

1. Whenever a case is created with origin as email then set status as new and Priority as Medium.

```
trigger CaseTrigger on Case (before insert) {
    for (Case c : Trigger.new) {
        // Check if the origin is 'Email'
        if (c.Origin == 'Email') {
            // Set Status to 'New' and Priority to 'Medium'
            c.Status = 'New';
            c.Priority = 'Medium';
        }
    }
}
```

2. Whenever Lead is created with LeadSource as Web then give rating as cold otherwise hot.

```
trigger LeadTrigger on Lead (before insert) {
    for (Lead lead : Trigger.new) {
        // Check if the LeadSource is 'Web'
        if (lead.LeadSource == 'Web') {
            // Set the Rating to 'Cold'
            lead.Rating = 'Cold';
        } else {
            // Set the Rating to 'Hot' for all other LeadSource values
            lead.Rating = 'Hot';
        }
    }
}
```

3. Whenever New Account Record is created then needs to create associated Contact Record automatically.

```

trigger AccountTrigger on Account (after insert) {
    List<Contact> contactsToInsert = new List<Contact>();

    for (Account acc : Trigger.new) {
        // Create a Contact for each newly inserted Account
        Contact newContact = new Contact(
            FirstName = 'Default', // Set a default FirstName
            LastName = acc.Name,    // Use Account Name as LastName
            AccountId = acc.Id      // Associate the Contact with the Account
        );
        contactsToInsert.add(newContact);
    }

    // Insert the associated Contact records
    if (!contactsToInsert.isEmpty()) {
        insert contactsToInsert;
    }
}

```

4. Whenever the Account is created with Industry as Banking then create a contact for account, Contact Lastname as Account name and contact phone as account phone.


```

trigger AccountTrigger on Account (after insert) {
    // List to hold contacts to be inserted
    List<Contact> contactsToInsert = new List<Contact>();

    // Loop through new account records
    for (Account acc : Trigger.new) {
        // Check if the Industry is Banking
        if (acc.Industry == 'Banking') {
            // Create a new Contact record
            Contact newContact = new Contact(
                LastName = acc.Name, // Set LastName as Account Name
                Phone = acc.Phone,   // Set Phone as Account Phone
                AccountId = acc.Id   // Associate with the Account
            );
            // Add the contact to the list
            contactsToInsert.add(newContact);
        }
    }

    // Insert the contacts
    if (!contactsToInsert.isEmpty()) {
        insert contactsToInsert;
    }
}

```

 Copy code

5. Creates the number of contacts which are equal to the number which we will enter in the Number of Locations field on the Account Object.

```

trigger AccountTrigger on Account (after insert, after update) {
    // List to hold contacts to be inserted
    List<Contact> contactsToInsert = new List<Contact>();

    for (Account acc : Trigger.new) {
        // Ensure 'Number of Locations' has a valid number
        if (acc.Number_of_Locations__c != null && acc.Number_of_Locations__c > 0) {
            // Calculate the difference in number of contacts to create
            Integer numberOfContactsToCreate = acc.Number_of_Locations__c;

            // Create contacts based on the Number of Locations
            for (Integer i = 0; i < numberOfContactsToCreate; i++) {
                Contact newContact = new Contact(
                    LastName = acc.Name + ' - Location ' + (i + 1), // Unique LastName
                    AccountId = acc.Id                               // Associate with the A
                );
                contactsToInsert.add(newContact);
            }
        }
    }

    // Insert all new contacts
    if (!contactsToInsert.isEmpty()) {
        insert contactsToInsert;
    }
}

```

6. Upon Account Creation if Industry is not null and having value as 'Media' then populate Rating as Hot.

```

trigger AccountTrigger on Account (before insert) {
    for (Account acc : Trigger.new) {
        // Check if Industry is not null and equals 'Media'
        if (acc.Industry != null && acc.Industry == 'Media') {
            acc.Rating = 'Hot'; // Set the Rating to 'Hot'
        }
    }
}

```

7. Upon Opportunity Creation if Amount is not null and is greater than 100000 then populate 'Hot Opportunity' in description field.

```

apex

trigger OpportunityTrigger on Opportunity (before insert) {
    for (Opportunity opp : Trigger.new) {
        // Check if Amount is not null and greater than 100,000
        if (opp.Amount != null && opp.Amount > 100000) {
            opp.Description = 'Hot Opportunity'; // Set the Description
        }
    }
}

```

8. When an account inserts and CopyBillingToShipping (CustomField) checkbox is checked then automatically copy accountbilling address into account shipping address.

```
trigger AccountTrigger on Account (before insert, before update) {
    for (Account acc : Trigger.new) {
        // Check if CopyBillingToShipping__c is checked
        if (acc.CopyBillingToShipping__c == true) {
            // Copy Billing Address fields to Shipping Address fields
            acc.ShippingStreet = acc.BillingStreet;
            acc.ShippingCity = acc.BillingCity;
            acc.ShippingState = acc.BillingState;
            acc.ShippingPostalCode = acc.BillingPostalCode;
            acc.ShippingCountry = acc.BillingCountry;
        }
    }
}
```

9. Upon Creation of Position (Custom Object) if it is a New Position and Open Date, Min Pay & Max Pay are not populated then populated them with below values: a. Open Date = Today's Date b. Min Pay = 10000 c. Max Pay = 15000

```
trigger PositionTrigger on Position__c (before insert) {
    for (Position__c pos : Trigger.new) {
        // Check if it's a new position and fields are not populated
        if (pos.Open_Date__c == null) {
            pos.Open_Date__c = Date.today(); // Set Open Date to Today's Date
        }
        if (pos.Min_Pay__c == null) {
            pos.Min_Pay__c = 10000; // Set Min Pay to 10,000
        }
        if (pos.Max_Pay__c == null) {
            pos.Max_Pay__c = 15000; // Set Max Pay to 15,000
        }
    }
}
```

10. Create a related Contact when an Account is created.

```

trigger AccountTrigger on Account (after insert) {
    // List to store contacts for bulk insert
    List<Contact> contactsToInsert = new List<Contact>();

    for (Account acc : Trigger.new) {
        // Create a related Contact for each Account
        Contact contact = new Contact(
            FirstName = 'Default',          // Default first name
            LastName = acc.Name,            // Set Last Name to Account Name
            AccountId = acc.Id              // Associate the Contact with the Account
        );
        contactsToInsert.add(contact);
    }

    // Insert all Contacts in bulk
    if (!contactsToInsert.isEmpty()) {
        insert contactsToInsert;
    }
}

