IIIrd Year 1st Sem Salesforce Notes I-V units

Common for Alpha and Beta

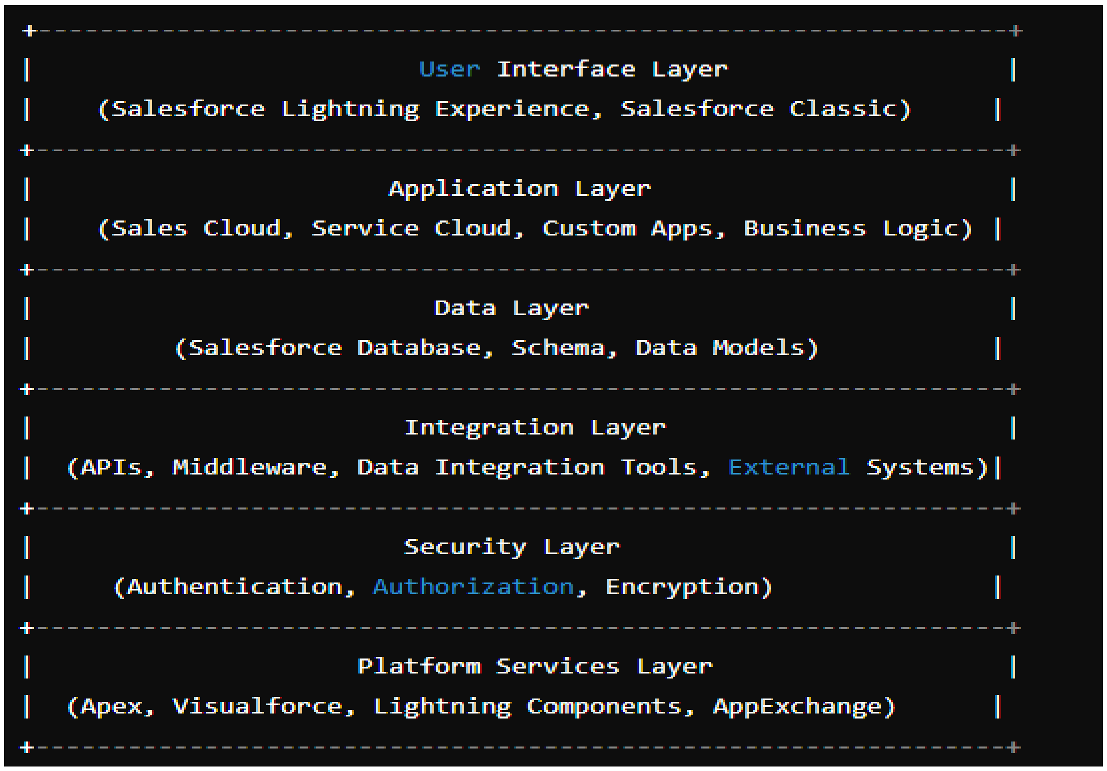
IIIrd Year 1st Sem Salesforce Notes I-V units
Questions

1 **What is Salesforce ? Explain the salesforce architecture with a diagram.**

Salesforce is customer success platform, designed to help sell, service, market, analyze, and connect with customers.Salesforce has everything one needs to run a business from anywhere.Salesforce helps to use standard products and features, manage relationships with prospects and customers, collaborate and engage with employees and partners, and store your data securely in the cloud. It is designed to improve business relationships, streamline processes, and enhance profitability by providing tools for sales, customer service, marketing, and more.

Salesforce architecture is designed to support a wide range of applications and services. It leverages a multi-tenant, cloud-based infrastructure to ensure scalability, security, and reliability. The architecture can be understood through its various layers and components:

Salesforce Architecture Layers



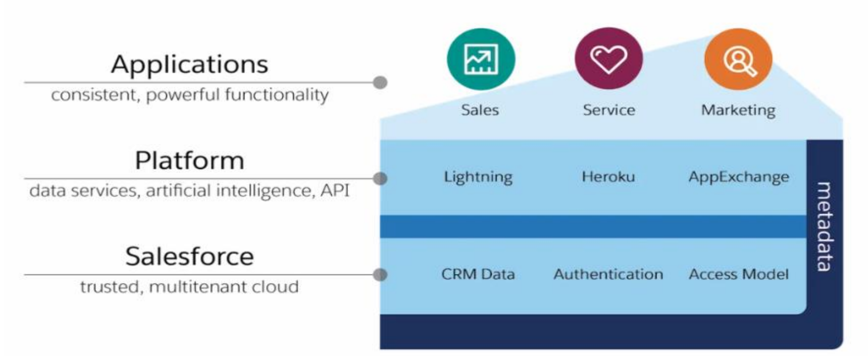
```
+---------------------------------------------------------------+
|                      User Interface Layer                      |
|    (Salesforce Lightning Experience, Salesforce Classic)      |
+---------------------------------------------------------------+
|                       Application Layer                        |
|    (Sales Cloud, Service Cloud, Custom Apps, Business Logic) |
+---------------------------------------------------------------+
|                          Data Layer                            |
|         (Salesforce Database, Schema, Data Models)            |
+---------------------------------------------------------------+
|                       Integration Layer                        |
|  (APIs, Middleware, Data Integration Tools, External Systems)|
+---------------------------------------------------------------+
|                        Security Layer                          |
|     (Authentication, Authorization, Encryption)               |
+---------------------------------------------------------------+
|                    Platform Services Layer                     |
|  (Apex, Visualforce, Lightning Components, AppExchange)       |
+---------------------------------------------------------------+
```

1. **User Interface (UI) Layer:**
   - **Components:** Web-based interface, Salesforce Lightning Experience, Salesforce Classic.
   - **Function:** Provides a user-friendly interface for interacting with Salesforce data and applications. It allows users to customize dashboards, reports, and page layouts.
2. **Application Layer:**
   - **Components:** Salesforce applications, custom objects, and business logic.

   - **Function:** This layer contains the core business applications and logic. It includes Salesforce's standard applications (e.g., Sales Cloud, Service Cloud) and custom applications built using Salesforce's development tools.
3. **Data Layer:**
   - **Components:** Salesforce database, schema, and data models.
   - **Function:** Manages the storage and retrieval of data. Salesforce uses a multi-tenant architecture where a single instance of the software serves multiple customers. Each customer's data is securely partitioned and managed.
4. **Integration Layer:**
   - **Components:** APIs (REST, SOAP), middleware, and integration tools.
   - **Function:** Facilitates communication between Salesforce and external systems. This includes data synchronization, third-party integrations, and connecting with other applications through APIs.
5. **Security Layer:**
   - **Components:** Authentication, authorization, encryption.
   - **Function:** Ensures that data and applications are secure. Salesforce provides features like user authentication, role-based access control, and data encryption to protect information.
6. **Platform Services Layer:**
   - **Components:** Salesforce Platform (AppExchange, Apex, Visualforce, Lightning Components).
   - **Function:** Provides a suite of development tools and services to build and deploy custom applications. This includes a development environment (Apex), user interface components (Visualforce, Lightning Components), and integration tools.

<u>Salesforce is structured at high-level :</u>

The underlying structure of the Salesforce architecture consists of a set of stacked layers, where each layer can be easily identified by how accessible it is to different types of end-users. The deeper the layers are lesser the access and control users have over them.



Let's try to understand the functionality and importance of each layer in this architecture.

The bottom-most layer of the architecture is familiar to us. It is simply the cloud Salesforce is hosted on.

The middle layer shown in the diagram, Salesforce Platform, is the most crucial component in the Salesforce architecture. It provides all the key features common and necessary to various Salesforce products such as data services, artificial intelligence, and APIs. The Platform layer uses another important component in the Salesforce architecture, metadata, to unify the actions taken by different users through numerous products. With application development features like Lightning, Heroku, and AppExchange, this layer grants developers the access to manipulate some Salesforce processes implemented under the hood.
The top-most Application layer houses all the popular Salesforce products, like Sales Cloud, Marketing Cloud, and Service Cloud, and any application you build using the Salesforce Platform. The majority of the user interactions on the Salesforce system takes place in this layer. While each application hosted here caters to a unique set of CRM use cases, as a whole, all of them are integrated with the key features provided by the Platform layer, such as predictive analysis and development framework.

2 **Define the following with an example in salesforce :**
a) objects b) Records c) Fields d) App

a) Objects are tables in the Salesforce database that store a particular kind of information. There are standard objects like Accounts and Contacts and custom objects like the Property object you see in the graphic.
b) Records are rows in object database tables. Records are the actual data associated with an object. Here, the Contemporary City Living property is a record.
c) Fields are columns in object database tables. Both standard and custom objects have fields. On our Property object, we have fields like Address and Price.
d) An app in Salesforce is a set of objects, fields, and other functionality that supports a business process. You can see which app you're using and switch between apps using the App Launcher (⠿).

3 **What is Data Modelling? Explain about standard and custom objects with an example.**

A data model is a way to model what database tables look like. In Salesforce CRM, we think about database tables as objects, we think about columns as fields, and rows as records. Salesforce supports several different types of objects. There are standard objects, custom objects, external objects, platform events, and BigObjects. In this module, we focus on the two most common types of objects: standard and custom.
Standard objects are objects that are included with Salesforce. Common business objects like Account, Contact, Lead, and Opportunity are all standard objects.

Custom objects are objects that you create to store information that's specific to your company or industry. For DreamHouse, D'Angelo wants to build a custom Property object that stores information about the homes his company is selling.
Objects are containers for your information, but they also give you special functionality. For example, when you create a custom object, the platform automatically builds things like the page layout for the user interface.

Create a Custom Object

1. Click the gear icon at the top of the page and launch setup.
2. Click the Object Manager tab.
3. Click Create | Custom Object in the top-right corner.
4. For Label, enter Property. Notice that the Object Name and Record Name fields auto-fill.
5. For Plural Label, enter Properties.
6. Prior to saving the custom object, scroll to the bottom of the page and select the checkbox Launch New Custom Tab Wizard after saving this custom object.
7. Leave the rest of the values as default and click Save.
8. On the New Custom Object Tab page, click the Tab Style field and select a style you like. The style sets the icon to display in the UI for the object.
9. Click Next, Next, and Save.

4 **What are the different types of Object relationships? Write the steps to create and modify lookup and master-detail relationship.**

Object relationships are a special field type that connects two objects together.
There are two main types of object relationships: lookup and master-detail.
Lookup Relationships
In our Account to Contact example above, the relationship between the two objects is a **lookup relationship**. A lookup relationship essentially links two objects together so that you can "look up" one object from the related items on another object.
Lookup relationships can be one-to-one or one-to-many. The Account to Contact relationship is one-to-many because a single account can have many related contacts. For our DreamHouse scenario, you could create a one-to-one relationship between the Property object and a Home Seller object.
Master-Detail Relationships
While lookup relationships are fairly casual, **master-detail relationships** are a bit tighter. In this type of relationship, one object is the master and another is the detail. The master object controls certain behaviors of the detail object, like who can view the detail's data.
For example, say the owner of a property wanted to take their home off the market. DreamHouse wouldn't want to keep any offers made on that property. With a master-detail relationship between Property and Offer, you can delete the property and all its associated offers from your system.

Typically, you use lookup relationships when objects are only related in some cases. Sometimes a contact is associated with a specific account, but sometimes it's just a contact. Objects in lookup relationships usually work as stand-alone objects and have their own tabs in the user interface.
In a master-detail relationship, the detail object doesn't work as a stand-alone. It's highly dependent on the master. In fact, if a record on the master object is deleted, all its related detail records are deleted as well. When you're creating master-detail relationships, you always create the relationship field on the detail object.
Finally, you could run into a third relationship type called a hierarchical relationship. Hierarchical relationships are a special type of lookup relationship. The main difference between the two is that hierarchical relationships are only available on the User object. You can use them for things like creating management chains between users.
Look-up Relationship

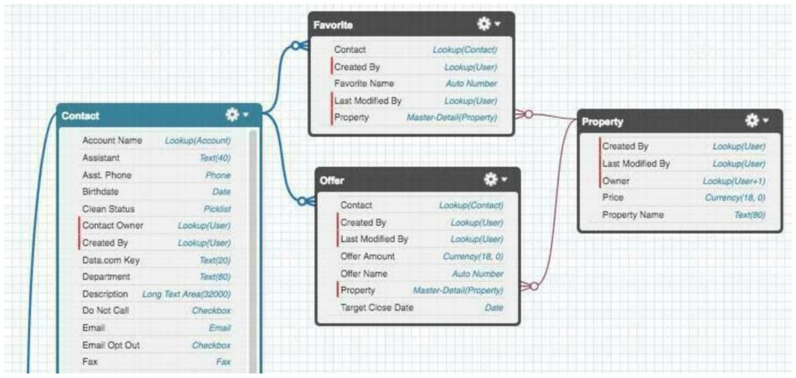
1. From Setup, go to **Object Manager | Favorite**.
2. On the sidebar, click **Fields & Relationships**.
3. Click **New**.
4. Choose **Lookup Relationship** and click **Next**.
5. For Related To, choose **Contact**. For the purposes of DreamHouse, contacts represent potential home buyers.
6. Click **Next**.
7. For Field Name, enter Contact, then click **Next**.
8. Click **Next**, **Next**, **Next**, and **Save**.

Create a Master-Detail Relationship
Now, create a second relationship field. You want a master-detail relationship where Property is the master and Favorite is the detail.

1. On the Object Manager page for the custom object, click **Fields & Relationships**.
2. Click **New**.
3. Select **Master-Detail Relationship** and click **Next**.
4. For Related To, choose **Property**.
5. Click **Next**.
6. For Field Name, enter Property and click **Next**.
7. Click **Next**, **Next**, and **Save**.

5 **Describe the advantages of using Schema Builder for data modelling. How do we use schema builder to create a schema for a given object model, to add custom object and custom field to your schema?**

Schema Builder is a tool that lets you visualize and edit your data model. It's useful for designing and understanding complex data models. it can help you visualize your data model in a useful way. Schema Builder is a handy tool for introducing your Salesforce customizations to a co-worker or explaining the way data flows throughout your system.



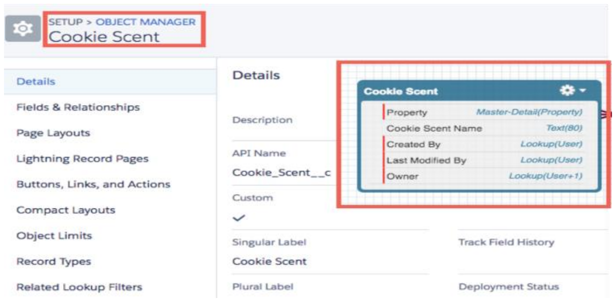
Create an Object with Schema Builder

1. In the left sidebar, click the **Elements** tab.
2. Click **Object** and drag it onto the canvas.
3. Enter information about your object. You can make it whatever you want!
4. Click **Save**.

Create Fields with Schema Builder

Creating fields with Schema Builder is just like creating objects.

1. From the Elements tab, choose a field type and drag it onto the object you just created. Notice that you can create relationship fields, formula fields, and normal fields in Schema Builder.
2. Fill out the details about your new field.
3. Click **Save**.


SETUP > OBJECT MANAGER
Cookie Scent

Details
Fields & Relationships
Page Layouts
Lightning Record Pages
Buttons, Links, and Actions
Compact Layouts
Object Limits
Record Types
Related Lookup Filters

Details

Description

API Name
Cookie_Scent__c

Custom
✓

Singular Label
Cookie Scent

Plural Label

Track Field History

Deployment Status

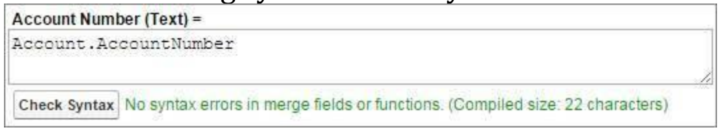
Cookie Scent
Property Master-Detail(Property)
Cookie Scent Name Text(80)
Created By Lookup(User)
Last Modified By Lookup(User)
Owner Lookup(User+1)

6 **Explain why formula fields are useful. Create a simple formula to display an account field on the contact detail page.**

Formula fields allow you to automatically calculate values based on other fields, ensuring data consistency and reducing manual data entry. They can be used to display information from related records, perform calculations, and create dynamic content that updates in real-time as the underlying data changes.

First create a Contact. If you've never created a Contact before, from the App Launcher ( ), find and open Contacts. Click New. Enter any value for Last Name. For Account Name, enter an existing account such as United Oil & Gas Corp. Click Save. Next we create a formula to display the account number on the Contact page.

