Progress®

Corticon
Data Integration

Progress® Corticon®

Copyright

© 2019 Progress Software Corporation and/or one of its subsidiaries or affiliates. All rights reserved.

These materials and all Progress® software products are copyrighted and all rights are reserved by Progress Software Corporation. The information in these materials is subject to change without notice, and Progress Software Corporation assumes no responsibility for any errors that may appear therein. The references in these materials to specific platforms supported are subject to change.

Corticon, DataDirect (and design), DataDirect Cloud, DataDirect Connect, DataDirect Connect64, DataDirect XML Converters, DataDirect XQuery, DataRPM, Deliver More Than Expected, Icenium, Kendo UI, Kinvey, NativeScript, OpenEdge, Powered by Progress, Progress, Progress Software Developers Network, Rollbase, SequeLink, Sitefinity (and Design), Sitefinity, SpeedScript, Stylus Studio, TeamPulse, Telerik, Telerik (and Design), Test Studio, and WebSpeed are registered trademarks of Progress Software Corporation or one of its affiliates or subsidiaries in the U.S. and/or other countries. Analytics360, AppServer, BusinessEdge, DataDirect Autonomous REST Connector, DataDirect Spy, SupportLink, DevCraft, Fiddler, JustAssembly, JustDecompile, JustMock, NativeChat, NativeScript Sidekick, OpenAccess, ProDataSet, Progress Results, Progress Software, ProVision, PSE Pro, SmartBrowser, SmartComponent, SmartDataBrowser, SmartDataObjects, SmartDataView, SmartDialog, SmartFolder, SmartFrame, SmartObjects, SmartPanel, SmartQuery, SmartViewer, SmartWindow, and WebClient are trademarks or service marks of Progress Software Corporation and/or its subsidiaries or affiliates in the U.S. and other countries. Java is a registered trademark of Oracle and/or its affiliates. Any other marks contained herein may be trademarks of their respective owners.

Updated: 2019/07/31

Table of Contents

Why your rules might want to access external data.....9

Corticon alternatives for data integration.....11

How Corticon concepts apply to Datasources.....15

About the sample projects referenced in this guide.....19

How Datasource information is viewed in the Vocabulary.....23

Getting Started with EDC.....27

Define a table namespace in the database.....28
Define the database connection for EDC.....28
Set the entities to store in the database.....30
Load the schema and data in the database.....31
Importing database metadata into an EDC Vocabulary.....32
Test the rules when reading from the database.....33
Test the rules when writing to the database.....34

Getting Started with ADC.....37

Overview of the Advanced Data Connector.....38
Define a table namespace in the database for ADC.....38
Create and map the ADC schema and queries.....39
Enable the project for ADC.....41
Define a database connection for ADC.....42
Define and import queries for ADC.....44
Import ADC Datasource metadata into a Vocabulary.....47
Use an ADC connection as a Ruleflow service callout.....48
Test the rules when reading from the ADC database.....49
Test the rules when writing to the ADC database.....51

Getting Started with Multiple Database Connectivity.....53

Define multiple table namespaces 54
Create and map the multiple database schemas.....54

Enable the project for multiple databases....57
Define multiple database connections....58
Define and import queries for multiple databases....61
Import multiple Datasource metadata into a Vocabulary....63
A closer look at MDB metadata....65
Use multiple database connections as Ruleflow service callouts....67
Test the rules when reading from multiple databases....69
Test the rules when writing to multiple databases....70

Getting Started with REST....73
Overview of the Autonomous REST Connector....74
Enable the project for REST SCO....74
Define a Datasource connection for REST....75
Create and map the REST schema....77
Use REST data sources in a Ruleflow....80
Test rules when importing from the REST Datasource....81
Revise Connection and Service Call-out to retrieve data....82

Mixing REST and database access....85

Deploying projects that use data integration....89
Exporting the **Datasource** Configuration file....89
Package a project in Corticon Studio for Corticon Server....92

Getting Started with Batch....93

A closer look at how Corticon relates to Datasources....99
SmartMatching of Vocabularies to databases....99
Validation of names against SQL keywords and database restrictions....100
Support for catalogs and schemas....101
Fully-qualified table names....101
Filtering catalogs and schemas....101
Support for database views....102
Associations as join expressions....102

Advanced EDC Topics....107
Setting EDC Vocabulary properties....107
Editing Entity EDC properties....108
Editing Attribute EDC properties....111
Editing Association EDC properties....120

Mapping and validating EDC database metadata....121
Mapping EDC database tables to Vocabulary Entities....121
Mapping EDC database fields (columns) to Vocabulary Attributes....123
Mapping EDC database relationships to Vocabulary Associations....123
Validating EDC database mappings....124
Types of mapping validation and validation errors....125
Setting additional EDC Datasource connection properties....126
How data from an EDC Datasource integrates into rule output....128
When Datasource access is Read Only....129
When Datasource access is Read/Update....134
Using caches in EDC for entities and queries....138
Specifying caching on Vocabularies and Rulesheets....139
Settings for EDC caching....141
Working with database caches....142
Metadata for Datastore Identity in XML and JSON Payloads....145
Relational database concepts in the Enterprise Data Connector 146
Identity strategies....147
Advantages of using Identity Strategy rather than Sequence Strategy....149
Key assignments....149
Conditional entities....151
Dependent tables....152
How EDC handles transactions and exceptions....152

Advanced ADC Topics....155

Mapping ADC database metadata....155
Mapping ADC database tables to Vocabulary Entities....156
Mapping ADC database fields to Vocabulary Attributes....157
Mapping ADC database relationships to Vocabulary Associations....157
Configuring ADC....158
Configuring ADC reads....159
Configuring ADC writes....160
Configuring batch....162
Configuration details....162
How Corticon is expressed in SQL....163
Tips and techniques in SQL data integration....164

Advanced REST Datasource Topics....167

Authentication on REST Service connections....168
Parameters on REST Service connections....170
Import REST Datasource metadata into a Vocabulary....171
Mapping REST Service metadata....173
Define your preferred schema....176
Steps in mapping REST Service metadata....176

Defining associations in REST Service metadata....180

Appendix A: Supported RDBMS brands and features for Corticon EDC.183

IBM DB2....184
Microsoft SQL Server....185
MySQL....186
Postgres....186
Oracle Database....187
Progress OpenEdge....188
Progress OpenAccess....189

1

Why your rules might want to access external data

Corticon provides the flexibility to either pass the data in on the call to the decision service or to have the decision service retrieve data, or a mix of both. What you choose depends on your needs.

In most Corticon deployments, the data is passed in. This simplifies the architecture because you don't need to account for Corticon connecting to external data sources. This is especially true when you have legacy systems which cannot be easily accessed. In some cases, the data passed in can be large. For example, all the data needed to process a loan application for a single applicant.

In some deployments, Corticon needs to retrieve supporting data. This adds additional connections to the architecture yet it can be a considerable savings to not have to pass in all the data in the request. This is especially true where the data is selective – when the rules are choosing some subset of data that is needed. This can also be useful with reference data that you want to cache.

In other deployments Corticon needs to retrieve the data for efficiency. An example would be a batch application where you need to process a billion records at the end of the month. In such cases, efficient moving of data is essential for performance.

2

Corticon alternatives for data integration

Corticon provides several techniques for data integration. Which ones are right for you depends on your use case – you can assemble the right mix to suit your needs.

When to use the Enterprise Data Connector (EDC)

Corticon EDC accesses data is a relational database to augment data when processing a discrete Decision Service request. It is used by rule authors who like the user friendly and intuitive way of modeling data access and persistence in their Rulesheets to conditionally enrich transactional data, and to use reference data for rule processing using a single backend relational database.

For example, a single request to adjudicate an insurance claim tells Corticon to retrieve all required data and related data from a database to service the request. Corticon performs well in this scenario including the persistence of the claim processing result back into the database.

Corticon EDC uses an object relational model (ORM) where entities mapped to a database are automatically enriched with database data and optionally saved back to the database. This approach makes it very easy to use database data in your rules but also introduces query and data processing overhead when reading data from a database with related tables. Both read and write performance can suffer in some use cases with EDC. When performing reads, the number of queries required to retrieve an entity and all associated entities can grow exponentially with each level of associations. When performing writes, each updated object is committed to the database as a distinct update instead of a large, single pass multi-record update.

Database read and update through EDC is a good choice for a single Decision Service request scenario with limited amounts of data. Examples are individual claims processing, single client eligibility requests, and a single transaction validation request. Many Corticon users have EDC running every day with EDC deployment scenarios.

EDC's limitations are in its performance in large, data intensive operations where large chunks of data are loaded into Corticon for processing and updating the database.

EDC can be thought of as the "Easy Data Connector" because it provides the simplest means of connection to a database. This contrasts with the "Advanced Data Connector" which requires greater knowledge of SQL but provides greater flexibility.

When to use Advanced Data Connectors (ADC)

ADC also provides for accessing data in a database but takes a different approach than EDC to address different use cases. ADC is very efficient in dealing with large data sets. ADC has the ability to connect to disparate databases--you can use ADC to read from and write to multiple databases in a single decision service. With EDC you're limited to one database. Both read and write performance is much better than EDC when processing larger data sets in a Decision Service request. ADC can read related data in a few passes from the database, where EDC requires discrete queries to fetch data. And ADC can write back data in chunks, where EDC writes data as discrete updates.

Both ADC and EDC are single-threaded -- a single request is executed by a single Decision Service reactor which has access to the full collection of data included in the request payload, database exposed entities and filter criteria in the Rulesheets (EDC) or ADC queries.

Using ADC, you get great performance when processing a large dataset through a single Decision Service request. You can use ADC to quickly process a set of unrelated records such as individual customers or work orders. You can also use it when you need to do operations such as aggregations and clustering. You can build rules that operate on the full collection of data. As examples, you can quickly adjudicate all medical claims in a month or approve specific procedures across all hospitals in a specified region, or calculate sales prices for all items in stock. In some situations, it is imperative to have access to the full collection of data for your rules to work properly. For example, when sales prices are calculated based on clustering rules whereby all sales prices for products in the same cluster are based on the average purchase price of their respective product cluster.

When to use the REST Datasource

The Corticon REST Datasource provides support for accessing REST services. It allows you to retrieve REST data to enrich the payloads being processed by your rules.

The Corticon REST Datasource uses the Data Direct Autonomous REST Connector which provides the ability to access REST services as if they were databases. This is beneficial to a Corticon user because the process of mapping a vocabulary to a REST service is the same as for EDC and ADC data sources.

To configure the REST Datasource you either perform schema discovery or supply your own schema file. When using schema discovery, you supply the URL of the REST data source and query parameters and allow the Autonomous REST Connector to generate a schema for your REST service. To supply your own schema, you can either export a discovered schema from Corticon Studio and make edits or create one from scratch. See the Autonomous REST Connector documentation for details on the its schema file format.

The query parameters for a REST Datasource can either be fixed or dynamically set by data in your payload. Dynamic setting of parameters allows you to access a REST service to retrieve information about a specific entity in your payload. You can also configure the security settings for accessing a REST service.

As REST access is limited to read-only, it is ideal for data enrichment. You could have one or several REST Datasources used by a decision service. You can even mix EDC or ADC with REST depending on your data access needs.

The wealth of REST data sources exposed through APIs means that you could be touching multiple sources to build the best complete data set possible. In marketing scenarios that might mean taking sparse info on a prospect from social or business contacts to enrich the data by discovering their profile and preferences to focus campaigns and assign local reps for follow-up. In medical applications, diagnoses and treatments can be enriched with claims approval histories or related clinical trials. For mortgage lenders, quickly scanning multiple credit review resources for a prospect, and then matching their home value and loan to retrieve the best rate from multiple lenders.

When to use batch processing

Batch processing is used to process large data sets either after hours, during periods of low system usage or to meet business demands such as monthly or quarterly reporting. Corticon's batch processing can be used with ADC to efficiently process huge amounts of data. Batch decision services can also use REST Datasources.

A requirement for batch processing is that each transaction stands on its own, not needing access to the full collection of data to make decisions on single transactions. As only so much data can be loaded into Corticon working memory at once, the data would need to be fed to the rules engine in chunks to then process the chunks concurrently based on resource capacity. Note that there are no return payloads in batch processing – the result of all the rule processing is persisted in the database.

Batch processing usually runs against the same input source to process large volumes of data so it is set to run at scheduled such as nightly or monthly. Corticon's Web Console can be used to schedule batch executions or you can use external tooling to perform scheduling and call Corticon REST API to start a batch execution..

3

How Corticon concepts apply to Datasources

Internally, Corticon accesses all data sources using SQL. A Corticon Vocabulary is fundamentally relational in nature, and conceptually equivalent to the elements of a typical relational database:

Corticon Vocabulary	Relational Database
Vocabulary	Schema
Vocabulary: Entity	Table
Vocabulary: Attribute	Table Column or Field
Vocabulary: Association	Relationship between Tables
Ruletest Output	Table Row(s) or Record(s)

How REST can conform to a relational database schema

Corticon uses the Progress® DataDirect® Autonomous Rest Connector to access REST Datasources, which maps the returned JSON to a relational database schema and translates SQL statements to REST API requests. When configuring a REST Datasource you can either have the Autonomous Rest Connector discover the schema for a REST service or provide your own schema file. The schema tells the Autonomous Rest Connector how to map JSON in memory database tables which Corticon will then access with SQL.

JSON data is hierarchical, not relational, but can be mapped to a relational model. When using schema discovery, Autonomous Rest Connector will determine how to perform this mapping. If you need finer control over the mapping, you can provide your own schema file.

The Autonomous Rest Connector handles the complexities of mapping JSON to a relational representation but you need to understand the rules applied to perform the mapping. This is best done by example. Imagine you have a REST service which returned the JSON.

```
{
    "applicant": [
        {
            "name": "Sydney Smith",
            "income": 57000,
            "address": {
                "street": "101 Main Street",
                "city": "Raleigh",
                "state": "NC"
            },
            "children": [
                {
                    "name": "Robert Smith",
                    "dob": "2017-04-19"
                },
                {
                    "name": "Chelsea Smith",
                    "dob": "2014-11-07"
                }
            ]
        }
    ]
}
```

Looking at this, you can see there is one applicant, Sydney Smith, with two children and other information about Sydney. The Autonomous REST Connector would represent this relationally as two tables, "applicant" and "children". The applicant table would contain columns for:

- name
- income
- street
- city
- state

The "children" table would contain columns for

- name
- dob

When mapping the JSON to in memory tables, the Autonomous Rest Connector would map Sydney to the "applicant" table and her two children to the "children" table. The Autonomous Rest Connector would also define a primary key/foreign key relationship between the tables so that the children for an applicant can be identified.

In this simple example:

- The applicant and their attributes were added to the applicant table
- The nested address data was "flattened" and made attributes of the applicant
- The nested array of children objects were mapped to an associated table

Corticon Vocabulary	Relational Database	REST mapping
Vocabulary	Schema	Schema (even if implicit)
Vocabulary: Entity	Table	Object

Corticon Vocabulary	Relational Database	REST mapping
Vocabulary: Attribute	Table Column or Field	Number, string, or null
Vocabulary: Association	Relationship between Tables	Array of objects, strings, or numbers
Ruletest Output	Table Row(s) or Record(s)	Object Instances

For an example a schema in a REST Datasource, see Mapping REST Service metadata on page 173 .

4

About the sample projects referenced in this guide

The *Getting Started* techniques for data integration are presented in this guide using a sample of medical patient records and treatments that have been performed on the patient. The samples provide the SQL statements that setup the table and sample data for many supported databases. All the samples build on each other so that you understand that what is under discussion is evolution of functionality in Corticon data integration.

The scenarios demonstrate the essential concepts of the various data integration options. The corresponding Corticon Studio sample projects use one or more databases or REST services. Where a database is needed, the samples include SQL scripts to define the schema and load sample data for many supported databases. Where REST is needed, the samples use a test REST service hosted by Progress on AWS. With the exception of the Batch Rule Process sample, the samples are independent. The Batch Rule Processing sample requires the ADC Database Connectivity sample's database configuration to have been performed and the decision service deployed to Corticon Server.

For database samples, this guide demonstrates usage with Microsoft SQL Server 2014. The techniques can be applied to other supported databases.

This guide refers to the included SQL scripts by their logical name. For example, the SQL script that sets up the patient schema and data on SQL Server is the file `sql/sqlserver/patient_sqlserver.sql`. This guide refers to that as simply `patient`. Once a script has been run in the database, it does need to run again for another sample as the script is the same.

Each sample section starts with advice about advancing from the previous section. Each topic within a *Getting Started* section indicates how hands-on users can just read through the steps that are pre-defined in the sample project assets.

If you choose, you could start at Mixed Connectivity, and work backwards to the other samples. You might see some unneeded data and tables yet all the required metadata and SQL Queries will process the samples as expected.

There are six Corticon samples that relate to data integration:

1. **EDC Database Connectivity** - The classic database connectivity in Corticon is EDC. The richness of database interaction is defined within Rulesheets. While this can be constraining, its simplicity is appropriate for many applications, as illustrated:

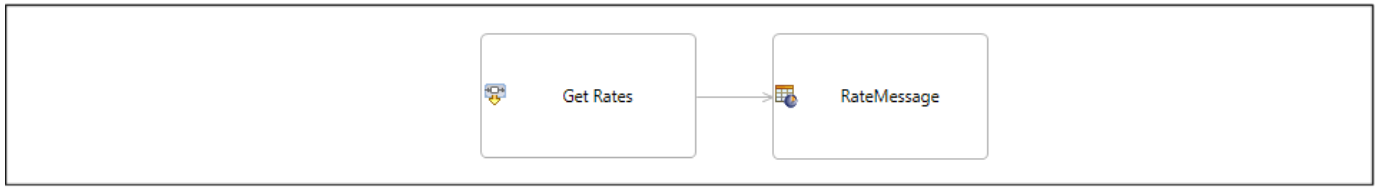
The EDC sample can be used as the basis for the ADC sample. It is a good idea though to close the EDC asset files to ensure that you keep the samples distinguished. SQL script: `patient`.

