poplar_oeaddlne Documentation

Release 1.3.0

Chris Binckly, 2665093 Ontario Inc.

Jul 22, 2020

Contents

CHAPTER 1

Walkthrough

1.1 Building the View Script

A view script it is. It will attach to the `OE0500` Order Headers detail view. On insert it needs to:

1. Check if the operation was successful.
2. Check if the item matches "A1-103/0".
3. Generate a new record in the view.
4. Populate the record with the Item "A1-105/0" Qty 1.
5. Save the record.

1.1.1 Import accpac and Scaffold

All scripts start the same, import all members of `accpac.py`:

```
from accpac import *
```

Note: Importing `*` is generally not a good idea, you don't know what is being pulled into the namespace. You can import only the things you need, plus some extras, with a little trial and error.

Now we need to decide which events to listen to, do we need to check the record before or after insert? Before the operation occurs we can't know whether it was successful and it may fail. If we were to act before a successful operation we may create new lines when the triggering line failed.

`onAfter` seems the correct choice. Add the appropriate method with the correct signature. The *`poplar_oeaddlne.resources.all_view_calls`* is a good place to start.

```
from accpac import *

def onAfterInsert(result):
    """After updating, if the item is "A1-103/0", insert a new line."""

    # Check if the operation was successful

    # Check if the item matches

    # Generate a new record in the view

    # Populate the record

    # Save the record
    pass
```

1.1.2 Implement the onAfter call

The docs for *poplar_oeaddlne.resources.all_view_calls.onAfterInsert()* tell us that the `result` argument contains 0 if the insert succeeded and that the function doesn't need to return.

Once we have checked the result, we need to see if the item matches. Use the special view object `me` exposed by `accpac` to access the current record in the view and use `me.get(field)` to retrive the item number inserted.

Field names can be found by looking them up using the Extender View Info Inquiry or using the Sage `accpacViewInfo` utility.

```
from accpac import *

def onAfterInsert(result):
    """After updating, if the item is "A1-103/0", insert a new line."""

    # Check if the operation was successful
    if result != 0:
        return

    # Check if the item matches
    if me.get("ITEM") == "A1-103/0":

        # Generate a new record in the view

        # Populate the record

        # Save the record
```

Generating a new line is generally accomplished by running two operations on the view. The first, `.recordClear()` resets the state of the view. The second, `.recordGenerate()`, creates a new record in the view. Both return 0 when successful.

What should happen if these operations fail? There is always a silent option, but then the user may be confused as to why the line doesn't isn't created when they expect it to.

Extender provides a number of ways to notify. The first is using the `showMessage(str)`, `showWarning(str)`, `showError(str)` method. These put messages on the error stack for Sage to display. They may not be displayed immediately, which can be helpful for situtations where errors may occur in bulk (such as during an import). They also provide levels and a familiar interace.

The second is to use `showMessageBox(str)` which will pop up a dialog immediately. This is generally a better option for things the user needs to know now and for any debugging you need. Show a message box to the user on failure.

```
from accpac import *

def onAfterInsert(result):
    """After updating, if the item is "A1-103/0", insert a new line."""

    # Check if the operation was successful
    if result != 0:
        return

    # Check if the item matches
    if me.get("ITEM") == "A1-103/0":

        # Generate a new record in the view
        rc = me.recordClear()
        rg = me.recordGenerate()

        if rc != 0 or rg != 0:
            showMessageBox("Failed to generate new line.")
            return

        # Populate the record

        # Save the record
```

Now we just need to populate the record and save it. Set fields in the current record by using `.put(field, value)`. Once populated, use `.insert()` to add write it to the database. These operations also return 0 on success.

```
from accpac import *

def onAfterInsert(result):
    """After updating, if the item is "A1-103/0", insert a new line."""

    # Check if the operation was successful
    if result != 0:
        return

    # Check if the item matches
    if me.get("ITEM") == "A1-103/0":

        # Generate a new record in the view
        rc = me.recordClear()
        rg = me.recordGenerate()

        if rc != 0 or rg != 0:
            showMessageBox("Failed to generate new line.")
            return

        # Populate the record
        pi = me.put("ITEM", "A1-105/0")
        pq = me.put("QTYORDERED", 1)

        if pi != 0 or pq != 0:
            showMessageBox("Failed to put values in new line.")
            return
```

(continues on next page)

(continued from previous page)

```
        # Save the record
        sv = me.insert()

        if sv != 0:
            showMessageBox("Failed to save new line.")

    return None
```

And there you have it. A view script that does exactly what we need. Are there any improvements to be had?

