforce APIs to perform the data migration. The advantage of this method is it has the ability to write complex transformation logic using programming languages like Java, C#, etc.
4. Excel Connector
 1. The Excel Connector allows you to import records from Excel directly into Salesforce CRM or Force.com Objects. This is a tool that should be leveraged for small scale data operations where you're dealing with low data volumes and easy data manipulation.
 2. [Click here](#) to read more about this tool.

Import Notes: https://help.salesforce.com/articleView?id=000230867&language=en_US&type=1

Import Email Templates: <https://help.salesforce.com/articleView?id=000231318&type=1>

Determine Recovery Mechanisms

Data Backup is one of the most important steps as part of any data migration projects and Data Governance processes for a seamless BAU. With regular data backups, the risk of losing critical data decreases and helps a client in restoring the data base to a previous state. It is always advised to take complete backup of data before any DML operations or field deletion is performed. There are various tools available through which data backups can be taken one time or scheduled for periodic exports but

most common and best way of doing this is through the “Data Export” interface available from any Salesforce.com application by going to:

Your Name | Setup | Data Management | Data Export and Export Now or Schedule Export.

The best practice is to schedule a bi-weekly export for critical business data (objects) and monthly export for complete organization data.

Alternatively, data backups can also be taken through various Force.com API based tools like Data Loader (batch scheduler or manual export) and various ETL tools.

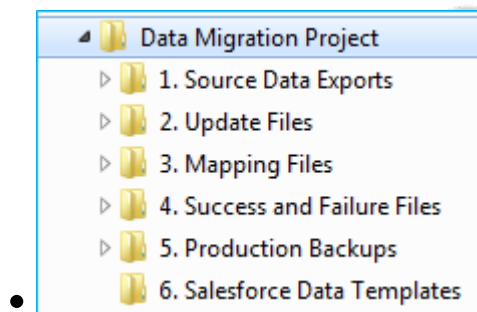
Record all Details

It is a good practice to document the results and the execution times for every entity during the data migration process in lower environments. This will assist in cutover planning and also performance analysis.

In addition to execution times document the success, error files and resolutions for each error. If necessary, make appropriate changes to the mapping documents.

It is advised to maintain a folder structure to store the Mapping Files, Success Files and Error Files. An example of a typical folder structure is depicted below:

Recommended Folder Structure:



Analyze Performance

This step can be skipped for smaller efforts but can be a nightmare for complex efforts that have a weekend release with multiple data loads or for ongoing feeds that take too long to process.

It is always a good practice to use Bulk API for better throughput, especially when working with large data volumes to increase load speed.

While loading large data volumes, the calculations can take a considerable amount of time. We can probably increase load performance by deferring the sharing calculations until after the load is complete.

Consider turning off business rules where they're unused. Utilize the execution times recorded to determine performance for each run and analyze the root cause.

Index fields as necessary and revisit the load sequencing if needed. In certain scenarios, due to multiple relations within data (such as self lookups and roll ups), the order in which data is inserted can help fix performance issues.

While loading large data volumes, the calculations can take a considerable amount of time. We can probably increase load performance by deferring the sharing calculations until after the load is complete

Determine Ways to Validate your Data

Source to data mapping is almost never 1:1. So, while performing a mapping exercise, its also a good idea to come up with metrics and criteria to help validate data in your target system.

Data validation should not be limited to count of records and data storage size but should also include data quality checks. Aim to volume-test all data in the scope as early as possible at the unit level.

Utilize tools like custom reports, help from developers to run queries using developer console or workbench to check data quality. You can also use SF Utility to compare field usage % with source data and determine the accuracy of data loads.

It is also important to do sanity testing directly in Salesforce. Login as different types of users and view a few records of different objects. Compare these same records manually with the original system. Although many tools are available, there is no replacement to manual verification when testing data migration.

Communicate Data Load Details to End Users

It is important to plan out the data migration. The users should be informed well in advance about the cut-off date, and possible issues that may happen. Also, it is always a good idea to have a few pilot users validate the migrated data before end users start using it.

It is also important to communicate dates when the data is expected.

- Dates to get data for QA & Unit testing.
- Dates to get data for UAT.
- Dates to get data for production. Sometimes few entities needed for production might be provided earlier as they may not have much changes(ex: account, users).

Data Loader

Troubleshooting Data Loader:

Data management standards doesn't address the approach to investigate a problem with Data Loader.

If you need to investigate a problem with Data Loader, or if requested by Salesforce Customer Support, you can access log files that track the operations and network connections made by Data Loader.

The log file, `sdl.log`, contains a detailed chronological list of Data Loader log entries. Log entries marked “INFO” are procedural items, such as logging in to and out of Salesforce. Log entries marked “ERROR” are problems such as a submitted record missing a required field. The log file can be opened with commonly available text editor programs, such as Microsoft Notepad.

If you are using Data Loader for Windows, view the log file by entering `%TEMP%\sdl.log` in either the Run dialog or the Windows Explorer address bar.

Configure the Data Loader Log File:

You can customize the Data Loader log file for advanced troubleshooting and tracking.

The `log-conf.xml` file is included with the Data Loader installer version 35.0.

In Windows, the `log-conf.xml` file is at `C:\Users\{userName}\dataloader\version\configs`

In macOS, the `log-conf.xml` file is at `/Users/{userName}/dataloader/version/configs`

Upload Attachments:

You can use Data Loader to upload attachments to Salesforce.

Before uploading attachments, note the following:

If you intend to upload with Bulk API, verify that Upload Bulk API Batch as Zip File on the Settings | Settings page is enabled.

If you are migrating attachments from a source Salesforce org to a target org, begin by requesting a data export for the source org. On the Schedule Export page, select Include Attachments to include the Attachment.csv file in your export. You can use this CSV file to upload the attachments. For more information on the export service, see Export Backup Data from Salesforce.