2. **ADC Database Connectivity** - Corticon Extensions are the foundation of the ADC functionality. The defined functions enable read and write functionality that are implemented in the sample's Ruleflow as Service Call-outs, where one call-out is enabling read functions while the other enables write functions, as illustrated:

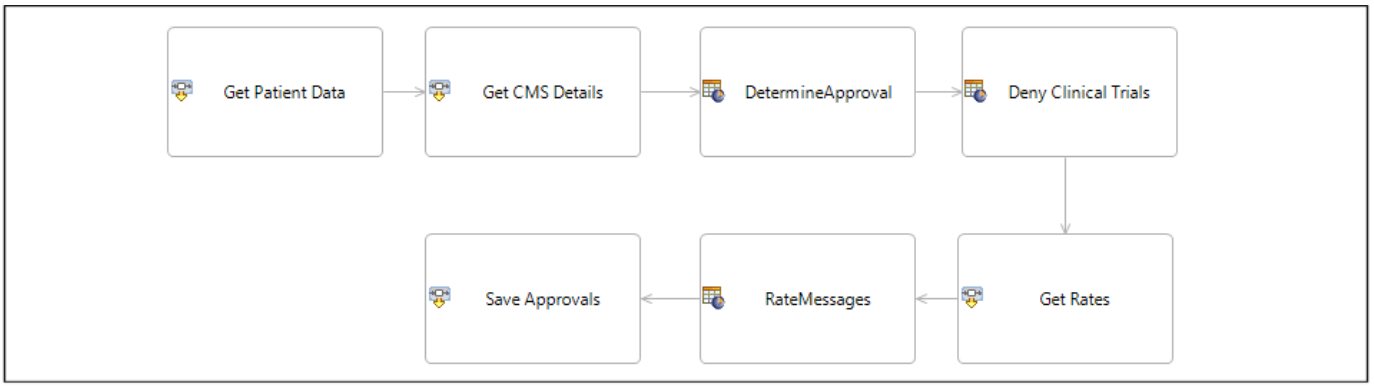
The ADC sample can be used as the basis for the Multiple Database sample and is needed by the Batch Rule Processing sample. SQL scripts: `patient` and `adc`.

3. **Multiple Database Connectivity** - With ADC you can access multiple databases. The data read in from one database can even be used when querying data from another database. This sample will demonstrate the use of ADC to read patient and treatment data from one database and then access a second database to retrieve detailed information about a type of treatment. The rules determine if a treatment is approved and the results are saved to the patient and treatment database, as illustrated::

4. **REST Connectivity** - The REST sample demonstrates the use of Corticon's REST Connectivity for accessing REST services from rules. The sample calls a REST service to retrieve the reimbursement rates for a medical procedure given a procedure code. There may be multiple rates for a procedure with different effective date ranges..

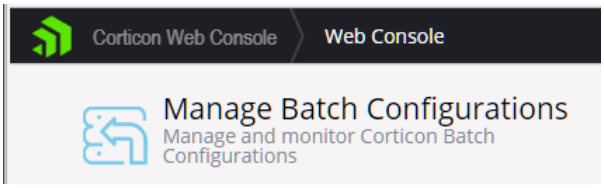


5. **Mixed Connectivity** - This sample mixes ADC and REST datasources to demonstrate the flexibility of Corticon's data access capabilities. The sample builds on the Multi Database Sample, adding to it the retrieval of reimbursement rate data as is done in the REST connectivity sample.



.

6. **Batch Rule Processing** - The batch sample does not include any rule assets. It contains SQL scripts to populate a test database with additional records to better explore the batch rule processing concepts. This guide will show how to use the Web Console perform batch rule processing.

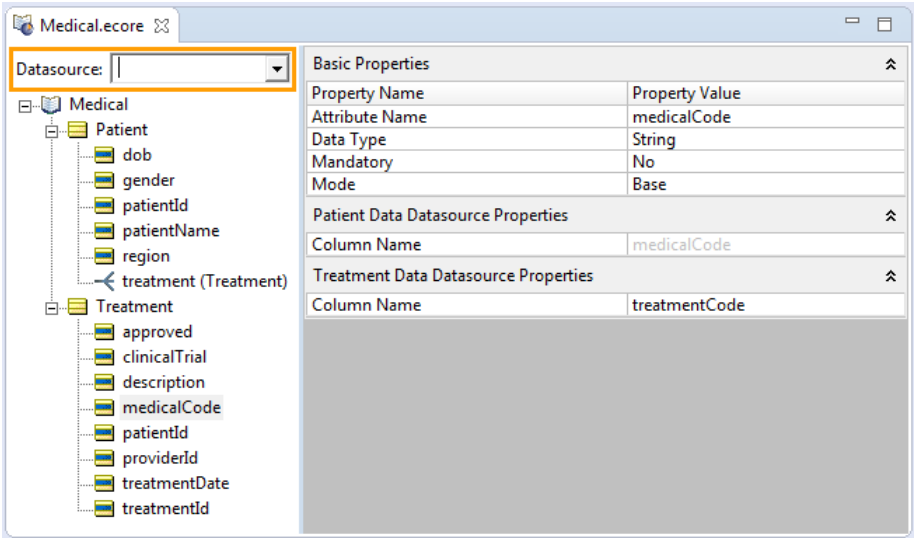


5

How Datasource information is viewed in the Vocabulary

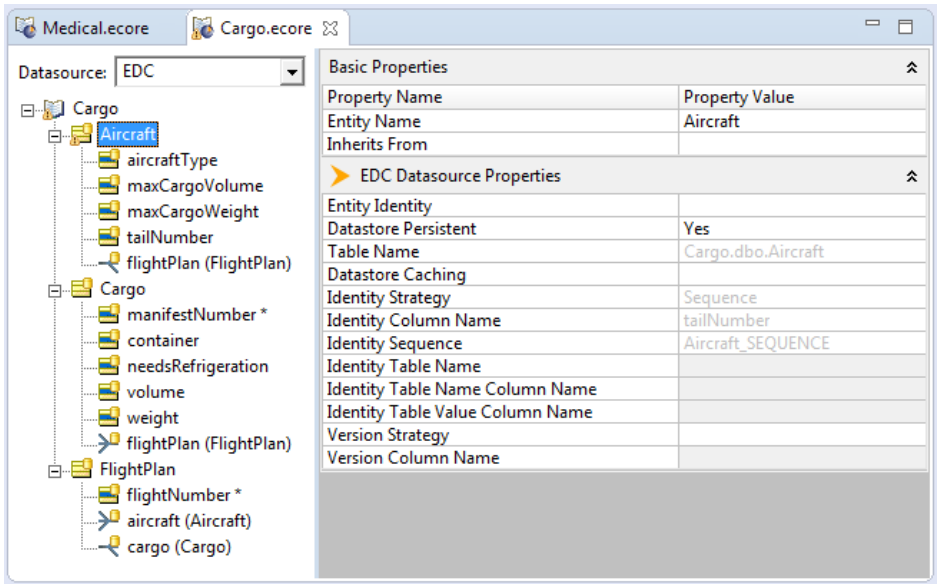
When an EDC, ADC or REST Datasource is added to a vocabulary, the Vocabulary editor is modified to place a **Datasource** pulldown menu above the Vocabulary tree, as illustrated:

Figure 1: Datasource not selected



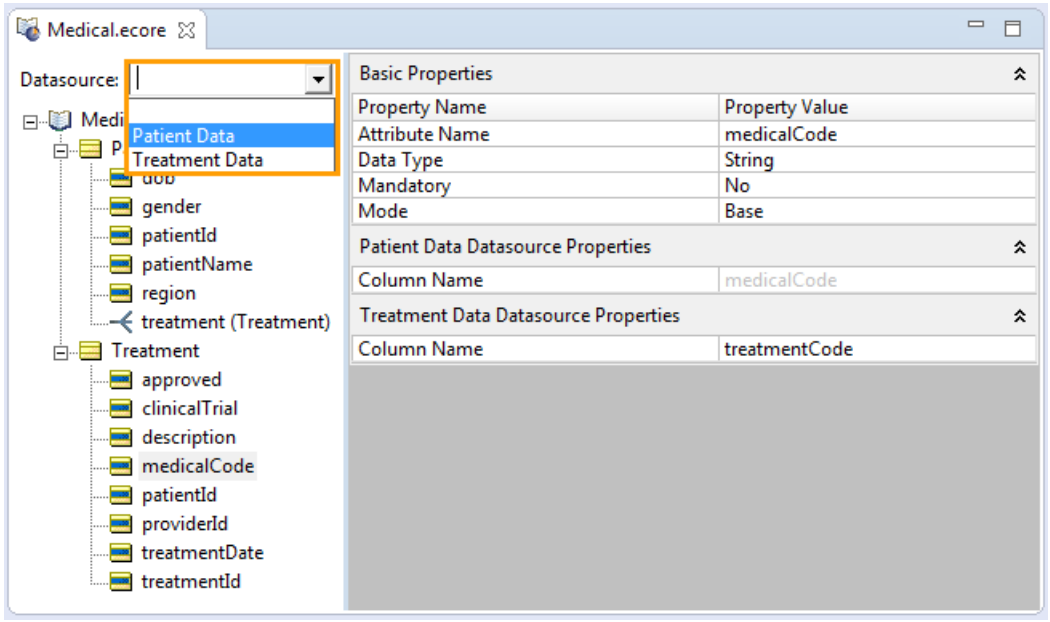
Once a Datasource is selected, the tree icons take on database 'decorations' on each persisted entity and attribute. The list of attributes in each persisted entity is re-arranged such that the one or more attributes that comprise the entity identity, the Primary Key, are at the top of each list. Here, the **EDC** Datasource is selected and the database decorations is shown:

Figure 2: EDC Datasource selected



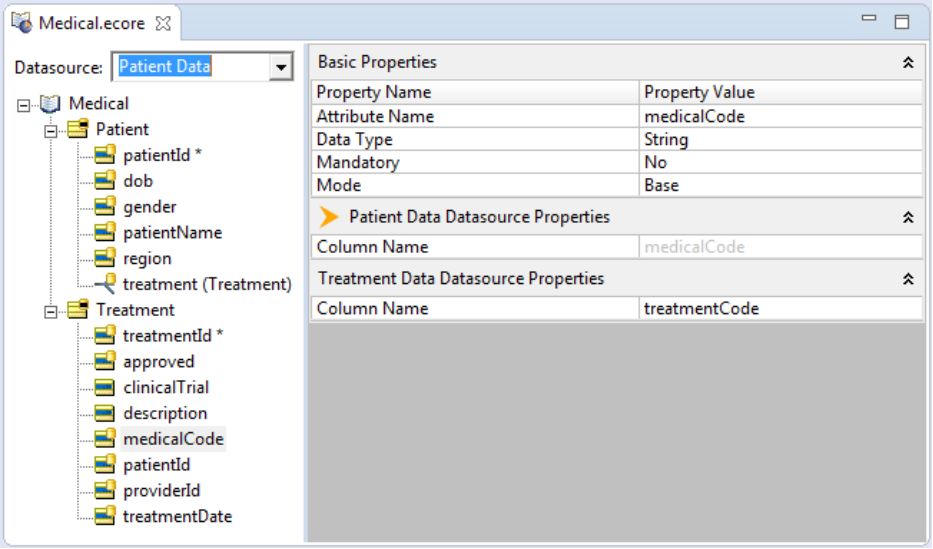
When more than one Datasource has been defined in a Vocabulary, clicking the pulldown lets you choose which Datasource you want to view, as illustrated:

Figure 3: Selecting from multiple Datasources



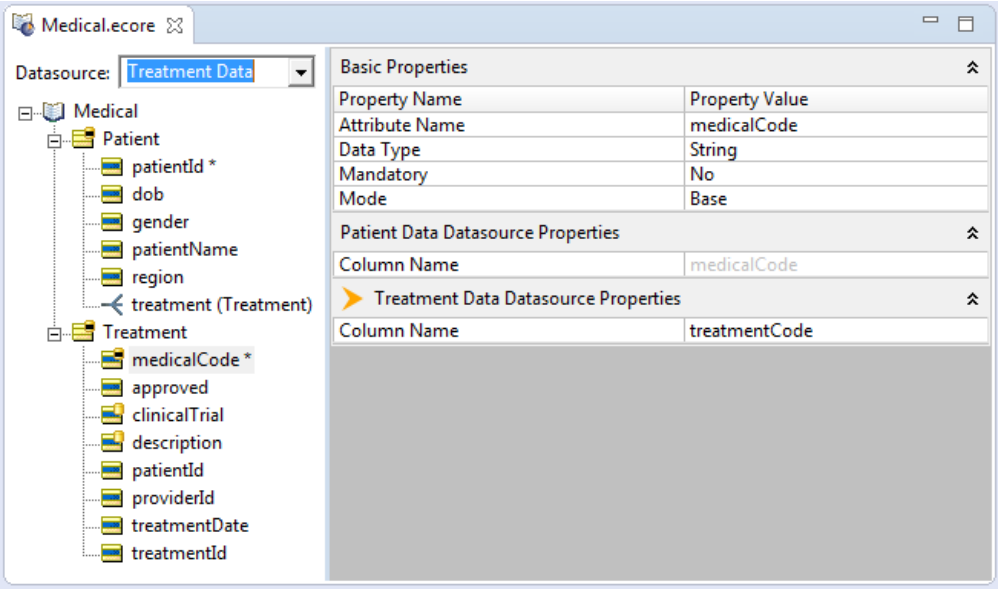
In the following illustration, the Patient Datasource is showing its persisted elements and keys, and the Datasource's section of the Properties panel is decorated with an orange arrow to indicate that it is the Datasource in the current view:

Figure 4: Patient Datasource keys and persistent elements



When you toggle the Datasource selector to the Treatment Datasource, it decorates its persisted elements and keys, and the Datasource's section of the Properties panel is decorated with an orange arrow to indicate that it is the Datasource in the current view:

Figure 5: Treatment Datasource keys and persistent elements



The next sections of this document take you through setting up and experiencing each type of data connectivity. That is followed by advanced material for each type of Datasource.

6

Getting Started with EDC

In this section, you walk through how an Enterprise Data Connector (EDC) connection is established, and then used and tested by rules. The EDC connection enables Corticon Decision Services to connect to a single database and perform read, write, and delete operations on it.

Some simple Vocabulary designs can take advantage of the EDC technique (if accepted by database administrators) that lets Studio export schema information directly to a database engine and generate the necessary table structure within an appropriately defined tablespace.

To load the EDC sample:

In Corticon Studio, choose the menu item **Help > Samples**. Select the Intermediate Sample **EDC Database Connectivity**, and then click **Done**. Follow the **Import** dialog to bring the sample into your workspace.

The topics guide you through experiencing this section by running the sample files in Corticon Studio.

For details, see the following topics:

- Define a table namespace in the database
- Define the database connection for EDC
- Set the entities to store in the database
- Load the schema and data in the database
- Importing database metadata into an EDC Vocabulary
- Test the rules when reading from the database
- Test the rules when writing to the database

Define a table namespace in the database

Note: Using the sample: The sample uses the namespace **PatientRecords**. If you completed Getting Started with ADC on page 37 or Getting Started with Multiple Database Connectivity on page 53, you can just continue with their namespaces.

Set up your database product in a network-accessible location, and then define a database name. Note the database URL and port as well as the new database name. Be sure that your database will not deny connection using credentials -- for example, using `SQL Server Authentication` and not `Windows Authentication`. These parameters are all that is typically required to connect the Vocabulary to the database, create the schema for the persistent entities, and then bring the database metadata back to the Vocabulary.

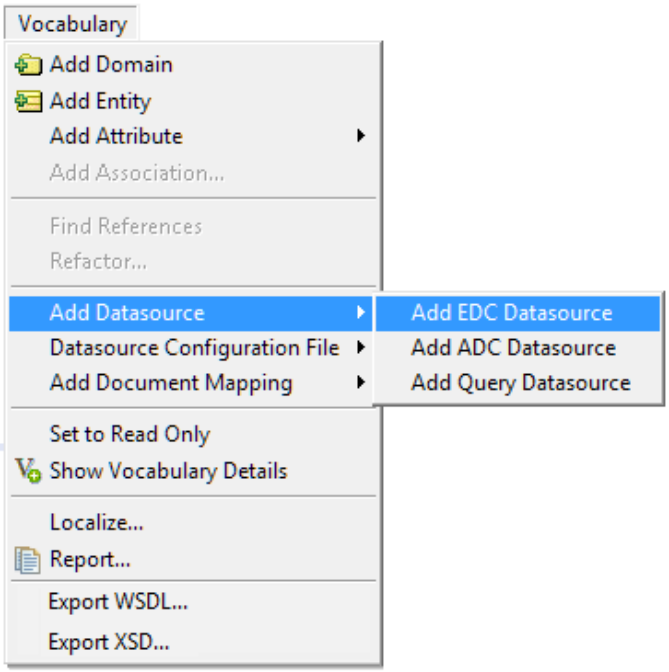
Note: Refer to the Progress Software web page Progress Corticon 6.0 - Supported Platforms Matrix to review the currently supported database brands and versions.

Define the database connection for EDC

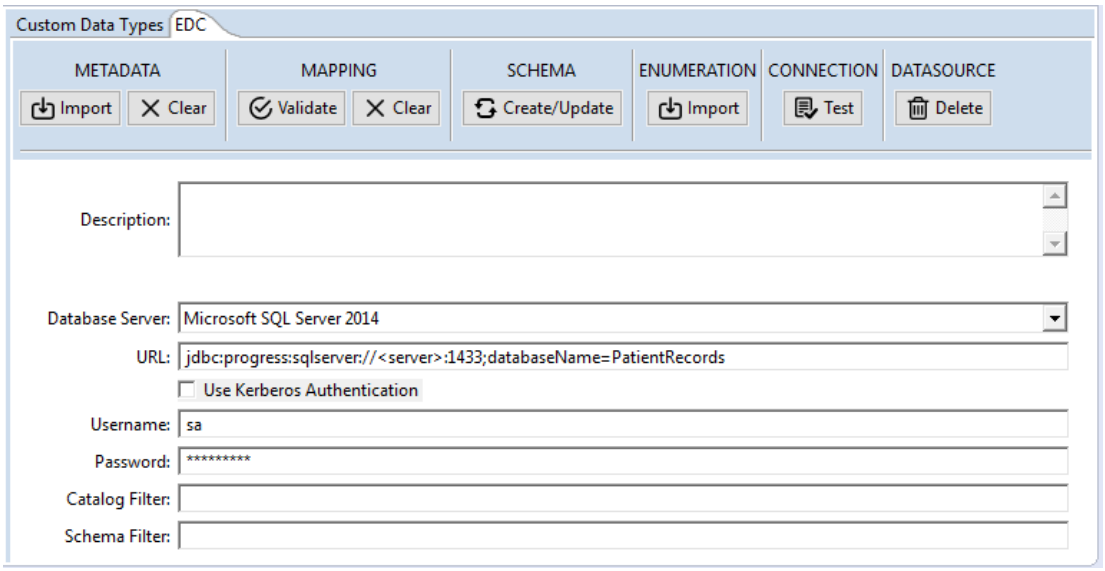
Note: Using the sample: The EDC database connection is defined in the Vocabulary. Enter your username and password, and then test the connection.