1.1.3 Add Parameters

What if the customer wants to change the items? Instead of "A1-103/0" triggering they may want "A1-900/G" to be the trigger. What if they wanted to add a quantity of 5 instead of 1? At present, they'd need to change the script because the items and quantity are hard coded.

View scripts support up to 4 user provided parameters of up to 250 characters, so 1000 characters of arguments to play with. They are exposed by `accpac` as `Parameter1`, `Parameter2`, `Parameter3`, `Parameter4`.

Let's change the script to accept parameters from the user.

```
"""OE0500_oe_add_line.py

Parameters
----------

    - Parameter1: Item number to trigger new line
    - Parameter2: Item number to set in new line
    - Parameter3: Item quantity to set.
"""
from accpac import *
# from accpac import (me,
#                     Parameter1, Parameter2, Parameter3,
#                     showMessageBox, )

def onAfterInsert(result):
    """After updating, if the item is Parameter1, insert a new line."""

    # Check if the operation was successful
    if result != 0:
        return

    # Check if the item matches
    if me.get("ITEM") == Parameter1:

        # Generate a new record in the view
        rc = me.recordClear()
        rg = me.recordGenerate()

        if rc != 0 or rg != 0:
            showMessageBox("Failed to generate new line.")
            return

        # Populate the record
        pi = me.put("ITEM", Parameter2)
```

(continues on next page)

(continued from previous page)

```
        pq = me.put("QTYORDERED", Parameter3)

        if pi != 0 or pq != 0:
            showMessageBox("Failed to put values in new line.")
            return

        # Save the record
        sv = me.insert()

        if sv != 0:
            showMessageBox("Failed to save new line.")

    return None
```

1.1.4 Testing

Time for testing. Fire up your favourite database and install the script in the Extender -> Setup -> Scripts screen.

Configure it by attaching it to the `OE0500` view in the Extender -> Setup -> View Events, Scripts and Workflow.

1.2 Building the Screen Script

A screen script it is. It will attach to the `OE1100` Order Entry screen. On insert of a new detail line, it needs to:

1. Check if the operation was successful.
2. Check if the item matches "A1-103/0".
3. Generate a new record in the view.
4. Populate the record with the Item "A1-105/0" Qty 1.
5. Save the record.

1.2.1 Import accpac and Scaffold

All scripts start the same, import all members of `accpac.py`:

```
from accpac import *
```

Note: Importing `*` is generally not a good idea, you don't know what is being pulled into the namespace. You can import only the things you need, plus some extras, with a little trial and error.

Because this is a screen script, we must create a new instance of the `accpac.UI` class, initialize its parent, and tell it to `.show()` itself. To get a custom UI or change going, create a subclass.

```
from accpac import *

class OeAddLineUI(UI):
    """A UI customization that monitors the order details for a trigger
    item being inserted and adds a new line."""
```

(continues on next page)

(continued from previous page)

```
    def __init__(self):
        super().__init__()
        self.show()

def main(*args, **kwargs):
    ui = OeAddLineUI()
```

When working with screens Extender exposes `accpac.UIDatasource` objects. These act very similar to views. They allow access to the data sources underlying the current screen. Data sources are always composed with one another, making coordinated access easy.

Datasources are opened using their unique `Module Name`. Module names can be found using the RotoID for the program (O/E Order Entry in our case - `OE1100`) and looking up the details in `accpacInfo`.

A little digging reveals that the datasource module id for Order Details is `adsOEORDD`. Open a datasource using `UI.openDataSource(module_id)`.

```
from accpac import *

class OeAddLineUI(UI):
    """A UI customization that monitors the order details for a trigger
    item being inserted and adds a new line."""

    def __init__(self):
        super().__init__()

        # Open the data source
        self.adsOEORDD = self.openDataSource("adsOEORDD")
        self.show()

def main(*args, **kwargs):
    ui = OeAddLineUI()
```

Now we need to decide which events to listen to, do we need to check the record before or after insert? Before the operation occurs we can't know whether it was successful and it may fail. If we were to act before a successful operation we may create new lines when the triggering line failed.

