Article
Fabio Goncalves · Apr 18, 2017 9m read

Tracking Data Changes - Audit Log - 1 of 2

Introduction

The common requirement in many applications is logging of data changes in a database - which data has changed, who changed them and when (audit logging). There are many articles about this question and there are different approaches on how to do that in Caché.

I'm sharing a mechanism that can help you implement your framework to track and record data changes. This mechanism creates a trigger through an "objectgenarator" method once your persistent class inherits from the "Audit Abstract Class" (Sample.AuditBase). Since your persistent class inherits Sample.AuditBase, when you compile your persistent class the trigger for auditing changes will be generated automatically.

Audit Class

This is the class where the changes will be recorded.

```
Class Sample.Audit Extends %Persistent
{
  Property Date As %Date;

  Property UserName As %String(MAXLEN = "");

  Property ClassName As %String(MAXLEN = "");

  Property Id As %Integer;

  Property Field As %String(MAXLEN = "");

  Property OldValue As %String(MAXLEN = "");

  Property NewValue As %String(MAXLEN = "");
}
```

Audit Abstract Class

This is the abstract class that your persistent class will inherits from. This class contains the trigger method (objectgenerator) that knows how to identify which field has been modified, who changed that, what are the old and new values, etc., besides writing the changes in the audit table (Sample.Audit).

```
Class Sample.AuditBase [ Abstract ]
{

Trigger SaveAuditAfter [ CodeMode = objectgenerator, Event = INSERT/UPDATE, Foreach =
row/object, Order = 99999, Time = AFTER ]
{
  #dim %compiledclass As %Dictionary.CompiledClass
  #dim tProperty As %Dictionary.CompiledProperty
  #dim tAudit As Sample.Audit

  Do %code.WriteLine($Char(9)_"; get username and ip adress")
  Do %code.WriteLine($Char(9)_"Set tSC = $$$OK")
```

```
  Do %code.WriteLine($Char(9)_"Set tUsername = $USERNAME")

  Set tKey = ""
  Set tProperty = %compiledclass.Properties.GetNext(.tKey)
  Set tClassName = %compiledclass.Name

  Do %code.WriteLine($Char(9)_"Try {")
  Do %code.WriteLine($Char(9,9)_"; Check if the operation is an update - %oper = UPDATE")
  Do %code.WriteLine($Char(9,9)_"if %oper = ""UPDATE"" { ")

  While tKey '= "" {
    set tColumnNbr = $Get($$$EXTPROPsqlcolumnnumber($$$pEXT,%classname,tProperty.Name))
    Set tColumnName = $Get($$$EXTPROPsqlcolumnname($$$pEXT,%classname,tProperty.Name))

    If tColumnNbr '= "" {

      Do %code.WriteLine($Char(9,9,9)_";")
      Do %code.WriteLine($Char(9,9,9)_";")
      Do %code.WriteLine($Char(9,9,9)_"; Audit Field: "_tProperty.SqlFieldName)
      Do %code.WriteLine($Char(9,9,9)_"if {" _ tProperty.SqlFieldName _ "*C} {")
      Do %code.WriteLine($Char(9,9,9,9)_"Set tAudit = ##class(Sample.Audit).%New()")
      Do %code.WriteLine($Char(9,9,9,9)_"Set tAudit.ClassName = """_tClassName_"""")
      Do %code.WriteLine($Char(9,9,9,9)_"Set tAudit.Id = {id}")
      Do %code.WriteLine($Char(9,9,9,9)_"Set tAudit.UserName = tUsername")
      Do %code.WriteLine($Char(9,9,9,9)_"Set tAudit.Field = """_tColumnName_"""")
      Do %code.WriteLine($Char(9,9,9,9)_"Set tAudit.Date = +$Horolog")

      Do %code.WriteLine($Char(9,9,9,9)_"Set tAudit.OldValue = {"_tProperty.SqlFieldName_"*O}")
      Do %code.WriteLine($Char(9,9,9,9)_"Set tAudit.NewValue = {"_tProperty.SqlFieldName_"*N}")

      Do %code.WriteLine($Char(9,9,9,9)_"Set tSC = tAudit.%Save()")
      do %code.WriteLine($Char(9,9,9,9)_"If $$$ISERR(tSC) $$$ThrowStatus(tSC)")

      Do %code.WriteLine($Char(9,9,9)_"}")
    }
    Set tProperty = %compiledclass.Properties.GetNext(.tKey)
  }

  Do %code.WriteLine($Char(9,9)_"}")

  Do %code.WriteLine($Char(9)_"} Catch (tException) {")

  Do %code.WriteLine($Char(9,9)_"Set %msg = tException.AsStatus()")
  Do %code.WriteLine($Char(9,9)_"Set %ok = 0")
  Do %code.WriteLine($Char(9)_"}")

  Set %ok = 1
}

}
```