To connect a Vocabulary to a database:

1. Choose **Add EDC Datasource** as shown:



2. The **EDC** tab is added to the root level of the Vocabulary, as illustrated:



where:

- **Description**: An informative description of the intended use for the database you are accessing.
- **Database Server**: The database product. Click the dropdown menu on the right side of the entry area to list the available database brands. Corticon embeds Progress DataDirect JDBC Drivers for each database. These drivers provide robust, configurable, high-availability functionality to RDBMS brands. The drivers are pre-configured and do not require performance tuning.
- **Database URL**: The preconfigured URL for the selected database server. You must edit the default entry to replace (1) `<server>` with the machine's DNS-resolvable hostname or IP address and port , and (2) `<database name>` with the database name that was set up (typically case-sensitive).
- **Use Kerberos Authentication**: When selected, indicates that the user will use the Kerberos authentication protocol. The Username and Password are greyed as they are not relevant for the Kerberos protocol:

- **Username**: The user credentials that enable connection to the database. The credentials are encrypted when the database access file is exported for deployment.
- **Password**: The specified user's password.
- **Catalog Filter and Schema Filter**: Patterns that refine the metadata that is imported during **Import Database Metadata** and **Create/Update Database Schema**.

Note: Filters are a good idea for production databases that might have hundreds or even thousands of schemas. As the Catalog filter value does not support wildcards, distinguishing two metadata import filters enables the use of wildcards in the Schema filter value: Underscore (_) provides a pattern match for a single character. Percent sign (`%`) provides a pattern match for multiple characters (similar to the SQL `LIKE` clause). For example, you could restrict the filter to only schemas that start with `DATA` by specifying: `DATA%`. The ability to specify patterns is especially valuable when testing performance on RDBMS brands with applications that use multiple schemas.

- **Additional properties**: Extended properties are not typically needed on an EDC connection. For more about these properties, see Setting additional EDC Datasource connection properties on page 126.

3. Click the CONNECTION **Test** button. An alert indicates success.

Importing Datasource and Database Access configurations

You might be given a Datasource XML configuration file that defines the EDC connection as well as its security credentials. In that case, do the following:

1. On the project's Vocabulary menu, select **Datasource Configuration File > Import** .
2. Browse to locate and select the `.xml` configuration file to apply.
3. When you click **OK**, the EDC Datasource definition is added to your Vocabulary. If you already have an EDC Datasource defined, the EDC definition will not be overwritten.

If you have an older Database Access Properties file, you can import it as follows:

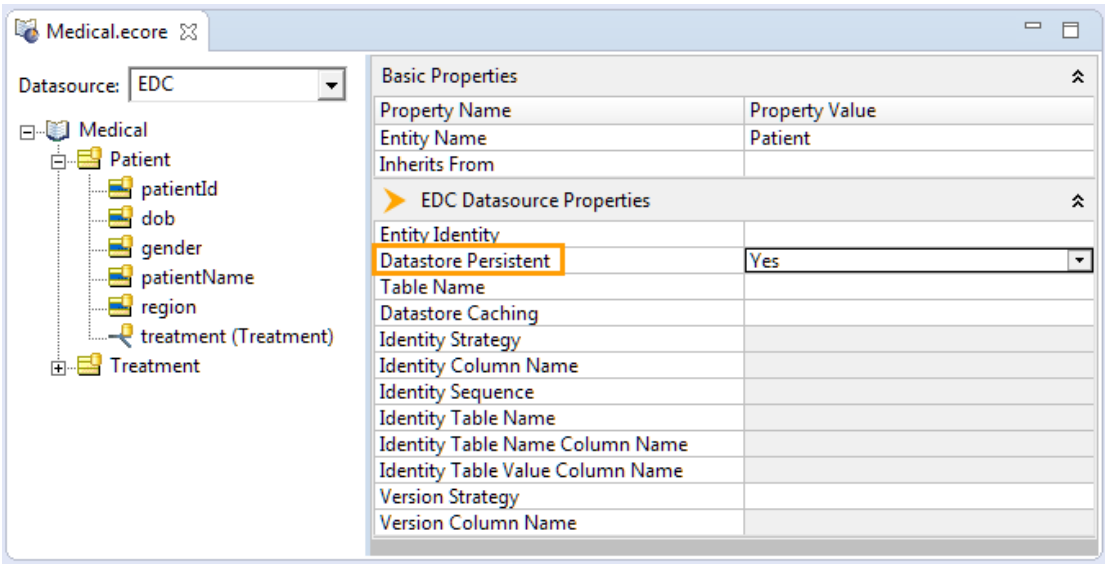
1. On the project's Vocabulary menu, select **Datasource Configuration File > Import Database Access Properties**.
2. Browse to locate and select the `.properties` configuration file to apply.
3. When you click **OK**, the EDC Datasource definition is added to your Vocabulary. If you already have an EDC Datasource defined, the EDC definition will not be overwritten.

Set the entities to store in the database

Note: Using the sample: The property values in this topic are preset as described in the sample.

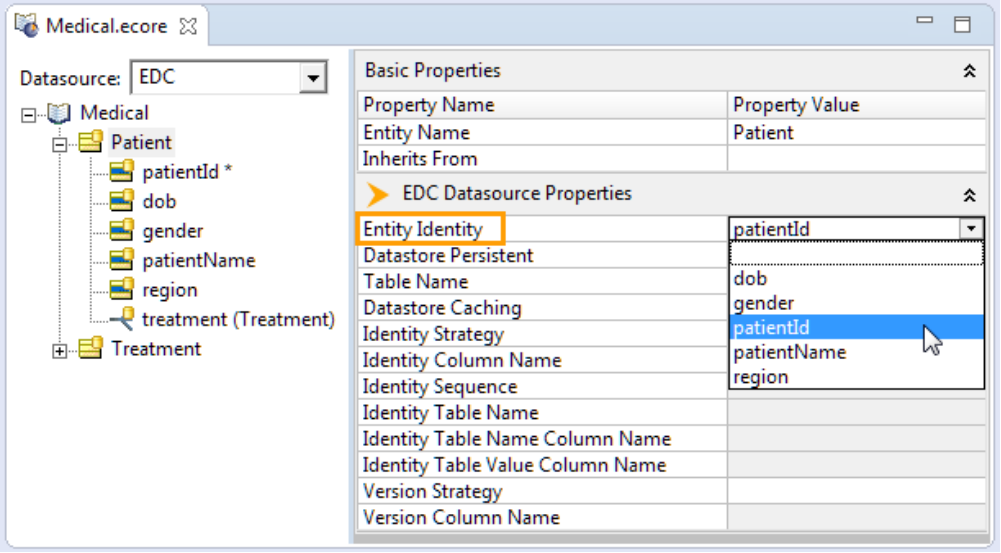
In the **EDC Datasource Properties** section of the Vocabulary, each entity that will be mapped to a database needs to be declared, and then specify its entity identity that will set the Primary Key.

1. In the Vocabulary editor, click on each entity that will be included in the database schema, to do the following:
   a. Set its **Datastore Persistent** property to **Yes**, as shown:



The entity and all its attributes now display their icon with a database decoration.

b. Set its **Entity Identity** - While you might consider alternative Identity strategies on page 147, typically Application Identity is used to assign an attribute as the primary key for the entity. Click the **Entity Identity** Property Value to open its menu, and then choose from the listed attributes, as shown:



The selected attribute is now at the top of the listed attributes and marked with an asterisk (*). That's the primary key (PK) in the database table.

You can specify a key that is a compound of two attributes by holding down the `Ctrl` key while clicking the attributes you want in the key.

Note: Setting the Entity Identity but leaving Datastore Persistent set to No has no effect.

Load the schema and data in the database

Create the schema in the database

A Corticon EDC Datasource connection enables you to push the schema into the database. For a basic use case, this a great timesaver.

It is more often the case that a database administrator would create the schema for the persistent entities as tables and columns in the database. For the EDC sample, you need to have executed the Corticon SQL script `patient` for your database in your database management tool's editor.

Once the database has been setup for the Vocabulary, you need to import the metadata into Corticon Studio to complete the binding. In Corticon Studio on the Vocabulary's **EDC** tab, click METADATA: **Import**. The mapping metadata from the database is added into the Vocabulary for the Entities (tables), Attributes (columns), and Associations (join expressions). For more about mapping and possible anomalies, see Mapping and validating EDC database metadata on page 121.

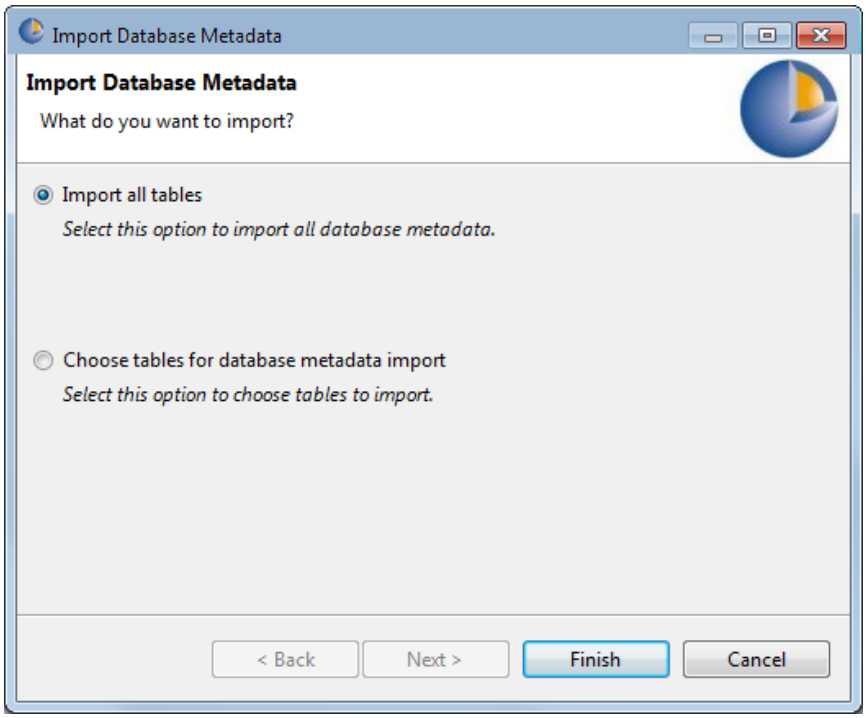
Importing database metadata into an EDC Vocabulary

When the database schema exists, its metadata can be imported into Corticon Studio to refine and complete the mappings between the Vocabulary and the metadata.

You can control the tables that are accessed to transfer metadata to the Vocabulary. When only a small subset of tables will supply the metadata that is needed, the time and space overhead of the process is reduced by delimiting the tables.

In the Vocabulary editor with the EDC database connection established, select the Vocabulary root, and then select the tab of the database connection metadata you want to import. In its panel, click METADATA **Import**.

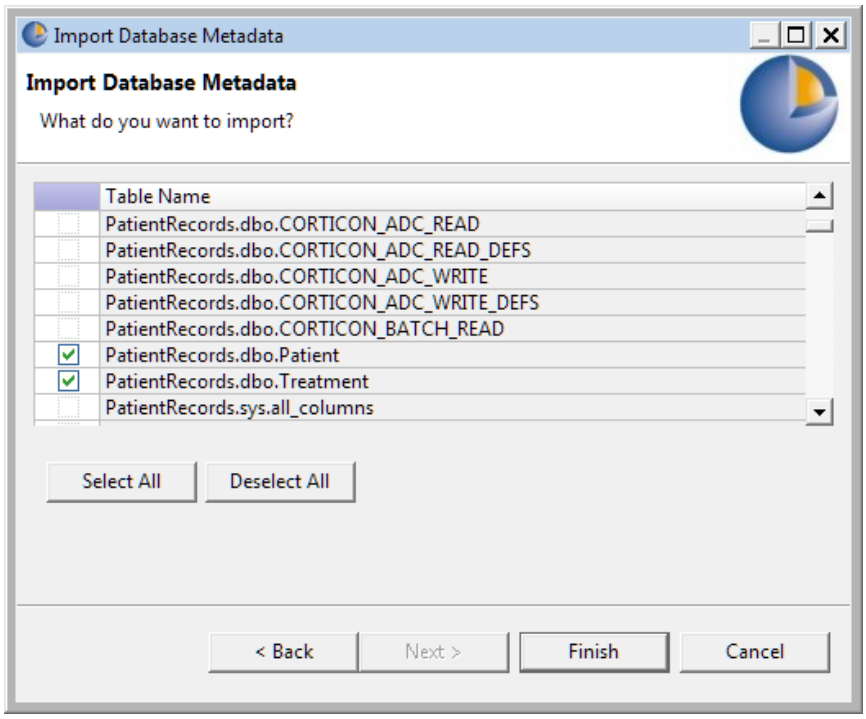
In the dialog, accept **Import all tables**, and click **Finish**, or...



... or click **Choose tables for database metadata import** and click **Next**.

The panel lists all the tables in the connected database.

If you do not want all, click **Deselect All**, and then choose specific tables.

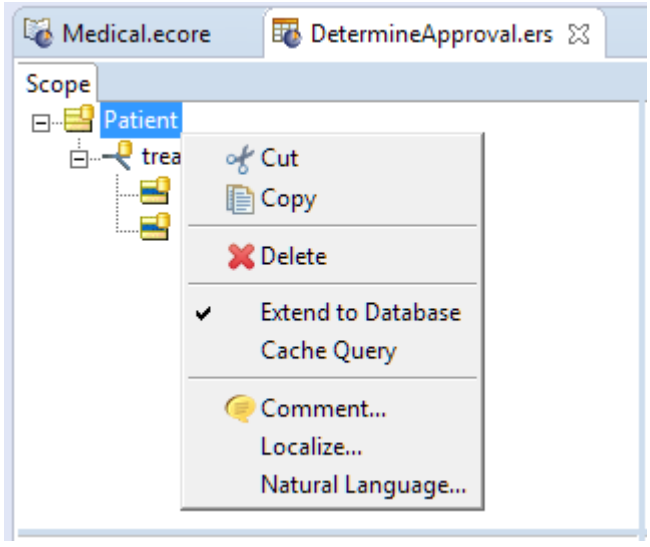


In this example, just two tables are selected. Click **Finish** to perform the task.

As database metadata is imported into a Vocabulary, the Vocabulary Editor's automatic mapping feature attempts to find the *smart match* for each piece of metadata. An entity will be auto-mapped to a table if the two names are spelled the same way, regardless of case.

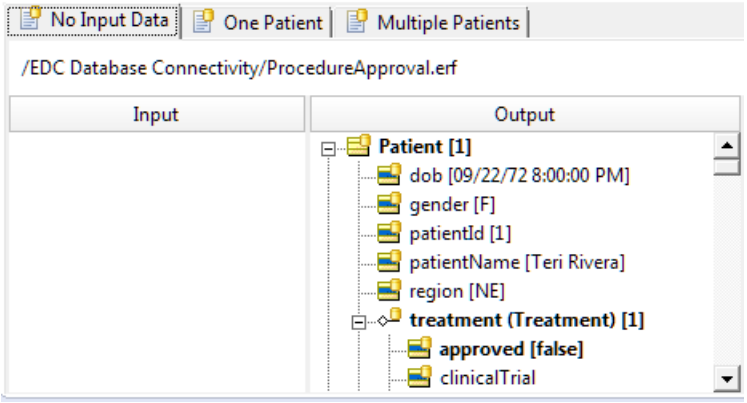
Test the rules when reading from the database

In the EDC sample's Rulesheet, `DetermineApproval.ers`, its **Advanced** view's **Scope** shows that the `Patient` entity on the Rulesheet is set to **Extend to Database**, as shown:

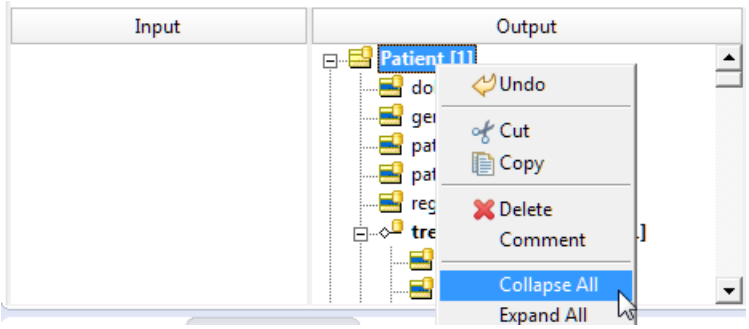


This setting tells the rules to get all database records that relate to the query. As the rules do not filter or aggregate data, the results will be more than you might expect.

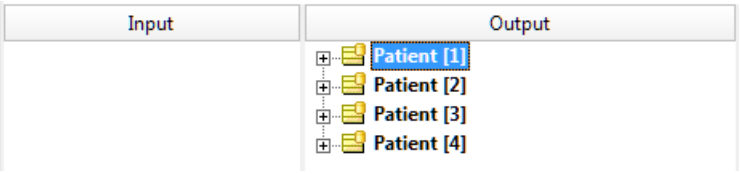
Open the sample's Ruletest, `ProcedureApproval.ert`. It opens on the Testsheet **No Input Data**. Click **Run** to compile and run the rules. The output shows patient and treatment data.



Right-click anywhere in the output column, and then choose **Collapse All**, as shown:



Now we can see that all the patient records and their treatments were retrieved.



So too for the **One Patient** and **Multiple Patients** Testsheets. That is not what we want in this use case.

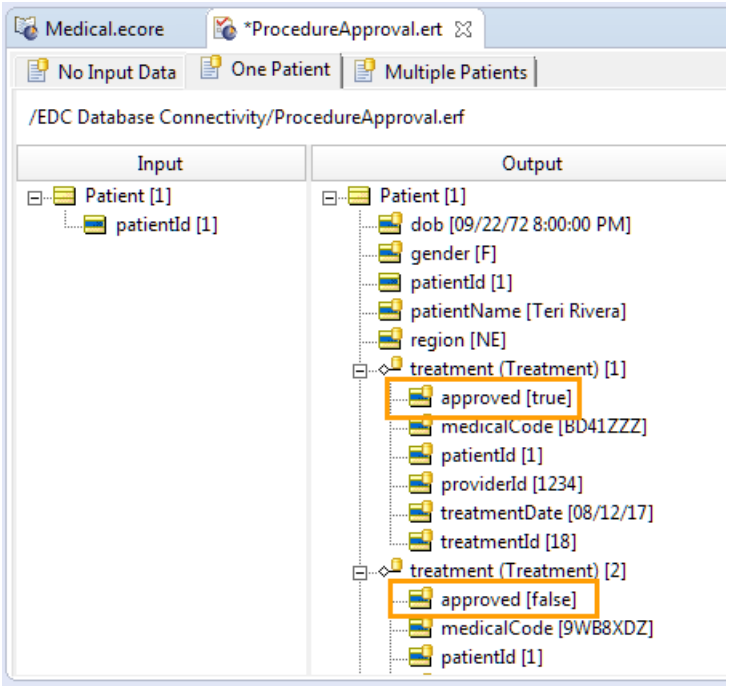
Note: Extending to database can produce a variety of useful results. For examples see How data from an EDC Datasource integrates into rule output on page 128

Run test without extending to database

In the Rulesheet's **Scope**, right-click on `Patient`, and then clear the option to **Extend to Database**. Then, return to the Ruletest. When you run the test for the **No Input Data** testsheet, you get no results. When you run the Testsheet for the **One Patient** and **Multiple Patients** Testsheets, you get exactly that one patient and the specified three patients..

