

DevTest Solutions - 8.3

OData Dynamic Virtual Services (DVS)

Date: 11-Aug-2015



This Documentation, which includes embedded help systems and electronically distributed materials, (hereinafter referred to as the "Documentation") is for your informational purposes only and is subject to change or withdrawal by CA at any time. This Documentation is proprietary information of CA and may not be copied, transferred, reproduced, disclosed, modified or duplicated, in whole or in part, without the prior written consent of CA.

If you are a licensed user of the software product(s) addressed in the Documentation, you may print or otherwise make available a reasonable number of copies of the Documentation for internal use by you and your employees in connection with that software, provided that all CA copyright notices and legends are affixed to each reproduced copy.

The right to print or otherwise make available copies of the Documentation is limited to the period during which the applicable license for such software remains in full force and effect. Should the license terminate for any reason, it is your responsibility to certify in writing to CA that all copies and partial copies of the Documentation have been returned to CA or destroyed.

TO THE EXTENT PERMITTED BY APPLICABLE LAW, CA PROVIDES THIS DOCUMENTATION "AS IS" WITHOUT WARRANTY OF ANY KIND, INCLUDING WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NONINFRINGEMENT. IN NO EVENT WILL CA BE LIABLE TO YOU OR ANY THIRD PARTY FOR ANY LOSS OR DAMAGE, DIRECT OR INDIRECT, FROM THE USE OF THIS DOCUMENTATION, INCLUDING WITHOUT LIMITATION, LOST PROFITS, LOST INVESTMENT, BUSINESS INTERRUPTION, GOODWILL, OR LOST DATA, EVEN IF CA IS EXPRESSLY ADVISED IN ADVANCE OF THE POSSIBILITY OF SUCH LOSS OR DAMAGE.

The use of any software product referenced in the Documentation is governed by the applicable license agreement and such license agreement is not modified in any way by the terms of this notice.

The manufacturer of this Documentation is CA.

Provided with "Restricted Rights." Use, duplication or disclosure by the United States Government is subject to the restrictions set forth in FAR Sections 12.212, 52.227-14, and 52.227-19(c)(1) - (2) and DFARS Section 252.227-7014(b)(3), as applicable, or their successors.

Copyright © 2015 CA. All rights reserved. All trademarks, trade names, service marks, and logos referenced herein belong to their respective companies.

Table of Contents

OData Dynamic Virtual Service Features	8
Method Support	8
POST	8
GET	10
PUT	11
DELETE	12
Resource Paths Supported	12
\$value	13
\$count (resource path keyword)	13
\$ref	13
\$link	14
Special Resource Paths	15
Service Document	15
\$metadata	15
System Query Options	17
\$format	17
\$filter	18
\$select	18
\$orderby	19
\$skip	20
\$top	20
\$expand	21
\$count (System Query Option)	22
Filter Operators	22
Built-in Filter Functions	23
String functions	23
Date and Time Functions	25
Mathematical Functions	26
Request and Response Headers	26
Odata-Version	27
OData-MaxVersion	27
Prefer	27
return=minimal	27
return=representational	27
Location	27
OData-EntityId	28

Extensions Outside of OData Standard	28
Automatic Generation of Property Values	28
REST API to Manage Data	28
Custom Properties Defined at Run-Time	28
Defining a Custom Property	29
Working with Custom Properties	31
.....	31
DVS Limitations	31
Known Issues	32

Building Dynamic Virtual Service 33

Prerequisites	33
Java SE Development Kit (JDK)	33
Apache Ant	34
Install Required Ant Extensions	34
Install Apache Maven	35
Build the DVS Components	36
Next Steps	37

Installing Dynamic Virtual Service 38

Prerequisites for Installing the DVS Eclipse Plug-in	38
Installation	38
Configuring the Eclipse Plug-in	39
Installing the DVS Servlet	40
Prerequisites	40
Installing the DVS Servlet	40
Configuring the DVS Servlet	40
.....	41
Verifying DVS Servlet Operation	41
Installing the OData Dynamic Virtualization Services Extension	42
Installing the Example Bookshop DevTest Project	42
Next Steps	43

Designing Your Virtual Service 44

Defining a VS in AEDM Format	44
Element Names	44

Elements of an AEDM	45
Schema	45
EntitySet	45
EntityType	45
Property	46
NavigationProperty	48
Association	48
EnumType	49
ComplexType	49
Defining a VS in RAML Format	50
Metadata declaration	50
Sample Data	51

Deploying Your Virtual Service 53

Prerequisites for Deploying When Using the DVS Assistant (Eclipse Plug-in) and DevTest Solutions	53
Creating the Virtual Service MAR File Through Eclipse	53
Deploying the MAR File	55
Prerequisites for Deploying When Using the DVS Servlet	55
Creating and Deploying	56
Working with the MAR as a DevTestProject	56
Removing a Virtual Service	57
Next Steps	57

Managing Your Virtual Service Data 58

Runtime Management of Virtualization Data	58
Web Client Requests	58
Unsupported Requests	60
Backing up Data	61

OData Dynamic Virtual Services (DVS)

OData Dynamic Virtual Services (DVS) is an extension to CA Service Virtualization and CA Application Test. DVS allows users to create OData service virtualizations that are hosted in a virtual service environment (VSE) that behave as real OData services.

You use POST, PUT, and DELETE to create, modify, or delete OData Entity Instances (records). Subsequent data retrieval through GET reflects these updates. With limited exceptions imposed by DevTest full OData v4, [OData Minimal Conformance Level](#) for an update service is provided with many intermediate and advanced features.

You manage the data that you create through a REST API to save and load useful data sets. Development and QA use DVS in creating repeatable tests for applications using OData services without requiring an instance of a real service.

DVS consists of the following two components:

- Eclipse plug-in that allows designers to create the MAR file.
The MAR file instantiates the OData service from either a RAML or a configuration file (AEDM). This file is similar to an Entity Data Model file.
- WAR file that can be deployed into a web server instance such as Tomcat, that supports a REST API.
This file allows direct creation and deployment of an OData virtual service from a RAML.

DVS uses an extension to CA Service Virtualization named the OData Virtual Services Extension which provides the OData emulation.

Click any of the following images for additional information:



OData Dynamic Virtual Service Features

DVS provides support at the [OData Minimal Conformance Level](#) for an updatable OData v4 service. Limited exceptions are listed in the [DVS Limitations](#) section. DVS also supports many intermediate and advanced features. Legacy OData v3 compliance to the same level is provided with exceptions, but is not being actively developed. See [DVS Limitations](#) for a list of limitations in the DVS implementation.

Most of the example URLs given in this document use the **Bookshop** example service. We recommend that the Bookshop example service is installed in one of your VSE. Try out the examples and play with the virtual service (VS) using an interactive REST client such as *PostMan* or *Advanced Rest Client*. Remember, you can restore the example data to its original state. Restore the data by either stopping and restarting the VS or by reloading the data using the Administration API from `seeddata.sql`. In most cases, the example URLs have the protocol, server, port, and base path to the service elided for brevity. If the full URL is `http://myVSEserver:8844/odata/v4/Bookshop/Authors`, the example is written as `.../Authors`. When trying the example, replace the `...` with the elided portions of the URL. Ensure accuracy when specifying your server name in place of `myVSEserver`. All examples use GET as the HTTP method unless explicitly noted. Most examples are based on the Bookshop example project that is provided with DVS.

You should be familiar with OData terminology and the meaning of the following terms:

- Entity set
- Entity type
- Property
- Navigation property

See www.odata.org for current **external** documentation for OData related topics, example service implementations, and other useful resources.

Method Support

HTTP methods POST, GET, PUT, and DELETE are used to create, read, update, and delete OData Entity Instances (records). For all requests that include a request body, specify **Content-Type:application/json** in the request header.

POST

POST is used when creating records. The resource URL must be an entity set URL. The resource can also be a URL whose final segment is a navigation property that points to a collection of entity types. In the second case, an implicit linkage is made between the entity instance (record) whose navigation property was used, and the new record. In each case, the call must include a request body in JSON format that contains at least one property for the entity type the resource path resolves. The key and

required properties must be present, except for those key and required properties the model configuration (AEDM file) for the VS defines as automatically generated. Property names must be in double quotes and are case-sensitive. Property values for string and date time types must be in double quotes. Numeric and Boolean types can either be quoted or unquoted. The presence of an invalid property name causes the request to fail with status **400**.

You can perform a **deep insert** by optionally including in the request body the additional record data for other new records to be created simultaneously as the **parent** record. Each of these records is implicitly linked through a navigation property in the **parent** record.

By default, result of POST is echoed back in the response body. This result is useful when some properties have values that are generated or have defaults.

A successful POST creates a new record and returns status **201**.

Example 1 – Create a new author record by posting directly to the entity set.

POST ../Authors

```
{
  "Name": "Anonymous",
  "DOB": null,
  "DOD": null,
  "Bio": "Formerly the most prolific author of all."
}
```

POST is the http method, ../Authors is the abbreviated URL which would be http://myVSEserver:8844/odata/v4/Bookshop/Authors in actual use and the block that is delimited by the curly braces the request body.

Example 2 – Create a book record and create an implicit link between that book and author by posting through a navigation property in the parent record.

POST ../Authors('101')/Books

```
{
  "Id": "FIC-101-005",
  "AuthorId": "101",
  "Title": "Pickwick Papers",
  "Synopsis": "Still funny after all these years",
  "Weight": 1.0,
  "Size": {
    "Height": 9.0,
    "Width": 5.5,
    "Thickness": 1.0
  }
}
```

Example 3 – Create a book record and create a comment on the book simultaneously. This example is using **deep insert** to create multiple records at a time.

Note: In this example, because the **Inventory** navigation property is **one to one** and can only reference at most one record, the navigation property value is a single JSON object with the record value. Where the navigation property is **one to many**, or **many to many**, the navigation property points to a collection of records. The value of the navigation property is a set of zero or more comma separated record instances as JSON objects, all within square brackets braces.

POST ../Books

```
{
  "Id": "FIC-101-006",
  "AuthorId": "101",
  "Title": "The Old Curiosity Shop",
  "Synopsis": "Schmaltz sells.",
  "ListPrice": 19.95,
  "Weight": 1,
  "Size": {
    "Height": 9,
    "Width": 5.5,
    "Thickness": 1
  },
  "Inventory":
  {
    "odata.type": "Inventory",
    "Id": "FIC-101-006",
    "Price": 14.99,
    "InStock": 6
  }
}
```

Note : DVS has a defect where the response body for a deep insert only contains the **parent** record.

POST is used to create references (linkages) between records. See **\$ref** for this usage of POST.

GET

Used to return either single Entity Instances (records), or Collections of records, and in concert with **\$expand** can return records or Collections of records from multiple resource paths. Data that reaches a resource path is returned in JSON format, with scope and presentation of the data returned can be further controlled through System Query Options **\$filter**, **\$select**, **\$format**, **\$top**, **\$skip**, **\$count** and **\$orderby**, and also by certain request headers.

GET can also return a single property when the resource path resolves to one specific property instance. If this property is not a collection, or a complex property, **\$value** can be used in the URL to limit response to only the value data. (see **\$value**)

GET is also used to return a service document, or metadata about the OData service. See Service Document and **\$metadata** for more information.

A successful GET returns status **200 OK**..

For examples of using GET, see the information under System Query Options, Filter Operations, and Filter Functions.

PUT

Used to perform a full update in-place update of a single record. Request body is in JSON format, and all required properties **must** be present. Key properties can be omitted and always retain their original value. Any non-key properties absent from the request body are set to **null**. A successful PUT, by default returns no response body and have a return code of **204** No Content. If a Prefer request header is specified and contains the value return=representational, then the updated record is returned, and the return code is **200 OK**.

Example 1 – Update author with Id '103'. Bio is set to a new value. DOB is set to the value previously held. Id retains the value of 103 and would not be changed even if specified in the request body. DOD is set to null because it is not specified and is not a key property.