```

## 11. Create a related Opportunity when an Account is created.

```

trigger AccountTrigger on Account (after insert) {
    // List to store opportunities for bulk insert
    List<Opportunity> opportunitiesToInsert = new List<Opportunity>();

    for (Account acc : Trigger.new) {
        // Create a related Opportunity for each Account
        Opportunity opp = new Opportunity(
            Name = acc.Name + ' Opportunity', // Opportunity Name based on Account Name
            StageName = 'Prospecting',        // Default Stage
            CloseDate = Date.today().addDays(30), // Default Close Date (30 days from today)
            Amount = 10000,                   // Default Amount
            AccountId = acc.Id                // Associate Opportunity with the Account
        );
        opportunitiesToInsert.add(opp);
    }

    // Insert all Opportunities in bulk
    if (!opportunitiesToInsert.isEmpty()) {
        insert opportunitiesToInsert;
    }
}

```

## 12. When a Case is created on any Account, put the latest case number on the Account in the 'Latest Case Number' field.

```

trigger CaseTrigger on Case (after insert) {
    // Map to store Account Ids and latest Case Numbers
    Map<Id, String> accountToLatestCaseMap = new Map<Id, String>();

    for (Case c : Trigger.new) {
        if (c.AccountId != null) {
            accountToLatestCaseMap.put(c.AccountId, c.CaseNumber);
        }
    }

    // Query Accounts needing update
    List<Account> accountsToUpdate = [SELECT Id, Latest_Case_Number__c FROM Account WHERE Id IN :accountToLatestCaseMap.keySet()];

    // Update Latest Case Number field
    for (Account acc : accountsToUpdate) {
        acc.Latest_Case_Number__c = accountToLatestCaseMap.get(acc.Id);
    }

    // Perform the update
    if (!accountsToUpdate.isEmpty()) {
        update accountsToUpdate;
    }
}

```

13. Account records should have a field named 'Recent OpportunityAmount'. It should contain the opportunity amount of the latest created opportunity on account.

```

Trigger OpportunityTrigger on Opportunity (after insert, after update) {
    // Map to store Account Ids and the latest Opportunity Amount
    Map<Id, Decimal> accountToLatestOpportunityAmountMap = new Map<Id, Decimal>();

    // Loop through the Opportunity records to gather data
    for (Opportunity opp : Trigger.new) {
        // Check if Opportunity has an Account and that it's not deleted
        if (opp.AccountId != null && opp.IsDeleted == false) {
            accountToLatestOpportunityAmountMap.put(opp.AccountId, opp.Amount);
        }
    }

    // Query the Accounts to get the latest created Opportunity per Account
    List<Account> accountsToUpdate = [SELECT Id, Recent_Opportunity_Amount__c, (SELECT Id, Amount, CreatedDate FROM Opportunities ORDER BY CreatedDate DESC LIMIT 1)
    FROM Account WHERE Id IN :accountToLatestOpportunityAmountMap.keySet()];

    // Update the 'Recent Opportunity Amount' field for each Account
    for (Account acc : accountsToUpdate) {
        // Set the most recent Opportunity amount (the first Opportunity in the list after sorting by CreatedDate DESC)
        if (!acc.Opportunities.isEmpty()) {
            acc.Recent_Opportunity_Amount__c = acc.Opportunities[0].Amount;
        }
    }

    // Perform the update operation for all affected Accounts
    if (!accountsToUpdate.isEmpty()) {
        update accountsToUpdate;
    }
}

```

14. On Account create two checkbox elds labeled as Contact and Opportunity. Now when a new Account record is created and if a particular Contact or Opportunity checkbox is checked then createthat related record. Also Opportunity record should be created only ifthe Account record Active picklist is populated with a Yes.

```

trigger CreateRelatedRecords on Account (after insert) {
    // Lists to store Contacts and Opportunities to be created
    List<Contact> contactsToInsert = new List<Contact>();
    List<Opportunity> opportunitiesToInsert = new List<Opportunity>();

    // Iterate through new Account records
    for (Account acc : Trigger.new) {
        // Check if the Contact checkbox is checked
        if (acc.Contact__c) {
            // Create a Contact associated with this Account
            Contact newContact = new Contact(
                FirstName = acc.Name, // Example: Using Account Name for simplicity
                LastName = acc.Name + ' Contact', // Required field
                AccountId = acc.Id
            );
            contactsToInsert.add(newContact);
        }

        // Check if the Opportunity checkbox is checked and Active is 'Yes'
        if (acc.Opportunity__c && acc.Active__c == 'Yes') {
            // Create an Opportunity associated with this Account
            Opportunity newOpportunity = new Opportunity(
                Name = acc.Name + ' Opportunity', // Example: Using Account Name for Opportunity
                AccountId = acc.Id,
                CloseDate = System.today().addMonths(1), // Example: Close Date set to one month later
                StageName = 'Prospecting' // Example: Default Stage Name
            );
            opportunitiesToInsert.add(newOpportunity);
        }
    }

    // Insert the Contacts and Opportunities in bulk
    if (!contactsToInsert.isEmpty()) {
        insert contactsToInsert;
    }
    if (!opportunitiesToInsert.isEmpty()) {
        insert opportunitiesToInsert;
    }
}

```

15. If the Account phone is updated then populate below message  
indescriptionDescription = Phone is Updated!

```

trigger UpdateDescriptionOnPhoneChange on Account (before update) {
    for (Account acc : Trigger.new) {
        Account oldAcc = Trigger.oldMap.get(acc.Id);
        // Check if the Phone field has been updated
        if (acc.Phone != oldAcc.Phone) {
            acc.Description = 'Phone is Updated!';
        }
    }
}

```

16. When an account is inserted or updated and theCopyBillingToShippingcheckbox is checked then automatically copy the account billing addressinto account shipping address.

```

trigger CopyBillingToShipping on Account (before insert, before update) {
    for (Account acc : Trigger.new) {
        // Check if the CopyBillingToShipping checkbox is checked
        if (acc.CopyBillingToShipping__c) {
            acc.ShippingStreet = acc.BillingStreet;
            acc.ShippingCity = acc.BillingCity;
            acc.ShippingState = acc.BillingState;
            acc.ShippingPostalCode = acc.BillingPostalCode;
            acc.ShippingCountry = acc.BillingCountry;
        }
    }
}


