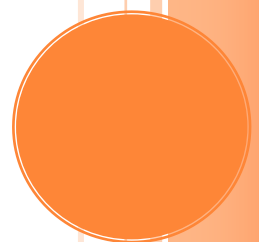


ABAP-SAP S/4.DEBUGGER SCRIPTING: ADVANCED(PERSONAL EXPERIENCE)

Sergio Cannelli

28/11/2023



SAP ABAP Debugger Scripting is a new tool added in SAP Netweaver 7.0 EHP2. It is a feature available in the new ABAP debugger. However, just like some of the other debugger options, you need to check whether the security team has provided access for the debugger script tool.

Most of ABAP programmers regularly use Break-Points & Watch-Points for debugging. These are necessary in an ABAP programmers role to find the bugs in custom developments, analyze the Standard SAP program or find a BADI for that matter.

Designed To Make Debugging ABAP Code Easy

The debugger scripting is a tool is designed to make debugging easy. Sometimes debugging can be very tiresome, especially when debugging SAP standard code is involved. The debugger scripting tool can **sometimes** come to the rescue of ABAP programmers, as it helps automate the process of debugging.

Some of the benefits of Debugger Scripting are:

1. Automation of repeatedly changing the value of some variable in debugging – You can write a script to do all the changes required.
2. Perform any custom trace possible – Even track the call to the ABAP stack. It can be used to trace when new programs are called, as the ABAP stack is updated every time a new program is called.
3. Create custom (conditional) watch-points and break-points. script watch-points & break-point are different from the normal debugger watch-points or break-points. Scripts can be implemented for these to work only when required.
4. Program Tools for standard tasks like checking data consistency, test error handling, etc.
5. It can help make the debugging to be user interactive (pop-up based) where one can enter the value, rather than each time double clicking on the variable to change the value. Since the script can be saved and loaded again, this makes the task for programmer easier to use.

A Debugger script is simply a local Object Oriented ABAP program that is run by the New ABAP Debugger. The script uses the ADI (ABAP Debugger Interface) to automate actions and to add capabilities that otherwise would not be available.

Let's Take A Look At The Debugger Scripting Tool

In the new ABAP debugger there is a separate TAB “Script” .

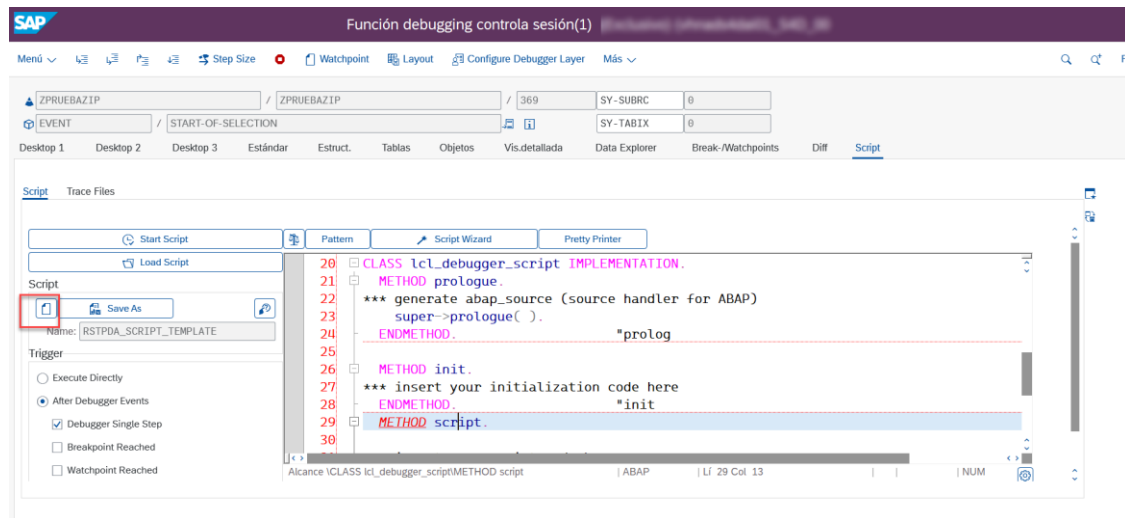



Fig 1

In this TAB, are various options for creating and executing an ABAP script. Let us explore these options.

There is a standard create  button, to create a new script, as per our debugging requirement. Each new script that is created has four methods that can be implemented. These methods are triggered at specific moments and thus it defines their purpose. Also as you can see in Fig 1, all this code is generated automatically whenever a script is created.

A class “**LCL_DEBUGGER_SCRIPT**” is implemented, which has:

Method PROLOGUE

After the script is started, the Debugger framework first calls the PROLOGUE method of the script. The default implementation registers the ABAP_SOURCE instance to the DEBUG_STEP event so that the source code information you get from the ABAP_SOURCE instance is always up-to-date.

Method INIT

INIT method is called only once, directly after the PROLOGUE method. You use this method for initialization. It is also a place to add user interaction, for example, a dialog box, where you ask the user for input, which you need for the script.

Method SCRIPT

SCRIPT method is where we write the Object Oriented ABAP code that executes. This method is called each time the debugger script executes. It has an importing parameter p_trigger, which holds the trigger type that caused this script to execute (Breakpoint, Watchpoint, Single Step, Single Run).

Method END

When the script stops, END method is called.

Besides these custom methods, the Script Super class also includes two already implemented methods :

break

This returns the control to the user. You can stop the script or continue here.

raise_error

This method is used to exit a script, with respective message displayed to the user.

Like the standard ABAP Editor, in Debugging Script Tool also there are buttons for syntax check, include patterns & pretty printer. Besides these there is a button for the Script Wizard, that can be used to generate the code automatically so that the programmer doesn't have to write the entire code, you can just modify the generated code, based on your specific requirement.

The triggering of debugger script can be done in a number of ways.

- It can be Triggered directly.
- It can be triggered when a Breakpoint is encountered while executing the code.
- It can be triggered when a Watchpoint is encountered while executing the code.
- It can be triggered after each debugger step.

Since the debugger script is a local OOPs program that is executed in the new ABAP debugger, it can be easily saved for later use and loaded whenever required. Which removes the task (and hassle) of doing same things again and again as in a normal debugger.

