Phone Trigger for NPC

Prepared by: Manasa Korkanti

8 April 2024

Due to the increased integrations available in Salesforce, proper formatting of phone numbers is more important than ever. This allows us to utilise dialling functionality directly from the record view.

Use cases

A few use cases of number formatting

- Australian Landline numbers
    - Area code with the phone number 0312345678 formatted to +61 3 1234 5678
    - Area code with the phone number 03 03 1234 5678 formatted to +61 3 1234 5678
- Australian mobile numbers
    - 0412345678 formatted to +61 412 345 678
    - 61412345678 formatted to +61 412 345 678
- Australian Toll-Free numbers will not include country code.
    - 611300123456 formatted to 1300 123 456

Solution

- An Apex class (PhoneNoUtilities) that has a single static method where you can pass the parameters and test it.
- An Apex class for each of the standard objects (Account, Contact, Lead) that utilises the 'PhoneNoUtilities' Class to format the numbers.
- Finally, a trigger Apex for all three objects to fire the trigger after insertion and after updating to format the numbers as per our requirement.

This formatting style can be used for any custom objects requiring similar formatting by implementing an Apex class for the custom object and utilising the PhoneNoUtilities class along with a trigger on the custom object.

PhoneNoUtilities Apex Class

The first Apex class to be written should be PhoneNoUtilities, which contains a single method that does not instantiate any objects of the class.

The method should accept a string phone number (e.g., "04 123 456 78"), a string variable countryCode to store the value '+61', a string variable newPhone to store the phone number after modifications, and an empty string ph to store the final output.

```
String countryCode='+61';
String newPhone=phone;
String ph='';
try{
    phone = phone.deleteWhitespace().replaceAll('\\D','').replaceFirst( '^0+', '');
    if (phone.startsWith('61')){
        phone = phone.replaceFirst( '61', '');
    }
```

We used the try and catch block to catch any errors or exceptions if the try block failed to execute the code.

The first step of the code is formatting the number to remove any spaces, or special characters, starting with zero and 61 to be replaced with an empty string.

```
//phone number like 1800123456
if ((ph.startsWith('1800') || ph.startsWith('1300'))  && ph.length()==10){
    ph = ph.substring(0, 4)+' '+ph.substring(4, 7)+' '+ph.substring(7, 10);
            //1800          +     123             +      456
    newPhone = ph;  //1800 123 456
```

Toll-free numbers will be formatted without the country code that starts with 1300 and 1800.

When the phone number starts with 4, indicating Australian mobile numbers, we format it accordingly.

```
} else if (ph.startswith('2') || ph.startswith('3')|| ph.startswith('7')|| ph.startswith('8')){
    // New code to handle the rare occasion that someone enters a number like (08) 08 1234 5678
    if (ph.length()==11){
        String testAC=ph.subString(1,3);
        // Only do this if there is an Australian area code
        if (testAC.equals('02') || testAc.equals('03') || testAC.equals('07') || testAC.equals('08'))
        {
            ph=ph.substring(2);
        } else {
            return newPhone;
        }
    }
    ph = countryCode+' '+ph.substring(0, 1)+' '+ph.substring(1, 5)+' '+ph.substring(5, 9);
```

Finally, if the phone numbers have Australian landline codes such as 02, 03, 04, or 07, we format them accordingly. We have also included handling for rare instances like the country code being repeated twice, as in "(08) 0812345678", to format the number appropriately. Once the formatting is completed, we exit the method, handling any exceptions in the catch block.

AccPhNoFormatter Apex Class

The second step is to write an apex class logic to handle our triggers. Let's consider this for the account object where the salesforce org has personal accounts enabled.

```
public class AccPhNoFormatter {
    public static Boolean isFirstTime = true;
```

We have created a property named 'isFirstTime' and initialised it to true. We utilise this property in the trigger to ensure that the trigger logic executes only once per transaction. This helps prevent continuous recursion caused by trigger firing repeatedly. The trigger updates the phone format according to the logic defined in the Account Apex class.

```
public static void accountPhoneValidation(List<Account> AccountList){
    list<Account> updateAccount = new list<Account>();
```

The method accountPhoneValidation accepts a list of accounts, which is stored in the AccountList parameter. Another list named update account is used to store the updated accounts that have been formatted according to the specified conditions.

```
try{
    for (Account acc : AccountList){
        Boolean changeMade=false;
        if (acc.Phone != null){
            acc.Phone=PhoneNoUtilities.formatNumber(acc.Phone);
            changeMade=true;
        }
        if(acc.PersonMobilePhone!=null){
            acc.PersonMobilePhone=PhoneNoUtilities.formatNumber(acc.PersonMobilePhone);
            changeMade=true;
        }
        if(acc.PersonHomePhone!=null){
            acc.PersonHomePhone=PhoneNoUtilities.formatNumber(acc.PersonHomePhone);
            changeMade=true;
        }
        if(acc.PersonOtherPhone!=null){
            acc.PersonOtherPhone=PhoneNoUtilities.formatNumber(acc.PersonOtherPhone);
            changeMade=true;
        }
        if (changeMade){
            updateAccount.add(acc);
        }
```

We utilise a try-catch block to handle any errors that may occur within the try block. The provided code illustrates the types of fields available in person account-enabled orgs, and we apply the formatting logic to each phone-type field. You have the flexibility to exclude or include any fields based on your preferences.

In the first step, we initialise a Boolean parameter called changeMade and set it to false. This allows us to track accounts where changes have been made.