Data Class (Persistent Class)

This is the user data class that the user (application) make changes, create records, delete records, do whatever you allow him to do. :). In summary, this is usually your %Persistent class.

To start traking and record changes you will need to inherits the Persistent Class from the Abstract Class (Sample.AuditBase).

```
Class Sample.Person Extends (%Persistent, %Populate, Sample.AuditBase)
{
  Property Name As %String [ Required ];

  Property Age As %String [ Required ];

  Index NameIDX On Name [ Data = Name ];
}
```

Testing

Since you have inherited the data class Sample.Person) from the Audit Abstract Class (Sample.AuditBase) you are able to insert data, make changes and look at the changes recorded on the Audit Class (Sample. Audit).

In order to test that you will need to create a Test() class method on the Sample.Person class or any other class of your choice.

```
ClassMethod Test(pKillExtent = 0)
{
   If pKillExtent '= 0 {
  Do ##class(Sample.Person).%KillExtent()
   Do ##class(Sample.Audit).%KillExtent()
   }

   &SQL(INSERT INTO Sample.Person (Name, Age) VALUES ('TESTE', '01'))
   Write "INSERT INTO Sample.Person (Name, Age) VALUES ('TESTE', '01')",!
   Write "SQLCODE: ",SQLCODE,!!!

   Set tRS = $SYSTEM.SQL.Execute("SELECT * FROM Sample.Person")
   Do tRS.%Display()

   &SQL(UPDATE Sample.Person SET Name = 'TESTE 2' WHERE Name = 'TESTE')
   Write !!!
   Write "UPDATE Sample.Person SET Name = 'TESTE 2' WHERE Name = 'TESTE'",!
   Write "SQLCODE:",SQLCODE,!!!

   Set tRS = $SYSTEM.SQL.Execute("SELECT * FROM Sample.Person")
   Do tRS.%Display()

   Quit
}
```

Run the Test() method:

```
 d ##class(Sample.Person).Test(1)
```

Parameter 1 will kill extent from Sample.Person and Sample.Audit classes.

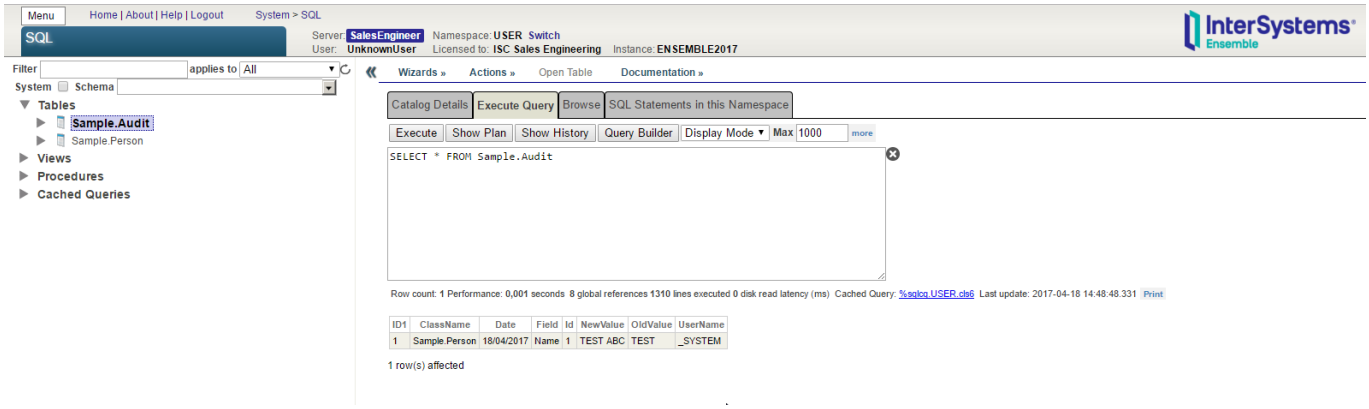
```
d ##class(Sample.Person).Test(1)
INSERT INTO Sample.Person (Name, Age) VALUES ('TEST', '01')
SQLCODE: 0

ID  Age Name
1   01  TEST

1 Rows(s) Affected


UPDATE Sample.Person SET Name = 'TEST ABC' WHERE Name = 'TEST'
SQLCODE:0


ID  Age Name
1   01  TEST ABC

1 Rows(s) Affected
```