As tracing is a strong component of this tool, a button is specifically provided on the screen to display a trace. This inherently triggers a Transaction code **SAS**, used for viewing a debugger trace.

Let'S Have A Practical Example Of The Use Of Debugger Scripting Tool.

Normally watch-points are used to stop the execution of a program whenever the value of some variable changes or is different from the set value.

As an example we are going to use a new concept in ABAP to order data, which is called VIRTUAL SORT, using this Debugger script tool you can customize your watch-point, to see how it behaves.

Example of Virtual Sort.

```
REPORT ZDEMO_VIRTUAL_SORT.

CLASS DEMO DEFINITION.
  PUBLIC SECTION.
    CLASS-METHODS MAIN.
ENDCLASS.

CLASS DEMO IMPLEMENTATION.
  METHOD MAIN.
    TYPES:
      BEGIN OF LINE,
        COL1 TYPE I,
        COL2 TYPE I,
        COL3 TYPE STRING,
        COL4 TYPE STRING,
      END OF LINE,
      ITAB TYPE STANDARD TABLE OF LINE WITH EMPTY KEY.

    DATA(RND) = CL_ABAP_RANDOM_INT=>CREATE( SEED = + SY-UZEIT
                                              MIN  = 1
                                              MAX  = 10 ).

    DATA(ITAB) = VALUE ITAB( FOR I = 1 UNTIL I > 10
      ( COL1 = RND->GET_NEXT( )
        COL2 = RND->GET_NEXT( )
        COL3 = SUBSTRING(
          VAL = SY-ABCDE
          OFF = RND->GET_NEXT( ) - 1
          LEN = 1 )
        COL4 = SUBSTRING(
          VAL = SY-ABCDE
          OFF = RND->GET_NEXT( ) - 1
          LEN = 1 ) ) ).

    DATA(OUT) = CL_DEMO_OUTPUT=>NEW( ).

    OUT->WRITE( ITAB ).

    OUT->NEXT_SECTION( Virtual Sort by col1, col2, Ascending ).

    DATA(V_INDEX) =
      CL_ABAP_ITAB_UTILITIES=>VIRTUAL_SORT(
        IM_VIRTUAL_SOURCE = VALUE #(
          ( SOURCE = REF #( ITAB )
            COMPONENTS = VALUE #( ( NAME = 'col1' )
                                   ( NAME = 'col2' ) ) ) ).

    OUT->WRITE( V_INDEX ).
```

```

DATA SORTED_TAB TYPE ITAB.
SORTED_TAB = VALUE #( FOR IDX IN V_INDEX ( ITAB[ IDX ] ) ).

DATA(TEST_TAB) = ITAB.
SORT TEST_TAB STABLE BY COL1 COL2.
ASSERT SORTED_TAB = TEST_TAB.

OUT->WRITE( SORTED_TAB ).

OUT->NEXT_SECTION( Virtual Sort by col3, col4, Descending ).

V_INDEX =
  CL_ABAP_ITAB_UTILITIES=>VIRTUAL_SORT(
    IM_VIRTUAL_SOURCE = VALUE #(
      ( SOURCE = REF #( ITAB )
        COMPONENTS = VALUE #(
          ( NAME = 'col3'
            ATEXT = ABAP_TRUE
            DESCENDING = ABAP_TRUE )
          ( NAME = 'col4'
            ATEXT = ABAP_TRUE
            DESCENDING = ABAP_TRUE ) ) ) ).

OUT->WRITE( V_INDEX ).

SORTED_TAB = VALUE #( FOR IDX IN V_INDEX ( ITAB[ IDX ] ) ).

TEST_TAB = ITAB.
SORT TEST_TAB STABLE BY COL3 AS TEXT DESCENDING
                        COL4 AS TEXT DESCENDING.
ASSERT SORTED_TAB = TEST_TAB.

OUT->WRITE( SORTED_TAB ).

OUT->DISPLAY( ).
ENDMETHOD.
ENDCLASS.

START-OF-SELECTION.
DEMO=>MAIN( ).

```

Ejecucion

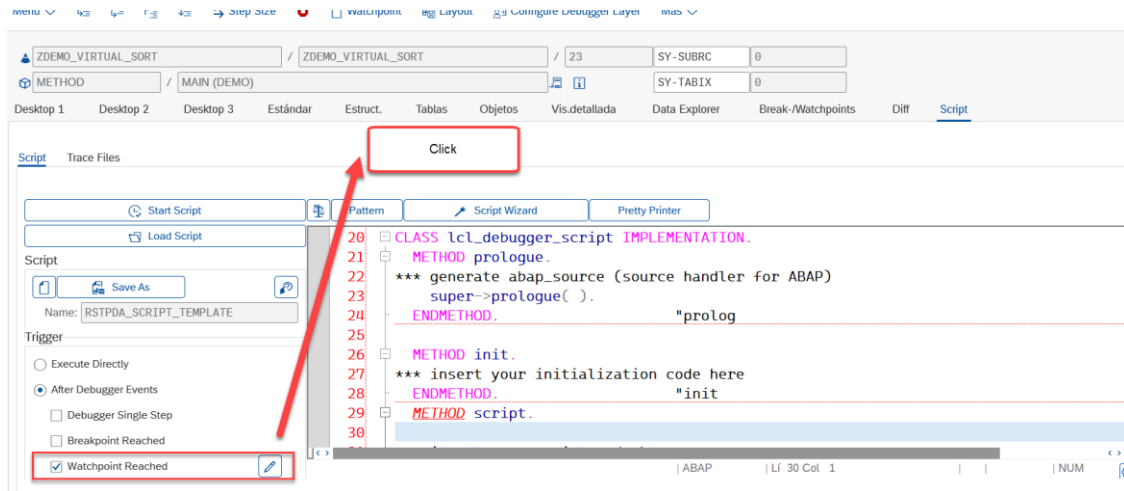
ITAB	COL1	COL2	COL3	COL4
8	10	C	D	
9	9	H	A	
9	4	E	B	
10	10	D	C	
1	9	H	I	
4	1	I	E	
6	1	C	A	
10	6	G	H	
2	2	E	H	
10	9	D	D	