1. From Setup, open the Object Manager and click Contact.
2. In the left sidebar click Fields & Relationships.

3. Click New.
4. For the field type, select Formula and click Next.
5. Call your field Account Number and select Text for the formula return type. Click Next.
6. In the Advanced Formula Editor, click Insert Field.
   Select Contact | Account | Account Number and then click Insert. Click Check Syntax. If there are no syntax errors, click Next. It's unlikely that you'll find a syntax error in a simple formula like this one, but it's a good idea to get in the habit of checking syntax for every formula.
   
   Account Number (Text) =
   ```
   Account.AccountNumber
   ```

   No syntax errors in merge fields or functions. (Compiled size: 22 characters)
7. Click Next to accept the field-level security settings, then click Save.

7 **Discuss about a roll up summary field . Create a roll up summary field where the total price of all products related to an opportunity is displayed.**

roll-up summary fields calculate values from a set of related records, such as those in a related list. You can create roll-up summary fields that automatically display a value on a master record based on the values of records in a detail record. These detail records must be directly related to the master through a master-detail relationship.
You can perform different types of calculations with roll-up summary fields. You can count the number of detail records related to a master record, or calculate the sum, minimum value, or maximum value of a field in the detail records. For example, you might want:

- A custom account field that calculates the total of all related pending opportunities.
- A custom order field that sums the unit prices of products that contain a description you specify.

A roll-up summary fields are based on master-detail relationships.
There are a few different types of summaries you can use.

Type	**Description**
COUNT	Totals the number of related records.
SUM	Totals the values in the field you select in the Field to Aggregate option. Only number, currency, and percent fields are available.
MIN	Displays the lowest value of the field you select in the Field to Aggregate option for all directly related records. Only number, currency, percent, date, and date/time fields are available.

MAX	Displays the highest value of the field you select in the Field to Aggregate option for all directly related records. Only number, currency, percent, date, and date/time fields are available.

Creating the Summary Field

1. From Setup, open Object Manager and click Account.
2. On the left sidebar, click Fields & Relationships.
3. Click New.
4. Choose the Roll-Up Summary field type, and click Next.
5. For Field Label, enter Sum of Opportunities and click Next.
6. The Summarized Object is the detail object that you want to summarize. Choose Opportunities.
7. Choose the SUM Roll-up type and choose Amount as the Field to Aggregate. If you're unable to see Amount in Field to Aggregate, disable the Advanced Currency Management in your Currency Setup.
8. Click Next, Next, and Save.

8 **What is Picklist? Explain in detail about types of picklist.**

A **Picklist** in Salesforce is a type of field that allows users to choose a value from a predefined list of options. This helps in maintaining data consistency and standardization, as users are restricted to selecting only from the values provided in the picklist.

three types of picklists:

1. Standard
2. Custom
3. Custom Multi-Select

picklist fields can have these properties:

- Restricted
- Dependent or Controlling

Values can be defined three ways:

1. Set individual values when you create the picklist. These are specific to a single picklist field.
2. Use the built-in set of values for the standard picklist fields that come with your Salesforce org.
3. Create a global value set. A global value set is a custom set of values you create to share with more than one picklist field.

Standard Picklists

Standard picklists are the ones that are included in your Salesforce org before any customization. Examples include the Lead Source picklist on the Lead object, the Opportunity Stage picklist on the Opportunity object, and others.

Standard picklist fields sometimes share a standard value set. For example, on the Lead object, the Lead Source picklist values are part of a standard value set. The same values are also used on the Account object for the Account Source picklist field. When you change a value from that set in the Lead Source picklist settings, the change also appears in the Account Source picklist field values.

ustom Picklists
Custom picklists are the ones you create. You can add your own values and configure a custom picklist's behavior. As you create a new custom field, select **Picklist** as the field type.
Here are the steps to create a custom picklist. You can do more customization later.

1. In Setup, click the **Object Manager** tab, and then select the object to contain the picklist field.
2. Click **Fields & Relationships**.
3. Click **New**. Select **Picklist**, and then click **Next**.
4. Enter a Label for the picklist field. The Field Name is automatically assigned. The Field Name is often also called the "API name" since it's a way to reference the field programmatically.
5. Select **Enter values, with each value separated by a new line**.
6. Enter your values.
7. Optionally, sort the values alphabetically or use the first value in the list as the default value, or both. If you select both options, Salesforce alphabetizes the entries and then sets the first alphabetized value as the default. You can use a formula to assign a default value dynamically. We get into that later.
8. Choose whether to restrict this picklist's values to an admin-approved list. This is where you can decide this should be a *restricted* picklist. You can change it later if you find that you do need to accept new values in the field.
9. Click **Next**.
10. Set field-level security for the picklist field, and then click **Next**. This list determines who can see or edit the field.
11. Choose the page layouts on which to include the picklist field.
12. Click **Save**.

Identify existing custom picklists on an object's **Fields & Relationships** page. Like other custom fields, a custom picklist's Field Name ends with **__c**.

Custom Multi-Select Picklists
you want your users to select more than one value from the picklist. When a user picks more than one value, the selected values show in the field, separated by a semicolon.
Restricted Picklists
Restricted picklists keep users from adding new values (either through the API or other apps). This restriction is useful for keeping your data consistent.
Dependent Picklists
A dependent picklist filters values for one picklist based on a selection from another picklist or a checkbox (the *controlling* value) on the same record.
https://trailhead.salesforce.com/content/learn/modules/picklist_admin/picklist_admin_start

9 **What is Data Management? What are the steps involved in importing and exporting data?**

Data Management in Salesforce involves the processes and tools used to handle data effectively within the Salesforce platform. This includes importing, exporting, updating, and maintaining data to ensure that it is accurate, relevant, and up-to-date.

Data Management encompasses various tasks, including:

- **Data Import:** Bringing data from external sources into Salesforce.
- **Data Export:** Extracting data from Salesforce to use outside of the platform.
- **Data Updates:** Modifying existing data within Salesforce.
- **Data Maintenance:** Ensuring data quality and consistency, including deduplication and validation.

Salesforce offers two main methods for importing data.

- **Data Import Wizard—this tool, accessible through the Setup menu, lets you import data in common standard objects, such as contacts, leads, accounts, as well as data in custom objects. It can import up to 50,000 records at a time. It provides a simple interface to specify the configuration parameters, data sources, and the field mappings that map the field names in your import file with the field names in Salesforce.**
- **Data Loader—this is a client application that can import up to 150 million records at a time, of any data type, either from files or a database connection. It can be operated either through the user interface or the command line. In the latter case, you need to specify data sources, field mappings, and other parameters via configuration files. This makes it possible to automate the import process, using API calls.**

Salesforce offers two main methods for exporting data.

- **Data Export Service**—an in-browser service, accessible through the Setup menu. It allows you to export data manually once every 7 days (for weekly export) or 29 days (for monthly export). You can also export data automatically at weekly or monthly intervals. Weekly exports are available in Enterprise, Performance, and Unlimited Editions. In Professional Edition and Developer Edition, you can generate backup files only every 29 days, or automatically at monthly intervals only.
- **Data Loader**—a client application that you must install separately. It can be operated either through the user interface or the command line. The latter option is useful if you want to automate the export process, or use APIs to integrate with another system.

Steps Involved in Importing Data

1. **Plan and Prepare:**
   - **Define Objectives:** Determine what data you need to import and why.
   - **Data Mapping:** Map your external data fields to Salesforce fields to ensure proper alignment.

   - **Clean Data:** Ensure that the data you are importing is clean, accurate, and free from duplicates.
2. **Select an Import Tool:**
   - **Data Import Wizard:** A user-friendly tool for importing standard and custom object data. Ideal for smaller, less complex imports.
   - **Data Loader:** A more advanced tool for importing, updating, and exporting large volumes of data. Suitable for complex data operations.
3. **Prepare Your Data File:**
   - **Format Data:** Ensure your data file (CSV format is commonly used) is properly formatted, with headers matching Salesforce fields.
   - **Review Data:** Check for any inconsistencies or errors in your data file.
4. **Perform the Import:**
   - **Using Data Import Wizard:**
     - Navigate to the Data Import Wizard in Salesforce.
     - Choose the object you want to import data into (e.g., Contacts, Accounts).
     - Upload your CSV file and map the fields.
     - Start the import process and monitor the progress.
   - **Using Data Loader:**
     - Open Data Loader and select the import operation (e.g., Insert, Update).
     - Log in to Salesforce using your credentials.
     - Select the object and upload your CSV file.
     - Map the fields and execute the import.
5. **Verify the Import:**
   - **Check Results:** Verify that the data has been imported correctly by checking records in Salesforce.
   - **Review Errors:** If there are any errors, review the error logs provided by the import tool to address and correct issues.
6. **Clean Up and Validate:**
   - **Validate Data:** Ensure that the imported data meets your quality standards.
   - **Perform Cleanup:** Address any duplicate records or errors that were identified.
   -

Steps Involved in Exporting Data

1. **Plan and Prepare:**
   - **Define Objectives:** Determine which data you need to export and the format required.
   - **Identify Scope:** Choose the objects and records you need to export.
2. **Select an Export Tool:**
   - **Data Export Wizard:** A tool available in Salesforce for exporting data. Suitable for smaller datasets or scheduled exports.
   - **Data Loader:** Can also be used to export data, especially useful for larger datasets.
3. **Perform the Export:**
   - **Using Data Export Wizard:**

        - Navigate to the Data Export Wizard in Salesforce.
        - Choose the objects you want to export.
        - Select the export format (e.g., CSV) and schedule the export if needed.
        - Initiate the export and download the file when ready.
    - **Using Data Loader:**
        - Open Data Loader and select the export operation.
        - Log in to Salesforce.
        - Choose the object and specify the fields to export.
        - Execute the export and download the file.

4. **Verify the Export:**
    - **Check Data:** Review the exported data file to ensure it contains the correct information.
    - **Ensure Completeness:** Confirm that all required data has been exported.
5. **Secure and Manage Data:**
    - **Store Data Safely:** Ensure that exported data is stored securely, especially if it contains sensitive information.
    - **Backup Regularly:** Regularly backup data to prevent loss and ensure continuity.

https://trailhead.salesforce.com/content/learn/modules/lex_implementation_data_management

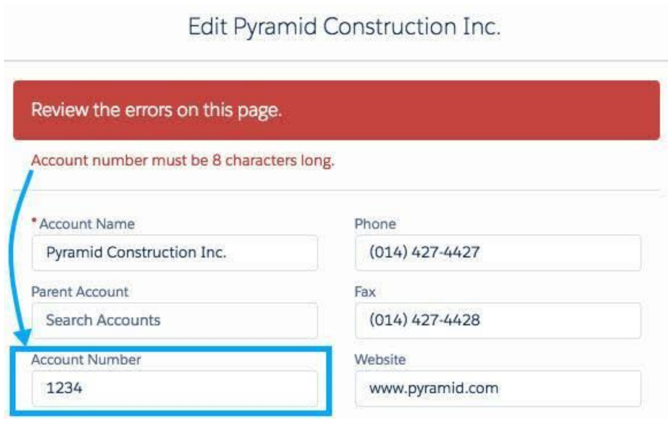
10 **Discuss about Validation Rules.**

Validation rules verify that data entered by users in records meets the standards you specify before they can save it. A validation rule can contain a formula or expression that evaluates the data in one or more fields and returns a value of “True” or “False.” When the validation rule returns a value of "True", this confirms that the data entered by the user contains an invalid value. Validation rules can also include error messages to display to users when they enter invalid values based on specified criteria. Using these rules effectively contributes to quality data. For example, you can ensure that all phone number fields contain a specified format or that discounts applied to certain products never exceed a defined percentage.
Defining Validation Rules
You can create validation rules for objects, fields, campaign members, or case milestones. In these steps, we create a validation rule that fires when a user tries to save an account with an account number of incorrect length.
Creating a Validation Rule

1. From Setup, go to Object Manager and click Account.
2. In the left sidebar, click Validation Rules.
3. Click New.
4. Enter the following properties for your validation rule:

    a. Rule Name: Account_Number_8_Characters
    b. Error Condition Formula: LEN(AccountNumber) <> 8
5. Error Message: Account number must be 8 characters long.
6. To check your formula for errors, click Check Syntax.

7. Click Save to finish**.**


Edit Pyramid Construction Inc.

Review the errors on this page.

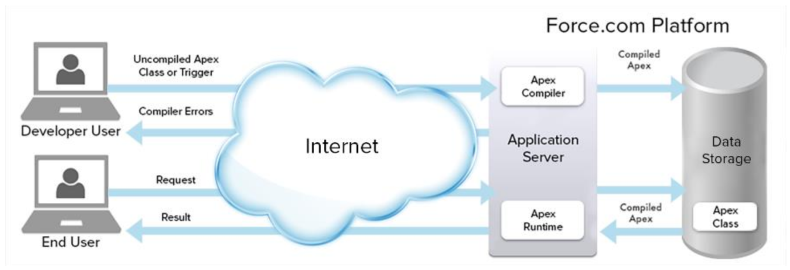
Account number must be 8 characters long.

*Account Name: Pyramid Construction Inc.	Phone: (014) 427-4427
Parent Account: Search Accounts	Fax: (014) 427-4428
Account Number: 1234	Website: www.pyramid.com

11 **Define Apex and write the features of Apex Programming in Salesforce.**

Apex is a strongly typed, object-oriented programming language that allows developers to execute flow and transaction control statements on the Salesforce platform.
Apex is a programming language that uses Java-like syntax and acts like database stored procedures. Apex enables developers to add business logic to system events, such as button clicks, updates of related records, and Visualforce pages.
As a language, Apex is:

- Hosted—Apex is saved, compiled, and executed on the server—the Lightning Platform.
- Object oriented—Apex supports classes, interfaces, and inheritance.
- Strongly typed—Apex validates references to objects at compile time.
- Multitenant aware—Because Apex runs in a multitenant platform, it guards closely against runaway code by enforcing limits, which prevent code from monopolizing shared resources.
- Integrated with the database—It is straightforward to access and manipulate records. Apex provides direct access to records and their fields, and provides statements and query languages to manipulate those records.
- Data focused—Apex provides transactional access to the database, allowing you to roll back operations.
- Easy to use—Apex is based on familiar Java idioms.
- Easy to test—Apex provides built-in support for unit test creation, execution, and code coverage. Salesforce ensures that all custom Apex code works as expected by executing all unit tests prior to any platform upgrades.
- Versioned—Custom Apex code can be saved against different versions of the API.



12 **State sObject data type. Describe in detail about its type with suitable example.**

Apex is tightly integrated with the database, you can access Salesforce records and their fields directly from Apex. Every record in Salesforce is natively represented as an sObject in Apex. For example, the Acme account record corresponds to an Account sObject in Apex. The fields of the Acme record that you can view and modify in the user interface can be read and modified directly on the sObject as well.

Each Salesforce record is represented as an sObject before it is inserted into Salesforce. Likewise, when persisted records are retrieved from Salesforce, they're stored in an sObject variable.
Standard and custom object records in Salesforce map to their sObject types in Apex. Here are some common sObject type names in Apex used for standard objects.

- Account
- Contact
- Lead
- Opportunity

Create sObject Variables
To create an sObject, you need to declare a variable and assign an sObject instance to it. The data type of the variable is the sObject type.
The following example creates an sObject of type Account with the name Acme and assigns it to the acct variable.
Account acct = new Account(Name='Acme');
sObject and Field Names
Apex references standard or custom sObjects and their fields using their unique API names.
API names of object and fields can differ from their labels. For example, the Employees field has a label of Employees and appears on the account record page as Employees but its API name is NumberOfEmployees. To access this field in Apex, you'll need to use the API name for the field: NumberOfEmployees.
The following are highlights of some rules used for API names for custom objects and custom fields.
For custom objects and custom fields, the API name always ends with the __c suffix. For custom relationship fields, the API name ends with the __r suffix. For example:

- A custom object with a label of Merchandise has an API name of Merchandise__c.
- A custom field with a label of Description has an API name of Description__c.
- A custom relationship field with a label of Items has an API name of Items__r.

In addition, spaces in labels are replaced with underscores in API names. For example, a custom field name of Employee Seniority has an API name of Employee_Seniority__c.

13 **Write down the collection types in Apex and discuss the various methods of List collection with its implementation.**

In Apex, there are three primary collection types: Lists, Sets, and Maps. Each of these collection types serves different purposes and has unique characteristics.
1. Lists
Lists are ordered collections of elements that can contain duplicates. They are similar to arrays in other programming languages. Lists are indexed, meaning you can access elements by their position in the list.

