Creazione OData

La guida è basata sulla mia esperienza personale, confronto con colleghi, imprecazioni in latino e tante guide dei nostri migliori amici dall’India. Va da sé che potrebbero esserci imprecisioni.

Contents

[1. OData in generale 2](#_Toc129253791)

[2. Creazione progetto semplice 3](#_Toc129253792)

[2.1 Data Model 5](#_Toc129253793)

[2.1.1 Creazione Entity / EntitySet 5](#_Toc129253794)

[2.1.2 Creazione Association 7](#_Toc129253795)

[2.2 Runtime Artifacts 9](#_Toc129253796)

[3. Configurazione e Richiamo del Servizio OData 11](#_Toc129253797)

[4. Service Implementation 13](#_Toc129253798)

[4.1 GET 13](#_Toc129253799)

[4.2 POST 18](#_Toc129253800)

[4.3 PUT / DELETE 19](#_Toc129253801)

[5. Creazione Progetto Complesso (Deep Entity e annidamenti) 20](#_Toc129253802)

[5.1 Deep Entity 22](#_Toc129253803)

[5.2 GET Deep Entity 23](#_Toc129253804)

[5.2.1 Definizione Types nella classe \*MPC\_EXT 23](#_Toc129253805)

[5.2.2 Binding Deep Structure 25](#_Toc129253806)

[5.2.3 Ridefinizione GET\_EXPANDED\_ENTITY 25](#_Toc129253807)

[5.2.4 Test GET Deep\_Entity 29](#_Toc129253808)

[5.3 POST Deep Entity 29](#_Toc129253809)

[6. Cancellazione di un progetto 33](#_Toc129253810)

# OData in generale

L’OData è un tipo di servizio REST il quale, a seconda di come viene manipolato il link di chiamata, assume comportamenti differenti. È Utilizzato soprattutto sulle APP Fiori.

Di base, implementa il cosiddetto pattern MVC (Model View Controller) che prevede una separazione dei ruoli principali da implementare -> Il Model definisce la struttura dei dati da utilizzare, la View si occupa dell’input e dell’output. Infine, il Controller il quale costituisce l’elemento che si occupa della gestione dei dati intesa come manipolazione degli stessi (CRUD -> Creation Reading Update Deletion).

***La guida coprirà i seguenti argomenti:***

* Come creare un progetto sia semplice (GET e POST di tabella di testata con relative posizioni)
* Attivazione del servizio e richiamo tramite Gateway Client
* Manipolazione del link e navigazione nell’OData
* Come creare un progetto complesso (GET e POST di una struttura complessa con N livelli di annidamento)

Creare l’OData è un processo lunghetto e macchinoso su svariati aspetti perciò, prima di sperimentare, leggere tutta la guida nell’ordine in cui è stata scritta, special modo se avete strutture da gestire che sono annidate (Ad esempio, se voleste creare un BP sicuramente in input avrete tipi tabella annidati in strutture, annidati in altri tipi tabella ecc.)

# Creazione progetto semplice

Per implementare un OData, va definito il relativo progetto che conterrà il modello dati e le implementazioni.

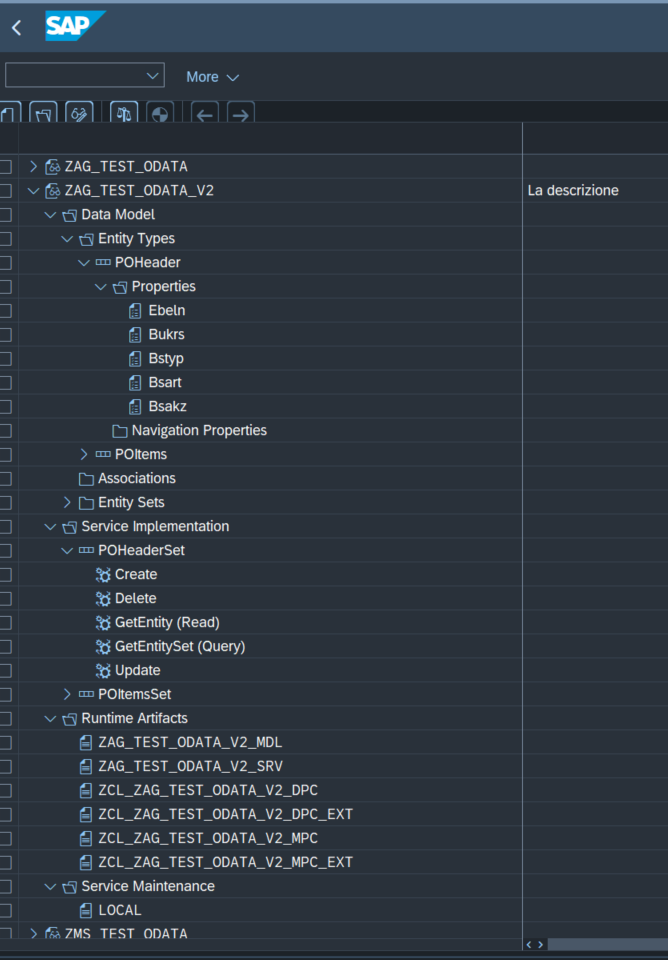
***NOTA 1: PER LA CREAZIONE E GESTIONE DEI PROGETTI BISOGNA LOGGARSI IN INGLESE, ALTRIMENTI…SUCCEDONO COSE BRUTTE***

***NOTA 2: I NOMI DEI COMPONENTI SONO CaseSensitive, PER CUI...OCCHIO ALLE MAIUSCOLE***

Accedere alla transazione SEGW:

Creare un nuovo progetto. Come da screen a seguire, si noterà che un progetto è costituito da quattro macro-componenti:

* Data Model
* Runtime Artifacts
* Service Implementation
* Service Maintenance



A seguire verranno descritti i Data Model e i Runtime Artifacts.

I service Implementation invece, saranno spiegati meglio dopo aver illustrato come attivare il servizio e richiamarlo tramite Gateway.

I Service Maintenance non saranno trattati in quanto sono argomento per sistemi che gestiscono le implementazioni del servizio e la configurazione dello stesso tramite sistemi diversi ( non è stato il mio caso per cui non ne parlerò ).

## Data Model

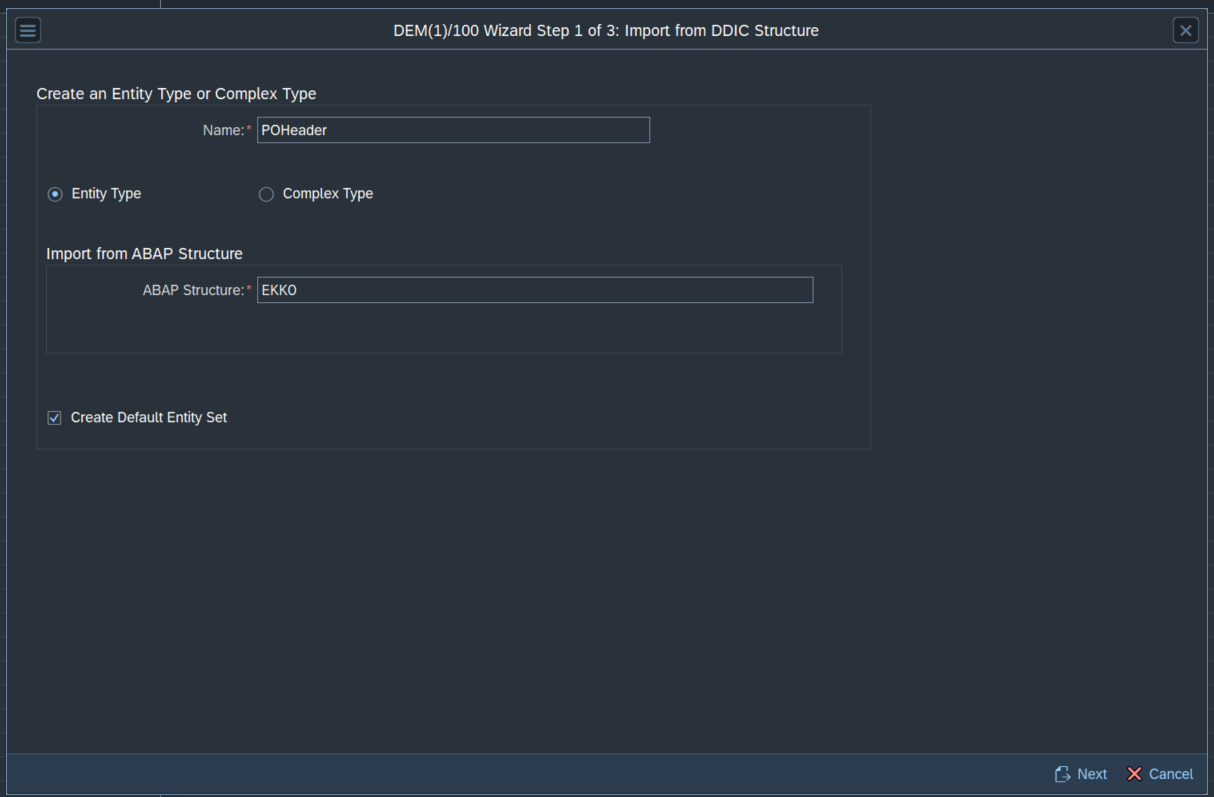
Il Data Model costituisce lo scheletro del progetto. In questo ramo vengono dichiarate quelle che sono definite “Entity”. Ciascuna Entity rappresenta un oggetto DDIC importabile oppure un oggetto che può essere creato manualmente specificandone i nomi dei campi e relativi data element che lo costituiscono.

Il classico esempio di entity, visibile anche nello screen può essere il POHeader (Importato dalla tabella EKKO) oppure POItem (EKPO). Ciascuna entity è composta da delle “Properties”, ossia l’equivalente dei campi a DDIC (EBELN, EBELP, ecc.).

### Creazione Entity / EntitySet

Per creare una entity importandola da DDIC, posizionarsi con il cursore sul nodo Data Model ->Tasto destro -> Import-> DDIC Structures

Dovrete inserire il nome della Entity (Case Sensitive), la tabella da cui effettuare l’import, i campi di interesse e infine la CHIAVE che costituisce la vostra entity. Il concetto di chiave verrà approfondito dopo insieme al concetto di EntitySet



A screenshot of a computer

Description automatically generated with medium confidence

Graphical user interface, text

Description automatically generated

Con gli odata viene introdotto il concetto di Entity ed EntitySet. In sostanza la Entity rappresenta la struttura della vostra entità, o se vogliamo, una work area. L’EntitySet invece rappresenta una tabella del vostro tipo Entity. Da qui, il motivo per cui vi è la necessità di inserire una chiave. Può capitare che dobbiate importare un tipo tabella che, come sappiamo, non ha necessariamente un legame a livello di DDIC costituito da una chiave...essendo però questa obbligatoria, specificatene una che abbia il maggior senso possibile. Il loro utilizzo pratico sarà più chiaro in seguito quando verranno approfonditi i Service Implementation.