```

17. If opportunity Stage is updated upon its creation or update then updatedescription as either 'Opp is Closed Lost' or 'Opp is Closed Won' or 'Opp ISOpen '

```

trigger UpdateDescriptionOnStageChange on Opportunity (before insert, before update) {
    for (Opportunity opp : Trigger.new) {
        // Check the StageName and update the Description field accordingly
        if (opp.StageName == 'Closed Lost') {
            opp.Description = 'Opp is Closed Lost';
        } else if (opp.StageName == 'Closed Won') {
            opp.Description = 'Opp is Closed Won';
        } else {
            opp.Description = 'Opp is Open';
        }
    }
}

```



18. If the Account phone is updated then populate the phone num



```

trigger UpdatePhoneNumberField on Account (before update) {
    for (Account acc : Trigger.new) {
        // Get the old value of the Account from Trigger.oldMap
        Account oldAcc = Trigger.oldMap.get(acc.Id);

        // Check if the Phone field has been updated
        if (acc.Phone != oldAcc.Phone) {
            // Update the custom field with the new Phone value
            acc.UpdatedPhoneNumber__c = acc.Phone;
        }
    }
}

```

19. If the Account phone is updated then populate the phone number on all related Contacts (Home Phone eld). [Using Parent-Child SOQL]

```

trigger UpdateContactsHomePhone on Account (after update) {
    // List to store Contacts that need to be updated
    List<Contact> contactsToUpdate = new List<Contact>();

    // Iterate through updated Accounts
    for (Account acc : Trigger.new) {
        Account oldAcc = Trigger.oldMap.get(acc.Id);

        // Check if the Phone field has been updated
        if (acc.Phone != oldAcc.Phone) {
            // Query related Contacts using Parent-Child SOQL
            for (Contact con : [SELECT Id, HomePhone FROM Contact WHERE AccountId = :acc.Id]) {
                con.HomePhone = acc.Phone;
                contactsToUpdate.add(con);
            }
        }
    }

    // Update all modified Contacts
    if (!contactsToUpdate.isEmpty()) {
        update contactsToUpdate;
    }
}

```

20. If the Account billing address is updated then update related contactsmailing address. [Using Map]

```

trigger AccountTrigger on Account (after update) {
    // Map to hold Account Id and updated Billing Address
    Map<Id, Account> updatedAccountsMap = new Map<Id, Account>();
    // Loop through updated accounts and store the ones where the Billing Address has changed
    for (Account acc : Trigger.new) {
        Account oldAcc = Trigger.oldMap.get(acc.Id);
        if (acc.BillingStreet != oldAcc.BillingStreet ||
            acc.BillingCity != oldAcc.BillingCity ||
            acc.BillingState != oldAcc.BillingState ||
            acc.BillingPostalCode != oldAcc.BillingPostalCode ||
            acc.BillingCountry != oldAcc.BillingCountry) {
            updatedAccountsMap.put(acc.Id, acc);
        }
    }
    // Proceed only if there are updated accounts
    if (!updatedAccountsMap.isEmpty()) {
        // Query related contacts for updated accounts
        List<Contact> contactsToUpdate = [
            SELECT Id, MailingStreet, MailingCity, MailingState, MailingPostalCode, MailingCountry
            FROM Contact
            WHERE AccountId IN :updatedAccountsMap.keySet()
        ];
        // Update Contact mailing addresses with Account billing address
        for (Contact contact : contactsToUpdate) {
            Account updatedAccount = updatedAccountsMap.get(contact.AccountId);
            contact.MailingStreet = updatedAccount.BillingStreet;
            contact.MailingCity = updatedAccount.BillingCity;
            contact.MailingState = updatedAccount.BillingState;
            contact.MailingPostalCode = updatedAccount.BillingPostalCode;
            contact.MailingCountry = updatedAccount.BillingCountry;
        }
        // Perform DML update on Contacts
        if (!contactsToUpdate.isEmpty()) {
            update contactsToUpdate;
        }
    }
}

```

21. If the Account billing address is updated then update related contact mailing address.  
[Using Parent-Child SOQL]

```

trigger AccountTrigger on Account (after update) {
    // List to hold Contacts for update
    List<Contact> contactsToUpdate = new List<Contact>();
    // Iterate through updated accounts
    for (Account acc : {
        SELECT Id, BillingStreet, BillingCity, BillingState, BillingPostalCode, BillingCountry,
            (SELECT Id, MailingStreet, MailingCity, MailingState, MailingPostalCode, MailingCountry FROM Contacts)
        FROM Account
        WHERE Id IN :Trigger.newMap.keySet()
    }) {
        Account oldAcc = Trigger.oldMap.get(acc.Id);

        // Check if Billing Address has changed
        if (acc.BillingStreet != oldAcc.BillingStreet ||
            acc.BillingCity != oldAcc.BillingCity ||
            acc.BillingState != oldAcc.BillingState ||
            acc.BillingPostalCode != oldAcc.BillingPostalCode ||
            acc.BillingCountry != oldAcc.BillingCountry) {

            // Update related Contacts' Mailing Address
            for (Contact contact : acc.Contacts) {
                contact.MailingStreet = acc.BillingStreet;
                contact.MailingCity = acc.BillingCity;
                contact.MailingState = acc.BillingState;
                contact.MailingPostalCode = acc.BillingPostalCode;
                contact.MailingCountry = acc.BillingCountry;

                contactsToUpdate.add(contact);
            }
        }
    }

    // Perform DML update on Contacts if there are any updates
    if (!contactsToUpdate.isEmpty()) {
        update contactsToUpdate;
    }
}

```

22. When a Opportunity Stage (eld) is changed, create a Task record on Opportunity and assign it to Logged In User/Opportunity Owner / Any User.

```
Trigger OpportunityTrigger on Opportunity (after update) {
    // List to hold Task records for insertion
    List<Task> tasksToInsert = new List<Task>();

    // Iterate through updated opportunities
    for (Opportunity opp : Trigger.new) {
        Opportunity oldOpp = Trigger.oldMap.get(opp.Id);

        // Check if StageName has changed
        if (opp.StageName != oldOpp.StageName) {
            // Create a new Task record
            Task newTask = new Task();
            newTask.WhatId = opp.Id; // Link the Task to the Opportunity
            newTask.Subject = 'Follow-up due to stage change';
            newTask.Description = 'The Opportunity Stage was changed to ' + opp.StageName;
            newTask.Priority = 'High'; // Set default priority
            newTask.Status = 'Not Started'; // Set default status

            // Assign the Task
            if (UserInfo.getUserId() != null) {
                newTask.OwnerId = UserInfo.getUserId(); // Assign to the logged-in user
            } else {
                newTask.OwnerId = opp.OwnerId; // Fallback to Opportunity Owner
            }

            // Add the Task to the list for insertion
            tasksToInsert.add(newTask);
        }
    }

    // Insert Task records
    if (!tasksToInsert.isEmpty()) {
        insert tasksToInsert;
    }
}
```