`onAfter` seems the correct choice. Assign a call back function to the `onAfterInsert` attribute of the `adsOEORDD` datasource.

```
from accpac import *

class OeAddLineUI(UI):
    """A UI customization that monitors the order details for a trigger
    item being inserted and adds a new line."""

    def __init__(self):
        super().__init__()

        # Open the data source
        self.adsOEORDD = self.openDataSource("adsOEORDD")

        # Assign the onAfter callback to the *function*
        self.adsOEORDD.onAfterInsert = self.adsOEORDDonAfterInsert

        self.show()
```

(continues on next page)

(continued from previous page)

```
    def adsOEORDDonAfterInsert(self, result):
        """After updating, if the item is "A1-103/0", insert a new line."""
        # Check if the item matches

        # Generate a new record in the view

        # Populate the record

        # Save the record
        pass

def main(*args, **kwargs):
    ui = OeAddLineUI()
```

1.2.2 Implement the onAfter call

The `accpac.UIDatasource.onAfterInsert` callback does not receive arguments and is only triggered on a successful insert. Now we need to see if the item matches. Use the `.get(field)` method on the datasource to get the current value. Field names can be found by looking them up using the Extender View Info Inquiry or using the Sage `accpacViewInfo` utility.

```
from accpac import *

class OeAddLineUI(UI):
    """A UI customization that monitors the order details for a trigger
    item being inserted and adds a new line."""

    def __init__(self):
        super().__init__()

        # Open the data source
        self.adsOEORDD = self.openDataSource("adsOEORDD")

        # Assign the onAfter callback to the *function*
        self.adsOEORDD.onAfterInsert = self.adsOEORDDonAfterInsert

        self.show()

    def adsOEORDDonAfterInsert(self, result):
        """After updating, if the item is "A1-103/0", insert a new line."""

        # Check if the item matches
        if self.adsOEORDD.get("ITEM") == "A1-103/0":

        # Generate a new record in the view

        # Populate the record

        # Save the record
        pass

def main(*args, **kwargs):
    ui = OeAddLineUI()
```

Generating a new line is generally accomplished by running two operations on the datasource. The first, `.recordClear()` resets the state of the datasource. The second, `.recordGenerate()`, creates a new record in the datasource. Both return 0 when successful.

What should happen if these operations fail? There is always a silent option, but then the user may be confused as to why the line doesn't isn't created when they expect it to.

Extender provides a number of ways to notify. The first is using the `showMessage(str)`, `showWarning(str)`, `showError(str)` method. These put messages on the error stack for Sage to display. They may not be displayed immediately, which can be helpful for situtations where errors may occur in bulk (such as during an import). They also provide levels and a familiar interace.

The second is to use `showMessageBox(str)` which will pop up a dialog immediately. This is generally a better option for things the user needs to know now and for any debugging you need. Show a message box to the user on failure.

```
from accpac import *

class OeAddLineUI(UI):
    """A UI customization that monitors the order details for a trigger
    item being inserted and adds a new line."""

    def __init__(self):
        super().__init__()

        # Open the data source
        self.adsOEORDD = self.openDataSource("adsOEORDD")

        # Assign the onAfter callback to the *function*
        self.adsOEORDD.onAfterInsert = self.adsOEORDDonAfterInsert

        self.show()

    def adsOEORDDonAfterInsert(self, result):
        """After updating, if the item is "A1-103/0", insert a new line."""

        # Check if the item matches
        if self.adsOEORDD.get("ITEM") == "A1-103/0":

            # Generate a new record in the view
            rc = self.adsOEORDD.recordClear()
            rg = self.adsOEORDD.recordGenerate()

            if rc != 0 or rg != 0:
                showMessageBox("Failed to generate new line.")
            return

            # Populate the record

            # Save the record
        pass

def main(*args, **kwargs):
    ui = OeAddLineUI()
```

Now we just need to populate the record and save it. Set fields in the current record by using `.put(field, value)`. Once populated, use `.insert()` to add write it to the database. These operations also return 0 on success.

```
from accpac import *

class OeAddLineUI(UI):
    """A UI customization that monitors the order details for a trigger
    item being inserted and adds a new line."""

    def __init__(self):
        super().__init__()

        # Open the data source
        self.adsOEORDD = self.openDataSource("adsOEORDD")

        # Assign the onAfter callback to the *function*
        self.adsOEORDD.onAfterInsert = self.adsOEORDDonAfterInsert

        self.show()

    def adsOEORDDonAfterInsert(self, result):
        """After updating, if the item is "A1-103/0", insert a new line."""

        # Check if the item matches
        if self.adsOEORDD.get("ITEM") == "A1-103/0":

            # Generate a new record in the view
            rc = self.adsOEORDD.recordClear()
            rg = self.adsOEORDD.recordGenerate()

            if rc != 0 or rg != 0:
                showMessageBox("Failed to generate new line.")
            return

            # Populate the record
            pi = self.adsOEORDD.put("ITEM", "A1-105/0")
            pq = self.adsOEORDD.put("QTYORDERED", 1)

            if pi != 0 or pq != 0:
                showMessageBox("Failed to put values in new line.")
                return

            # Save the record
            sv = self.adsOEORDD.insert()

            if sv != 0:
                showMessageBox("Failed to save new line.")

        return None

def main(*args, **kwargs):
    ui = OeAddLineUI()
```

And there you have it. A view script that does exactly what we need. Are there any improvements to be had?

