# Delta Queries and Server-Driven Pagination for HANA XS OData Services

This document describes how to add delta token and skip token support to an existing XSOData service. These feature are added by means of a wrapper XSJS service that translates such requests and delegates the execution to the original service. In order to add delta support, both the original service and underlying database need to be adjusted in order to track changes, and expose them to the wrapper service. A Delivery unit with a support library is provided in order to facilitate the implementation.

The following sections will guide you through the following setup steps, and also provide additional background information:

* Importing the wrapper service utility Delivery Unit
* Creating a technical user for the wrapper service utility configuration
* Implementing change detection by means of shadow tables and database triggers
* Exposing the change detection fields as an XSOData service
* Creating a destination pointing to the adjusted service
* Creating a wrapper service for the adjusted service

# Changes

## 1.0.0

Initial release.

## 1.0.1

Fixed issue caused by duplicate headers

## 1.0.2

Fixed crash when invalid collection names were requested

Fixed crash when unsupported query options are sent to the backend

Proper support for multiply defined headers

Delta and page links are now absolute, not relative to the host

## 1.0.3

Fixed concurrent modification issue in delta token postprocessor

Fixed crash when service query path had trailing slashes

Fixed ordering problem of decorators with URL rewriting

Fixed that sometimes prevented the skip token decorator to take effect

Fixed response failing to determine media type

Fixed response failing to return correct media type

Fixed upstream request sometimes generating erroneous bodies

Fixed $batch sub-entities failing to request JSON content

Fixed a crash when retrieving single entities

Fixed a crash due to trailing slashes

Fixed a crash due to failure to read configuration

Added postprocessing error handling

Improved debug traces

# Important Notes and Limitations

## Supported Formats

Due to a limitation in the current XS Engine, the wrapper service only supports the JSON format, both upstream and downstream. If XML support is mission critical, the wrapper service library must be enhanced by content negotiation capabilities and the corresponding parsers for upstream service consumption and renderers for the consuming clients. Since this would incur an additional performance penalty, it is recommended to adopt the JSON exchange format. The only exception is the service metadata document, which by definition is in CSDL format, which only has an XML representation.

## Unsupported Parameters

The server-driven pagination wrapper will not become active when one of $top or $skip is used, and it will not become active when the user imposes an ordering on the collection. This is because custom orderings may negatively impact the data integrity. It will also strip $inlinecount results from the response, since internally it uses this value to determine if \_\_next links should be generated. Since the pagination is implemented in terms of filters, the results would not be consistent across pages either.

## Deviations from Metadata Document

In the current implementation the metadata document will reflect the changes made to the wrapped XSOData service, even though these changes are not exposed beyond the XSJS wrapper service. This cannot be easily addressed due to the format restrictions outlined above.

## Preferences

The current implementation does not support preferences as per the OData V4 specification. That means that the skip token wrapper will always use the server-side configuration for the pagination, and that the delta token wrapper will always send delta links, even If not requested by the client. In practice this should not negatively impact client applications.

## Reserved Property Names

In order for delta query support to function, designated fields containing a last-modified timestamp and an entity deletion flag need to be exposed by the XSOData service . While the names are configurable, they must be unique, since the wrapping service is deep-inspecting the XSOData response, removing those fields from the final response. Should there be a name clash, then additional properties might accidentally be stripped from the response.

In addition, it is recommended to avoid naming properties uri (lower-case specifically), since this is commonly used in metadata and links and is subject to URL rewriting.

## Removing Tombstones

Over time, the shadow table may accumulate a large number of tombstones, i.e. entries referring to deleted entries. It is recommended that a policy is implemented in order to keep this number rather small. For instance, client applications could be forced to perform a full re-sync after one month of not connecting, and server-side a daily XSJS cronjob could be used to remove entries older than a month.

## Key Property Types

Edm.Binary key property types are not supported by the wrapper library due to the missing Base64-Codec in HANA XS. This limitation only applies to server-driven pagination.

