Salesforce Interview Preparation Guide!

OOPs Concept

Object-Oriented Programming (OOP) is a programming paradigm centered around the concepts of objects and classes. Apex, the programming language used in Salesforce, supports OOP principles, which allow developers to create modular, reusable, and maintainable code. The core OOP concepts in Apex are:

1. Classes and Objects

Class: A blueprint or template for creating objects. It defines the properties (attributes) and methods (functions) that the objects created from the class will have.
Object: An instance of a class. It represents a specific entity with defined attributes and behaviors.

Abstract Classes

An abstract class in Apex is a class that cannot be instantiated directly. It is intended to be a base class for other classes. Abstract classes can include abstract methods, which are methods without an implementation that must be implemented by any concrete subclass. They can also include regular methods with implementations.
Characteristics of Abstract Classes:

Cannot be instantiated directly.
Can include both abstract methods and regular methods.
Subclasses must implement all abstract methods.

Example:

```
 public abstract class Shape {

    public abstract Double area(); // Abstract method

     public void display() { // Regular method
         System.debug('This is a shape.');
    }
}

public class Circle extends Shape {
    public Double radius;
```

Virtual Classes

In Apex, classes are virtual by default, meaning they can be extended unless marked as Avirtual class allows its methods to be overridden in subclasses. This feature enables polymorphism. **Characteristics of Virtual Classes:**

```
     public Circle(Double radius) {
        this.radius = radius;
    }

    public override Double area() {
        return Math.PI * radius * radius;
    }
}

public class Square extends Shape {
    public Double side;

    public Square(Double side) {
        this.side = side;
    }

    public override Double area() {
        return side * side;
    }
 }
```

`final` .

All Apex classes are virtual by default.
Can be extended by other classes.
Methods can be overridden in subclasses.

Example:

```
 public class Employee {
    public virtual String getRole() {
        return 'Employee';
    }
```

```
 }

public class Manager extends Employee {
    public override String getRole() {
        return 'Manager';
    }
}

public class Engineer extends Employee {
    public override String getRole() {
        return 'Engineer';
    }
 }
```

Interfaces

An interface in Apex defines a contract that implementing classes must adhere to. Interfaces can only contain method signatures without implementations. Classes that implement an interface must provide implementations for all the methods defined in the interface.

Characteristics of Interfaces:

- Cannot contain any implementation, only method signatures.
- Aclass can implement multiple interfaces.
- Provides a way to achieve multiple inheritance.

Example:

```
 public interface Drivable {
    void start();
    void stop();
}

public class Car implements Drivable {
    public void start() {
        System.debug('Car is starting');
    }

    public void stop() {
        System.debug('Car is stopping');
```

```
    }
}

public class Motorcycle implements Drivable {
    public void start() {
        System.debug('Motorcycle is starting');
    }

    public void stop() {
        System.debug('Motorcycle is stopping');
    }
 }
```

Differences between Abstract Classes, Virtual Classes, and Interfaces

Feature	Abstract Class	Virtual Class	Interface
Instantiation	Cannot be instantiated directly	Can be instantiated	Cannot be instantiated directly
Method Implementation	Can have both abstract and regular methods	Can have only regular methods	Cannot have method implementations
Inheritance	Can be extended by one subclass	Can be extended by subclasses	Can be implemented by multiple classes
Multiple Inheritance	~~Not supported~~	Not applicable	Supported (class can implement multiple interfaces)
Use Case	When you need to define a base class with common functionality and enforce a contract for subclasses	When you need to define a class that can be extended and have methods overridden	When you need to define a contract that multiple classes should adhere to without providing any implementation

Summary

Abstract Classes are useful when you have a base class with some common implementation and some methods that must be implemented by subclasses.

Virtual Classes (the default in Apex) allow methods to be overridden in subclasses, enabling polymorphic behavior.
Interfaces are useful when you want to define a contract for multiple classes to implement, supporting multiple inheritance.

2. Encapsulation

Encapsulation is the concept of restricting direct access to some of an object's components, which can prevent the accidental modification of data.

3. Inheritance

Inheritance allows a class to inherit properties and methods from another class. This promotes code reuse and establishes a natural hierarchy between classes.

4. Polymorphism

Polymorphism allows methods to do different things based on the object it is acting upon, even if they share the same name. This can be achieved through method overloading and method overriding.

Method Overloading: Multiple methods with the same name but different parameters.
Method Overriding: A subclass provides a specific implementation of a method that is already defined in its superclass.

5. Abstraction

Abstraction is the concept of hiding the complex implementation details and showing only the necessary features of an object. In Apex, this can be achieved through abstract classes and interfaces.

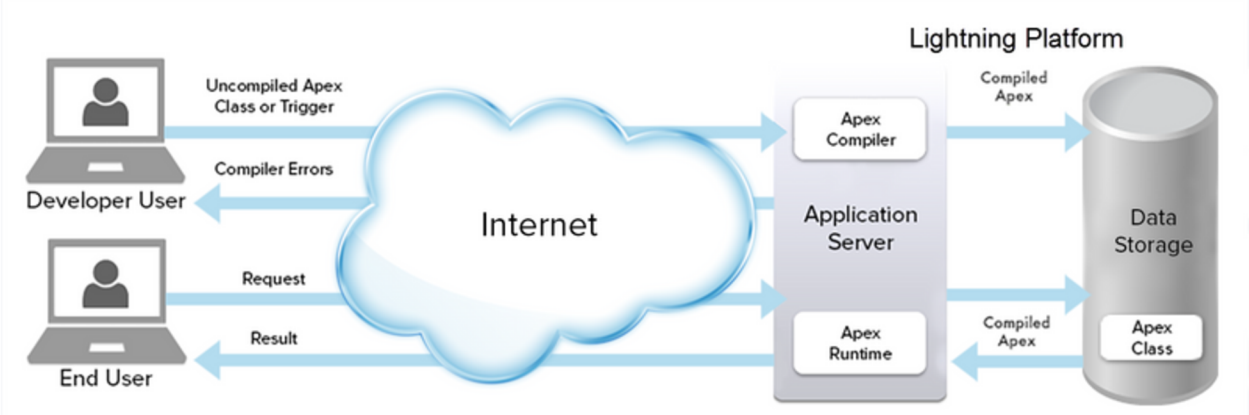
Abstract Class: A class that cannot be instantiated and often includes abstract methods that must be implemented by subclasses.

Interface: Defines a contract that other classes must implement.

Apex Programming [Apex Devloper Guide]

How Does Apex Work?

All Apex runs entirely on-demand on the Lightning Platform. Developers write and save Apex code to the platform, and end users trigger the execution of the Apex code via the user interface.



When a developer writes and saves Apex code to the platform, the platform application server first compiles the code into an abstract set of instructions that can be understood by the Apex runtime interpreter, and then saves those instructions as metadata.

When an end user triggers the execution of Apex, perhaps by clicking a button or accessing a Visualforce page, the platform application server retrieves the compiled instructions from the metadata and sends them through the runtime interpreter before returning the result. The end user observes no differences in execution time from standard platform requests.

Synchronous Execution

Synchronous execution means that the code runs immediately and waits for a response before moving on to the next line of code. It’s a linear and predictable way of executing code, where each operation is completed before the next one starts.

In Apex, synchronous operations are typically used for immediate tasks that need to be completed without delay. For example, when you save a record or update a field, the operation is synchronous, meaning it happens right away and the user or process waits for the operation to finish before proceeding.

Types of synchronous operations in Apex:

1. DML Operations

DML (Data Manipulation Language) operations are used to insert, update, delete,upsert and undelete records in Salesforce. These operations are synchronous, and the system waits for the DML statement to complete before proceeding to the next line of code.

2. SOQL and SOSL Queries

SOQL (Salesforce Object Query Language) and SOSL (Salesforce Object Search Language) are used to retrieve data from Salesforce. These queries are synchronous and return results immediately.

SOQL (Salesforce Object Query Language)

SOQL is used to construct simple and powerful query strings in Apex or in the Salesforce REST API. It is similar to SQL but is specifically designed for querying Salesforce objects.

// Query to retrieve Id and Name from Account object

List<Account> accounts = [SELECT Id, Name FROM Account];

-**Filtering with WHERE Clause:**

List<Account> accounts = [SELECT Id, Name FROM Account WHERE Name = 'Acme'];

-**Using LIKE with Wildcards:**

List<Account> accounts = [SELECT Id, Name FROM Account WHERE Name LIKE 'Ac%'];

-**Ordering Results:**

List<Account> accounts = [SELECT Id, Name FROM Account ORDER BY Name];

-**Limiting Results:**

List<Account> accounts = [SELECT Id, Name FROM Account LIMIT 10];

-**Aggregate Functions:**

Integer count = [SELECT COUNT() FROM Account];

-**Relationship Queries (Parent-to-Child and Child-to-Parent):**

List<Contact> contacts = [SELECT Id, Name, Account.Name FROM Contact];

// Parent-to-Child relationship query

List<Account> accounts = [SELECT Id, Name, (SELECT Id, Name FROM Contacts) FROM Account];

SOSL (Salesforce Object Search Language)

SOSL is used to perform text searches across multiple objects. It is more flexible than SOQL when searching for specific text strings.

Basic SOSL Syntax

List<List<SObject>> searchResults = [FIND 'Acme' IN ALL FIELDS RETURNING Account(Id, Name), Contact(Id, Name)];

Advanced SOSL Queries

-**Searching Across Multiple Objects:**

List<List<SObject>> searchResults = [FIND 'Joe' IN ALL FIELDS RETURNING Account(Id, Name), Contact(Id, Name)];

-**Specifying Fields to Search:**

List<List<SObject>> searchResults = [FIND 'Acme' IN Name FIELDS RETURNING Account(Id, Name)];

-**Using Wildcards in SOSL:**

List<List<SObject>> searchResults = [FIND '*me*' IN ALL FIELDS RETURNING Account(Id, Name)];

Combining SOQL and SOSL in Apex

You can use SOQL and SOSL within the same Apex class to leverage their respective strengths. For example, use SOQL for precise queries and SOSL for full-text searches.

```
public class QueryExamples {

    public void runQueries() {

        // SOQL query example

        List<Account> soqlAccounts = [SELECT Id, Name FROM Account WHERE Name LIKE 'Ac%'];

        for (Account acc : soqlAccounts) {

            System.debug('SOQL Account: ' + acc.Name);

        }



        // SOSL query example

        List<List<SObject>> soslResults = [FIND 'Acme' IN ALL FIELDS RETURNING Account(Id, Name), Contact(Id, Name)];
```

```
        List<Account> soslAccounts = (List<Account>)soslResults[0];
        for (Account acc : soslAccounts) {
            System.debug('SOSL Account: ' + acc.Name);
        }
    }
}
```

Key Differences Between SOQL and SOSL

SOQL	SOSL
You know in which object the data you are searching for resides	You are not sure in which object the data might be.
The data needs to be retrieved from a single object or related objects.	You want to retrieve multiple objects and fields, which might not be related to each other.
SOQL queries can be used in Classes and Triggers.	They are only supported in Apex Classes and Anonymous blocks.
We can perform DML operations on query results.	We can not perform DML operations.