Virtual Sort by col1, col2, Ascending	V_INDEX
8	9
9	6
9	7
10	1
2	2
4	3
10	4
1	5

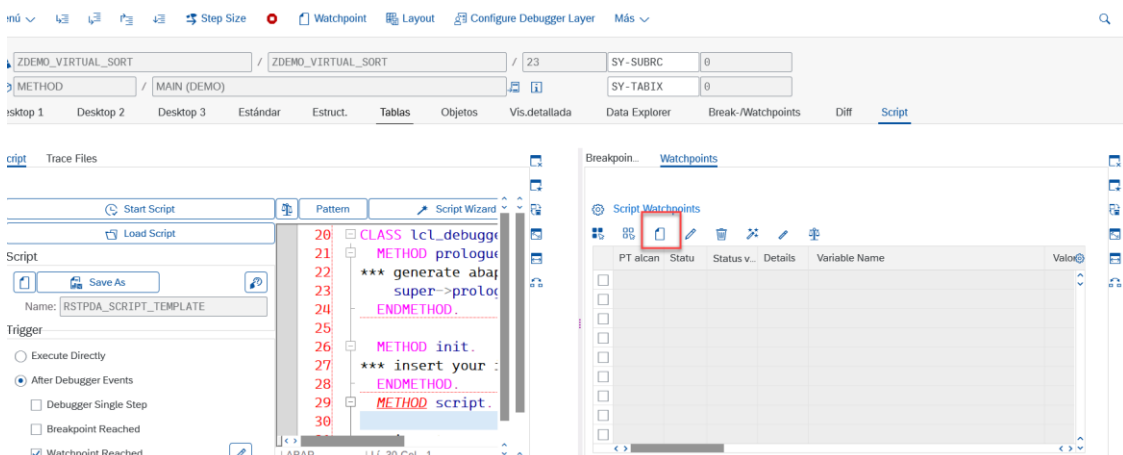
SORTED_TAB	COL1	COL2	COL3	COL4
1	9	H	I	
2	2	E	H	
4	1	I	E	
6	1	C	A	
6	10	C	D	
9	9	H	A	
9	4	E	B	
10	9	D	C	
10	6	G	H	

Virtual Sort by col3, col4, Descending	V_INDEX
6	9
5	8
2	4

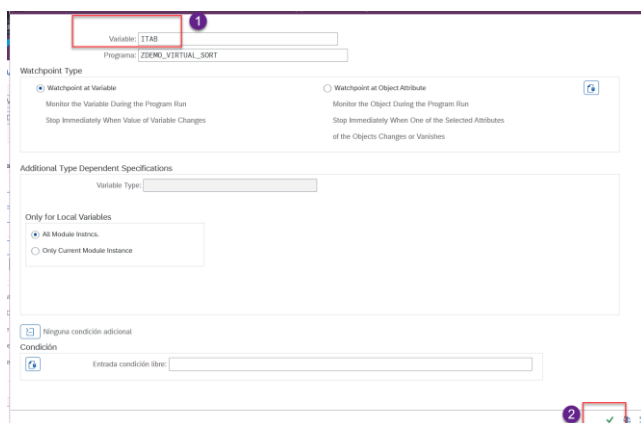
- Execute the program in debugging mode and go to Script Tab to create a script.
- Choose a triggering mode as **watch-point reached**.



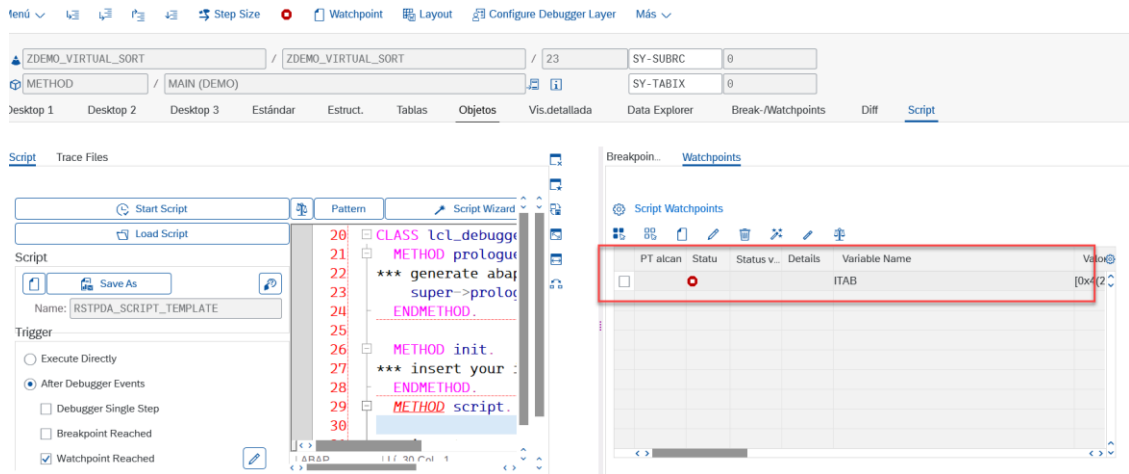
Now select the **Edit** button to add a watch-point. Choose the internal table. (Note : A script watch-point is different from the normal debugger watch-point)



