Person Accounts

If you create an application that references contacts, you need to be aware of a curious entity called a “person account”. Person accounts only exist on organizations for which they are specifically enabled. They are not enabled by default, and in order to enable them you have to file a case, and convince Salesforce support that you really know what you are doing and that you understand that the conversion can’t be reversed. Once the person accounts feature is enabled on an organization, it can never be disabled.

Of course, the easiest way to enable person accounts during development is to specify it as a feature on a newly created scratch org. You do this by adding the “personaccount” feature to the scratch org definition file.

A person account is differentiated from a regular account by the account record type. Person accounts have the following characteristics with regards to Apex programming:

- Each person account has a “shadow” contact object. The ID of that object can be retrieved using the PersonContactID field on the account object.
- You can access standard contact fields for person accounts two ways – by querying the underlying contact and accessing the field as you would a normal contact field, or by accessing the field on the account using a special name that typically consists of the contact field name preceded by the word “Person”. Thus, the contact Email field can be retrieved on the account using the field name PersonEmail. The contact fields FirstName, LastName and Salutation do not have this prefix.
- You can access custom contact fields for the person account object two ways – by querying the underlying contact and accessing the field as you would a normal contact field, or by accessing the field on the account using a special name that consists of the contact field name ending with the suffix __pc instead of __c.
- Contact triggers do not fire on person accounts. Only account triggers fire on these accounts, even if you change a field on the underlying Contact object.

If the package you are developing does not reference the contact object, you can probably ignore person accounts. Otherwise, you should design your code with them in mind, as packages that use contacts probably won't work correctly with person accounts without additional work.

Your first step in supporting person accounts will be to create a scratch org with the PersonAccounts feature specified in its configuration file, or create a separate developer org and request that it have person accounts enabled.

Next, you can implement a couple of helpful functions to work with person accounts. The first is a static function to let your application determine if it is running on an organization with person accounts enabled.

```
private static Set<string> accountFields = null;

public static Boolean isPersonAccountOrg()
{
   if(accountFields==null) accountFields =
      Schema.Sobjecttype.Account.fields.getMap().keyset();
   return AccountFields.contains('personcontactid');
}
```

The isPersonAccountOrg field caches the set of account fields so that you can efficiently verify person account fields.

Another function that can prove useful maps contact field names to their equivalent person account name:

```
// Map from contact field to account field
public static String getPersonAccountAlias(String fieldName)
{
   fieldName = fieldname.toLowerCase();// Case insensitive

   // Unchanged - FirstName, LastName, etc.
   if(accountFields.contains(fieldName)) return fieldName;

   // Replace aliased __c with __pc
   fieldName = fieldName.replace('__c', '__pc');
   if(accountFields.contains(fieldName)) return fieldname;

   if(accountFields.contains('person' + fieldName))
      return ('person' + fieldName);

   return null;
}
```

You should always access person account fields using dynamic Apex, and should only access them on account objects where the PersonAccountID field is true.

Person Account Triggers

To see the challenges you can face when working with person accounts, consider the following very simple example – a Contact trigger that sets the Level2__c field based on the LeadSource field as described in the following pseudocode:

```
If the LeadSource is ‘Web’ or ‘Phone Inquiry’,
   set Level2__c to ‘Primary’
Otherwise set Level2__c to ‘Secondary’.
```

If you were not concerned about person accounts, you would probably implement this as follows[1]:

```
trigger OnContact1 on Contact (before update, before insert)
{
   PersonAccountSupport.processContactTrigger(
      trigger.isBefore, trigger.new, trigger.oldMap);
}
```

and in a class (in this case class PersonAccountSupport):

```
public static void processContactTrigger(Boolean isBefore,
```

[1] This particular sample code is located in class PersonAccountSupport because it is related to this section of the book. Despite the name of the class, the sample code at this point only applies to contacts.

```
   List<Contact> newList, Map<ID, Contact> oldMap)
{
   for(Contact ct: newList)
   {
      if(ct.LeadSource=='Web' || ct.LeadSource=='Phone Inquiry')
         ct.Level2__c = 'Primary';
      else ct.Level2__c = 'Secondary';
   }
}
```