Test the rules when writing to the database

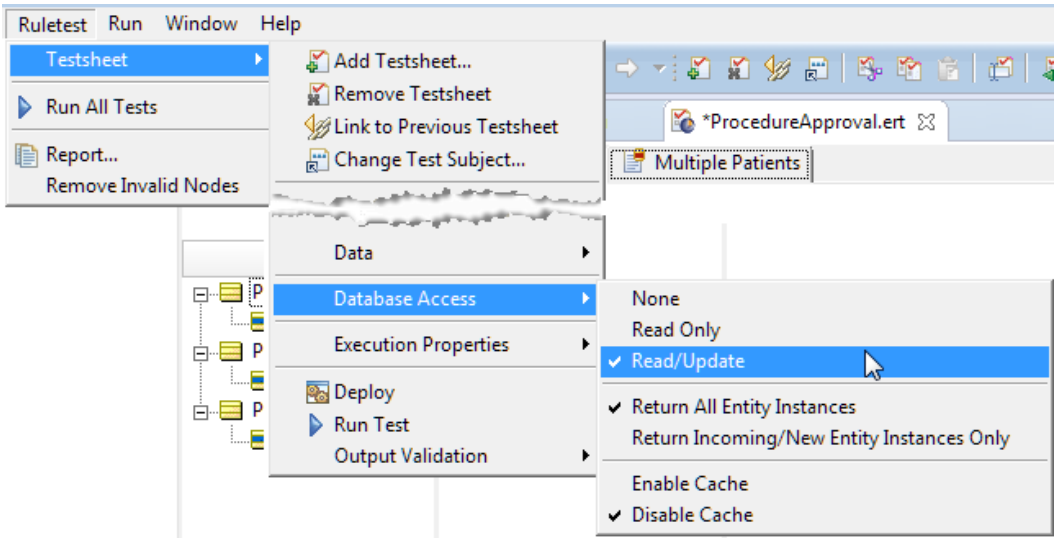
When the rules ran, they determined the approval of treatment, as shown:



That information might have been adequate in the response. If the intent was to also persist the approval, it is not being entered into the database, as shown:

treatmentId	approved	medicalCode	providerId	treatmentDate	patientId
1	*NULL*	9WB8XDZ	1234	2017-07-15	1
2	*NULL*	BL30Y0Z	5678	2017-08-01	4
3	*NULL*	9WB8XKZ	5678	2017-03-12	2
4	*NULL*	F09Z1ZZ	1234	2017-07-01	6
5	*NULL*	0313090	4321	2017-09-05	7
6	*NULL*	F09Z0KZ	1234	2017-09-28	2
7	*NULL*	BD41ZZZ	1234	2017-08-03	3
8	*NULL*	BL30ZZZ	4321	2017-06-12	1
9	*NULL*	B512ZZZ	8765	2017-10-30	8
10	*NULL*	F09Z0KZ	4321	2017-10-04	7

To persist to the database, choose the Ruletest's Testsheet **Multiple Patients**, and then choose the menu command **Ruletest > Testsheet > Database Access** to choose **Read/Update**, as shown:



Notice the red bar that decorates the database corner of the **Multiple Patients** tab:

That shows that it is in Read/Update mode. Now, click **Run** to compile and execute the rules. The output shows patient and treatment data for just the three specified patients. And the database shows that records were updated with the `Approved` status for only those three patients, as shown:

treatmentId	approved	medicalCode	providerId	treatmentDate	patientId
1	False	9WB8XDZ	1234	2017-07-15	1
2	*NULL*	BL30Y0Z	5678	2017-08-01	4
3	False	9WB8XKZ	5678	2017-03-12	2
4	*NULL*	F09Z1ZZ	1234	2017-07-01	6
5	*NULL*	0313090	4321	2017-09-05	7
6	False	F09Z0KZ	1234	2017-09-28	2
7	True	BD41ZZZ	1234	2017-08-03	3
8	True	BL30ZZZ	4321	2017-06-12	1
9	*NULL*	B512ZZZ	8765	2017-10-30	8
10	*NULL*	F09Z0KZ	4321	2017-10-04	7

That's the basics of Corticon's Enterprise Data Connector. Now you can get A closer look at how Corticon relates to Datasources on page 99 and the Advanced EDC Topics on page 107, or proceed to Getting Started with ADC on page 37.

7

Getting Started with ADC

Corticon's Advanced Data Connector (ADC) provides an alternative to Corticon's Enterprise Data Connector (EDC) for accessing database data. It provides greater control over the query and insert statements that are used. This is beneficial when you need finer control for performance or need to retrieve large amounts of data, as is the case in batch processing. With ADC you define a mapping of your vocabulary to a database, define queries, and control when queries are performed to retrieve data.

To load the ADC sample:

In Corticon Studio, choose the menu item **Help > Samples**. Select the Advanced Sample **ADC Database Connectivity**, and then click **Done**. Follow the **Import** dialog to bring the sample into your workspace.

Note: ADC is different from EDC in a few ways. If you are following along after walking through Getting Started with EDC on page 27, there are a few things to reset to make this section flow smoothly:

- In Corticon Studio, choose **File > Close All**.
- In the database table `Treatments`, either reset all the approved values to *Null*, or just delete and recreate the database

---

See more topics on ADC usage in the section Advanced ADC Topics on page 155

For details, see the following topics:

- Overview of the Advanced Data Connector
- Define a table namespace in the database for ADC
- Create and map the ADC schema and queries
- Enable the project for ADC

- Define a database connection for ADC
- Define and import queries for ADC
- Import ADC Datasource metadata into a Vocabulary
- Use an ADC connection as a Ruleflow service callout
- Test the rules when reading from the ADC database
- Test the rules when writing to the ADC database

Overview of the Advanced Data Connector

ADC functions as a *service callout* that accesses data as a step in a Ruleflow. They are based on Corticon Extensions -- ones that you might create yourself, as described in *the Corticon Extensions Guide* -- that are packaged opaquely for ADC Datasource read/write functions through SQL queries stored in the database.

To use ADC:

1. **Map your vocabulary to a database** - In the Corticon vocabulary editor, map the entities, attributes, and associations that tell ADC how to construct entities and associations for data retrieved from the database and how to save data when storing to the database.
2. **Define parameterized SQL statements for the queries** - You have full control over these queries. They can be parameterized so that substitutions can be performed at runtime. To make these statements easy to manage, they are also stored in a database--the same database or a database separate from the data to be queried.
3. **Add the ADC callout to a Ruleflow** - In the Corticon Ruleflow editor, when you add a Service Call-out to a Ruleflow, you configure it to identify the queries to be performed by selecting one of the SQL statements you have defined. To make this easier, you can give the SQL statements logical names.

When all steps are completed you are ready to deploy your Ruleflow or test it in the Corticon tester. When ADC runs, it performs substitutions into the statement to access data. For queries, ADC constructs entities, sets attributes, and defines associations using the Vocabulary mapping. For inserts, ADC uses the mapping data for storing to the database.

You can use multiple instance of ADC in a Ruleflow. A typical use case would be to have an instance at the start of a Ruleflow to retrieve data and one later in the Ruleflow to save data.

Define a table namespace in the database for ADC

Note: The sample uses the namespace **PatientRecords**. If you completed Getting Started with EDC on page 27 or Getting Started with Multiple Database Connectivity on page 53, you can just continue with their namespaces.

Set up your database product in a network-accessible location, and then define a database name. Note the database URL and port as well as the new database name. Be sure that your database will not deny connection using credentials -- for example, using `SQL Server Authentication` and not `Windows Authentication`. These parameters are all that is typically required to connect the Vocabulary to the database, create the schema for the persistent entities, and then bring the database metadata back to the Vocabulary.

Note: Refer to the Progress Software web page Progress Corticon 6.0 - Supported Platforms Matrix to review the currently supported database brands and versions.

Create and map the ADC schema and queries

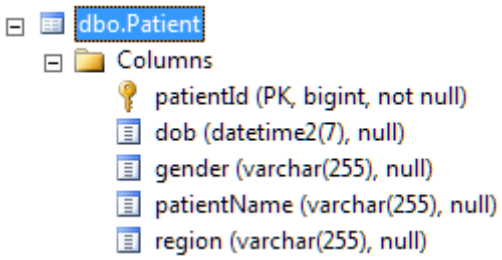
Note: Using the sample: For the ADC sample, you need to have executed the Corticon SQL scripts `patient` and `adc` for your database in your database management tool's editor. The sample's metadata, primary keys, and join expressions are already mapped to the database.

Create the schema in the database

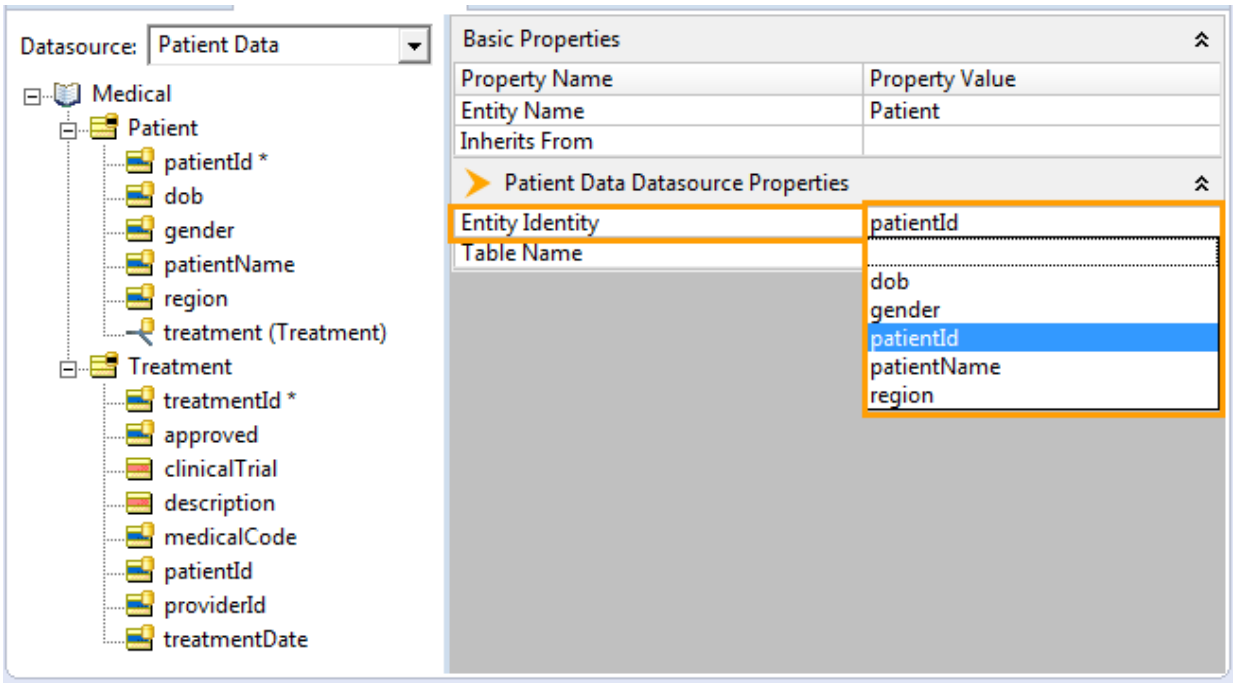
Typically, the database administrator creates the tables in the namespace, and then the columns with their data types, the declared primary key, and any joins between tables.

Create the entities and their identity, then their attributes, and associations

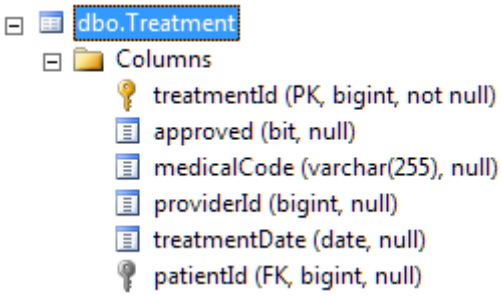
1. Define the database tables that will participate in rules as Corticon entities. For the sample, the first table is `Patient`:



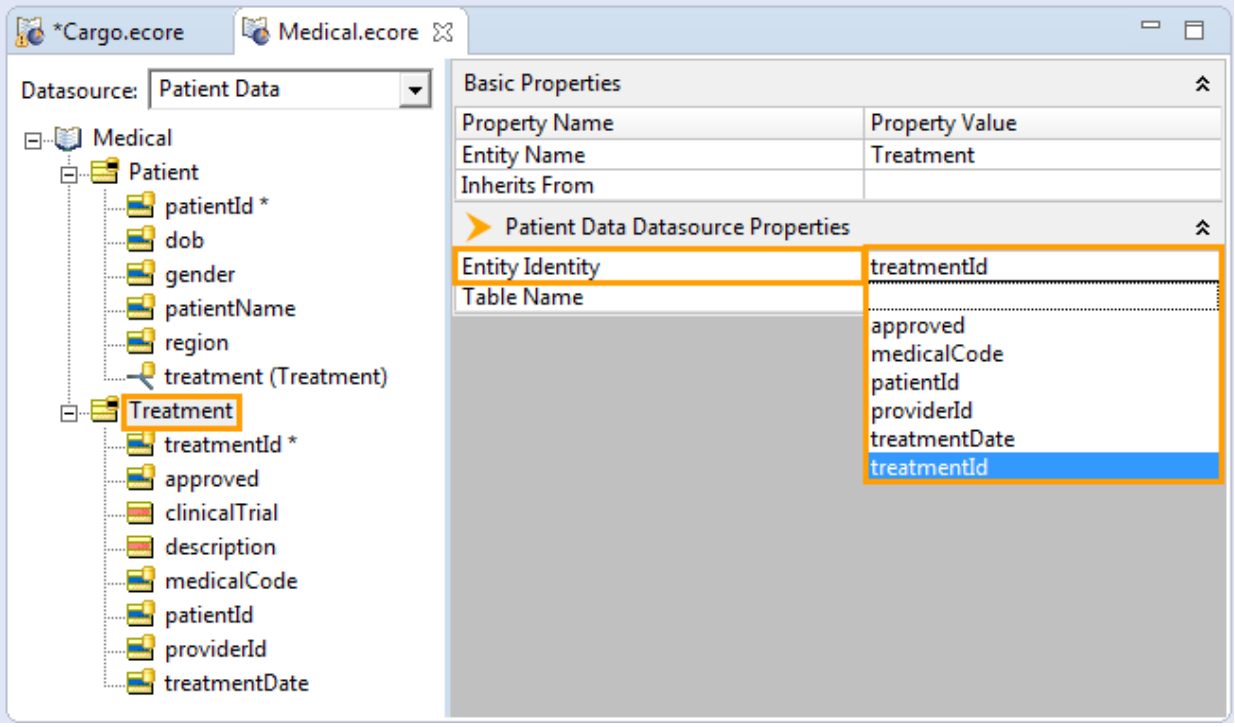
2. Add the required columns for that table as Corticon attributes with corresponding data types.
3. Specify the Primary Key (PK) in the database table as the Entity Identity by clicking the **Entity Identity** Property Value to open its menu, and then choose from the listed attributes, as shown:



4. Add additional tables, such as the sample's `Treatment` table:



5. Specify its Primary Key (PK) in the database table as the Entity Identity by clicking the **Entity Identity** Property Value to open its menu, and then choose from the listed attributes, as shown:



6. In Corticon Studio on the Vocabulary's Datasource tab, click METADATA: **Import**. The mapping metadata from the database is added into the Vocabulary for the Entities (tables), Attributes (columns), and Associations (join expressions). If the imported tables and columns do not align with entities and their attributes, those values will require manual mapping.

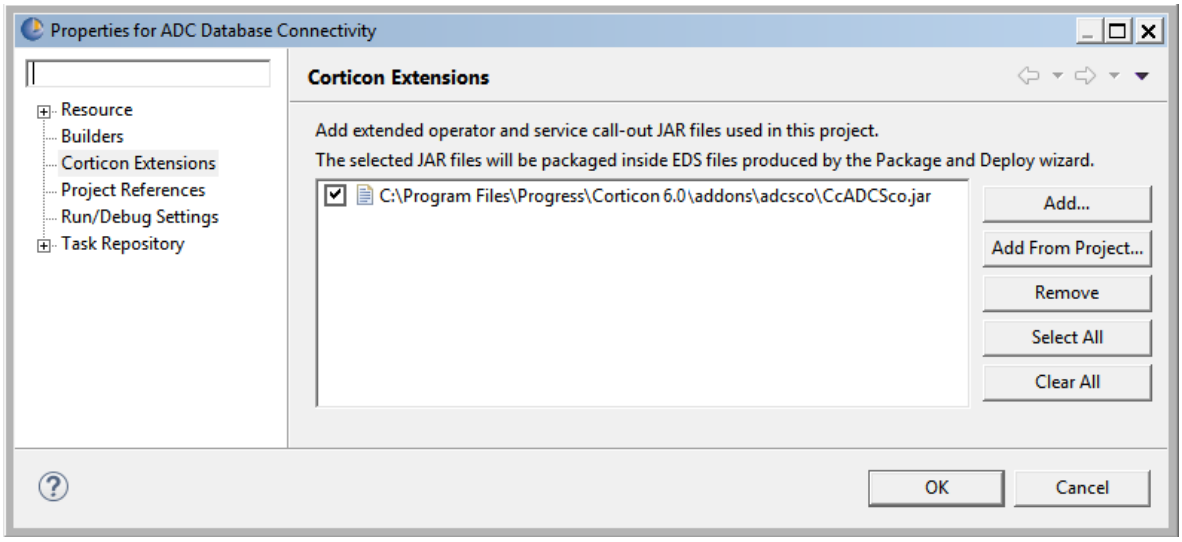
For more about mapping, see Mapping ADC database metadata on page 155.

Enable the project for ADC

Note: The sample has added a link to the required JAR file. This path is valid if your Corticon Studio installation path is the default, as illustrated. Otherwise, remove the default listed and add your path to the JAR file.