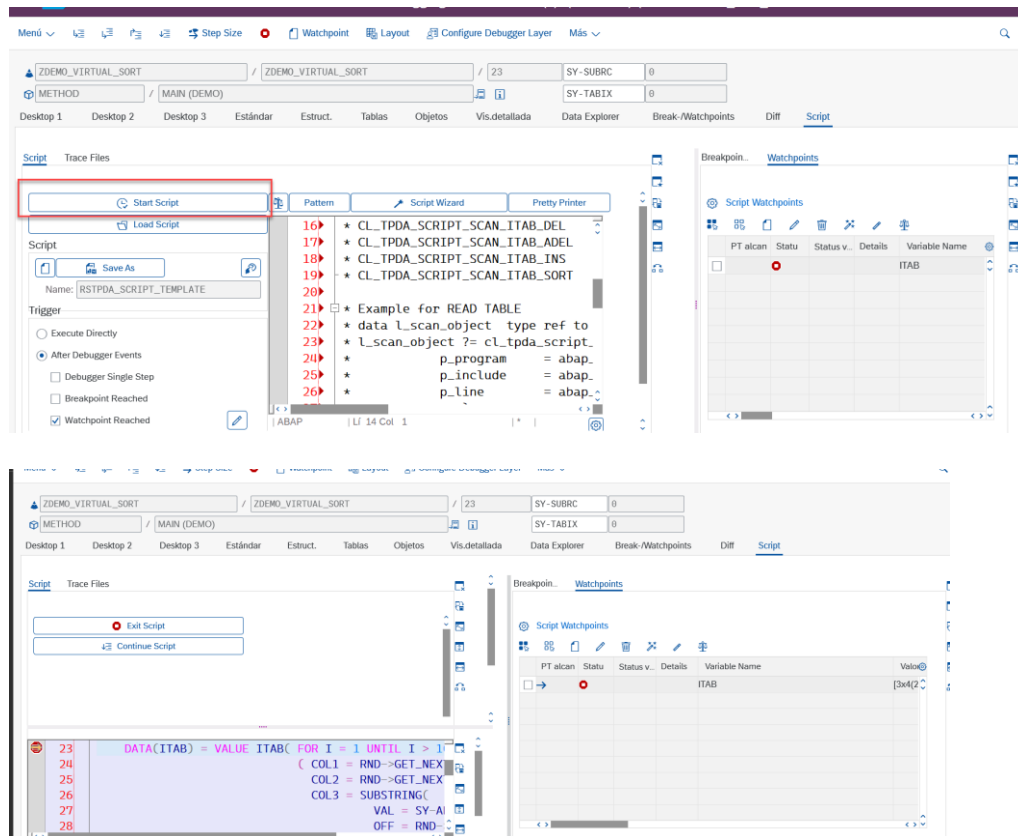
Click in New Button



Result.

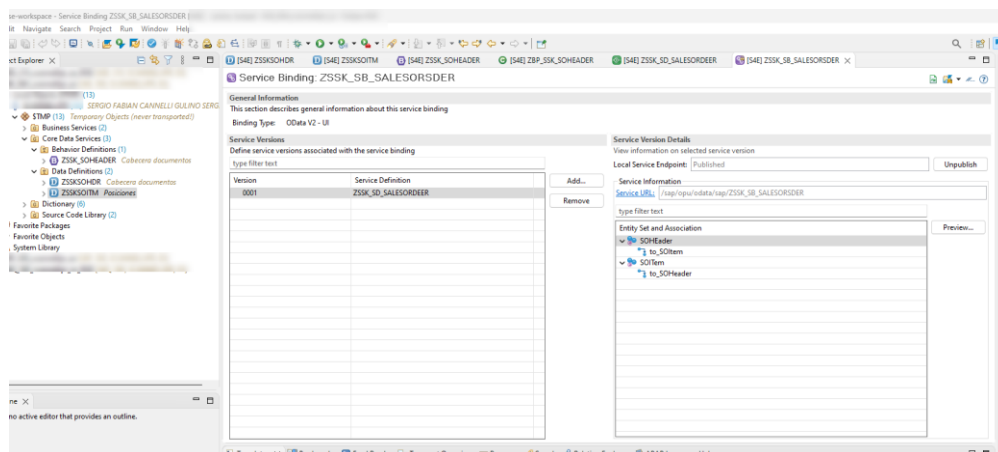
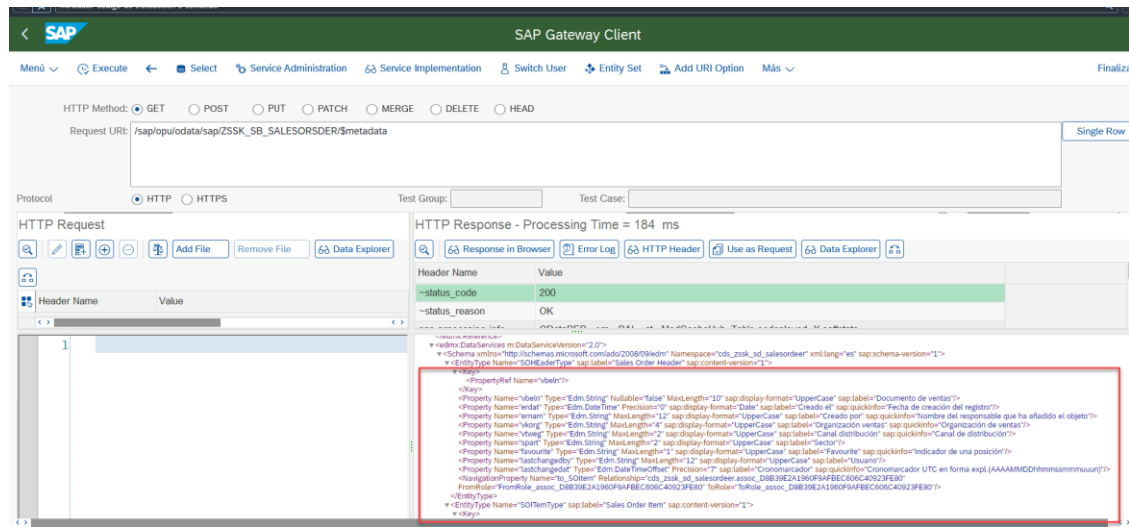