PUT ../Authors('103')

```
{
  "Name": "Julius G. A. Payder IV ",
  "Bio": "Unpublished aspiring author",
  "DOB": "1956-07-04",
}
```

PUT can also be used to update a single property. This property can be either a simple property, a collection, or a complex property. DVS has a limitation in that complex properties that are included within another complex property cannot be updated. Nor can any property under such a complex property be updated.

Example 2 – Update a Complex Property within the Book instance with key 'FIC-101-001'.

PUT ../Books('FIC-101-001')/Size

```
{ "value": { "Height": 9, "Width": 5.5, "Thickness": 0.2 } }
```

Example 3 – Example of a syntactically valid PUT that fails due a DVS implementation limitation.

PUT ../Books('FIC-101-001')/Size/Thickness

```
{ "value": 0.2 }
```

The message, **Multiple level of complex type property is not supported yet** with status **501** Not yet implemented is returned.

DELETE

DELETE is used to delete a single record. DVS prevents an entity being deleted if it has linkages with one or more other entities. Until the links to those entities are removed to assure referential integrity is maintained. DELETE can manage linkages between records. See [\\$ref](#) for this usage of POST. DELETE can also be used to set individual properties to **null** or collection properties to an empty set. Required or key properties cannot be set to null. The same limitations that apply to single property PUT updates apply to DELETE.

A successful DELETE returns status **204** No Content.

Example 1 – Delete the Book with Id 'FIC-101-001'

```
DELETE ../Books('FIC-101-001')
```

Example 2 – Set the SkillSet of employee 'EMP00127' to the empty set.

```
DELETE ../Personnel('EMP00127')/SkillSet
```

Resource Paths Supported

For any OData service virtualized by DVS, **all** resource paths that originate from an entity set and pass through the navigation properties defined by the developer for the OData VS are supported. The resource path does not need to be defined in the DevTest VSI to be processed. DVS calculates the valid paths using the EDM defined in the EDMtoDB.xml configuration file for the VS. Calls against the VS can use URLs that pass through any number of Navigation Properties (NPs) to any depth, as long as the NPs are valid for the type containing the NP, **and** the path is syntactically valid per OData standards. For example, OData only permits navigation properties to collections that are not qualified by a key as the last resource segment of a URL. DVS correctly returns an error when processing a URL that violates that rule.

Navigation properties can be defined as either one way or bidirectional. Properties can map one to one, one to zero or one, one to many, many to one, or many to many relationships between entity instances (records) that can be in the same or different entity sets.

Resource path can terminate in a resource that resolves to either an entity set, an entity type instance (record), a single entity type instance property, or a collection property.

Certain keywords can also appear in resource paths that resolve to user-defined resources. Only one of these keywords can be present in any (valid) URL.

Specific instances within an entity set or a navigation property that resolves to an entity set is selected by appending (*keyvalue*) or (*keyprop=keyvalue*) to the resource segment. The second syntax is required if there are multiple key properties with each *keyprop=keyvalue* pair that is separated by a comma. Non-numeric key values must be within single quotes.

\$value

\$value is used with GET and must be the last segment in the resource path. Like all Single Property requests, the Resource path (URL) must otherwise resolve to a single Property instance. For example:

h `http://myVSEserver:8769/mybaseresourcepath/myEntitySet('A_Key_Value')/anEntityProperty / $value`

Causes response body to be simply the value of the requested Property, in text/plain form without any surrounding JSON formatting.

\$count (resource path keyword)

\$count (resource path keyword) can only be used with GET and must be the last segment in the resource path. The resource path must otherwise resolve to a collection (a set of zero or more records).

For example:

`http://myVSEserver:8769/mybaseresourcepath/myEntitySet /$count`

Causes response to be the integer count of the number of matching items in text/plain form without any surrounding JSON formatting. The value that is returned reflects the effect of any **\$filter**, **\$skip**, and **\$top** System Query Options present as parameters.

\$ref

Available for OData version **4.0** compatible virtual services only. \$ref must be the last segment in the resource path. \$ref changes the meaning of the URL from referring to the object to a reference of the object.

\$ref is used with POST, PUT, and DELETE, and is used to manage navigation links between entity instances.

The OData standard requires that a navigation property resource path segment to immediately precede the \$ref keyword because the navigation property specifies what association is modified. It is valid for multiple navigation properties to exist between the same two entity sets. For example, an employees entity set can contain records of entity type employee where the employee has both a supervisor and staff navigation property pointing back to the employees entity set. While both would link records of the same type, they would represent different semantics.

If the navigation property was defined with a partner, both the navigation property that is specified, and its partner are affected. For example, if the supervisor represented an employees direct manager, it would be defined as the partner navigation to staff. The staff navigation property be a one to many relationship. An employee can have zero or more employees reporting to them, but an employee would only have one direct supervisor.

Examples:

POST .../WorkTeams('Associates') / Members/\$ref

With a request body of the following:

```
{
  "url" : "Personnel('EMP00133')"
}
```

Would explicitly associate the Personnel record with Id 'EMP00133' with the WorkTeam record with key 'Associates'.

DELETE .../WorkTeams('Associates') / Members/\$ref is the reverse of the POST example and removes any association that had been established between the two records through the navigation property.

Both will return a Status 204 "No Content" on success.

Note: DVS does not support the \$id keyword. Resource URLs containing \$id as a parameter are not supported.

\$link

\$link is available for OData version **3.0** compatible virtual services only. OData version **3** is the version 3 equivalent to \$ref, and functions in a similar way. One notable difference is that the **\$link** keyword appears **before** the last resource path segment, which **must** be a navigation property.

Two Examples:

POST .../myEntitySet('EndpointOneKey')/\$links/NPName

With a request body of the following:

```
{
  "url" : "TargetEntitySet('keyvalue')"
}
```

Would explicitly associate the record in Entity Set myEntitySet with key 'EndpointOneKey' to the record in TargetEntitySet with key 'keyvalue'.

DELETE .../myEntitySet('EndpointOneKey') / \$links/NPName('keyvalue') is the reverse of the POST example that is given previously, and removes any association that had been established between the two records through the navigation property.

Both return a Status 204 "No Content" on success.

Special Resource Paths

In addition to resource paths that resolve to user-defined resources, two special resource paths are provided.

Service Document

A GET to the base URL of the OData VS returns a **Service Document** in JSON format, which is a list of the entity sets defined in the VS.

For example: GET <http://myVSEserver:8844/odata/v4/Bookshop> returns 200 OK with a response body of

```
{
  "@odata.context": "http://myVSEserver:8844/odata/v4/Bookshop/$metadata",
  "value": [
    {
      "name": "Customers",
      "kind": "EntitySet",
      "url": "Customers"
    },
    {
      "name": "Comments",
      "kind": "EntitySet",
      "url": "Comments"
    }
  ],
}
```

\$metadata

A GET to the base URL of the OData VS followed by the key word **\$metadata** returns an XML representation of the Entity Data Model schema for the VS.

See [OData Version 4.0 Part 3: Common Schema Definition Language](#) for a full description of the elements that can be returned from a \$metadata.

In its **\$metadata** response, DVS returns

Element	Child Elements
EnumType	Member
ComplexType	Property
EntityType	Key, Property, NavigationProperty
EntitySet	NavigationPropertyBinding

For example: GET [http://myVSEserver:8769/mybaseresourcepath/\\$metadata](http://myVSEserver:8769/mybaseresourcepath/$metadata) can return a response similar to the following response:

```

<SchemaName version=1.0>

<EnumType Name="SomeEnumeratedType" UnderlyingType="Edm.Int32">

<Member Name="4", Value="0" />

. . . more Member definitions ..

</EnumType>

... more EnumType definitions ...

<ComplexType Name="SomeComplexType1">

<Property Name="APropertyForThisType" Nullable="false" Type="Edm.String",

<Property Name="AnotherPropertyForThisType" Nullable="true" Type="Edm.Int32",

... more Property definitions ...

</ComplexType>

... more ComplexType definitions ...

<EntityType Name="SomeEntityType1">

<Key>

<PropertyRef Name="AKeyField">

</Key>

<Property Name="AKeyField" Nullable="false" MaxLength=16 Type="Edm.String" />

<Property Name="AFieldwithDefault" Nullable="false" DefaultValue="something" Type="Edm.String"
/>

... more Property definitions ...

<NavigationProperty Name="NPName" Partner="NameOfReverseNavigationInTarget" Type="
Collection(SchemaName.TargetEntityType)" />

<NavigationProperty Name="NPName2" Type="SchemaName.YetAnotherTargetEntityType" />

... more Navigation Property definitions ...

</EntityType>

... more EntityType definitions ...

<EntityContainer>

<EntitySet Name="SomeEntitySet1" EntityType="SchemaName.SomeEntityType1" >

```



```

<NavigationPropertyBinding Path="NPName", Target="TargetEntitySet" />

<NavigationPropertyBinding Path="NPName2", Target="YetAnotherTargetEntitySet" />

... more Navigation Property Binding declarations ...

</EntitySet>

... more EntitySet declarations ...

</SchemaName>

```

System Query Options

Query Options (SQOs) are parameters added to an OData URI to control aspects of the data. Most parameters are only valid with GET. SQOs are keywords that always appear after the `?` that terminated the resource path portion of the OData URI from the parameter portion. Each SQO has an argument which is separated from the keyword by '='. Multiple SQOs can be specified in a URI by separating them with `&`. Each SQO can only appear once for a given resource path.

\$format

\$format controls the representation of the data in the response body. In general, DVS returns response bodies in JSON format with the exception that `$metadata` is always returned as XML. URLs containing the `$value`, and `$count` keywords always return data as text/plain. DVS can use a valid default for each request. Setting the data format is never required. Format can also be specified in the request header. OData v4, and OData v3 support different options for JSON responses.

\$format=json is the default for all calls that would return a JSON response.

\$format=verbosejson can be used with OData version 3.0 services only and causes more meta data to be included in the response body.

\$format=text/plain must be used with `$count`, and `$value`.

\$format=xml must be used for `$metadata` requests.

With OData version 4.0, you can explicitly control the amount of meta data returned.

\$format=application/json;odata.metadata=minimal is the default metadata setting, and causes the "@odata.context" to be included in the response body.

\$format=application/json;odata.metadata=none suppresses all metadata in the response body.

\$format=application/json;odata.metadata=full causes all available metadata to be returned. For DVS, this includes `@odata.context` and `@data.type`, `@data.id`, and `@data.editlink` for each record, and `NavigationPropertyName@odata.associationLink`, and `NavigationPropertyName@odata.navigationLink` for each Navigation Property in each record.

With either OData version, you can specify the format in the request header.

- **Accept: application/json** - for the default less verbose JSONoutput
- **Accept: application/json;odata=verbose** - for verbose JSON output (Odata v3)
- **Accept: application/json;odata.metadata=minimal** - same as parameter equivalent.
- **Accept: application/json;odata.metadata=none** - same as parameter equivalent.
- **Accept: application/json;odata.metadata=full** - same as parameter equivalent.
- **Accept: application/xml** – for \$metadata requests
- **Accept: text/plain** – for \$count and \$value requests

A \$format specification in the URL overrides the Accept settings in the request header.

\$filter

\$filter is used to restrict the data returned to only those records which meet certain criteria. It is only used with GET. The argument to **\$filter** is a logical expression that is comprised of references to properties in the entity pointed to by the current resource path context, or that can be reached through navigation properties from the current resource path context, filter operators, filter functions, and/or literals. **\$filter** SQOs only affect the resource for the resource path context in which they are declared. Requests that reference multiple entities can have separate **\$filter** SQOs applied to each resource (see **\$expand**).

See the sections for filter operators, and filter functions for the list of supported operators and functions. DVS does not support **\$count** in **\$filter** arguments.

Example 1 - Return only those books that fit on a 10 inch high shelf.