Each project that intends to use ADC must add a packaged set of Corticon extensions into its project. On Corticon Studio, do the following:

1. Right-click on the project name in the Project Explorer, and then choose **Properties**.
2. Click **Corticon Extensions**.
3. Click **Add**, and then locate and choose the file `[CORTICON_HOME]\addons\adcsco\CcADCSco.jar`, as illustrated:



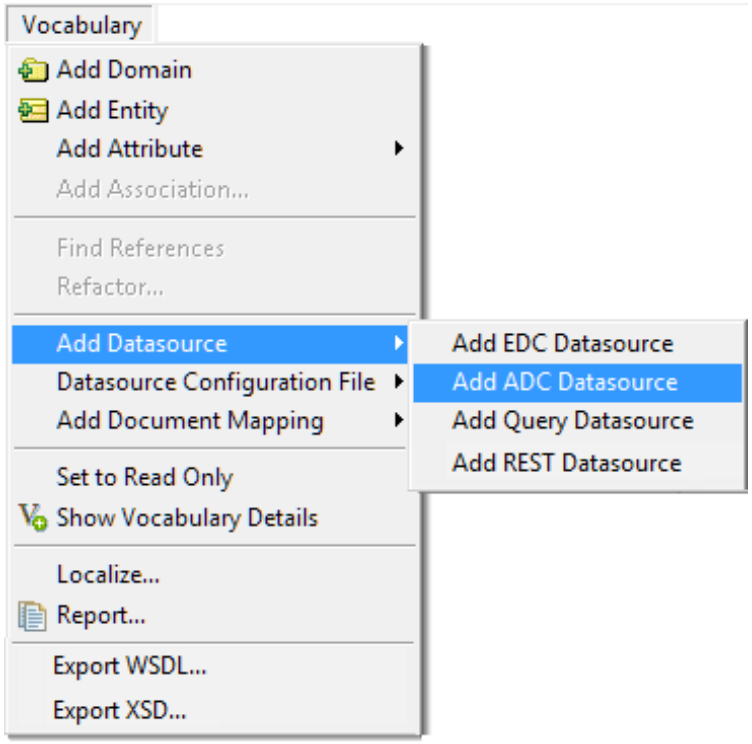
4. Click **OK**.

Note: If you installed Corticon to other than the default location, you'll notice that updating this preset value resolves the error shown in the project.

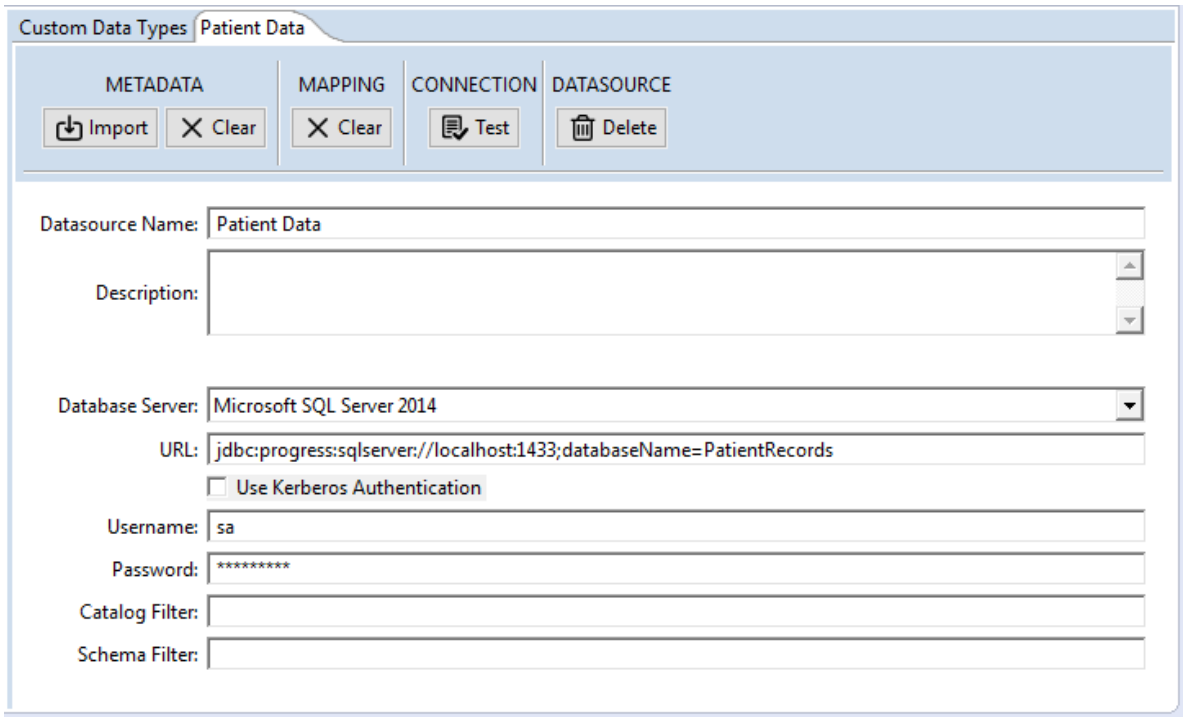
Define a database connection for ADC

Note: Using the sample: The ADC database connection is defined in the Vocabulary. Enter your username and password, and then test the connection.

To connect a Vocabulary to a defined database where you will use SQL queries, define the connection as an ADC Datasource. In the project's Vocabulary editor, select the Vocabulary command **Add Datasource > Add ADC Datasource**, as shown:



An **ADC** tab is added to the root level of the Vocabulary. The ADC sample renamed this Datasource to **Patient Data** which is now the name of its tab, as illustrated:



where:

- **Datasource Name**: The connection name that displays on the Vocabulary tab. This is also its name that will bind this ADC connection to an instance of the service call-out in a Ruleflow. While you can change this

name, it is a good idea to do so before you specify it in Ruleflow Service Call-out's **Service Name**. When you add additional ADC Datasource tabs, the default names will increment `n` in `ADC_n`.

- **Description**: Provide an informative description of the use of the Datasource you are adding.
- **Database Server**: The database product. Click the dropdown menu on the right side of the entry area to list the available database brands. Corticon embeds Progress DataDirect JDBC Drivers for each database. These drivers provide robust, configurable, high-availability functionality to RDBMS brands. The drivers are pre-configured and do not require performance tuning.
- **Database URL**: The preconfigured URL for the selected database server. You must edit the default entry to replace (1) `<server>` with the machine's DNS-resolvable hostname or IP address and port, and (2) `<database name>` with the database name that was set up (typically case-sensitive).
- **Use Kerberos Authentication**: Indicates that the user will use the Kerberos authentication protocol. The Username and Password are greyed as they are not relevant for the Kerberos protocol:

- **Username**: The user credentials that enable connection to the database. The credentials are encrypted when the database access file is exported for deployment.
- **Password**: The specified user's password.
- **Catalog Filter and Schema Filter**: Patterns that refine the metadata that is imported during **Import Database Metadata**.

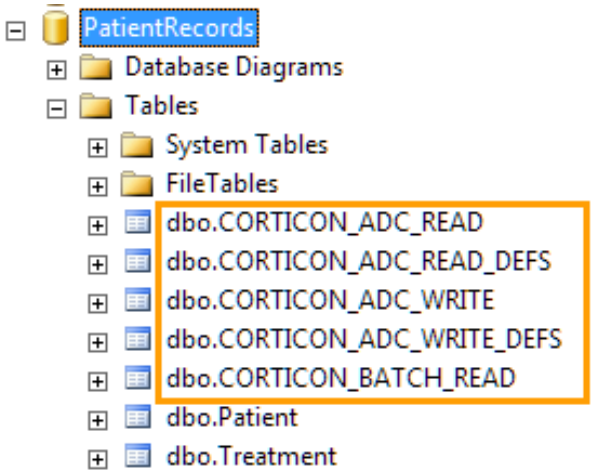
Note: Filters are a good idea for production databases that might have hundreds or even thousands of schemas. As the Catalog filter value does not support wildcards, distinguishing two metadata import filters enables the use of wildcards in the Schema filter value: Underscore (_) provides a pattern match for a single character. Percent sign (`%`) provides a pattern match for multiple characters (similar to the SQL `LIKE` clause). For example, you could restrict the filter to only schemas that start with `DATA` by specifying: `DATA%`. The ability to specify patterns is especially valuable when testing performance on RDBMS brands with applications that use multiple schemas.

Click the CONNECTION **Test** button. Confirm that you have a valid connection before proceeding.

Define and import queries for ADC

Queries are essential to how ADC functions. With just a few queries, a lot of the SQL tasks are minimized as the rules processing handles complex conditions and actions.

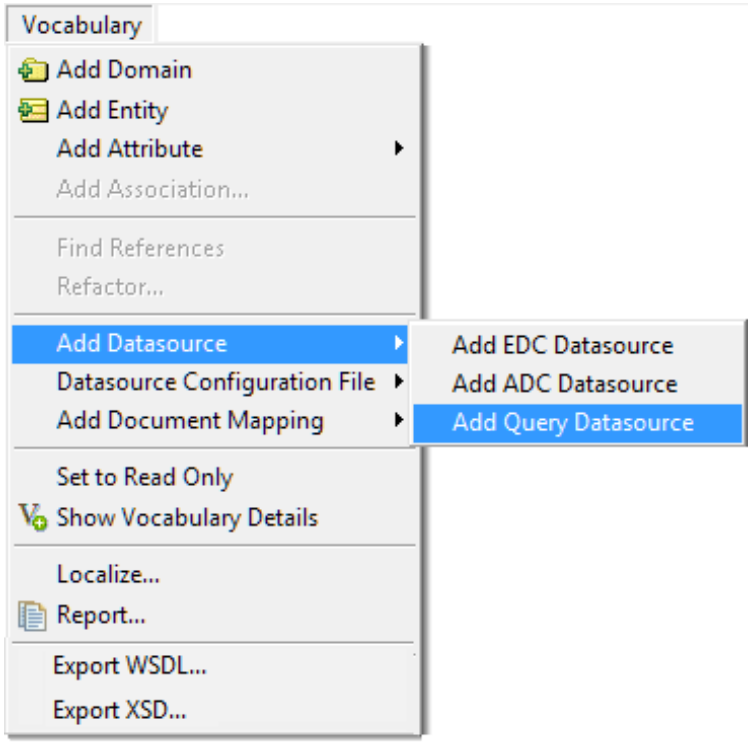
Running the sample's `adc` script created five tables that will be referenced by the query service. `ADC_READ` and `ADC_WRITE` access their `DEF` table to do the steps requested by your Ruleflow Service Call-outs.



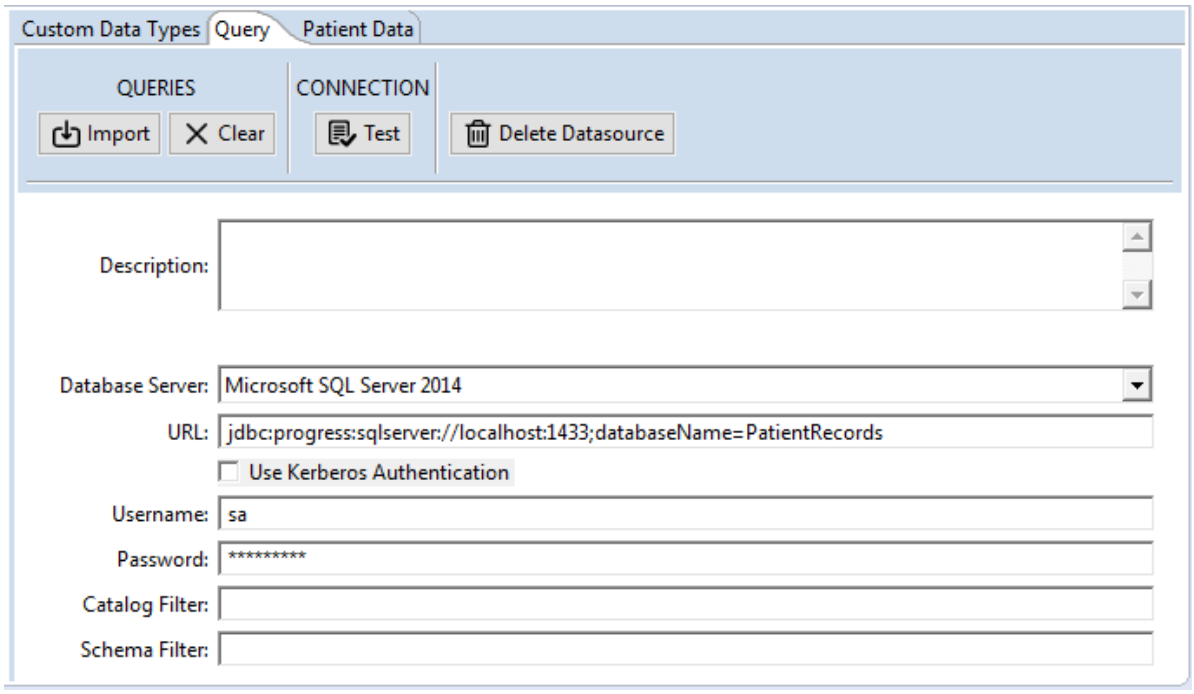
Note: The `BATCH_READ` table inserts the queries you will use in Getting Started with Batch on page 93. For more about the sample's Corticon's queries, see How Corticon is expressed in SQL on page 163.

Define the Query Datasource connection

To connect a Vocabulary to a defined database where you will read the SQL queries, define the connection as a Query Datasource. In the project's Vocabulary editor, select the Vocabulary command **Add Datasource > Add Query Datasource**, as shown:

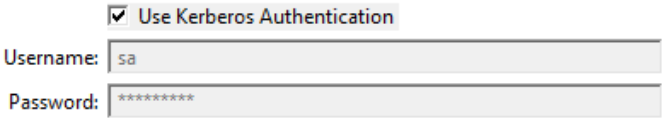


The **Query** tab is added to the root level of the Vocabulary as a fixed-name tab, as illustrated:



where:

- **Description**: An informative description of the Query Datasource.
- **Database Server**: The database product. Click the dropdown menu on the right side of the entry area to list the available database brands. Corticon embeds Progress DataDirect JDBC Drivers for each database. These drivers provide robust, configurable, high-availability functionality to RDBMS brands. The drivers are pre-configured and do not require performance tuning.
- **Database URL**: The preconfigured URL for the selected database server. You must edit the default entry to replace (1) `<server>` with the machine's DNS-resolvable hostname or IP address and port , and (2) `<database name>` with the database name that was set up (typically case-sensitive).
- **Use Kerberos Authentication**: When selected, indicates that the user will use the Kerberos authentication protocol. The Username and Password are greyed as they are not relevant for the Kerberos protocol:

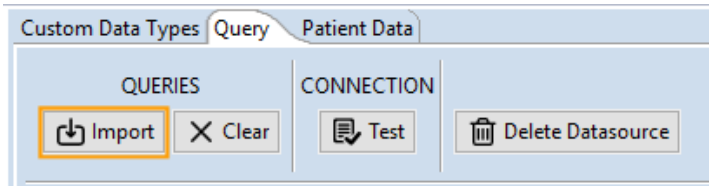


- **Username**: The user credentials that enable connection to the database. The credentials are encrypted when the database access file is exported for deployment.
- **Password**: The specified user's password.
- **Catalog Filter and Schema Filter**: Patterns that refine the metadata that will be imported.

Click the CONNECTION **Test** button. Confirm that you have a valid connection before proceeding.

Import the queries

To access the queries for use in Decision Services, click the QUERIES **Import** button:



The current query names in the Query Datasource are accessed and brought into the local persistent cache. Whenever query names are revised, you need to re-import the queries. If the query defs referenced by a query name change, the latest defs will be accessed by the query name. .

Import ADC Datasource metadata into a Vocabulary

When the database schema exists, its metadata can be imported into Corticon Studio to refine and complete the mappings between the Vocabulary and the metadata.

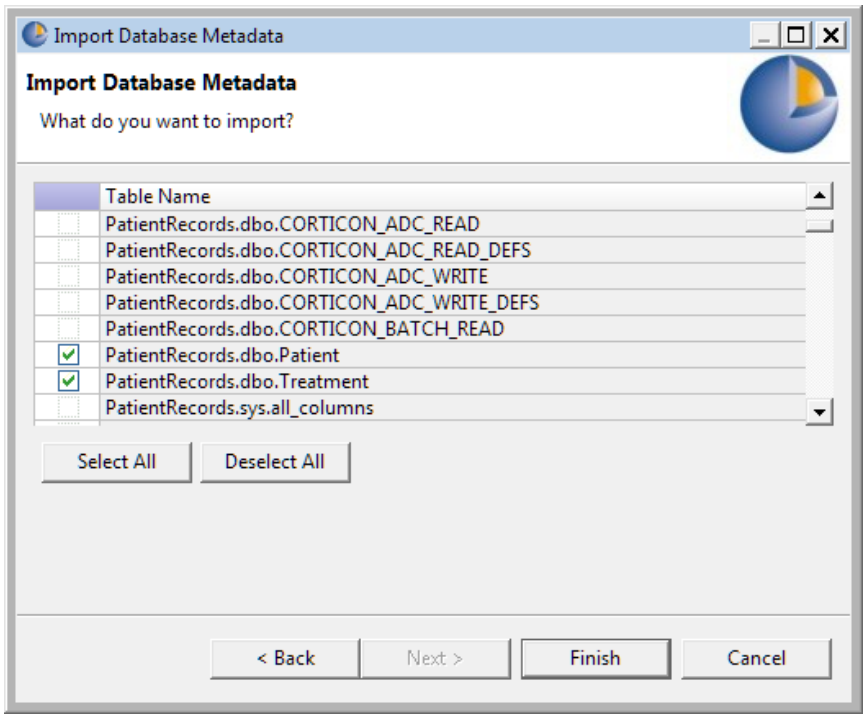
You can control the tables that are accessed to transfer metadata to the Vocabulary. When only a small subset of tables will supply the metadata that is needed, the time and space overhead of the process is reduced by delimiting the tables.

In the Vocabulary editor with the ADC database connection established, select the Vocabulary root, and then select the tab of the database connection metadata you want to import. In its panel, click METADATA **Import**.

In the dialog, accept **Import all tables** or click **Choose tables for database metadata import**, and then click **Next**.

The panel lists all the tables in the connected database.

If you do not want all, click **Deselect All**, and then choose specific tables.



In this example, just two tables are selected. The query tables listed do not have metadata so they can be deselected.

Click **Finish** to perform the task.