Next, we check if each phone-type field is not null. If this condition is true, we invoke the method and class from the PhoneNoUtilities class to apply formatting to the corresponding phone field. After formatting, we set changeMade to true.

Once formatting is complete for all phone field types, we check if there are any accounts with changeMade set to true, and add them to the updateAccounts list.

TriggerOnAccount

The third step is to fire a trigger to specify when to format the numbers. For this, we write TriggerOnAccount and consider after insert and after update of the fields in the Account object.

```
trigger TriggerOnAccount on Account (after insert, after update) {
    List<Account> accountList = new List<Account>();
```

This line defines the trigger named TriggerOnAccount on the Account object. The trigger fires after records are inserted or updated and declare an empty list of Account records named accountList.

```
try {
    // Query all Account records from Trigger.new
    accountList = [SELECT Id, Phone, PersonHomePhone, PersonOtherPhone, PersonMobilePhone, IsPersonAccount
                   FROM Account
                   WHERE Id IN :Trigger.new];
```

Queries Account records from the Trigger.new context variable. Selects specific fields such as Id, Phone, PersonHomePhone, PersonOtherPhone, PersonMobilePhone, and IsPersonAccount. We consider IsPersonAccount to verify if the account type is a Person account.

```
if (!accountList.isEmpty() && AccPhNoFormatter.isFirstTime) {
    AccPhNoFormatter.isFirstTime = false;
```

Check if the accountlist is not empty and isFirstTime true, the property we considered in AccPhNoFormatter so the trigger can fire. The static variable AccPhNoformatter.isFirstTime is set to false to block from recurring and fire only one time per account.

```
// Separate standard accounts and person accounts
List<Account> standardAccounts = new List<Account>();
List<Account> personAccounts = new List<Account>();
for (Account acc : accountList) {
    if (acc.IsPersonAccount) {
        personAccounts.add(acc);
    } else {
        standardAccounts.add(acc);
    }
```

Now we need to consider the trigger context for both standard accounts and person accounts. Iterates through the queried account list and separates standard accounts from person accounts based on the IsPersonAccount field. Adds standard accounts to the standardAccounts list and person accounts to the personAccounts list.

```
        // Validate phone numbers for standard accounts
        if (!standardAccounts.isEmpty()) {
            AccPhNoFormatter.accountPhoneValidation(standardAccounts);
        }

        // Validate phone numbers for person accounts
        if (!personAccounts.isEmpty()) {
            AccPhNoFormatter.accountPhoneValidation(personAccounts);
        }
    }
```

Validates phone numbers for both standard accounts and person accounts using a method called accountPhoneValidation in the AccPhNoFormatter class. Calls the accountPhoneValidation method for both standard and person accounts separately.

The final step is to write the test class for AccNoPhFormatter Apex class and the coding is always done on the sandbox or scratch orgs and tested rigorously. The code can only be deployed to the production org if the code coverage for the classes is at least 75% or above.

```
@isTest
public class TestAccPhNoFormatter {
    @isTest
    private static void testInsertAccount() {
```

This above code declares a test class named TestAccPhNoFormatter with the @isTest annotation, indicating that it contains test methods. This line declares a test method named testInsertAccount. The method is annotated with @isTest, indicating that it's a test method.

```
List<Account> Acct = new List<Account>();
Account c1= new Account ( NAME='Account ABC', Phone='(61)245783214.');
Acct.add(c1);
Account c2 = new Account (LASTNAME='Peter', Phone='61-490-490-479.',PersonMobilePhone='0478879913',
PersonHomePhone='0323444444',PersonOtherPhone='234567893');
Acct.add(c2);
Account c3 = new Account (LASTNAME='Parker', Phone='61-1300907987',PersonMobilePhone='1800234512',
PersonHomePhone='61 425566772',PersonOtherPhone='08 0812343456');
Acct.add(c3);
Insert cont;
```

The account variable is considered to store the list of accounts that is initially empty. We have created our test class to hold a standard account (Account c1) where the account name is mandatory. Account c2 and Account c3 are person accounts as you can see Lastname is mandatory and the type of fields you have are person account fields. We added the created accounts to the empty list variable Acct.

```
Account A3= [SELECT NAME, Phone,PersonMobilePhone,PersonHomePhone,
            PersonOtherPhone FROM Account where Id=:c1.Id];
Account A4= [SELECT LASTNAME, Phone,PersonMobilePhone,PersonHomePhone,
            PersonOtherPhone FROM Account where Id=:c2.Id];
Account A5= [SELECT LASTNAME, Phone,PersonMobilePhone,PersonHomePhone,
            PersonOtherPhone FROM Account where Id=:c3.Id];
System.assertEquals('+61 2 4578 3214',A3.phone);
System.assertEquals('+61 490 490 479',A4.phone);
System.assertEquals('+61 478 879 913',A4.PersonMobilePhone);
System.assertEquals('+61 3 2344 4444',A4.PersonHomePhone);
System.assertEquals('+61 2 3456 7893',A4.PersonOtherPhone);
System.assertEquals('1300 907 987',A5.phone);
System.assertEquals('1800 234 512',A5.PersonMobilePhone);
System.assertEquals('+61 425 566 772',A5.PersonHomePhone);
System.assertEquals('+61 8 1234 3456',A5.PersonOtherPhone);
```

Queries the inserted Account records (A3, A4, A5) from the database based on their Ids in the test case created Accounts. Asserts that the formatted phone numbers match the expected values for each Account record.

Similar way you can use the PhoneNoUtilities class and write an apex class to handle the logic and triggers to fire after insert, after update for any standard or custom objects in salesforce org.