.../Books?\$filter=Size/Height lt 10

Example 2 - Return only those books which have a cost less than 20.00 dollars and written by Dickens.

.../Books?\$filter=ListPrice lt 20.00 and Author/Name eq 'Charles Dickens'

Example 3 - Return only those books which cost less than 20.00 dollars and were written by Dickens, OR has a name that contains the string **Gulden**.

.../Books?\$filter=(ListPrice lt 20.00 and Author/Name eq 'Charles Dickens') or contains(Title,'London')

\$select

\$select controls which properties are returned in the response body. **\$select** is only valid for GET and can appear as a subsidiary parameter for **\$expand**.

Argument to **\$select** is either a comma-separated list of property names, or the asterisk (*) character. **\$select** is a shortcut equivalent to a list of all non-navigation properties. The property names must be valid for the entity type corresponding to the resource path against which **\$select** is applied, and DVS does not support selecting only some properties from within a complex property that is part of that Entity, so for DVS, only property names that are directly in the object can be arguments. ... */myEntitySet?\$select=APropertyName,AComplexPropertyName* is valid, assuming the two property names that are listed are part of the entity type for entity Set *myEntitySet*, ... */myEntitySet?\$select=APropertyName,AComplexPropertyName/AComplexPropertyProperty* is not supported by DVS.

Example 1 - Return the title and synopsis for each book

.../Books?\$select=Title,Synopsis

Example 2 - Return the name title and synopsis for each book which weighs over two pounds. (Demonstrates multiple SQOs in a request)

.../Books?\$select=Title,Synopsis&\$filter=Weight gt 2.0

Example 3 - Return all the property values for a specific Author.

Note: Returning all properties is the default behavior for an OData service. Explicitly specifying **\$select=*** is typically not required.

.../Authors('100')?\$select=*

Note: For OData version 3.0 only, you can include properties reached through navigation properties in the argument list for **\$select**. This is because the limited **\$expand** syntax supported by OData version 3.0 had no provision for associating SROs with specific branches of an expand.

\$orderby

\$orderby allows control of the order in which records are returned. Any simple property in the set of entities and by the resource path be used in the argument list and simple properties reachable using navigation paths from the current context. These navigation paths must all resolve to a single instance. One to many or many to many navigation paths are not permitted. Keyword **desc** can be used to override the default sort order of low to high. Keyword **asc** is also available if you want to state the sort order.

Example 1 - Return the title, synopsis, and list price for each book that is ordered from the lowest list price to the highest list price.

Note: Null values sort low.

.../Books?\$select=Title,Synopsis,&\$orderby=ListPrice

Example 2 - Return the title, synopsis and list price for each book ordered from the highest list price to the lowest list price, and sort ties by title in ascending order. The **asc** keyword is optional.

.../Books?\$select=Title,Synopsis,ListPrice&\$orderby=ListPrice desc,Title asc

Example 3 - Return the title, synopsis and list price for each book ordered by the actual price asked.

.../Books?\$select=Title,Synopsis,ListPrice&\$orderby=Inventory/Price&\$expand=Inventory (\$select=Price,InStock)

Note: Currently, DVS does not enforce the restriction prohibiting one to many or many to many navigation paths to properties used in an \$orderby and will return reasonable results. This behavior is not dependable as the presence of one to many or many to many navigation paths means that the path does not necessarily resolve to one single value. The results can be ambiguous or undefined. Defect [218056](#) is opened to resolve this. The correct behavior is to return a 400 Bad Request status.

\$skip

\$skip takes as an argument an integer which is the number of otherwise qualifying records to pass over and omit from the response body. **\$filter** is applied to the result set before **\$skip** when determining the result set in the response. When counting records for **\$skip**, the record at the same level as **\$skip** and any records nested within that record due **\$expand** are counted as one record. Together with **\$top** this is used by an application to page through data where it would be inconvenient to retrieve all of it at one time.

Example - Fetch the name and description for all authors, but skip the first three authors.

.../Authors?\$select=Name,Bio&\$skip=3

\$top

\$top allows the user to restrict the number of records returned by specifying the maximum number of records to return as an argument. The effect of **\$filter** and then **\$skip** are applied before **\$top**. Like **\$skip**, when counting records for **\$top**, the record at the same level as **\$top** and any records nested within that record due **\$expand** are counted as one record. Together with **\$skip** this is used by an application to page through data where it would be inconvenient to retrieve all of it at one time.

Example 1 - Fetch the title and list price for all books whose list price is less than 50.00 but skip the first three books, and only return a maximum of five books.

.../Books?\$select=Title,ListPrice&\$filter=ListPrice lt 50.00&\$skip=3&\$top=5

Example 2 - Same as previous example, but include the comments on the books. The same number of books is returned with selected properties from the comment records on these books.

```
.../Books?$select=Title,ListPrice&$filter=ListPrice lt 50.00&$skip=3&$top=5&$expand=Comments
($select=CriticName,Comment)
```

\$expand

\$expand allows a user to request related data from multiple Entity Sets. **\$expand** accepts a comma-separated list of resource paths composed of navigation properties originating from context in which the **\$expand** is specified. Each top-level record is returned with child record data nested within it. If an expand resource path traverses several entities, all entities traversed are expanded, and data for each entity at a deeper level is nested within the parent record in the level above. DVS will **ignore** a **\$ref** keyword that is the last segment of an **\$expand** resource path argument.

OData version 4.0 compatible services allow each of the resource paths arguments to expand to specify System Query Options to apply only to that resource path. These options appear within parenthesis directly after the resource path, with multiple System Query Options separated by a semi-colon (";"). The System Query Options supported DVS within **\$expand** are **\$filter**, **\$select**, **\$top**, **\$skip**, and **\$orderby**. Advanced feature **\$levels** is not supported.

Some examples using the vPub schema (omitting the base path to the OData service)

Example 1 - Return all the author records with book records nested under each author. **A uthors** is an entity set, and **books** is a navigation property in entity type author.

```
.../Authors?$expand=Books
```

Example 2 - Return all the book records for author('101') with the comment records for each book nested under each author. **A uthors** is an entity set. **B ooks** are a navigation property in entity type author. **C omments** is a navigation property in entity type book.

```
.../Authors('101')/Books?$expand=Comments
```

Example 3 - Both these URLs returns all author records with book records nested under each author and comments on each book nested each book record. The second form is only supported under OData version 4.0.

```
.../Authors?$expand=Books/Comments
```

```
.../Authors?$expand=Books($expand=Comments)
```

Example 4 - Return all author records with book records nested under each author in title order, but only for those book records whose **ListPrice** Property value is over 10.00, and only the **Title** and **Synopsis** . Comments on each book are nested under the book record, but only return the **CriticName** , and **Comment** Property values.

```
.../Authors?$expand=Books($filter=ListPrice gt 10.0;$orderby=Title;$select=Title,Synopsis;$expand=Comments($select=CriticName,Comment))
```

Example 5 - As Example 4, except that **Authors** without books whose list price is greater than 10 dollars are excluded. Remember comparisons to null evaluate to false and System Query Options are only applied to the resources at the same level they appear.

.../Authors?\$expand=Books(\$filter=ListPrice gt 10.0;\$orderby=Title;\$select=Title,
Synopsis;\$expand=Comments(\$select=CriticName,Comment))&\$filter=Books/ListPrice gt 10

\$count (System Query Option)

The **\$count** keyword is used in two ways within OData. It can be used as part of a resource path, or starting with OData version **4.0** as a System Query Option. As a System Query Option it allows the user to control whether **@odata.count** is part of the response body. If the argument to \$count is **true**, then **@odata.count** is returned and has a value of the integer count of the qualifying records. The count is affected by **\$filter** but ignores the effect of **\$skip** and **\$top**. This number does not reflect the actual number of records in the response body.

If \$count is absent or has an argument of **false**, then **@odata.count** is omitted from the response.

Example 1 - Fetch the title and list price for all books whose list price is less than 50.00 but skip the first three books, and only return a maximum of five books. Include the count of all qualifying records.

.../Books?\$select=Title,ListPrice&\$filter=ListPrice lt 50.00&\$skip=3&\$top=5&**\$count=true**

While a maximum of five records are returned due to **\$top**, the **@odata.count** value reflects the full number of records that match the **\$filter** restriction, and that would have been returned if both **\$top**, and **\$skip** were absent.

Note: \$count is not a valid System Query Option in OData version 3.0. Instead **\$inlinecount** is used with an argument of either **allpages** or **none**.

Filter Operators

DVS supports all \$filter Operators that are defined in OData version 3.0. DVS does not support the has operator that is introduced with OData version 4.0.

Sub-expressions can be grouped using parenthesis to override precedence. Operators are processed in precedence level order (level 1 = highest priority). Operators within an expression with equal precedence are evaluated left to right.

Operator	Description	Precedence Level
mul	Arithmetic multiplication . Arguments must be of numeric types.	1
div	Arithmetic division . Arguments must be of numeric types. Right Argument of zero will fail query and return a status 500	1
mod	Arithmetic modulo . Arguments must be of numeric types. Right Argument of zero will fail query and return a status 500	1

Operator	Description	Precedence Level
add	Arithmetic addition . Arguments must be of numeric types	2
sub	Arithmetic subtraction . Arguments must be of numeric types.	2
eq	Compare arguments and evaluate to true if arguments are equal	3
ne	Compare arguments and evaluate to true if arguments are not equal	3
gt	Compare arguments and evaluate to true if Left argument Greater than Right Argument	3
ge	Compare arguments and evaluate to true if Left argument Greater than or equal to Right Argument	3
lt	Compare arguments and evaluate to true if Left argument Less than Right Argument	3
le	Compare arguments and evaluate to true if Left argument Less than or Equal to Right Argument	3
not	Logical not . Evaluates to true if right argument is false, evaluates to false if right argument is true	4
and	Logical and . Evaluates to true if both arguments are true	5
or	Logical or . Evaluates to true if either argument is true	6

Note: Comparisons must be made between expressions of compatible types. For example, an expression that resolves to any numeric types may be compared against other expression that resolves to a numeric type, but may not be compared to an expression whose base type is a string.

Any property which is not a key, and was not defined as being required may have a null value to indicate its value is undefined. Any comparison where one or both of the arguments are **null** will evaluate to **false**, except comparisons using **eq** or **ne** against the keyword literal **null**. Both `$filter=APropertyWithNullValue le SomeLiteral` and `$filter=APropertyWithNullValue ge SomeLiteral` will evaluate to **false** when `APropertyWithNullValue` contains a value of **null** even if `SomeLiteral` is the literal **null**.

Built-in Filter Functions

DVS supports nearly the full set of filter functions. The exception being those functions such as **isof**, which allow tests for specific entity types.

String functions

contains(string,substring) - Returns **true** if second string argument is the same as, or is found within, the first string argument, or if `substring` is **null**. Will return **false** if `string` is **null**. Introduced as of OData version 4.0.

.../Books?\$filter=**contains**(Title,'London')

substringof(*substring*,*string*) - Returns true if first string argument is the same or is found within the second string argument, or if *substring* is **null**. Will return false if *string* is **null**. This is the OData version 3.0 equivalent of **contains**. This is not supported by Odata version 4.0, and also notes the reversal of the parameters as compared to other string functions.

.../Books?\$filter=**substringof**('London',Title)

startswith(*string*,*substring*) - Returns true if second string argument matches the first string argument starting from the beginning of the first string argument, or if *substring* is **null**. Will return false if *string* is **null**.

.../Books?\$filter= **startswith** (Title,'T')

endswith(*string*,*substring*) - Returns true if second string argument matches the end of the first string argument, or if *substring* is **null**. Will return false if *string* is **null**.

.../Books?\$filter= **endswith**(Title,'Cities')

length(*string*) - Returns an integer with value equal to the number of characters in the string. If the value of *string* is **null**, **null** is returned.