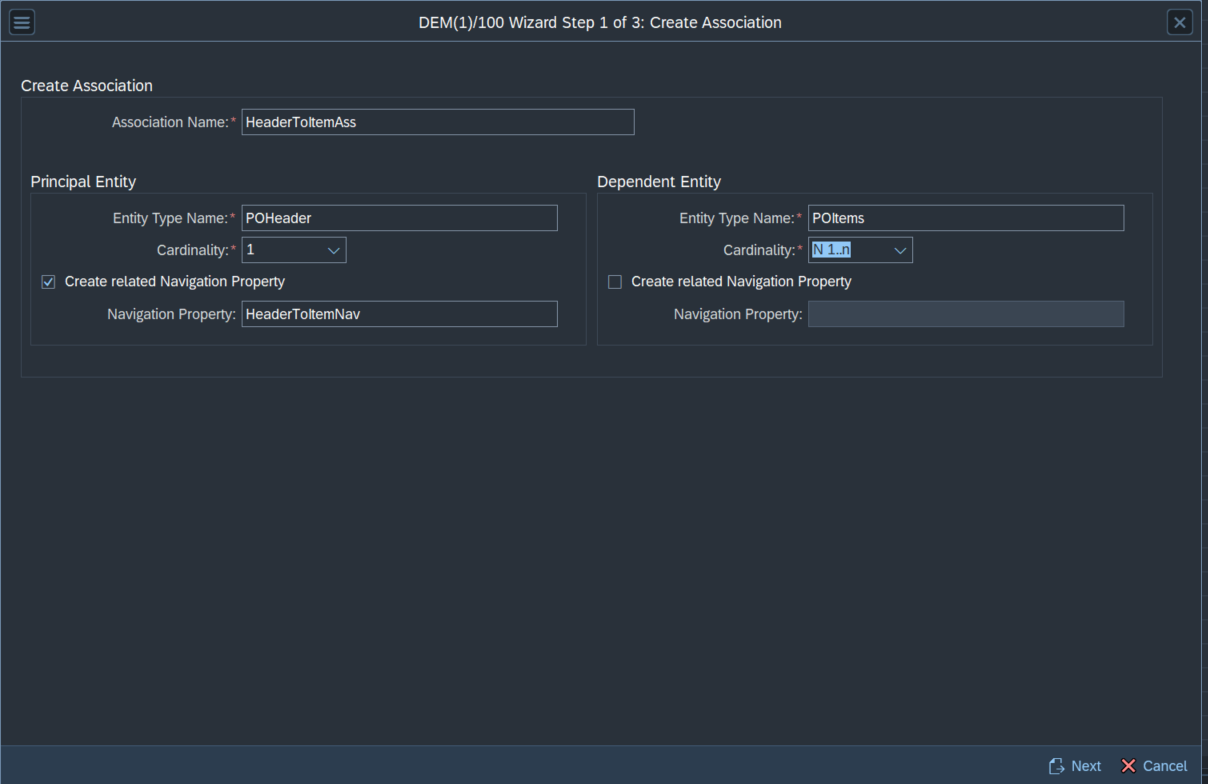
### Creazione Association

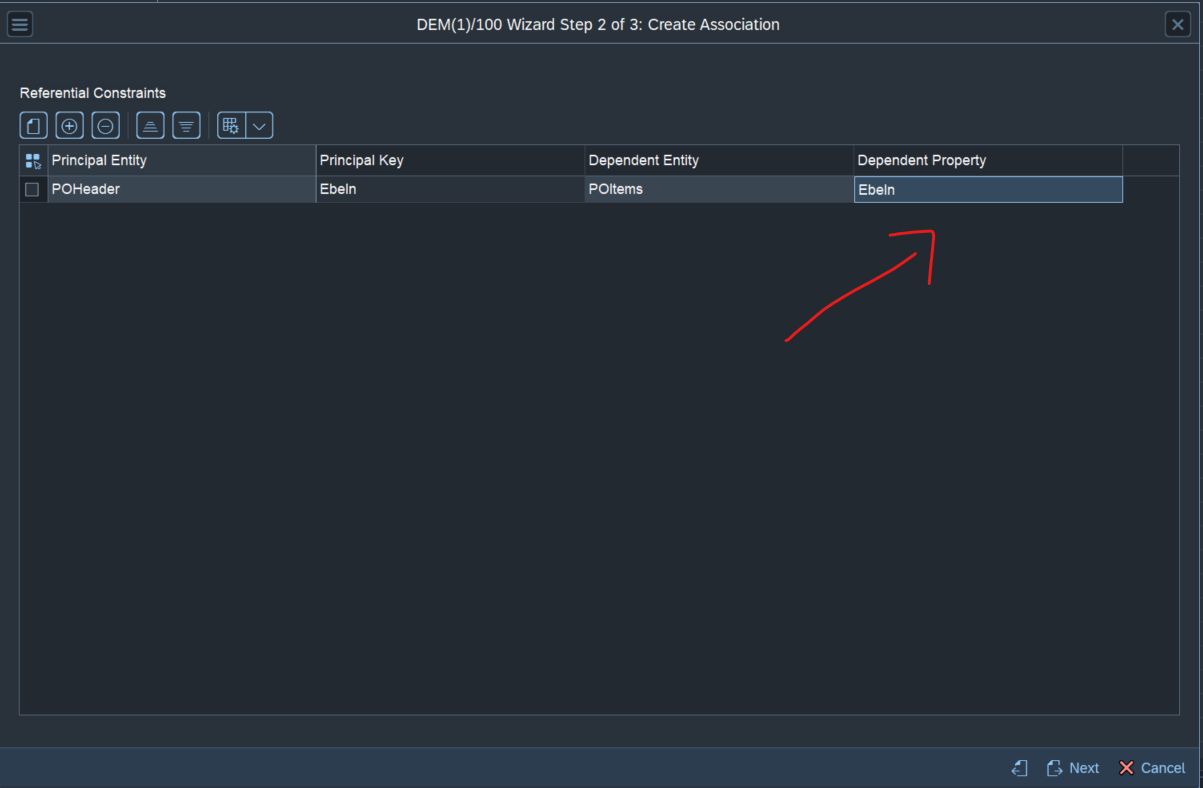
Si premette che: se noi creassimo delle entity come POHeader, POItem che rappresentano rispettivamente la EKKO e la EKPO, lo sappiamo noi che sono collegate. L’OData non lo sa. Per creare il legame dovranno essere definite delle “Association”.

***Tale operazione è necessaria e manuale per ogni legame che vorrete creare tra delle entity. Non è una cosa automatica.***

Posizionando il cursore sul nodo Data Model / Entity Types / Associations, createne una seguendo il wizard:

Date un nome alla vostra Associazione, definite l’entity di partenza e l’entity di arrivo con le cardinalità appropriate e create la “Navigation Property”. Quest’ultimo elemento è quello che il frontend richiama e che permette all’OData di effettuare l’effettiva navigazione tra la testata e le posizioni. In seguito, sarà più chiaro ma per ora, date per buono. Diciamo che l’Association fa da metadata e la Nav. Property fa da implementazione.



Inserite il campo di collegamento tra la testata e la posizione (Occhio, SAP non lo mette in automatico)

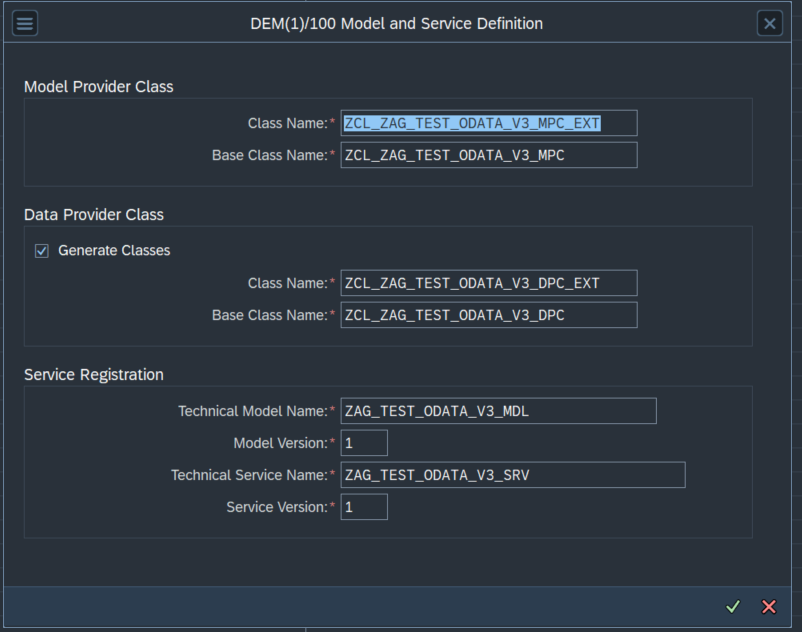
Anche per le Association, sarà creato il relativo EntitySetA screenshot of a computer

Description automatically generated

## Runtime Artifacts

Arrivati a questo punto, le vostre entity saranno pronte. Potrete generare il progetto da apposita icona o semplicemente con un CTRL+F3.

Apparirà la seguente schermata



Ci sono 3 coppie di oggetti e 2 in particolare costituiranno il core delle vostre logiche.

1. Model Provider Class -> Queste 2 classi contengono le dichiarazioni del vostro modello dati (i Types delle tabelle, Association, ecc.). Tra le 2, potrete scrivere il vostro codice solo in quella che termina con EXT in quanto, ad ogni rigenerazione/attivazione, la classe normale viene rigenerata e quindi sovrascritta.
2. Data Provider Class -> Queste 2 classi contengono il core delle logiche e come per il model, anche in questo caso dovrete scrivere il vostro codice nella \*\_EXT. Qui potrete effettuare la manipolazione di dati, richiamo di bapi, ecc.
3. Service registration -> Questi 2 oggetti rappresentano il Modello del servizio come “metadato” e il servizio ICF che verrà richiamato dall’esterno. (Approfondimento più a valle)

Come già accennato, le classi più importanti sono la \*MPC\_EXT e la \*DPC\_EXT

Partendo dalla DPC\_EXT, si potrà notare navigandoci, che contiene una serie di metodi per ognuna delle Entity/EntitySet che abbiamo dichiarato:



Ciascuna entity avrà quindi i suoi CRUD methods e solo nel caso dei GET (quindi estrazione dati) vi sarà anche il relativo <Entity>\_GET\_ENTITYSET. Perché? Perché le EntitySet, come già detto, essendo delle “Tabelle Interne” possono essere restituite massivamente in output. La creazione, aggiornamento e cancellazione invece, come si può dedurre coinvolge una singola entity per chiamata. Ovviamente ci sono dei work-around, ma ne parleremo in seguito quando verrà introdotto il concetto di DEEP\_ENTITY con i tipi annidati.

Per poter utilizzare i metodi, questi andranno ridefiniti. In debug sarà poi possibile verificare i parametri di input e output di ciascun metodo. Nel prossimo paragrafo, vedremo come lanciare il servizio e quindi richiamare ciascuna ridefinizione in base alle nostre esigenze. In questo modo verificheremo quindi che tipo di codice inserire in ciascun metodo.

# Configurazione e Richiamo del Servizio OData

A questo punto, abbiamo configurato le basi del nostro progetto.

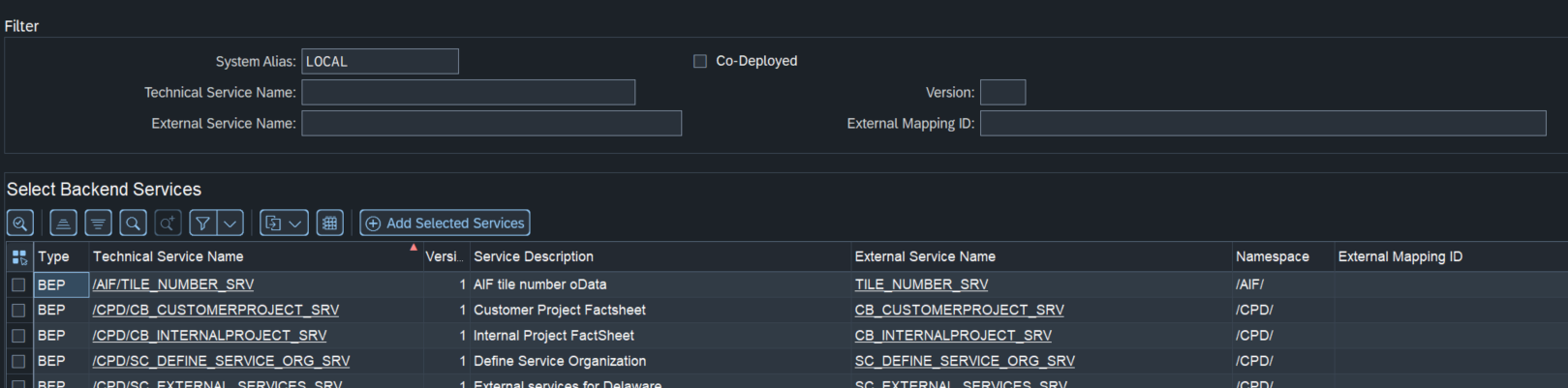
Per poterlo richiamare, andrà attivato il servizio.