The test code for this class might be as follows (see class TestPersonAccount in the sample code):

```
static testMethod void testWithContacts()
{
    List<Contact> contacts =
      TestDiagnostics2.createContacts('patst', 3);
    contacts[0].LeadSource='Web';
    contacts[1].LeadSource='Phone Inquiry';
    contacts[2].LeadSource='Other';
    Test.StartTest();
    insert contacts;
    Test.StopTest();
    // On query we'll get the same 3 contacts
    Map<ID, Contact> contactMap = new Map<ID, Contact>(
      [Select ID, Level2__c from Contact]);
    system.assertEquals(
      contactMap.get(contacts[0].id).Level2__c,'Primary');
   system.assertEquals(
      contactMap.get(contacts[1].id).Level2__c,'Primary');
   system.assertEquals(
      contactMap.get(contacts[2].id).Level2__c,'Secondary');
}
```

How do we extend this to support a person account organization? There are a number of challenges:

- Not every account on a person account organization is, in fact, a person account. They are distinguished by record type.
- When creating or updating a person account, only the account trigger fires.

Your first thought might be to use a before trigger and to update the fields on the person account. For testing purposes we'll use an account trigger that processes both before and after triggers, though we'll only look at the before functionality for the moment.

```
trigger OnAccount1 on Account (after insert, after update,
   before insert, before update)
{

   PersonAccountSupport.processAccountTrigger(
      trigger.isBefore, trigger.new, trigger.oldMap);
}
```

Let's look at how the processAccountTrigger handles the before trigger case.

```
public static void processAccountTrigger(Boolean isBefore,
   List<Account> newList, Map<ID, Account> oldMap)
{
   if(!isPersonAccountOrg()) return;

   if(isBefore)
   {
      // Using before approach
      String leadSourceAlias = getPersonAccountAlias('LeadSource');
      String levelAlias = getPersonAccountAlias('Level2__c');
      for(Account act: newList)
      {
         if(leadSourceAlias!=null && levelAlias!=null &&
            (Boolean)act.get('IsPersonAccount'))
         {  // Will only be valid on person accounts
            if(act.get(leadSourceAlias)=='Web' ||
               act.get(leadSourceAlias)=='Phone Inquiry')
               act.put(levelAlias,'Primary');
               else act.put(levelAlias,'Secondary');
            }
        }
     }
}
```

This approach suffers from two problems. First, it duplicates the functionality in the processContactTrigger code. That means that you have to support and maintain the same exact functionality in two different places, taking care to update both if any changes are required.

That may not mean much on a simple example like this, but for a complex example, this type of support or maintenance can result in significant costs over the lifetime of the software.

The second problem is more serious. Remember – this is a package, and your development and deployment organizations will almost certainly not be person account organizations. That means your code must meet code coverage requirements on a non-person account organization in order to be uploaded as a package. As it stands, the code coverage provided on the PersonAccountSupport class by the TestPersonAccount unit tests stands at 42%.

Figure 1 illustrates the current architecture.

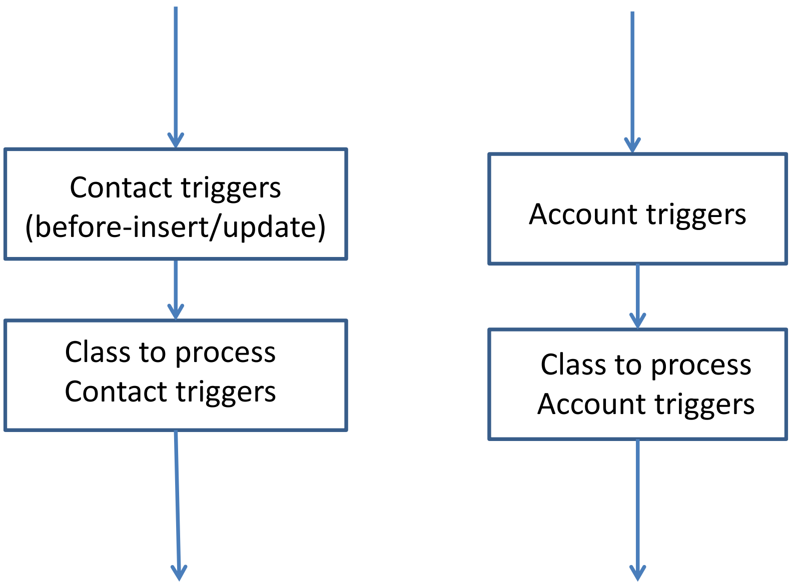
Contact triggers (before-insert/update)
Class to process Contact triggers
Account triggers
Class to process Account triggers