23. Write a trigger on Account when Account Active eld is updated from 'Yes' to 'No' then check all opportunities associated with the account. Update all Opportunities Stage to close lost if stage not equal to closewon

```

trigger AccountTrigger on Account (after update) {
    // List to hold Opportunities for update
    List<Opportunity> opportunitiesToUpdate = new List<Opportunity>();

    // Iterate through updated Accounts
    for (Account acc : Trigger.new) {
        Account oldAcc = Trigger.oldMap.get(acc.Id);

        // Check if Active field has been updated from 'Yes' to 'No'
        if (oldAcc.Active == 'Yes' && acc.Active == 'No') {
            // Query associated Opportunities where StageName is not 'Closed Won'
            List<Opportunity> relatedOpportunities = [
                SELECT Id, StageName
                FROM Opportunity
                WHERE AccountId = :acc.Id AND StageName != 'Closed Won'
            ];

            // Update the StageName of eligible Opportunities
            for (Opportunity opp : relatedOpportunities) {
                opp.StageName = 'Closed Lost';
                opportunitiesToUpdate.add(opp);
            }
        }
    }

    // Perform DML update on Opportunities if any exist
    if (!opportunitiesToUpdate.isEmpty()) {
        update opportunitiesToUpdate;
    }
}

```

24. Account records cannot be deleted if active is Yes.

```

trigger AccountTrigger on Account (before delete) {
    for (Account acc : Trigger.old) {
        // Check if Active is 'Yes'
        if (acc.Active == 'Yes') {
            acc.addError('Accounts with Active set to "Yes" cannot be deleted.');
        }
    }
}

```

25. Prevent account record from being edited if the record is created 7daysback.

```

trigger AccountTrigger on Account (before update) {
    for (Account acc : Trigger.new) {
        // Check if the Account is being edited after 7 days of creation
        if (acc.CreatedDate != null &&
            acc.CreatedDate.addDays(7) <= DateTime.now()) {
            acc.addError('This Account record cannot be edited as it was created more than 7 days ago.');
        }
    }
}

```

26. Apply validation using addError() method in trigger. While Creation of Opportunity is Amount is null then throw an error message.

```
trigger OpportunityTrigger on Opportunity (before insert) {
    for (Opportunity opp : Trigger.new) {
        // Check if the Amount field is null
        if (opp.Amount == null) {
            opp.addError('Amount is required and cannot be null for Opportunity creation.');
```

27. When an opportunity is updated to Closed Lost and Closed LostReason (eld) is not populated then throw validation error that 'Please populate Closed Lost Reason' on opportunity. [before update]

```
Trigger OpportunityTrigger on Opportunity (before update) {
    for (Opportunity opp : Trigger.new) {
        // Check if the StageName is being updated to "Closed Lost"
        if (opp.StageName == 'Closed Lost' &&
            Trigger.oldMap.get(opp.Id).StageName != 'Closed Lost' &&
            String.isBlank(opp.Closed_Lost_Reason__c)) {
            opp.addError('Please populate Closed Lost Reason when updating the Opportunity to Closed Lost.');
```

28. Write a trigger on Account and check only System Administrator profile users should be able to delete an account.

```
trigger AccountTrigger on Account (before delete) {
    // Fetch the current user's profile
    String userProfileName = [SELECT Profile.Name FROM User WHERE Id = :UserInfo.getUserId()].Profile.Name;

    // Check if the user's profile is not "System Administrator"
    if (userProfileName != 'System Administrator') {
        for (Account acc : Trigger.old) {
            acc.addError('You do not have permission to delete this Account. Only System Administrator users can perform this action.');
```

29. If an opportunity is closed then, no one should be able to delete it except the user having a System Administrator profile.

```
trigger OpportunityTrigger on Opportunity (before delete) {
    // Fetch the current user's profile
    String userProfileName = [SELECT Profile.Name FROM User WHERE Id = :UserInfo.getUserId()].Profile.Name;

    for (Opportunity opp : Trigger.old) {
        // Check if the Opportunity is closed and the user is not a System Administrator
        if (opp.IsClosed && userProfileName != 'System Administrator') {
            opp.addError('Closed Opportunities can only be deleted by users with the System Administrator profile.');
```

30. Prevent deletion of an account if there is any opportunity related to that account.

```

trigger AccountTrigger on Account (before delete) {
    // Query related Opportunities for the accounts being deleted
    Set<Id> accountIds = new Set<Id>();
    for (Account acc : Trigger.old) {
        accountIds.add(acc.Id);
    }

    // Check for related Opportunities
    Map<Id, AggregateResult> accountWithOpportunities = new Map<Id, AggregateResult>(
        [SELECT AccountId, COUNT(Id) opportunityCount
         FROM Opportunity
         WHERE AccountId IN :accountIds
         GROUP BY AccountId]
    );

    // Prevent deletion if related Opportunities exist
    for (Account acc : Trigger.old) {
        if (accountWithOpportunities.containsKey(acc.Id)) {
            acc.addError('This Account cannot be deleted because it has related Opportunities.');
        }
    }
}

```

31. Prevent deletion of an account if there is any case related to that account.

```

trigger AccountTrigger on Account (before delete) {
    // Collect all Account IDs from the records being deleted
    Set<Id> accountIds = new Set<Id>();
    for (Account acc : Trigger.old) {
        accountIds.add(acc.Id);
    }

    // Query for related Cases to the Accounts being deleted
    Map<Id, Integer> accountWithCases = new Map<Id, Integer>(
        [SELECT AccountId, COUNT(Id)
         FROM Case
         WHERE AccountId IN :accountIds
         GROUP BY AccountId]
    );

    // Check if any Account has related Cases and prevent deletion
    for (Account acc : Trigger.old) {
        if (accountWithCases.containsKey(acc.Id) && accountWithCases.get(acc.Id) > 0) {
            acc.addError('This Account cannot be deleted because it has related Cases.');
        }
    }
}

```

32.