Next



Trace RFC

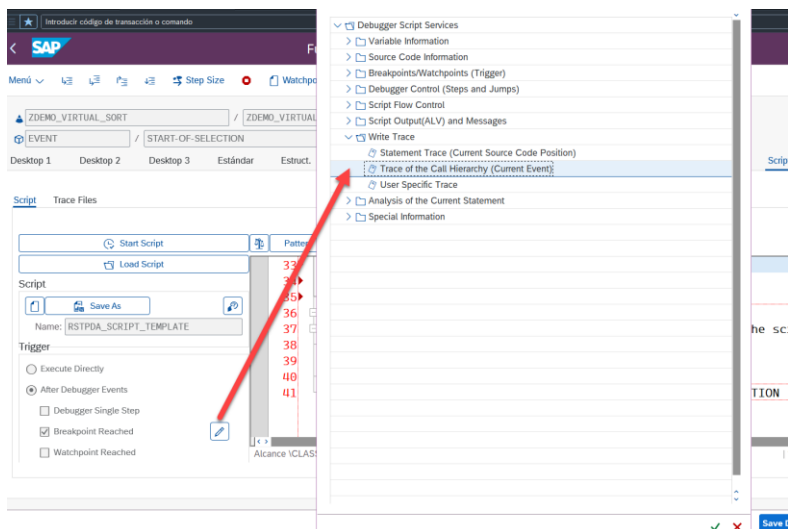
I have my service Created, which is born from a RAP object



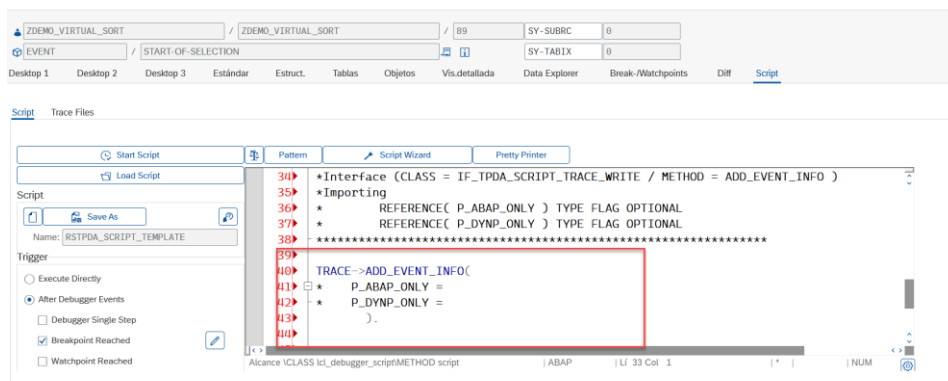
My debug is in the class that executes the RAP.

Procedure : I run my process and insert a DEBUG at startup, before processing any data. As a test, I document it in my Example program.

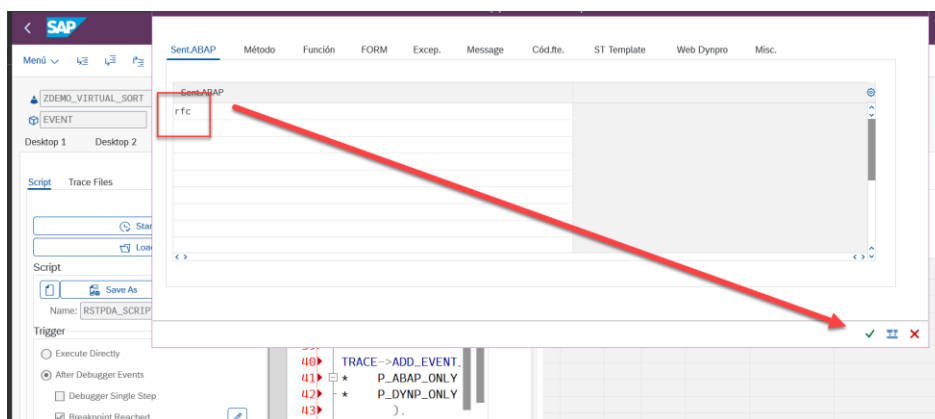
Insert the call to method ADD_EVENT_INFO() of IF_TPDA_SCRIPT_TRACE_WRITE



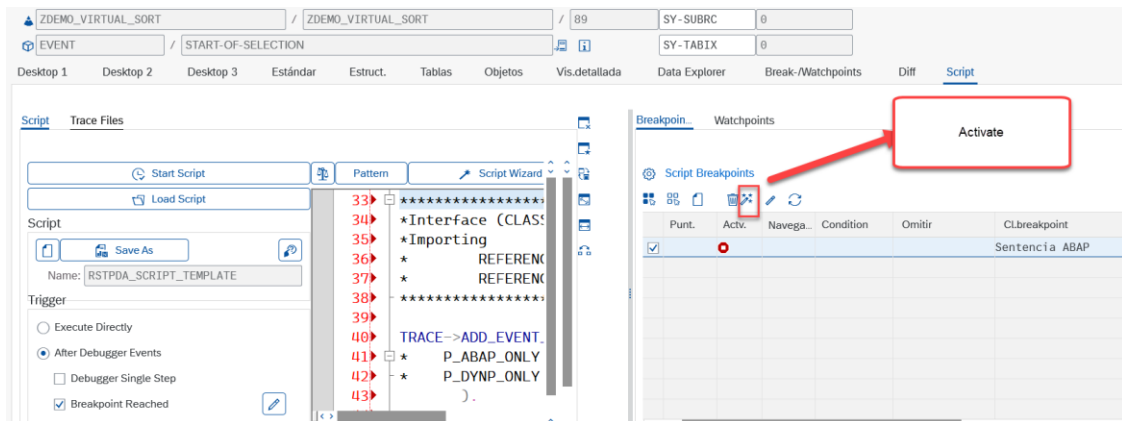
Inserted code



Create an RFC breakpoint. Setting a breakpoint on the ABAP Command 'RFC' tells the Debugger to run the script every time an RFC call is made.