1.2.3 Add Parameters

What if the customer wants to change the items? Instead of "A1-103/0" triggering they may want "A1-900/G" to be the trigger. What if they wanted to add a quantity of 5 instead of 1? At present, they'd need to change the script because the items and quantity are hard coded.

View scripts support up to 4 user provided parameters of up to 250 characters, so 1000 characters of arguments to play with. They are exposed by `accpac` as `Parameter1`, `Parameter2`, `Parameter3`, `Parameter4`.

Unfortunately, screens scripts do not allow user provided parameters. We will see another solution later, when we add custom tables.

1.2.4 Testing

Screen scripts must follow a specific naming convention and have a particular structure. The Roto ID of the screen being customized must be present in one of the first two `.` delineated parts of the filename:

- `OE1100.COMPANY.script_name.py`: good
- `COMPANY.OE1100.script_name.py`: good
- `COMPANY-script_name-OE1100.py`: bad

The file must also start with a single line comment that includes the Roto ID:

```
# OE1100
# ... The rest of my script.
```

With the file named correctly and the comment in place, time for testing. Fire up your favourite database and install the script in the Extender -> Setup -> Scripts screen.

There is no need to configure screen scripts, if they are installed whey will be loaded when the screen is loaded. It is best to restart the Sage desktop after installing but before testing your customization.

1.3 Making an Extender Module

Now that we have a working version, it is time to think about delivering it to the client. In almost all cases, the correct way to distribute your scripts is in a module, even if you only have one.

A module allows you to group scripts together, track versions and author data, and autoamtically configure scripts at module import time. Always wrap your tools in modules.

1.3.1 Module format

Extender modules use an INI style file that includes the metadata, scripts, and configuration in one file.

Create the new *OEADDLNE* module:

```
[MODULE]
id=OEADDLNE
name=OEADDLNE
desc=After an order line with itam A11030 is entered, enter one for A11050
company=2665093 Ontario Inc.
version=0.1.0
website=https://2665093.ca/

[SCRIPT]
FILENAME=OEADDLNE_OE1100_oe_add_line.py
>>> SCRIPT >>>
# OE1100
from accpac import *
```

(continues on next page)

(continued from previous page)

```
...
<<< SCRIPT <<<
```

That is all it takes. Install modules from the Extender -> Setup -> Modules screen. The script will be installed automatically, and when screen scripts are installed they are configured by default.

1.3.2 View Scripts

View scripts get an extra clause in the module file that defines the scripts configuration: which view it attaches to, which parameters it accepts, etc.

Had we been deploying out view script, we could have made a module that included an additional `VIEWSCRIPT` block:

```
[MODULE]
id=OEADDLNE
name=OEADDLNE
desc=After an order line with itam A11030 is entered, enter one for A11050
company=2665093 Ontario Inc.
version=0.1.0
website=https://2665093.ca/

[SCRIPT]
FILENAME=OEADDLNE_OE0500_oe_add_line.py
>>> SCRIPT >>>
from accpac import *
...
<<< SCRIPT <<<

[VIEWSCRIPT]
VIEWID=OE0500
UNIQID=2019050100000004
ACTIVE=1
ORDER=0
SCRIPT=OEADDLNE_OE0500_oe_add_line.py
P1=A1-103/0
P2=A1-105/0
P3=1
```

Now that we have a module, let's get back to the question of how to make the mappings user configurable.

1.4 Custom Tables for Item Mappings

The screen script solution does what we need but is limited in that the item mappings are hard-coded. The user cannot add new mappings themselves and if there are many then keeping them in the source becomes unwieldy.

Enter Extender's Custom Tables. They make it easy to store, retrieve, and update tables specific to your customization in the database.

Assume for now that each Item will only ever map to one other. We can define a database table, `OEADDLNE.VIITMMAP`, to allow the user to manage the mappings.

1.4.1 Designing the Table Schema

Before creating a table it needs a schema. The correct fields and keys (which are also indexes) need to be identified. Every item only maps to one other, we can use the trigger item as a key.

When thinking about keys, think about how you access the data. If you always use two fields together to retrieve from a table, be sure to add a compound key on those fields.

We need the same fields as we had view script parameters, so the following should do:

```
Table name: VIITMMAP
Fields:
    - TRIGITEM (str): item that triggers the new line
    - NEWITEM (str): the item to insert
    - QTYORDERED (bcd): the quantity ordered
Keys:
    - (TRIGITEM, )
```

1.4.2 Creating the Table Schema

The table schema will be embedded in the module file. You have two options for creating it: interactively with the Custom Tables tool or writing it by hand in the modules file.