## MERGE and PATCH

Since HANA XS only explicitly supports the [HTTP methods defined in RFC 2616](https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html) (GET, PUT, POST, DELETE, HEAD, OPTIONS, TRACE and CONNECT), MERGE and PATCH requests are not supported in single (i.e. non-$batch) requests. If MERGE and PATCH requests are required, it is necessary to implement them in terms of $batch requests.

## Offline OData Best Practices

In addition it is recommended to follow the recommendations and best practices outlined on these pages:

* [Offline OData Version Support and Limitations](http://help.sap.com/saphelp_smp3012sdk/helpdata/en/88/9d29b3fac0456b812d86b5794c6e54/content.htm?frameset=/en/88/9d29b3fac0456b812d86b5794c6e54/frameset.htm&current_toc=/en/7c/01cda6700610149b10c2f2a86d335b/plain.htm&node_id=465&show_children=true#jump465)
* [Additional Best Practices for Defining Requests and Offline OData Store Provisioning](http://help.sap.com/saphelp_smp3012sdk/helpdata/en/3a/e5147e21e94908aff858772ad9639a/content.htm?frameset=/en/a7/d7b40c47024809aee453b3016650a3/frameset.htm&current_toc=/en/7c/01cda6700610149b10c2f2a86d335b/plain.htm&node_id=471)

# Setup

The following section will guide you through the steps required to enable server-driven pagination, and more importantly, delta query support to your system. The changes required are summarized in Figure 1.

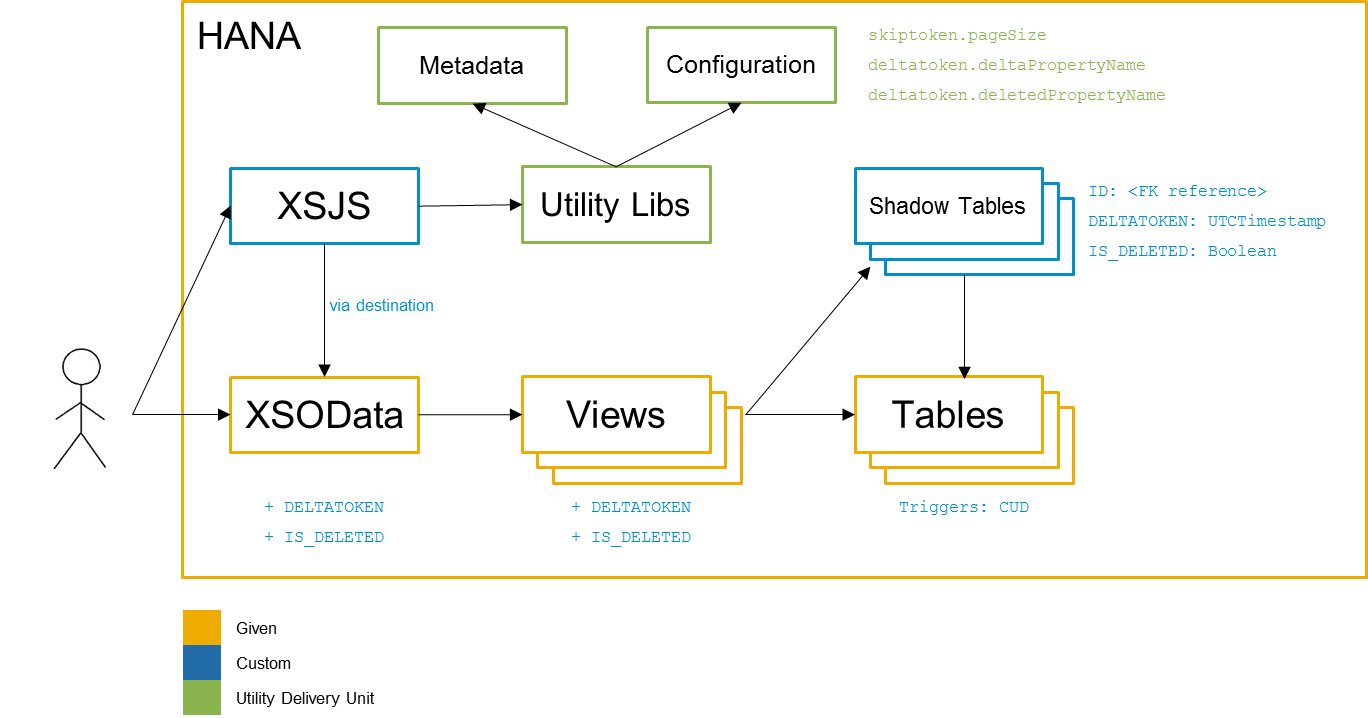


Figure 1: System change overview. If only skip token support is required, only the XSJS wrapper and the Utility Delivery Unit are required.