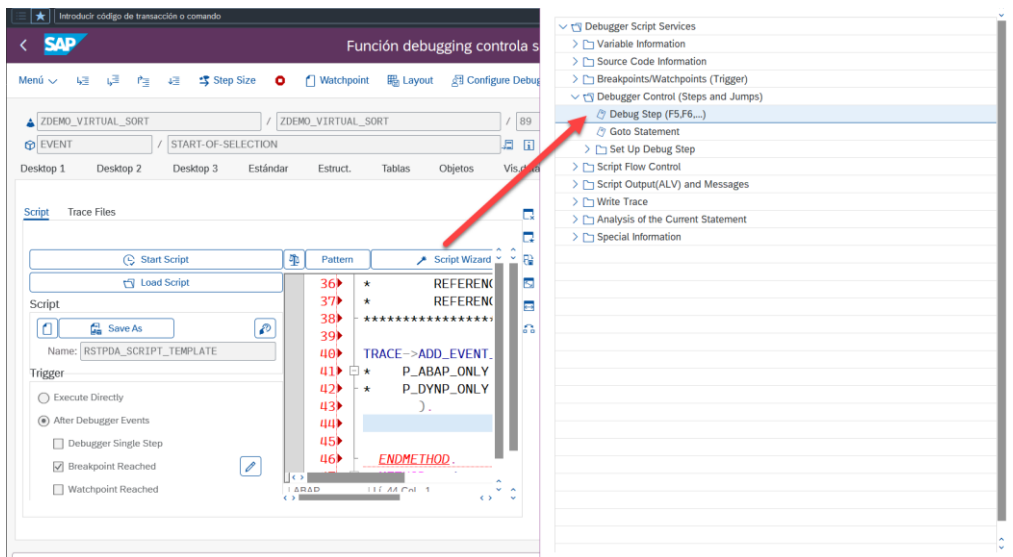
Next.



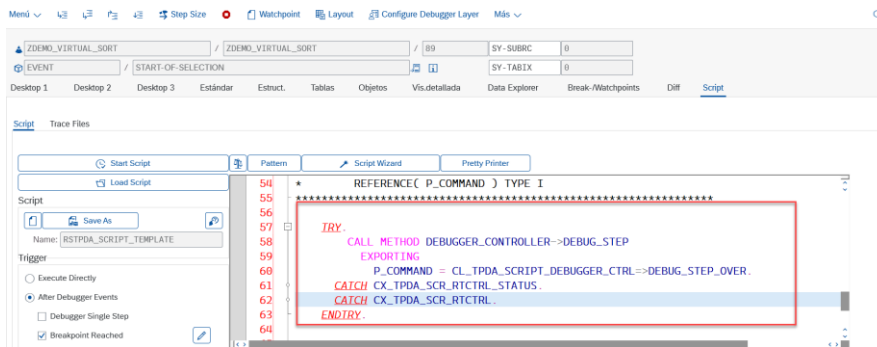
Insert your script code after `me->break()`.

You'll probably want to comment out the BREAK instruction. If you wish to, you can have the script run it when it encounters bad data returned by an RFC. You can stop the program that you are debugging and use the debugger to look at the problem more closely. You could call that an **'intelligent Debugger Script breakpoint'**. Since the script runs whenever an RFC call is to be made, we'll need this logic: 0.1. Run a debugger step to execute the RFC call 0.2. Read the variable or variables that we want to check 0.3. Apply tests to the value of the variable(s), our own 'application logic', so to speak. 0.4. If we find bad data, write a user-defined trace message, stop with a debugger breakpoint, or do both actions. Insert the first instruction – running the RFC function call – by using the Script Wizard to insert this debugger method call.

Next Step.



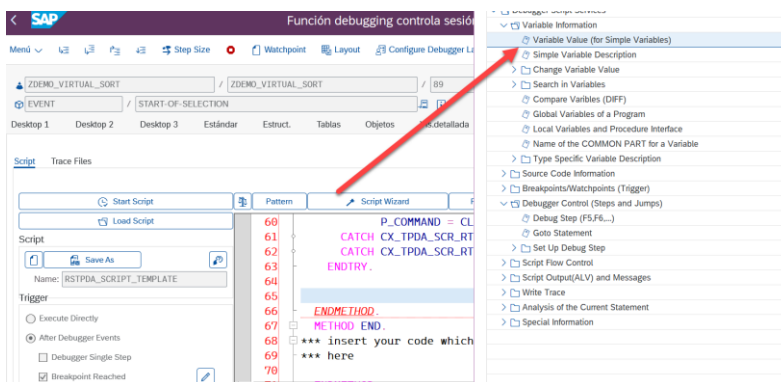
Inserted code



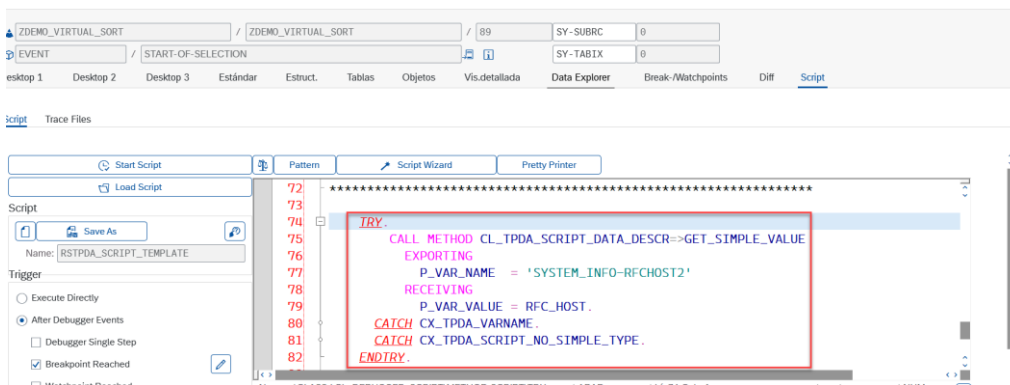
The method signature lists the constants for the different types of debugger steps and jumps.

Next Step.

Then add the second element of the script logic: A debugger method call to read a variable that is to be checked. Here, we use a variable from the RFCSI structure returned by the special RFC call RFC_SYSTEM_INFO, which requires no logon. This is a convenient way to simulate the error that occurred. Again, most of what we need is a debugger wizard



Inserted code



We might want to report exceptions with trace method calls after the CATCH instruction, The third element of the script is a little bit of logic to check whether SYSTEM_INFO-RFCHOST2 contains valid data.