1.4.2.1 Using Custom Tables

This is the best way to start. Once you get the hang of it you can craft them by hand.

To use the Custom Tables tool, start by creating a module:

1. Navigate to Extender -> Setup -> Modules
2. Insert a new row using the module name (`OEADLNE`)

Define the custom table:

1. Navigate to Extender -> Setup -> Custom Tables
2. Set the table name to `OEADDLNE.VIITMMAP`
3. Define the fields. Use `accpacViewInfo` or the extender enquiries to find the correct field names, finder information, and description lookups.
4. Add the key in the `Keys` tab.
5. Save the table.

Now that the table is defined you can export the module from the Modules screen and the table definition will be included.

1.4.2.2 Crafting by Hand

After a while you'll be able to write table definitions by hand. To start, this is what is automatically generated on module export:

```
[MODULE]
id=OEADDLNE
name=OEADDLNE
desc=Add an OE Order Line after a particular item is added.
```

(continues on next page)

(continued from previous page)

```
company=2665093 Ontario Inc
version=0.1.0
website=https://2665093.ca

[TABLE]
name=OEADDLNE.VIITMMAP
dbname=VIITMMAP

[FIELD1]
field=TRIGITEM
datatype=1
size=24
mask=%24C
desc=Trigger item number
ftable=IC0310
ffield=ITEMNO
lookup=FMTITEMNO

[FIELD2]
field=NEWITEM
datatype=1
size=24
mask=%24C
desc=Item number to insert
ftable=IC0310
ffield=ITEMNO
lookup=FMTITEMNO

[FIELD3]
field=QTYORDERED
datatype=6
size=10
decimals=5
desc=Quantity

[KEY1]
KEYNAME=TRIGITEM
FIELD1=TRIGITEM
allowdup=0
```

Simply add the `SCRIPT` block and we have a full module.

1.4.3 Adding Entries to the Table

Adding entries is easy using the Extender -> Setup -> Custom Table Editor. Simply open the editor, open the `OEADDLNE.VIITMMAP` table, and start adding.

For now, try to add the item we know about, A1-103/0 -> A1-105/0@1.

1.4.4 Adding the Lookup to the Script

Now we just need to replace out hard coded values with a lookup from our custom table. Custom tables are accessed through the view layer. Instead of opening them based on the View ID (i.e. `VI0107`), always access them by module qualified table name. There is no guarantee that your table will always have the same roto, so don't use it.

Because `TRIGITEM` is our key field, we will use it to look up the mapping. The lookup will go something like this:

```
view = openView("OEADDLNE.VIITMMAP")
view.recordClear()
view.put("TRIGITEM", itemno)
r = view.read()
```

Extender functions return 0 on success, so if `r` is 0 then there is a mapping for `itemno` and the view has read it in. Any other return indicates that `itemno` is not a trigger item and no action is required.

```
from accpac import *

class OeAddLineUI(UI):
    """A UI customization that monitors the order details for a trigger
    item being inserted and adds a new line."""

    def __init__(self):
        super().__init__()

        # Open the data source
        self.adsOEORDD = self.openDataSource("adsOEORDD")

        # Open the custom table.
        self.viitmmap = openView("OEADDLNE.VIITMMAP")

        # Assign the onAfter callback to the *function*
        self.adsOEORDD.onAfterInsert = self.adsOEORDDonAfterInsert

        self.show()

    def get_trigger(self, itemno):
        """Get the trigger information for this item number from the table.

        :param itemno: item number to loopkup trigger information for
        :type itemno: str
        :rtype: list
        :returns:
            - If the item number is not in the map table: []
            - if the item number is in the table: [newitem, qtyordered]
        """

        rc = self.viitmmap.recordClear()
        pt = self.viitmmap.put("TRIGITEM", itemno)
        r  = self.viitmmap.read()

        if r != 0 or pt != 0 or rc != 0:
            return []

        return [self.viitmmap.get("NEWITEM"),
                self.viitmmap.get("QTYORDERED"), ]



    def adsOEORDDonAfterInsert(self, result):
        """After updating, if the item is "A1-103/0", insert a new line."""

        # Check if the item matches an item in the custom table map :
        # trigger will be [newitem, qty] if it does, [] (False) if it
```

(continues on next page)

(continued from previous page)

```
        # doesn't.
        trigger = self.get_trigger(me.get("ITEM"))
        if trigger:

            # Generate a new record in the view
            rc = me.recordClear()
            rg = me.recordGenerate()

            if rc != 0 or rg != 0:
                showMessageBox("Failed to generate new line.")
            return

            # Populate the record
            pi = me.put("ITEM", trigger[0])
            pq = me.put("QTYORDERED", trigger[1])

            if pi != 0 or pq != 0:
                showMessageBox("Failed to put values in new line.")
                return

            # Save the record
            sv = me.insert()

            if sv != 0:
                showMessageBox("Failed to save new line.")

        return None

def main(*args, **kwargs):
    ui = OeAddLineUI()
```

Done. The custom table is now intergated with the script. The user can add and manage as many mappings as they'd like with the custom table editor.