Methods of List Collection
add(element): Adds an element to the end of the list.
```
List<String> fruits = new List<String>();
fruits.add('Apple');
fruits.add('Banana');
```

add(index, element): Inserts an element at the specified index.
```
fruits.add(1, 'Orange'); // Inserts 'Orange' at index 1s
```

get(index): Retrieves the element at the specified index.
```
String fruit = fruits.get(0); // Retrieves 'Apple'
```

set(index, element): Replaces the element at the specified index with a new element.

```
fruits.set(1, 'Grapes'); // Replaces 'Orange' with 'Grapes'
```

remove(index): Removes the element at the specified index.
```
fruits.remove(1); // Removes 'Grapes'
```

size(): Returns the number of elements in the list.
```
Integer size = fruits.size(); // Returns 2
```

2. Sets
Sets are unordered collections of unique elements. They do not allow duplicates and do not maintain any particular order.
3. Maps
Maps are collections of key-value pairs. Each key is unique, and each key maps to a single value. Maps are useful for storing and retrieving data based on unique keys.

14 **What are the steps to create an Apex class with methods and test your Apex Classes with an example?**

Apex is a strongly typed, object-oriented programming language that allows developers to execute flow and transaction control statements on the Salesforce platform.
Create an Apex Class
The Developer Console is where you write and test your code in Salesforce.

1. Click the **setup gear** and select **Developer Console**.
2. From the **File** menu, select **New | Apex Class**.
3. For the class name, enter OlderAccountsUtility and then click **OK**.(you can enter any name)

Syntax

```
public class OlderAccountsUtility {
}
```

Add a Method to the Class
A class usually contains one or more methods that do something useful. In this step, you'll create the updateOlderAccounts method, which gets the first five Account records ordered by the date created. It then updates the description field to say that this is a "Heritage Account," meaning accounts that are older than other accounts.

```
   public static void updateOlderAccounts() {
     // Get the 5 oldest accounts
     Account[] oldAccounts = [SELECT Id, Description FROM Account ORDER BY CreatedDate ASC
LIMIT 5];
     // loop through them and update the Description field
     for (Account acct : oldAccounts) {
        acct.Description = 'Heritage Account';
     }
     // save the change you made
     update oldAccounts;
   }
```

Click **File | Save**.
Invoke and Test the Code
Run the Code
An anonymous block is Apex code that does not get stored, but can be compiled and executed on demand right from the Developer Console.

1. In the Developer Console, select **Debug | Open Execute Anonymous Window**.
2. In the Enter Apex Code window, enter the following:
3. OlderAccountsUtility.**updateOlderAccounts**();

4. At the bottom right, click **Execute**.
   https://trailhead.salesforce.com/content/learn/projects/quickstart-apex

NOTE: You can create your own examples,methods

15 How to retrieve data from the Salesforce database according to the specified conditions and objects using SOQL Queries.

To read a record from Salesforce, you must write a query. Salesforce provides the Salesforce Object Query Language, or SOQL in short, that you can use to read saved records. SOQL is similar to the standard SQL language but is customized for the Lightning Platform.

Because Apex has direct access to Salesforce records that are stored in the database, you can embed SOQL queries in your Apex code and get results in a straightforward fashion. When SOQL is embedded in Apex, it is referred to as **inline SOQL.**
To include SOQL queries within your Apex code, wrap the SOQL statement within square brackets and assign the return value to an array of sObjects. For example, the following retrieves all account records with two fields, Name and Phone, and returns an array of Account sObjects.
Account[] accts = [SELECT Name,Phone FROM Account];

Syntax:
SELECT fields FROM ObjectName [WHERE Condition]
The WHERE clause is optional. Let's start with a very simple query. For example, the following query retrieves accounts and gets Name and Phone fields for each account.
SELECT Name,Phone FROM Account

The query has two parts:

1. SELECT Name,Phone: This part lists the fields that you would like to retrieve. The fields are specified after the SELECT keyword in a comma-delimited list. Or you can specify only one field, in which case no comma is necessary (e.g. SELECT Phone).
2. FROM Account: This part specifies the standard or custom object that you want to retrieve. In this example, it's Account. For a custom object called Invoice_Statement, it is Invoice_Statement__c.

Filter Query Results with Conditions

If you have more than one account in the org, they will all be returned. If you want to limit the accounts returned to accounts that fulfill a certain condition, you can add this condition inside the WHERE clause. The following example

retrieves only the accounts whose names are SFDC Computing. Note that comparisons on strings are case-insensitive

SELECT Name,Phone FROM Account WHERE Name='SFDC Computing'

The WHERE clause can contain multiple conditions that are grouped by using logical operators (AND, OR) and parentheses. For example, this query returns all accounts whose name is SFDC Computing that have more than 25 employees:
SELECT Name,Phone FROM Account WHERE (Name='SFDC Computing' AND NumberOfEmployees>25)

Order Query Results
When a query executes, it returns records from Salesforce in no particular order, so you can't rely on the order of records in the returned array to be the same each time the query is run. You can however choose to sort the returned record set by adding an ORDER BY clause and specifying the field by which the record set should be sorted. This example sorts all retrieved accounts based on the Name field.
SELECT Name,Phone FROM Account ORDER BY Name

The default sort order is in alphabetical order, specified as ASC. The previous statement is equivalent to

SELECT Name,Phone FROM Account ORDER BY Name ASC

To reverse the order, use the DESC keyword for descending order:

SELECT Name,Phone FROM Account ORDER BY Name DESC

Limit the Number of Records Returned

You can limit the number of records returned to an arbitrary number by adding the LIMIT n clause where n is the number of records you want returned.

Account oneAccountOnly = [SELECT Name,Phone FROM Account LIMIT 1];

Combine All Pieces Together

You can combine the optional clauses in one query, in the following order:

```
SELECT Name,Phone FROM Account
            WHERE (Name = 'SFDC Computing' AND NumberOfEmployees>25)
            ORDER BY Name
            LIMIT 10
```

https://trailhead.salesforce.com/content/learn/modules/apex_database/apex_database_soql

16 Discuss the way to search fields across multiple standard and custom object records in Salesforce using SOSL queries.

Salesforce Object Search Language (SOSL) is used for performing text searches in records. It differs from SOQL (Salesforce Object Query Language) in that it can search multiple objects simultaneously and is particularly useful for searching text, email, and phone fields.

Basic Structure of a SOSL Query

A SOSL query consists of the following main parts:

1. FIND clause: Specifies the text expression to search for.
2. IN clause: Specifies the fields to search within.
3. RETURNING clause: Specifies the objects and fields to return.

Example SOSL Query

Here's a basic SOSL query:

List<List<SObject>> searchList = [FIND 'Sample' IN ALL FIELDS RETURNING Account(Name, BillingCity), Contact(FirstName, LastName, Email), CustomObject__c(Name__c, CustomField__c)];

FIND 'Sample': Searches for the term "Sample".
IN ALL FIELDS: Searches all fields.
RETURNING: Specifies objects (Account, Contact, CustomObject__c) and their fields to return.

```
public class SOSLExample {
   public List<List<SObject>> performSearch(String searchTerm) {
        // Perform the SOSL query
        List<List<SObject>> searchResults = [FIND :searchTerm IN ALL FIELDS
                                   RETURNING Account(Name, BillingCity),
                                         Contact(FirstName, LastName, Email),
                                         CustomObject__c(Name__c, CustomField__c)];
        return searchResults;
    }
}
```

17 **Perform the following DML operations on Account Object,**

i) Insert
ii) Update
iii) Upsert
iv) Delete

DML provides a straightforward way to manage records by providing simple statements to insert, update, merge, delete, and restore records.
Because Apex is a data-focused language and is saved on the Lightning Platform, it has direct access to your data in Salesforce. Unlike other programming languages that require additional setup to connect to data sources, with Apex DML, managing records is made easy! By calling DML statements, you can quickly perform operations on your Salesforce records.
Example

```
// Create the account sObject
Account acct = new Account(Name='Acme', Phone='(415)555-1212', NumberOfEmployees=100);
// Insert the account by using DML
insert acct;
```

This example adds the Acme account to Salesforce. An account sObject is created first and then passed as an argument to the insert statement, which persists the record in Salesforce.

- Insert -The inser DML operation createa a new record. When inserting records, the system assigns an ID for each record. In addition to persisting the ID value in the database, the ID value is also auto-populated on the sObject variable that you used as an argument in the DML call.
  Example :

  ```
  // Create the account sObject
  Account acct = new Account(Name='Acme', Phone='(415)555-1212',
  NumberOfEmployees=100);
  // Insert the account by using DML
  insert acct;
  ```
- Update-Used to update an existing record.   The following example updates a contact and its related account using two updatestatements.

```
// Query for the contact, which has been associated with an account.
Contact queriedContact = [SELECT Account.Name
                  FROM Contact
```

```
                  WHERE FirstName = 'Mario' AND LastName='Ruiz'
                  LIMIT 1];
// Update the contact's phone number
queriedContact.Phone = '(415)555-1213';
// Update the related account industry
queriedContact.Account.Industry = 'Technology';
```

- upsert- The upsert DML operation creates new records and updates sObject records within a single statement, using a specified field to determine the presence of existing objects, or the ID field if no field is specified.
- Delete= You can delete persisted records using the delete statement. Deleted records aren't deleted permanently from Lightning Platform, but they're placed in the Recycle Bin for 15 days from where they can be restored.

Example : This example shows how to delete all contacts whose last name is Smith

```
Contact[] contactsDel = [SELECT Id FROM Contact WHERE LastName='Smith'];
delete contactsDel;
```

https://trailhead.salesforce.com/content/learn/modules/apex_database/apex_database_dml

18 Explain in detail about Apex Triggers and its types in Salesforce.

Apex triggers enable you to perform custom actions before or after events to records in Salesforce, such as insertions, updates, or deletions. Just like database systems support triggers, Apex provides trigger support for managing records. Typically, you use triggers to perform operations based on specific conditions, to modify related records or restrict certain operations from happening. You can use triggers to do anything you can do in Apex, including executing SOQL and DML or calling custom Apex methods.
Use triggers to perform tasks that can't be done by using the point-and-click tools in the Salesforce user interface. For example, if validating a field value or updating a field on a record, use validation rules and flows. Use Apex triggers if performance and scale is important, if your logic is too complex for the point-and-click tools, or if you're executing CPU-intensive operations.
Triggers can be defined for top-level standard objects, such as Account or Contact, custom objects, and some standard child objects. Triggers are active by default when created. Salesforce automatically fires active triggers when the specified database events occur.

Trigger Syntax

```
trigger TriggerName on ObjectName (trigger_events) {
  code_block
}
```

Trigger Events

- **before insert**
- **before update**
- **before delete**
- **after insert**
- **after update**
- **after delete**
- **after undelete**

Example

1. In the Developer Console, click File | New | Apex Trigger.
2. Enter HelloWorldTrigger for the trigger name, and then select Account for the sObject. Click Submit.

3. Replace the default code with the following.

4. trigger HelloWorldTrigger on Account (before insert) {
5.   System.debug('Hello World!');
}

6. To save, press Ctrl+S.
7. To test the trigger, create an account.
    a. Click Debug | Open Execute Anonymous Window.
    b. In the new window, add the following and then click Execute.

    c. Account a = new Account(Name='Test Trigger');
insert a;

6. In the debug log, find the Hello World! statement. The log also shows that the trigger has been executed.

Types of Triggers
There are two types of triggers.

- *Before triggers* are used to update or validate record values before they're saved to the database.
- *After triggers* are used to access field values that are set by the system (such as a record's Id or LastModifiedDate field), and to affect changes in other records. The records that fire the *after trigger* are read-only.

https://trailhead.salesforce.com/content/learn/modules/apex_triggers/apex_triggers_intro

19 **Create a Table to list and describe the various Context Variables in Apex Trigger.**

To access the records that caused the trigger to fire, use context variables. For example, Trigger.new contains all the records that were inserted in insert or update triggers. Trigger.old provides the old version of sObjects before they were updated in update triggers, or a list of deleted sObjects in delete triggers. Triggers can fire when one record is inserted, or when many records are inserted in bulk via the API or Apex. Therefore, context variables, such as Trigger.new, can contain only one record or multiple records. You can iterate over Trigger.new to get each individual sObject.
This example is a modified version of the HelloWorldTrigger example trigger. It iterates over each account in a for loop and updates the Description field for each.

```
trigger HelloWorldTrigger on Account (before insert) {
  for(Account a : Trigger.new) {
    a.Description = 'New description';
  }
}
```

Variable	Usage
`isExecuting`	Returns true if the current context for the Apex code is a trigger, not a Visualforce page, a Web service, or an `executeanonymous()` API call.
`isInsert`	Returns `true` if this trigger was fired due to an insert operation, from the Salesforce user interface, Apex, or the API.
`isUpdate`	Returns `true` if this trigger was fired due to an update operation, from the Salesforce user interface, Apex, or the API.
`isDelete`	Returns `true` if this trigger was fired due to a delete operation, from the Salesforce user interface, Apex, or the API.
`isBefore`	Returns `true` if this trigger was fired before any record was saved.
`isAfter`	Returns `true` if this trigger was fired after all records were saved.
`isUndelete`	Returns `true` if this trigger was fired after a record is recovered from the Recycle Bin. This recovery can occur after an undelete operation from the Salesforce user interface, Apex, or the API.
`new`	Returns a list of the new versions of the sObject records. This sObject list is only available in `insert`, `update`, and `undelete` triggers, and the records can only be modified in `before` triggers.
`newMap`	A map of IDs to the new versions of the sObject records. This map is only available in `before update`, `after insert`, `after update`, and `after undelete` triggers.
`old`	Returns a list of the old versions of the sObject records. This sObject list is only available in `update` and `delete` triggers.
`oldMap`	A map of IDs to the old versions of the sObject records. This map is only available in `update` and `delete` triggers.
`operationType`	Returns an enum of type System.TriggerOperation corresponding to the current operation. Possible values of the `System.TriggerOperation` enum are: `BEFORE_INSERT`, `BEFORE_UPDATE`, `BEFORE_DELETE`, `AFTER_INSERT`, `AFTER_UPDATE`, `AFTER_DELETE`, and `AFTER_UNDELETE`. If you vary your programming logic based on different trigger types, consider

	using the `switch` statement with different permutations of unique trigger execution enum states.
`size`	The total number of records in a trigger invocation, both old and new.

20 Discuss in detail about the Bulk Trigger Design Patterns.

Apex triggers are optimized to operate in bulk. When you use bulk design patterns, your triggers have better performance, consume less server resources, and are less likely to exceed platform limits.
The benefit of bulkifying your code is that bulkified code can process large numbers of records efficiently and run within governor limits on the Lightning Platform. These governor limits are in place to ensure that runaway code doesn't monopolize resources on the multitenant platform.
The following trigger (MyTriggerNotBulk) assumes that only one record caused the trigger to fire. This trigger doesn't work on a full record set when multiple records are inserted in the same transaction. A bulkified version is shown in the next example.

```
trigger MyTriggerNotBulk on Account(before insert) {
   Account a = Trigger.new[0];
   a.Description = 'New description';
}
```

This example (MyTriggerBulk) is a modified version of MyTriggerNotBulk. It uses a for loop to iterate over all available sObjects. This loop works if Trigger.new contains one sObject or many sObjects.

```
trigger MyTriggerBulk on Account(before insert) {
   for(Account a : Trigger.new) {
      a.Description = 'New description';
   }
}
```

Write the challenge as example
https://trailhead.salesforce.com/content/learn/modules/apex_triggers/apex_triggers_bulk

21 What is the purpose of writing unit tests in Apex, and how do they contribute to maintaining code quality?

A unit is the smallest logical piece of code that is possible to test. Sometimes the unit is a method, but often it is part of a method. Multiple unit tests often execute the same code but exercise different logical paths through the code. For example, one unit test might exercise the else condition of an if/else block, while another exercises the if condition.
There are three general buckets of software tests: unit, integration, end-to-end. Unit and integration tests are interested in either the functionality of the code, or the code's ability to integrate with external systems. Unit testing is more specific, focusing on the smallest possible bit of code that can be tested. This means you may have a unit test that focuses on the output of a method when certain conditions are met.

Additionally, unit tests are often automated to some degree, and they run either on a schedule or in response to an event. For instance, Salesforce runs your unit tests when you deploy from a development environment to production. Because of this, it's safer to make code changes with unit tests. If you see a unit test failing after a code change, it may mean that the code is not working as expected anymore. Unit tests are your best ally to make your code reliable and trustable.