## Importing the Service Library

Before you can create your own XSOData service wrappers, you need to import the ODATA\_UTIL delivery unit you should have obtained together with this document as described in the [SAP Help Portal](https://help.sap.com/saphelp_hanaplatform/helpdata/en/e6/c0c1f7373f417894e1f73be9f0e2fd/content.htm).

## Creating a User for Configuration

First, create an arbitrary user as described in the [SAP Help Portal](https://help.sap.com/saphelp_hanaplatform/helpdata/en/c0/555f0bbb5710148faabb0a6e35c457/content.htm).

Next, assign the following rules to the newly created user:

* sap.hana.xs.selfService.user.roles::USSExecutor
* sap.odata.util.roles::readWrite

The former role enables the user to be used as a technical user; the latter grants read access to the actual configuration table.

Then, assign the user to the odataUtil.xssqlcc connection, which you will find in the sap.odata.util.lib package, in your HANA XS Admin cockpit (/sap/hana/xs/admin/#/package/sap.odata.util). From this moment onwards the wrapper library will be able to read the wrapper configuration table.

## Implementing Change Detection

The Change detection will be implemented using two components: One is a shadow table containing entity IDs, last-modified dates, and a flag that indicates if the corresponding entry was deleted. The second component is a number of triggers that update the corresponding information. For each of the exposed OData entities one such pair of triggers and shadow table must be implemented in order to be able to support delta token queries.

### Creating the Shadow Table

By convention, shadow tables are named like the table they are referring to, plus a “\_SHADOW” suffix. In order to create a shadow table for an existing table T1, the SQL code is therefore:

CREATE COLUMN TABLE T1\_SHADOW (ID VARCHAR(10) PRIMARY KEY, DELTATOKEN TIMESTAMP, IS\_DELETED NVARCHAR(1) DEFAULT 'N');

### Creating the Triggers

In order to keep the shadow table updated when a change occurs, we need to create one trigger for each possible change that may occur – i.e. we need to insert an entry when a new entity is created, we need to update the timestamp when an existing entity is updated, and in addition we need to set the deleted flag when an entity is deleted. For the current example of table T1, this is the corresponding SQL code:

**CREATE** TRIGGER T1\_INS **AFTER** **INSERT** **ON** T1

REFERENCING NEW **ROW** newrow

**FOR** EACH **ROW**

**BEGIN**

**INSERT** **INTO** T1\_SHADOW ( ID, DELTATOKEN, IS\_DELETED ) **VALUES**( :newrow.ID, CURRENT\_UTCTIMESTAMP, 'N' );

-- If it is possible to re-use a primary key after a delete then you will need to use the following instead.

-- UPSERT T1\_SHADOW ( ID, DELTATOKEN, IS\_DELETED ) VALUES( :newrow.ID, CURRENT\_UTCTIMESTAMP, 'N' ) WHERE ID = :newrow.ID;

**END**;

**CREATE** TRIGGER T1\_UPD **AFTER** **UPDATE** **ON** T1

REFERENCING NEW **ROW** newrow

**FOR** EACH **ROW**

**BEGIN**

**UPDATE** T1\_SHADOW **set** DELTATOKEN = CURRENT\_UTCTIMESTAMP **WHERE** ID = :newrow.ID;

**END**;

**CREATE** TRIGGER T1\_DEL **AFTER** **DELETE** **ON** T1

REFERENCING OLD **ROW** oldrow

**FOR** EACH **ROW**

**BEGIN**

**UPDATE** T1\_SHADOW **set** DELTATOKEN = CURRENT\_UTCTIMESTAMP, IS\_DELETED='Y' **WHERE** ID = :oldrow.ID;

**END**;

### Dealing with Related Entities

One might wonder how the scenario is dealt with when we have related entities, such as sales orders and sales order items. When following the Offline OData Best Practices outlined in Section “Important Notes and Limitations” it becomes clear that the Mobile Platform does not support $expand delta queries *directly*, but uses the referential constraints from the metadata document to track associations between entities. This means that when these constraints are given, as is generally the case in XSOData, no special treatment is required for related entities. Implement the basic change tracking for all collections, and you are good to go.