### 32. When the Employee record is deleted then update 'Left Employee Count' on Account.

```
trigger EmployeeTrigger on Employee__c (before delete) {
    // Create a map to track Accounts and their related employees
    Map<Id, Integer> accountEmployeeCountMap = new Map<Id, Integer>();

    // Loop through the Employee records that are being deleted
    for (Employee__c emp : Trigger.old) {
        // Check if the Employee is linked to an Account
        if (emp.Account__c != null) {
            // Increment the count of left employees for the related Account
            if (!accountEmployeeCountMap.containsKey(emp.Account__c)) {
                accountEmployeeCountMap.put(emp.Account__c, 0);
            }
            accountEmployeeCountMap.put(emp.Account__c, accountEmployeeCountMap.get(emp.Account__c) + 1);
        }
    }

    // Update the 'Left Employee Count' on the related Accounts
    List<Account> accountsToUpdate = new List<Account>();
    for (Id accountId : accountEmployeeCountMap.keySet()) {
        Account acc = new Account(Id = accountId);
        acc.Left_Employee_Count__c = accountEmployeeCountMap.get(accountId);
        accountsToUpdate.add(acc);
    }

    // Perform DML operation to update Accounts
    if (!accountsToUpdate.isEmpty()) {
        update accountsToUpdate;
    }
}
```

### 33. Undelete Employee record and set Active as true.

```
trigger EmployeeTrigger on Employee__c (after undelete) {
    List<Employee__c> employeesToUpdate = new List<Employee__c>();

    for (Employee__c employee : Trigger.new) {
        if (employee.IsDeleted == false) {
            employee.Active__c = true;
            employeesToUpdate.add(employee);
        }
    }

    try {
        if (!employeesToUpdate.isEmpty()) {
            update employeesToUpdate;
        }
    } catch (DmlException e) {
        System.debug('Error updating employee records: ' + e.getMessage());
    }
}
```

### 34. When the Employee record is inserted, deleted and undeleted then update 'Present Employee Count' on related Account. [Parent-Child SOQL]



```

trigger EmployeeTrigger on Employee__c (after insert, after delete, after undelete) {
    // Set to hold Account IDs for which the employee count needs to be updated
    Set<Id> accountIds = new Set<Id>();
    // Add Account IDs from the trigger records
    if (Trigger.isInsert || Trigger.isUndelete) {
        for (Employee__c emp : Trigger.new) {
            if (emp.Account__c != null) {
                accountIds.add(emp.Account__c);
            }
        }
    }
    if (Trigger.isDelete) {
        for (Employee__c emp : Trigger.old) {
            if (emp.Account__c != null) {
                accountIds.add(emp.Account__c);
            }
        }
    }
    // Query Accounts and count active employees
    List<Account> accountsToUpdate = new List<Account>();
    if (!accountIds.isEmpty()) {
        // Parent-child SOQL query
        Map<Id, Integer> employeeCountMap = new Map<Id, Integer>();
        for (Account acc : [
            SELECT Id, (SELECT Id FROM Employees__r WHERE Active__c = true) FROM Account
            WHERE Id IN :accountIds
        ]) {
            Integer activeEmployeeCount = acc.Employees__r.size();
            employeeCountMap.put(acc.Id, activeEmployeeCount);

            // Prepare Account for update
            Account accToUpdate = new Account(Id = acc.Id, Present_Employee_Count__c = activeEmployeeCount);
            accountsToUpdate.add(accToUpdate);
        }

        // Update Accounts
        try {
            update accountsToUpdate;
        } catch (DmlException e) {
            System.debug('Error updating Account records: ' + e.getMessage());
        }
    }
}

```

35. Upon contact creation an email should be sent to email populated onContact with specied template.

```

trigger ContactTrigger on Contact (after insert) {
    // Iterate through newly inserted Contacts and send an email
    for (Contact con : Trigger.new) {
        if (con.Email != null) {
            try {
                // Get email template
                EmailTemplate emailTemplate = [SELECT Id FROM EmailTemplate WHERE DeveloperName = 'Contact_Welcome_Template' LIMIT 1];

                // Create SingleEmailMessage
                Messaging.SingleEmailMessage email = new Messaging.SingleEmailMessage();
                email.setTemplateId(emailTemplate.Id);
                email.setTargetObjectId(con.Id); // The target contact
                email.setSaveAsActivity(false); // Do not save the email as an activity

                // Send the email
                Messaging.sendEmail(new List<Messaging.SingleEmailMessage> { email });
            } catch (Exception e) {
                System.debug('Error sending email: ' + e.getMessage());
            }
        }
    }
}

```

36. Create two record types named as “Partner Case” and “Customer Case” on

```

trigger CaseTrigger on Case (before insert) {
    // Map to store record type IDs
    Map<String, Id> recordTypeMap = new Map<String, Id>();

    // Fetch Record Type IDs
    for (RecordType rt : [SELECT Id, DeveloperName FROM RecordType WHERE SObjectType = 'Case']) {
        recordTypeMap.put(rt.DeveloperName, rt.Id);
    }

    // Assign Record Type based on some logic
    for (Case c : Trigger.new) {
        if (c.Origin == 'Partner') {
            c.RecordTypeId = recordTypeMap.get('Partner_Case');
        } else if (c.Origin == 'Customer') {
            c.RecordTypeId = recordTypeMap.get('Customer_Case');
        }
    }
}