It returns records.	It returns fields.
You can count retrieved records.	You cannot count retrieved records.

3. Triggers

Apex triggers are executed synchronously. When a record is saved, updated, or deleted, the trigger runs immediately before or after the operation completes, depending on the trigger's timing. ()

Types of Triggers

-Before Triggers:**** Used to update or validate record values before they are saved to the database.

-After Triggers:**** Used to access field values set by the system (such as a record's Id or LastModifiedDate field) and to make changes in other records. The records that fire the after trigger are read-only.

Trigger Events

The `trigger_events` can include:

-`before insert`

-`before update`

-`before delete`

-`after insert`

-`after update`

-`after delete`

-`after undelete`

Trigger Context Variables

Apex provides several context variables that contain information about the trigger execution:

-**Trigger.operationType** : Returns AFTER_INSERT,AFTER_DELETE all context we are using Switch method for identify.Example -

```
 trigger FinancialAccountFeeTrigger on FinancialAccountFee ( after insert ,
after update , after delete , after undelete ) {

     FinancialAccountFeeHandler . handleTrigger ( Trigger . new , Trigger . old ,
Trigger . oldMap , Trigger . operationType );

 }

 public with sharing class FinancialAccountFeeHandler {

   public static void handleTrigger ( List < FinancialAccountFee > newFees ,
List < FinancialAccountFee > oldFees , Map < Id , FinancialAccountFee > oldFeeMap ,
System . TriggerOperation triggerEvent ) {

         Map < Id , Set < String >> caseIdToFeeTypes = new Map < Id , Set < String >>(

         switch on triggerEvent {

             when AFTER_INSERT , AFTER_UNDELETE {

                 processFees ( newFees , caseIdToFeeTypes );

             }

             when AFTER_UPDATE {

                 processFees ( newFees , caseIdToFeeTypes );

                 processFees ( oldFees , caseIdToFeeTypes );

             }
```

```
            when AFTER_DELETE {

                processFees ( oldFees , caseIdToFeeTypes );

            }

        }

    }
```

-**`Trigger.isExecuting`:** Returns true if the current context for the Apex code is a trigger.

-**`Trigger.isInsert`:** Returns true if the trigger was fired due to an insert operation.

-**`Trigger.isUpdate`:** Returns true if the trigger was fired due to an update operation.

-**`Trigger.isDelete`:** Returns true if the trigger was fired due to a delete operation.

-**`Trigger.isBefore`:** Returns true if the trigger was fired before any record was saved.

-**`Trigger.isAfter`:** Returns true if the trigger was fired after all records were saved.

-**`Trigger.new`:** Returns a list of the new versions of the sObject records.

-**`Trigger.old`:** Returns a list of the old versions of the sObject records.

-**`Trigger.newMap`** : A map of IDs to the new versions of the sObject records.

-**`Trigger.oldMap`:** Amap of IDs to the old versions of the sObject records.

-**`Trigger.size`:** The total number of records in a trigger invocation.

Best Practices for Writing Triggers

1. **Use Trigger Frameworks:** Implement trigger frameworks to organize your trigger logic and avoid redundancy. Common frameworks include the Trigger Handler pattern.

2. **Avoid SOQL and DML Statements Inside Loops:** This prevents hitting governor limits. Instead, perform queries and DML operations outside loops.

3. **Bulkify Your Code:** Ensure your triggers can handle multiple records. Salesforce processes triggers in batches, so your code should be able to process multiple records efficiently.

4. **Use Context-Specific Handler Methods:** Separate your trigger logic into methods specific to each trigger event to improve readability and maintainability.

5. **Test Your Triggers Thoroughly:** Write comprehensive test classes to ensure your triggers behave as expected under various scenarios.

TRIGGER CONTEXT *VARIABLES MATRIX* *(TRIGGER.)*	ISINSERT		ISUPDATE		ISDELETE		ISUNDELETE
	isBefore	isAfter	isBeforebujhansi	isAfter	isBefore	isAfter	isAfter
new	true	true	true	true	false	false	true
old	false	false	true	true	true	true	false
newMap	false	true	true	true	false	false	true
oldMap	false	false	true	true	true	true	false

4. Synchronous Web Service Callouts

In some scenarios, synchronous web service callouts are necessary, especially when the response is required before proceeding.

```
ublic class WebServiceCallout {

    public String makeCallout() {

        HttpRequest req = new HttpRequest();

        req.setEndpoint('https://api.example.com/data');

        req.setMethod('GET');

        Http http = new Http();

        HttpResponse res = http.send(req);

        return res.getBody();

    }

}
```

Asynchronous Execution

Asynchronous execution means that the code runs independently of the main program flow. The system can start a process and then move on to other tasks without waiting for the process to complete. This is useful for operations that might take a long time to execute, such as complex calculations, large data processing, or integrations with external systems.

1. Future Methods

Future methods in Apex are designed for asynchronous execution, allowing you to run processes in the background. This is particularly useful for operations that may take a

significant amount of time to complete, such as web service callouts or large-scale data processing, and where you don't need the user to wait for these operations to complete.

Key Features of Future Methods

1. **Asynchronous Execution:** Future methods run asynchronously, meaning the current transaction is not blocked while the future method executes.
2. **Limitations on Salesforce Governor Limits:** Future methods can help avoid hitting certain governor limits, especially for operations that exceed the time limits for synchronous transactions.
3. **Callout Support:** They support callouts to external services, which is one of their primary use cases.

Syntax and Example

```
public class MyFutureClass {

@future

 public static void myFutureMethod(Id recordId) {

 // Future method logic here

 }

}
```

Considerations and Limitations

1. **Static Methods Only:** Future methods must be static and can only return void.
2. **Primitive Data Types Only:** The parameters of future methods must be primitive data types, arrays of primitive data types, or collections of primitive data types.
3. **No Synchronous Access:** Future methods cannot be called from within another future method, batch Apex, or a scheduled Apex method.
4. **Limit on Future Method Calls:** There is a limit on the number of future method invocations per 24-hour period (currently 250,000).
5. **Order of Execution:** The execution order of future methods is not guaranteed.

6. **Governor Limits:** Future methods are subject to governor limits, but they run in a separate context from the calling method, which helps in managing limits more effectively.

Use Cases

1. **Web Service Callouts:** One of the most common uses of future methods is to make callouts to external web services. Callouts can be time-consuming, and running them asynchronously helps keep the main thread responsive.
2. **Long-Running Processes:** Any operation that might take along time and can be performed without immediate feedback to the user is a good candidate for a future method.
3. **Processing Large Data Sets:** Future methods can be used to process large data sets asynchronously, especially when this processing would exceed synchronous execution limits.

Example: Making an HTTP Callout Using a Future Method

Here's an example of a future method that performs an HTTP callout to an external service:

```
public class HttpCalloutClass {

 @future(callout=true)

 public static void makeHttpCallout(String endpoint) {

 HttpRequest req = new HttpRequest();

 req.setEndpoint(endpoint);

 req.setMethod('GET');



 Http http = new Http();
```

```
HttpResponse res = http.send(req);



// Process the response

if (res.getStatusCode() == 200) {

System.debug('Callout successful: ' + res.getBody());

} else {

System.debug('Callout failed: ' + res.getStatus());

}

}

}
```

2. Batch Apex

Batch Apex in Salesforce allows you to process large volumes of data by breaking the job into manageable chunks. This is particularly useful for operations that need to handle more records than are allowed in a single transaction or when you need to perform complex processing on a large dataset.

Key Features of Batch Apex

1. **Processing Large Data Sets:** Batch Apex can process up to 50 million records.
2. **Efficient Resource Management:** It breaks the data into smaller batches, processing each batch separately to avoid hitting governor limits.
3. **Asynchronous Processing:** Runs asynchronously in the background, allowing the main execution to continue without waiting for the batch to complete.

4. **Flexible Execution:** You can control the size of each batch and schedule the job to run at specific times.

Batch Apex Structure

ABatch Apex class must implement the `Database.Batchable` interface, which requires three methods:

1. `start` -Defines the scope of the batch job.
2. `execute` -Contains the logic to be executed for each batch of data.
3. `finish` -Executes post-processing operations.

Syntax and Example

Batch Apex Class Example

```
public class AccountBatchUpdate implements Database.Batchable<sObject> {
 // Start method: defines the query to retrieve records
  public Database.QueryLocator start(Database.BatchableContext bc) {
  return Database.getQueryLocator('SELECT Id, Name FROM Account WHERE Name LIKE
\'OldName%\'');
  }



  // Execute method: processes each batch of records
  public void execute(Database.BatchableContext bc, List<Account> scope) {
  for (Account acc : scope) {
  acc.Name = 'NewName' + acc.Id;
```

```
  }

  update scope;

  }



  // Finish method: executes post-processing logic

  public void finish(Database.BatchableContext bc) {

  // Post-processing logic, e.g., sending a notification

  System.debug('Batch job completed.');

  }

}
```

Executing the Batch Apex Job:

```
AccountBatchUpdate batchJob = new AccountBatchUpdate();

Database.executeBatch(batchJob, 200);
```

In the above example, the batch job processes `Account` records in batches of 200.

Considerations and Best Practices

1. **Governor Limits:** Each batch is treated as a separate transaction, which helps in managing governor limits. For example, if a batch fails, only that batch is rolled back.
2. **Batch Size:** The default batch size is 200 records, but you can specify a batch size between 1 and 2,000 records.