- Ensuring that your Apex classes and triggers work as expected
- Having a suite of regression tests that can be rerun every time classes and triggers are updated to ensure that future updates you make to your app don't break existing functionality
- Meeting the code coverage requirements for deploying Apex to production or distributing Apex to customers via packages
- High-quality apps delivered to the production org, which makes production users more productive
- High-quality apps delivered to package subscribers, which increase your customers trust

Unit Test Example: Test the TemperatureConverter Class
The following simple example is of a test class with three test methods. The class method that's being tested takes a temperature in Fahrenheit as an input. It converts this temperature to Celsius and returns the converted result. Let's add the custom class and its test class.

1. In the Developer Console, click **File** | **New** | **Apex Class**, and enter TemperatureConverter for the class name, and then click **OK**.
2. Replace the default class body with the following.
3. public class TemperatureConverter {
4.   // Takes a Fahrenheit temperature and returns the Celsius equivalent.
5.   public static Decimal FahrenheitToCelsius(Decimal fh) {
6.   Decimal cs = (fh - 32) * 5/9;
7.   return cs.**setScale**(2);
8.   }

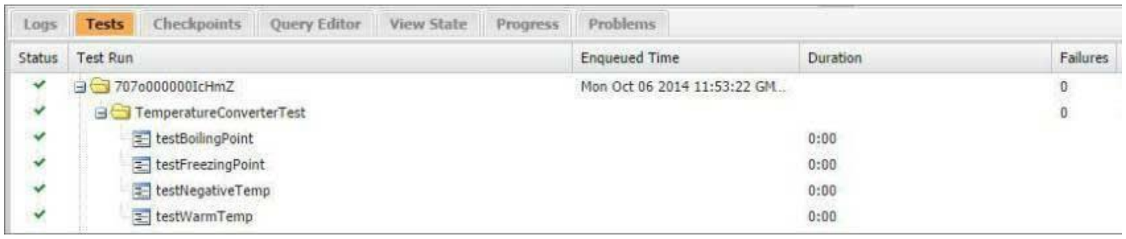
}

9. Press **Ctrl+S** to save your class.
10. Repeat the previous steps to create the TemperatureConverterTest class. Add the following for this class.
11. @isTest
12. private class TemperatureConverterTest {
13.   @isTest static void **testWarmTemp**() {
14.   Decimal celsius = TemperatureConverter.FahrenheitToCelsius(70);
15.   System.**assertEquals**(21.11,celsius);
16.   }
17.   @isTest static void **testFreezingPoint**() {
18.   Decimal celsius = TemperatureConverter.FahrenheitToCelsius(32);
19.   System.**assertEquals**(0,celsius);
20.   }
21.   @isTest static void **testBoilingPoint**() {
22.   Decimal celsius = TemperatureConverter.FahrenheitToCelsius(212);
23.   System.**assertEquals**(100,celsius,'Boiling point temperature is not expected.');
24.   }

```
    @isTest static void testNegativeTemp() {
     Decimal celsius = TemperatureConverter.FahrenheitToCelsius(-10);
     System.assertEquals(-23.33,celsius);
    }
}
```

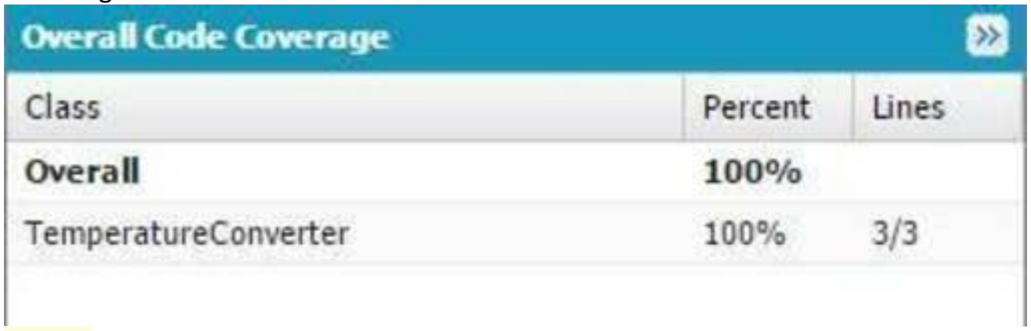
The TemperatureConverterTest test class verifies that the method works as expected by calling it with different inputs for the temperature in Fahrenheit. Each test method verifies one type of input: a warm temperature, the freezing point temperature, the boiling point temperature, and a negative temperature. The verifications are done by calling the System.assertEquals() method, which takes two parameters: the first is the expected value, and the second is the actual value. There is another version of this method that takes a third parameter—a string that describes the comparison being done, which is used in testBoilingPoint(). This optional string is logged if the assertion fails.
Let's run the methods in this class.

1. In the Developer Console, click **Test** | **New Run**.
2. Under **Test Classes**, click **TemperatureConverterTest**.
3. To add all the test methods in the TemperatureConverterTest class to the test run, click **Add Selected**.
4. Click **Run**.
5. In the Tests tab, you see the status of your tests as they're running. Expand the test run, and expand again until you see the list of individual tests that were run. They all have green checkmarks.



Status	Test Run	Enqueued Time	Duration	Failures
✓	707o000000IcHmZ	Mon Oct 06 2014 11:53:22 GM...		0
✓	TemperatureConverterTest			0
✓	testBoilingPoint		0:00	
✓	testFreezingPoint		0:00	
✓	testNegativeTemp		0:00	
✓	testWarmTemp		0:00	

After you run tests, code coverage is automatically generated for the Apex classes and triggers in the org. You can check the code coverage percentage in the Tests tab of the Developer Console. In this example, the class you've tested, the TemperatureConverter class, has 100% coverage, as shown in this image.



Overall Code Coverage

Class	Percent	Lines
Overall	**100%**	
TemperatureConverter	100%	3/3

https://trailhead.salesforce.com/content/learn/modules/apex_testing/apex_testing_intro

22 **Explain the @isTest annotation in Apex. Why is it used, and what are its key benefits?**

> The Apex testing framework enables you to write and execute tests for your Apex classes and triggers on the Lightning Platform. Apex unit tests ensure high quality for your Apex code and let you meet requirements for deploying Apex. Testing is the key to successful long-term development and is a critical component of the development process. The Apex testing framework makes it easy to test your Apex code.

Test Method Syntax

Test methods are defined using the @isTest annotation and have the following syntax:

```
@isTest static void testName() {
  // code_block
}
```

Test methods must be defined in test classes, which are classes annotated with @isTest. This sample class shows a definition of a test class with one test method.

```
@isTest
private class MyTestClass {
  @isTest static void myTest() {
    // code_block
  }
}
```

Test classes can be either private or public. If you're using a test class for unit testing only, declare it as private. Public test classes are typically used for test data factory classes, which are covered later.

23 **What are the best practices for writing effective unit tests in Apex?**

The Apex testing framework enables you to write and execute tests for your Apex classes and triggers on the Lightning Platform. Apex unit tests ensure high quality for your Apex code and let you meet requirements for deploying Apex.
Testing is the key to successful long-term development and is a critical component of the development process. The Apex testing framework makes it easy to test your Apex code. Apex code can only be written in a sandbox environment or a Developer org, not in production. Apex code can be deployed to a production org from a sandbox. Also, app developers can distribute Apex code to customers from their Developer orgs by uploading packages to the Lightning Platform AppExchange. In addition to being critical for quality assurance, Apex unit tests are also requirements for deploying and distributing Apex.
These are the benefits of Apex unit tests.

- Ensuring that your Apex classes and triggers work as expected
- Having a suite of regression tests that can be rerun every time classes and triggers are updated to ensure that future updates you make to your app don't break existing functionality

- Meeting the code coverage requirements for deploying Apex to production or distributing Apex to customers via packages
- High-quality apps delivered to the production org, which makes production users more productive
- High-quality apps delivered to package subscribers, which increase your customers trust

24 **How do you test an Apex Trigger to ensure it functions correctly in various scenarios?**

Before deploying a trigger, write unit tests to perform the actions that fire the trigger and verify expected results.

1. In the Developer Console, click **File | New | Apex Trigger**.
2. Enter AccountDeletion for the trigger name, and then select **Account** for the sObject. Click **Submit**.

```
trigger AccountDeletion on Account (before delete) {
  // Prevent the deletion of accounts if they have related opportunities.
  for(Account a : [SELECT Id FROM Account
    WHERE Id IN (SELECT AccountId FROM Opportunity) AND
    Id IN :Trigger.old]) {
    Trigger.oldMap.get(a.Id).addError('Cannot delete account with related
opportunities.');
  }
}
```

1. From Setup, search for Apex Triggers.
2. On the Apex Triggers page, click **Edit** next to the AccountDeletion trigger.
3. Select **Is Active**.
4. Click **Save**.

If your org contains triggers from a previous unit called AddRelatedRecord, CalloutTrigger, or HelloWorldTrigger, disable them. For example, to disable the AddRelatedRecordtrigger:

1. From Setup, search for Apex Triggers.
2. On the Apex Triggers page, click **Edit** next to the AddRelatedRecord trigger.
3. Deselect **Is Active**.
4. Click **Save**.

Add and Run a Unit Test

First, let's start by adding a test method. This test method verifies what the trigger is designed to do (the positive case): preventing an account from being deleted if it has related opportunities.

1. In the Developer Console, click **File | New | Apex Class**.
2. Enter TestAccountDeletion for the class name, and then click **OK**.

```
@isTest
private class TestAccountDeletion {
  @isTest
```

```
 static void TestDeleteAccountWithOneOpportunity() {
  // Test data setup
  // Create an account with an opportunity, and then try to delete it
  Account acct = new Account(Name='Test Account');
  insert acct;
  Opportunity opp = new Opportunity(
   Name=acct.Name + ' Opportunity',
   StageName='Prospecting',
   CloseDate=System.today().addMonths(1),
   AccountId=acct.Id);
  insert opp;
  // Perform test
  Test.startTest();
   Database.DeleteResult result = Database.delete(acct, false);
  Test.stopTest();
  // Verify
  // In this case the deletion should have been stopped by the trigger,
  // so verify that we got back an error.
  System.assert(!result.isSuccess());
  System.assert(result.getErrors().size() > 0);
  System.assertEquals('Cannot delete account with related opportunities.',
   result.getErrors()[0].getMessage());
 }
}
```

1. The test method first sets up a test account with an opportunity. Next, it deletes the test account, which fires the AccountDeletion trigger. The test method verifies that the trigger prevented the deletion of the test account by checking the return value of the Database.delete() call. The return value is a Database.DeleteResult object that contains information about the delete operation. The test method verifies that the deletion was not successful and verifies the error message obtained.
2. To run this test, click **Test | New Run**.
3. Under Test Classes, click **TestAccountDeletion**.
4. To add all the methods in the TestAccountDeletion class to the test run, click **Add Selected**.
5. Click **Run**. Find the test result in the Tests tab under the latest run.

The TestAccountDeletion test class contains only one test method, which tests for a single account record. Also, this test is for the positive case. Always test for more scenarios to ensure that the trigger works in all cases, including deleting an account without opportunities and bulk account deletions.
Test data is set up inside the test method, which can be time-consuming as you add more test methods.

25 **What is the importance of using Test.startTest() and Test.stopTest() methods in testing Apex Triggers?**

the startTest and stopTest methods are used to validate how close the code is to reaching governor limits.
The startTest method marks the point in your test code when your test actually begins. Each test method is allowed to call this method only once. All of the code before this method should be used to initialize variables, populate data structures, and so on, allowing you to set up everything you need to run your test. Any code that executes after the call to startTest and before stopTest is assigned a new set of governor limits.
The startTest method does not refresh the context of the test: it adds a context to your test. For example, if your class makes 98 SOQL queries before it calls startTest, and the first significant statement after startTest is a DML statement, the program can now make an additional 100 queries. Once stopTest is called, however, the program goes back into the original context, and can only make 2 additional SOQL queries before reaching the limit of 100.
The stopTest method marks the point in your test code when your test ends. Use this method in conjunction with the startTest method. Each test method is allowed to call this method only once. Any code that executes after the stopTest method is assigned the original limits that were in effect before startTest was called. All asynchronous calls made after the startTest method are collected by the system. When stopTest is executed, all asynchronous processes are run synchronously. An exception encountered during stopTest halts the synchronous processing. For example, an unhandled exception in a batch job's execute method will prevent the finish method from running in a test context.

26

How would you create test data in an Apex test class? Provide an example.

The Apex testing framework enables you to write and execute tests for your Apex classes and triggers on the Lightning Platform. Apex unit tests ensure high quality for your Apex code and let you meet requirements for deploying Apex.

Testing is the key to successful long-term development and is a critical component of the development process. The Apex testing framework makes it easy to test your Apex code. Apex code can only be written in a sandbox environment or a Developer org, not in production. Apex code can be deployed to a production org from a sandbox. Also, app developers can distribute Apex code to customers from their Developer orgs by uploading packages to the Lightning Platform AppExchange. In addition to being critical for quality assurance, Apex unit tests are also requirements for deploying and distributing Apex.

Common test utility classes are public test classes that contain reusable code for test data creation.

Public test utility classes are defined with the IsTest annotation, and as such, are excluded from the organization code size limit and execute in test context. They can be called by test methods but not by non-test code.

The methods in the public test utility class are defined the same way methods are in non-test classes. They can take parameters and can return a value. The methods must be declared as public or global to be visible to other test classes. These common methods can be called by any test method in your Apex classes to set up test data before running the test. While you can create public methods for test data creation in a regular Apex class, without the IsTest annotation, you don't get the benefit of excluding this code from the organization code size limit.

This is an example of a test utility class. It contains one method, createTestRecords, which accepts the number of accounts to create and the number of contacts per account. The next example shows a test method that calls this method to create some data.

```
@IsTest

public class TestDataFactory {

    public static void createTestRecords(Integer numAccts, Integer
numContactsPerAcct) {

        List<Account> accts = new List<Account>();



        for(Integer i=0;i<numAccts;i++) {

            Account a = new Account(Name='TestAccount' + i);

            accts.add(a);

        }

        insert accts;



        List<Contact> cons = new List<Contact>();
}

        for (Integer j=0;j<numAccts;j++) {

            Account acct = accts[j];

            // For each account just inserted, add contacts

            for (Integer k=numContactsPerAcct*j;k<numContactsPerAcct*(j+1);k++) {

                cons.add(new Contact(firstname='Test'+k,

                                     lastname='Test'+k,

                                     AccountId=acct.Id));

            }

        }

        // Insert all contacts for all accounts

        insert cons;

    }

}
```

The test method in this class calls the test utility method, createTestRecords, to create five test accounts with three contacts each.

```
@IsTest

private class MyTestClass {

    static testmethod void test1() {

        TestDataFactory.createTestRecords(5,3);

        // Run some tests

    }

}
```

What are the differences between using Test.loadData() and creating test data programmatically in an Apex test class?

Test.loadData()

Test.loadData() allows you to load test data from a static resource, typically in CSV format, directly into your test environment.

Pros:

- Reusability: You can reuse the same data across multiple test classes by loading it from a single static resource.
- Readability: The data is stored in a human-readable format (CSV), making it easier to understand and modify.
- Consistency: Ensures consistent test data across different test runs and environments.

Cons:

- Static Nature: The data is static and may not cover dynamic test cases that require data to be generated on the fly.
- Setup Overhead: Requires setting up and maintaining static resource files, which can be cumbersome for large datasets.
- Limited Complexity: Not ideal for scenarios requiring complex data relationships or dependencies.

```
@isTest
public class MyTestClass {
  @isTest static void testWithDataLoad() {
    // Load test data from a static resource
    List<MyObject__c> testData = (List<MyObject__c>) Test.loadData(MyObject__c.class,
'myTestData');

    // Perform test operations
    Test.startTest();
    // Your test logic here
    Test.stopTest();

    // Assertions
    System.assertEquals(expectedValue, actualValue);
  }
}
```

Creating Test Data Programmatically

This approach involves creating test data directly in your test methods using Apex code.

Pros:

- Flexibility: Allows for dynamic creation of test data to cover a wide range of scenarios and edge cases.
- Complex Data Relationships: Ideal for creating complex data structures and relationships programmatically.
- No Setup Required: Does not require setting up static resources, making it faster to write and execute tests.