That's all for now, just a few things to close out...

It is time to build the code. These narrated walkthroughs cover the development process for *poplar_oeaddlne.OE0500_oe_add_line* and *poplar_oeaddlne.OE1100_oe_add_line* Python scripts, putting the final script into an Extender module, and upgrading the module and script to lookup the item information from a custom table.

CHAPTER 2

Add OE Line - View Script

This view script is intended to trigger when a new line is added to an order. When triggered, it tries to write a new line to the order with an item defined in the parameters.

This script will always fail, causing an error to be raised from the view. It demonstrates a case in which the new script approach is not suitable, use a Screen script (*poplar_oeaddlne.OE1100_oe_add_line*).

Insert a new OE line following the insert of a line with a particular item.

Part of an introductory training to Extender. Learn more at https://poplar_addoelne.rtfd.io.

Parameters:

- Parameter1: Item number to trigger insert after.
- Parameter2: Item number to insert
- Parameter3: Item

author: Chris Binckly
email: chris@2665093.ca

`poplar_oeaddlne.OE0500_oe_add_line.`**`onAfterInsert`**(*result*)

After a line with item `Parameter1` is entered, insert line with item `Parameter2`.

Triggered after an insert of an OE Detail line. If the insert was successful and the item in the line is the same as that provided in `Parameter1`, a new line is added with:

- `LINETYPE`: 1 - standard item line
- `ITEM`: `Parameter2`
- `QTYORDERED`: `Parameter3`

poplar_oeaddlne.OE0500_oe_add_line.**onOpen**()
onOpen of the script, take no action.

CHAPTER 3

Add OE Line - Screen Script

This screen script doesn't make any customizations to the UI but monitors the data sources opened by the screen for changes. When a new detail line is added, the screen adds another line with a fixed item.

Unlike in the view script solution, *poplar_oeaddlne.OE0500_oe_add_line*, we cannot pass parameters to the screen script so the item mapping is hard coded.

Insert a new OE line following the insert of a line with a particular item.

Part of an introductory training to Extender. Learn more at https://poplar_addoelne.rtfd.io.

author: Chris Binckly
email: chris@2665093.ca

class poplar_oeaddlne.OE1100_oe_add_line.**AddOeLneUI**
A UI that monitors the order details and inserts a new line.

This UI class makes no changes to the OE1100 screen, it simply monitors and writes to the Data Sources connected to it.

adsOEORDDonAfterInsert()
If the item inserted was ITEM, add a new line.

Raises None

Return type None

poplar_oeaddlne.OE1100_oe_add_line.**ITEM = 'A1-103/0'**
The item number that triggers the new line.

poplar_oeaddlne.OE1100_oe_add_line.**NEW_ITEM_LINETYPE = 1**
The item line type to insert.

poplar_oeaddlne.OE1100_oe_add_line.**NEW_ITEM_NUM = 'A1-105/0'**
The item number to insert.

poplar_oeaddlne.OE1100_oe_add_line.**NEW_ITEM_QTY = 10**
The item quantity to insert.

poplar_oeaddlne.OE1100_oe_add_line.**main**(*args*)
Executed by VI when the screen is loaded. Create our UI.

CHAPTER 4

Resources

4.1 All View Calls

A helpful reference for all supported calls for view scripts.

This script shows all the entry points that you can use when writing a script for a view. It doesn't actually do anything useful.

Copyright 2015 Orchid Systems

Updated 2019 2665093 Ontario Inc.

poplar_oeaddlne.resources.all_view_calls.**onAfterAttributes**(*e*)
Called after the attributes of a view field are changed.

Parameters **result** (*int*) – result of the attribute operation, 0 on success.

Return type None

poplar_oeaddlne.resources.all_view_calls.**onAfterCancel**(*result*)
Called after a view is canceled.

Parameters **result** (*int*) – result of the cancel operation, 0 on success.

Return type None

poplar_oeaddlne.resources.all_view_calls.**onAfterClose**(*result*)
Called after a view is closed.

Parameters **result** (*int*) – result of the close operation, 0 on success.

Return type None

poplar_oeaddlne.resources.all_view_calls.**onAfterCompose**(*event*)
Called after a view is composed.