Accedere alla transazione /N/IWFND/MAINT\_SERVICES



Verificate che il vostro sia presente (alcuni sistemi lo aggiungono in automatico). Per farlo filtrate il nome tecnico con il vostro nome servizio \*SRV (Ricordate i Runtime Artifacts?).

In caso contrario dovrete cliccare su Add Services accedendo alla seguente schermata



Ponendo attenzione al valorizzare il System Alias (Nel caso dello screen LOCAL).

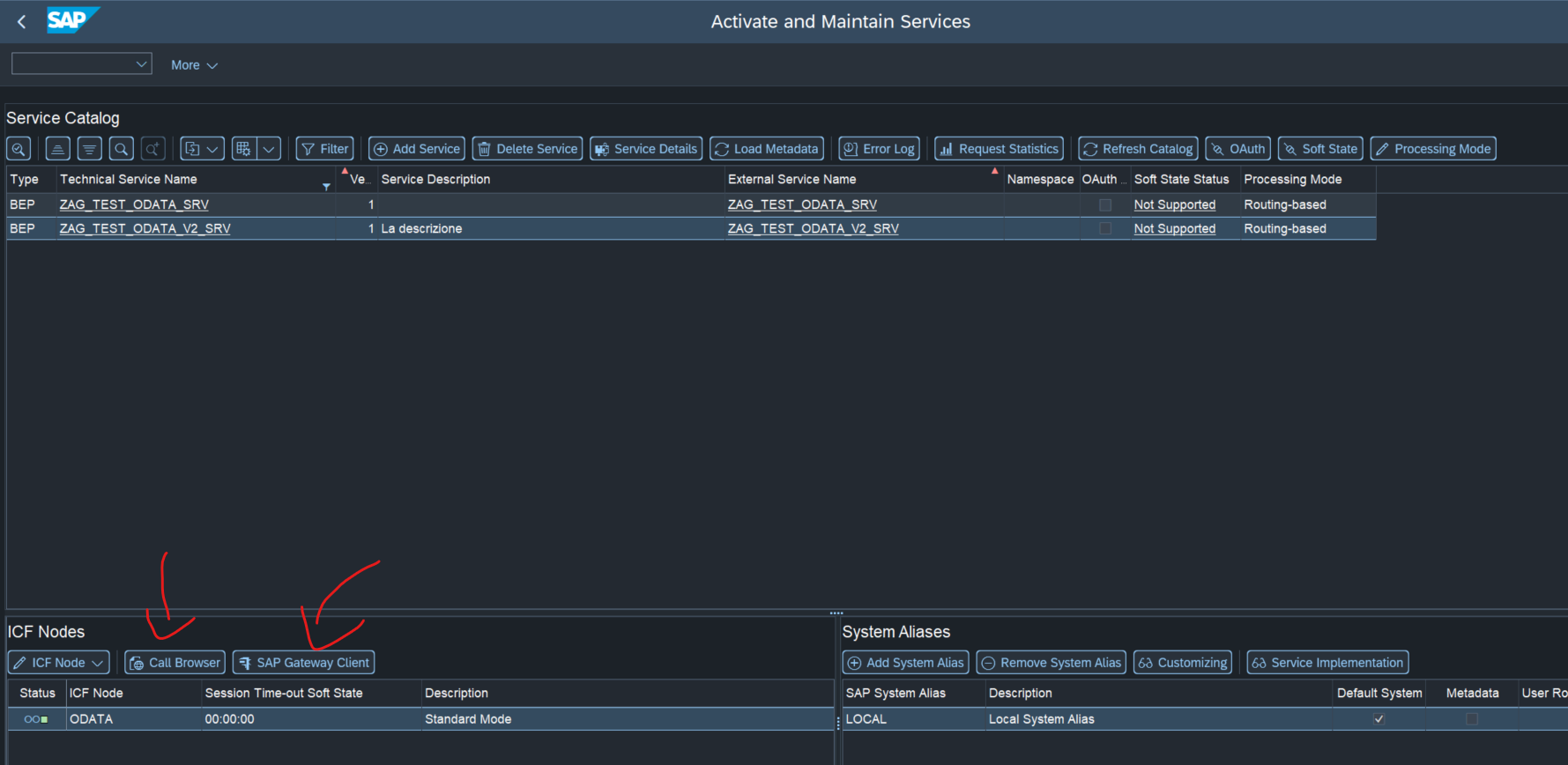
A quel punto, individuato il vostro servizio, selezionatelo e aggiungetelo da apposito pulsante.

Tornate quindi alla precedente schermata.

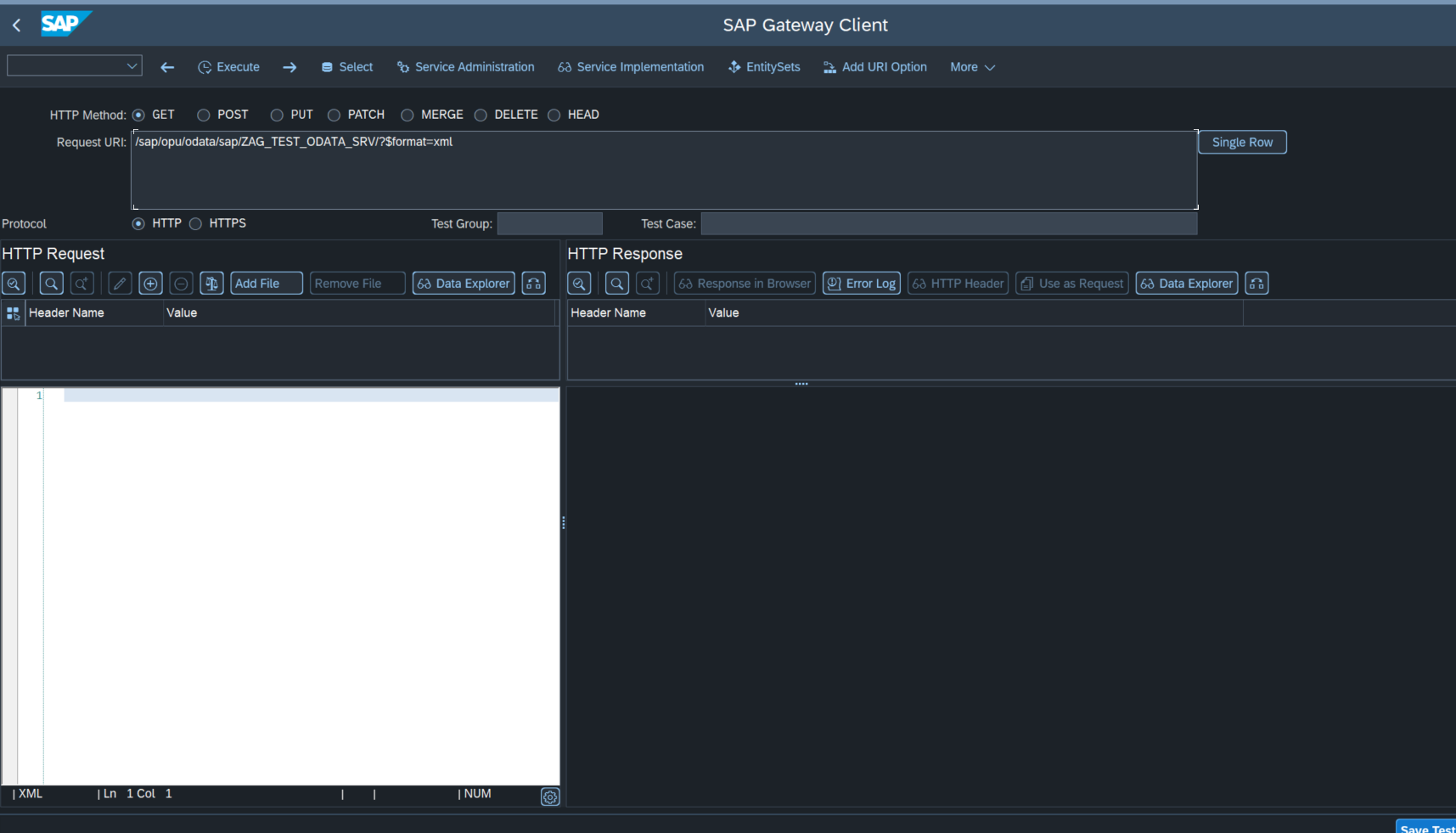
Nella sezione in basso a sinistra, dopo aver cliccato sul vostro servizio troverete 2 pulsanti.

Call Browser, si spiega da sé, consentirà di effettuare la chiamata da browser, verificando che il link sia effettivamente raggiungibile. Potrebbe essere necessario, ad esempio, sostituire la parte del vostro dominio con l’indirizzo IP diretto.

Il Gateway Client è invece il Core dei vostri Test.



Il Gateway Client (raggiungibile anche da transazione /N/IWFND/GW\_CLIENT) avrà la seguente interfaccia:



È possibile a questo punto, effettuare le chiamate al vostro servizio. In base al radiobutton che userete ( GET, POST, PUT, DELETE), saranno richiamati i rispettivi metodi CRUD che avrete opportunamente ridefinito nella classe \*DPC\_EXT.

# Service Implementation

## GET

La GET è il metodo che va usato con lo scopo di estrarre dati, fornendo eventuali chiavi e filtri.

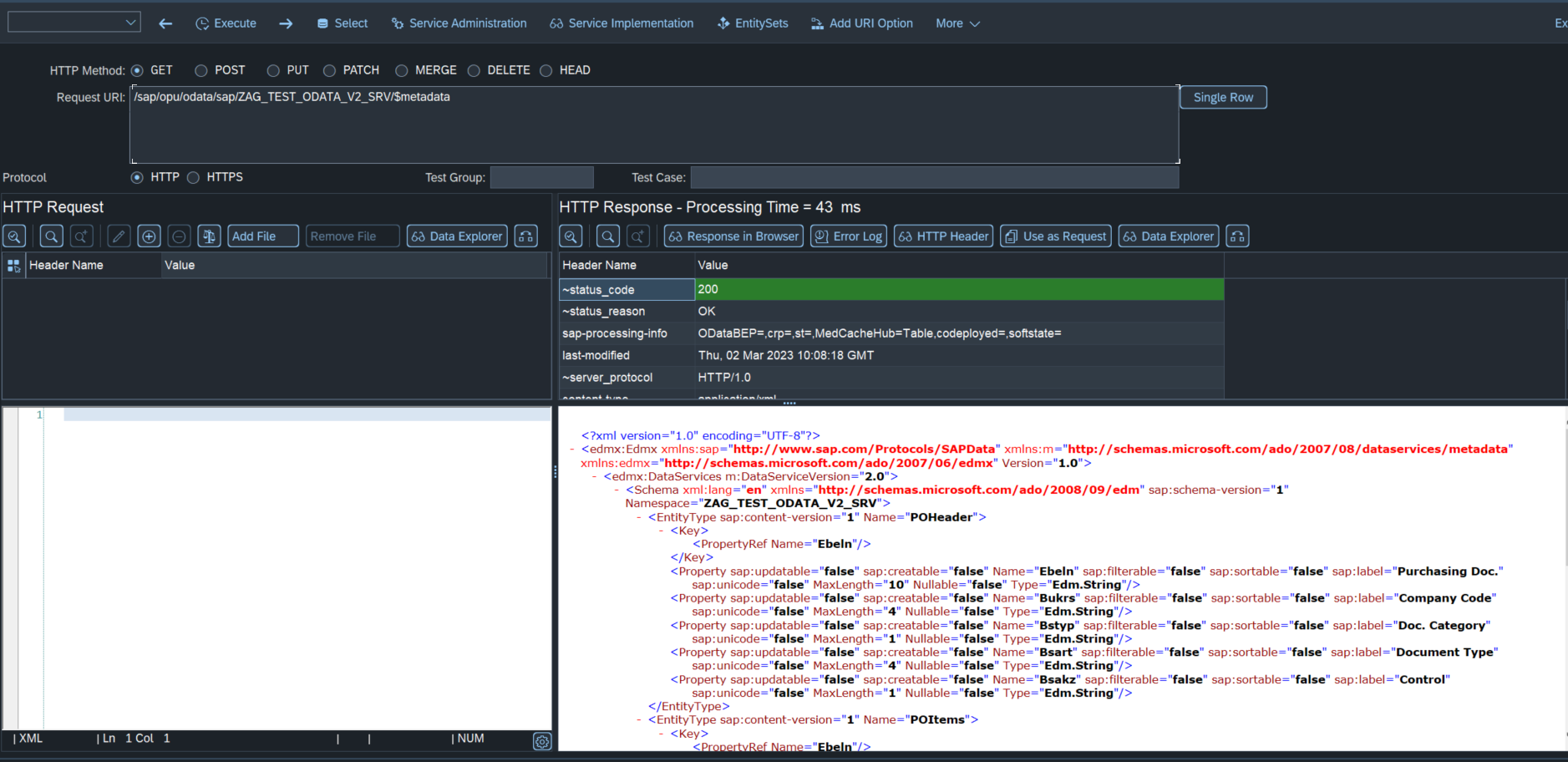
Inoltre, ha la funzione di “descrivere” il modello dati dell’OData stesso. Ricordate quando si parlava di “manipolare” il link? Bene, questa operazione viene fatta nelle GET.