Cons:

- Code Maintenance: Can lead to repetitive code if similar test data is created across multiple tests.
- Readability: Harder to visualize the test data as it is embedded within the code, making it less readable.
- Potential for Inconsistencies: Slight variations in data creation code can lead to inconsistencies in test results.

```
@isTest
public class MyTestClass {
  @isTest static void testWithProgrammaticData() {
    // Create test data programmatically
    MyObject__c testData = new MyObject__c(
      Name = 'Test Name',
      Field1__c = 'Value1',
      Field2__c = 'Value2'
    );
    insert testData;

    // Perform test operations
    Test.startTest();
    // Your test logic here
    Test.stopTest();

    // Assertions
    System.assertEquals(expectedValue, testData.SomeField__c);
  }
}
```

28 **Write a test method for the following Apex Trigger that updates the LastName field of a Contact record to "Updated" when the FirstName is "Test":**

```
 trigger ContactTrigger on Contact (before insert,
before update)
{
for (Contact c : Trigger.new) {
if (c.FirstName == 'Test') {
c.LastName = 'Updated';
}
}
}
```

Test the trigger for both insert and update operations.

```
@isTest
private class ContactTriggerTest {
  @isTest
```

```
    static void testContactTrigger() {
        // Test insert operation
        Contact contact1 = new Contact(FirstName = 'Test', LastName =
'OriginalLastName', Email = 'test1@example.com');
        insert contact1;

        // Retrieve the inserted contact from the database
        contact1 = [SELECT FirstName, LastName FROM Contact WHERE Id = :contact1.Id];
        System.assertEquals('Updated', contact1.LastName, 'The LastName should be
updated to "Updated"');

        // Test update operation
        Contact contact2 = new Contact(FirstName = 'AnotherName', LastName =
'OriginalLastName', Email = 'test2@example.com');
        insert contact2;

        // Update the FirstName to 'Test'
        contact2.FirstName = 'Test';
        update contact2;

        // Retrieve the updated contact from the database
        contact2 = [SELECT FirstName, LastName FROM Contact WHERE Id = :contact2.Id];
        System.assertEquals('Updated', contact2.LastName, 'The LastName should be
updated to "Updated"');
    }
}
```

29

Create a test class for the following Apex class that converts a list of strings to uppercase:

```
  public class StringUtils {
 public List<String> convertToUpper(List<String>
 inputList) {
 for (Integer i = 0; i < inputList.size(); i++) {
 inputList[i] = inputList[i].toUpperCase();
 }
 return inputList;
 }
 }
```

Write all tests that handle empty lists and null values.

```
@isTest
```

```
private class StringUtilsTest {
   @isTest
   static void testConvertToUpperWithValidInput() {
      // Create an instance of StringUtils
      StringUtils utils = new StringUtils();

      // Define a list of strings
      List<String> inputList = new List<String>{'hello', 'world', 'salesforce'};

      // Call the convertToUpper method
      List<String> result = utils.convertToUpper(inputList);

      // Verify the result
      System.assertEquals('HELLO', result[0], 'The first element should be converted to
uppercase');
      System.assertEquals('WORLD', result[1], 'The second element should be converted
to uppercase');
      System.assertEquals('SALESFORCE', result[2], 'The third element should be
converted to uppercase');
   }

   @isTest
   static void testConvertToUpperWithEmptyList() {
      // Create an instance of StringUtils
      StringUtils utils = new StringUtils();

      // Define an empty list of strings
      List<String> inputList = new List<String>();

      // Call the convertToUpper method
      List<String> result = utils.convertToUpper(inputList);

      // Verify the result
      System.assertEquals(0, result.size(), 'The result should be an empty list');
   }

   @isTest
   static void testConvertToUpperWithNullInput() {
      // Create an instance of StringUtils
      StringUtils utils = new StringUtils();

      // Define a null list
      List<String> inputList = null;

      try {
         // Call the convertToUpper method
```

```
            List<String> result = utils.convertToUpper(inputList);
            // Verify the result
            System.assert(false, 'The method should throw an exception when input is null');
        } catch (NullPointerException e) {
            // Expected exception
            System.assert(true, 'The method should throw a NullPointerException');
        }
    }
}
```

30 **Write a test class for the following Apex class that assigns the highest-priority case to a User:**

```
public class CaseAssignment {
public static Case assignHighPriorityCase(User user) {
List<Case> highPriorityCases = [SELECT Id, Priority FROM Case WHERE Priority =
'High' LIMIT 1];
if (!highPriorityCases.isEmpty()) {
highPriorityCases[0].OwnerId = user.Id;
update highPriorityCases[0];
return highPriorityCases[0];
}
return null;
}
}
```

Create test data with various priority levels to ensure comprehensive testing.

```
@isTest
private class CaseAssignmentTest {
    @isTest
    static void testAssignHighPriorityCase() {
        // Create test data with various priority levels
        Case lowPriorityCase = new Case(Subject = 'Low Priority Case', Priority = 'Low');
        Case mediumPriorityCase = new Case(Subject = 'Medium Priority Case', Priority =
'Medium');
        Case highPriorityCase = new Case(Subject = 'High Priority Case', Priority = 'High');

        insert new List<Case>{lowPriorityCase, mediumPriorityCase, highPriorityCase};

        // Create a test user
        User testUser = [SELECT Id FROM User WHERE Username = 'test@example.com'
LIMIT 1];

        // Call the assignHighPriorityCase method
        Case assignedCase = CaseAssignment.assignHighPriorityCase(testUser);
        // Verify the result
        System.assertNotEquals(null, assignedCase, 'A high priority case should be
assigned');
        System.assertEquals('High', assignedCase.Priority, 'The assigned case should have
high priority');
        System.assertEquals(testUser.Id, assignedCase.OwnerId, 'The OwnerId should be
updated to the test user');
    }

    @isTest
    static void testNoHighPriorityCase() {
        // Create test data without high priority cases
        Case lowPriorityCase = new Case(Subject = 'Low Priority Case', Priority = 'Low');
        Case mediumPriorityCase = new Case(Subject = 'Medium Priority Case', Priority =
'Medium');

        insert new List<Case>{lowPriorityCase, mediumPriorityCase};

        // Create a test user
        User testUser = [SELECT Id FROM User WHERE Username = 'test@example.com'
LIMIT 1];

        // Call the assignHighPriorityCase method
        Case assignedCase = CaseAssignment.assignHighPriorityCase(testUser);

        // Verify the result
        System.assertEquals(null, assignedCase, 'No high priority case should be assigned');
    }
}
```

31 **Why do we use Lightning Web Components in Salesforce? Write a program that displays Hello World in an input field.**

Modern browsers are based on web standards, and evolving standards are constantly improving what browsers can present to a user. Lightning Web Components uses core Web Components standards and provides only what's necessary to perform well in browsers supported by Salesforce. Because it's built on code that runs natively in browsers, Lightning Web Components is lightweight and delivers exceptional performance. Most of the code you write is standard JavaScript and HTML.

You'll find it easier to:

- Find solutions in common places on the web.
- Find developers with necessary skills and experience.
- Use other developers' experiences (even on other platforms).
- Develop faster.

- Utilize full encapsulation so components are more versatile.

Lightning web component creation is one, two, three steps. You create (1) a JavaScript file, (2) an HTML file, and optionally (3) a CSS file.

- HTML provides the structure for your component.
- JavaScript defines the core business logic and event handling.
- CSS provides the look, feel, and animation for your component.

Example:

Sample.html

```
<template>
  <input value={message}></input>
</template>
```

Sample.js

```
import { LightningElement } from 'lwc';
export default class App extends LightningElement {
  message = 'Hello World';
}
```

Sample.css

```
input {
  color: blue;
}
```

<u>Deploy Your Files</u>

Authenticate with your org using SFDX: Authorize an Org from the Command Palette in VS Code. When prompted, accept the Project Default and press Enter to accept the default alias. If prompted to allow access, click Allow.

Right-click on the force-app/main/default folder and select SFDX: Deploy This Source to Org.

32 **Explain in detail about LWC Module, Lifecycle Hooks and Decorators. Support with a program.**

Lightning Web Components (LWC)

LWC Modules Modules in LWC are JavaScript files that export a class, function, or variable. Each module is a reusable piece of code encapsulated within its own scope. In LWC, each component's JavaScript file is a module.

```
// greeting.js
const greeting = 'Hello, LWC!';
export { greeting };
```

Lifecycle Hooks

Lifecycle hooks in LWC are callback methods triggered at specific points during a component's lifecycle, allowing developers to execute code at these points.

Major Lifecycle Hooks:

1. constructor()
2. connectedCallback()
3. renderedCallback()
4. disconnectedCallback()
5. errorCallback()

Example of Lifecycle Hooks:

```
import { LightningElement } from 'lwc';

export default class LifecycleExample extends LightningElement {
  constructor() {
    super();
    console.log('Constructor called');
  }

  connectedCallback() {
    console.log('Connected Callback called');
  }

  renderedCallback() {
    console.log('Rendered Callback called');
  }

  disconnectedCallback() {
    console.log('Disconnected Callback called');
  }

  errorCallback(error, stack) {
    console.error('Error Callback called', error, stack);
  }
}
```

Decorators

Decorators in LWC are used to add functionality to properties and methods in a class. The three main decorators are @api, @track, and @wire.

1. @api: Exposes a public property or method, making it accessible to the parent component.
2. @track: Tracks changes to the properties of an object or array, re-rendering the component if changes are detected.
3. @wire: Connects a property or method to a Salesforce data source.

```
import { LightningElement, api, track, wire } from 'lwc';
import getContacts from
'@salesforce/apex/ContactController.getContacts';

export default class DecoratorExample extends LightningElement {
  @api greeting = 'Hello';
```

```
    @track contacts;

    @wire(getContacts)
    wiredContacts({ error, data }) {
      if (data) {
        this.contacts = data;
      } else if (error) {
        console.error('Error fetching contacts', error);
      }
    }
  }
```

33 Write the steps involved in Deploying Lightning Web Component Files.

Lightning Web Components uses core Web Components standards and provides only what's necessary to perform well in browsers supported by Salesforce. Because it's built on code that runs natively in browsers, Lightning Web Components is lightweight and delivers exceptional performance.

Create Lightning Web Components

we develop a Lightning web component using Visual Studio Code with the Salesforce extension.

Steps:

1. Create a project by selecting **SFDX: Create Project** from the Command Palette in VS Code. Accept the standard template and give it the project name
2. Under **force-app/main/default**, right-click the **lwc** folder and select **SFDX: Create Lightning Web Component**.
3. Enter the name of the new component. Eg : App
4. Press **Enter** and then press **Enter** again to accept the default `force-app/main/default/lwc`
5. Enter the code in html and JS file.Eg:App.html.App.js

Displaying a Component in an Org:

You have two options for displaying a Lightning web component in the UI.

1. Set the component to support various flexipage types (home, record home, and so on) then add it to a flexipage using the Lightning App Builder. This is the simplest approach and the one you follow in this unit.

2. You can also create a tab which points to an Aura component containing your Lightning web component. You can see the required pieces in the repo.

- Wrapper Components
- Tabs
- Visibility Settings
- Default application configuration file

Deploy Your Files
Authenticate with your org using SFDX: Authorize an Org from the Command Palette in VS Code.
When prompted, accept the Project Default and press Enter to accept the default alias. If prompted to allow access, click Allow.
Right-click on the force-app/main/default folder and select SFDX: Deploy This Source to Org.
Enter Remote Sites

1. To open your org, use **SFDX: Open Default Org** from the Command Palette in VS Code.
2. In Setup, enter trusted urls in the Quick Find box and then select **Trusted URLs**.
3. Click **New Trusted URL**.
4. Enter API Name
5. Enter URL.
6. Enter Description
7. Click **Save**.

Create a New Page for Your Component
To open your org, use SFDX: Open Default Org from the Command Palette in VS Code.
In Setup, enter Lightning App Builder in the Quick Find box and then select Lightning App Builder.
Click New.
Select App Page and click Next.
Give it the label and click Next.
Select One Region and click Done.
In Lightning App Builder, scroll down the Components list until you see your component.

34 Explain Handle Events in Lightning Web Components.

Handling events in Lightning Web Components (LWC) is a core aspect of creating interactive and dynamic user interfaces. Events allow components to communicate with each other. Here's a detailed explanation of how to handle events in LWC:
Basic Event Handling

1. Listening for Events: You add event listeners to your HTML template to handle events like clicks, input changes, etc.
2. Dispatching Events: You can create and dispatch custom events to communicate from child to parent components.

```
<template>
```

```
  <button onclick={handleClick}>Click Me!</button>
</template>

import { LightningElement } from 'lwc';

export default class EventExample extends LightningElement {
  handleClick() {
    console.log('Button clicked!');
  }
}
```

In this example:

- Listening: The onclick attribute in the HTML template listens for a click event and triggers the handleClick method in the JavaScript file.

1. Child Component:
   - HTML: Contains a button that, when clicked, triggers the handleButtonClick method.
   - JavaScript: The handleButtonClick method creates and dispatches a custom event called notify with a message in the detail property.
2. Parent Component:
   - HTML: Includes the child component and listens for the notify event using onnotify.
   - JavaScript: The handleNotification method receives the event and logs the message from the child component.

Event Properties

- type: The name of the event.
- bubbles: Boolean value indicating if the event propagates upwards through the DOM (default is false).
- composed: Boolean value indicating if the event propagates across the shadow DOM boundary (default is false).
- detail: An object to pass additional data with the event.

35 **Briefly write about Lightning Design System Styles with an example.**

The Lightning Design System (LDS) provides a set of CSS styles and guidelines to ensure a consistent look and feel across Salesforce applications. It is designed to help developers create applications that align with Salesforce's UI principles and best practices.

Key Features of Lightning Design System

- Predefined Classes: Provides a comprehensive set of classes to style components effortlessly.
- Responsive Design: Ensures that components work seamlessly across different devices and screen sizes.
- Accessibility: Follows best practices to create accessible and user-friendly interfaces.
- Consistency: Promotes a unified experience across all Salesforce apps.

Example: Using LDS Styles in LWC
HTML File (exampleLDSComponent.html):

```
<template>
  <div class="slds-box slds-theme_default">
    <h1 class="slds-text-heading_large">Hello, Lightning Design System!</h1>
    <button class="slds-button slds-button_brand">Click Me</button>
  </div>
</template>
```

CSS File (exampleLDSComponent.css):

```
/* Custom styles for your component can be added here */
```

36 **Briefly write about Apex Security and Sharing. Explain the importance of sharing rules.**

Apex Security
Apex security ensures that your code adheres to the security settings configured in Salesforce. It encompasses:

- CRUD Operations: Your code should check for Create, Read, Update, and Delete (CRUD) permissions to ensure users have appropriate access to objects.
- Field-Level Security (FLS): Respect the field visibility settings defined in field-level security.
- With Sharing/Without Sharing: Apex classes can enforce or bypass sharing rules using the with sharing or without sharing keywords, respectively.

Sharing Rules
Sharing rules are used to grant additional access to records beyond the organization-wide defaults (OWD). These rules are based on criteria or ownership, and they help manage record-level access control.

Importance of Sharing Rules in Salesforce

- **Granular Access Control**: Sharing rules provide precise control over who can view or edit records, enhancing security.
- **Enhanced Collaboration**: By sharing relevant data with the right users or groups, sharing rules facilitate better teamwork and data transparency.
- **Security Compliance**: Ensures only authorized users access sensitive data, maintaining compliance with internal policies and regulatory requirements.

37 **Define how SOQL differs from SQL.**

Feature	SQL	SOQL
Purpose	General-purpose query language for relational databases	Query language specifically for Salesforce data
Data Model	Works with tables and rows in relational databases	Works with Salesforce objects and fields
Syntax	Rich syntax supporting complex joins, subqueries, and multiple operations	Simpler syntax with limited support for joins and subqueries
Joins	Supports complex joins (INNER JOIN, LEFT JOIN, etc.)	Limited to semi-joins using relationship queries (parent-child, child-parent)
Usage	Used in various database management systems (MySQL, PostgreSQL, Oracle, etc.)	Used within Salesforce (Apex, Visualforce, Lightning, etc.)
Aggregation	Supports aggregate functions (SUM, AVG, COUNT, etc.)	Supports aggregate functions (COUNT, SUM, AVG, etc.)
Commands	Includes a wide range of commands (SELECT, INSERT, UPDATE, DELETE, CREATE, ALTER, DROP)	Primarily focuses on data retrieval with SELECT; no commands for data manipulation or schema changes
Filtering	Uses `WHERE` clause for filtering records	Uses `WHERE` clause for filtering records
Ordering	Uses `ORDER BY` clause to sort query results	Uses `ORDER BY` clause to sort query results

Limit Results	Uses `LIMIT` clause to restrict the number of results	Uses `LIMIT` clause to restrict the number of results

38 **Define a Cross-Site Request Forgery (CSFR) vulnerability.**

CSRF is a common web application vulnerability where a malicious application causes a user's client to perform an unwanted action on a trusted site for which the user is currently authenticated. Cross-Site Request Forgery (CSRF) flaws are less a programming mistake and more a lack of a defense. For example, an attacker has a web page at www.attacker.com that could be any web page, including one that provides valuable services or information that drives traffic to that site. Somewhere on the attacker's page is an HTML tag that looks like this: <img src="http://www.yourwebpage.com/yourapplication/createuser?email=attacker@attacker.com&type=admin....." height=1 width=1 />
In other words, the attacker's page contains a URL that performs an action on your website. If the user is still logged into your web page when they visit the attacker's web page, the URL is retrieved and the actions performed. This attack succeeds because the user is still authenticated to your web page. This attack is a simple example, and the attacker can get more creative by using scripts to generate the callback request or even use CSRF attacks against your AJAX methods.
Within the Lightning Platform, Salesforce implemented an anti-CSRF token to prevent such an attack. Every page includes a random string of characters as a hidden form field. Upon the next page load, the application checks the validity of this string of characters and doesn't execute the command unless the value matches the expected value. This feature protects you when using all of the standard controllers and methods.