### Dealing with Context-Sensitive Collections

There are cases in which the results of a query do not only depend on what is specified in the request URL, but also on other information available to the server. One typical example is private data that only a specific user might see, or data than is only visible depending on the authorizations of a user. This case needs special treatment, since we are conceptually dealing with related entities, but the parent entity is not explicitly modeled in OData. Continuing the authorization example, this would be the user: Depending on the user context different results are returned, even though the users are not an explicit OData collection that has associations to the objects they are allowed to access. In this case explicit link tracking is required, i.e. one must track when a user is allowed to see an entry, and also when the access was revoked.

### Dealing with Existing Data

Of course, the triggers we created will only be set off when a new change occurs. Existing entries are neglected. In order to rectify this, we need to insert an entry for each existing entity in table T1 just once, via running:

**INSERT** **INTO** T1\_SHADOW ( ID, DELTATOKEN, IS\_DELETED ) **SELECT** PARTNERID, CURRENT\_UTCTIMESTAMP, 'N' **FROM** T1;

### Capturing Specific Fields

In order to minimize false positives in cases where only specific fields are relevant to change detection, it is possible to further optimize the update trigger. If, for instance, we only expose the NAME and DESCRIPTION fields in the XSOData service, we could use the following update trigger In order to minimize deltas:

**CREATE** TRIGGER T1\_UPD **AFTER** **UPDATE** OF NAME, DESCRIPTION **ON** T1

REFERENCING NEW **ROW** newrow, OLD **ROW** oldrow

**FOR** EACH **ROW**

**BEGIN**

**UPDATE** T1\_SHADOW **set** DELTATOKEN = CURRENT\_UTCTIMESTAMP **WHERE** ID = :newrow.ID;

**END**;

## Exposing the Change Detection

### Creating a View for Change Tracking

As of now the database may be able to track changes, but it still is no good for viewing or exposing as an XSOData service, since the information we need is split across several tables. We can fix this easily by providing a view that joins the information we want:

**CREATE** **VIEW** T1\_DELTA **AS**

**SELECT**

SHADOW.ID **AS** ID,

-- ...remaining properties to expose

DELTATOKEN,

IS\_DELETED

**FROM** T1 \_SHADOW **AS** SHADOW **LEFT** **OUTER** **JOIN** T1 **AS** T

**ON** SHADOW.ID = T.ID;

### Updating the XSOData Service

Finally, we need to adjust the XSOData service to expose the T1\_DELTA view instead of T1, or we create a new one. Going forward we will assume that the wrapper service is developed in the com.example.delta package. Create the file delta.xsodata with the following contents:

service namespace "com.example.delta" {

"YOURSCHEMA"."T1\_DELTA"

as "T1"

key ("ID");

}

## Implementing CUD

When exposing tables as OData services, full CRUD is automatically supported. However, as one might figure, changing operations are not supported by views, and therefore it is necessary to implement CUD exits for the new service. In order to do this, first create an error data type:

// com/example/delta/error.hdbstructure

table.schemaName = "YOURSCHEMA";

table.columns = [

{name = "HTTP\_STATUS\_CODE"; sqlType = INTEGER;},

{name = "ERROR\_MESSAGE"; sqlType = NVARCHAR; length = 100;},

{name = "DETAIL"; sqlType = NVARCHAR; length = 100;}

];

Then we need to implement the actual CUD procedures as follows:

-- com/example/delta/t1\_insert.hdbprocedure

PROCEDURE "YOURSCHEMA"."com.example.delta::t1\_insert" (IN new "YOURSCHEMA"."T1\_DELTA", OUT error "com.example.delta::error")

LANGUAGE SQLSCRIPT

SQL SECURITY INVOKER

DEFAULT SCHEMA "YOURSCHEMA" AS

BEGIN

INSERT INTO T1

(ID, field2, ...)