3. **Asynchronous Execution Limits:** Batch Apex jobs are queued and executed asynchronously, and there are limits to how many jobs can be running or in the queue.
4. **Chaining Batch Jobs:** You can chain batch jobs by starting anew batch job in the finish method of the current batch job.
5. **State Management:** If you need to maintain state between executions, you can use instance variables, but be aware of their size limits.

Database.AllowCallouts

The Database.AllowCallouts interface is used when you need to perform callouts to external services from within a batch job. By implementing this interface, you can make HTTP requests, SOAP calls, or any other callouts within the execute method.

Database.Stateful

The Database.Stateful interface allows you to maintain state across different batch executions. This is useful when you need to keep track of data or state between different chunks of records processed by the batch job.

Database.BatchableContext

The Database.BatchableContext is an interface that provides methods to get information about the current batch job. It is passed as a parameter to the `execute and`

finish methods. This context allows you to manage the execution of your batch job and interact with the job's lifecycle.

Key Methods

1. **getJobId():** Returns the job ID of the current batch job.
2. **getTriggerId():** Returns the ID of the Apex trigger that fired the batch job, if applicable.
3. **getInstanceId():** Returns the instance ID of the current batch job execution.

Database.QueryLocator

The Database.QueryLocator is used in the start method of a Batch Apex job to define the scope of the records to be processed. It provides a way to handle large data sets by returning a query that Salesforce will use to fetch the records in chunks.

Key Features

1. **Handles Large Data Volumes:** Suitable for queries that return more than 50,000 records.
2. **Efficient Data Fetching:** The records are fetched in chunks as needed, reducing memory consumption and ensuring efficient processing.

Iterable<sObject>

The Iterable<sObject> interface is another way to define the scope of records to be processed in a batch job. Instead of returning a QueryLocator , you can return an

iterable collection of sObjects. This is useful when you need more control over the record retrieval process or when working with custom collections.

Key Features

1. **Custom Collections:** Allows processing of custom collections of sObjects.
2. **Flexibility:** Provides more control over the records being processed compared to asimple SOQL query.

3. Queueable Apex

Queueable Apex in Salesforce is a powerful tool for running asynchronous processes. It offers more flexibility and control compared to future methods, such as the ability to chain jobs and handle complex data types. Queueable Apex is ideal for background processing that requires the advantages of both future methods and batch jobs.

Key Features of Queueable Apex

1. **Chaining Jobs:** You can chain multiple jobs, allowing you to control the sequence of execution.

2. **Enhanced Parameter Support:** Unlike future methods, Queueable Apex can handle complex data types, such as sObjects and collections.
3. **Transactional Control:** Each job runs in its own execution context, similar to batch Apex, providing better governor limit management.
4. **Job Monitoring:** Queueable jobs can be monitored and tracked using the job ID.

Syntax

```
public class MyQueueableClass implements Queueable {

public void execute(QueueableContext context) {

// Your code here

Account acc = [SELECT Id, Name FROM Account WHERE Id = :someId];

acc.Name = 'Updated by Queueable Apex';

update acc;

}

}
```

Execution

```
MyQueueableClass myQueueable = new MyQueueableClass();

System.enqueueJob(myQueueable);
```

The limits for Queueable Apex include:

Up to 50 jobs can be added to the queue in a single transaction.
Amaximum of 250,000 jobs can be queued or running at one time.

Limits are subject to change, and it's best to refer to the latest Salesforce documentation for the most up-to-date information.

Comparison Table

Feature	Queueable Apex	Batch Apex
Purpose	For smaller jobs that need to be processed asynchronously	For large data volumes that require batch processing
Interface	`Queueable`	`Database.Batchable`
MethodSignature	`public void execute(QueueableContext context)`	`public Database.QueryLocator start(Database.BatchableContext bc)  public void execute(Database.BatchableContext bc, List<sObject> scope)  public void finish(Database.BatchableContext bc)`

Chaining Jobs	Yes, supports chaining jobs	No native chaining (requires workaround or separate job)
Complex Data Types	Supports complex types, including sObjects and collections	Limited to simple types and collections of primitives
Callouts	Yes, supports callouts if `callout=true`	Yes, supports callouts if `Database.AllowCallouts` is implemented
Governor Limits	Separate limits for each queued job	Shared limits for each batch execution
Transaction Control	Runs in its own transaction	Runs in a single transaction per batch
Job Size	Suitable for smaller jobs	Suitable for processing large volumes of records

Monitoring	Can be monitored using job ID and `AsyncApexJob object`	Can be monitored using `AsyncApexJob` and batch monitoring tools
Job Status Tracking	Provides job ID for tracking	Provides job ID and detailed batch execution tracking
Job Execution Order	FIFO (First-In, First-Out)	Batches can be executed in parallel or sequentially
Job Limits	50 jobs can be added to the queue per transaction Up to 250,000 queued jobs (including batch jobs)	Up to 5 concurrent batch jobs 5 queued or actively executing batch jobs Maximum batch size: 2000 records
Stateful Jobs	No stateful option	Can maintain state using `Database.Stateful` interface

ApexScheduler	Can be scheduled using Apex Scheduler	Can be scheduled using Apex Scheduler

Use Cases

Chaining jobs together.
Handling more complex asynchronous processing than future methods.

4. Scheduled Apex

Scheduled Apex in Salesforce allows you to run Apex classes at specific times and intervals. This feature is useful for automating repetitive tasks, such as nightly data processing, periodic system maintenance, or other time-based operations. To use Scheduled Apex, you need to implement the Schedulable interface in your Apex class and then schedule the class to run at specific times using the Salesforce user interface or programmatically.

Key Concepts

1. Schedulable Interface
   The Schedulable interface requires the implementation of the execute method, which contains the logic you want to run on a schedule.
2. Cron Expressions
   Acron expression is used to specify the schedule, determining when the job runs. It includes fields for seconds, minutes, hours, day of month, month, day of week, and optional year.

Implementing Scheduled Apex

To implement Scheduled Apex, follow these steps:

1. Create a Class that Implements the `Schedulable Interface`
   Define the logic you want to execute in the execute method.
2. Schedule the Class Using the Salesforce UI or Programmatically
   You can schedule the class using the Salesforce UI or by writing an Apex script to schedule it programmatically.

Example: Scheduled Apex Class

Here's an example of a simple Scheduled Apex class that updates account records:

```
public class ScheduledAccountUpdate implements Schedulable {

  public void execute(SchedulableContext sc) {

  // Your logic here

  List<Account> accounts = [SELECT Id, Name FROM Account WHERE SomeCondition__c =
true];

  for (Account acc : accounts) {

  acc.Name = 'Updated ' + acc.Name;

  }

  update accounts;

  }

 }
```

Scheduling the Apex Class

You can schedule the class using two methods: the Salesforce user interface and Apex code.

Scheduling Using the Salesforce User Interface

1. Navigate to Setup
2. Enter "Apex Classes" in the Quick Find box
3. Click on "Apex Classes"
4. Click "Schedule Apex"
5. Provide the job name and select the class to schedule
6. Specify the schedule using the user-friendly interface

Scheduling Programmatically

You can use the System.schedule method to schedule the job programmatically. This method takes three arguments: the job name, the cron expression, and an instance of the class to schedule.

Here's how to schedule the ScheduledAccountUpdate class programmatically:

```
String jobName = 'Scheduled Account Update Job';

String cronExp = '0 0 23 * * ?'; // This cron expression means every day at 11 PM



ScheduledAccountUpdate job = new ScheduledAccountUpdate();

System.schedule(jobName, cronExp, job);
```

Cron Expression Syntax

The cron expression consists of six mandatory fields and one optional field:

1. Seconds (0-59)
2. Minutes (0-59)
3. Hours (0-23)

4. Day of month (1-31)
5. Month (1-12 or JAN-DEC)
6. Day of week (1-7 or SUN-SAT)
7. Optional year (empty, 1970-2099)

Examples:

Every Day at Midnight: `0 0 0 * * ?`
Every Hour: `0 0 * * * ?`
Every Monday at 8 AM: `0 0 8 ? * MON`

Monitoring Scheduled Jobs

Scheduled jobs can be monitored through the Salesforce UI:

1. Navigate to Setup
2. Enter "Scheduled Jobs" in the Quick Find box
3. Click on "Scheduled Jobs"

This page shows the list of all scheduled jobs and their statuses.

Managing Scheduled Jobs

You can manage scheduled jobs using the System.abortJob method to cancel a scheduled job. This requires the job ID, which you can obtain from the `AsyncApexJob` object or the Scheduled Jobs page.

```
String jobId = 'yourJobId';

System.abortJob(jobId);
```

Best Practices

1. Error Handling

Implement robust error handling within your scheduled class to manage any exceptions and ensure that the job completes successfully.

2. Logging

   Use custom logging to record the execution details and status of your scheduled jobs for monitoring and debugging purposes.

3. Governor Limits

   Be mindful of Salesforce governor limits, and optimize your scheduled Apex code to handle bulk processing efficiently.

4. Test Coverage

   Ensure you have adequate test coverage for your scheduled Apex classes to meet Salesforce's testing requirements and ensure reliable operation.