39 **Define a Server-side request forgery vulnerability.**

A Server-Side Request Forgery (SSRF) vulnerability is a security flaw that allows an attacker to trick a server into making unauthorized requests to unintended locations, either internally within the server's network or externally to third-party services. Essentially, the attacker exploits the server to initiate requests on their behalf.

In an SSRF attack, the attacker manipulates the server to send a crafted request to a target that the attacker controls or wants to gain information from. This can lead to several harmful outcomes:

- **Data Exfiltration**: The server may leak sensitive data from internal services that are otherwise inaccessible from the outside.
- **Scanning and Reconnaissance**: Attackers can use the server to scan internal network resources to gather information for further attacks.
- **Unauthorized Actions**: The server might perform unintended actions on internal or external services, potentially leading to data corruption or service disruptions.

Example Scenario

Imagine a Salesforce application that fetches information about students from an internal service hosted on a non-routable address. The application is designed to make

a GET request to the server whose address is contained in the API request to retrieve student details, for example, studentApi=https://192.168.0.1/student.
An attacker observes that the requests sent from the client contain a path value that may be exploitable. They intercept the API call from the client, replace the endpoint value of the student service with a call to the metadata service (e.g., http://169.254.169.254), and exfiltrate sensitive service configuration data from the internal metadata endpoint.

Mitigation Strategies
To mitigate SSRF vulnerabilities, Salesforce provides built-in protections and developers should follow best practices such as:

- **Validate and Sanitize Inputs**: Ensure that input values are properly validated and sanitized to prevent the injection of malicious URLs.
- **Implement Allowlists**: Restrict the allowed destinations for outgoing requests by enforcing URL schema, port, and destination allowlists.
- **Use URL Parsing Libraries**: Utilize URL parsing libraries to parse and validate URLs before making requests.
- **Network Segmentation**: Implement network segmentation to restrict the server's ability to make requests to internal resources.

40 What is typecasting?

Apex enables *casting*, that is, a data type of one class can be assigned to a data type of another class, but only if one class is a subclass of the other class. Use casting when you want to convert an object from one data type to another.

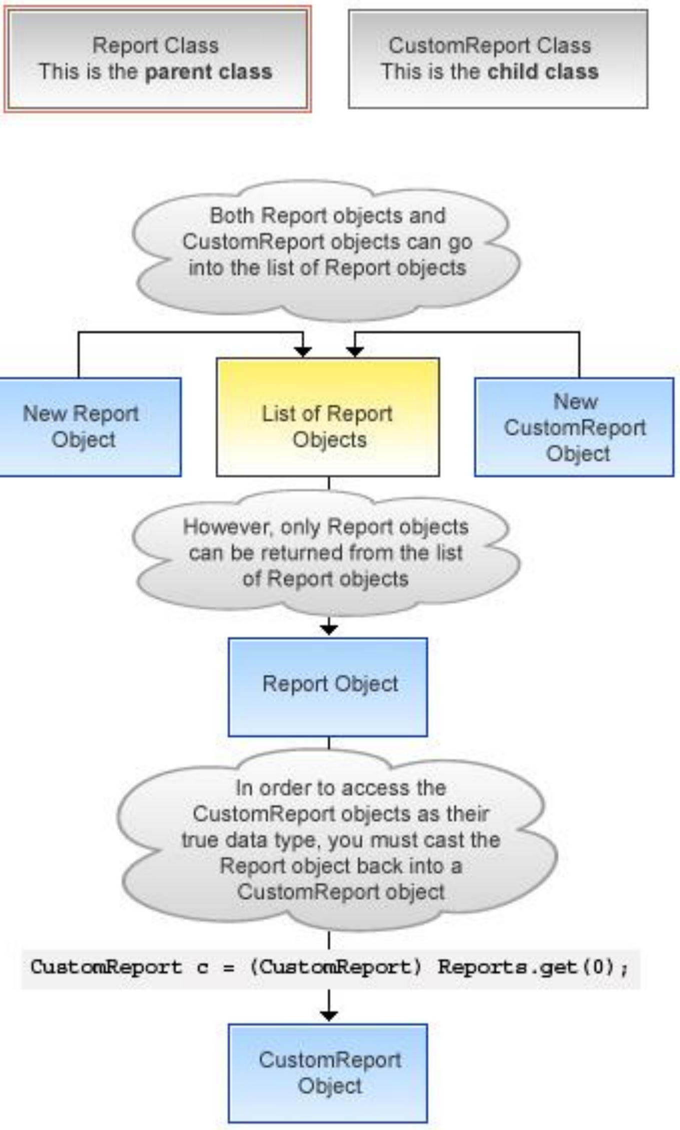
In the following example, `CustomReport` extends the class `Report`. Therefore, it is a subclass of that class. This means that you can use casting to assign objects with the parent data type (`Report`) to the objects of the subclass data type (`CustomReport`).

```
public virtual class Report {
}

public class CustomReport extends Report {
}
```

In the following code segment, a custom report object is first added to a list of report objects. Then the custom report object is returned as a report object, which is then cast back into a custom report object.

```
...
 // Create a list of report objects
 Report[] Reports = new Report[5];

 // Create a custom report object
 CustomReport a = new CustomReport();
```

```
// Because the custom report is a sub class of the Report class,
// you can add the custom report object a to the list of report objects
Reports.add(a);

// The following is not legal:
// CustomReport c = Reports.get(0);
// because the compiler does not know that what you are
// returning is a custom report.

// You must use cast to tell it that you know what
// type you are returning. Instead, get the first item in the list
// by casting it back to a custom report object
CustomReport c = (CustomReport) Reports.get(0);
...
```


Report Class
This is the **parent class**
CustomReport Class
This is the **child class**
Both Report objects and CustomReport objects can go into the list of Report objects
New Report Object
List of Report Objects
New CustomReport Object
However, only Report objects can be returned from the list of Report objects
Report Object
In order to access the CustomReport objects as their true data type, you must cast the Report object back into a CustomReport object
CustomReport c = (CustomReport) Reports.get(0);
CustomReport Object

41 What is Apex Callout?

An Apex callout enables you to tightly integrate your Apex code with an external service. The callout makes a call to an external web service or sends an HTTP request from Apex code, and then receives the response.
Apex callouts come in two flavors.

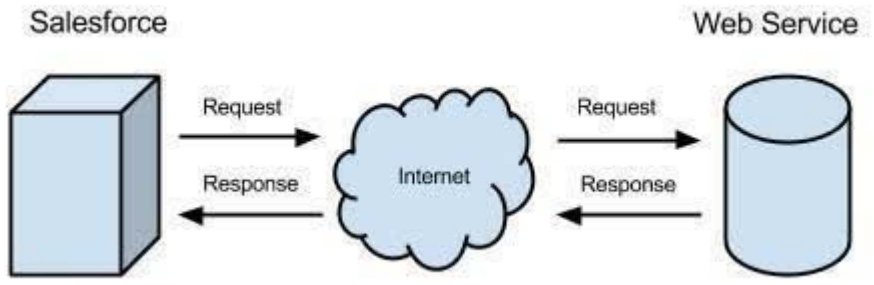
- Web service callouts to SOAP web services use XML, and typically require a WSDL document for code generation.
- HTTP callouts to services typically use REST with JSON.

These two types of callouts are similar in terms of sending a request to a service and receiving a response. But while WSDL-based callouts apply to SOAP Web services, HTTP callouts can be used with any HTTP service, either SOAP or REST.
Before any Apex callout can call an external site, that site must be registered in the Remote Site Settings page, or the callout fails. Salesforce prevents calls to unauthorized network addresses.
If the callout specifies a named credential as the endpoint, you don't need to configure remote site settings. A named credential specifies the URL of a callout endpoint and its required authentication parameters in one definition.
For SOAP-based integration, Apex allows developers to generate classes from WSDL files, providing a structured approach to interact with web services following the Simple Object Access Protocol.
On the other hand, for HTTP services adhering to REST principles, developers can employ Apex to make HTTP requests, utilizing standard methods like GET, POST, PUT, and DELETE to communicate with web service endpoints.

- **GET :** Used to retrieve resources from a server. It is a safe method as it does not change the state of the resource in any way
- **POST :** Used to submit an entity to the specified resource, often causing a change in state or side effects on the server.
- **PUT :** Used to update the existing resource on the server and it updates the full resource. If the resource does not exist, PUT may decide to create a new resource
- **DELETE :** Used to delete the resources from a server. It deletes the resource identified by the Request-URI

42 With the help of a diagram briefly explain about REST callouts

REST callouts are based on HTTP. Each callout request is associated with an HTTP method and an endpoint. The HTTP method indicates what type of action is desired.



The simplest request is a GET request (GET is an HTTP method). A GET request means that the sender wants to obtain information about a resource from the server. When the server receives and processes this request, it returns the request information to the recipient. A GET request is similar to navigating to an address in the browser. When you visit a web page, the browser performs a GET request behind the scenes. In the browser, the result of the navigation is a new HTML page that's displayed. With a callout, the result is the response object.

Table 1. Some Common HTTP Methods

HTTP Method	Description
GET	Retrieve data identified by a URL.
POST	Create a resource or post data to the server.
DELETE	Delete a resource identified by a URL.
PUT	Create or replace the resource sent in the request body.

In addition to the HTTP method, each request sets a URI, which is the endpoint address at which the service is located. For example, an endpoint can be http://www.example.com/api/resource. In the example in the "HTTP and Callout Basics" unit, the endpoint is https://th-apex-http-callout.herokuapp.com/animals.
When the server processes the request, it sends a status code in the response. The status code indicates whether the request was processed successfully or whether errors were encountered. If the request is successful, the server sends a status code of 200. You've probably seen some other status codes, such as 404 for file not found or 500 for an internal server error.

Examples:

```
public class AnimalsCallouts {
  public static HttpResponse makeGetCallout() {
    Http http = new Http();
    HttpRequest request = new HttpRequest();
    request.setEndpoint('https://th-apex-http-callout.herokuapp.com/animals');
    request.setMethod('GET');
    HttpResponse response = http.send(request);
    // If the request is successful, parse the JSON response.
```

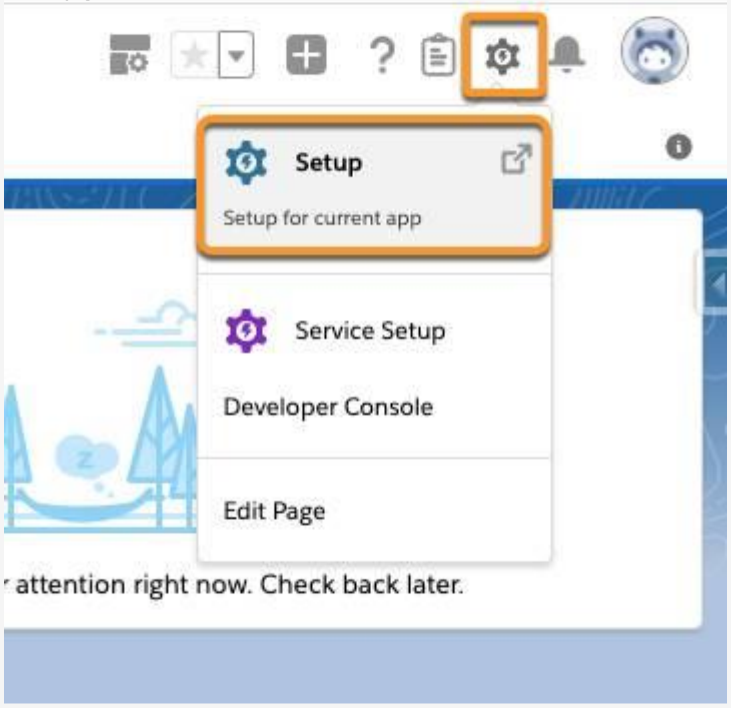
```
            if(response.getStatusCode() == 200) {
                // Deserializes the JSON string into collections of primitive data types.
                Map<String, Object> results = (Map<String, Object>)
    JSON.deserializeUntyped(response.getBody());
                // Cast the values in the 'animals' key as a list
                List<Object> animals = (List<Object>) results.get('animals');
                System.debug('Received the following animals:');
                for(Object animal: animals) {
                    System.debug(animal);
                }
            }
            return response;
        }
        public static HttpResponse makePostCallout() {
            Http http = new Http();
            HttpRequest request = new HttpRequest();
            request.setEndpoint('https://th-apex-http-callout.herokuapp.com/animals');
            request.setMethod('POST');
            request.setHeader('Content-Type', 'application/json;charset=UTF-8');
            request.setBody('{"name":"mighty moose"}');
            HttpResponse response = http.send(request);
            // Parse the JSON response
            if(response.getStatusCode() != 201) {
                System.debug('The status code returned was not expected: ' +
                    response.getStatusCode() + ' ' + response.getStatus());
            } else {
                System.debug(response.getBody());
            }
            return response;
        }
    }
```

43 **Create an Apex REST class that contains methods for each HTTP method.**

```
@RestResource(urlMapping='/exampleEndpoint/*')
global with sharing class ExampleRestResource {

    @HttpGet
    global static String doGet() {
        RestRequest req = RestContext.request;
        String recordId = req.requestURI.substring(req.requestURI.lastIndexOf('/')+1);
        if (String.isBlank(recordId)) {
            return 'No ID provided';
        }
        // Fetch record logic here
        return 'Record details for ID: ' + recordId;
    }

    @HttpPost
    global static String doPost(String name, String description) {
        RestRequest req = RestContext.request;
        // Create new record logic here using name and description
        return 'Record created with name: ' + name + ' and description: ' + description;
    }

    @HttpPut
    global static String doPut(String recordId, String newName) {
        RestRequest req = RestContext.request;
        // Update record logic here using recordId and newName
        return 'Record with ID: ' + recordId + ' updated to new name: ' + newName;
    }

    @HttpDelete
    global static String doDelete() {
        RestRequest req = RestContext.request;
        String recordId = req.requestURI.substring(req.requestURI.lastIndexOf('/')+1);
        if (String.isBlank(recordId)) {
            return 'No ID provided';
        }
        // Delete record logic here
        return 'Record with ID: ' + recordId + ' deleted';
    }
}
```

44 **Write the steps to create Travel Approval Lightning App, add tabs and customize page layouts.**

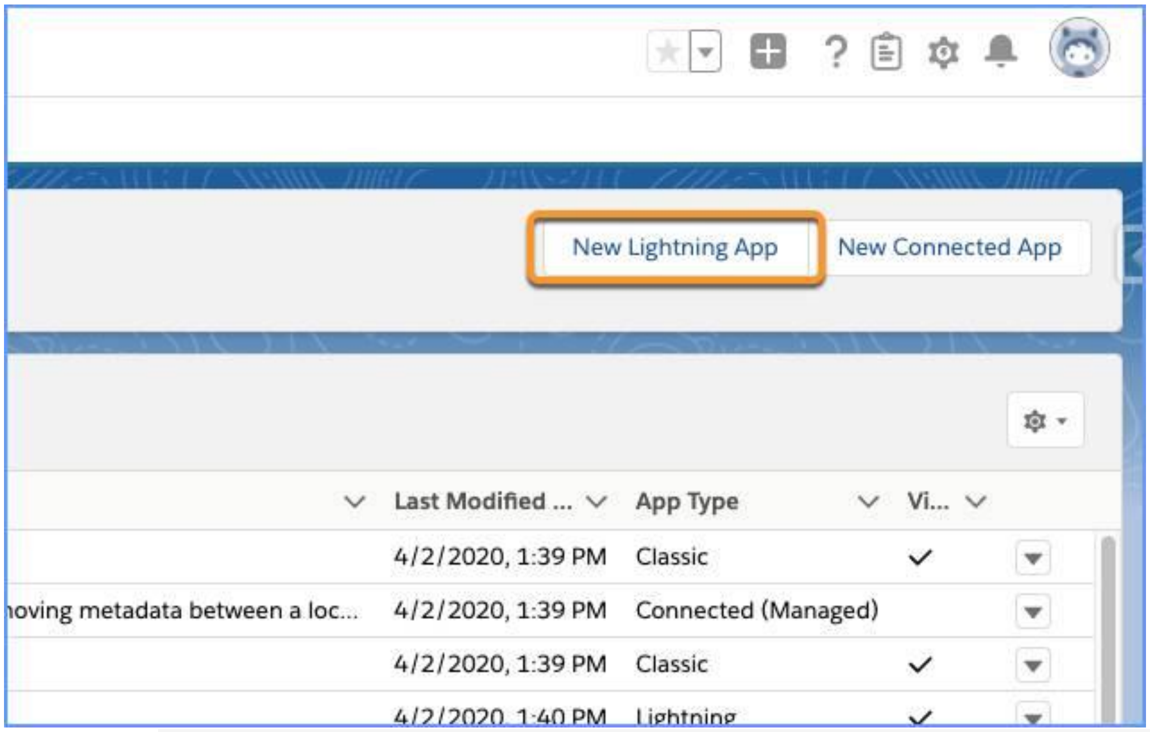
your Trailhead Playground, click and then



select **Setup**.