.../Comments?\$filter= **length** (Comment) gt 40

indexOf(*string*,*substring*) - Returns an integer with value equal to the position of *substring* within *string*, with 1 being the first position. If *substring* is not in *string*, 0 is returned. If either argument has a value of **null**, **null** is returned.

.../Comments?\$filter=**indexOf** (Comment,'read') ne 0

substring(*string*,*position*) - Returns a String whose value is the value of *string* starting at *position*, where a position of 1 is the first character of the string argument. If either argument has a value of **null**, **null** is returned.

.../Books?\$filter=**substring**(Title,7) eq 'House'

substring(*string*,*position*,*length*) - Returns a String whose value is the value of *string* starting at *position* up to a maximum of *length* characters, where the first character of the string has position 1. If any argument has a value of **null**, **null** is returned.

.../Authors?\$filter=**substring**(Bio,14,5) eq 'Irish'

tolower(*string*) - Returns a String whose value is the original value with all uppercase characters converted to lowercase. If the value of *string* is **null**, **null** is returned.

.../Comments?\$filter=indexof(**tolower**(Comment),'dark') ne 0

toupper(*string*) - Returns a String whose value is the original value with all lowercase characters converted to uppercase. If the value of *string* is **null**, **null** is returned.

.../Comments?\$filter=indexof(**toupper**(Comment),'DARK') ne 0

trim(string) - Returns a String whose value is the original value with all leading and trailing white space (space, tab, etc. characters) removed. If the value of *string* is **null**, **null** is returned.

```
.../Authors?$filter= trim ( substring(Name,1,8)) eq trim ( ' Charles ')
```

concat(string1,string2) - Returns a String whose value is the *string2* appended to *string1*. If either argument has a null value the value of the non-null argument is returned, and if both arguments have null values, **null** is returned.

```
.../Comments?$filter=contains(Comment,concat('child','ren'))
```

replace(string,substring1,substring2) - OData version **3.0** only. Returns a string whose value is the value of *string*, except that each occurrence of *substring1* is replaced by *substring2*. If the value of *string* is **null**, **null** is returned. If *substring1* is null, the value of *string* is returned unchanged, and if *substring2* is null, all occurrences of *substring1* are removed from the result string. In the example, all space characters are being removed from name before the comparison to the literal.

```
.../SomeEntitySet?$filter=contains(replace(SomeProperty,' ',null), 'SomeStringLiteral')
```

Date and Time Functions

year(DateTimeOffset) - Return the year portion of a DateTimeOffset as an integer. If *DateTimeOffset* is null, returned value is **null**.

```
.../Comments?$filter=year(Posted) eq 2015
```

month(DateTimeOffset) - Return the month portion of a DateTimeOffset as an integer (January equals 1, December equals 12). If *DateTimeOffset* is null, returned value is **null**.

```
.../Comments?$filter=month(Posted) eq 5
```

day(DateTimeOffset) - Return the day of the month portion of a DateTimeOffset as an integer. If *DateTimeOffset* is null, returned value is **null**.

```
.../Comments?$filter=day(Posted) eq 2
```

hour(DateTimeOffset) - Return the hour portion of a DateTimeOffset as an integer. If *DateTimeOffset* is null, returned value is **null**.

```
.../Comments?$filter=hour(Posted) lt 11
```

minute(DateTimeOffset) - Return the minute portion of a DateTimeOffset as an integer. If *DateTimeOffset* is null, returned value is **null**.

```
.../Comments?$filter=minute(Posted) ge 30
```

second(DateTimeOffset) - Return the second portion of a DateTimeOffset as an integer. If *DateTimeOffset* is null, returned value is **null**.

```
.../Comments?$filter=second(CommentTimestamp) lt 30
```

fractionalseconds(*DateTimeOffset*) - Return the fractional seconds portion of a DateTimeOffset as a float. If *DateTimeOffset* is null, returned value is **null**.

.../Comments?\$filter=fractionalseconds(CommentTimestamp) eq 0.5

now() - Return the current DateTimeOffset. Timezone is determined by the timezone in effect on the server hosting CA Service Virtualization Virtual Service Environment hosting the OData VS.

.../Comments?\$filter=Posted gt **now()**

mindatETIME() - Return the earliest date that is supported by the OData VS. DVS returns January 1st 1753, which is the date many nations that are switched to the Gregorian Calendar.

.../Comments?\$filter=year(Posted) eq 2015 and year(**mindatETIME()**) eq **1753** and month(**mindatETIME()**) eq **1** and day(**mindatETIME()**) eq **1**

maxdatETIME() - Return the most distant date that is supported by the OData VS. DVS returns Dec 31st, 9999. However this is not actually true, as DVS with the H2 database will support dates millions of years in the future.

.../Comments?\$filter=year(Posted) eq 2015 and year(**maxdatETIME()**) eq **9999** and month(**maxdatETIME()**) eq **12** and day(**maxdatETIME()**) eq **31**

Mathematical Functions

round(*decimalValue*) - Returns the input numeric value that is rounded to the nearest integer value. If the decimal portion of the input is less than 0.5, the next lowest integer value is returned. Otherwise, the value is increased to the next integer. If the input value resolves to null, **null** is returned. Both following examples returns the same result set.

.../Books?\$filter=**round**(Weight) eq 1.0

.../Books?\$filter=**round**(Weight) eq 1

floor(*decimalValue*) - Returns the input numeric value that is reduced to the integer value. That is if the decimal portion of the input is set to zero. If the input value resolves to null, **null** is returned.

.../Books?\$filter=**floor**(Weight) eq 0

ceiling(*decimalValue*) - Returns the input numeric value that is increased to the next highest integer value if the decimal portion of the input is not zero. If the input value resolves to null, **null** is returned.

.../Books?\$filter=**ceiling**(Weight) eq 2

Request and Response Headers

In addition to the headers described under **\$format**, DVS supports or generates the following headers.

Odata-Version

OData-Version can be provided as a request header. Value should resolve to a number. DVS checks this numeric value against the OData version the target VS is configured to emulate, and if versions are not equal the request is rejected with status 400, and a message indicating what OData version is supported. At present a DVS configured for OData version 4.0 does not support OData version 3.0 requests, and the other way around, and no other version of OData is supported.

Odata-Version is always returned as a response header whose value is either 3.0, or 4.0 depending on the OData version the target VS is configured to emulate.

OData-MaxVersion

OData-MaxVersion may be provided as a request header. Its value is ignored if OData-Version is also present. If OData-MaxVersion has a value that resolves to a number equal to or greater than the OData version the target VS is configured to emulate, then the request passes, but is otherwise rejected with status 400. A message indicating the OData version the target VS is configured to emulate.

Prefer

By default POST returns a response body containing the newly created entity in JSON format, while PUT and by default, PATCH responds with a **204 No Content**. The following prefer headers allow explicit control of this behavior.

return=minimal

For successful calls to methods POST, PUT, (and if supported PATCH) with a VS configured for OData version **4.0**, **return=minimal** suppresses the response body. For a VS configured for OData version, **3.0**, **return-no-content** has the same effect.

return=representational

For successful calls to methods POST, PUT, (and if supported PATCH) with a VS configured for OData version **4.0**, **return=representational** forces a return of a response body containing the newly created or modified values in JSON format. For a VS configured for OData version **3.0**, **return-content** has the same effect.

Location

A response header, only returned when performing a POST to a VS configured for OData version **4.0**. Value of header is the edit URL of the newly created entity.

odata-EntityId

A response header, only returned when performing a POST to a VS configured for OData version **4.0** where a Prefer header with a value of return=minimal was in effect. Value of header is the entity ID (key or keys), of the newly created entity.

Extensions Outside of OData Standard

The following features support DVS, but are not part of the OData standard.

Automatic Generation of Property Values

You define the entities in your DVS VS to have one or more of the property values automatically generated.

Property values can be configured to be generated once when the entity instance is first created using POST, or for every update. The first case is useful for automatically generating key values, or a creation timestamp. The second case can be used to generate the value for a property whose value is the last modification timestamp.

See Property in [Designing Your Virtual Service](#) about configuring a property to use automatic value generation.

REST API to Manage Data

Any DVS VS also supports a REST API that allows saving the current set of entity data out to a file or loading of the entity data from a file. This feature is useful when there is a need to reset data to a known state for testing.

See [Managing Your Virtual Service Data](#) for details.

Custom Properties Defined at Run-Time

DVS supports adding custom properties (CPs) at runtime for entity sets whose entity type has the feature enabled. If an entity type is enabled, there is an implicit entity type and entity set created with the names `x Property`, and `x Properties`. Where `x` is the name of the entity type having the custom properties feature enabled. Users should avoid defining names that would conflict with these generated names.

For example, entity type `order` in the Bookshop example is defined with custom properties enabled. Therefore

```
<EntityType Name="OrderProperty">
  <Key>
    <PropertyRef Name="name"/>
  </Key>
```

```

<Property MaxLength="256" Name="name" Nullable="false" Type="Edm.String"/>
<Property MaxLength="256" Name="dataType" Type="Edm.String"/>
<Property MaxLength="4096" Name="description" Type="Edm.String"/>
<Property MaxLength="256" Name="defaultValue" Type="Edm.String"/>
<Property MaxLength="4096" Name="possibleValues" Type="Edm.String"/>
<Property MaxLength="256" Name="producerMappingProperty" Type="Edm.String"/>
<Property MaxLength="256" Name="entityName" Type="Edm.String"/>
</EntityType>

```

and

```
<EntitySet EntityType="Bookshop.OrderProperty" Name="OrderProperties"/>
```

appear in the \$metadata for the Bookshop VS.

Defining a Custom Property

To create or modify custom properties, standard OData operations are performed against this associated entity set. For example, to create a new custom property for order:

```
POST ../OrderProperties
```

```

{

  "name": "ANewCustomProperty",

  "dataType": "edm.String",

  "description": "free form text description of property"

}

```

With the following response:

```

{

  "@odata.context": "http://\\./MyVSEServer:8844/odata/v4/Bookshop\\
/$metadata#OrderProperties",

  "value": [

    {

      "name": "ANewCustomProperty",

      "dataType": "Edm.String",

      "description": "free form text description of property",

      "defaultValue": null,

      "possibleValues": null,

```

```

"producerMappingProperty": null,

"entityName": null

}

]

}

```

The following properties that can be specified:

- **name:** The identified of the CP and must conform to the syntax of a property name. Name must be unique within all the properties of the entity type containing the CP.
- **dataType:** The datatype to be used for the CP. Only simple types such as Edm.String, and Edm.Int32 are supported.
- **description:** Intended to contain free form text which the designer can use to describe the CP.
- **defaultValue:** The value to give the CP if CP is not specified in a POST or PUT action.
- **possibleValues:** A comma delimited list of values which is acceptable as valid for the CP.
- **producerMappingProperty:** For future use, do not specify.
- **entityName:** For future use, do not specify.

You can modify characteristics of a previously defined custom property by using PUT. For example, the following changes the description and add a default value to the CP defined in the previous example:

```

PUT ../OrderProperties('ANewCustomProperty')
{

"dataType": "edm.String",

"description": "updated free form text description of property",
"defaultValue": "SomeDefaultStringValue"

}

```

Some limitations are imposed by the underlying database. For example, changing the datatype can fail if values stored for the CP are incompatible with the new type.

Working with Custom Properties

Depending on how the `EntityType` is defined, the new custom property can either be accessible as a simple property in the entity itself or as one of a set of name value pairs in a collection (NVP CP). See `WorkTeam` in the Bookshop example for an entity with CPs used this way. The name of the collection is specified as part of the configuration, and the collection consists of instances of an implicit complex type with properties **`propertyName`**, and **`propertyValue`**. See the following example and order in the Bookshop example for an entity with CPs used this way.

```
POST ../Orders
```

```
{
  "CustomerId": "CUS00137",
```

Other regular properties:

```
"Special": [
  {
    "propertyName": "ANewCustomProperty",
    "propertyValue": "MyValue"
  }
]
}
```

See `EntityType` in [Designing Your Virtual Service](#) for defining an entity type with this feature enabled.