Parameters **event** (accpac.viewComposeArgs) – compose event information.

Returns must return accpac.Continue or the view will not load.

Return type int

poplar_oeaddlne.resources.all_view_calls.**onAfterDelete**(*result*)
Called after a view is deleted.

Parameters **result** (*int*) – result of the delete operation, 0 on success.

Return type None

poplar_oeaddlne.resources.all_view_calls.**onAfterDirty**(*result*)
Called after a view is marked dirty.

Parameters **result** (*int*) – result of the mark dirty operation, 0 on success.

Return type None

poplar_oeaddlne.resources.all_view_calls.**onAfterFetchLock**(*result*)
Called after a view releases a lock for a fetch.

Parameters **result** (*int*) – result of the fetch lock operation, 0 on success.

Return type None

poplar_oeaddlne.resources.all_view_calls.**onAfterGet**(*event*)
Called after a field view is retrieved.

Params event undocumented.

Return type None

poplar_oeaddlne.resources.all_view_calls.**onAfterInit**(*result*)
Called after a view is initialized.

Parameters **result** (*int*) – result of the initialization, 0 on success.

Return type None

poplar_oeaddlne.resources.all_view_calls.**onAfterInsert**(*result*)
Called after a view is inserted.

Parameters **result** (*int*) – result of the insert operation, 0 on success.

Return type None

poplar_oeaddlne.resources.all_view_calls.**onAfterPost**(*result*)
Called after a view is posted.

Parameters **result** (*int*) – result of the post operation, 0 on success.

Return type None

poplar_oeaddlne.resources.all_view_calls.**onAfterProcess**(*result*)
Called after a view finishes processing.

Parameters **result** (*int*) – result of processing, 0 on success.

Return type None

poplar_oeaddlne.resources.all_view_calls.**onAfterPut**(*event*)
Called after a value is put in the view.

Params event undocumented.

Return type None

poplar_oeaddlne.resources.all_view_calls.**onAfterRead**(*result*)
Called after a record is read through the view.

Parameters **result** (*int*) – result of the read operation, 0 on success.

Return type None

poplar_oeaddlne.resources.all_view_calls.**onAfterReadLock**(*result*)
Called after a view releases a lock for a read.

Parameters **result** (*int*) – result of the read lock operation, 0 on success.

Return type None

poplar_oeaddlne.resources.all_view_calls.**onAfterRecordClear**(*result*)
Called after a view is cleared.

Parameters **result** (*int*) – result of the clear operation, 0 on success.

Return type None

poplar_oeaddlne.resources.all_view_calls.**onAfterUnlock**(*result*)
Called after a view is unlocked.

Parameters **result** (*int*) – result of the unlock operation, 0 on success.

Return type None

poplar_oeaddlne.resources.all_view_calls.**onAfterUpdate**(*result*)
Called after a view is updated.

Parameters **result** (*int*) – result of the update operation, 0 on success.

Return type None

poplar_oeaddlne.resources.all_view_calls.**onAfterVerify**(*result*)
Called after a view is verified.

Parameters **result** (*int*) – result of the verify operation, 0 on success.

Return type None

poplar_oeaddlne.resources.all_view_calls.**onBeforeAttributes**(*event*)
Called before the attributes of a view field are changed.

Params event undocumented.

Returns `accpac.Continue` or `accpac.Abort`

Return type int

poplar_oeaddlne.resources.all_view_calls.**onBeforeCancel**()
Called before a record is canceled through the view.

Returns `accpac.Continue` or `accpac.Abort`

Return type int

poplar_oeaddlne.resources.all_view_calls.**onBeforeClose**()
Called before a view is closed.

Return type int

Returns `accpac.Continue` to allow the close, `accpac.Abort` to disallow it.

poplar_oeaddlne.resources.all_view_calls.**onBeforeCompose**(*event*)
Called before a view is composed.

Parameters **event** (`accpac.viewComposeArgs`) – compose event information.

Returns must return `accpac.Continue` or the view will not load.

Return type int

Get the handles for composed views to make it easier to work with related data.

A script attached to the Order Header `OE0520` view can access the composed Order Details `OE0522` view. Get a handle on before compose, store it in the `__main__` (script) namespace, and use it in other calls.

```
oeordd = None
def onBeforeCompose(event):
    if len(event.views):
        oeordd = event.views[0]

def onAfterPut(result):
    # If the put succeeded and a field = value, add a new order line
    if result == 0 and me.get("FIELD") == "VALUE":
        r = oeordd.recordClear()
        r = oeordd.recordGenerate()
        r = oeordd.put(...)
        ...
```

poplar_oeaddlne.resources.all_view_calls.**onBeforeDelete**()
Called before a record is deleted through the view.