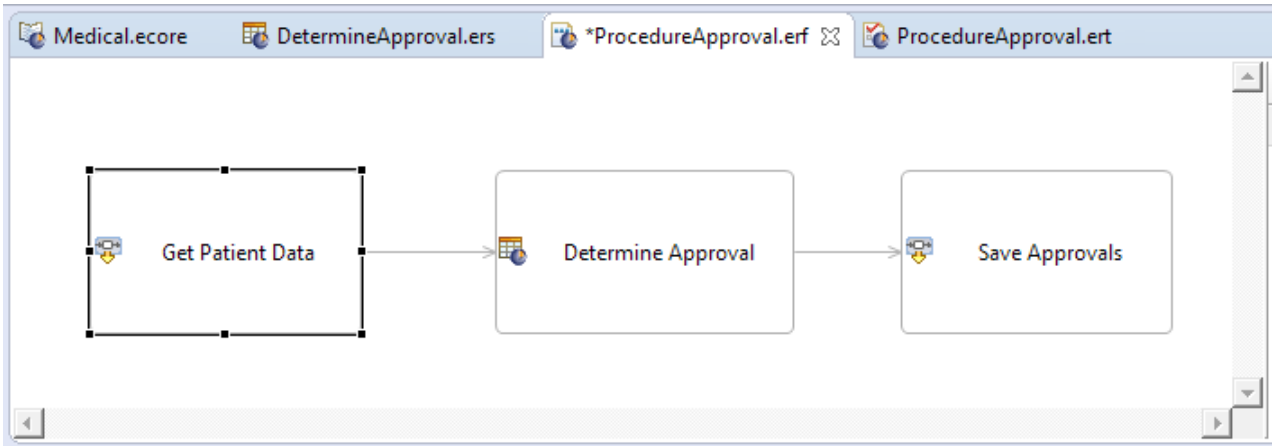
As database metadata is imported into a Vocabulary, the Vocabulary Editor's automatic mapping feature attempts to find the *smart match* for each piece of metadata. An entity will be auto-mapped to a table if the two names are spelled the same way, regardless of case.

Use an ADC connection as a Ruleflow service callout

Note: Using the sample: The Ruleflow objects and their runtime properties are already defined in the sample.

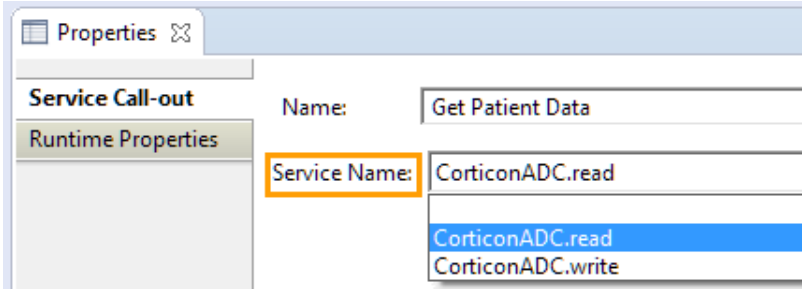
To use an ADC connection in a service callout:

1. In a Ruleflow where you want to use an ADC connection, create a Service Call-out object on the Ruleflow canvas. In this example, the Ruleflow is defined with the project's Rulesheet plus a Service Call-out to read and another one to write back to the database table after rule processing, as shown:

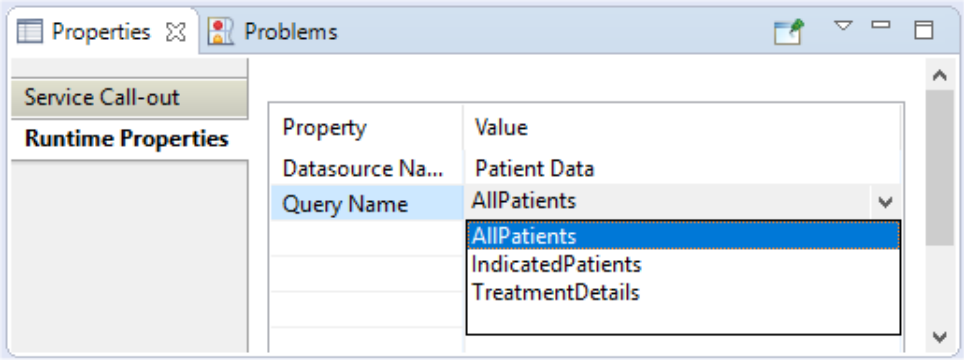


   The ADC sample points out that you can have one ADC connection that has several read and write actions.

2. Click on the **Get Patient Data** object, and then, on the object's **Properties** tab.
3. On its **Service Call-out** tab, click the **Service Name** pulldown to select the service you want for this use, as shown here where the sample uses `CorticonADC.read` for this service, as illustrated:



4. On its **Runtime Properties** tab. Use the **Property** pulldown to:

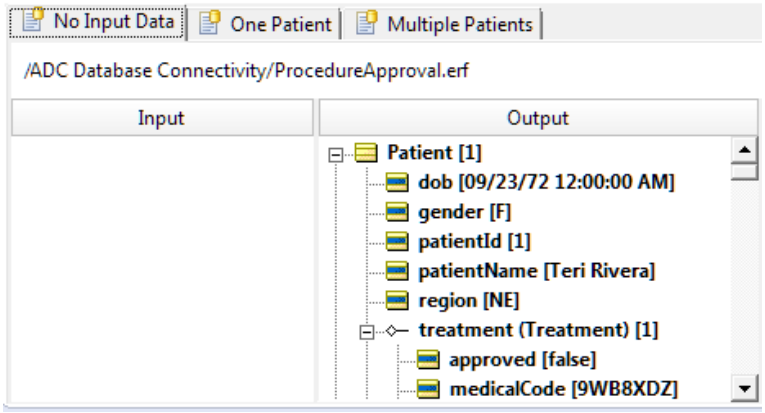


a. Select **Datasource Name**, and then, for its value, use the dropdown menu to select the Datasource that corresponds to the name of the appropriate ADC definition. For the ADC sample, choose **Patient Data**.

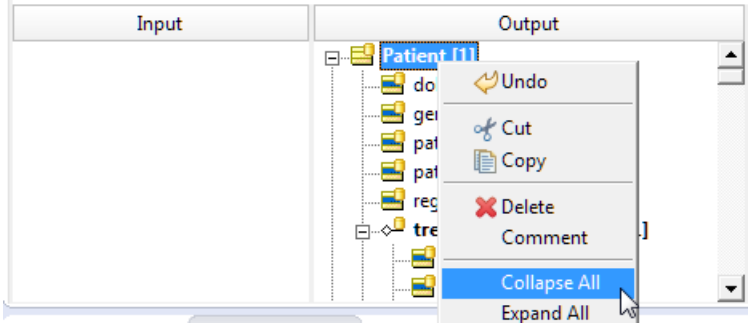
b. Select **Query Name**, and then, for its value, use the dropdown menu to select the appropriate query. For the ADC sample, choose **AllPatients**. The listed values are initially imported and are updated by the **QUERIES: Import** function on the **Query** tab.

Test the rules when reading from the ADC database

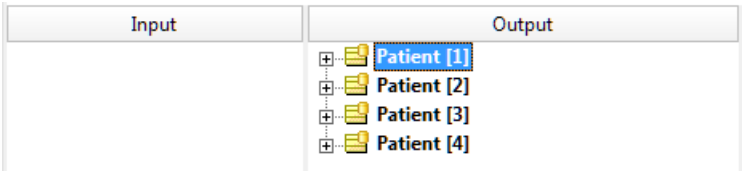
Open the sample's Ruletest, `ProcedureApproval.ert`. It opens on the Testsheet **No Input Data**. Click **Run** to compile and run the rules. The output shows patient and treatment data.



Right-click anywhere in the output column, and then choose **Collapse All**, as shown:

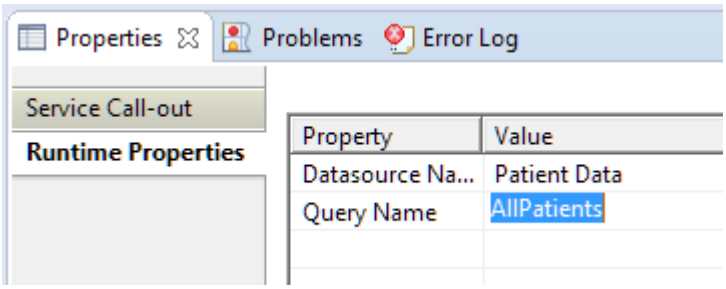


Now we can see that all the patient records and their treatments were retrieved.



So too for the **One Patient** and **Multiple Patients** Testsheets. That is not what we want in this use case.

Looking at the Service Call-out's **Query Name**, we see that it is `AllPatients`:



We got what we asked for!

Time for some SQL

That query was really two queries combined:

```
SELECT * FROM Patient
SELECT * FROM Treatment WHERE patientId IN ({Patient.patientId})
```

Together, the two queries tell ADC to get all the patients and all the treatments for those patients. There are two queries. One query gets the set of Patients, and the other gets the set of Treatments for each patient. All the needed data is retrieved with these two queries and the associations are automatically established in Corticon working memory.

Run test with a different query

Change the Service Call-out's **Query Name** to `IndicatedPatients`. Then, return to the Ruletest. When you run the test for the **No Input Data** testsheet, you get no results. When you run the Testsheet for the **One Patient** and **Multiple Patients** Testsheets, you get exactly that one patient and the specified three patients.

The first part of this SQL statement is:

```
SELECT * FROM Patient WHERE patientId IN ({Patient.patientId})
```

The curly braces indicate tokens in the query that will be replaced by the data passed to Corticon -- in this case, selecting the patients to process.

In other words, `{Patient.patientId}` indicates that one or more attributes defined in the vocabulary can be used in the parameterization of your query. These value for each query parameter can be provided in the request message or set by rules to conditionally fetch data from the database.

Note: For more about the format of Corticon's queries, see How Corticon is expressed in SQL on page 163.

Test the rules when writing to the ADC database

In the database, the approval status is being evaluated but it is not being entered into the database, as shown:

treatmentId	approved	medicalCode	providerId	treatmentDate	patientId
1	*NULL*	9WB8XDZ	1234	2017-07-15	1
2	*NULL*	BL30Y0Z	5678	2017-08-01	4
3	*NULL*	9WB8XKZ	5678	2017-03-12	2
4	*NULL*	F09Z1ZZ	1234	2017-07-01	6
5	*NULL*	0313090	4321	2017-09-05	7
6	*NULL*	F09Z0KZ	1234	2017-09-28	2
7	*NULL*	BD41ZZZ	1234	2017-08-03	3
8	*NULL*	BL30ZZZ	4321	2017-06-12	1
9	*NULL*	B512ZZZ	8765	2017-10-30	8
10	*NULL*	F09Z0KZ	4321	2017-10-04	7

On the Ruleflow canvas, click on the `Save Approvals` Service Call-out on the canvas. Its Query Name is `UpdateTreatment` whose query SQL is:

```
UPDATE Treatment SET approved={Treatment.approved} WHERE
treatmentId={Treatment.treatmentId}
```

When you run the Testsheet, the approval values are written to the database for patient treatments, as shown:

treatmentId	approved	medicalCode	providerId	treatmentDate	patientId
1	False	9WB8XDZ	1234	2017-07-15	1
2	*NULL*	BL30Y0Z	5678	2017-08-01	4
3	False	9WB8XKZ	5678	2017-03-12	2
4	*NULL*	F09Z1ZZ	1234	2017-07-01	6
5	*NULL*	0313090	4321	2017-09-05	7
6	False	F09Z0KZ	1234	2017-09-28	2
7	True	BD41ZZZ	1234	2017-08-03	3
8	True	BL30ZZZ	4321	2017-06-12	1
9	*NULL*	B512ZZZ	8765	2017-10-30	8
10	*NULL*	F09Z0KZ	4321	2017-10-04	7

Where to now?

The flow of this document leads to next Getting Started with Multiple Database Connectivity on page 53, and then Deploying projects that use data integration on page 89, an important preparation for Getting Started with Batch on page 93. Beyond that, the material focuses less on the samples by using various configurations to present advanced topics.

8

Getting Started with Multiple Database Connectivity

Corticon's database connectivity reaches another level when it enables a rules project to access more than one Datasource. You could mix one EDC Datasource with several ADC Datasources, performing Rulesheet-based action and filters while the ADC implementation uses multiple Service Call-outs on a Ruleflow. Together, these enable the Decision Service to be running queries on one database, processing that data, and then possibly branching to write to either of two other databases.

To load the Multiple Database Connectivity sample:

In Corticon Studio, choose the menu item **Help > Samples**. Select the Advanced Sample **Multiple Database Connectivity**, and then click **Done**. Follow the **Import** dialog to bring the sample into your workspace.

The Multiple Database sample expands on the ADC sample's medical treatment approval scenario. Now the patients and the treatments they have received are in one database, while each treatment's description and clinical trial status is in another database. The rules connect to both databases to determine whether a treatment is approved.

Note: If you are just starting to follow the samples hands-on, all the resources for the Corticon Studio and for SQL Server 2014 are included. If you are following along after walking through Getting Started with ADC on page 37, there are a few things to adjust to make this section flow smoothly:

- In Corticon Studio, choose **File > Close All**.
- Execute the Corticon SQL script `cms` for your database in your database management tool's editor.

For details, see the following topics:

- Define multiple table namespaces

- Create and map the multiple database schemas
- Enable the project for multiple databases
- Define multiple database connections
- Define and import queries for multiple databases
- Import multiple Datasource metadata into a Vocabulary
- Use multiple database connections as Ruleflow service callouts
- Test the rules when reading from multiple databases
- Test the rules when writing to multiple databases

Define multiple table namespaces

Note: Using the sample: This sample uses two namespaces. If you completed Getting Started with EDC on page 27 or Getting Started with ADC on page 37, you can just continue with its database, **PatientRecords,** as-is. You need to add another namespace, **CMSDetail**. If you choose to put one of these databases on another brand, you will need to use the brand's queries and data loaders supplied in Corticon Studio.

Set up your database product in a network-accessible location, and then define a database name. Note the database URL and port as well as the new database name. Be sure that your database will not deny connection using credentials -- for example, using `SQL Server Authentication` and not `Windows Authentication`. These parameters are all that is typically required to connect the Vocabulary to the database, create the schema for the persistent entities, and then bring the database metadata back to the Vocabulary.

If you want to use database installations on different machines, the database connections will handle the connection information.

Note: Refer to the Progress Software web page Progress Corticon 6.0 - Supported Platforms Matrix to review the currently supported database brands and versions.

Create and map the multiple database schemas

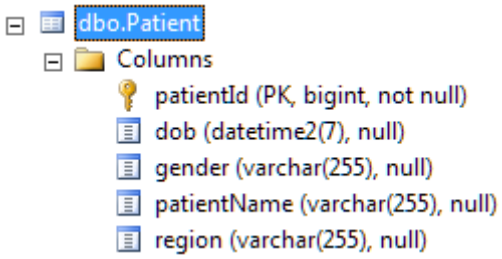
Note: Using the sample: For the Multiple Database sample, you need to have executed the Corticon SQL scripts `patient`, `adc` and `cms` for your database in your database management tool's editor. The metadata, primary keys, and join expressions are already mapped to the database.

Create the schema in the databases

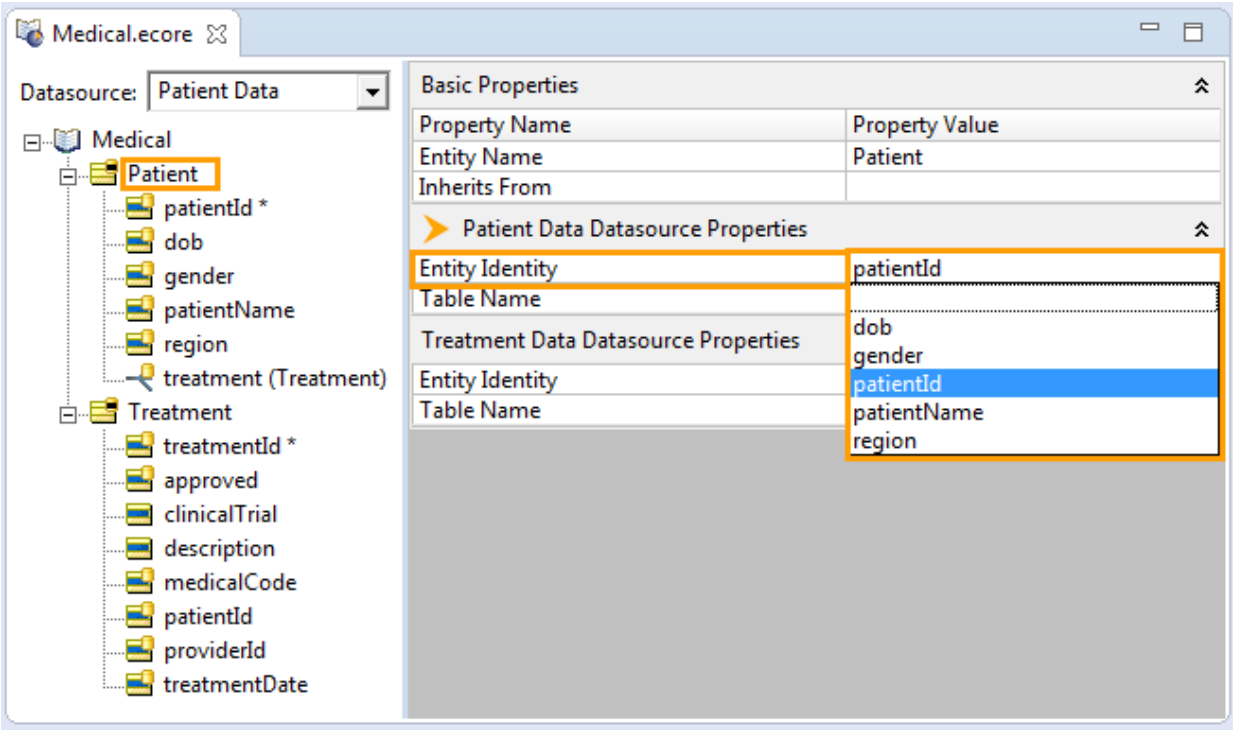
Typically, the database administrator creates the tables in the namespaces, and then the columns with their data types, the declared primary key, and any joins between tables.