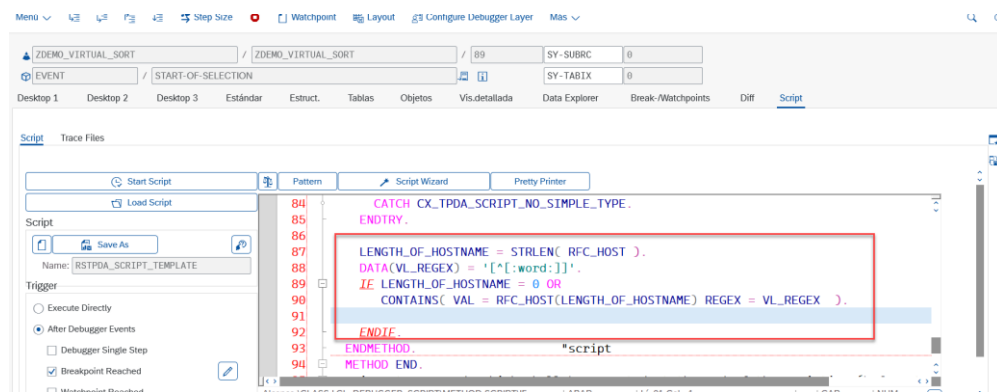
We must declare the following variables.

```
METHOD SCRIPT.

DATA RFC_HOST TYPE TPDA_VAR_VALUE.
DATA length_of_hostname TYPE i.
DATA target_destination TYPE rfcdst.
DATA my_trace TYPE tpda_trace_custom.
```

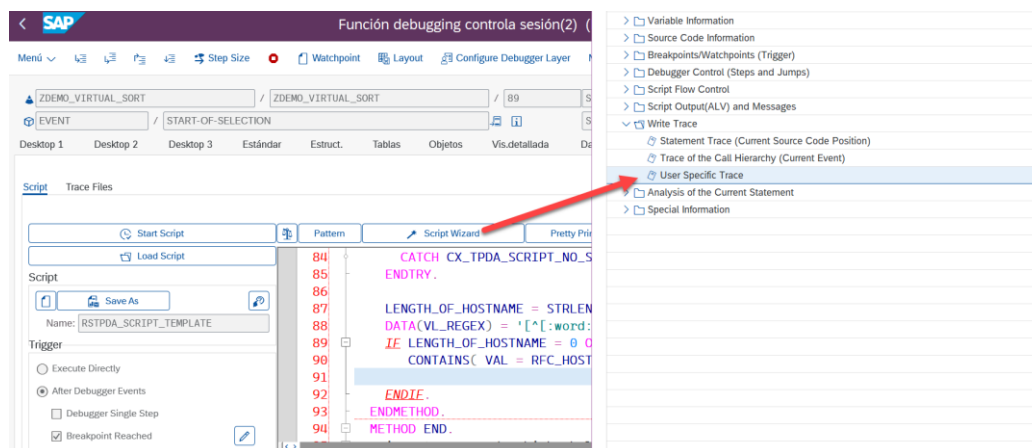
After the last inserted method, we are going to insert an IF to validate the RFC data are correct.

Inserted code



Check that a hostname was returned and then that the hostname contains only the alphanumeric characters that we expect. By the way, the program **DEMO_REGEX_TOY**, is a useful tool for testing your REGEX expressions. 0.1. The final element of the script is a trace message in the event that the script finds bad data returned by an RFC call.

Next Step.



Inserted code

TRY.

```
CALL METHOD CL_TPDA_SCRIPT_DATA_DESCR=>GET_SIMPLE_VALUE
EXPORTING
  P_VAR_NAME = 'LS_ALSYSTEMS-DESTINAT'
RECEIVING
  P_VAR_VALUE = TARGET_DESTINATION.
CONCATENATE 'This destination returns bad data:'
            TARGET_DESTINATION
            INTO MY_TRACE-VALUE SEPARATED BY SPACE.
```

```
CALL METHOD TRACE->ADD_CUSTOM_INFO
```

```
EXPORTING
```

```
  P_TRACE_ENTRY = MY_TRACE.
```

CATCH CX_TPDA_VARNAME CX_TPDA_SCRIPT_NO_SIMPLE_TYPE.

```
MY_TRACE-VALUE = 'Error in script'.
```

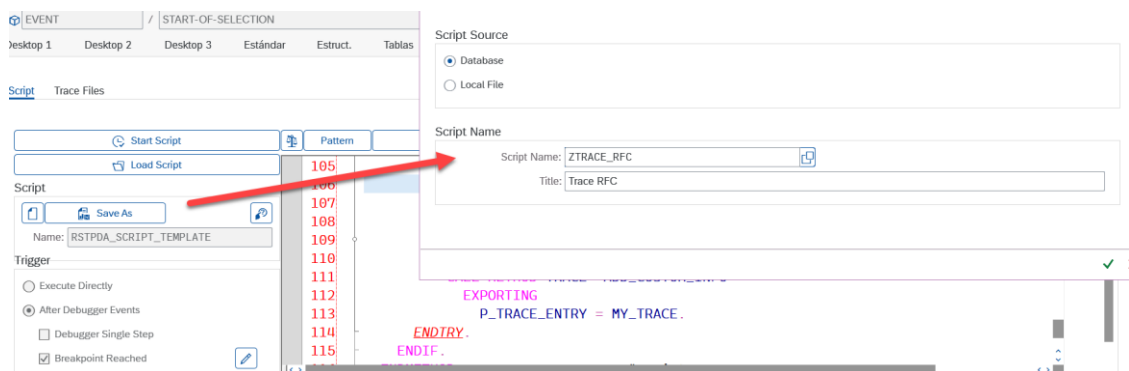
```
CALL METHOD TRACE->ADD_CUSTOM_INFO
```

```
EXPORTING
```

```
  P_TRACE_ENTRY = MY_TRACE.
```