DVS Limitations

DVS does not support any feature that is only required for intermediate or advanced OData v4 conformance, unless indicated. Features that are part of the [OData Minimal Conformance Level](#) for an updatable service are typically implemented with exceptions.

Omissions to OData Minimal Conformance Level

PATCH method for differential update. This method should be available for a compliant updatable OData service. CA Service Virtualization does not allow HTTP methods apart from POST/GET/PUT/DELETE. Until HTTP methods are supported, PATCH cannot be implemented.

Other omissions

These omissions are not required for minimal conformance. DVS returns a **501 Not Supported** message for unsupported features.

- DVS does not support stream, media, geometric, and geographic types.
- DVS does not support **\$filter** functions related to class. For example, the **isof** functions are not supported. Due to limitations in the Olingo 4.0.0-Beta-02, parser **\$ref** cannot be used with **\$filter** or **\$orderby**.
 - `bool IsOf(type value)`, for example, `isof('NorthwindModel.Order')`
 - `bool IsOf(expression value, type targetType)`, for example, `isof(ShipCountry,'Edm.String')`
- No support for **\$count** in **\$filter** arguments.
- **\$filter** does not support the **has** operator.
- **\$expand** does not support **\$format**, **\$count**, or **\$levels**.
- **\$select** does not support selecting only some properties from within a complex property that is part of that entity against which **\$select** is applied.
- **\$id** URL keyword is not supported.
- Deep insert where nested records have generated keys does not work.
- Cannot POST or PUT to properties within complex properties.
- DELETE of a **custom** property is not supported.

Known Issues

See README.txt in the distribution to see a list of known defects at the time of posting.

Building Dynamic Virtual Service

DVS is not distributed as binary files that can be directly used for installation. The download contains the necessary source files and build artifacts to create the install binaries. Third-party components are loaded from public repositories as needed. The build process is simple. Only commonly available, free open source tools are required to perform the build.

The DVS sources builds two primary artifacts for the OData Dynamic Virtualization Services project. The artifacts are the DVS servlet and the OData Management Extension Assistant (an Eclipse plug-in). The DVS sources packages for deployment the OData DVS extension. The extension is deployed into DevTest Virtual Service Environments to enable the OData emulation.

Prerequisites

To build the DVS, the DVS must be downloaded and the following tools must be installed and configured:

- Java SE Development Kit (JDK) to compile the source code
- Apache Ant to orchestrate the build
- A small set of Ant extensions (installed in lib directory of Ant)
- Maven to build the Eclipse Plug-in

Note: Line breaks do not display correctly in certain editors when DVS is downloaded as a zip and expanded into a folder. This limitation is caused by UNIX style line terminators in the text files. The build is not effected but can be difficult examining or modifying the files under Windows. Avoid this issue by cloning the dvs_ce repository to your desktop instead of downloading the zip.

Java SE Development Kit (JDK)

A Java SE Development Kit must be installed on the system where DVS is built.

Note: If you want to run DVS components in Java 7 and Java 8 runtime environments, JDK 7 must be used. If you are currently running your Java applications (Eclipse and Tomcat7) on Java 8, use JDK 8 to build.

1. If not already present, download the JDK from the [Oracle download site](#). Ensure that you select a Java Platform (JDK) installer most suitable for your host platform (x86 or x64).

2. Follow Oracles instructions for installing the JDK.
3. Set the environment variable **JAVA_HOME** to the folder containing the newly installed Java SDK (for example, C:\Program Files\Java\jdk1.7.0_79).
4. Add **%JAVA_HOME%\bin** to your user PATH (for example set PATH=%JAVA_HOME%\bin;%PATH%).
5. Open a new console (cmd.exe) window and verify that JAVA_HOME is set and the JDK is in your search path.
6. Run **java -version** and examine the output to make you are seeing the expected version of Java.

Apache Ant

Any current version of Apache Ant is expected to work. Version 1.9.4 was used in testing this procedure. The Apache documentation, [Installing Apache Ant](#) can be used as a reference when installing Ant.

Follow these steps:

1. Download Ant from the [Apache download site](#), selecting the **.zip** of the Ant distribution.

Note: You can elect to use the tar distribution. Long file names in the tar archive can require gnu tar to perform the extraction.

2. Unzip the Ant distribution into your tools folder (that is, C:/tools). A folder reflecting the version of Ant chosen (for example, **apache-ant-1.9.4** is created under your tools folder).
3. Set environment variable ANT_HOME to the Ant directory path. (for example, set ANT_HOME=c:\tools\apache-ant-1.9.4).
4. Add **%ANT_HOME%\bin** to your user PATH (for example, set PATH=%PATH%;%ANT_HOME%\bin).
5. Open a new console (cmd.exe) window to verify that ANT_HOME is set and running **ant -version** returns the expected version of Ant.

Install Required Ant Extensions

The download for DVS includes the Ant build target to deploy the required extensions to Ant. These extensions include Ivy (dependency management), ant-contrib (if, else, and so on) & jacoco (code coverage reporting).

1. Download DVS into a folder.

2. Make the folder where the download has been extracted the current folder.
For example, if the DVS download was extracted into the C:\proj\dvs_ce cd /D C:\proj\dvs_ce
3. Run Ant specifying setup.xml as the build file.

ant -f setup.xml

The script takes a few seconds to run.
4. Verify that the output is similar to the following output:

```

_init:
[get] Getting: http://search.maven.org/remotecontent?filepath=ant-contrib/ant-contrib/1.0b3/ant-contrib-1.0b3.jar
[get] To: C:\tools\apache-ant-1.9.4\lib\ant-contrib-1.0b3.jar
[get] http://search.maven.org/remotecontent?filepath=ant-contrib/ant-contrib/1.0b3/ant-contrib-1.0b3.jar moved to https://repo1.maven.org/maven2/ant-contrib/ant-contrib/1.0b3/ant-contrib-1.0b3.jar
[get] Getting: http://search.maven.org/remotecontent?filepath=org/apache/ivy/ivy/2.4.0/ivy-2.4.0.jar
[get] To: C:\tools\apache-ant-1.9.4\lib\ivy-2.4.0.jar
[get] http://search.maven.org/remotecontent?filepath=org/apache/ivy/ivy/2.4.0/ivy-2.4.0.jar moved to https://repo1.maven.org/maven2/org/apache/ivy/ivy/2.4.0/ivy-2.4.0.jar

install-antlibs:
[ivy:resolve] :: Apache Ivy 2.4.0 - 20141213170938 :: http://ant.apache.org/ivy/
::
[ivy:resolve] :: loading settings :: file = C:\proj\dvs_ce\build-essentials\ivysettings.xml
[ivy:resolve] :: resolving dependencies :: com.ca.dvs#dvs.build;working@artal03-m
[ivy:resolve] confs: [build]
[ivy:resolve] found ant-contrib#ant-contrib;1.0b3 in mvnrepository
[ivy:resolve] found org.jacoco#org.jacoco.ant;0.7.4.201502262128 in mvnrepository
[ivy:resolve] found org.jacoco#org.jacoco.agent;0.7.4.201502262128 in mvnrepository
[ivy:resolve] found org.jacoco#org.jacoco.report;0.7.4.201502262128 in mvnrepository
[ivy:resolve] found org.jacoco#org.jacoco.core;0.7.4.201502262128 in mvnrepository
[ivy:resolve] :: resolution report :: resolve 218ms :: artifacts dl 16ms
-----
| | modules || artifacts |
| conf | number| search|dwnlded|evicted|| number|dwnlded|
-----
| build | 5 | 0 | 0 | 0 || 5 | 0 |
-----
[ivy:retrieve] :: retrieving :: com.ca.dvs#dvs.build
[ivy:retrieve] confs: [build]
[ivy:retrieve] 5 artifacts copied, 0 already retrieved (778kB/31ms)

all:
BUILD SUCCESSFUL
Total time: 6 seconds

```

Install Apache Maven

1. Download Maven from Apache download [site](#).
Any current version of Apache Maven works. Version 3.3.1 was used in testing this procedure.

1. Open a command window.
2. Set the current directory to your project root folder. For example, if the DVS download was extracted into C:\proj\dvs_c cd /D C:\proj\dvs_c
3. Execute Ant.

When the build completes, the content in your console window displays similar to the following screen:

Screen capture of command prompt screen

The following output build artifacts are located in the build directory.

- 36/61

Next Steps

See [Installing DVS](#) for instructions about how to deploy the newly created build artifacts.

Installing Dynamic Virtual Service

DVS software contains two separate components and has a dependency on a third component. The third component is an extension that is named OData Dynamic Virtualization Extension.

The two components that are part of this project are:

- ODME Assistant - an Eclipse plug-in
The ODME Assistant is used to create MAR files containing a DevTest project.
The project implements an OData DVS that is based on a definition you provide. ODME Assistant is not required to be deployed on the same system as the CA Virtual Service Environment (VSE) that the OData Dynamic Virtualization is deployed to DevTest is not required to be installed.
- A war file
War file which can be deployed into a Web Server instance such as Tomcat. The war supports a REST API which allows direct creation and deployment of an OData Virtual Service from a RAML. The war can reside on a separate system but must be configured with connection information to a VSE. Using this component is optional.

The third dependent component contains jars that are provided by CA Technologies with other vendors. The jars must be deployed into every DevTest VSE that the OData Dynamic Virtualization is using.

Before installing these components for the first time, they must be built. See Building Dynamic Virtual Services for the required steps to build the components.

Prerequisites for Installing the DVS Eclipse Plug-in

The developer portion of DVS is implemented as an Eclipse plug-in. The install process was tested with Kepler Service Release 2. Any compatible version of the Eclipse IDE can be used. See <https://eclipse.org> for access to Eclipse downloads and instructions for installing Eclipse.

Installing Eclipse or the DVS Eclipse plug-in is not required for the following situations:

- Only defining the OData virtual services using the RAML format
- Only deploying the virtual services through the REST API supported by the DVS servlet

Installation

1. From the Eclipse main menu bar, select **Help, Install New Software...**
Install dialog displays.
2. Click **Add** to add a new site and click **Archive** from the **Add Repository** dialog.

3. Browse to, or enter the location where the site assembly for the ODME Assistant was built or copied to.
4. Verify that the name **DVS DevTest Eclipse PlugIn** populates in the **Name** column and click **OK**.
5. Check the **DVS DevTest Eclipse PlugIn** checkbox and click **Next**.
6. If you installed the plug-in previously, select **Update my installation to be compatible with the items being installed** and click **Next**.
7. Verify the **Install Details** and click **Next**.
8. Accept the license and click **Finish**.
Because a trusted certificate was not used to sign the code, a warning displays.
9. Click **OK** to proceed.
10. (Optional) Click **Run in Background** to run the install in the background.
11. When the installation is complete, click **Yes** to restart Eclipse and finish the installation.

Configuring the Eclipse Plug-in

Typically there is no reason to configure the plug-in. One exception is if you have an OData virtual service definition with many entities and many interconnecting navigation properties. The number of transactions added to the LISA Virtual Service Image (VSI) file can increase to thousands even with the default **Maximum depth of the resource path** of 4. These transaction are only added to the VSI for documentation purposes. The OData virtual service extension does not actually use these specific paths. Instead, the extension resolves the resource paths of any depth solely using the meta model that it reads from the EDMtoDB.xml (AEDM file).

If you do not need to review the transactions, you can perform the following options:

- Reduce the **Maximum depth of the resource path**.
- Turn off non-generic transaction generation.

Follow these steps:

1. In Eclipse, select the **Windows, Preferences**, and click **DVS Assistant** in the left panel of the dialog.
2. To have DVS only generate the minimum number of generic transactions, clear the **Enable to Populate Transactions** checkbox.
3. To control the number of transactions added to the VSI file, enter the integer value (1-10) for the **Maximum depth of the resource path**.