Returns `accpac.Continue` or `accpac.Abort`

Return type int

poplar_oeaddlne.resources.all_view_calls.**onBeforeDirty**()
Called before a view is marked dirty.

Returns `accpac.Continue` or `accpac.Abort`

Return type int

poplar_oeaddlne.resources.all_view_calls.**onBeforeFetch**()
Called before a record is fetched through the view.

Returns `accpac.Continue` or `accpac.Abort`

Return type int

poplar_oeaddlne.resources.all_view_calls.**onBeforeFetchLock**()
Called before a view locks for a fetch.

Returns `accpac.Continue` or `accpac.Abort`

Return type int

poplar_oeaddlne.resources.all_view_calls.**onBeforeGet**(*event*)
Called before the value of a field is retrieved through the view.

Params event undocumented.

Returns `accpac.Continue` or `accpac.Abort`

Return type int

poplar_oeaddlne.resources.all_view_calls.**onBeforeInit**()
Called before a view is initialized.

Returns `accpac.Continue` or `accpac.Abort`

Return type int

poplar_oeaddlne.resources.all_view_calls.**onBeforeInsert**()
Called before a record is inserted into the view.

Returns accpac.Continue or accpac.Abort

Return type int

poplar_oeaddlne.resources.all_view_calls.**onBeforePost**()
Called before a record is posted through the view.

Returns accpac.Continue or accpac.Abort

Return type int

poplar_oeaddlne.resources.all_view_calls.**onBeforeProcess**()
Called before a view runs processing.

Returns accpac.Continue or accpac.Abort

Return type int

poplar_oeaddlne.resources.all_view_calls.**onBeforePut**(*event*)
Called before the value of a field is put in a view field.

Params event undocumented.

Returns accpac.Continue or accpac.Abort

Return type int

poplar_oeaddlne.resources.all_view_calls.**onBeforeRead**()
Called before a record is read from the view.

Returns accpac.Continue or accpac.Abort

Return type int

poplar_oeaddlne.resources.all_view_calls.**onBeforeReadLock**()
Called before a view locks for a read.

Returns accpac.Continue or accpac.Abort

Return type int

poplar_oeaddlne.resources.all_view_calls.**onBeforeRecordClear**()
Called before a view is cleared.

Returns accpac.Continue or accpac.Abort

Return type int

poplar_oeaddlne.resources.all_view_calls.**onBeforeUnlock**()
Called before a view is unlocked.

Returns accpac.Continue or accpac.Abort

Return type int

poplar_oeaddlne.resources.all_view_calls.**onBeforeUpdate**()
Called before a record is updated through the view.

Returns accpac.Continue or accpac.Abort

Return type int

poplar_oeaddlne.resources.all_view_calls.**onBeforeVerify**()
Called before a view is verified.

Returns accpac.Continue or accpac.Abort

Return type int

poplar_oeaddlne.resources.all_view_calls.**onCommitRecord**(*op*)
Called when a record in the revision list has been posted.

Parameters **op** (*int*) – operation code, one of:

- 1 = insert
- 2 = update
- 3 = delete
- 4 = move

Return type None

poplar_oeaddlne.resources.all_view_calls.**onOpen**()
Called when a view is opened.

Return type int

Returns `accpac.Continue` to enable the script, 0 to disable, anything else stops the view loading.

poplar_oeaddlne.resources.all_view_calls.**onOpenReadOnly**()
Called when a view is opened in readonly mode.

Return type int

Returns `accpac.Continue` to enable the script, 0 to disable, anything else stops the view loading.

poplar_oeaddlne.resources.all_view_calls.**onRevisionCancelled**()
Called when the revision list is cancelled, changes are discarded.

Return type None

poplar_oeaddlne.resources.all_view_calls.**onRevisionPosted**()
Called when a revision has been posted.

Return type None

Some references to get started with:

- The presentation: Extender Introduction.
- Sample scripts from Orchid.
- Sample scripts from 2665093
- Orchid Knowledgebase

There are sample customizations for the `OE1100` screen from Orchid in `poplar_oeaddlne.resources.examples` as well as a searchable, annotated version of *All View Calls*.

This package contains the content for a quick introductory training session to Orchid Extender. The training was delivered to a team endeavouring to build their own scripts to manage OE Orders.

The content was developed and delivered by 2665093 Ontario Inc. If you'd like to receive custom Orchid Extender training or hands on help solving a problem using Extender, send us an email.

Download the git repository and follow along with the presentation.

Once you've reached the `Build It` section, continue to *Building the View Script*.

Python Module Index

p

Index

P