SELECT ID, field2, ... FROM :new;

IF 0 > 1 THEN

-- Room for error handling

error = SELECT 500 AS http\_status\_code,

'Internal server error' error\_message,

'Internal server error' detail FROM dummy;

ROLLBACK;

END IF;

END;

-- com/example/delta/t1\_update.hdbprocedure

PROCEDURE "YOURSCHEMA"."com.example.delta::t1\_update" (IN new "YOURSCHEMA"."T1\_DELTA", IN old "YOURSCHEMA"."T1\_DELTA", OUT error "com.example.delta::error")

LANGUAGE SQLSCRIPT

SQL SECURITY INVOKER

DEFAULT SCHEMA "YOURSCHEMA" AS

BEGIN

DECLARE id ...;

DECLARE field2 ...;

SELECT ID into id from :old;

SELECT field2 into fiedl2 from :new;

UPDATE T1 AS t

SET field2 = :email

WHERE t.ID = :id;

IF 0 > 1 THEN

-- Room for error handling

error = SELECT 500 AS http\_status\_code,

'Internal server error' error\_message,

'Internal server error' detail FROM dummy;

ROLLBACK;

END IF;

END;

-- com/example/delta/t1\_delete.hdbprocedure

PROCEDURE "YOURSCHEMA"."com.example.delta::t1\_delete" (IN old "YOURSCHEMA"."T1\_DELTA", OUT error "com.example.delta::error")

LANGUAGE SQLSCRIPT

SQL SECURITY INVOKER

DEFAULT SCHEMA "YOURSCHEMA" AS

BEGIN

DECLARE id ...;

SELECT ID INTO id FROM :old;

DELETE FROM T1 AS t

WHERE t.ID = :id;

IF 0 > 1 THEN

-- Room for error handling

error = SELECT 500 AS http\_status\_code,

'Internal server error' error\_message,

'Internal server error' detail FROM dummy;

ROLLBACK;

END IF;

END;

Finally reference the CUD handlers in your XSOData service:

service namespace "com.example.delta" {

"YOURSCHEMA"."T1\_DELTA"

as "T1"

key ("ID")

create using "com.example.delta::t1\_insert"

update using "com.example.delta::t1\_update"

delete using "com.example.delta::t1\_delete";

}

## Implementing the Wrapper Service

Now the system is prepared for the final steps, in which we actually wrap the service in a second one that adds delta query and server-driver pagination support.

### Creating a Target Destination

In order for the wrapper to be able to target the XSOData service, you need to create a destination. We assume that your application is running on instance 00 on host myhana, and that your XSJS application package is com.example.wrapper. In your application package, create the file delta.xshttpdest and add the following contents:

host = "myhana";

port = 8000;

pathPrefix = "/com/example/delta";

authType = basic;

Activate the destination file. Next, open the HANA XS Admin cockpit and navigate to your application package (/sap/hana/xs/admin/#/package/com.example.wrapper). Find the destination you just created, and configure the security settings according to your preferences. Please refer to the Security section of this document for additional information.

### Creating the Wrapper Service

With all these pieces in place, we can finally create the wrapper XSJS service. Create delta.xsjs in your application package, and paste the following contents:

**var** oDataUtils = $.import('sap.odata.util', 'decorators');

**var** destination = $.net.http.readDestination(com.example.wrapper', 'delta');

oDataUtils

.decorate(destination)

.withSkipTokens()

.withDeltaTokens()

.and.applyDecorators();

Activate the service and run some queries to see your XSJS-wrapped XSOData service with server-driven pagination and delta query support in action! In case you only want to enable either feature, just remove the corresponding line from your XSJS service. The service decorators do not depend on each other, but they may interact with each other. See the implementation notes for further information.