Installing the DVS Servlet

This section provides information and instructions to install the DVS servlet.

Note: Installing DVS servlet is optional. Installation is only required if you intend to use the DVS REST API.

Prerequisites

The war file containing the DVS servlet was tested with Apache Tomcat 7.0.57. Other versions of Tomcat or other web servers are supported.

A Java 7 or Java 8 run-time environment must be installed on the system that hosts the web server. To obtain the download and instructions from www.oracle.com. If Apache Tomcat 7.0.57 is not installed, go to <https://tomcat.apache.org> for the download and instructions.

Installing the DVS Servlet

1. Stop your Tomcat installation by running the shutdown.bat (Windows), or shutdown.sh (Unix) batch files from Tomcat bin folder.
2. Copy the war file that is created by the build process to the webapps folder of your Tomcat instance.
3. (Recommended) Shorten any long war file names. The name of the war file determines the base portion of the URL resource path that is used to access the DVS REST API. In the examples, instructions and testing the war file is renamed to **dvs.war**.
4. Start your Tomcat instance.
5. The batch file startup.bat (Windows) or startup.sh (Unix) is a recommended. Tomcat expands the war file into a folder with the same name as the war file. If your DevTest VSE is installed on the same system as the Tomcat instance, no further configuration is necessary.

Configuring the DVS Servlet

The following configuration items can be changed:

- **vseServerUrl** - URI for the DevTest VSE that hosts the virtual service instances. Default is <http://localhost:1505/api/Dcm/VSEs/VSE>.

- **vseServicePortRange** Range from which the listen port for deployed VS instances are allocated. Multiple ranges are comma-separated. A range can be either a single integer or min-max. The default is 46001-46999.
- **vseServiceReadyWaitSeconds** - Delay between when service is deployed and servlet **pings** the service to make certain it is still up.
- This delay is the major part of the time it takes for the **PUT /raml/vs** call to execute. The delay and a ping guard against reporting success because the VS was successfully deployed but having the VS die during initialization. This result can happen with hand-crafted AEDM files, but has not yet been seen with AEDMs generated from a RAML. (Generating a VS from a RAML can fail if the RAML is malformed or if the RAML contains incomplete or inconsistent data. The failure is detected before deployment). Delay was made configurable because VS initialization failures have always been observed to occur within 7-9 seconds. You can reduce the value if your VSE consistently responds quickly. Increase the value if your VSE is unusually slow. The default is 15.

To change the configuration:

1. Stop your Tomcat installation.
2. Modify the contents of context.xml using any text/XML editor. The context.xml file is located in the webapps/dvs/META-INF folder of the Tomcat instance. Change the value of the **value** attribute of the **Environment** XML element whose **name** attribute value matches the configuration item of interest.
3. Save file.
4. Restart Tomcat.
5. Confirm that servlet has restarted and the new configuration values are in place. Follow the step in [Verifying DVS Servlet Operation](#) in the next section.

Verifying DVS Servlet Operation

The simplest way to verify the operation is to query the DVS servlet for its configuration. Place the URL **Protocol:hostname:Port/dvs/rest/config** into a web browser. You can also use the same URL with a GET in a REST client. **Protocol** is either http or https depending on how your Tomcat is configured, **HostName** is the system where Tomcat is running. **Port** is the Tomcat listen port. The port number is typically 80 or 8080. The name **dvs** is used to the copy of the war file that is deployed into the webapps folder.

If Tomcat and the servlet is running, a similar output follows:

```
{
  "vseServerUrl": "http://MyHostName:1505/api/Dcm/VSEs/VSE",
  "vseServicePortRange": "46001-46999",
  "vseServiceReadyWaitSeconds": 10
}
```

Installing the OData Dynamic Virtualization Services Extension

1. Obtain the **dvs.lisa-extensions.odata-install.zip** file from the **dvs.lisa-extensions.odata** folder. This folder is located under the build folder of your DVS build on the system that is used to build DVS.
2. If the OData Dynamic Virtualization Extensions were previously deployed, back up the files and folders in your DevTest **hotDeploy** folder listed in step 5.
3. If the OData Dynamic Virtualization Extensions were previously deployed, stop any running DevTest services on the system hosting your VSE before deploying the new jar files. When these instructions were written, there was an open defect in DevTest where the code managing the hotDeploy jars was not releasing dependent jars. If the jars that are to be replaced are still in use, you will not be able to copy over them.
4. Extract the full contents of the zip archive to the **hotDeploy** folder of the DevTest instance that the VSE hosts the OData DVS.
For a Windows system, this is typically **C:\Program Files\CA\Lisa\hotDeploy**.
5. Confirm that the target **hotDeploy** folder now includes the following file and folder:
 - **dvs.lisa-extensions.odata.jar**
 - Folder **dvs.lisa-extensions.odata**
6. Restart any DevTest services that are stopped in step 3.

Installing the Example Bookshop DevTest Project

Part of the DVS download is an example DevTest project named Bookshop. The Bookshop example implements an OData version 4 virtual service which illustrates the features supported by OData and DVS.

To explore and use this project to perform the following steps:

1. Copy the examples\Bookshop folder from your DVS download to a location that can be accessed from your DevTest Workstation.
Even if a download is already on the system hosting your DevTest Workstation, make a copy of the Bookshop folder. You have a local backup to revert any changes that are introduced when using the project.
2. From DevTest Workstation, use the **File, Open ,Project, File System** menu path or click the **Open project** icon.
3. From the **Open Project** dialog, browse to the **Bookshop** folder created in step 1 and click **Open**.
A DevTest project displays.

4. To deploy the virtual service to the default VSE for this Workstation, right-click the bookshop.mari file located under MARInfos and select **Deploy/Redeploy to VSE@Default** Alternately, you can right-click the bookshop.mari file under MARInfos, and select **Build Model Archive....** This allows you to create a Model Archive (MAR file) for this project. This MAR file can then be deployed into any VSE using any of the methods supported by DevTest.
5. Once deployed, verify operation by performing a GET to `https://ServerHostingVSE:8844/odata/v4/Bookshop` using a REST client, or type the URL into a browser. In either case, a response listing the Entities that are supported by the Bookshop virtual service is returned.

Next Steps

To define your own virtual service see [Designing Your Virtual Service](#).

Designing Your Virtual Service

While the mechanics of generating a DVS can be performed from start to the final deployment within minutes, preparing the design specification for your VS should be done carefully. Designers specify what entities the virtual service supports, how the data looks, and how the entities relate to each other. DVS supports two formats for specifying this information. The first is the Augmented Entity Definition Model (AEDM) file format. The AEDM file format is similar to a standard EDM and has extensions specific to DVS. The second is a REST API Modeling Language (RAML) format, which is not tied to any product. The AEDM format provides the most power. All supported DVS features can be expressed with an AEDM definition. While RAML has the advantage that it is not specific to DVS, RAML can be used with the DVS REST API. This advantage also allows the specification of the data to populate the virtual service.

Defining a VS in AEDM Format

The AEDM file format is the internal configuration format used to describe the metadata of all the entities exposed by the VS. Even DVS virtual services defined through a RAML file, contain an EDMtoDB.xml file in their data folder. The EDMtoDB.xml file was generated from the information in the RAML. Because the configuration information required is similar to the information exposed through an OData GET \$metadata, it made sense that this configuration information is in an XML format similar to the EDM format used by OData. Since the underlying database should have additional implementation-specific information, the format has been modified to accommodate this. DVS provides no tooling for creating an AEDM, but any XML or text editor can be used.

The EDMtoDB.xml file in the **data** folder of the Bookshop example project was designed to illustrate many of the features. Using a copy of this file as the basis for your own designs is a good approach when getting started. All the example references in this section refer to this file. The **schema.sql** file in the **data\sql** folder should also be examined to help you understand how the AEDM configuration maps to the generated database schema.

Element Names

Many of the elements in an AEDM have a name attribute. The values for these attributes must all conform to the requirements for a JSON identifier. Typically, each element type has a different name space. It is possible for the same name to be used for an entity set, entity type, and a property. For example, see **Personnel**. Property and navigationproperty share the name space within the scope of the same parent element. You cannot have a property and navigationproperty with the same name in the same entity type. You can have the same property name in multiple complextype, and entity type declarations.

Elements of an AEDM

The primary elements in an AEDM are **EntitySet**, **EntityType**, and **Association**. **Property** and **NavigationProperty** elements help define the **EntityType**. The **schema** element provides information that is relevant to your entire VS. **ComplexType** allows other properties to be grouped for reference and **EnumType** allows definition of a property with values restricted to members of a set of valid values.

Schema

The **schema** element is the parent element for the entire configuration and holds information for the configuration as a whole.

- **Namespace** specifies the name for the scope of the **EntityType** and **EntitySet** definitions in a reversed internet domain style similar to Java packages. For example, **com.ca.example.Bookshop**.
- The optional attribute **Alias** is a shorthand version of namespace, and generally is equal to the last segment of the namespace value. In our example is **Bookshop**. When there is a need to qualify a name with a namespace, such as when specifying the value for the type attribute within a property element, or the value for **odata.type** in a POST, the value of **Alias** is used to refer to items declared in this model. If absent, DVS assumes that the last segment of the namespace value is the alias. As a best practice is to explicitly include this.
- **Version** should be set to a value of the form **n.m**. Where **n** is an integer major version and **m** is an integer minor version. This is currently not inspected, but it is strongly suggested that the major version is kept at **1**. The designer should increment this whenever there is a significant change in the format of the AEDM itself. While the minor versions be used to track changes to your specific version of the namespace being defined.

EntitySet

An **EntitySet** defines one of the resources available from the VS. Each **EntitySet** references one **EntityType**.

- The **Name** attribute value will be the name of the collection sets that are the first URL segment after the base path. By convention the name that is used is the plural of the Entity Type Name. If you want to define a collection of books, give the Entity Type the Name of *Book* and the Entity Set the Name of *Books*. However, the name space for Entity Sets and Entity Types are separate and can share the same name. It is often the case where the plural of an Entity Type name is the same as the singular. For example, *Personnel*.
- The **EntityType** attribute is the Name of the **EntityType** for this **EntitySet**.

EntityType

An **EntityType** element and its nested elements specify the details about the structure of a resource and maps to a TABLE in the underlying database. In most cases, a **EntityType** references a single **EntitySet**. However it does not have to be. Having a **Hidden** **EntityType** is useful in the specific case of supporting many to many relationships. Where the most efficient representation is to have an internal table that contains the keys for the two Entities being associated.

- The **Name** attribute value is the identifier that is used when referring to the EntityType . By convention it is a non-plural noun describing the content. For example, *Author*.
- The optional **DBTable** attribute allows specification of the name of the TABLE supporting the EntityType. If omitted, the TABLE name is an upper case version of the Name value. See the *Order* EntityType for an example where the table name does not use the default.
- The optional **CustomPropertiesEnabled** attribute if set to the value **true** enables the run-time addition of custom properties to this Entity Type in a way that is not part of the OData standard, but which is part of the feature set for DVS.
- The optional **CustomPropertiesName** attribute allows specification of the name of a special collection property that contains a set of name value pairs for the custom properties if CustomPropertiesEnabled is set to true. This name must not be the same as the name of another Property or Navigation Property for the Entity Type. See the *Order* EntityType for an example of this usage. If CustomPropertiesEnabled is set to true, and CustomPropertiesName is not part of the EntityType definition, then the custom properties appear inline with the statically defined Properties for the EntityType. See the *WorkTeam* EntityType for an example of this usage. See Custom Properties in OData Dynamic Virtual Service Features for more information about custom properties.

An EntityType contains one or more nested **property** elements, and zero or more **NavigationProperty** elements.

See the definition for the *Membership* EntityType for an example of a hidden Entity Type being used to generate an internal table for use in a many to many navigation.

Note: Management of the content of these tables is handled implicitly by DVS. Do not modify the contents directly.