Comparison of Asynchronous Methods

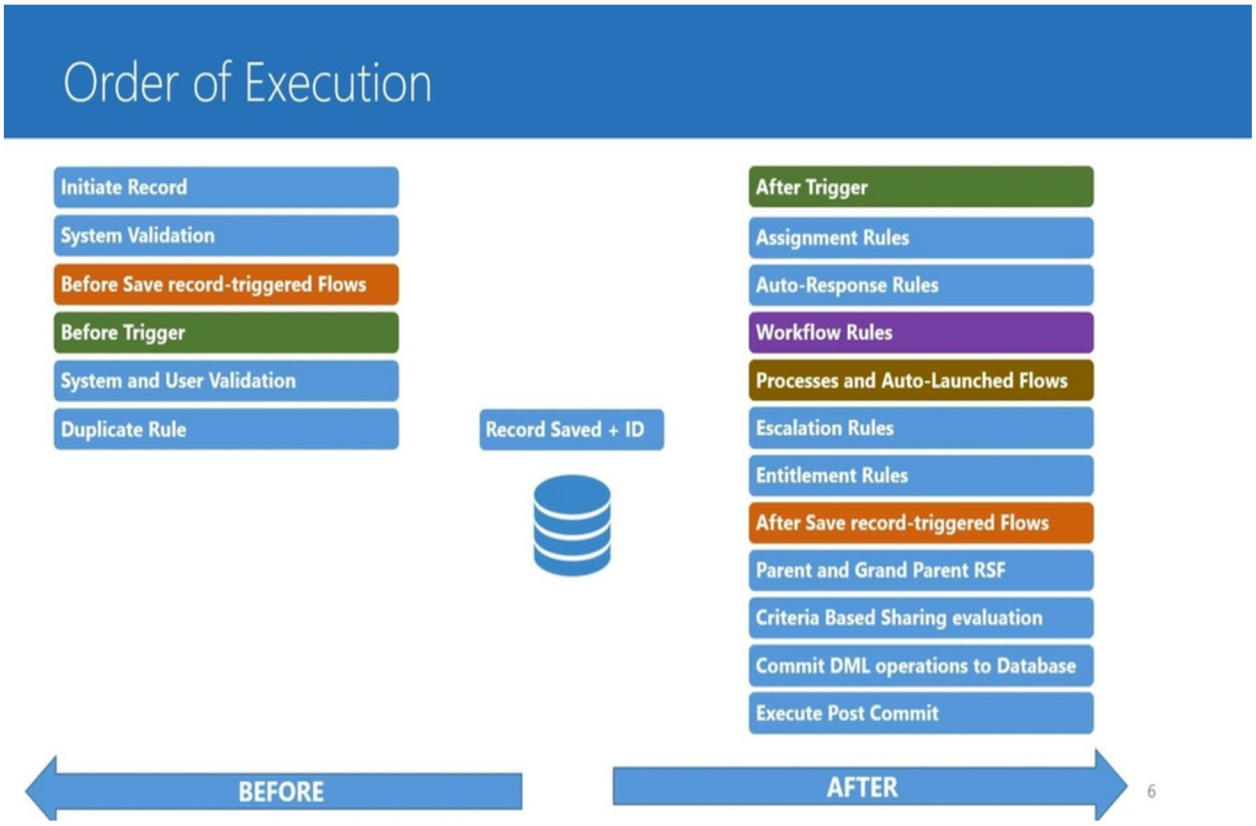
Feature	Future Methods	Batch Apex	Queueable Apex	Scheduled Apex
Use Case	Callouts, simple tasks	Large data sets	Chained jobs, complex tasks	Scheduled tasks, periodic jobs
Chaining	No	No	Yes	No
Governor Limits	Limited	Higher limits	Similar to future	N/A
Order of Execution	Uncertain	Controlled	Controlled	Controlled
Transaction Control	No	Yes	Yes	N/A

Interface	@future annotation	Batchable Interface	Queueable Interface	Schedulable Interface

Order of Execution

1. System Validation Rules - System validation rules are applied to the record. These include checking for required fields and unique field constraints at the database level.
2. Before Triggers - Executes all before triggers, allowing you to update or validate values before they are saved to the database.
3. Custom Validation Rules - Validates the record against custom validation rules.
4. Duplicate Rules - Duplicate rules are executed, which prevent the creation of duplicate records.
5. Save to the Database (Initial Save) - The record is saved to the database, but not committed yet.
6. After Triggers - Executes all after triggers, allowing you to access the record after it has been saved to the database.
7. Assignment Rules - If there are any assignment rules defined (such as for leads or cases), they are processed.
8. Auto-Response Rules - Auto-response rules (such as those for leads and cases) are processed.
9. Workflow Rules - Workflow rules are executed. If a workflow rule updates a field, both before and after triggers are re-executed once (and only once). These field updates do not initiate additional workflow rule evaluations.
10. Processes and Flows (Process Builder) - Executes any processes defined in Process Builder. Like workflows, any field updates here will re-trigger `before and after triggers` once.
11. Escalation Rules - Escalation rules (for cases) are evaluated and processed.
12. Entitlement Rules - Entitlement rules (for cases) are evaluated and processed.
13. Roll-Up Summary Fields and Cross-Object Workflow Updates - If the record update causes changes to roll-up summary fields, parent records are updated, which can trigger additional workflows and triggers.
14. Parent Roll-Up Summary Fields (If Updated) - If a roll-up summary field in a parent record is updated, the system recursively evaluates and updates the parent record.

15. Criteria-Based Sharing Rules - Criteria-based sharing rules are evaluated.
16. Commit - The transaction is committed to the database.
17. Post-Commit Logic - Post-commit logic such as sending email notifications.



Salesforce Governor Limits

Governor Limit	Limit	Exception Message

Total number of SOQL queries issued	100 (synchronous), 200 (asynchronous)	`Too many SOQL queries: 101`
Total number of records retrieved by SOQL queries	50,000	`Too many query rows: 50001`
Total number of SOSL queries issued	20	`Too many SOSL queries: 21`
Total number of records retrieved by a single SOSL query	2,000	`Too many SOSL query rows: 2001`
Total number of DML statements issued	150	`Too many DML statements: 151`

Total number of records processed as aresult of DML statements, Approval.process, or database.emptyRecycleBin	10,000	`Too many DML rows: 10001`
Total number of callouts (HTTP requests or web services calls)	100	`Too many callouts: 101`
Total number of sendEmail methods allowed	10	`Too many Email Invocations: 11`
Total heap size	6MB (synchronous), 12 MB (asynchronous)	`Heap size too large: 6000001`

Maximum CPU time	10,000 ms (synchronous), 60,000 ms (asynchronous)	`Apex CPU time limit exceeded`
Total number of executed code statements	200,000 (synchronous), 1,000,000 (asynchronous)	`Total number of DML statements or Database.executeBatch exceeded`
Total number of Apex classes and triggers in the org	6MB per transaction (for apex class and trigger code), 12 MB per asynchronous transaction (batch apex, queueable, future)	`Apex heap size too large`
Maximum number of Apex classes per org	5,000	-

Maximum number of Apex triggers per org	5,000	-
Maximum number of queued jobs for Batch Apex	5active or queued jobs at a time	-
Maximum number of future method invocations per 24-hour period	250,000 or the number of user licenses in the org multiplied by 200, whichever is greater	-
Single email recipients	5,000 (per email)	SendEmail failed. First exception on row 0; first error: INVALID_ID_FIELD, Set ID is not valid for sending emails

Daily single email message limit	5,000 per day	-
Daily mass email message limit	500 per day	-
Total number of callouts (HTTP requests or web services calls) in a transaction	100	-
Maximum timeout for callouts (HTTP requests or web services calls)	120 seconds	-
Maximum number of Apex jobs added to the queue with System.enqueueJob	50 jobs per transaction	-

Maximum number of batch Apex jobs in the Apex flex queue that are in Holding status	100	-
Maximum number of batch Apex job start method concurrent executions	5	-
Maximum number of push notifications sent in a 24-hour period	1,000 per user or 200,000 per org	-
Maximum number of sendEmail method calls per transaction	10	-

Total number of script statements that can be executed (Visualforce Controllers)	200,000 (synchronous), 1,000,000 (asynchronous)	-
Maximum view state size (Visualforce Controllers)	170 KB	-

Best Practices to Avoid Governor Limit Exceeding

1. **Bulkify Your Code:** Process records in bulk to avoid hitting limits.
2. **Use Collections and Efficient Queries:** Optimize SOQL queries to retrieve only necessary data.
3. **Avoid SOQL and DML in Loops:** Move SOQL and DML operations outside loops to minimize the number of statements executed.
4. **Implement Efficient Error Handling:** Handle errors gracefully to avoid excessive retries or rollbacks.
5. **Use @future, Batch Apex, and Queueable Apex Appropriately:** Handle operations that exceed synchronous limits with asynchronous processing.
6. **Monitor and Log Usage:** Use debug logs and monitoring tools to track the consumption of governor limits.
7. **Optimize Data Model:** Ensure your data model is designed efficiently to reduce the need for complex processing.

Why do we use the governor limit ?

Governor limits are used in Salesforce to ensure the efficient and fair use of shared resources in a multi-tenant environment. Salesforce operates on a multi-tenant architecture where multiple customers share the same instance of the software.

Without governor limits, a single customer could monopolize server resources, causing performance issues for other customers.

Managed and Unmanaged Packages

AppExchange solutions are containers for apps, tabs, and objects. Packages come in two flavors: managed and unmanaged.

Attribute	Managed Packages	Unmanaged Packages
Upgrades	The provider can automatically upgrade the solution.	To receive an upgrade, you must uninstall the package from your org and then reinstall a new version from AppExchange.
Customization	You can't view or change the solution's code or metadata.	You can customize code and metadata, if desired.
Org limits	The contents of the package don't count against the app, tab, and object limits in your org.	The contents of the package count against the app, tab, and object limits in your org.

Salesforce Api

In Salesforce, APIs (Application Programming Interfaces) provide a way for external systems to interact with Salesforce data and functionality programmatically. Salesforce offers various APIs catering to different integration scenarios, including accessing data, executing business logic, and managing the Salesforce environment. Here are some key Salesforce APIs:

REST API

Purpose: Provides a simple and powerful way to interact with Salesforce data using RESTful principles.

Features:

- CRUD operations (Create, Read, Update, Delete) on Salesforce records.
- Querying records using SOQL (Salesforce Object Query Language).
- Access to metadata information.
- Support for composite resources and batch requests.

Use Cases:

- Integrating Salesforce data with web and mobile applications.
- Building custom user interfaces that interact with Salesforce data.
- Performing lightweight operations like querying and updating records.

SOAP API

Purpose: Offers a standards-based web services API for accessing Salesforce data and functionality.

Features:

- Full CRUD operations on Salesforce objects.
- Querying records using SOQL.
- Support for custom objects and fields.
- Access to metadata information.

Use Cases:

- Integrating Salesforce with enterprise systems using SOAP web services.
- Interacting with Salesforce data from applications built on SOAP-compatible platforms.k

Bulk API

Purpose: Enables the processing of large volumes of data asynchronously.