La manipolazione viene fatta attraverso una serie di “funzioni” da inserire nel link con la nomenclatura $<nome\_funzione>. Ad esempio, $metadata per ricevere in output la struttura dettagliata del model, oppure $format=<formato> per predisporre l’output in formato xml o json.

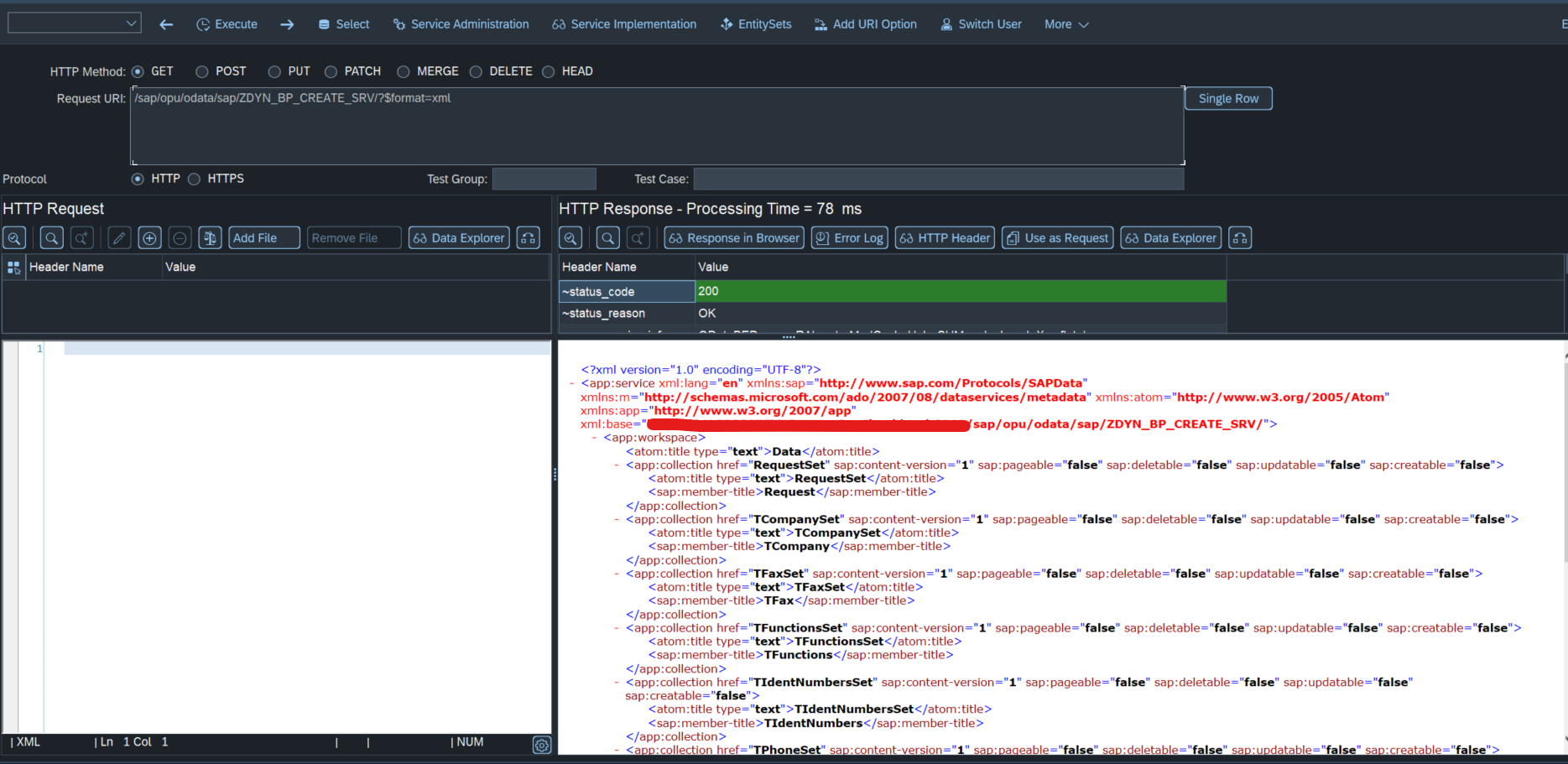
È possibile combinare queste funzioni concatenandole con l’uso della “&”. Alcune, come la $format, richiedono l’aggiunta del carattere “?” che sta a indicare l’esecuzione di una query.

Esempi a seguire

$metadata



$format=xml



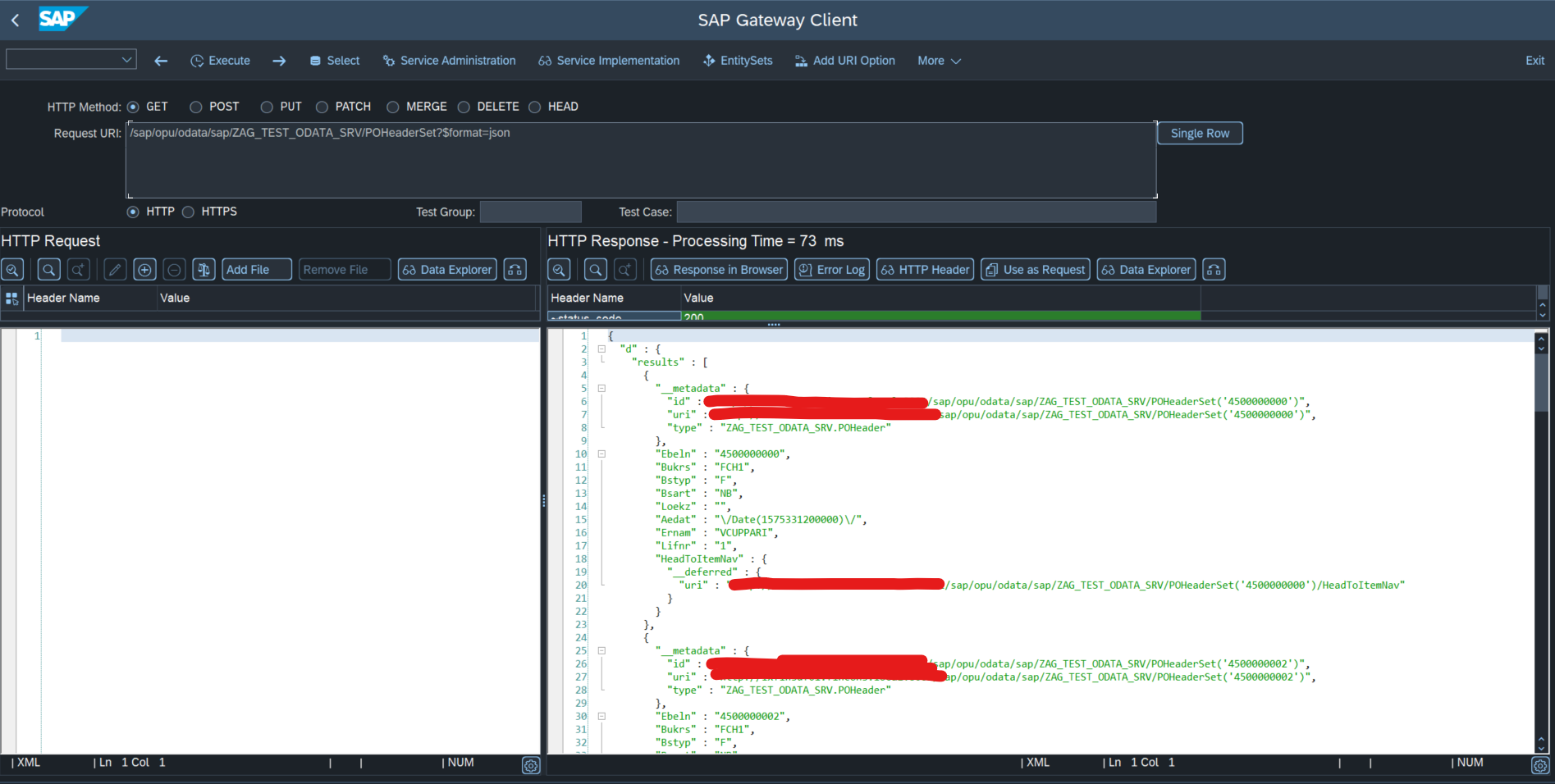
$format=json

A computer screen capture

Description automatically generated with low confidence

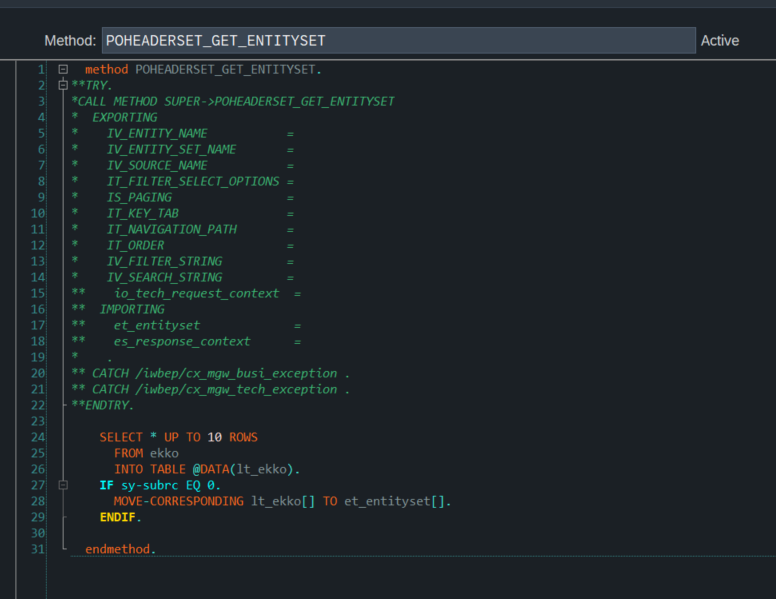
A questo punto, possiamo effettuare la nostra GET per ottenere dei dati conformi alle entity che abbiamo dichiarato.