Property

The **Property** element describes the individual data items that comprise the resource and map to a database COLUMN within the TABLE.

- The mandatory **Name** attribute is simply the name for the data item. It must be unique with the names of the Properties, and **Navigation properties** for this EntityType.
- The mandatory **Type** attribute specifies the format of the data. Types can be either base Types, Complex Types, EnumTypes or Collections of base Types, Complex Types, or EnumTypes.

The following supported base types include:

- "Edm.String" a text string. "Edm.Boolean" a Boolean value "Edm.Date" a date value. Input format is 'YYYY-MM-DD', for example: '2014-09-15'.
- "Edm.DateTimeOffset" a date-time value. Full Input format is 'YYYY-MM-DD[T]HH:MM:SS.sss[+-]ZZZ', for example, '2013-10-21T12:10:43.123' format, but sss (milliseconds), and SS.sss (seconds and milliseconds). The timezone offset can be omitted. Either a **T**, or a space (" ") can be used to separate the date portion from the time portion. All other values are mandatory, but Months, Days, Hours can omit leading zeros.

- "Edm.Decimal" a decimal numeric value (for example, 3.14). When in putting a decimal literal, you can add a d suffix to indicate clearly this is a decimal and not a double (for example, 3.14d).
- "Edm.Double" a double (floating-point) numeric value.
- "Edm.Int32" an integer numeric value.

To indicate that a Property is a collection, set the value of Type to **Collection(*type*)**. Where *type* is the qualified name of the type of the object which composes the collection. For example, the SkillSet Property in Bookshop.Personnel is a collection of strings as has Type="Collection(Edm.String)".

The value of Type can also be the name of a Complex Type or an Enumerated Type that is defined in your schema. Both the Schema and the Name of the Complex Type or EnumType must be specified using the format "*Alias.TypeName*". For example, the ContactInfo Property in Customer has Type="Bookshop.ContactInfo".

- The optional **MaxLength** attribute is used with the **Edm.String** type to specify the maximum length that is allowed for the string.
- The optional **DBColumn** attribute can specify the name of the COLUMN in the database. If absent, COLUMN name is the property name in upper case.
- The optional **Key** attribute indicates which Properties can be specified as a key predicate in a collection navigation. For example, the property that would be compared against the value x in the following URL, [http://myserver:9999/basepath/entityset\(x\)](http://myserver:9999/basepath/entityset(x)). At least one property MUST have the Key attribute set to true. These properties form the PRIMARY KEY for the TABLE in the database.
- The optional **DefaultValue** attribute if present holds the value that is used to set the value of the property if the property is not explicitly given a value in a POST, or PUT request body. The value that is specified is interpreted as a SQL DEFAULT expression. A string literal default must be specified within single quotes. See the ShipVia Property in the Order Entity Type, for an example of a string literal default. This convention also allows you to set the value to an expression, or SQL function, such as Current_Timestamp(). This convention is often useful for automatically generating values. See the Posted Property in the Comment Entity Type for an example.
- The optional **Nullable** attribute if set to **false** indicates that the property is required. If absent or set to **true**, the property can be omitted from POST and PUT request bodies. The value is to null or the value that is specified for DefaultValue.
- The optional **Generated** attribute if set to **once** it indicates that the value of this Property is automatically generated by DVS when the record is first created, and any value that is passed by the user is ignored. If set to **always**, then the value generation happens each time that the record is updated.
- The optional **GenMethod** attribute if set controls how a **generated** property generates its value. Valid values for this attribute are **Init**, or **Sequence**, with default **Init**. If **Init** then the value from the DefaultValue attribute is used. If **Sequence** then the value is created using a unique monotonically increasing integer which starts at 100.

Note: All properties in all entities share the same sequence generator. There is no guarantee that records added to the same entity will be given sequential numbers.

- The optional **GenPattern** attribute if set allows the user apply a format specifier to the sequence number to create a string key. The value of GenPattern is a format string using the same syntax as the Java String.format method. The format value **must** generate a unique value from a unique integer input. In practice this means that the pattern should include a "%d", or "%x" format as part of the specification. GenPattern is only used if the Property is of type String, and GenMethod is "Sequence", it is otherwise ignored. See the Id Property of Personnel f or an example of how Generated, GenMethod, and GenPattern work together.

NavigationProperty

The **NavigationProperty** element describes the relationships between EntityTypes and defines the allowable navigation from this EntityType.

- The **Name** attribute is the identifier of the NavigationProperty. It must be unique with the scope of names that are used by any of the properties and navigation properties for this entity type. This also determines the string used for the resource segment in a URL that for this navigation.
- The **Type** attribute specifies the name of the destination EntityType in the navigation.
- The **Relationship** attribute specifies the name of the association that contains the database implementation details for the navigation. In cases where two-way navigation is allowed between two entity types, each EntityType with have a NavigationProperty containing the same relationship value.
- The **Multiplicity** attribute specifies if there can be multiple entity instances of the targeted entity type reachable through this navigation (""), or zero or one instances ("0..1"). **For instance the Books NavigationProperty in the Author EntityType has a "{" Multiplicity value, because any single Author can have written multiple books. On the other hand, the Inventory NavigationProperty in the Book EntityType has a Multiplicity value of "0..1", because in the model that is used for Bookshop, a Book record is associated with only one Inventory record.**

Association

An **Association** element specifies the implementation details of one (for unidirectional navigations), or two (for bidirectional navigations) Navigation Properties.

- The **Name** attribute must be unique within Association elements, and must match the Relationship value for the affiliated navigation properties.
- The **DBLinkTable** attribute MUST be the name of one of the EntityType TABLE names when the navigation is one to one or one to many. If the navigation is many to many, then this value MUST be the TABLE name of the intermediate table which holds the keys of the two Entities that are the end points of the navigation. See the *Worker_Team* Association element definition for an example of this usage. If not many to many, and Properties exist in both tables that can be used to associate records of the two Entity Types, this value MUST be the name of the table on which the database referential integrity constraint is placed. See the *Author_Book* Association element definition for an example of this case. Finally, in an association is for a navigation to and from the same Entity Type (and Table). The value is the table name of the one table involved. See the *Personnel_Supervisor* or *Personnel_Staff* Association element definitions for an example.

- The **DBJoinOn** attribute MUST be a list of one or more JOIN pairs of the form *TABLE1.COLUMN1=TABLE2.COLUMN2*.

The simplest case is where the association is between two different entities and a property exists in both Entity types that can be compared for equality. For example, see the *Author_Book* Association element definition for an example of this case.

If multiple properties are needed to connect two entities that both contain the properties that can be compared for equality, then each JOIN pair is a separate entry into the comma-separated list.

In the case where an association is for a navigation property that points back to the same Entity Type, the JOIN pair looks like *TABLE1.COLUMN1=TABLE1.COLUMN2* and which Property COLUMN has the role of *COLUMN1* and which has the role of *COLUMN2* effects the results that are returned from the navigation property. The *COLUMN1* role should be the Property COLUMN that should be matched for the Parent record. For example, the Navigation Property *Supervisor* in *Personnel* has the semantics of pointing to the *Personnel* record of the supervisor for a given *Personnel* record, while the Navigation Property *Staff* has the opposite meaning. As such each of the navigation properties required a separate Association element (*Personnel_Supervisor* and *Personnel_Staff*). *Personnel_Supervisor* has a DBJoinOn value of "PERSONNEL.SUPERVISORID=PERSONNEL.ID" because SUPERVISORID has the key for the supervisor's *Personnel* record, so DVS processed from the employees record to the supervisors record. Conversely, *Personnel_Staff* has a DBJoinOn value of "PERSONNEL.ID=PERSONNEL.SUPERVISORID" because the intent is to match all records whose SUPERVISORID matches the supervisor's *Personnel* record.

Finally in the case of many to many navigation properties, or navigation properties where there do not exist properties in each entity type that can be matched together, the JOIN pair list must contain equijoins from the TABLE for the source entity type to a hidden internal TABLE, and from the hidden internal TABLE to the destination TABLE. See the *Membership* entity type definition, the *Personnel*, and *Teams* navigation properties in the *WorkTeam* and *Personnel Entity* Types, and the *Worker_Team* Association for an example of how this all hangs together.

EnumType

The **EnumType** is a way of declaring a data type that only accepts certain values.

- The **Name** attribute when combined with the namespace Alias is used as the type value for those properties which are using this EnumType. For example, see the status enumtype and the status property in order .
- The optional attribute **UnderlyingType** which can have the value of any integer type or **Edm.String** . If absent the default is **Edm.String**. This controls the type of the database column that is created for any properties declared to use the EnumType.
- In addition, an EnumType declaration **must** include one or more member elements. Each **member** element must include a **name** attribute, which is used as one of the permissible values for Properties declared to use the EnumType, and an optional **value** attribute, whose value is the numeric sort weight that is given to this member. If absent, the sort weight is one higher than the sort weight of the previous member. The sort weight starts at zero where there is no previous explicit declaration.

ComplexType

A **ComplexType** is a means by which related properties can be grouped and referenced as a whole.

- The **name** attribute when combined with the namespace Alias is used as the type value for those properties which are using this ComplexType.
- In addition, an ComplexType declaration **must** include one or more property elements. These property declarations cannot include generated values.

Defining a VS in RAML Format

In general, the RAML format does not provide as fine as degree of control over your VS model. It does have the advantage of allowing the initial data to be included directly in the RAML file, and can also be processed into a virtual service without having to use the eclipse plug-in. The library.raml file contains an example of a simple model with authors and books which includes initial data.

Metadata declaration

You must define top-level schemas containing type definitions for all EntityTypes and EntitySets. Schemas must conform to the schema format in {+} <http://json-schema.org/draft-04/schema+>. In order to identify key fields, each schema definition must have an additional member (primaryKeys), containing a list of primary key property names.

For example:

```
- book: |
  {
    "$schema": "http://json-schema.org/draft-04/schema",
    "type": "object",
    "description": "A single book",
    "properties": {
      "id": { "type": "integer" },
      "name": { "type": "string" },
      "isbn": { "type": "string" },
      "author_id": { "type": "integer" }
    },
    "required": [ "id", "name", "isbn" ],
    "primaryKeys": [ "id" ]
  }

- books: |
  {
    "$schema": "http://json-schema.org/draft-04/schema",
    "type": "object",
    "description": "a collection of books",
    "properties": {
      "size": { "type": "integer" },
      "books": {
        "type": "array",
        "items": { "$ref": "book" }
      }
    },
    "required": [ "size" ]
  }
```

All resource paths must be defined contiguously, according to standard OData format. Segmented path definitions are **not** handled.

For example:

```
/books:
  get:
```

```

        description: Get a list of all of the books in the system
        responses:
          200:
            Body
            :
    /books({id}):
      get:
        description: Get a specific book
        responses:
          200:
            Body
            :

```

Do NOT specify like the following information:

```

    /books:
      get:
        description: Get a list of all of the books in the system
        responses:
          200:
            Body
            :
    ({id}):
      get:
        description: Get a list of all of the books in the system
        responses:
          200:
            Body
            :

```

Sample Data

Sample data must be inserted as JSON and is only considered when associated with a GET method for a resource. The associated schema name is specified within the application/json body type section.