```

37. When any Opportunity is created with amount populated or OpportunityAmount is updated then populate total Amount on Account Level for all related opportunities in Annual Revenue Field. If opportunity is deleted or undeleted then update Amount on Account as well. (Hint: rollup summary)

```

trigger OpportunityTrigger on Opportunity (after insert, after update, after delete, after undelete) {
    // Set of Account IDs to be processed
    Set<Id> accountIds = new Set<Id>();

    // Add Account IDs from Trigger context
    if (Trigger.isInsert || Trigger.isUpdate || Trigger.isUndelete) {
        for (Opportunity opp : Trigger.new) {
            if (opp.AccountId != null) {
                accountIds.add(opp.AccountId);
            }
        }
    }
    if (Trigger.isDelete) {
        for (Opportunity opp : Trigger.old) {
            if (opp.AccountId != null) {
                accountIds.add(opp.AccountId);
            }
        }
    }
}

// Map to hold Account totals
Map<Id, Decimal> accountTotals = new Map<Id, Decimal>();

// Query total Opportunity Amounts grouped by Account
if (!accountIds.isEmpty()) {
    AggregateResult[] results = [
        SELECT AccountId, SUM(Amount) totalAmount
        FROM Opportunity
        WHERE AccountId IN :accountIds AND IsDeleted = false
        GROUP BY AccountId
    ];

    // Populate totals
    for (AggregateResult ar : results) {
        accountTotals.put((Id) ar.get('AccountId'), (Decimal) ar.get('totalAmount'));
    }
}

// Prepare Accounts for update
List<Account> accountsToUpdate = new List<Account>();
for (Id accountId : accountIds) {
    Account acc = new Account(Id = accountId);

    // Set AnnualRevenue to the total amount or 0 if no related Opportunities
    acc.AnnualRevenue = accountTotals.containsKey(accountId) ? accountTotals.get(accountId) : 0;
    accountsToUpdate.add(acc);
}

// Update Accounts
try {
    if (!accountsToUpdate.isEmpty()) {
        update accountsToUpdate;
    }
} catch (DmlException e) {
    System.debug('Error updating Account AnnualRevenue: ' + e.getMessage());
}
}

```

38. Write a trigger, if the owner of an account is changed then the owner for the related contacts should also be updated. [Using Map]

```

trigger AccountOwnerChangeTrigger on Account (after update) {
    // Map to store Account IDs where the owner has changed
    Map<Id, Id> accountToNewOwnerMap = new Map<Id, Id>();

    // Collect Account IDs and new owner IDs where the owner has changed
    for (Account acc : Trigger.new) {
        Account oldAcc = Trigger.oldMap.get(acc.Id);
        if (acc.OwnerId != oldAcc.OwnerId) {
            accountToNewOwnerMap.put(acc.Id, acc.OwnerId);
        }
    }

    // Proceed only if there are changes in Account owners
    if (!accountToNewOwnerMap.isEmpty()) {
        // Query Contacts related to the affected Accounts
        List<Contact> contactsToUpdate = [
            SELECT Id, AccountId, OwnerId
            FROM Contact
            WHERE AccountId IN :accountToNewOwnerMap.keySet()
        ];

        // Update the owner of related Contacts
        for (Contact con : contactsToUpdate) {
            con.OwnerId = accountToNewOwnerMap.get(con.AccountId);
        }

        // Perform the DML operation
        try {
            if (!contactsToUpdate.isEmpty()) {
                update contactsToUpdate;
            }
        } catch (DmlException e) {
            System.debug('Error updating Contact owners: ' + e.getMessage());
        }
    }
}

```

39. Whenever a new User having prole “System Administrator” is inserted and is Active, add the user to the public group “Admins”. Create a public group named Admins.

```

trigger UserTrigger on User (after insert) {
    // Get the Id of the "Admins" public group
    Group adminsGroup = [SELECT Id FROM Group WHERE Name = 'Admins' AND Type = 'Regular' LIMIT 1];

    // Prepare a list of GroupMember objects for users to be added to the group
    List<GroupMember> groupMembers = new List<GroupMember>();

    for (User newUser : Trigger.new) {
        // Check if the user is active and has the "System Administrator" role
        if (newUser.IsActive && newUser.UserRoleId != null) {
            // Query the role name to ensure it's "System Administrator"
            String roleName = [SELECT Name FROM UserRole WHERE Id = :newUser.UserRoleId LIMIT 1].Name;
            if (roleName == 'System Administrator') {
                groupMembers.add(new GroupMember(GroupId = adminsGroup.Id, UserOrGroupId = newUser.Id));
            }
        }
    }

    // Insert the group members
    try {
        if (!groupMembers.isEmpty()) {
            insert groupMembers;
        }
    } catch (DmlException e) {
        System.debug('Error adding users to Admins group: ' + e.getMessage());
    }
}

```

40. Write a trigger on contact to prevent duplicate records based on Contact Email.

```

trigger PreventDuplicateContact on Contact (before insert, before update) {
    // Set to store incoming emails to check for duplicates
    Set<String> incomingEmails = new Set<String>();
    // Map to store existing emails from the database
    Map<String, Id> existingEmailToContactId = new Map<String, Id>();

    // Collect emails from the trigger context
    for (Contact con : Trigger.new) {
        if (con.Email != null) {
            incomingEmails.add(con.Email.toLowerCase());
        }
    }

    if (!incomingEmails.isEmpty()) {
        // Query existing contacts with matching emails
        for (Contact existingCon : [
            SELECT Id, Email FROM Contact WHERE Email IN :incomingEmails
        ]) {
            existingEmailToContactId.put(existingCon.Email.toLowerCase(), existingCon.Id);
        }
    }

    // Check for duplicates in both the database and within the same transaction
    for (Contact con : Trigger.new) {
        if (con.Email != null) {
            String emailLower = con.Email.toLowerCase();

            // Check if the email exists in the database
            if (existingEmailToContactId.containsKey(emailLower) &&
                existingEmailToContactId.get(emailLower) != con.Id) {
                con.addError('A Contact with this email already exists.');
            }

            // Check for duplicates within the same transaction
            for (Contact innerCon : Trigger.new) {
                if (innerCon.Id != con.Id &&
                    innerCon.Email != null &&
                    innerCon.Email.toLowerCase() == emailLower) {
                    con.addError('A duplicate Contact with the same email exists in this transaction.');
                }
            }
        }
    }
}

```

41. Set OWD as Private for Account. Once an Account record is created, it should be automatically shared with any one user who belongs to Standard User profile.

```

trigger AccountSharingTrigger on Account (after insert) {
    // Fetch the Standard User profile Id
    Id standardUserProfileId = [SELECT Id FROM Profile WHERE Name = 'Standard User' LIMIT 1].Id;
    // Query one active user with the Standard User profile
    User standardUser = [
        SELECT Id
        FROM User
        WHERE ProfileId = :standardUserProfileId AND IsActive = true
        LIMIT 1
    ];
    // Prepare the list of AccountShare records
    List<AccountShare> accountShares = new List<AccountShare>();

    for (Account acc : Trigger.new) {
        AccountShare accountShare = new AccountShare();
        accountShare.AccountId = acc.Id;
        accountShare.UserOrGroupId = standardUser.Id; // User to share the account with
        accountShare.AccountAccessLevel = 'Edit'; // Specify access level (Read, Edit, etc.)
        accountShare.RowCause = Schema.AccountShare.RowCause.Manual; // Row cause
        accountShares.add(accountShare);
    }

    // Insert the AccountShare records
    try {
        if (!accountShares.isEmpty()) {
            insert accountShares;
        }
    } catch (DmlException e) {
        System.debug('Error sharing Account: ' + e.getMessage());
    }
}