Create the entities and their identity, then their attributes, and associations

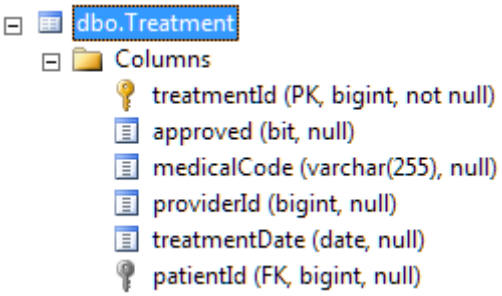
1. Define the database tables that will participate in rules as Corticon entities. For the sample, the first table is `Patient`:



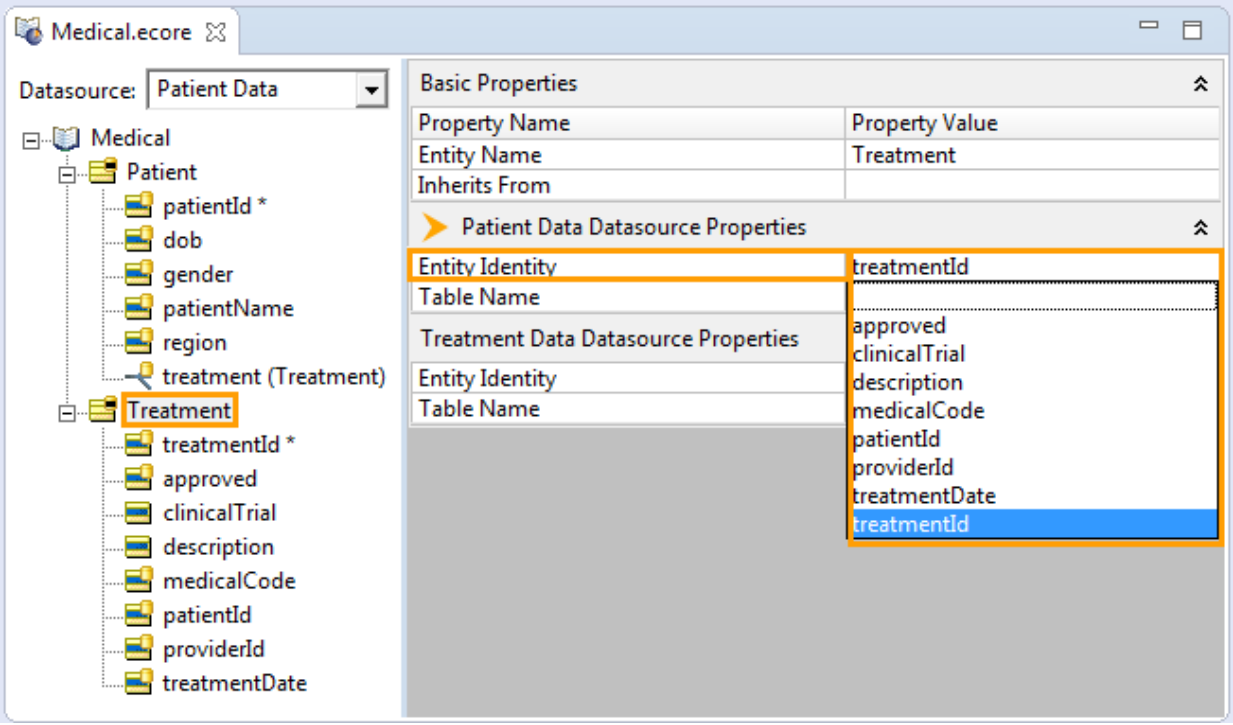
2. Add the required columns for that table as Corticon attributes with corresponding data types.
3. Specify the Primary Key (PK) in the database table as the Entity Identity by clicking the **Entity Identity** Property Value to open its menu, and then choose from the listed attributes, as shown:



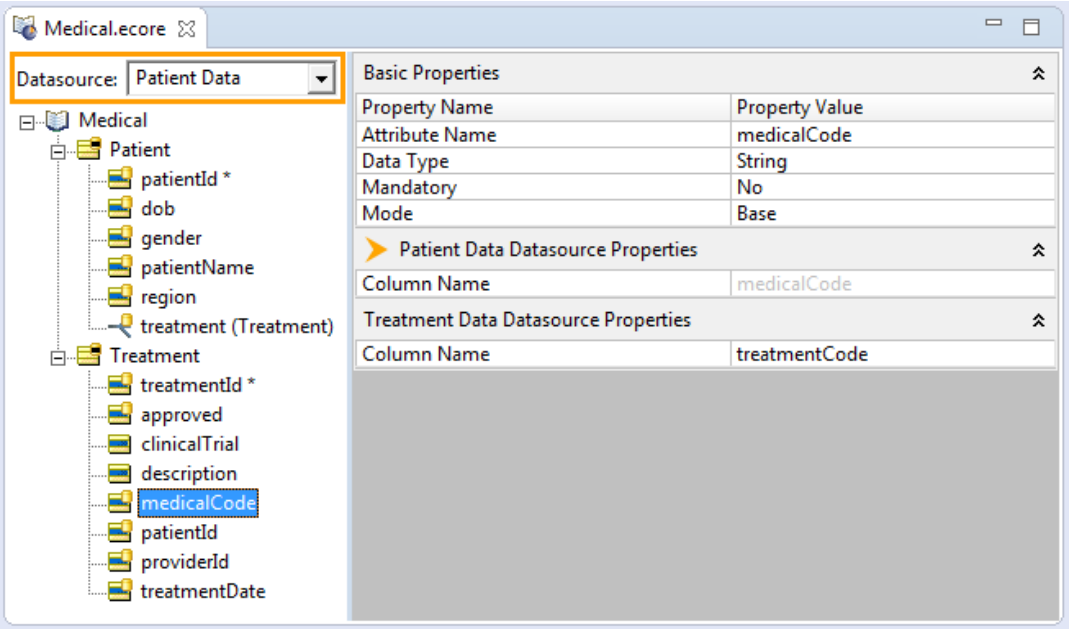
4. Add additional tables, such as the sample's `Treatment` table:



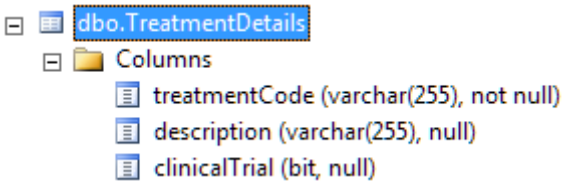
5. Specify its Primary Key (PK) in the database table as the Entity Identity by clicking the **Entity Identity** Property Value to open its menu, and then choose from the listed attributes, as shown:



6. The Treatment entity also links to another database through the attribute `medicalCode`



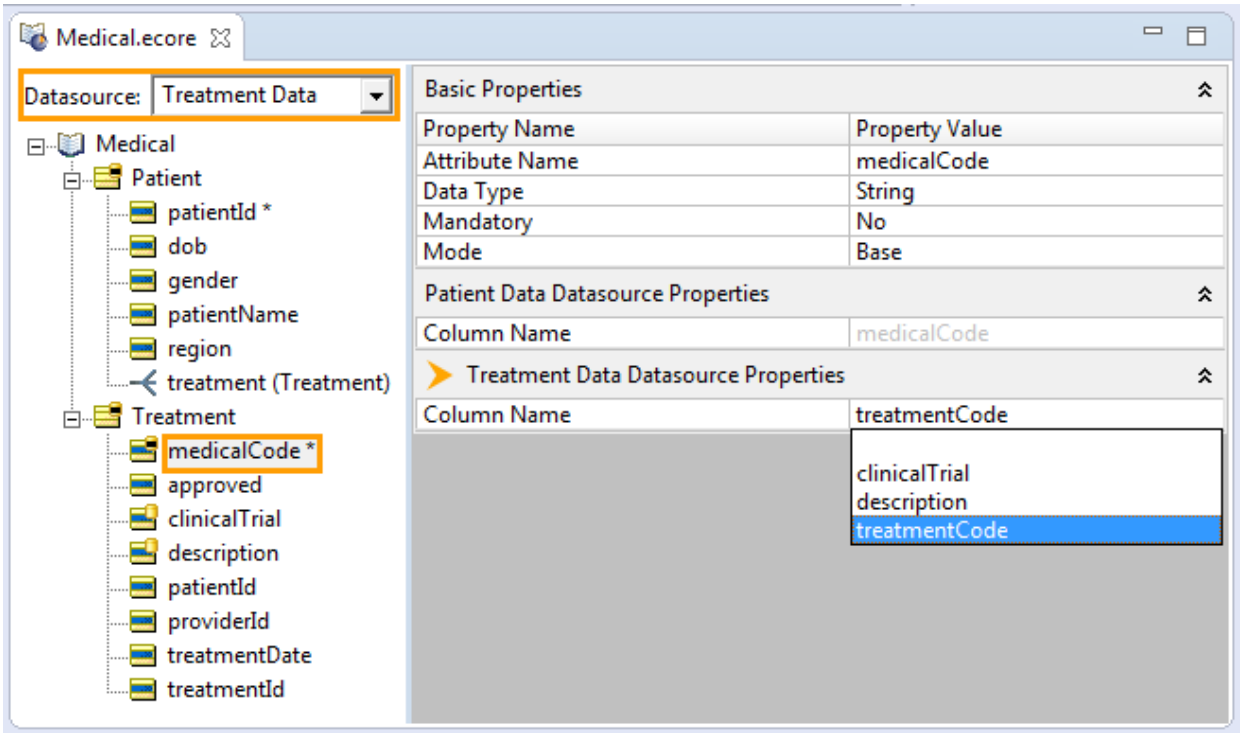
7. Each `medicalCode` needs to link to the Primary Key (PK) `TreatmentCode` in the table `TreatmentDetails` in the `CMSDetail` database



8. In Corticon Studio on the Vocabulary's Datasource tab, click METADATA: **Import**. The mapping metadata from the database is added into the Vocabulary for the Entities (tables), Attributes (columns), and Associations

(join expressions). If the imported tables and columns do not align with entities and their attributes, those values will require manual mapping.

9. One such case is noted here. *SmartMatching* won't infer the table name through a link based on an attribute so you need to pull down the **Table Name** options to choose `CMSDetail.dbo.TreatmentDetails` table, as illustrated:



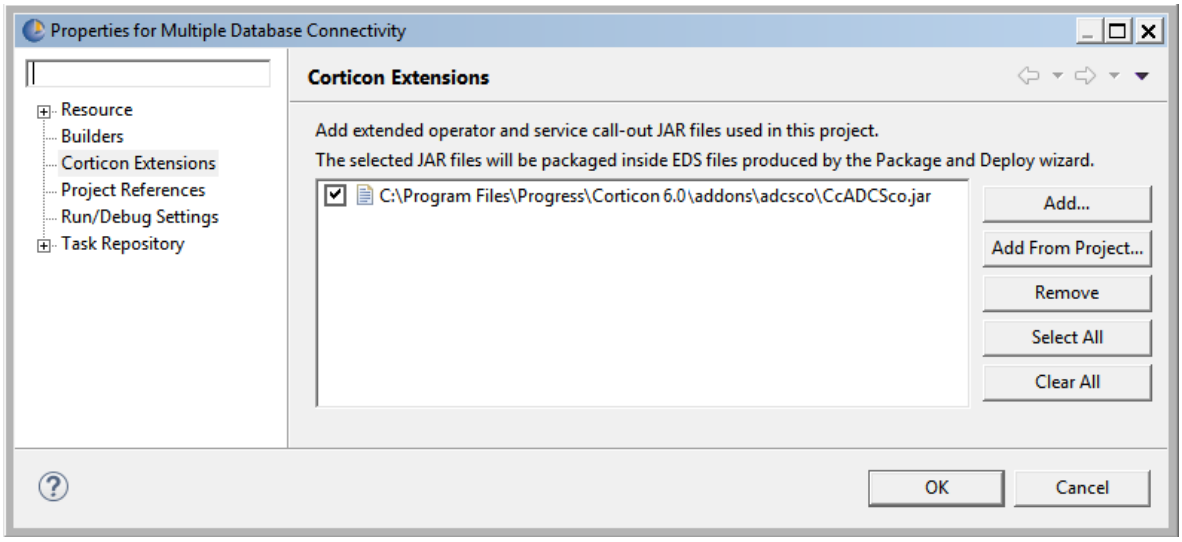
If other imported tables and columns do not align with entities and their attributes, those values will require manual mapping. For more about mapping, see Mapping ADC database metadata on page 155.

Enable the project for multiple databases

Note: The sample has added a link to the required JAR file. This path is valid if your Corticon Studio installation path is the default, as illustrated. Otherwise, remove the default listed and add your path to the JAR file. Updating this preset value resolves the error shown in the project.

Each project that intends to use ADC must add a packaged set of Corticon extensions into its project. On Corticon Studio, do the following:

1. Right-click on the project name in the Project Explorer, and then choose **Properties**.
2. Click **Corticon Extensions**.
3. Click **Add**, and then locate and choose the file `[CORTICON_HOME]\addons\adcsco\CcADCSco.jar`, as illustrated:

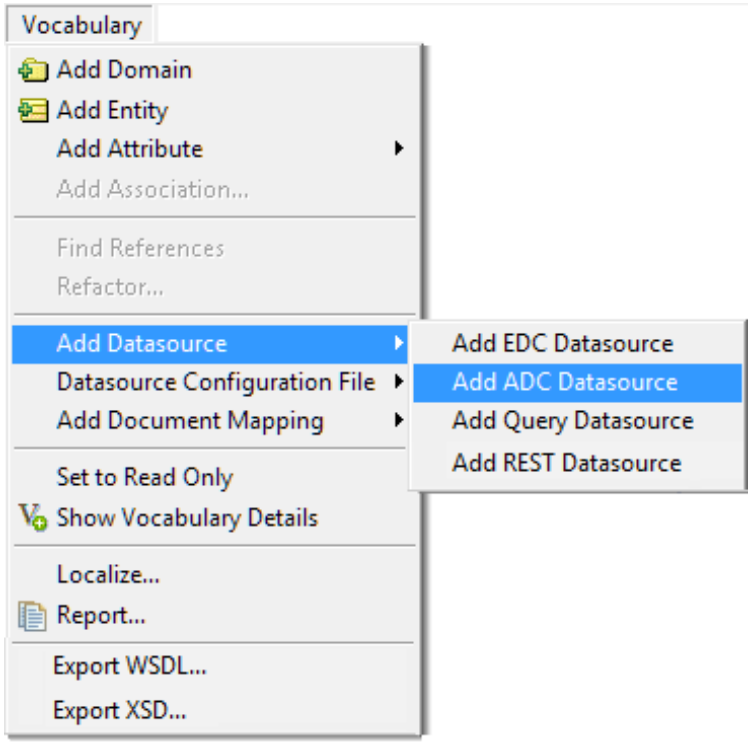


4. Click **OK**.

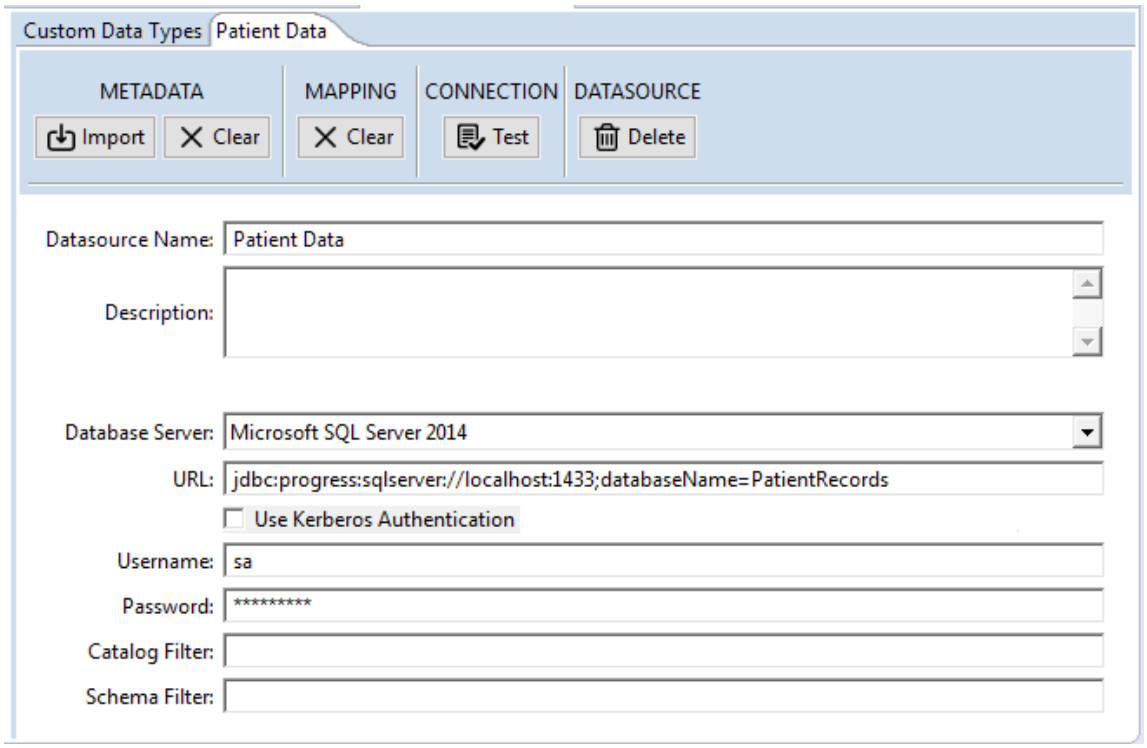
Define multiple database connections

Note: Two ADC database connections are defined in the Vocabulary. Just enter their credentials, and then test each connection.