The Test class method does the following:

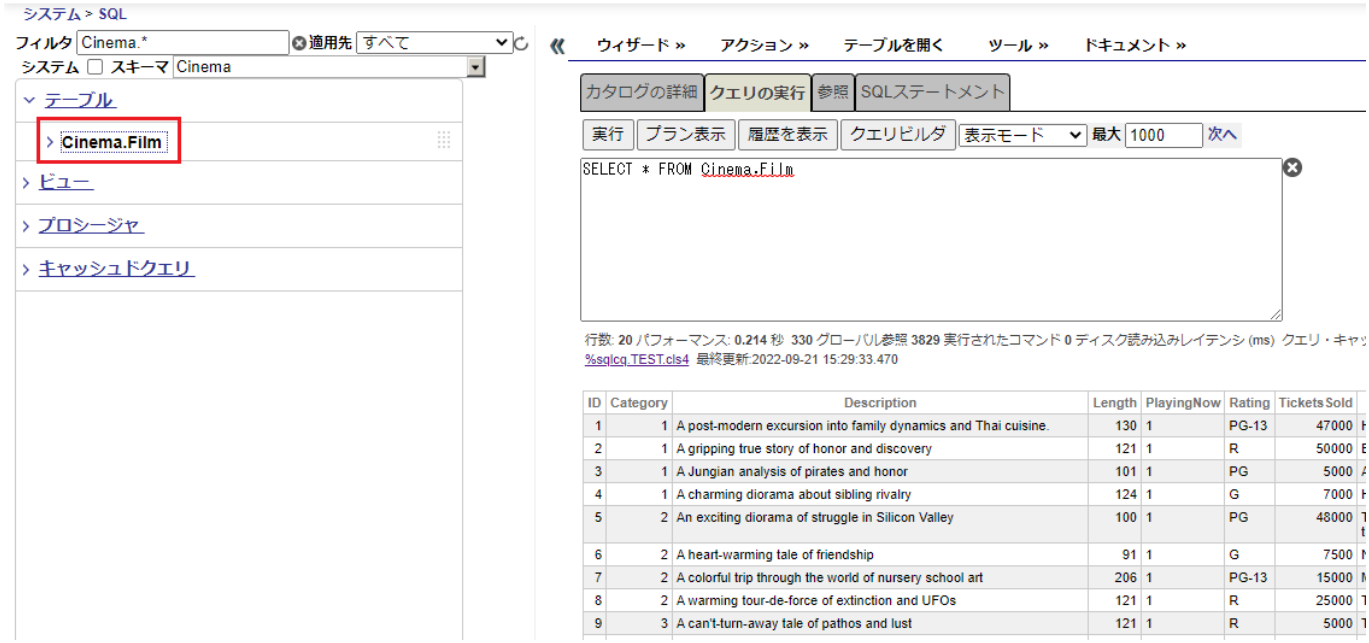
- Insert a new person with name "TEST";
- Shows the insert result;
- Update the person "TEST" to "TEST ABC";
- Show the update result;

Now you can check the audit log table. In order to do that open the System Managment Portal->System Explore->SQL. (Do not forget to switch to your namespace)

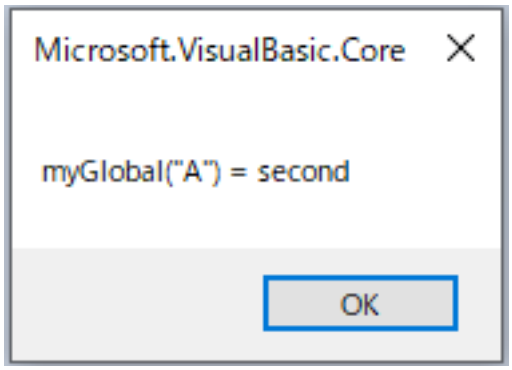
Run the following SQL command and check the results:
SELECT * FROM Sample.Audit