Come? Ci sono due modi. Il primo è utilizzando l’EnitySet (non l’entity mi racc.) in modo semplice. Quindi, <link>/<EntitySet>

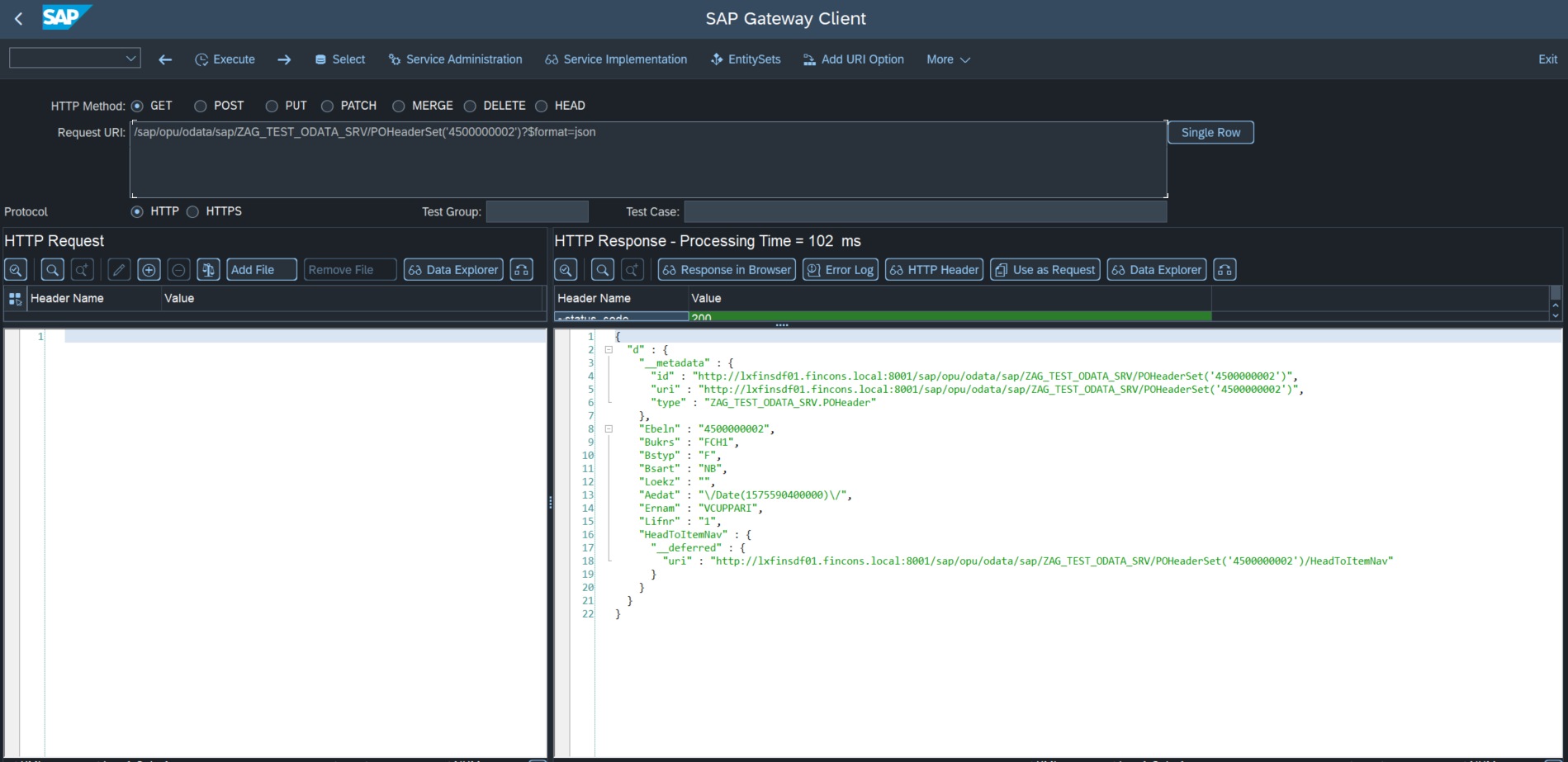


È possibile inserire dei filtri tramite funzione $filter. Per l’uso specifico, Google.

Come da screen, sono stati estratti dei dati. La funzione di estrazione è stata implementata nel metodo POHeaderSet\_GET\_ENTITYSET. Con un Break Esterno, richiamando il servizio potrete esplorare i parametri in input alla funzione.



Il secondo modo è il richiamo specificando una chiave nell’EntitySet. ***Ricordate la chiave inserita quando la Entity è stata creata? Questo è il momento di usarla. Se specificheremo una chiave all’interno della chiamata, lo standard si comporterà in maniera differente. Anziché richiamare il metodo POHeader\_GET\_ENTITYSET, in automatico, verrà richiamato il metodo POHeader\_GET\_ENTITY.***



A seguire, l’implementazione del metodo POHeader\_GET\_ENTITY

Text

Description automatically generated

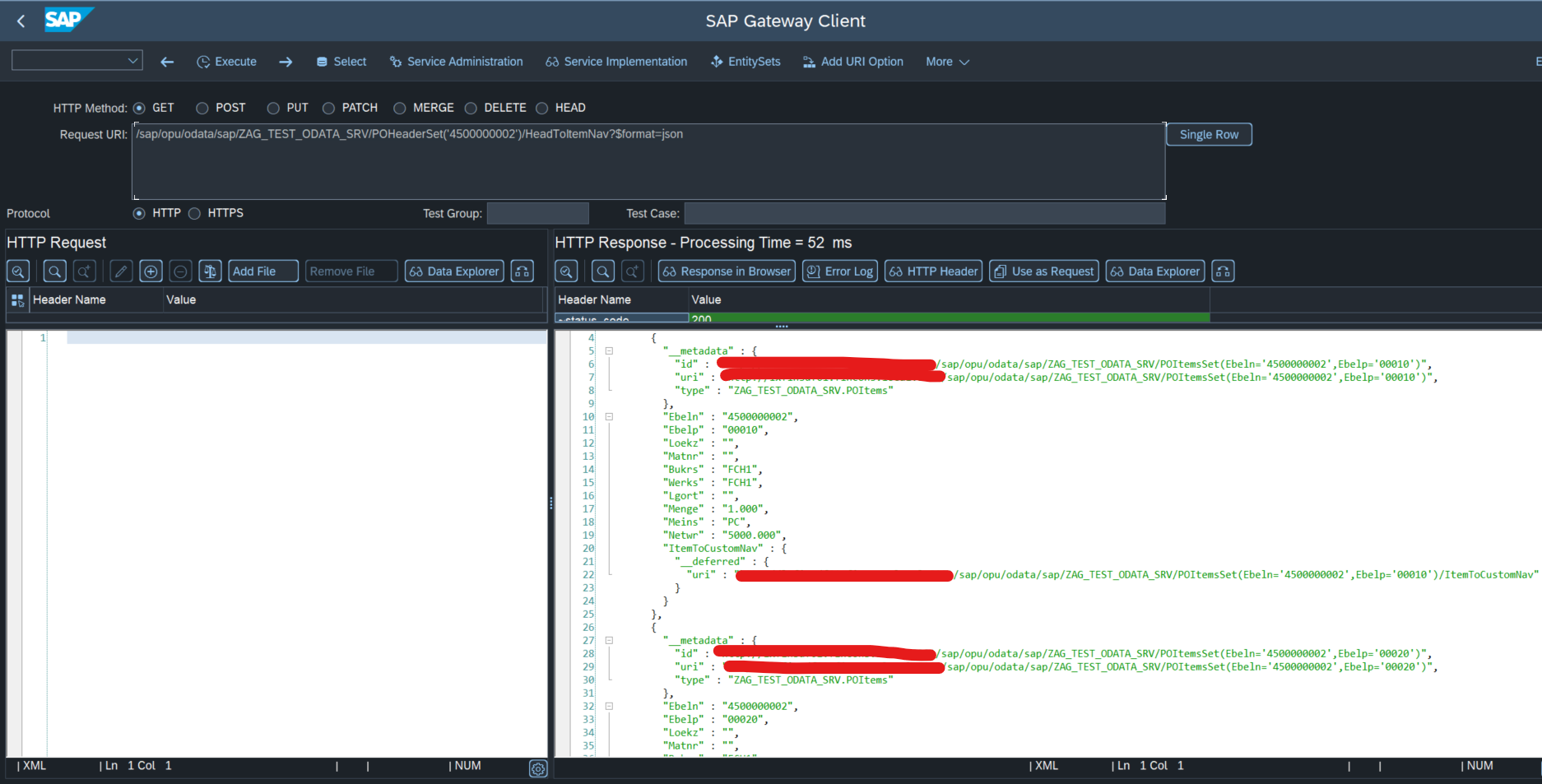
Si precisa che nel link posso anche utilizzare la chiamata diretta agli ItemSet e implementandone il relativo metodo. Tutto ciò è circostanziale rispetto all’esigenza.

A questo punto, abbiamo gli strumenti per poter estrarre sia le testate dei nostri documenti, sia le posizioni o qualunque altra entity abbiamo dichiarato.

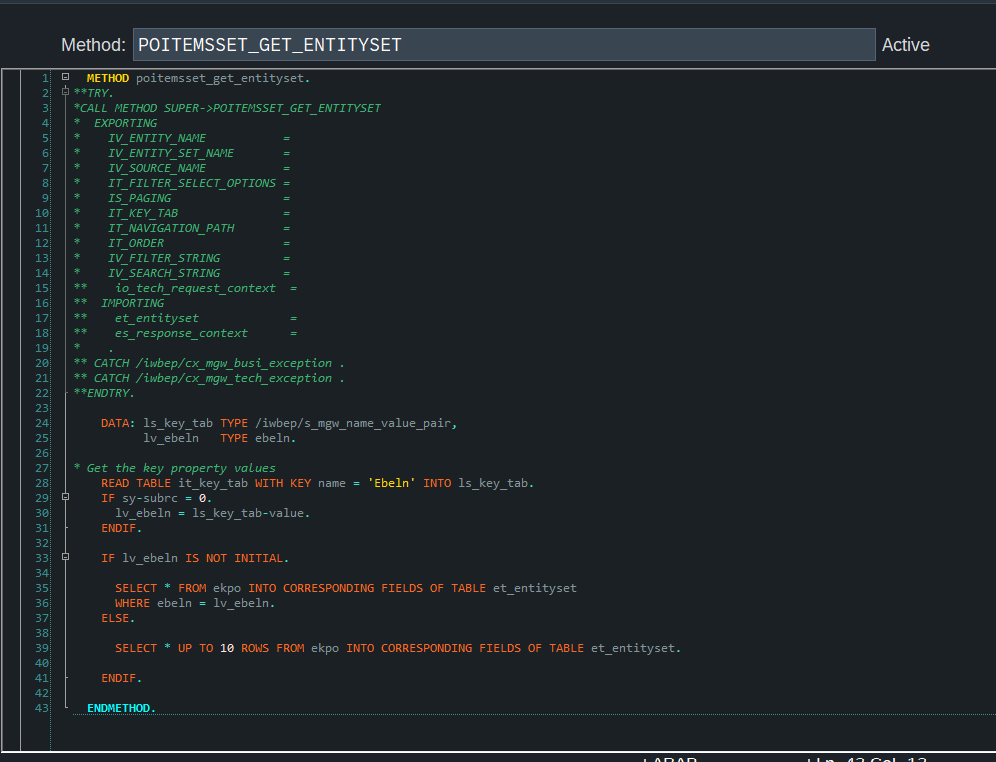
In un caso realistico, l’ideale sarebbe restituire in output sia la testata sia le posizioni con un'unica chiamata.

Si introduce quindi l’uso della “Association” e del “Navigation property” che abbiamo definito in fase di creazione del progetto. In particolare, posso aggiungere nel link di chiamata un “/<nav\_property>” quindi nell’esempio POHeader(‘4500000002’)/HeadToItemNav.

Come da screen, verranno estratte solo le posizioni dell’ordine 4500000002.



Il lavoro però non è automatico. Difatti, sarà necessario implementare l’EntitySet delle posizioni, gestendo la sia la casistica del richiamo “semplice” sia quello derivante da testata. A seguire l’esempio di implementazione.



NB, la chiamata descritta pocanzi, espone in output solo i dati di posizione (legati ovviamente alla testata specificata).

Al fine di esporre sia i dati di testata che quelli di posizione, si introduce un importante concetto, ossia la funzione $expand.