Features:

Bulk CRUD operations for inserting, updating, upserting, deleting, and querying records.
Optimized for handling large data sets efficiently.
Supports batch processing with job monitoring and error handling.

Use Cases:

Importing large data sets into Salesforce.
Performing mass updates or deletions on Salesforce records.
Extracting large data sets from Salesforce for reporting or analytics purposes.

Streaming API

Purpose: Provides a mechanism for receiving near-real-time notifications about changes to Salesforce data.

Features:

Push-based notifications for changes to records that match specified criteria.
Supports subscribing to various types of events, including create, update, delete, and undelete operations.
Delivers events over long-lived connections using CometD protocol.

Use Cases:

Building real-time dashboards and monitoring applications.
Implementing event-driven integrations between Salesforce and external systems.
Enabling collaboration and notifications within Salesforce.

Metadata API

Purpose: Allows developers to retrieve, deploy, create, update, and delete metadata in Salesforce.

Features:

Access to metadata components such as objects, fields, workflows, and customizations.
Supports retrieving metadata in XML format and deploying changes using metadata packages.
Enables automation of development, deployment, and customization tasks.

Use Cases:

Automating deployments between Salesforce environments (sandbox to production).
Building tools for managing Salesforce configurations and customizations.
Integrating with version control systems for source-driven development.

Tooling API

Purpose: Provides programmatic access to developer tools and features within Salesforce.

Features:

Access to metadata, debug logs, Apex code, and Lightning components.
Supports dynamic Apex code execution and debugging.
Enables querying of tooling objects for development insights and automation.

Use Cases:

Building developer tools and IDE integrations for Salesforce.
Analyzing code quality, performance, and usage metrics.
Automating development tasks like code deployments and testing.

Apex REST API

Purpose: Allows developers to expose custom Apex classes as RESTful web services.

Features:

Define custom RESTful endpoints using Apex classes and methods.
Supports HTTP methods like GET, POST, PUT, DELETE.
Enables custom processing and logic for incoming requests.

Use Cases:

Exposing custom business logic and integrations as RESTful APIs.
Building custom integrations with external systems using RESTful endpoints.
Implementing lightweight web services within Salesforce for internal use.

These are some of the key APIs provided by Salesforce to support various integration and development scenarios. Depending on your requirements, you can choose the most suitable API for your integration or development needs.

Reports

Reports and dashboards are powerful tools in Salesforce for visualizing and analyzing data. They enable users to gain insights into their Salesforce data, track key metrics, and make data-driven decisions. Here's an overview of reports and dashboards in Salesforce:

What are Reports?

Reports in Salesforce are customizable views of your data, allowing you to organize, summarize, and analyze information stored in Salesforce.
Reports can be created on standard and custom objects and can include fields from related objects using cross-object relationships.
Salesforce provides a wide range of standard report types, such as Tabular, Summary, Matrix, and Joined reports, which serve different analytical purposes.

Creating Reports:

1. Select Report Type: Choose the appropriate report type based on the data you want to analyze.
2. Define Filters: Apply filters to narrow down the data based on specific criteria.
3. Customize Columns: Select fields to display in the report and customize column formatting.

4. Group and Summarize Data: Group records by certain criteria and add summary fields (e.g., sum, average) for numeric data.
5. Run and Preview: Run the report to see the data preview and make adjustments as needed.
6. Save and Share: Save the report and share it with other users or groups in your organization.

Key Features:

Drill-Down: Users can drill down into report data to view details at a more granular level.
Charts and Graphs: Reports can include various chart types (e.g., bar charts, pie charts) to visualize data trends.
Scheduled Refresh: Reports can be scheduled to automatically refresh at specific intervals to ensure users have access to up-to-date information.

Types of reports

1. Tabular Reports

Description: Tabular reports are the simplest type of report, displaying data in a table format with rows and columns.

Use Case: Best suited for displaying detailed data records in a simple, easy-to-read format.

Key Features:

Displays all selected fields in a row-by-row format.
Limited to displaying up to 2,000 rows of data.

2. Summary Reports

Description: Summary reports group data based on specified criteria and display summary information, such as totals and subtotals.

Use Case: Useful for summarizing data and analyzing trends across different groups or categories.

Key Features:

- Allows grouping of data by one or more fields.
- Includes summary rows and columns showing totals, subtotals, averages, etc.
- Supports sorting and filtering options.

3. Matrix Reports

Description: Matrix reports organize data into a grid format, allowing for both row and column grouping.

Use Case: Ideal for comparing data across two different categories simultaneously.

Key Features:

- Supports both row and column grouping.
- Displays summary information at the intersection of rows and columns.
- Provides a comprehensive view of data relationships and comparisons.

4. Joined Reports

Description: Joined reports combine multiple report blocks into a single report, each block containing its own set of fields, filters, and chart types.

Use Case: Useful for analyzing data from multiple sources or objects in a single report.

Key Features:

- Allows users to view data from different sources side by side.
- Each block within a joined report can have its own unique formatting and chart types.
- Enables comparative analysis across different datasets.

5. Cross-Tab Reports

Description: Cross-tab reports display data in a grid format, with rows representing one field and columns representing another field. Each cell contains summary information.

Use Case: Suitable for analyzing data where you want to compare two fields against each other.

Key Features:

- Provides a cross-tabular view of data, making it easy to compare values across different dimensions.
- Supports aggregation functions like sum, average, count, etc., for each cell.

6. Summary Matrix Reports

Description: Summary matrix reports combine the features of both summary and matrix reports, allowing for both row and column grouping along with summary information.

Use Case: Offers a more detailed and structured view of summarized data compared to standard summary reports.

Key Features:

- Provides both row and column grouping capabilities.
- Displays summary information at the intersection of rows and columns.
- Offers a more granular view of summarized data.

Dashboards

What are Dashboards?

- Dashboards in Salesforce are visual representations of data, consisting of components like charts, graphs, tables, and metrics.
- Dashboards provide a consolidated view of key metrics and performance indicators, allowing users to monitor their business at a glance.
- Users can customize dashboards by arranging and resizing components to fit their needs.

Creating Dashboards:

1. Select Source Reports: Choose the reports that will serve as data sources for the dashboard components.
2. Add Dashboard Components: Add various components (e.g., charts, tables) to the dashboard canvas and configure them to display specific report data.
3. Arrange and Customize: Arrange the components on the dashboard canvas and customize their appearance and settings.
4. Preview and Adjust: Preview the dashboard to see how the components look together and make adjustments as needed.
5. Save and Share: Save the dashboard and share it with users or groups who need access to the insights it provides.

Key Features:

Real-Time Data: Dashboards display real-time data, providing users with the most current insights into their business operations.
Interactive Filters: Users can apply filters to dashboards to dynamically change the displayed data based on specific criteria.
Drill-Down and Drill-Across: Users can drill down into dashboard components to explore detailed information or drill across related reports for deeper analysis.

Use Cases

Sales Performance Monitoring: Track sales pipeline, opportunities, and revenue trends using reports and dashboards.
Service Case Analysis: Analyze service case data to identify trends, agent performance, and customer satisfaction metrics.
Marketing Campaign Effectiveness: Measure the success of marketing campaigns by analyzing lead conversion rates, campaign ROI, and customer engagement metrics.
Executive Dashboards: Provide executives with a high-level view of key business metrics, such as revenue, profitability, and customer satisfaction scores.

Types of dashboard

1. Standard Dashboards

Description: Standard dashboards provide a basic, out-of-the-box solution for visualizing data in Salesforce. They are easy to create and offer essential components for monitoring key metrics.

How to Create:

1. Navigate to Dashboards: Access the Dashboards tab in Salesforce Setup.
2. Create New Dashboard: Click on the "New Dashboard" button.
3. Choose Components: Select the components (charts, tables, metrics) you want to include on the dashboard.
4. Configure Settings: Customize the dashboard title, description, and refresh settings.
5. Save and Share: Save the dashboard and share it with relevant users or groups.

Rationale: Standard dashboards are ideal for quickly creating basic visualizations of key metrics without the need for extensive customization. They offer a straightforward way to monitor essential performance indicators.

2. Dynamic Dashboards

Description: Dynamic dashboards provide a personalized view of data based on the user viewing the dashboard. Each user sees data relevant to their role or permissions.

How to Create:

1. Create Multiple Dashboard Components: Build separate dashboard components tailored to different user roles or criteria.
2. Configure Filters: Apply filters to each component to ensure data is displayed dynamically based on user context.
3. Assign to Users or Roles: Assign the appropriate dashboard component to specific users or roles.
4. Set as Default Dashboard: Optionally, set a dynamic dashboard as the default view for certain users or profiles.

Rationale: Dynamic dashboards offer a personalized experience for users by presenting data relevant to their roles or responsibilities. This ensures that each user sees the most relevant information to support their decision-making process.

3. Custom Dashboards

Description: Custom dashboards provide full flexibility and customization options, allowing users to design dashboards tailored to their specific needs and preferences.

How to Create:

1. Design Layout: Determine the layout and structure of the dashboard, including the placement of components.
2. Choose Components: Select from a wide range of available components, including charts, tables, metrics, and custom visualizations.
3. Customize Appearance: Customize the appearance of each component, including colors, fonts, and labels.
4. Add Interactivity: Implement interactive features such as drill-down capabilities, filters, and dynamic refresh options.
5. Test and Iterate: Test the dashboard with sample data and iterate on the design as needed to ensure optimal usability and effectiveness.

Rationale: Custom dashboards offer complete control over the design and functionality, allowing users to create highly tailored visualizations that meet their unique requirements. They are ideal for advanced analytics and specialized reporting needs.

4. Operational Dashboards

Description: Operational dashboards focus on real-time monitoring of ongoing operations and activities within the organization. They provide immediate insights into critical processes and workflows.

How to Create:

1. Identify Key Metrics: Determine the key performance indicators (KPIs) and operational metrics that need to be monitored in real time.
2. Select Appropriate Components: Choose components that offer real-time data updates, such as live charts, streaming datasets, or dynamic tables.

3. Configure Alerts: Set up alerts and notifications to trigger when certain thresholds or conditions are met.
4. Optimize for Accessibility: Ensure the dashboard is easily accessible to relevant users, such as frontline staff or operations managers, on their preferred devices.