For example, data that are associated with an entity set should look like the following information:

```

    /books:
      get:
        description: Get a list of all of the books in the system
        responses:
          200:
            body:
              application/json:
                schema: books
                example: |
                  {
                    "size": 10,
                    "books": [
                      { "id": 1, "name": "The Hobbit; or, There and
Back Again", "isbn": "054792822X", "author_id": 1 },
                      { "id": 2, "name": "The Fellowship of the
Ring", "isbn": "B007978NPG", "author_id": 1 },
                      { "id": 3, "name": "The Two Towers", "isbn":
"B007978PKY", "author_id": 1 },
                      { "id": 4, "name": "The Return of the King",
"isbn": "054792819X", "author_id": 1 },
                      { "id": 5, "name": "1984", "isbn":
"0451524934", "author_id": 2 },
                      { "id": 6, "name": "Animal Farm: A Fairy
Story", "isbn": "B003K16PUU", "author_id": 2 },
                      { "id": 7, "name": "The Sun Also Rises",
"isbn": "0743297334", "author_id": 3 },
                      { "id": 8, "name": "For Whom the Bell Tolls",
"isbn": "0684803356", "author_id": 3 },
                      { "id": 9, "name": "The Old Man and the Sea",

```

```

"isbn": "B000FC0SH8", "author_id": 3 },
{ "id": 10, "name": "A Farewell To Arms",
"isbn": "0684801469", "author_id": 3 }
]
}

```

Sample data that are associated with a single entity should look like the following information:

```

/books({id}):
  uriParameters:
    id:
      description: Book ID
      type: integer
  get:
    description: Get a book
    responses:
      200:
        body:
          application/json:
            schema: book
            example: |
              {
                "id": 1, "name": "The Hobbit; or, There and Back
Again", "isbn": "054792822X", "author_id": 1
              }

```

Deploying Your Virtual Service

This page provides information about deploying your virtual service.

Prerequisites for Deploying When Using the DVS Assistant (Eclipse Plug-in) and DevTest Solutions

1. DVS Assistant Eclipse plug-in is installed.
2. OData DVS is deployed into the hotDeploy folder of your target VSE.
3. You have access to or have created an AEDM or RAML configuration file.
The configuration file describes the entities and navigation properties that your VS uses. See [Designing Your Virtual Service](#) for more information.

If any of these prerequisites are not met, see [Installing Dynamic Virtual Service](#) for more information about installing DVS.

Creating the Virtual Service MAR File Through Eclipse

1. From the main Eclipse menu, select **DVS Assistant, DVS Assistant View**.

The **DVS Assistant View** window displays:

The screenshot shows the 'DVS Assistant View' window. It contains several sections for configuring a VS Model:

- VS Model:** Name: Bookshop, OData Version 4 (dropdown).
- Save VSM File To:** C:\Users\Booksh\dvs (with a Browse button).
- Virtual HTTP/S Listener:** Listen port: 8844, Base URL: /odata/v4/Bookshop.
- Specify Service Model File:** C:\Users\Booksh\Documents\LISA\Projects\Bookshop\data\EDMToC (with a Browse button).
- Service Model:** AEDM (selected) and Raml (radio buttons).
- Virtual Service Transactions:** A table with columns: Method, Resource Path, and Description. It lists various GET requests for Authors, Books, and Comments.
- Information:** The VS Model has been saved as C:\Users\Booksh\dvs\Bookshop.mar. Please check the log file for the details C:\Users\Booksh\ca.com.ca.dvs.utilities.lisamar\logs\default.log.
- Generate VS:** A button at the bottom right.

2. Enter the following information in the fields:

- **VS Model:** The name of the MAR file that is created and the project contain in the zip file.
- **OData Version selector:** Select either OData Version 4 or OData Version 3.
- **Save VSM File To:** The folder where the MAR file is created.


Note : If you extract the contents of your MAR file to examine it through the DevTest Workstation, do **not** extract it into the same folder where the MAR resides. The DVS Assistant uses this location to build a temporary image of the MAR file which it cleans up afterwards. If your project is located there, your project is removed as part of the clean-up.

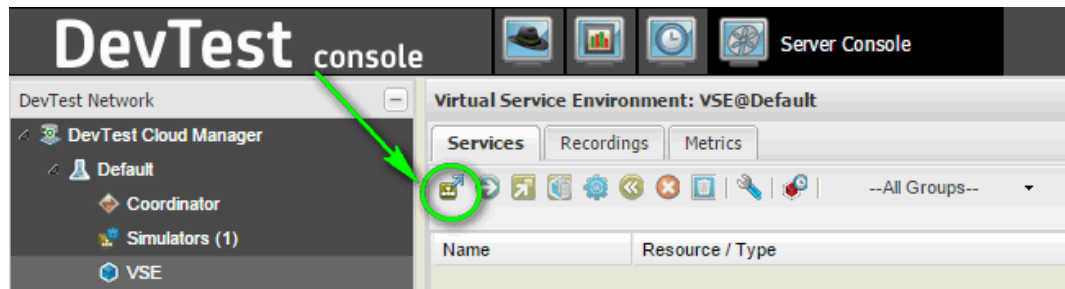
- **Listen port:** Integer port value. Specify a value that is not used on the systems where your VS is deployed.
 - **Base URL:** Base part of the URL for your VS.
 - **Specify Service Model File:** Full file specification for the configuration file which describes the VSE. The AEDM or RAML buttons display which files are shown when browsing to select the Service Model file. The selected buttons also determine how that file is processed.
3. If the plug-in has been configured to populate transactions, you preview your valid resource paths by right-clicking any of the column headers of the **Virtual Service Transactions** table, and selecting **Populate All Transactions**.

4. Click **Generate VS** to create your MAR file.
If successful, a message displays at the bottom of the screen with Information about the MAR file.
5. If a warning or an error status displays, do the following:
 - a. Review the log.
 - b. Correct the problem in your configuration definition and repeat step 4.

Deploying the MAR File

Deploy the newly created MAR file into any VSE with the OData DVS extension that is installed the hotDeploy folder. Use any of the following methods supported by DevTest Solutions:

- Uploading from the DevTest Server Console by clicking **Deploy/Redeploy a virtual service to the environment**  :



Screen capture of the Server Console

- Using the deployMar method of the Invoke 2.0 REST API for DevTest Solutions. See [CA Support Documentation](#) for the log in credentials that are required.

Once deployed, the virtual service is managed using any of the facilities that are provided by DevTest. See DevTest Solutions documentation for more information.

Prerequisites for Deploying When Using the DVS Servlet

1. DVS Servlet into a Tomcat instance is installed.
2. OData Dynamic VS is deployed into the **hotDeploy** folder of the target VSE.
3. You can access and have created a RAML configuration file describing the entities and navigation properties your VS uses.
See [Designing Your Virtual Service](#) for more information.

If any of these prerequisites are not met, see [Installing Dynamic Virtual Service](#) for more information.

Creating and Deploying

The entire operation is done using on REST call to the **Transform RAML to OData Virtual Service** service. See [OData Dynamic Virtual Service Features](#) for parameter details.

After, you can download the MAR file from your VSE using the DevTest Server Console. Then use the contents of the MAR file as a DevTest project as described in the following section.

Working with the MAR as a DevTestProject

1. Using an archive tool such as zip extract the contents of the MAR file into any folder.
2. From the DevTest Workstation, open the project.
3. You can perform any operations that you could typically perform on a project.
4. To redeploy the modified project, right-click the **MAR Info file** and select **Deploy/Redeploy to VSE@Default...** Or select **Build Model Archive...**, then deploy the new MAR file using any of the methods described under **Deploying the MAR File** section.

Note: Do **NOT** deploy the VSM file. Deploying the VSM file will only deploy the VSM and not the supporting data file. As a result of deploying the VSM file, the virtual service fails. Typically you see an exception similar to the one below, but other errors are possible.

The following graphic displays an exception:



See [OData Virtual Service Features](#) for a description of the supported OData feature. See [Managing Your Virtual Service Data](#) for more information about how to manage data.

Managing Your Virtual Service Data

While a DVS is running, entity data is held in an in-memory H2 SQL Database instance. This process allows quick response and simple deployment because there is no need to have access to or an installation of RDBMS.

The database is resident in memory only. Each time DVS starts, data is re-initialized from the SQL script **current.sql** in the DVS data folder for the virtual service. The feature is beneficial because it allows reset to a known state. If modifications to the data have been made and retained, each time a VS is restarted, **all** updates are reset upon restart. There is a way to retain all the updates and inserts. Upon the shutdown, contents of the database are written out to **new.sql** in the DVS data folder for the virtual service. You copy this file over **current.sql** before the virtual service is restarted. All the updates made up to the previous shutdown are preserved. As a best practice, use the Runtime Management of Virtualization Data features. These features create named snapshots of the data and manage your data through this REST API.

The DVS data folder is under LISA_HOME (default value **C:\Program Files\CA\Lisa** on windows) with the rest of the folder path that is taken from the DDME_DATA_DIR LISA variable in projects.cfg. By default, this is **dvs.data/project_name**. The SQL initialization files for a DVS DevTest virtual service named **Nightsky**, by default is located in the folder **C:\Program Files\CA\Lisa\dvs.data\Nightsky**.

Note: Older projects can have data stored in a folder named **casd.data**. The folder that is used can be determined by examining the DDME_DATA_DIR LISA variable in projects.cfg.

Runtime Management of Virtualization Data

The DVS Admin API provides a means of servicing internal operations on any service that is based on the Dynamic Virtualization model. Currently, this API only provides for the management of SQL initialization scripts for the Dynamic Virtualization Data Store.

Web Client Requests

Obtain a List of Existing DataStore Checkpoints

Method: GET

Path: /dynvs_admin/Checkpoints

Keys: (none)

Results:

Result	HTTP-Status-Code	HTTP-Status-Text	Content-Type	Body	Description
SUCCESS	200	OK	application/json	List of JSON objects describing the scripts found	Method succeeded

Load a New Data Store From Specified Checkpoint

Method: GET

Path: /dynvs_admin/Checkpoints({id})

Keys: (none)

Params:

Name	Type	Description	Example
id	String	The name of the script used to initialize a new datastore. The id key may not be specified. The associated value is a quoted string value, indicating the name of an existing DataStore initialization script.	GET /casd_vs_admin/Checkpoints('new.sql')

Results:

Result	HTTP-Status-Code	HTTP-Status-Text	Content-Type	Body	Description
SUCCESS	200	OK	application/json	JSON objects describing script used to initialize datastore	Method succeeded. Existing DataStore is replaced by a new one, initialized by the specified script.
FAILURE	404	Not Found	text/plain	Message describing failure	Method failed because the specified script could not be located.
FAILURE	500	Internal Server Error	text/plain	Message describing failure	Method failed (see body for more information).

Save Active Data Store to Script

Method: PUT

Path: /dynvs_admin/Checkpoints({id})

Keys: (none)

Params:

Name	Type	Description	Example
id	String	The name of the script that is used to initialize a new datastore. The id key may not be specified. The associated value is a quoted string value, indicating the name of an existing DataStore initialization script.	PUT /casd_vs_admin/Checkpoints('new.sql')

Results:

Result	HTTP-Status-Code	HTTP-Status-Text	Content-Type	Body	Description
SUCCESS	204	No Content	(none)	(none)	Method succeeded. Existing DataStore was saved to the specified initialization script
FAILURE	500	Internal Server Error	text/plain	Message describing failure	Method failed (see body for more information)

Delete a Data Store Initialization Script

Method: DELETE

Path: /dynvs_admin/Checkpoints({id})

Keys: (none)

Params:

Name	Type	Description	Example
id	String	The name of the script that is used to initialize a new datastore. The id key can be specified or cannot be specified. The associated value is a quoted string value, indicating the name of an existing DataStore initialization script.	DELETE /casd_vs_admin/Checkpoints('new.sql')

Results:

Result	HTTP-Status-Code	HTTP-Status-Text	Content-Type	Body	Description
SUCCESS	204	No Content	(none)	None	Method succeeded. Specified DataStore initialization script was deleted.
FAILURE	403	Forbidden	text/plain	Message describing failure	Method failed because the specified script could not be deleted.
FAILURE	404	Not Found	text/plain	Message describing failure	Method failed because the specified script could not be located.

Unsupported Requests

**** Any requests other than the requests documented previously return a HTTP-Status-Code of 405 (Method Not Allowed)***

Backing up Data

To back up DVS data for a DVS DevTest VS, make a copy of the DVS data folder for that service.

Note: If the service is running, any updates that are not yet written to a SQL script are not preserved.