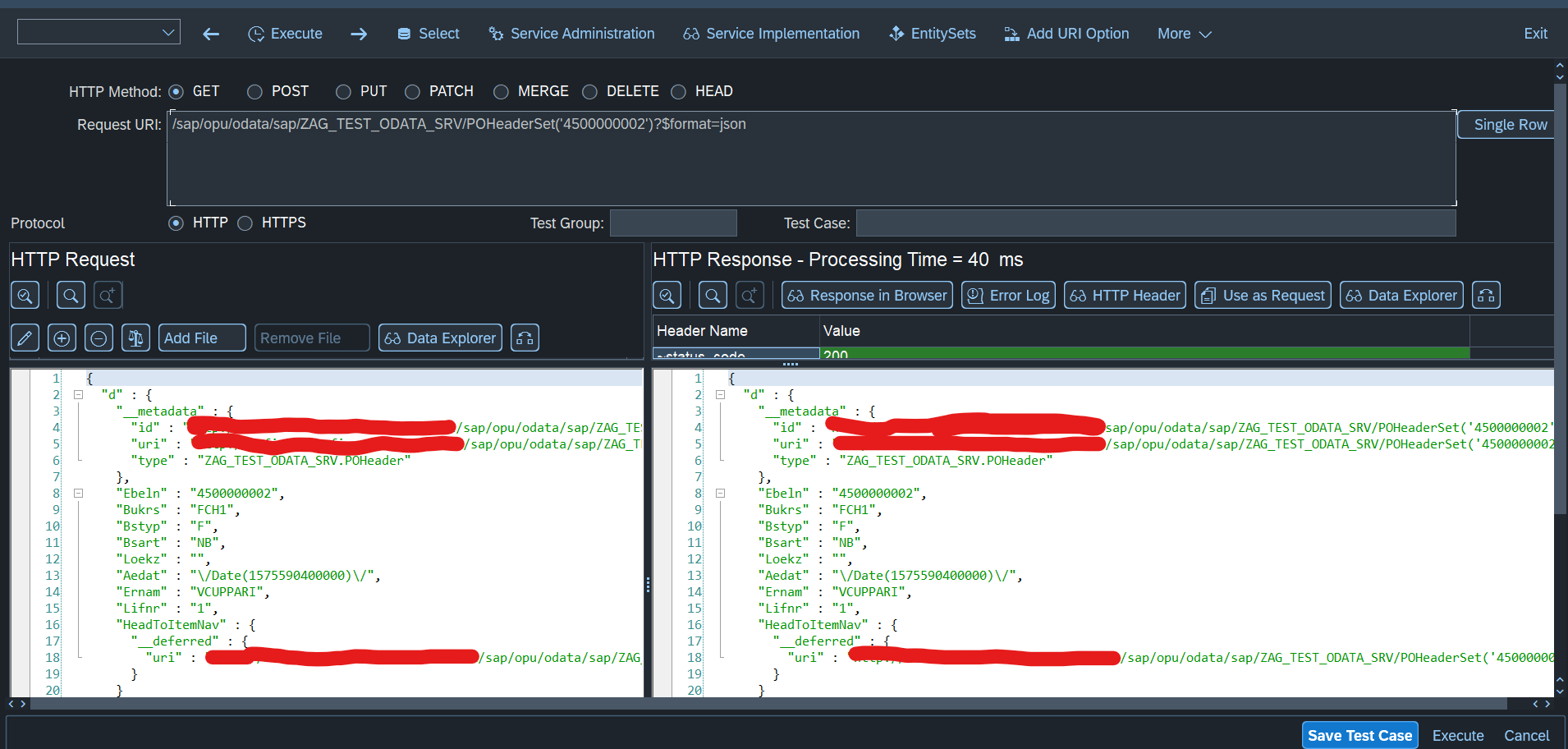
Tale funzione sarà necessaria per poter avere una GET completa del nostro Data Model utilizzabile sia per la funzione POST, sia per reperire la serie di possibili livelli di annidamento delle nostre entity.

Nello specifico, il link di chiamata sarà leggermente differente ->

<link>/POHeaderSet(‘4500000002’)/?$expand=HeadToItemNav

## POST

Il metodo POST è il metodo preposto per l’inserimento di nuovi dati nel DB. Di conseguenza, il metodo che l’OData richiamerà sarà il CREATE\_ENTITY. Come già accennato, la POST lavora per singola Entity. Per poterla utilizzare, andrebbe richiamato prima il metodo GET per ogni entity che per cui si vuole effettuare una POST e utilizzare il risultato come nuova richiesta (apposito pulsante Use As Request):



A questo punto, basta adattare i dati ed effettuare la POST ponendo attenzione a inserire nel link solo l’EntitySet, senza alcuna chiave



Attenzione: l’esempio mostrato indica come effettuare la POST per ogni Entity a sé stante.

In un caso reale la POST dovrà essere unica e dovrà comprendere sia la testata che le posizioni.

Si rimanda quindi ai prossimi paragrafi in cui verrà mostrato un esempio di progetto complesso con strutture aventi N livelli di annidamento, con una corretta gestione della messaggistica di ritorno.

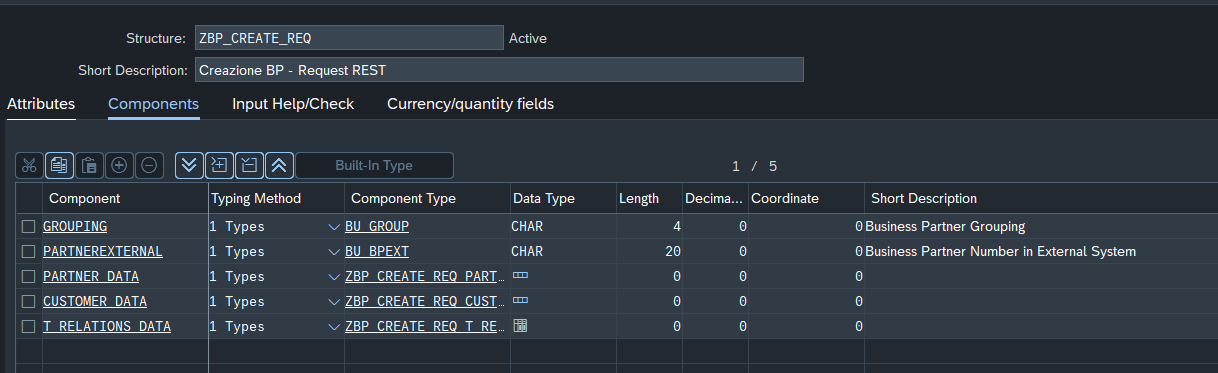
## PUT / DELETE

Senza descrizioni approfondite per evitare ripetitività, la PUT e la DELETE richiameranno i metodi di UPDATE\_ENTITY e DELETE\_ENTITY e seguiranno la stessa procedura della POST, con la differenza che sarà necessaria una chiave per poterle correttamente utilizzare.

# Creazione Progetto Complesso (Deep Entity e annidamenti)

In questa sezione verrà illustrato come strutturare l’OData per gestire in input/output delle entity complesse con N livelli di annidamento possibili. Come esempio si definirà come acquisire una struttura utile per poter creare un Business Partner.

Seguono alcuni screen di come è composta a livello macroscopico la struttura in oggetto



Graphical user interface, text

Description automatically generated

Come si evince, vi sono tipi tabella annidati in strutture, annidati a loro volta ecc.

La parte di definizione delle Entity e delle Association a livello di import da DDIC, resta identica a quella già descritta nei paragrafi precedenti.

In aggiunta, una nota sui tipi struttura: lo standard identifica i tipo struttura come “Complex Types”, e hanno una gestione leggermente differente quando vi sono annidamenti al loro interno (Approfondito dopo).

Inoltre, nel momento in cui si importa il tipo tabella, vi è un flag nella fase di selezione dei campi “Assign Structure” -> Flaggatelo.

A seguire quindi uno screen delle varie Entity Importate

Graphical user interface

Description automatically generated with medium confidence

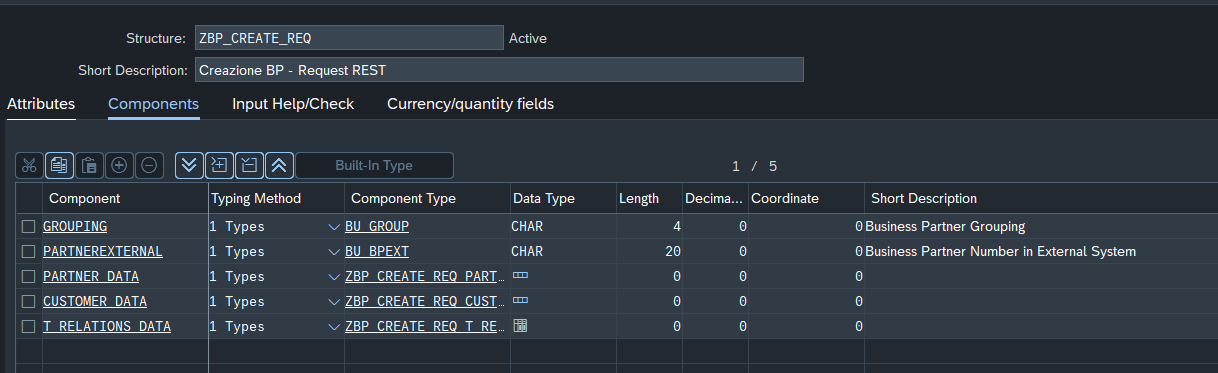
Considerata la procedura anche solo per dichiarare le Entity e le relative Association (e dovete ancora vedere quello che seguirà), questo è il motivo per il quale l’uso dell’OData è più frequente con strutture semplice di input/output. Ci sono dei work-around per semplificare il tutto, la procedura illustrata è la best practice.

Si spiegherà ora il concetto di Deep Entity e come far sì che questo particolare oggetto possa essere importato/esportato.

## Deep Entity

Una Deep Entity consiste in un Entity composita la quale viene definita a codice, combinando le Entity importate nel Model.

Si riporta nuovamente il seguente screen per comodità



Graphical user interface

Description automatically generated

Come si evince, T\_RELATIONS\_DATA è un tipo tabella annidato nella struttura ZBP\_CREATE\_REQ, avente al suo interno ulteriori tipi tabella.

Un caso reale è quello di, dato un BP in input in una richiesta GET, restituirne i relativi dati.

Se usassimo la $expand come illustrato in precedenza, potremmo solamente ricevere in output i record presenti nel T\_RELATIONS\_DATA utilizzando il Navigation Property opportunamente creato, con un link di questo tipo:

/sap/opu/odata/sap/ZDYN\_BP\_CREATE\_SRV/RequestSet('dummy')/?$format=json&$expand= ReqToRelationsNav.

Considerato che parliamo di una tabella, e quindi di oggetti con chiave, per poter addentrarci ulteriormente,

per esempio, fino a visualizzare i numeri di telefono, sarebbe necessario specificare una chiave per le relation come di seguito:

/sap/opu/odata/sap/ZDYN\_BP\_CREATE\_SRV/RequestSet('dummy')/?$format=json&$expand= ReqToRelationsNav(‘<una\_chiave>’)/ RelatToPhoneNav.

Il nostro scopo è invece quello di visualizzare tutti i record di ogni tabella dichiarata nelle Entity.

Il link finale di cui quindi abbiamo necessità, avrà la seguente strutturazione

/sap/opu/odata/sap/ZDYN\_BP\_CREATE\_SRV/RequestSet('dummy')/?$format=json&$expand=PartnerToIdentNav,PartnerToTaxnumNav,PartnerToPhoneNav,PartnerToFaxNav,PartnerToSmtpNav,CustToCompanyNav,CustToSalesNav,CustToSalesNav/SalesToFunctNav,ReqToRelationsNav,ReqToRelationsNav/RelatToFaxNav,ReqToRelationsNav/RelatToSmtpNav,ReqToRelationsNav/RelatToPhoneNav

Si evince quindi che ogni livello di annidamento va dichiarato a partire dalla sua radice separando con uno “/” e concatenando con le “,” i vari nodi.