```

#### 42. when ever a create a lead object automatically converted account ,contact,opportunity

```

trigger LeadConversionTrigger on Lead (after insert) {
    List<Database.LeadConvert> leadConverts = new List<Database.LeadConvert>();
    for (Lead lead : Trigger.new) {
        if (!lead.isConverted) { // Only process unconverted leads
            Database.LeadConvert lc = new Database.LeadConvert();
            lc.setLeadId(lead.Id);
            lc.setConvertedStatus('Closed - Converted'); // Ensure this status exists in your org
            lc.setDoNotCreateOpportunity(false); // Set to true if you don't want to create an Opportunity
            leadConverts.add(lc);
        }
    }

    if (!leadConverts.isEmpty()) {
        List<Database.LeadConvertResult> results = Database.convertLead(leadConverts, false); // false for non-AllOrNone
        for (Database.LeadConvertResult result : results) {
            if (!result.isSuccess()) {
                System.debug('Lead Conversion Failed: ' + result.getErrors());
            }
        }
    }
}

```

#### 43. Write a trigger on contact to prevent duplicate records based on Contact Email & Contact Phone.

```

trigger PreventDuplicateContacts on Contact (before insert, before update) {
    // Collect all email and phone values from incoming records
    Set<String> emailSet = new Set<String>();
    Set<String> phoneSet = new Set<String>();

    for (Contact con : Trigger.new) {
        if (String.isNotBlank(con.Email)) {
            emailSet.add(con.Email.toLowerCase()); // Ensure case-insensitivity
        }
        if (String.isNotBlank(con.Phone)) {
            phoneSet.add(con.Phone);
        }
    }

    // Query existing Contacts with matching email or phone
    Map<String, Id> emailToIdMap = new Map<String, Id>();
    Map<String, Id> phoneToIdMap = new Map<String, Id>();

    if (!emailSet.isEmpty() || !phoneSet.isEmpty()) {
        for (Contact existingCon : [
            SELECT Id, Email, Phone
            FROM Contact
            WHERE Email IN :emailSet OR Phone IN :phoneSet
        ]) {
            if (String.isNotBlank(existingCon.Email)) {
                emailToIdMap.put(existingCon.Email.toLowerCase(), existingCon.Id);
            }
            if (String.isNotBlank(existingCon.Phone)) {
                phoneToIdMap.put(existingCon.Phone, existingCon.Id);
            }
        }
    }

    // Check for duplicates within Trigger.new and against database
    for (Contact con : Trigger.new) {
        if (Trigger.isInsert || (Trigger.isUpdate && con.Email != Trigger.oldMap.get(con.Id).Email)) {
            if (String.isNotBlank(con.Email) && emailToIdMap.containsKey(con.Email.toLowerCase())) {
                con.addError('Duplicate Email: A contact with this email already exists.');
            }
        }
        if (Trigger.isInsert || (Trigger.isUpdate && con.Phone != Trigger.oldMap.get(con.Id).Phone)) {
            if (String.isNotBlank(con.Phone) && phoneToIdMap.containsKey(con.Phone)) {
                con.addError('Duplicate Phone: A contact with this phone number already exists.');
            }
        }
    }
}

```

44. Write a trigger, only the system admin user should be able to delete the task.

```

trigger RestrictTaskDeletion on Task (before delete) {
    // Get the profile name of the current user
    String profileName = [SELECT Profile.Name FROM User WHERE Id = :UserInfo.getUserId()].Profile.Name;

    // Check if the profile is not System Administrator
    if (profileName != 'System Administrator') {
        for (Task task : Trigger.old) {
            task.addError('You do not have permission to delete Tasks. Only System Administrators can delete Tasks.');
        }
    }
}

```

45. Upload any pdf le into Document rst. Send an email as an attachment to the lead email Id when a lead is created.



```

trigger UpdateAccountTotalOnContactUpdate on Contact (after update) {
    // Collect all related Account IDs
    Set<Id> accountIds = new Set<Id>();
    for (Contact con : Trigger.new) {
        if (con.AccountId != null) {
            accountIds.add(con.AccountId);
        }
    }

    // Query Opportunities grouped by Account
    Map<Id, Decimal> accountOpportunitySums = new Map<Id, Decimal>();
    if (!accountIds.isEmpty()) {
        for (AggregateResult result : [
            SELECT AccountId, SUM(Amount) totalAmount
            FROM Opportunity
            WHERE AccountId IN :accountIds
            GROUP BY AccountId
        ]) {
            accountOpportunitySums.put((Id)result.get('AccountId'), (Decimal)result.get('totalAmount'));
        }
    }

    // Prepare Account updates
    List<Account> accountsToUpdate = new List<Account>();
    for (Id accountId : accountIds) {
        Account acc = new Account(Id = accountId);
        acc.Total_Opportunity_Amount__c = accountOpportunitySums.get(accountId) != null
            ? accountOpportunitySums.get(accountId)
            : 0;
        accountsToUpdate.add(acc);
    }

    // Perform DML update
    if (!accountsToUpdate.isEmpty()) {
        update accountsToUpdate;
    }
}

```

46. Create an asset when create an OpportunityLineItem with associated Account.

```

trigger CreateAssetOnOpportunityLineItemInsert on OpportunityLineItem (after insert) {
    List<Asset> assetsToInsert = new List<Asset>();

    for (OpportunityLineItem oli : Trigger.new) {
        // Ensure OpportunityLineItem has a valid related Opportunity and Account
        if (oli.OpportunityId != null) {
            Opportunity opp = [SELECT AccountId FROM Opportunity WHERE Id = :oli.OpportunityId LIMIT 1];

            if (opp.AccountId != null) {
                // Create a new Asset
                Asset asset = new Asset(
                    Name = oli.Product2.Name, // Assuming Product Name for the Asset Name
                    AccountId = opp.AccountId, // Associate Asset with Account
                    OpportunityLineItemId_c = oli.Id, // Custom field to relate Asset with OpportunityLineItem
                    Quantity = oli.Quantity,
                    UnitPrice = oli.UnitPrice,
                    Status = 'Active', // Set an appropriate status
                    InstallDate = Date.today(), // Example of setting an install date
                    Product2Id = oli.Product2Id // Relate Asset with Product
                );
                assetsToInsert.add(asset);
            }
        }
    }

    // Insert the Assets
    if (!assetsToInsert.isEmpty()) {
        insert assetsToInsert;
    }
}