Rationale: Operational dashboards provide immediate visibility into critical processes and workflows, enabling proactive decision-making and rapid response to changing conditions or issues.

5. Executive Dashboards

Description: Executive dashboards offer a high-level view of organizational performance and strategic initiatives. They focus on presenting summarized data and strategic insights to senior leadership.

How to Create:

1. Focus on Strategic Metrics: Highlight key strategic objectives and performance indicators relevant to executive stakeholders.
2. Use Summary Views: Present data in summarized formats such as aggregated charts, trend analysis, and comparative metrics.
3. Include Forecasting and Predictive Analytics: Incorporate predictive analytics and forecasting models to provide insights into future trends and opportunities.
4. Ensure Accessibility and Mobility: Design the dashboard to be easily accessible on various devices, including mobile phones and tablets, for on-the-go decision-making.

Rationale: Executive dashboards provide senior leadership with a concise overview of organizational performance, helping them make strategic decisions and drive business growth effectively.

Data Security Model

Data security model in Salesforce is designed to ensure that data is accessible only to authorized users while protecting it from unauthorized access. It encompasses several layers of security that control access at the organization, object, field, and record levels. Here’s a comprehensive overview of the Salesforce Data Security Model:

Profiles

profiles are used to control access to data and the various features within the Salesforce environment. They are an essential part of the security and data access model, defining what users can do within Salesforce by specifying permissions at various levels. Here's a detailed overview of profiles in Salesforce:

Key Components of Profiles

1. Object Permissions

    Create, Read, Update, Delete (CRUD) Permissions: Determine what actions users can perform on records of a particular object.

        Create: Allows users to create new records.

        Read: Allows users to view records.

        Update: Allows users to edit existing records.

        Delete: Allows users to delete records.

2. Field-Level Security

    Controls visibility and editability of individual fields within an object.

    Ensures sensitive data fields are only accessible to users with appropriate permissions.

3. Tab Settings

    Default On: The tab is visible in the app.

    Default Off: The tab is hidden but can be made visible.

    Hidden: The tab is not accessible to the user.

4. App Settings

    Determines which standard and custom apps are visible to users.

    Controls the default app that appears when a user logs in.

5. Record Types

    Defines which record types are available to users for each object.

Allows customization of page layouts and picklist values based on record types.

6. Page Layouts

   Specifies which page layouts users can access and use for viewing and editing records.

   Controls the organization and visibility of fields, sections, and related lists on a record detail page.

7. User Permissions

   General User Permissions: Basic permissions such as viewing reports, running dashboards, and exporting data.

   Administrative Permissions: Advanced permissions for system administration tasks such as managing users, profiles, and custom objects.

   Custom Permissions: Specific custom permissions that can be created and assigned to provide access to custom features or processes.

8. Login Hours and IP Restrictions

   Login Hours: Define the hours during which users can log in to Salesforce.

   IP Restrictions: Limit the IP addresses from which users can access Salesforce.

Standard Profiles

Salesforce provides several standard profiles out of the box, each with a predefined set of permissions tailored to common roles. Examples include:

System Administrator: Full access to all data and features. Can manage users, customize Salesforce, and perform administrative tasks.

Standard User: Basic permissions to use most Salesforce features. Can read, create, edit, and delete records for standard objects.

Read Only: Can view records but cannot create, edit, or delete them.

Solution Manager: Can manage solutions and related records, with permissions similar to those of the Standard User profile.

Marketing User: Additional permissions related to managing campaigns and marketing activities

Permission sets

Permission sets in Salesforce are a powerful tool for extending users' access without changing their profiles. They provide a flexible way to grant additional permissions to users on an as-needed basis, making them essential for managing access in a dynamic and scalable manner. Here’s an in-depth look at permission sets in Salesforce:

Key Features of Permission Sets

1. Granular Control

   Object Permissions: Grant Create, Read, Update, and Delete (CRUD) permissions for specific objects.

   Field Permissions: Control access to specific fields within objects, including Read and Edit permissions.

   User Permissions: Assign specific user permissions, such as running reports or importing data.

   App Permissions: Enable access to specific applications within Salesforce.

   Apex Class Access: Provide access to specific Apex classes that execute within Salesforce.

   Visualforce Page Access: Allow access to specific Visualforce pages.

2. Flexible Assignment

    Users can be assigned multiple permission sets, allowing for a mix-and-match approach to permissions.

    Permission sets are not tied to a user's profile, providing additional layers of permissions without modifying the base profile.

3. Temporary Permissions

    Grant temporary permissions for special projects or temporary roles without changing the user's profile.

    Ideal for scenarios like cross-functional teams or temporary assignments..

1. Organization-Level Security

Login Access Policies

IP Restrictions: Control which IP ranges users can log in from.
Login Hours: Specify the hours during which users can log in to Salesforce.

Two-Factor Authentication (2FA)

Enforce 2FA for an additional layer of security during login.

2. Object-Level Security

Profiles

Define what actions (Create, Read, Update, Delete) users can perform on each object.
Control access to apps, tabs, and system permissions.

Permission Sets

Extend permissions to users without changing their profiles.
Useful for granting temporary or additional access.

3. Field-Level Security

Field Accessibility

Control visibility and editability of individual fields within an object.
Set at the profile and permission set levels.

4. Record-Level Security

Organization-Wide Defaults (OWD)

Organization-Wide Defaults (OWD) in Salesforce are the baseline level of access to data records for all users within an organization. They are the first line of defense in the security model and determine the default sharing settings for records. OWD settings can be configured for each standard and custom object in Salesforce and are crucial for controlling data visibility across the organization. Here's a detailed look at OWD:

Key Components of Organization-Wide Defaults

1. Access Levels:
   Private: Users can only see and edit records they own. Other users cannot see these records unless explicitly shared.
   Public Read Only: All users can view records, but only owners or users with higher-level permissions can edit them.
   Public Read/Write: All users can view and edit records.
   Public Read/Write/Transfer: Users can view, edit, and transfer ownership of records (available for some objects like Leads and Cases).
   Controlled by Parent: The access to a child record is determined by the parent record's sharing settings.
2. Objects Covered:
   Standard objects such as Accounts, Contacts, Opportunities, Cases, Leads, etc.
   Custom objects created by the organization.

Role Hierarchies

Role Hierarchies are used to control data access and visibility within an organization by structuring users in a hierarchical format. This hierarchy mimics the organizational structure and helps in establishing data visibility rules, ensuring that managers can access the data of their subordinates. Here's an in-depth look at Role Hierarchies in Salesforce:

Key Features of Role Hierarchies

1. Inheritance of Record Access
    - Users at higher levels in the hierarchy (e.g., managers) automatically gain access to all records owned by users below them.
    - This access is based on the user's role and the roles of their subordinates.
2. Record-Level Security

Works in conjunction with Organization-Wide Defaults (OWD) to open up record access.
Role hierarchies do not override field-level security or other security settings.

3. Customizable Hierarchies
    - Roles can be customized to fit the specific needs and structure of an organization.
    - Multiple levels can be created to accurately represent the organization.

Example Scenario

Scenario: A Sales Manager needs access to all Opportunities owned by their Sales Reps.

Implementation:

1. Set OWD for Opportunities to Private.
2. Create a Role Hierarchy:
    - CEO > VP of Sales > Sales Manager > Sales Rep.
3. Assign Users to Roles:
    - Assign the Sales Manager and Sales Reps to their respective roles.

4. Verify Access:
   The Sales Manager will automatically have access to Opportunities owned by the Sales Reps under them.

Sharing Rules

Sharing Rules in Salesforce are used to extend data access to users in public groups, roles, or territories. They are particularly useful when you need to open up access beyond what is defined by Organization-Wide Defaults (OWD) and Role Hierarchies. Sharing Rules provide a flexible way to share records based on criteria or ownership, ensuring that the right users can access the data they need.

Key Features of Sharing Rules

1. Types of Sharing Rules
   Owner-Based Sharing Rules: Share records based on the record owner.
   Criteria-Based Sharing Rules: Share records based on field values (criteria).
2. Access Levels
   Read Only: Users can view but not edit the records.
   Read/Write: Users can view and edit the records.
3. Target Groups
   Public Groups : A group of users defined by the administrator.
   Roles: Specific roles in the role hierarchy.
   Roles and Subordinates: Arole and all roles below it in the hierarchy.
   Territories: Specific territories in a territory management setup.

Manual Sharing

Manual Sharing in Salesforce is a way to grant specific users access to individual records on an as-needed basis. This feature is particularly useful when you need to provide access to records that are otherwise restricted by Organization-Wide Defaults (OWD) and other sharing settings. Here’s an in-depth look at Manual Sharing in Salesforce:

Key Features of Manual Sharing

1. Record-Level Access
    - Allows sharing of individual records with specific users or groups.
    - Provides fine-grained control over data access.
2. Temporary and Situational Access
    - Ideal for granting temporary access to records for special cases or specific tasks.
    - Can be used to handle exceptional situations without changing global sharing settings.
3. User Interface Access
    - Users with the appropriate permissions can manually share records directly from the record detail page.
    - Typically available to record owners, users above the owner in the role hierarchy, and administrators.

Manual Sharing Capabilities

- Who Can Share Records Manually?
    - Record Owners: The user who owns the record.
    - Users Above the Owner in the Role Hierarchy: Managers or supervisors.
    - Administrators: Users with the “Modify All Data” permission.
    - Users with Full Access: Those granted full access through sharing settings.
- Levels of Access
    - Read Only: Allows the user to view the record.
    - Read/Write: Allows the user to view and edit the record.
    - Full Access: Allows the user to view, edit, delete, and share the record further.

Sharing Sets

Sharing Sets in Salesforce are a feature specifically designed for communities (now known as Experience Cloud sites) that allow you to grant community users access to records that are associated with their accounts or contacts. Sharing Sets simplify the

process of granting access to records without the need for complex sharing rules or manual sharing.

Key Features of Sharing Sets

1. Community-Specific Access
   Designed to grant access to external users in communities.
   External users can include customer, partner, or employee community users.
2. Association-Based Access
   Grants access based on the user's account or contact relationship.
   Simplifies the sharing model by leveraging existing relationships between records.
3. Automated Sharing
   Automatically assigns access when a user is added to a community.
   Removes the need for manual sharing or extensive sharing rules.