Per poter però far sì che tale meccanismo funzioni, sono necessari alcuni interventi a codice che verranno specificati a seguire, a seconda che si desideri una GET o una POST della nostra Deep Entity.

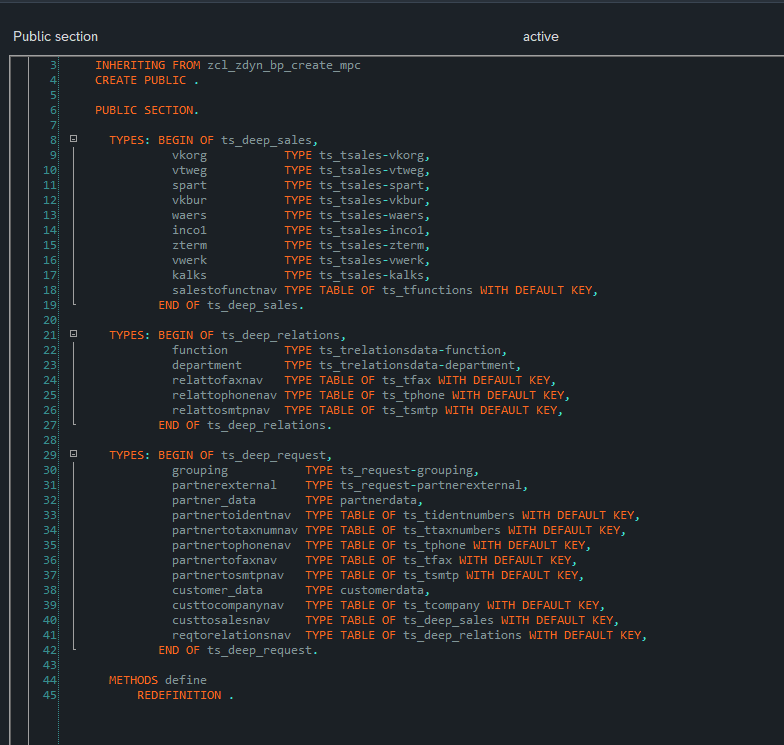
## GET Deep Entity

Per poter restituire in output alla nostra GET un file JSon completo con tutti i livelli di annidamento necessari, andranno effettuate delle ridefinizioni di alcuni elementi sia nella \*MPC\_EXT che nella \*DPC\_EXT.

### Definizione Types nella classe \*MPC\_EXT

Il primo step da eseguire sarà quello di definire dei Types all’interno della classe \*MPC\_EXT.

Nello specifico, data una struttura X con all’interno il tipo tabella Y, dovremo creare un <deep\_type> che conterrà i campi di X e in corrispondenza del tipo tabella, vi sarà il nome della Navigation Property dichiarata come Type Table Of TS\_<nome\_tipotabella\_annidata>. Nello screen di seguito un esempio



Da dichiarazioni nel Data Model sappiamo che la Request dell’esempio ha come campo il tipo tabella T\_RELATIONS (Screen nelle pag. precedenti), e sappiamo anche che è stata creata l’Association ReqToRelationsAss e il relativo Navigation Property ReqToRelationsNav. Vediamo, come da screen, che la nuova struttura DEEP\_REQUEST ha al posto di T\_RELATIONS il campo REQTORELATIONSNAV. Ma, sappiamo che anche T\_RELATIONS ha dei tipi tabella annidati. Come si può dedurre, va eseguita la stessa procedura finché non si raggiunge il livello di profondità desiderato, dichiarando le deep\_structures sostituendo i nomi dei campi tecnici in corrispondenza dei tipi tabella con i nomi delle Navigation Properties.

Nella DEEP\_REQUEST, se ricordate gli screen del DDIC del nostro esempio vi è una particolarità: l’annidamento era request->customer\_data->company. Bene, in fase di creazione delle Association vi sarete accorti che il tipo di partenza non è la struttura customer\_data/partner\_data ma bensì l’Entity Request in cui customer\_data e partner\_data sono contenute. In sostanza è come se lo standard “non vedesse” i tipi struttura come un livello intermedio. Per cui, i tipi tabella annidati ( Fax, Tel, Mail, Company, Sales) vengono associate nella deep entity al livello immediatamente sopra quello dei tipi struttura, ossia la request. Lo so, è un macello.

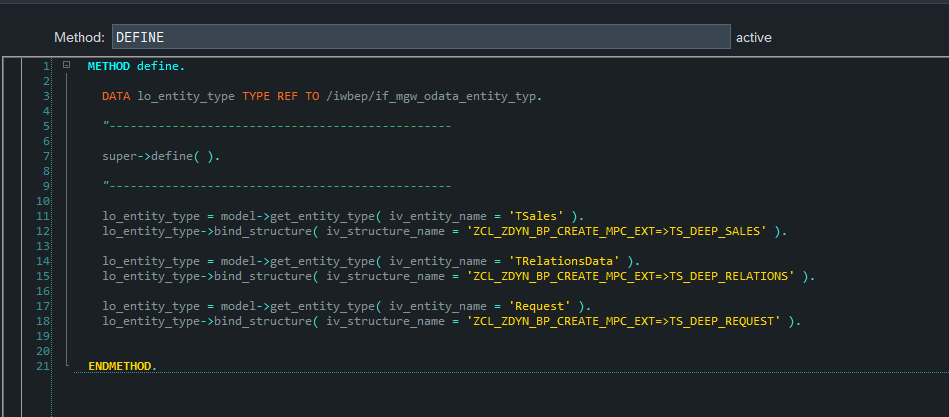
***NB: escludendo i Navigation Property, i nomi dei campi tecnici nel types dichiarato deve essere lo stesso del dictionary altrimenti la lettura del JSON in Posting non funzionerà.***

### Binding Deep Structure

Ora che i nostri DEEP\_TYPES sono pronti, va effettuato lo step successivo per gestire la nostra GET.

Sempre all’interno della classe \*MPC\_EXT, sarà necessario ridefinire il metodo DEFINE. In questo metodo, dovremo indicare allo standard quali strutture da noi definite dovrà considerare, ”sostituendo” quelle di default che andrebbe a leggere dalle Entity.

Nello screen l’esempio



NB: Richiamare il metodo super->define().

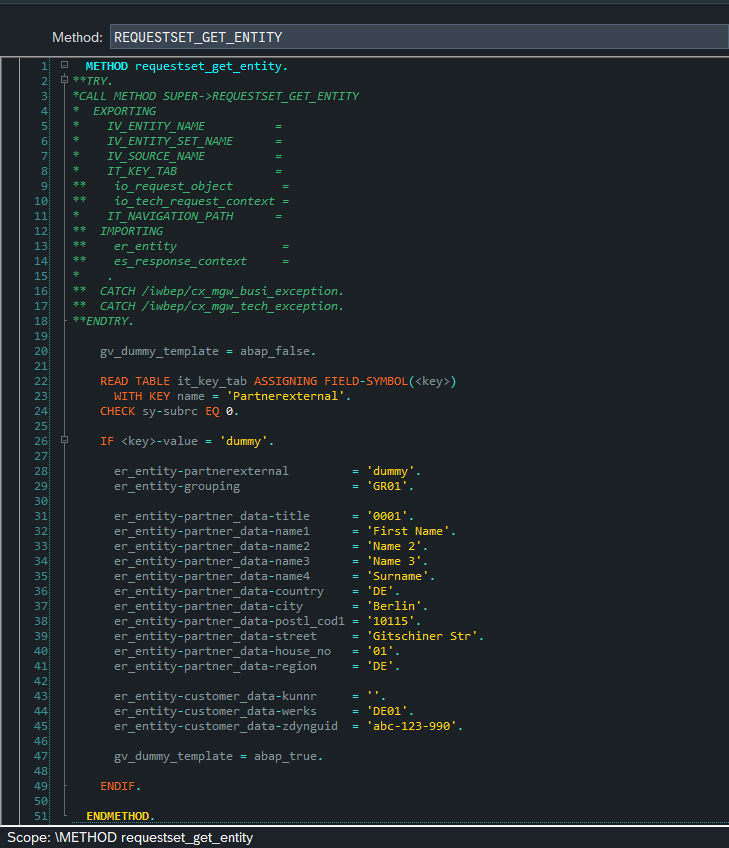
### Ridefinizione GET\_EXPANDED\_ENTITY

Nello step finale bisogna questa volta ridefinire il metodo /IWBEP/IF\_MGW\_APPL\_SRV\_RUNTIME~GET\_EXPANDED\_ENTITYSET all’interno della classe \*DPC\_EXT.

Nello specifico, il processo standard accederà prima alla GET\_ENTITY della struttura di primo livello (quindi rimanendo sull’esempio, accederà alla REQUEST\_GET\_ENTITY) e a seguire, accederà N volte nella GET\_EXPANDED\_ENTITYSET, una per ogni Tipo Tabella presente nella vostra Deep Structure finale.

Se stiamo quindi gestendo una GET per template come in questo caso, ha senso nella GET\_ENTITY di primo livello valorizzare un Flag Globale ‘dummy\_template’ per distinguere questo tipo di GET da una “Classica”.

Vedere screen di esempio a seguire



A questo punto, se tale flag è valorizzato, verranno gestite le sotto-entity nel GET\_EXPANDED\_ENTITYSET.

Potete vedere, come da screen, che i tipi struttura vengono valorizzati qui (in virtù del discorso fatto in precedenza sui deep\_types da dichiarare).

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Assicurarsi di richiamare il metodo copy\_data\_to\_ref come da screen a valle per travasare i dati in output.

### Test GET Deep\_Entity

Una volta completate tutte le definizioni dei types e dei relativi binding, siamo pronti per testare la nostra GET. Nel caso specifico utilizzeremo la GET in questione per poter ricevere in output un file JSon precompilato con dei dati ‘Dummy’, utile poi per una eventuale POST.

NB: Se non viene valorizzato un dato tipo tabella a codice, questo non verrà visualizzato nel JSon in output!

Come anticipato, il link finale avrà la seguente struttura

/sap/opu/odata/sap/ZDYN\_BP\_CREATE\_SRV/RequestSet('dummy')/?$format=json&$expand=PartnerToIdentNav,PartnerToTaxnumNav,PartnerToPhoneNav,PartnerToFaxNav,PartnerToSmtpNav,CustToCompanyNav,CustToSalesNav,CustToSalesNav/SalesToFunctNav,ReqToRelationsNav,ReqToRelationsNav/RelatToFaxNav,ReqToRelationsNav/RelatToSmtpNav,ReqToRelationsNav/RelatToPhoneNav

## POST Deep Entity

Una volta ottenuto il nostro file JSon con tutte le valorizzazioni di esempio per ogni struttura usando la nostra GET Deep Entity, sarà possibile ora effettuare una POST.