# Configuration

The service wrapper library supports configuration for the property names using which the last-modified timestamp and the deletion flag are exposed by the upstream XSOData service, and the server-driven pagination page size. The configuration values can be found in table ODATA\_UTIL.sap.hana.odata.util.data::config.Configuration and you will need the sap.odata.util.roles::admin role in order to modify the settings. Please refer to the below table for the available settings.

|  |  |
| --- | --- |
| **Key** | **Description** |
| skipToken.maxPageSize | Page size for server-driven pagination (default 10000) |
| deltaToken.deltaPropertyName | Name of the property using which last-modified timestamps are exposed in the XSOData service |
| deltaToken.deletedPropertyName | Name of the property using which deletion flags are exposed in the XSOData service |
| deltaToken.deletedPropertyYesValue | Value of the property using which deletion flags are exposed indicating that an entity was deleted |
| deltaToken.stripDeltaFields | Tells if the change tracking properties (as configured via deltaToken.deletedPropertyName and deltaToken.deltaPropertyName) should be stripped from entities. |
| deltaToken.replaceDeletedEntities | Tells if entities marked as deleted should be replaced with OData deletedEntity entries. |

These values can be specified at various granularities: Collection-level, service-level and globally. More specific settings always override more general settings. The below table contains a number of examples for these cases.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **SERVICE\_ID** | **COLLECTION\_NAME** | **Key** | **Value** | **Comment** |
|  |  | skipToken.maxPageSize | 10000 | Globally limit page size for server-driven pagination to 10000 rows **(default)** |
|  |  | deltaToken.deltaPropertyName | DELTATOKEN | DELTATOKEN is the globally used property identifying change tracking timestamps **(default)** |
|  |  | deltaToken.deletedPropertyName | IS\_DELETED | IS\_DELETED is the globally used property identifying the flag indicating if a property was deleted **(default)** |
|  |  | deltaToken.deletedPropertyYesValue | Y | When the property configured via deltaToken.deletedPropertyName equals Y, the entity is globally considered as deleted **(default)** |
|  |  | deltaToken.stripDeltaFields | Y | Globally removes the change tracking properties from all entities before they are sent to the client **(default)** |
|  |  | deltaToken.replaceDeletedEntities | Y | Globally replaces deleted entites with OData deletedEntity entries. **(default)** |
| /com/example/wrapper/delta.xsjs |  | skipToken.maxPageSize | 20 | Limits the page size for server-driven pagination to 20 rows, exclusively for the specified service |
| /com/example/wrapper/delta.xsjs | T1 | skipToken.maxPageSize | 20 | Limits the page size for server-driven pagination to 20 rows, exclusively for the specified collection in the specified service |

# Security

## Option 1: Technical User

The wrapped service and data can be protected easily, including CSRF protection, by creating a dedicated user for the xshttpdest the wrapper service is using. In order to avoid request overhead we will make sure that while the wrapper service is protected against CSRF attacks, the (now internal) XSOData service no longer is. Since only the technical user is supposed to access this service, this is not a security issue.

We are assuming that the wrapper service resides in the com.example.wrapper package, and the XSOData service is in the com.example.delta package. This is crucial, since we need to protect both separately.

### Protecting the Wrapper Service

In order for the wrapper service to be protected we need to do the following:

1. Create an assignable role that grants the “Access” application privilege
2. Declare the “Access” application privilege
3. Require the “Access” application privilege
4. Assign the corresponding role to all eligible users

Create the Access.hdbrole file in the com.example.wrapper package with the following contents:

**role** com.example.wrapper::Access

{

**application** **privilege**: com.example.wrapper::Access;

}

Create the .xsprivileges file in the com.example.wrapper package with the following contents:

{

"privileges" : [

{"name" : "Access", "description" : "XSOData Wrapper Named User Access"}

]

}

Adjust the .xsaccess file in com.example.wrapper package so that it reflects the following settings:

{

"authorization": ["com.example.wrapper::Access"],

"prevent\_xsrf" : true,

...

}

Finally you need to make sure that all your named users are assigned the com.example.wrapper::Access role. Assuming that you are creating a wrapper for an existing service, the easiest way is to add the newly created authorization as a role to your existing authorization. This way all users that were allowed to access the old service may now access the wrapper service.

### Protecting the XSOData Service

The steps to protect the XSOData service are basically the same as above, albeit with slightly different file contents. Most notably, it is essential that CSRF be turned off for the technical user to be able to forward updates to the XSOData service.