How Sharing Sets Work

Access Based on Profiles: Sharing Sets are configured based on user profiles, which means you can define different access rules for different types of community users. **Target Object Access:** You can specify which objects the Sharing Set applies to and what level of access (Read Only or Read/Write) users should have. **Criteria-Based:** Sharing Sets use criteria to determine which records are shared. Typically, this is based on matching the user's account or contact to the records.

Apex Sharing

Apex Sharing in Salesforce allows developers to programmatically grant or revoke access to records beyond what is possible with the standard sharing settings, such as Organization-Wide Defaults (OWD), Role Hierarchies, Sharing Rules, and Manual Sharing. With Apex Sharing, you can enforce custom logic to determine which users have access to specific records based on your organization's requirements.

Key Features of Apex Sharing

1. Custom Access Logic: Developers can implement custom logic to determine record access, enabling complex sharing scenarios that cannot be achieved with standard Salesforce sharing settings alone.
2. Granular Control: Apex Sharing provides fine-grained control over record access, allowing developers to specify exactly which users or groups should have access to specific records and under what conditions.
3. Dynamic Sharing: Sharing rules can be dynamically applied based on changing business requirements or record attributes, ensuring that access remains up-to-date and aligned with organizational needs.

1. **`share` and `unshare` Methods**

   The `share` method is used to grant access to records programmatically.
   The `unshare` method is used to revoke access to records that were previously shared.

```
public class ShareRecordExample {

    // Method to share a record with a specific user

    public static void shareRecordWithUser(Id recordId, Id userId) {

        // Create a new sharing row for the record

        AccountShare recordShare = new AccountShare();

        recordShare.ParentId = recordId; // Set the record Id

        recordShare.UserOrGroupId = userId; // Set the user Id

        recordShare.AccountAccessLevel = 'Read'; // Set the access level (Read, Edit, Full)

        recordShare.OpportunityAccessLevel = 'None'; // Set the access level for related opportunities
```

```
            insert recordShare; // Insert the sharing row

        }

    }
```

Flows

Flows are powerful automation tools in Salesforce that allow you to automate business processes by guiding users through a series of screens and performing actions in Salesforce based on user input or predefined criteria. Flows can be used to streamline and automate a wide range of tasks, from simple data entry to complex business processes.

Key Features of Flows

1. **Visual Design:** Flows are built using a point-and-click visual designer, making them easy to create and modify without any code.
2. **Drag-and-Drop Elements:** Flows consist of various elements such as screens, variables, decisions, and actions, which can be dragged onto the canvas and connected to create a flowchart-like sequence of steps.
3. **Reusable Components:** Flows can be reused across multiple processes and can include reusable elements such as subflows and components.
4. **Integration:** Flows can integrate with other Salesforce features such as Process Builder, Apex, and external systems using Apex actions and invocable methods.
5. **Mobile-Ready:** Flows are mobile-friendly and can be run on both desktop and mobile devices, allowing users to complete tasks from anywhere.

Types of Flows

Salesforce comes with two main Flow types – a Screen Flow and an Auto-launched Flow. The second type is further divided **into four more types.** Hence, collectively, we have five Flow types in Salesforce:

1. Screen Flow
2. Schedule-Triggered Flow
3. Record-Triggered Flow
4. Platform Event-Triggered Flow
5. Auto-Launched Flow.

Screen Flow

Screen Flow requirements are used whenever we need user input. With a screen sample Flow implementation, you can **create a screen** (a custom UI) that displays messages, takes input, and guides the user throughout the Flow process.While creating a Screen Element Flow, click on **the "Screen" element.** You'll see a bunch of components to add to the screen. For example:

1. Text area
2. Picklist
3. Lookup
4. Name
5. Radio custom buttons
6. Checkboxes
7. Date and time ranges.

You can create a complete Form to get all the database records, tables, display notes, and warnings. Once you set up your screen and **take inputs** from the user, you can add elements to create contact records, single update records, perform external actions, etc.

AScreen Flow can prove beneficial in many aspects, such as reducing the requirement for multiple validation rules, providing a logical flow of targeting input from the user, and much more.

Schedule-Triggered Flow

As the name suggests, a Schedule-Triggered Flow is scheduled and launched at **the specified time** for the given frequency. You can compare this Flow type with Apex batch jobs.When you create a Schedule-Triggered Flow, you get to set the schedule of Flow. It asks for the date and time along with the frequency. A Schedule-Triggered Flow can be set to run once, daily, or weekly.To monitor an already scheduled Flow, you can go to **the "Scheduled Jobs"** page in the "Setup."

Record-Triggered Flow

ARecord-Triggered Flow is mainly incorporated when there is a need to make additional updates on the triggered record. When you create a new Flow and select a Record-Triggered Flow, you need to consider **the object** on which you want Flow.

You also need to take into account a Record-Triggered Flow, which can be triggered when:

A**new record** is created
An **existing record** is updated
A**record is created** or updated
A**record is deleted.**

There are a few points that you need to consider before using this Flow:

ARecord-Triggered Flow can only **make changes** in the record's field values.
It can't **update records** other than the triggered record.
It **doesn't support Flow** actions other than "Assignment," "Decision," "Loop Element," and "Get Records."
If an object **hosts more** than one Record-Triggered Flow, you can't determine the order of execution of each individual Flow.

Platform Event-Triggered Flow

APlatform Event-Triggered Flow can **help you manage** all your Salesforce automations in one Flow. An Auto-Launched Flow runs in the background and is launched when a new platform event message is received. It can process up to 2K event messages at a time. Although, the order of execution of these batches is unknown. A platform event allows communication of internal or external applications to Salesforce. It is based on **event-driven architecture** and follows the publish and subscribe model. It also mainly handles the queue of incoming and processed events.

Auto-Launched Flow

Anon-trigger Auto-Launched Flow is invoked only by Apex class, REST API, or a Process Builder. This Flow type **covers abstraction,** as you can perform complex problems triggered and solved in the background without letting the user know anything.

Calling an Apex Class in Flow:

1. Create an Invocable Apex Method:
   Define a method in your Apex class with the `@InvocableMethod` annotation. This method can accept input parameters and return data to the Flow.

```
public with sharing class MyApexClass {

  @InvocableMethod(label='My Invocable Method' description='Description of my invocable method')

  public static List<MyWrapperClass> myMethod(List<String> inputParams) {

  // Your logic here

  return myResultList;

  }

}
```

2. Use the Apex Action in Flow:
   In your Flow, add an Apex action and select the invocable method you created. Provide input parameters as needed.

Integration in salesforce

Integration in Salesforce refers to the process of connecting Salesforce with other systems, applications, or services to exchange data and automate business processes. Integrations allow organizations to streamline operations, improve data accuracy, and provide a unified view of information across different platforms. Here's an overview of integration in Salesforce:

Key Concepts in Integration:

1. Data Integration:
   - Data integration involves synchronizing data between Salesforce and external systems to ensure consistency and accuracy.
   - Common data integration scenarios include bidirectional synchronization of customer data, product information, order details, and more.
2. Process Integration:
   - Process integration focuses on automating business processes that span multiple systems or applications.
   - This includes workflow orchestration, event-driven automation, and real-time processing of data across different platforms.
3. User Interface Integration:
   - User interface integration enables seamless user experiences by embedding Salesforce components within external applications or vice versa.
   - This allows users to access Salesforce functionality from within other systems and vice versa, without switching between applications.
4. Authentication and Security:
   - Integration requires secure authentication mechanisms to ensure that only authorized users and systems can access Salesforce data.
   - Salesforce provides various authentication methods such as OAuth, username-password authentication, and session ID authentication.
5. APIs and Integration Tools:
   - Salesforce offers a variety of APIs (Application Programming Interfaces) and integration tools to facilitate data exchange and process automation.
   - These include SOAP API, REST API, Bulk API, Streaming API, Platform Events, and Integration Hub.

REST API:

1. Representational State Transfer (REST):

   REST is an architectural style for designing networked applications, often used in web services development.
   It is based on the principles of statelessness, client-server architecture, and uniform interface.
   RESTful APIs use standard HTTP methods (GET, POST, PUT, DELETE) to perform operations on resources, represented by URLs (Uniform Resource Locators).
   Data is typically exchanged in lightweight formats such as JSON (JavaScript Object Notation) or XML (eXtensible Markup Language).
   REST APIs are commonly used for building scalable, stateless, and interoperable web services.

Advantages of REST API:

Lightweight: REST APIs use lightweight data formats such as JSON, making them efficient for data exchange.
Flexibility: REST APIs provide flexibility in data formats and support a wide range of clients, including web browsers, mobile devices, and IoT devices.
Stateless: RESTful services are inherently stateless, which simplifies scalability and reliability.
Caching: REST APIs can leverage HTTP caching mechanisms to improve performance and reduce server load.
Web Standards: REST APIs are based on standard web protocols such as HTTP and HTTPS, ensuring compatibility and interoperability.

Use Cases for REST API:

Mobile Applications: Building APIs for mobile apps to access backend services and data.
Web Services: Exposing web services for third-party integration and external system communication.
Single-Page Applications (SPAs): Integrating with client-side JavaScript frameworks like AngularJS, React, or Vue.js.
Internet of Things (IoT): Exchanging data with IoT devices and sensors over HTTP.

SOAP API:

1. Simple Object Access Protocol (SOAP):
   SOAP is a protocol for exchanging structured information in the implementation of web services.
   It uses XML for message formatting and relies on a set of standards for message exchange, security, and reliability.
   SOAP APIs define operations and data structures using a formal contract called a Web Services Description Language (WSDL).
   SOAP messages are typically transported over HTTP, SMTP, or other application-layer protocols.

Advantages of SOAP API:

Formal Contract: SOAP APIs use WSDL to define operations and data structures, providing a formal contract for service consumers.
Security: SOAP supports built-in security features such as WS-Security for message-level encryption, authentication, and integrity.
Reliable Messaging: SOAP includes features for reliable messaging, including message acknowledgment, retries, and transaction support.
Interoperability: SOAP APIs are designed for interoperability across different platforms, languages, and operating systems.
Tooling Support: SOAP APIs have mature tooling support, including code generation tools, IDE plugins, and debugging tools.