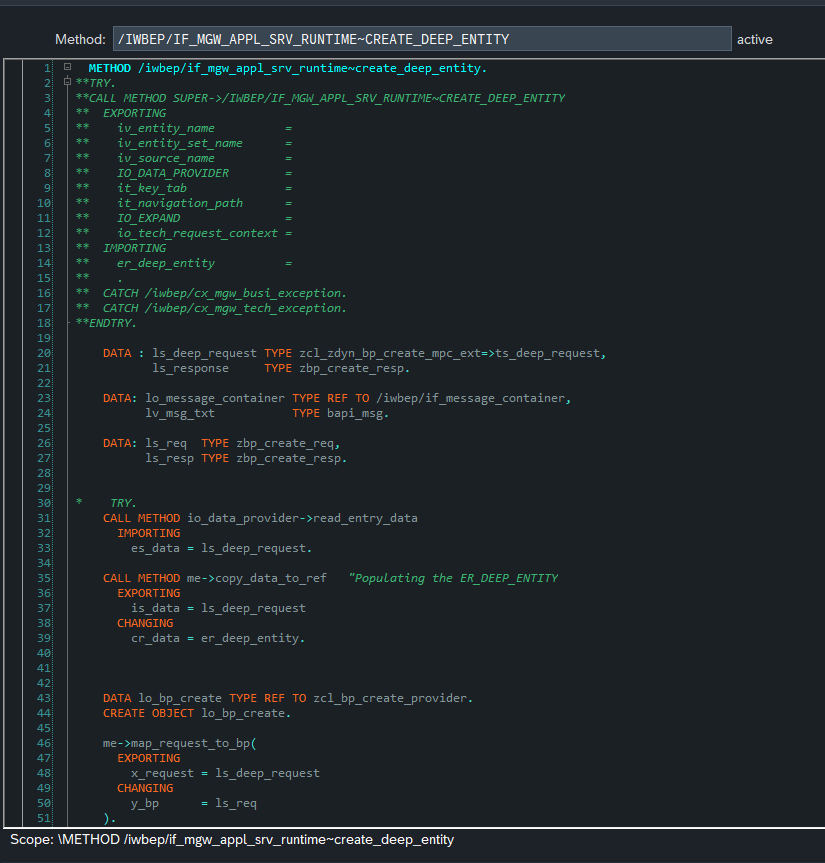
Per tale metodo, come già detto, il link avrà solo la seguente struttura

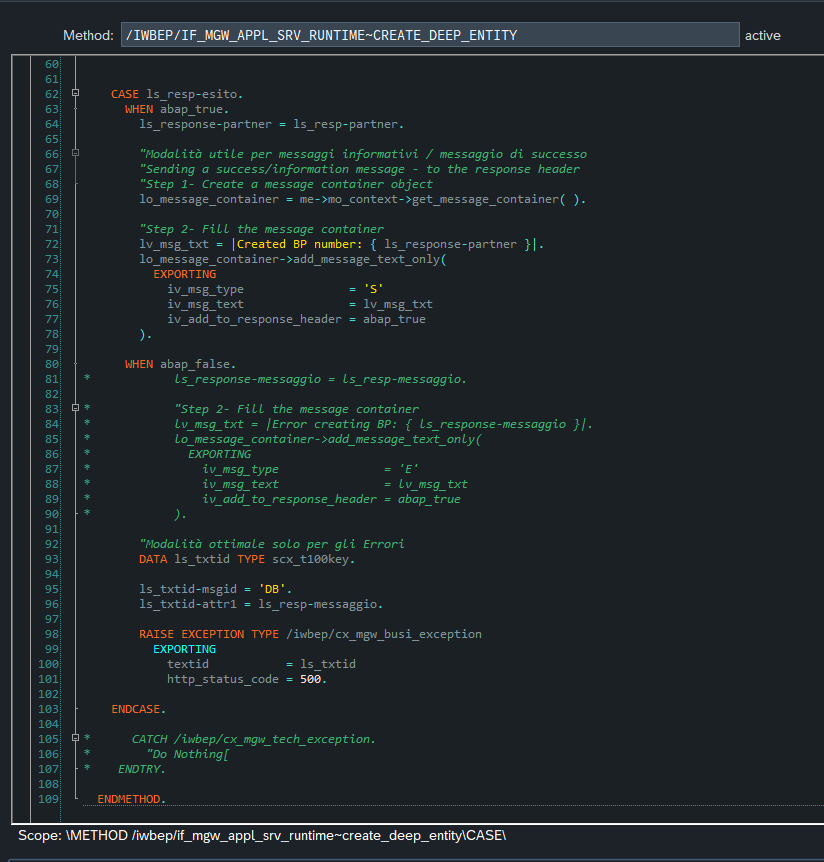
/sap/opu/odata/sap/ZDYN\_BP\_CREATE\_SRV/RequestSet.

La differenza sostanziale ora, è che lo standard rileverà automaticamente dal JSon in input, che vi è un annidamento di strutture/tabelle. Per cui, non richiamerà il metodo CREATE\_ENTITY ma bensì un’altra ridefinizione che dovremo effettuare.

Nella classe \*DPC\_EXT dovremo ridefinire il metodo /IWBEP/IF\_MGW\_APPL\_SRV\_RUNTIME~CREATE\_DEEP\_ENTITY.

Sarà questo il metodo che lo standard richiamerà automaticamente. A seguire screen e commenti





Il primo step è quindi la lettura del JSon e il travaso nella struttura interna di tipo DEEP\_STRUCTURE dichiarato nella \*MPC\_EXT (Uso dei metodi read\_entry\_data e copy\_to\_data\_ref).

Seguirà quindi il Core della vostra logica che ovviamente non vado a dettagliare (Bapi per creare il BP in questo caso) e infine la gestione dell’output di ritorno.

***NB: In output non potete dare una struttura diversa da quella in Input***

Si aprono quindi tre scenari possibili di ritorno:

1. Si vuole restituire un messaggio di successo/informativo

In questo caso, come da screen, verrà acquisita la lista messaggi del processo (riga 69 dello screen) e si aggiungerà tramite apposito metodo il vostro messaggio. Questo comparirà all’interno della header della response come parametro SAP-MESSAGE

1. Si vuole restituire un messaggio di errore

In questo caso invece, andrà effettuata una rise exception della classe /iwbep/cx\_mgw\_busi\_exception come da screen a riga 98, indicando il return code 500 (“Internal Server Error”) e il messaggio desiderato.

Come da screen, si vede una try\_catch commentata. Se viene utilizzata la rise exception non funzionerà, per cui va lasciata commentata.

1. Si necessita di un JSon in output di struttura complessa.

In questo caso nella deep\_request si dovranno già predisporre i campi che verranno valorizzati nell’output, lasciandoli vuoti in fase di input.

# Cancellazione di un progetto

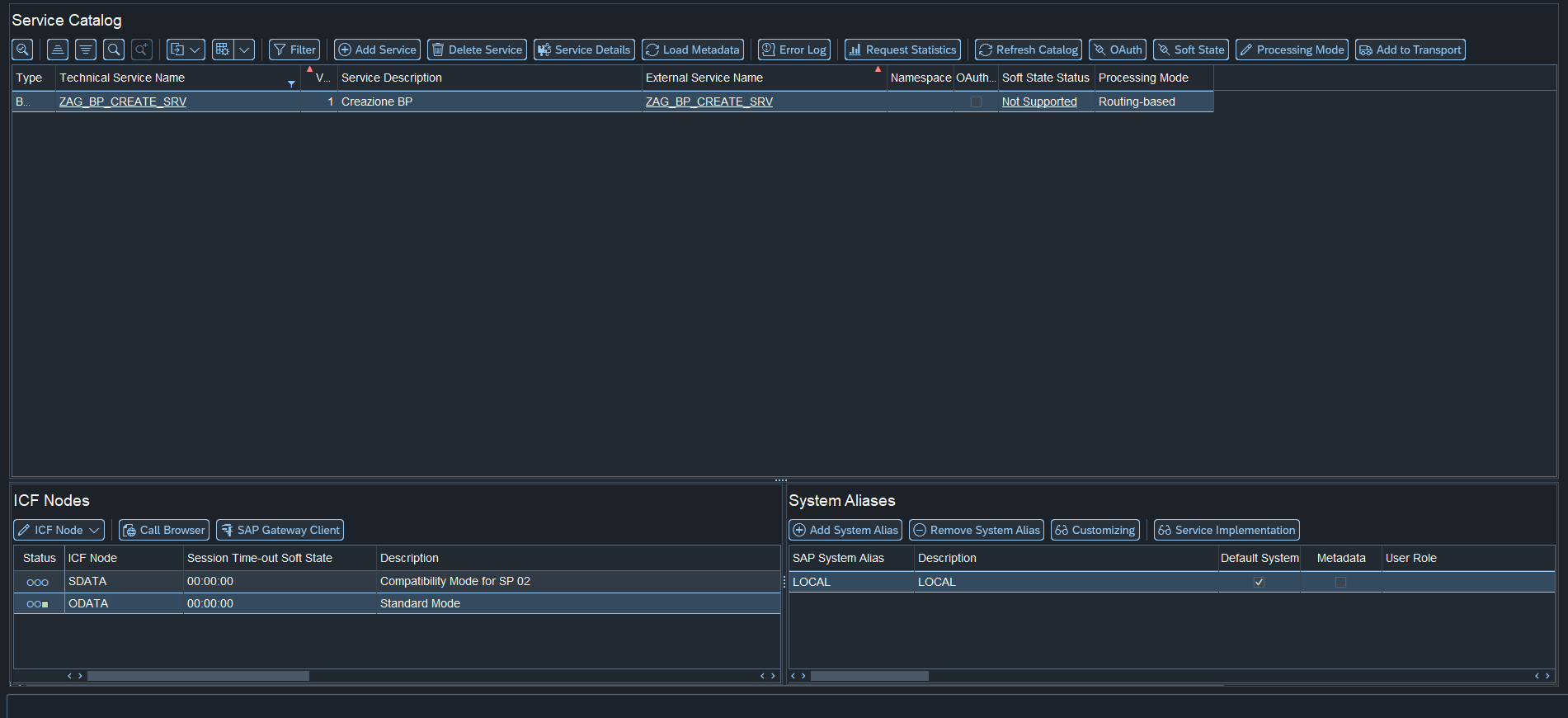
Un paragrafo per cancellare un progetto? Ma veramente? Esatto.

Purtroppo, nel momento in cui un progetto viene cancellato, SAP lascia dei residui che dovranno essere cancellati manualmente. In caso contrario, le nomenclature degli oggetti dei progetti con stesso nome, magari ricreato per questioni di ordine/pulizia, potrebbero presentare degli sfasamenti.

Quindi, a seguito della cancellazione del progetto, sarà necessario in primis cancellare manualmente le classi dalla SE24, quindi la DPC, DPC\_EXT, MPC, MPC\_EXT.

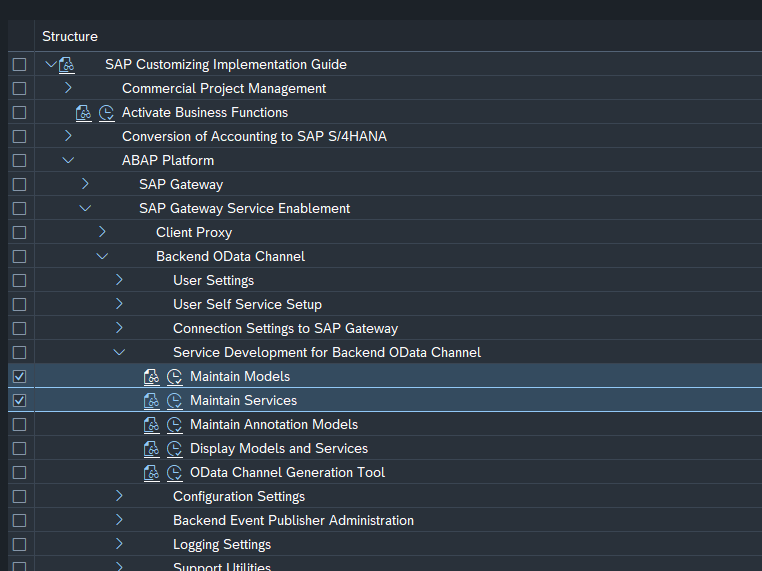
In seguito, recarsi nella transazione /N/IWFND/MAINT\_SERVICES.

Selezionare il vostro servizio,



Rimuovere prima i System Alias (in basso a destra), poi disattivare e cancellare i nodi ICF (in basso a sinistra), e solo infine, cancellare il servizio.

Ultimo step, sarà necessario accedere alla SPRO.



Accedere al ramo ABAP Platform, SAP Gateway Service Enablement (A seconda della versione, potreste trovare direttamente questa anziché ABAP Platform), Backend OData Channel, Service Development for Backend OData Channel.

Da qui:

* Maintain Service: Cancellate il vostro oggetto \_SRV
* Maintain Model: Cancellate il vostro oggetto \_MDL