5. In the Quick Find box, enter `App Manager` and select **App Manager**.
6. Click **New Lightning App**.
7. In the App Details & Branding window, enter these details.
    - For App Name, enter `Travel App`.

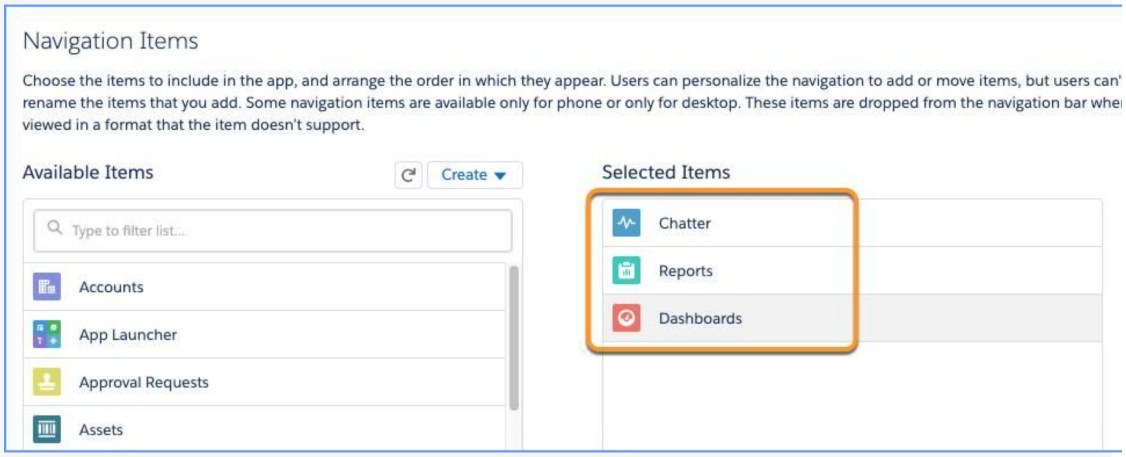
- For the Image, click **Upload** and select **travel.png** from the files you



	Last Modified ...	App Type	Vi...
	4/2/2020, 1:39 PM	Classic	✓
noving metadata between a loc...	4/2/2020, 1:39 PM	Connected (Managed)	
	4/2/2020, 1:39 PM	Classic	✓
	4/2/2020, 1:40 PM	Lightning	✓

downloaded above.

8. Click **Next**.
9. On the App Options screen, select **Standard navigation** and click **Next**.
10. On the Utility Items screen, click **Next**
11. On the Navigation Items screen, select **Chatter, Reports,** and **Dashboards** from the Available Items list, and move them to the Selected Items list using the arrow. Then click **Next**.

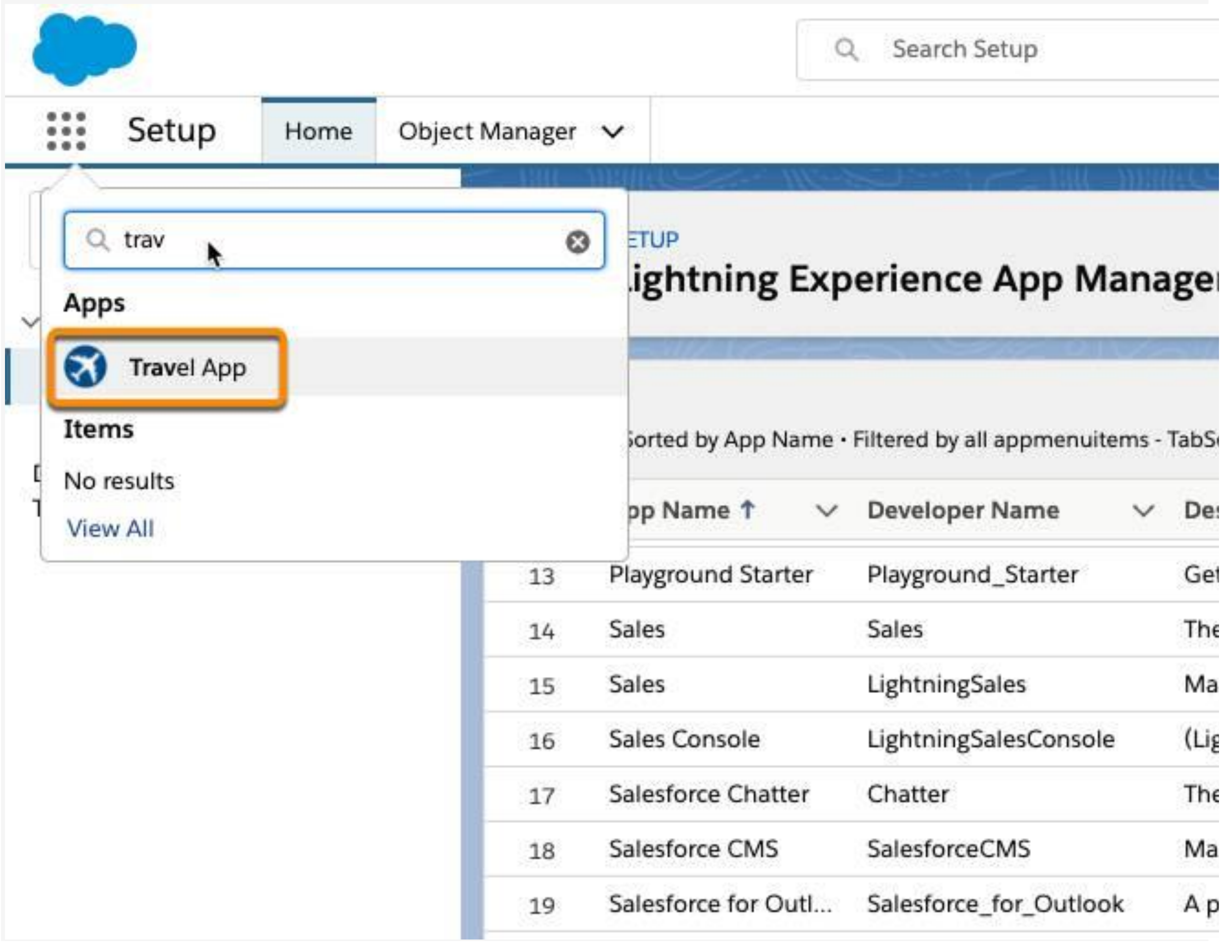

Navigation Items

Choose the items to include in the app, and arrange the order in which they appear. Users can personalize the navigation to add or move items, but users can' rename the items that you add. Some navigation items are available only for phone or only for desktop. These items are dropped from the navigation bar whe viewed in a format that the item doesn't support.

12. On the User Profiles screen, select **System Administrator** and add it to Selected Profiles, then click **Save & Finish**.

Now navigate to your new Travel App application to see what it looks like so far.

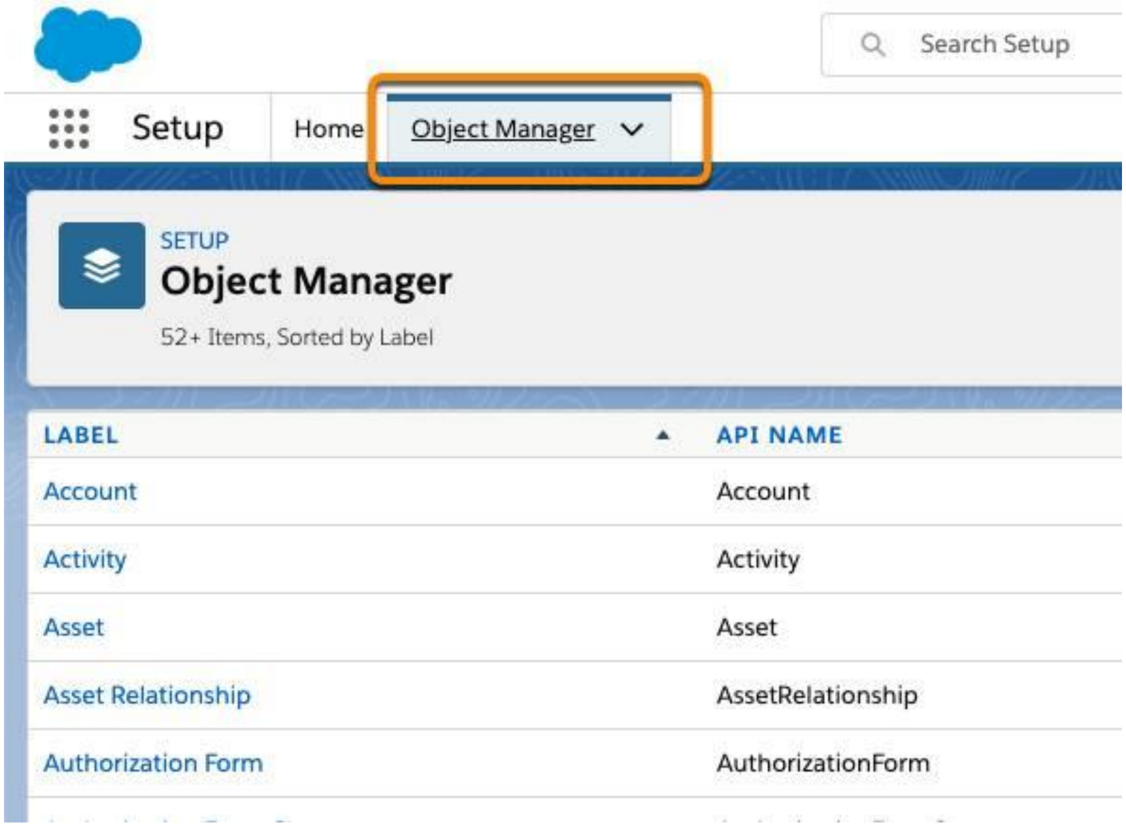
Click , then search for and select **Travel App**.


Search Setup
Setup
Home
Object Manager
trav
Apps
Travel App
Items
No results
View All
ETUP
ightning Experience App Manage
orted by App Name • Filtered by all appmenuitems - TabS

	pp Name ↑	Developer Name	De
13	Playground Starter	Playground_Starter	Ge
14	Sales	Sales	The
15	Sales	LightningSales	Ma
16	Sales Console	LightningSalesConsole	(Lig
17	Salesforce Chatter	Chatter	The
18	Salesforce CMS	SalesforceCMS	Ma
19	Salesforce for Outl...	Salesforce_for_Outlook	A p

The next step in creating your organization's travel approval app is to create the data model. The first object to create is the Department object. This object stores information about your departments such as name and cost center code.

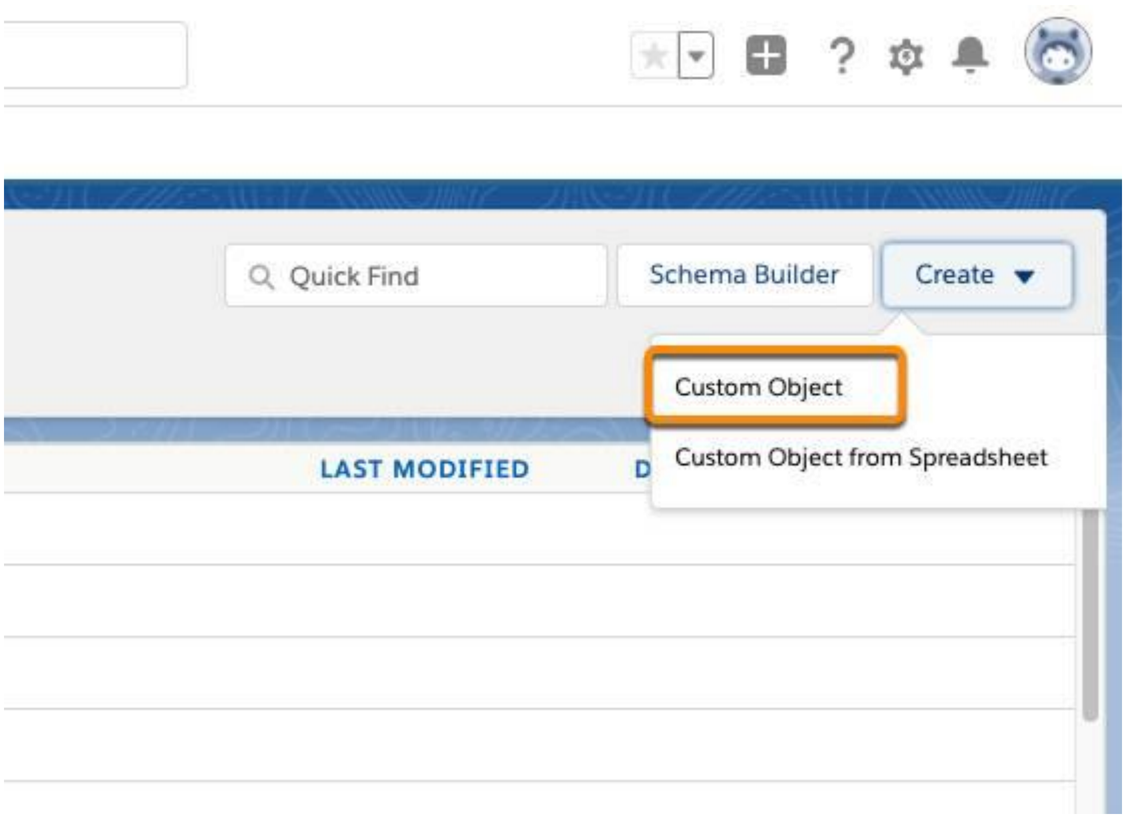
1. **From Setup, click Object Manager.**



SETUP
Object Manager
52+ Items, Sorted by Label

LABEL	API NAME
Account	Account
Activity	Activity
Asset	Asset
Asset Relationship	AssetRelationship
Authorization Form	AuthorizationForm