Figure 1 – Separate person account code is hard to maintain and test

The first problem is quite easy to solve, and demonstrates yet another reason why you should always implement your trigger code in a class.

Remember that every person account has a shadow contact object. Why not update that object directly?

You can't do this during a before trigger, but you can do it during an after trigger. The approach is as follows:

- Use an after-insert/update trigger on the Account object.
- If it is a person account organization, build a list of any accounts that have a PersonContactID value (these are the person accounts).
- Query for these contacts.
- Call the class method used to process contact triggers.
- Update the contacts (being careful not to process the resulting account trigger).

This approach is illustrated in Figure 2.

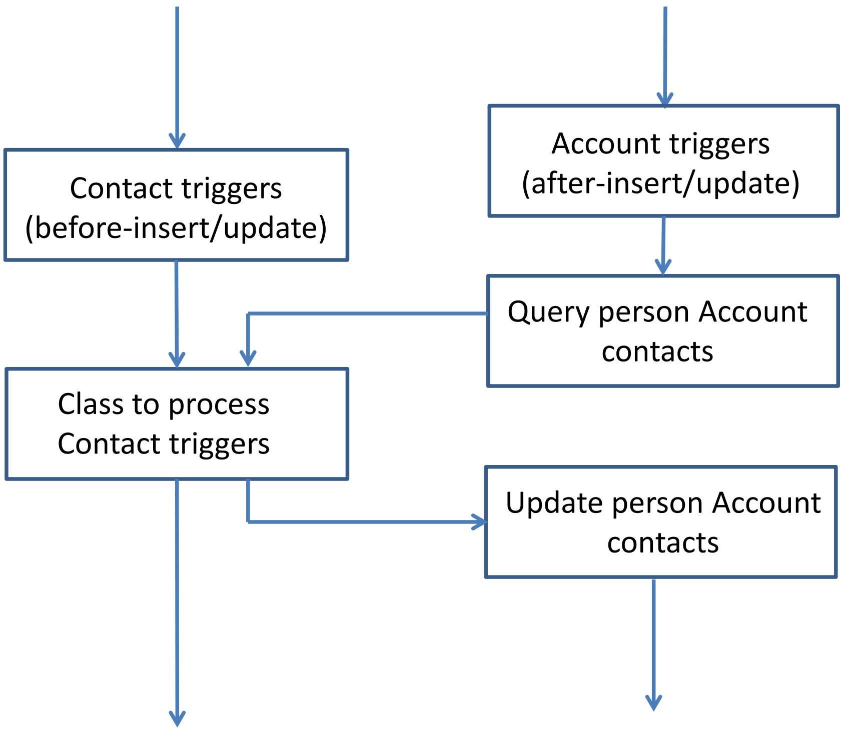
Account triggers (after-insert/update)
Contact triggers (before-insert/update)
Query person Account contacts
Class to process Contact triggers
Update person Account contacts

Figure 2 – Leveraging contact functionality when using person accounts

The way to address the code coverage issue is to use static variables to allow most of your code to run even on a non-person account organization.