Create the Access.hdbrole file in the com.example.delta package with the following contents:

**role** d053411.odata.xsodata::Access

{

**application** **privilege**: com.example.delta::Access;

}

Create the .xsprivileges file in the com.example.delta package with the following contents:

{

"privileges" : [

{"name" : "Access", "description" : "XSOData Technical User Access"}

]

}

Adjust the .xsaccess file in com.example.delta package so that it reflects the following settings:

{

"authorization": ["com.example.delta::Access"],

"prevent\_xsrf" : false,

...

}

Now create a designated technical user for the XSOData access, e.g. ODATA\_UTIL\_HTTP, and assign the com.example.delta::Access role to it. Then configure this user as a technical user with Basic authentication for the destination created in Section “Creating a Target Destination”.

## Option 2: Assertion Propagation

Alternatively you may configure SAP Logon/Assertion Ticket authentication in the HANA XS Admin UI for both the wrapper and the XSOData service and configure SAP Assertion Ticket as the authentication method for the destination pointing to the XSOData service. Please note that in this scenario the end user needs to be allowed to access both, the wrapper and the XSOData service. It is also still necessary to have CSRF disabled for the XSOData service as described in Option A: Technical User, so extra care should be taken that no external application accidentally consumes the CSRF-unprotected XSOData service directly. Protection via CSRF is still guaranteed via the wrapper service.

Please refer to the [help portal for additional information about the required configuration](https://help.sap.com/saphelp_hanaplatform/helpdata/en/59/b77dc9fd7c45299c2fe87741c88405/content.htm).

# Versioning

Since the XSOData metadata document must be retrieved and cached in order for the server-side pagination to work reliably, the OData utility libraries are sensitive to changes to the underlying XSOData service. It is generally recommended to follow the [OData v4 specification concerning model versioning](http://docs.oasis-open.org/odata/odata/v4.0/errata02/os/complete/part1-protocol/odata-v4.0-errata02-os-part1-protocol-complete.html#_Toc406398209) in order to guarantee that a change to the XSOData service does not negatively affect the wrapper service.

# Addendum: Implementation Notes

## URL Rewriting

URLs are rewritten if and only if all of the following conditions apply:

1. They occur in a document other than the metadata or service document
2. They occur in a property name uri (lower-case)
3. They refer to an XSOData service with the same base name as the XSJS service

## Delta Token Calculation

### Time Zones

As can be seen in the trigger definitions, the OData utility libraries implement delta calculation based on the UTC timestamp of changes. This should be noted when working with existing change tracking implementations.

### Snapshot Isolation

Under rare circumstances it is possible for deltas being missed if the current request time is being used as a delta token, viz. if there is a lengthy ongoing updating transaction that has set off some of the change detection triggers, but has not committed the changes while the request was carried out. Therefore the delta token calculation is carried out in the database as follows:

**SELECT** LOCALTOUTC(

**IFNULL**( **MAX**(START\_TIME), NOW() ),

(**SELECT** **VALUE** **FROM** M\_HOST\_INFORMATION

**WHERE UPPER**(**KEY**) = 'TIMEZONE\_NAME' **LIMIT** 1) ) **AS** DELTATOKEN

**FROM** SYS.M\_TRANSACTIONS

**WHERE** **UPPER**( TRANSACTION\_STATUS ) <> 'INACTIVE'

**AND** UPDATE\_TRANSACTION\_ID > 0

## Interactions between Decorators

The decorators implementing delta queries and server-driven pagination will interact with each other as follows:

* The delta query decorator will leverage the pagination start time encoded in the skip tokens in order to provide delta links relative to the pagination start time
* Delta queries themselves may be paginated if there is an exceedingly large number of changes, and the server-driven pagination decorator will reliably pick up delta tokens if available in order not to accidentally replace delta tokens with the pagination start time.
* The order in which the decorators are applied is irrelevant, since both interactions depend solely on the request parameters, not on the presence of either of the wrappers.

For an example of the interaction, see the “Get Buyers Delta (page 2)” sample query in the Postman test suite.