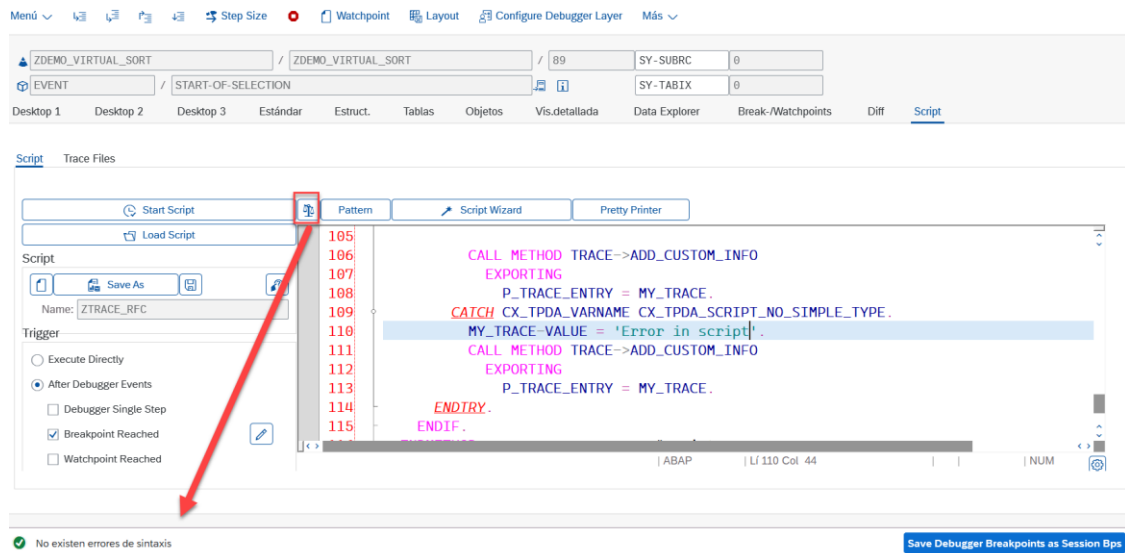
ENDTRY.

Next, Save Script.



Can be stored in a local or transport request.

Before executing check syntax.



When the SCRIPT is activated by clicking on "START SCRIPT", the result can be viewed in the "TRACE FILES" tab.

Complete Code.

```

*
*-----
*
CLASS LCL_DEBUGGER_SCRIPT DEFINITION INHERITING FROM
CL_TPDA_SCRIPT_CLASS_SUPER .

    PUBLIC SECTION.

        METHODS: PROLOGUE REDEFINITION,
                  INIT REDEFINITION,
                  SCRIPT REDEFINITION,
                  END REDEFINITION.

ENDCLASS.                                "lcl_debugger_script DEFINITION

*-----
*
*      CLASS lcl_debugger_script IMPLEMENTATION
*-----
*
*-----
*
CLASS LCL_DEBUGGER_SCRIPT IMPLEMENTATION.
    METHOD PROLOGUE.
*** generate abap_source (source handler for ABAP)
        SUPER->PROLOGUE ( ) .

```

```

ENDMETHOD.                                "prolog

METHOD INIT.
*** insert your initialization code here
ENDMETHOD.                                "init
METHOD SCRIPT.

    DATA RFC_HOST TYPE TPDA_VAR_VALUE.
    DATA LENGTH_OF_HOSTNAME TYPE I.
    DATA TARGET_DESTINATION TYPE RFCDEST.
    DATA MY_TRACE TYPE TPDA_TRACE_CUSTOM.
*** insert your script code here
    ME->BREAK ( ).
*****
*Interface (CLASS = IF_TPDA_SCRIPT_TRACE_WRITE / METHOD =
ADD_EVENT_INFO )
*Importing
*
    REFERENCE( P_ABAP_ONLY ) TYPE FLAG OPTIONAL
*
    REFERENCE( P_DYNP_ONLY ) TYPE FLAG OPTIONAL
*****

    TRACE->ADD_EVENT_INFO (
*
    P_ABAP_ONLY =
*
    P_DYNP_ONLY =
        ).
*****
* debugger commands (p_command):
* Step into(F5)    -> CL_TPDA_SCRIPT_DEBUGGER_CTRL=>DEBUG_STEP_INT0
* Execute(F6)      -> CL_TPDA_SCRIPT_DEBUGGER_CTRL=>DEBUG_STEP_OVER
* Return(F7)       -> CL_TPDA_SCRIPT_DEBUGGER_CTRL=>DEBUG_STEP_OUT
* Continue(F8)     -> CL_TPDA_SCRIPT_DEBUGGER_CTRL=>DEBUG_CONTINUE
*****
*****
*Interface (CLASS = CL_TPDA_SCRIPT_DEBUGGER_CTRL / METHOD = DEBUG_STEP
)
*Importing
*
    REFERENCE( P_COMMAND ) TYPE I
*****

TRY.
    CALL METHOD DEBUGGER_CONTROLLER->DEBUG_STEP
    EXPORTING
        P_COMMAND = CL_TPDA_SCRIPT_DEBUGGER_CTRL=>DEBUG_STEP_OVER.
    CATCH CX_TPDA_SCR_RTCTRL_STATUS.
    CATCH CX_TPDA_SCR_RTCTRL.
ENDTRY.

*****

```



```

*Interface (CLASS = CL_TPDA_SCRIPT_DATA_DESCR / METHOD =
GET_SIMPLE_VALUE )
*Importing
*      REFERENCE( P_VAR_NAME ) TYPE TPDA_VAR_NAME
*Returning
*      VALUE( P_VAR_VALUE ) TYPE TPDA_VAR_VALUE
*****

    TRY.
        CALL METHOD CL_TPDA_SCRIPT_DATA_DESCR=>GET_SIMPLE_VALUE
            EXPORTING
                P_VAR_NAME  = 'SYSTEM_INFO-RFCHOST2'
            RECEIVING
                P_VAR_VALUE = RFC_HOST.
        CATCH CX_TPDA_VARNAME.
        CATCH CX_TPDA_SCRIPT_NO_SIMPLE_TYPE.
    ENDTRY.

    LENGTH_OF_HOSTNAME = STRLEN( RFC_HOST ).
    DATA(VL_REGEX) = '^[[:word:]]'.
    IF LENGTH_OF_HOSTNAME = 0 OR
        CONTAINS( VAL = RFC_HOST(LENGTH_OF_HOSTNAME) REGEX = VL_REGEX
    ).

        ENDIF.
    ENDMETHOD.                                "script
    METHOD END.

    *** insert your code which shall be executed at the end of the
    scripting (before trace is saved)
    *** here

    ENDMETHOD.                                "end
ENDCLASS.                                    "lcl_debugger_script IMPLEMENTATION

```