Here's what the resulting processAccountTrigger function looks like with both of these solutions implemented.

```
@TestVisible public static Boolean fakePersonAccountDuringTest = false;

public static List<ID> fakePersonContactIDs = null;

private static Boolean updatingPersonContact = false;

public static void processAccountTrigger(
   Boolean isBefore, List<Account> newList, Map<ID, Account> oldMap)
{
    if(!isPersonAccountOrg() && !fakePersonAccountDuringTest ||
        updatingPersonContact) return;

    if(!isBefore)
    {
        // Better approach can work on after trigger
        Set<ID> personContactIds = new Set<ID>();
        for(Integer x = 0; x<newList.size(); x++)
        {
            if(fakePersonAccountDuringTest ||
                newlist[x].get('PersonContactID')!=null )
                personContactIds.add(
                    (fakePersonAccountDuringTest)? fakePersonContactIDs[x]:
                    (ID)newList[x].get('PersonContactID') );
        }
        if(personContactIds.size()==0) return;
        Map<ID, Contact> personContacts = new Map<ID, Contact>(
                [Select ID, LeadSource, Level2__c
                from Contact where ID in :personContactIds]);
        processContactTrigger(true, personcontacts.values(),
            personcontacts);
        updatingPersonContact = true;
        update personcontacts.values();
        updatingPersonContact = false;
    }
}
```

There are a number of subtle issues to note in this code. First, the fakePersonAccountDuringTest constant is tested before the check for the personContactID field. Apex processes conditionals in left to right order, so testing fakePersonAccountDuringTest first prevents the test for the personContactID field (which would cause a missing field error) when testing.

Remember to gate the entire function with code that validates that you are even running on a person account org, and exit if not (except when testing). Never forget that non-person accounts will also raise this trigger.

The updatePersonContact flag is used to prevent the function from being processed again when it is retriggered by the update to the underlying Contact object. If you were using the centralized trigger model described in Chapter 6 of the book, you wouldn't necessarily need to do this, depending on how you chose to handle triggers that occur during a trigger handler DML operation.

The personContactIds set is used by the test code to provide fake shadow contacts in order to obtain code coverage.

To conclude, let's look at the test code:

```
static testMethod void testWithAccounts() {
   List<Contact> contacts = createContacts('patst', 3);
   List<Account> accounts = createAccounts('patest', 3);
   contacts[0].LeadSource='Web';
   contacts[1].LeadSource='Phone Inquiry';
   contacts[2].LeadSource='Other';
   insert contacts;
   String leadSourceAlias = PersonAccountSupport.
      getPersonAccountAlias('LeadSource');
   String levelAlias = PersonAccountSupport.
      getPersonAccountAlias('Level2__c');
   if(PersonAccountSupport.isPersonAccountOrg())
   {
      accounts[0].put(leadSourceAlias,'Web');
      accounts[1].put(leadSourceAlias,'Phone Inquiry');
      accounts[2].put(leadSourceAlias,'Other');
   }
   else {
      PersonAccountSupport.fakePersonContactIDs = new List<ID>();
      PersonAccountSupport.fakePersonAccountDuringTest = true;
      for(Contact ct: contacts)
         PersonAccountSupport.fakePersonContactIDs.add(ct.id);
   }
   Test.StartTest();
   insert accounts;
   Test.StopTest();
   // We'll get the same 3 contacts
```

```
    Map<ID, Contact> contactMap = new Map<ID, Contact>(
         [Select ID, Level2__c from Contact]);
    system.assertEquals('Primary',
       contactMap.get(contacts[0].id).Level2__c);
    system.assertEquals('Primary',
       contactMap.get(contacts[1].id).Level2__c);
    system.assertEquals('Secondary',contactMap.get(
       contacts[2].id).Level2__c);
    if(PersonAccountSupport.isPersonAccountOrg())
    {

       Map<ID, Account> accountMap = new Map<ID, Account>(
          (List<Account>)Database.query('Select ID, ' +
          levelAlias +' from Account'));
       system.assertEquals('Primary', accountMap.get(
          accounts[0].id).get(levelAlias));
       system.assertEquals('Primary', ccountMap.get(
          accounts[1].id).get(levelAlias));
       system.assertEquals('Secondary',accountMap.get(
          accounts[2].id).get(levelAlias));
    }
}
```

The unit test does create person accounts if possible, so it actually does validate functionality on person account orgs.

But the real magic in this approach is that it resolves our code coverage issues on non-person account orgs, bringing the code coverage up to 93% from the previous 42%. That means that it can be successfully deployed on non-person account orgs.