```

47. Add a eld Multi-select Picklist on Account and Opportunity as well and add values A,B,C,D,F. Now if we update an Opportunity with this multiselect value Account should also update with the same picklist values.

```

trigger SyncMultiSelectPicklist on Opportunity (after update) {
    Map<Id, String> accountPicklistUpdates = new Map<Id, String>();

    for (Opportunity opp : Trigger.new) {
        Opportunity oldOpp = Trigger.oldMap.get(opp.Id);

        // Check if the multi-select picklist field has changed
        if (opp.MultiSelectPicklist__c != oldOpp.MultiSelectPicklist__c && opp.AccountId != null) {
            accountPicklistUpdates.put(opp.AccountId, opp.MultiSelectPicklist__c);
        }
    }

    // Query accounts to update
    if (!accountPicklistUpdates.isEmpty()) {
        List<Account> accountsToUpdate = new List<Account>();
        for (Id accountId : accountPicklistUpdates.keySet()) {
            Account acc = new Account(
                Id = accountId,
                MultiSelectPicklist__c = accountPicklistUpdates.get(accountId)
            );
            accountsToUpdate.add(acc);
        }

        // Perform update
        if (!accountsToUpdate.isEmpty()) {
            update accountsToUpdate;
        }
    }
}

```

48. Once an Opportunity line item is created, insert a quotation also.

```

trigger CreateQuoteOnOpportunityLineItemInsert on OpportunityLineItem (after insert) {
    Map<Id, Opportunity> opportunityMap = new Map<Id, Opportunity>();
    List<Quote> quotesToInsert = new List<Quote>();

    // Collect all Opportunity IDs from the inserted OpportunityLineItems
    for (OpportunityLineItem oli : Trigger.new) {
        if (oli.OpportunityId != null) {
            opportunityMap.put(oli.OpportunityId, null);
        }
    }

    // Query related Opportunity records to use their details in the Quote
    if (!opportunityMap.isEmpty()) {
        for (Opportunity opp : [
            SELECT Id, Name, AccountId, CloseDate
            FROM Opportunity
            WHERE Id IN :opportunityMap.keySet()
        ]) {
            opportunityMap.put(opp.Id, opp);
        }
    }

    // Create Quote records for each OpportunityLineItem
    for (OpportunityLineItem oli : Trigger.new) {
        Opportunity relatedOpp = opportunityMap.get(oli.OpportunityId);
        if (relatedOpp != null) {
            Quote quote = new Quote(
                Name = 'Quote for ' + relatedOpp.Name,
                OpportunityId = relatedOpp.Id,
                AccountId = relatedOpp.AccountId,
                Pricebook2Id = oli.PricebookEntry.Pricebook2Id, // Assuming Pricebook2Id should match
                ExpirationDate = relatedOpp.CloseDate.addDays(30), // Example of setting expiration date
                Status = 'Draft' // Default status for the new Quote
            );
            quotesToInsert.add(quote);
        }
    }

    // Insert the Quote records
    if (!quotesToInsert.isEmpty()) {
        insert quotesToInsert;
    }
}

```

49. Write a trigger on the Opportunity line item when a line item deletes delete an opportunity as well.

```

trigger DeleteOpportunityOnLineItemDelete on OpportunityLineItem (after delete) {
    // Collect Opportunity IDs from the deleted OpportunityLineItems
    Set<Id> opportunityIds = new Set<Id>();
    for (OpportunityLineItem oli : Trigger.old) {
        if (oli.OpportunityId != null) {
            opportunityIds.add(oli.OpportunityId);
        }
    }

    // Query Opportunities with remaining line items
    Map<Id, Opportunity> opportunitiesToDelete = new Map<Id, Opportunity>();
    if (!opportunityIds.isEmpty()) {
        // Count remaining line items per Opportunity
        for (AggregateResult ar : [
            SELECT OpportunityId, COUNT(Id) totalCount
            FROM OpportunityLineItem
            WHERE OpportunityId IN :opportunityIds
            GROUP BY OpportunityId
        ]) {
            Id oppId = (Id) ar.get('OpportunityId');
            Integer totalCount = (Integer) ar.get('totalCount');
            if (totalCount == 0) {
                opportunitiesToDelete.put(oppId, null);
            }
        }

        // Query details of Opportunities to delete
        opportunitiesToDelete.putAll([SELECT Id FROM Opportunity WHERE Id IN :opportunitiesToDelete.keySet()]);
    }

    // Delete Opportunities with no remaining line items
    if (!opportunitiesToDelete.isEmpty()) {
        delete opportunitiesToDelete.values();
    }
}

```

50. Write a trigger on Account when an account is update when account type changes send an email to all its contacts that your account information has been changed. Subject: Account Update Info Body: Your account information has been updated successfully. Account Name: XYZ.

```

trigger SendEmailOnAccountTypeChange on Account (after update) {
    // Collect Account IDs where the Type field has changed
    Set<Id> accountIdsToNotify = new Set<Id>();

    for (Account acc : Trigger.new) {
        Account oldAcc = Trigger.oldMap.get(acc.Id);
        if (acc.Type != oldAcc.Type) {
            accountIdsToNotify.add(acc.Id);
        }
    }

    // Query Contacts for the affected Accounts
    if (!accountIdsToNotify.isEmpty()) {
        List<Messaging.SingleEmailMessage> emails = new List<Messaging.SingleEmailMessage>();

        // Query related Contacts
        List<Contact> contacts = [
            SELECT Email, Account.Name
            FROM Contact
            WHERE AccountId IN :accountIdsToNotify AND Email != null
        ];

        // Compose and add emails to the list
        for (Contact contact : contacts) {
            Messaging.SingleEmailMessage email = new Messaging.SingleEmailMessage();
            email.setToAddresses(new List<String>{contact.Email});
            email.setSubject('Account Update Info');
            email.setPlainTextBody('Your account information has been updated successfully.\n\n' +
                'Account Name: ' + contact.Account.Name);
            emails.add(email);
        }

        // Send the emails
        if (!emails.isEmpty()) {
            Messaging.sendEmail(emails);
        }
    }
}

```

===== ALL THE BEST =====