Use Cases for SOAP API:

Enterprise Integration: Integrating with enterprise systems such as ERP (Enterprise Resource Planning) and CRM (Customer Relationship Management) systems.
Legacy Systems: Communicating with legacy systems that support SOAP-based web services.
Government and Healthcare: Interacting with government agencies and healthcare providers that require standardized messaging formats.
Financial Services: Exchanging financial data and performing transactions with banking and financial institutions.

Choosing Between REST and SOAP API:

REST API is preferred for lightweight, stateless, and scalable web services, especially in modern web and mobile application development.
SOAP API is suitable for complex, enterprise-grade integrations that require formal contracts, security features, and reliable messaging.

Terms Using in Salesforce

1. **Authorization:**
   Authorization refers to the process of determining what actions a user or system is allowed to perform within Salesforce.
   It involves granting permissions to access resources and perform operations based on the user's identity and roles.
2. **Authentication:**
   Authentication is the process of verifying the identity of a user or system attempting to access Salesforce.
   It typically involves providing credentials such as username/password, session tokens, or digital certificates to prove identity.
3. **Web Services:**
   Web services are software systems designed to allow interaction between different applications over a network.
   In Salesforce, web services are used to expose or consume functionality and data between Salesforce and external systems.
4. **Named Credential:**
   Named Credentials are a secure and convenient way to authenticate to external web services from Salesforce.
   They store authentication details such as username/password or OAuth tokens securely and can be used in Apex code, flows, or Process Builder.
5. **Auth Provider (Authentication Provider):**
   Auth Providers in Salesforce allow users to log in to Salesforce using external identity providers such as Google, Facebook, or Microsoft.
   They enable single sign-on (SSO) and federated authentication for Salesforce orgs.
6. **Connected App:**
   AConnected App is an external application that integrates with Salesforce using APIs and OAuth.

It defines the integration settings, permissions, and security policies for accessing Salesforce data.

7. JWT (JSON Web Token):

```
    @HttpGet

    global static String doGet() {

    // Implementation logic for GET request

    }




    @HttpPost

    global static String doPost(String requestBody) {

    // Implementation logic for POST request

    }

}
```

2. **@HttpGet, @HttpPost, @HttpPut, @HttpDelete:**
   These annotations are used to define methods within a RESTful web service class that handle specific HTTP methods (GET, POST, PUT, DELETE).
   Each method annotated with one of these annotations corresponds to a specific HTTP method and processes incoming requests accordingly.
3. **@RestResource(unsupportedHttpMethods='PATCH'):**

Salesforce does not directly support the PATCH HTTP method in Apex REST services.
However, you can use the `unsupportedHttpMethods attribute to specify` that the class does not support certain HTTP methods.

Example:

```
@RestResource(urlMapping='/myApi/*', unsupportedHttpMethods='PATCH')

global class MyRestResource {

 @HttpPatch

 global static String doPatch() {

 // Implementation logic for PATCH request

 }

}
```

4. **@HttpDelete, @HttpPatch, @HttpPut:**
   Although Salesforce does not directly support these HTTP methods in Apex REST services, you can use these annotations for documentation purposes or when working with other platforms that support them.
5. **@HttpHeader:**
   The @HttpHeader annotation allows you to specify custom HTTP headers for a method in a RESTful web service.

It is used to access headers sent with the HTTP request.

Example:

```
@HttpPost

global static String doPost(String requestBody, String contentType) {

 // Implementation logic for POST request

}
```

6. **@RestResource(urlMapping='/myApi/*', compression='GZIP'):**
   The compression attribute specifies the compression format for responses returned by the RESTful web service.
   Supported values are 'GZIP' and 'DEFLATE'.

Example:

```
@RestResource(urlMapping='/myApi/*', compression='GZIP')

global class MyRestResource {

 @HttpGet

 global static String doGet() {
```

```
  // Implementation logic for GET request


  }
```

Apex code for http callout

```
public class HttpCalloutExample {
   // Define a method to make an HTTP callout
   public static void makeHttpCallout() {
       // Define the endpoint URL
       String endpoint = 'https://jsonplaceholder.typicode.com/posts/1';

        // Instantiate a new HTTP request object
       HttpRequest request = new HttpRequest();

        // Set the HTTP method (GET, POST, PUT, DELETE, etc.)
       request.setMethod('GET');

        // Set the endpoint URL
       request.setEndpoint(endpoint);

        // Instantiate a new HTTP object
       Http http = new Http();

        try {
           // Send the HTTP request and get the response
           HttpResponse response = http.send(request);

            // Check if the request was successful (HTTP status code 200)
           if (response.getStatusCode() == 200) {
               // Parse the JSON response
               Map<String, Object> jsonResponse = (Map<String, Object>)
JSON.deserializeUntyped(response.getBody());

                // Extract data from the JSON response
               String title = (String) jsonResponse.get('title');
               String body = (String) jsonResponse.get('body');

                // Log the response data
               System.debug('Title: ' + title);
               System.debug('Body: ' + body);
```

```
            } else {
                // Log error message if the request was not successful
                System.debug('HTTP callout failed with status code: ' +
response.getStatusCode());
            }
        } catch (Exception e) {
            // Log any exceptions that occur during the HTTP callout
            System.debug('Error occurred during HTTP callout: ' +
e.getMessage());
        }
    }
 }
```

In this example:

- We define a static method `makeHttpCallout() in the HttpCalloutExample class` to make the HTTP callout.
- We specify the endpoint URL (`https://jsonplaceholder.typicode.com/posts/1)that we want to make the` HTTP GET request to.
- We create an HttpRequest object and set the HTTP method (GET)and endpoint URL.
- We create an Http object to send the HTTP request.
- We send the HTTP request using the send() method of the Http object and get the HttpResponse object in return.
- We check if the request was successful by verifying the HTTP status code (200).
- If the request was successful, we parse the JSON response using the JSON.deserializeUntyped() method and extract the title and body fields from the response.
- We log the response data using `System.debug()`.
- If the request was not successful, we log an error message with the HTTP status code.
- We catch and log any exceptions that occur during the HTTP callout using a try-catch block.

What is the difference between SOAP and REST?

SOAP is a protocol	REST is an architectural style.
SOAP stands for Simple Object Access Protocol.	REST stands for REpresentational State Transfer.

SOAP can't use REST	REST can use SOAP
SOAP uses services interfaces to expose the business logic.	REST uses URI to expose business logic.
SOAP defines standards to be strictly followed.	REST does not define too many standards like SOAP.
SOAP defines its own security.	RESTful web services inherit security measures from the underlying transport.
SOAP permits XML data format only.	REST permits different data formats such as Plain text, HTML, XML, JSON etc.
SOAP is less preferred than REST.	REST is more preferred than SOAP.

JSON vs XML

JSON	**XML**
JavaScript Object Notation has a type like String, number, Object, Boolean Extensible markup language is type less, and should be string.	It is a way of representing objects
Retrieving value is easy.	Retrieving value is difficult.
It does not provide any support for namespaces.	It supports namespaces.
It is less secured	It is more secure than JSON

Salesforce License Types and Use Cases

Salesforce License Types and Use Cases

License Type	Description	Use Case	Key Features
Salesforce	Full access to standard Salesforce features.	Users who need comprehensive CRM functionality, including sales, service, and custom apps.	Access to all standard and custom objects.

Salesforce Platform	Access to custom apps and limited standard CRM functionality.	Users needing access to custom applications with limited CRM features.	Custom objects, accounts, contacts, reports, and dashboards.
Salesforce Platform One	Restricted access, typically for external users.	External users needing limited access to custom applications.	Limited custom objects and tabs.
Chatter Free	Basic access to Chatter for internal users.	Employees who need to collaborate but do not need access to other Salesforce data.	Chatter groups, profiles, and feeds.
Chatter Only (Chatter Plus)	Access to Chatter and limited Salesforce objects.	Internal users needing collaboration tools and basic CRM access.	Accounts, contacts, reports, and dashboards in addition to Chatter.
Chatter External	Access to Chatter for external users.	External stakeholders, such as customers or partners, need to collaborate within Chatter groups.	Chatter groups and feeds.
Customer Community	Access to customer-facing communities with limited CRM features.	External customers require access to community content and basic case management.	Access to cases, accounts, and contacts.
Customer Community Plus	Enhanced access for external customers, including reports.	External users needing more advanced community features, like dashboards and custom objects.	Reports, dashboards, cases, and custom objects.
Partner Community	Comprehensive access for external partners, including sales data.	Partners and resellers needing access to leads, opportunities, and other sales-related objects.	Leads, opportunities, campaigns, and custom objects.
Health Cloud	Tailored for healthcare organizations.	Healthcare providers need to manage patient data, care plans, and health records.	Patient data, care plans, EHR integration.

Financial Services Cloud	Designed for financial services organizations.	Financial advisors and service reps needing to manage client relationships, financial accounts, and plans.	Client data, financial accounts, wealth management tools.
Service Cloud	Full access to customer service and support features.	Customer service reps need to manage cases, service contracts, and knowledge base articles.	Cases, service contracts, entitlements, knowledge base.
Sales Cloud	Full access to sales features.	Sales reps and managers need to manage leads, opportunities, forecasts, and campaigns.	Leads, opportunities, forecasts, campaigns.
Einstein Analytics	Advanced analytics and AI-powered insights.	Users needing sophisticated data analytics and visualization tools to gain insights from Salesforce data.	Data integration, analytics dashboards, AI insights.
Pardot	Marketing automation features.	Marketing teams need to manage lead generation, email campaigns, and automated marketing workflows.	Email marketing, lead scoring, automated workflows.
CPQ (Configure, Price, Quote)	Tools for managing product configurations, pricing, and quoting.	Sales teams need to streamline the quoting process with accurate pricing and configurations.	Product configurations, pricing, quoting tools.
Salesforce Developer	Full access to development tools and environments.	Developers building and customizing applications on the Salesforce platform.	Development environments, API access, debugging tools.
Salesforce Administrator	Access to administrative and setup functions.	Administrators managing Salesforce setup, user permissions, and configurations.	Setup and configuration tools, user management, security settings.

For more valuable content like this

FOLLOW ME ON LINKEDIN

Sandeep Sharma

Be sure to repost to share with your audience