2. **Click Create, then select Custom Object.**



3. Enter these details:

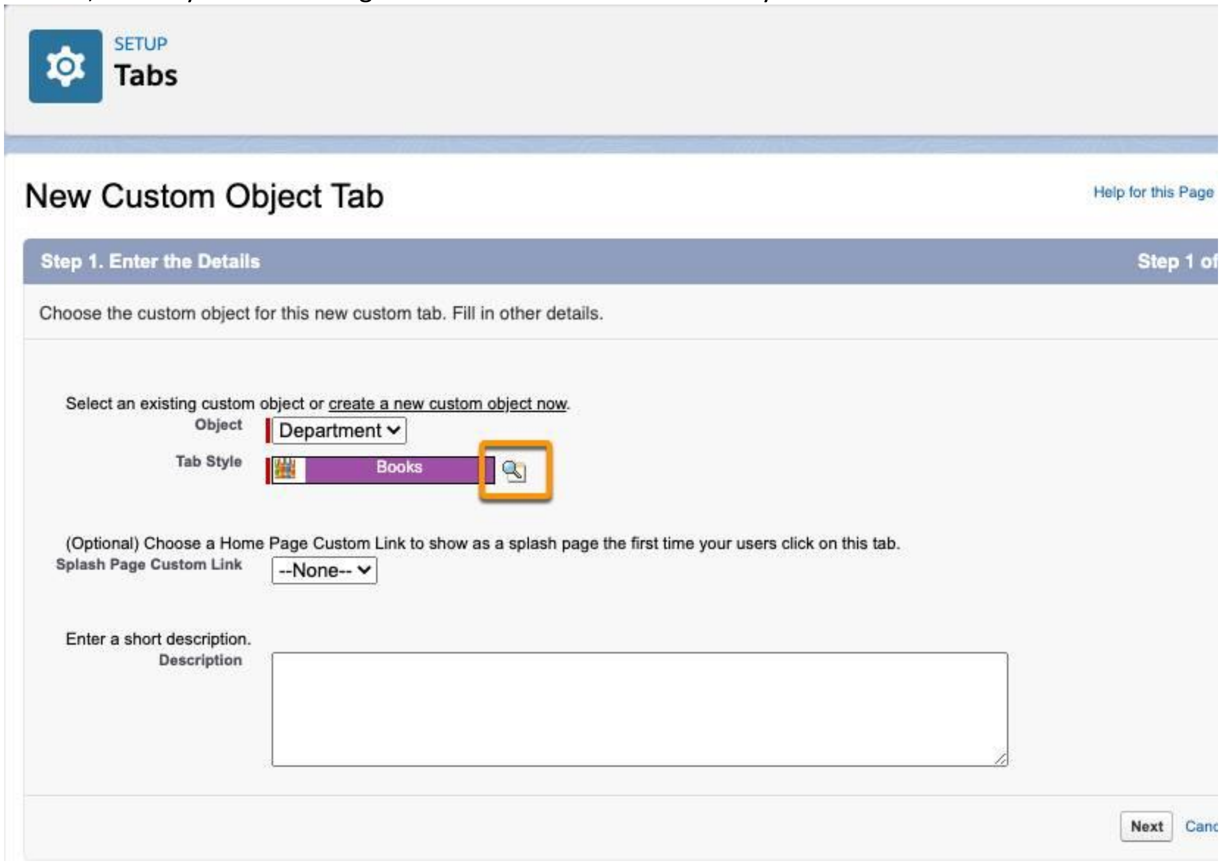
Field	Value
Label	Department
Plural Label	Departments
Object Name	Department (this field auto-populates)
Record Name	Department Name
Data Type	Text

4. Now scroll down and select these options:

- Allow Reports (1)
- Allow Search (2)
- Launch New Custom Tab Wizard after saving this custom object (3)

5. Click Save.

6. Now, define your tab settings. Click the next to the Tab Style field and select Books.



SETUP

Tabs

New Custom Object Tab

Help for this Page

Step 1. Enter the Details

Step 1 of

Choose the custom object for this new custom tab. Fill in other details.

Select an existing custom object or create a new custom object now.

Object: Department

Tab Style: Books

(Optional) Choose a Home Page Custom Link to show as a splash page the first time your users click on this tab.

Splash Page Custom Link: --None--

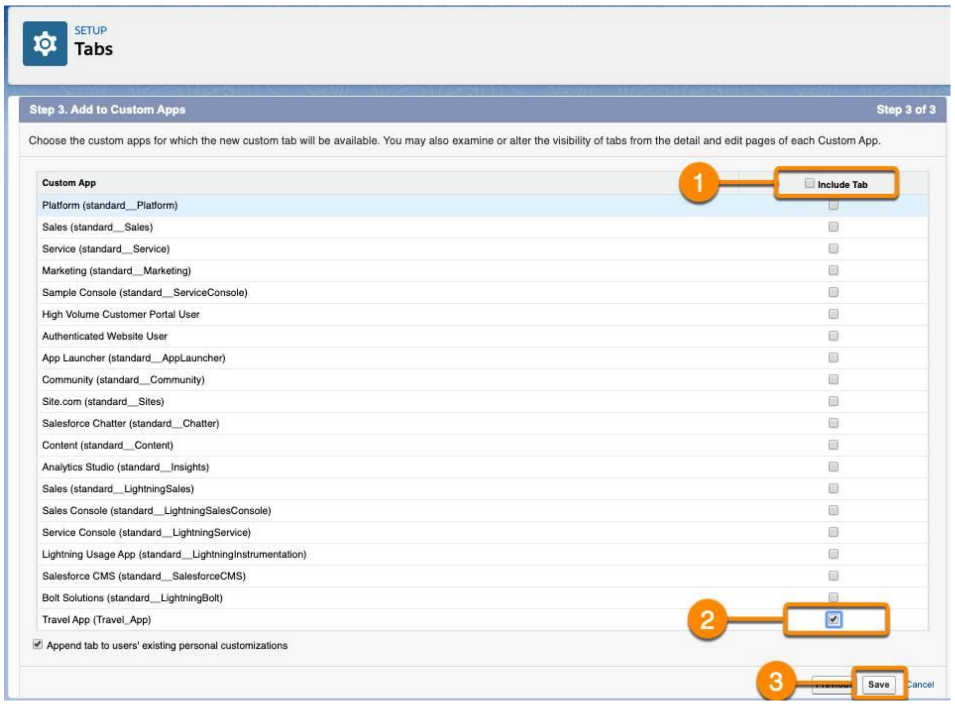
Enter a short description.

Description

7. Click Next. Leave the next screen as is and click Next.

8. Next, choose which applications to associate with this tab.

- Click the Include Tab (1) option at the top of the column to deselect all apps.
- Select only the Travel App (2) app from the list.
- Click Save (3).



Step 3. Add to Custom Apps — Step 3 of 3

Choose the custom apps for which the new custom tab will be available. You may also examine or alter the visibility of tabs from the detail and edit pages of each Custom App.

Custom App	☐ Include Tab
Platform (standard__Platform)	☐
Sales (standard__Sales)	☐
Service (standard__Service)	☐
Marketing (standard__Marketing)	☐
Sample Console (standard__ServiceConsole)	☐
High Volume Customer Portal User	☐
Authenticated Website User	☐
App Launcher (standard__AppLauncher)	☐
Community (standard__Community)	☐
Site.com (standard__Sites)	☐
Salesforce Chatter (standard__Chatter)	☐
Content (standard__Content)	☐
Analytics Studio (standard__Insights)	☐
Sales (standard__LightningSales)	☐
Sales Console (standard__LightningSalesConsole)	☐
Service Console (standard__LightningService)	☐
Lightning Usage App (standard__LightningInstrumentation)	☐
Salesforce CMS (standard__SalesforceCMS)	☐
Bolt Solutions (standard__LightningBolt)	☐
Travel App (Travel_App)	✓

✓ Append tab to users' existing personal customizations

9. On the next screen, click Fields & Relationships, then click New.
10. Select Text as the data type, then click Next.
11. Enter these details.
    - For Field Label, enter Department Code
    - For Length, enter 10
    - Select Required
    - Select Unique, then select case sensitive

12. Click Next.
13. Leave the Establish field-level security screen as is, and click Next.
14. Leave the Add to page layouts screen as is, and click Save.

Add the custom tab to the Lightning app

1. On the Setup Home page, click the App Launcher icon.
2. Search for the app in the Search apps and items box and open the respective app.
3. In the app navigation menu, click Edit.
4. In the Edit Contact Center App Navigation Items window, click Add More Items.
5. In the Add Items window, search for the custom tabs that you had created and select them.
6. Click Add 1 Nav Item.
7. Click Save.

45 **Write steps to add remote sites to the remote site settings to allow callouts to external sites.**

Remote Site Settings in Salesforce are a security feature that allows users to specify which external URLs Salesforce can access

These resources can include services, APIs, or websites that your Salesforce org needs to communicate with. By default, Salesforce blocks calls to external URLs from within its platform due to security considerations. However, you can use Remote Site Settings to explicitly allow these calls. Remote site settings help you maintain a secure environment while allowing your Salesforce organization to interact seamlessly with external web resources as needed for your business processes.

To add a remote site setting:

1. From Setup, enter Remote Site Settings in the Quick Find box, then select Remote Site Settings.
2. Click New Remote Site.
3. Enter a descriptive term for the Remote Site Name.
4. Enter the URL for the remote site.
5. Optionally, enter a description of the site.
6. Click Save.

46 **Describe the difference between web services and HTTP callouts.**

Features	**Web Services**	**HTTP Callouts**
Definition	**Standardized way of integrating applications using SOAP and WSDL.**	**Simpler HTTP requests using standard HTTP methods (GET, POST, PUT, DELETE).**
Protocols	**SOAP (Simple Object Access Protocol), WSDL (Web Services Description Language).**	**HTTP/HTTPS.**
Complexity	**Higher complexity with strict standards and extensive features.**	**Lower complexity, easier to implement.**
State	**Can be stateful or stateless.**	**Typically stateless.**

Overhead	**Higher due to comprehensive features and standards.**	**Lower, more lightweight.**
Security	**Advanced security features, including WS-Security standards.**	**Basic security via HTTPS; additional security needs to be manually implemented.**
Use Case	**Ideal for enterprise-level applications requiring standardized communication and strong security.**	**Suitable for lightweight interactions with external APIs and RESTful services.**
Error Handling	**More robust error handling mechanisms.**	**Simpler error handling.**
Setup	**Requires WSDL for setup and integration.**	**Simple setup using standard HTTP methods.**

47 **Explain REST and SOAP.**

SOAP(Simple Object Access Protocol) is an XML-based messaging protocol for exchanging information between systems and applications. SOAP is a communication protocol designed to communicate via the Internet. SOAP API is extensible, neutral, and independent.
REST is a set of architectural principles attuned to the needs of lightweight web services and mobile applications. Because it's a set of guidelines, it leaves the implementation of these recommendations to developers.

Features	**SOAP API**	**REST API**
Standard	SOAP API has official standards because it's a protocol.	REST API has no official standards because it's an architectural style of coding and tags.
Standard	SOAP only operates with HTTP and XML standards	REST API uses multiple standards, such as URL and HTTP

Functionality	SOAP uses a different interface to perform operations through a standardized set of messaging patterns	REST accesses data through a consistent interface with names of resources
Business logic	SOAP API uses service interfaces like @WebService	REST API uses a URL interface like @path(" /CricketService")
Bandwidth	SOAP uses XML to create a payload that results in large file sizes	REST API takes up little bandwidth and resources
Language	SOAP API uses Web Services Description language to describe service	REST API uses Web Application Description Language to describe service.
Rules	Rules for SOAP are very important because we can't reach any level of standardization without them	REST is more flexible but it also requires standardization.
Performance and Scalability	SOAP-based reads cannot be cached.	REST has better performance and scalability. REST reads can be cached.
Security	SOAP is more secure and uses **WS-Security** for enterprise-level security.	REST is less secure than SOAP API.

Reliable Messaging	SOAP provides asynchronous processing and a guaranteed level of reliability and security.	Rest doesn't have a standard messaging system and the client has to handle reliability.
Atomic Transaction	If we need ACID in the transaction, we have to use SOAP API	REST API does not support ACID.
Data Format	It supports XML data message	It supports JSON data.

48 Write the step-by-step procedure to Load data using the data import wizard.

Step 1: Access Data Import Wizard

1. Log in to Salesforce: Go to your Salesforce instance and log in with your credentials.
2. Navigate to Setup: Click on the gear icon in the top right corner and select "Setup".
3. Find Data Import Wizard: In the Quick Find box, type "Data Import Wizard" and select it from the results.

Step 2: Start the Import Process

1. Launch the Wizard: Click on "Launch Wizard" to start the data import process.
2. Choose Data Type: Select the type of data you want to import (e.g., Accounts, Contacts, Leads, etc.).

Step 3: Upload Your Data File

1. Select the CSV File: Click on "Choose CSV" and select the CSV file from your computer that contains the data you want to import.
2. Review and Start Mapping: Click on "Next" to proceed to the field mapping screen.

Step 4: Map Fields

1. Map Salesforce Fields: Map the fields in your CSV file to the corresponding Salesforce fields. Salesforce will try to auto-map fields, but you should review and adjust mappings as needed.
2. Review All Mappings: Ensure all required fields are mapped correctly. If any fields are not mapped, you'll be prompted to map them before proceeding.

Step 5: Review and Start Import

1. Review Settings: Review your import settings, such as matching criteria for duplicates, and other options.
2. Start Import: Click on "Start Import" to begin the data import process.

Step 6: Monitor and Review Import Results

1. Monitor Progress: You can monitor the progress of your import in the Data Import Wizard interface.
2. Review Import Summary: Once the import is complete, review the summary to check for any errors or issues.
3. Verify Data: Go to the relevant Salesforce records to verify that the data has been imported correctly.

49 **Write the steps to create department object in Travel Approval App.**

Step 1: Access Object Manager

1. Log in to Salesforce: Go to your Salesforce instance and log in with your credentials.
2. Navigate to Setup: Click on the gear icon in the top right corner and select "Setup".
3. Open Object Manager: In the Quick Find box, type "Object Manager" and select it from the results.

Step 2: Create a Custom Object

1. Click Create: In the Object Manager, click on the "Create" button and select "Custom Object".
2. Enter Object Details: Fill in the following details:
   - Plural Label: Departments
   - Object Name: Department (this field auto-populates)
   - Record Name: Department Name
   - Data Type: Text
3. Select Options: Scroll down and select the following options:
   - Allow Reports
   - Allow Search
   - Launch New Custom Tab Wizard after saving this custom object

Step 3: Save the Object

1. Click Save: Click on the "Save" button to create the Department object.

Step 4: Define Tab Settings

1. Create a Tab: After saving the object, click on the "New" button next to the Tab Style field and select a tab style (e.g., Books).
2. Select Applications: Choose which applications to associate with this tab. Deselect all apps and select only the Travel App.
3. Save Tab Settings: Click "Save" to finalize the tab settings.

Step 5: Add Fields to the Department Object

1. Click Fields & Relationships: In the Object Manager, click on "Fields & Relationships" and then click "New".
2. Create Fields: Add fields such as Department Code with the following details:
   - Field Label: Department Code
   - Data Type: Text
   - Length: 10
   - Required: Check this option
   - Unique: Check this option and select case-sensitive

3. Save Field: Click "Save" to add the field to the Department object.

Step 6: Add to Page Layouts

1. Add to Page Layouts: Ensure the Department object is added to relevant page layouts for easy access and visibility.

50 **Write the steps to create an Expense Item Object.**

Step 1: Access Object Manager

1. Log in to Salesforce: Go to your Salesforce instance and log in with your credentials.
2. Navigate to Setup: Click on the gear icon in the top right corner and select "Setup".
3. Open Object Manager: In the Quick Find box, type "Object Manager" and select it from the results.

Step 2: Create a Custom Object

1. Click Create: In the Object Manager, click on the "Create" button and select "Custom Object".
2. Enter Object Details: Fill in the following details:
    - Label: Expense Item
    - Plural Label: Expense Items
    - Object Name: Expense_Item (this field auto-populates)
    - Record Name: Expense Item Number
    - Data Type: Auto Number (you can choose the format you prefer, such as EI-{0000})
3. Select Options: Scroll down and select the following options:
    - Allow Reports
    - Allow Activities
    - Track Field History
    - Allow Search
    - Deploy (check this box to make the object available immediately)

Step 3: Save the Object

1. Click Save: Click on the "Save" button to create the Expense Item object.

Step 4: Define Tab Settings

1. Create a Tab: After saving the object, click on the "New" button next to the Tab Style field and select a tab style (e.g., Expense Reports).
2. Select Applications: Choose which applications to associate with this tab. Make sure it is linked to the relevant application, like an Expense Management app.
3. Save Tab Settings: Click "Save" to finalize the tab settings.

Step 5: Add Fields to the Expense Item Object

1. Click Fields & Relationships: In the Object Manager, click on "Fields & Relationships" and then click "New".
2. Create Fields: Add relevant fields such as:
    - Field Label: Expense Date
        - Data Type: Date
    - Field Label: Amount

        - Data Type: Currency
    - Field Label: Description
        - Data Type: Text Area (Long)
    - Field Label: Expense Type
        - Data Type: Picklist (create values such as Travel, Meals, Lodging, etc.)
3. Save Each Field: Click "Save" after defining each field to add it to the Expense Item object.

Step 6: Configure Page Layouts

1. Add to Page Layouts: Ensure the Expense Item object fields are added to the relevant page layouts for easy access and visibility.
2. Customize Layouts: Drag and drop fields to organize the page layout as desired.

Step 7: Set Up Record Types (Optional)

1. Record Types: If you need different layouts or processes for different types of expense items, you can set up record types.
2. Assign Layouts: Assign different page layouts to each record type as needed.