Note that the OldValue is "TEST" and the NewValue is "TEST ABC". From now on you can make your own tests by changing the name of "TEST ABC" to "Your own name" and or change the Age values, for example. See:
UPDATE Sample.Person SET Name = 'Fabio Goncalves' WHERE Name = 'TEST ABC'



ID	Category	Description	Length	PlayingNow	Rating	Tickets Sold
1	1	A post-modern excursion into family dynamics and Thai cuisine.	130	1	PG-13	47000
2	1	A gripping true story of honor and discovery	121	1	R	50000
3	1	A Jungian analysis of pirates and honor	101	1	PG	5000
4	1	A charming diorama about sibling rivalry	124	1	G	7000
5	2	An exciting diorama of struggle in Silicon Valley	100	1	PG	48000
6	2	A heart-warming tale of friendship	91	1	G	7500
7	2	A colorful trip through the world of nursery school art	206	1	PG-13	15000
8	2	A warming tour-de-force of extinction and UFOs	121	1	R	25000
9	3	A can't-turn-away tale of pathos and lust	121	1	R	5000



Generated Code

Considering that you have implemented the auditing mechanism bellow, on your computer launch the Studio (or Atelier), open the Persistent Class (Sample.Person) and inspect the intermediate code generated after compiling Sample.Person class. To do so type Ctrl + Shift + V (View Other Source Code) - inspect .INT. Scroll down up to zSaveAuditAfterExecute label and take a look at the generated code:

```
zSaveAuditAfterExecute(%oper=0,pNew,pOld,pChanged,%ok,%msg)
    ; get username and ip adress
    Set tSC = 1
    Set tUsername = $USERNAME
    Try {
        ; Check if the operation is an update - %oper =  UPDATE
        if %oper = "UPDATE" {
            ;
            ;
            ; Audit Field: Age
            if pChanged(2) {
                Set tAudit = ##class(Sample.Audit).%New()
                Set tAudit.ClassName = "Sample.Person"
                Set tAudit.Id = pNew(1)
                Set tAudit.UserName = tUsername
                Set tAudit.Field = "Age"
                Set tAudit.Date = +$Horolog
                Set tAudit.OldValue = pOld(2)
                Set tAudit.NewValue = pNew(2)
                Set tSC = tAudit.%Save()
                If ('tSC) Throw ##class(%Exception.StatusException).ThrowIfInterrupt(tSC)
            }
            ;
            ;
            ; Audit Field: Name
            if pChanged(3) {
                Set tAudit = ##class(Sample.Audit).%New()
                Set tAudit.ClassName = "Sample.Person"
                Set tAudit.Id = pNew(1)
                Set tAudit.UserName = tUsername
                Set tAudit.Field = "Name"
                Set tAudit.Date = +$Horolog
                Set tAudit.OldValue = pOld(3)
                Set tAudit.NewValue = pNew(3)
                Set tSC = tAudit.%Save()
                If ('tSC) Throw ##class(%Exception.StatusException).ThrowIfInterrupt(tSC)
            }
        }
```

Advantages

It is simple to implement audit logging based on rolling out old data. You do not need additional tables. Maintenance is simple, too. If you decide to remove old data, then it is the matter of one SQL.

If you need to implement audit logging in more tables, just inherits from the Abstract Class (Sample.AuditBase)

Make changes according to your need. e.g.: record changes on Streams.

Record just modified fields. Do not save the entire record changed.

Disadvantages

The problem can be that when data changes, then the whole record is copied, i.e. also data which does not change.

If table person has a column "photo" with binary data (stream) containing the photography then each and every time yhe user changes the picture the role stream is recorded (consuming disk space).

Another disadvantage is that the complexity of each table supporting audit logging increases. You must have in mind all the time that retrieving the records can not be simple. You always have to use the SELECT clause with condition: "...WHERE Status = active" or considering some "DATE INTERVAL"

All data changes are logged in a common table.

Think about transcation an rollbacks.

Aduting is an important requirement for some applications to be efficient. Typically, to determine data changes, application developers must implement a custom tracking method in their applications by using a combination of

triggers, timestamp columns, and additional tables. Creating these mechanism usually involves a lot of work to implement, leads to schema updates, and often carries a high performance overhead. This is a simple example that can help you to start creating your own framework.

Take a look at the next article!

#Best Practices #Object Data Model #ObjectScript #Caché