To connect a Vocabulary to a defined database where you will use SQL queries, define the connection as an ADC Datasource. In the project's Vocabulary editor, select the Vocabulary command **Add Datasource > Add ADC Datasource**, as shown:

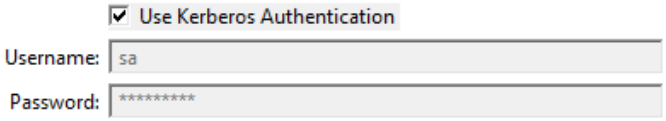


An **ADC** tab is added to the root level of the Vocabulary. The ADC sample renamed the first Datasource to **Patient Data** which is now the name on its tab, as illustrated:



where:

- **Datasource Name**: The connection name that displays on the Vocabulary tab. This is also its name that will bind this ADC connection to an instance of the service call-out in a Ruleflow. While you can change this name, it is a good idea to do so before you specify it in Ruleflow Service Call-out's **Service Name**. When you add additional ADC Datasource tabs, the default names will increment `n` in `ADC_n`.
- **Description**: An informative description of the use of the Datasource you are adding.
- **Database Server**: The database product. Click the dropdown menu on the right side of the entry area to list the available database brands. Corticon embeds Progress DataDirect JDBC Drivers for each database. These drivers provide robust, configurable, high-availability functionality to RDBMS brands. The drivers are pre-configured and do not require performance tuning.
- **Use Kerberos Authentication**: Indicates that the user will use the Kerberos authentication protocol. The Username and Password are greyed as they are not relevant for the Kerberos protocol:



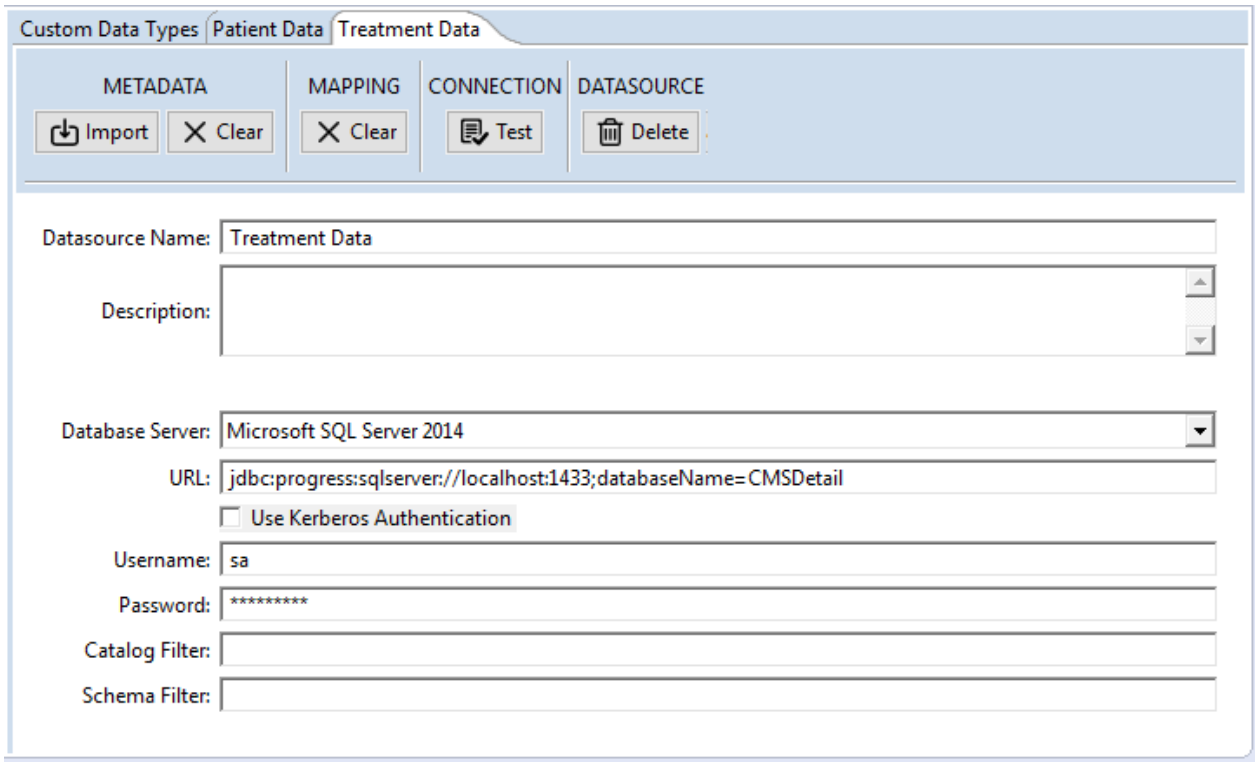
- **Database URL**: The preconfigured URL for the selected database server. You must edit the default entry to replace (1) `<server>` with the machine's DNS-resolvable hostname or IP address and port , and (2) `<database name>` with the database name that was set up (typically case-sensitive).
- **Username**: The user credentials that enable connection to the database. The credentials are encrypted when the database access file is exported for deployment.
- **Password**: The specified user's password.
- **Catalog Filter and Schema Filter**: Patterns that refine the metadata that is imported during **Import Database Metadata**

Note: Filters are a good idea for production databases that might have hundreds or even thousands of schemas. As the Catalog filter value does not support wildcards, distinguishing two metadata import filters enables the use of wildcards in the Schema filter value: Underscore (_) provides a pattern match for a single character. Percent sign (%) provides a pattern match for multiple characters (similar to the SQL LIKE clause.) For example, you could restrict the filter to only schemas that start with DATA by specifying: `DATA%`. The ability to specify patterns is especially valuable when testing performance on RDBMS brands with applications that use multiple schemas.

Click the CONNECTION **Test** button. Confirm that you have a valid connection before proceeding.

Another database connection

When another **ADC** tab is added to the root level of the Vocabulary, you define a separate database connection. In the sample, the second Datasource was renamed to **Treatment Data** which is now the name on its tab, as illustrated:

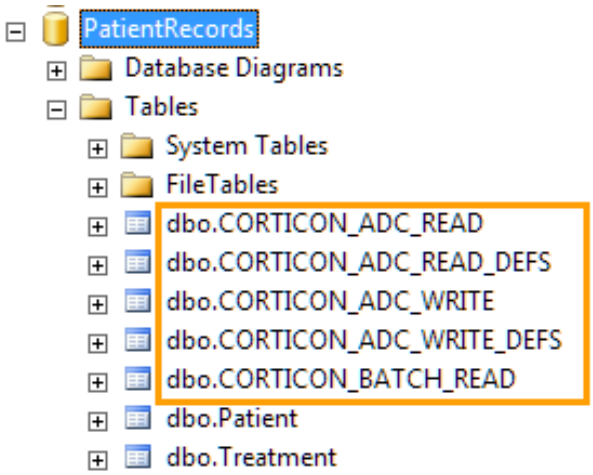


Confirm that you have a valid connection before proceeding to define a Query Datasource.

Define and import queries for multiple databases

Queries are essential to how ADC functions. With just a few queries, a lot of the SQL tasks are minimized as the rules processing handles complex conditions and actions.

Running the sample's `adc` script created five tables that will be referenced by the query service. `ADC_READ` and `ADC_WRITE` access their `DEF` table to do the steps requested by your Ruleflow Service Call-outs.

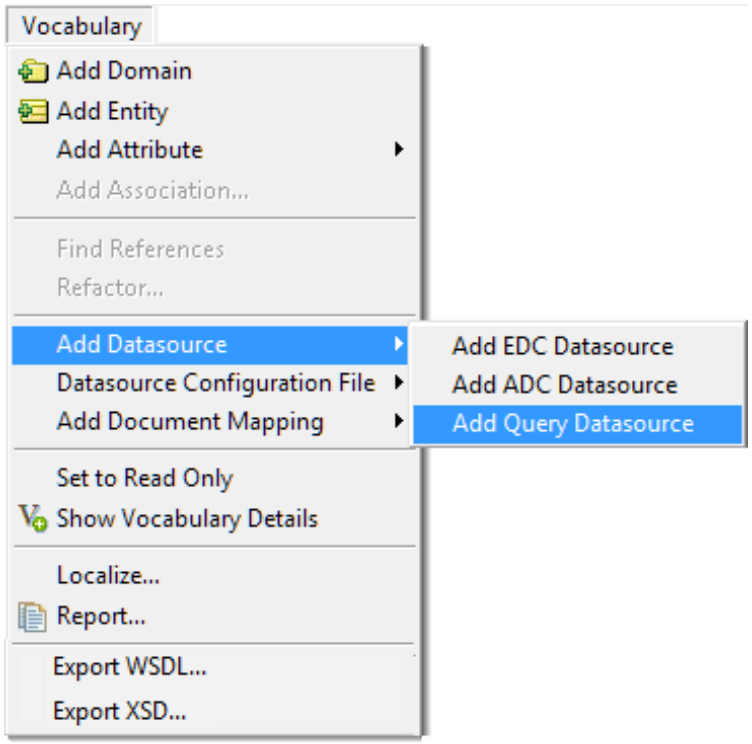


Note: The `BATCH_READ` table inserts the queries you will use in Getting Started with Batch on page 93. For more about the sample's Corticon's queries, see How Corticon is expressed in SQL on page 163.

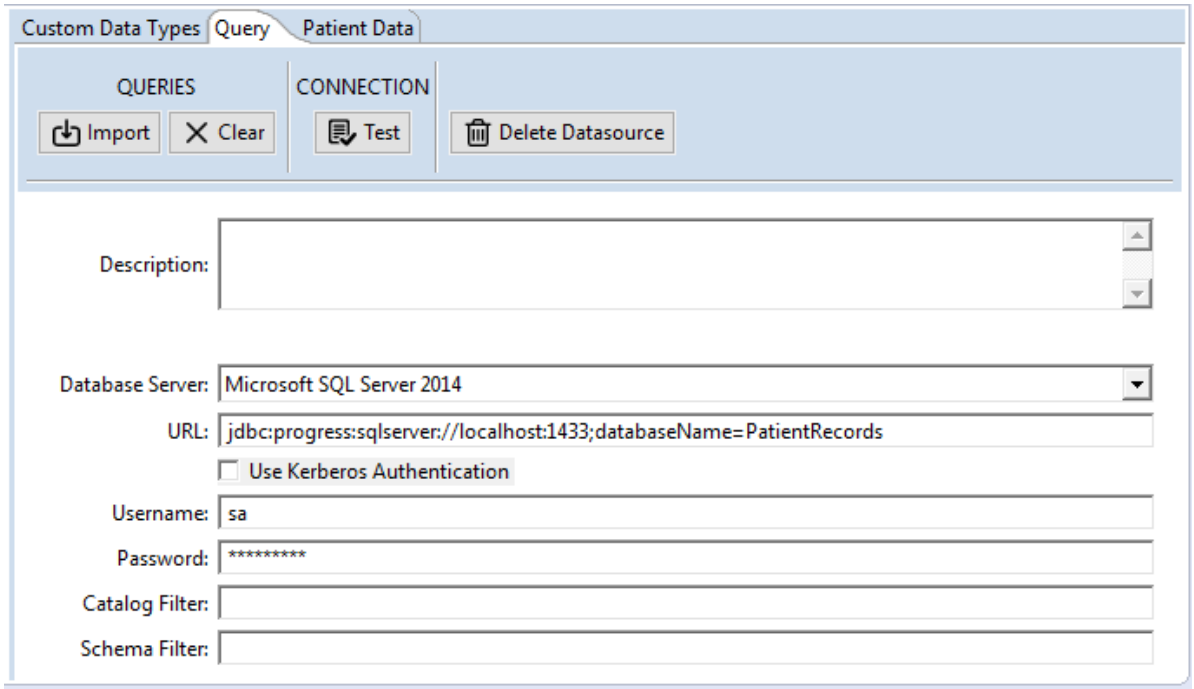
Define the Query Datasource connection

No matter how many ADC connections you make for the Vocabulary in your project, you need to define one and only one Query Datasource that will be used by all services and batch processes in the project's Decision Services.

To connect a Vocabulary to a defined database where you will read the SQL queries, define the connection as a Query Datasource. In the project's Vocabulary editor, select the Vocabulary command **Add Datasource > Add Query Datasource**, as shown:



The **Query** tab is added to the root level of the Vocabulary as a fixed-name tab, as illustrated:



where:

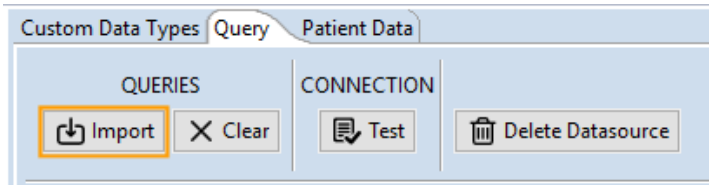
- **Description**: An informative description of the Query Datasource.
- **Database Server**: The database product. Click the dropdown menu on the right side of the entry area to list the available database brands. Corticon embeds Progress DataDirect JDBC Drivers for each database. These drivers provide robust, configurable, high-availability functionality to RDBMS brands. The drivers are pre-configured and do not require performance tuning.
- **Database URL**: The preconfigured URL for the selected database server. You must edit the default entry to replace (1) `<server>` with the machine's DNS-resolvable hostname or IP address and port , and (2) `<database name>` with the database name that was set up (typically case-sensitive).
- **Use Kerberos Authentication**: When selected, indicates that the user will use the Kerberos authentication protocol. The Username and Password are greyed as they are not relevant for the Kerberos protocol:

- **Username**: The user credentials that enable connection to the database. The credentials are encrypted when the database access file is exported for deployment.
- **Password**: The specified user's password.
- **Catalog Filter and Schema Filter**: Patterns that refine the metadata that will be imported.

Click the CONNECTION **Test** button. Confirm that you have a valid connection before proceeding.

Import the queries

To access the queries for use in Service Callouts in Ruleflows, click the QUERIES **Import** button:



The current query names in the Query Datasource are accessed and brought into the local persistent cache. Whenever query names are revised, you need to re-import the queries. If the query defs referenced by a query name change, the latest defs will be accessed by the query name.

Import multiple Datasource metadata into a Vocabulary

When the database schema exists, its metadata can be imported into Corticon Studio.

You can control the tables that are accessed to transfer metadata to the Vocabulary. When only a small subset of tables will supply the metadata that is needed, the time and space overhead of the process is reduced by delimiting the number of tables.

To import the metadata from the databases into the Vocabulary:

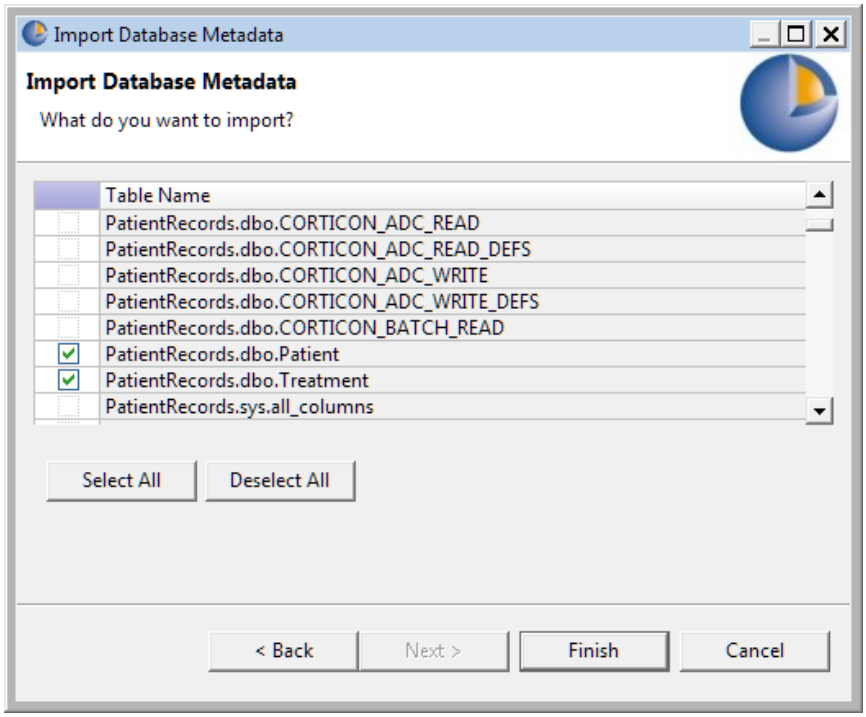
1. Open the project's Vocabulary into its editor.
2. Select the Vocabulary root
3. Select, in turn, each database connection tab:

a. Click **Test Connection** to confirm that there is a good connection..

b. Click METADATA **Import**

c. In the dialog, accept **Import all tables** or click **Choose tables for database metadata import**, and then click **Next**. The panel lists all the tables in the connected database.

d. If you do not want all, click **Deselect All**, and then choose specific tables. For the sample's **Patient Data** Datasource, just two tables are selected. The query tables listed do not have metadata so they can be deselected.

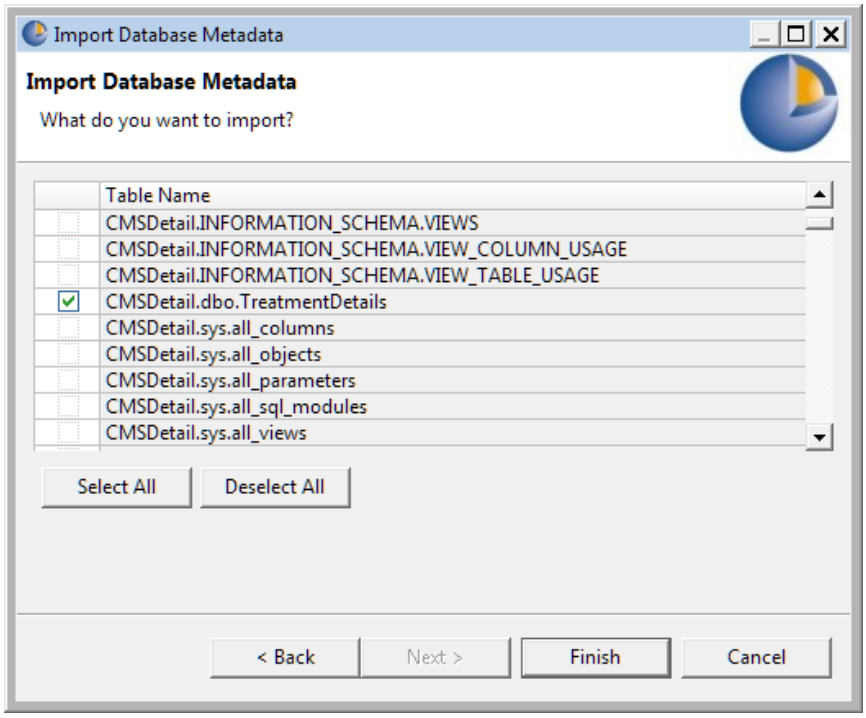


e. Click **Finish** to perform the task.

4. As database metadata is imported into a Vocabulary, the Vocabulary Editor's automatic mapping feature attempts to find the *smart match* for each piece of metadata. An entity will be auto-mapped to a table if the two names are spelled the same way, regardless of case.

Importing another Datasource's metadata

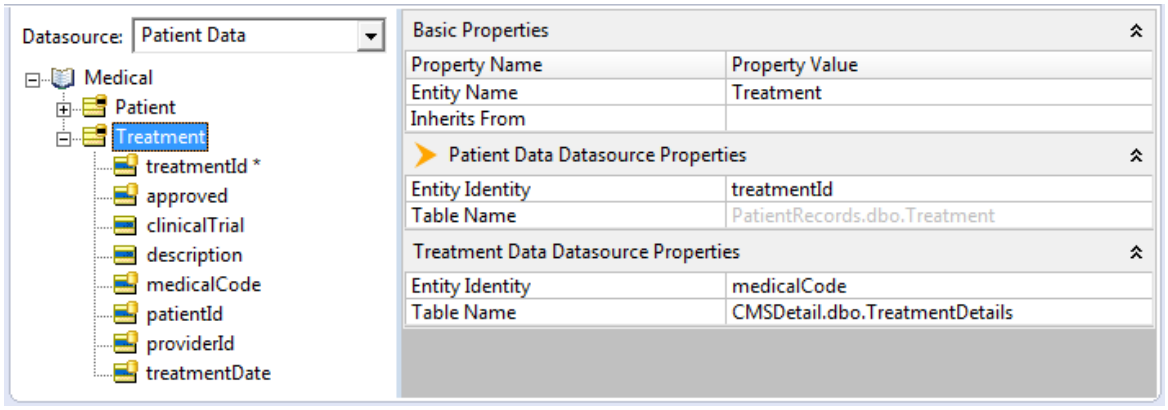
When the Vocabulary is supporting multiple database connections, each source is mapped separately to its respective tables. For the sample's **Treatment Data** Datasource, just one table is selected.



A closer look at MDB metadata

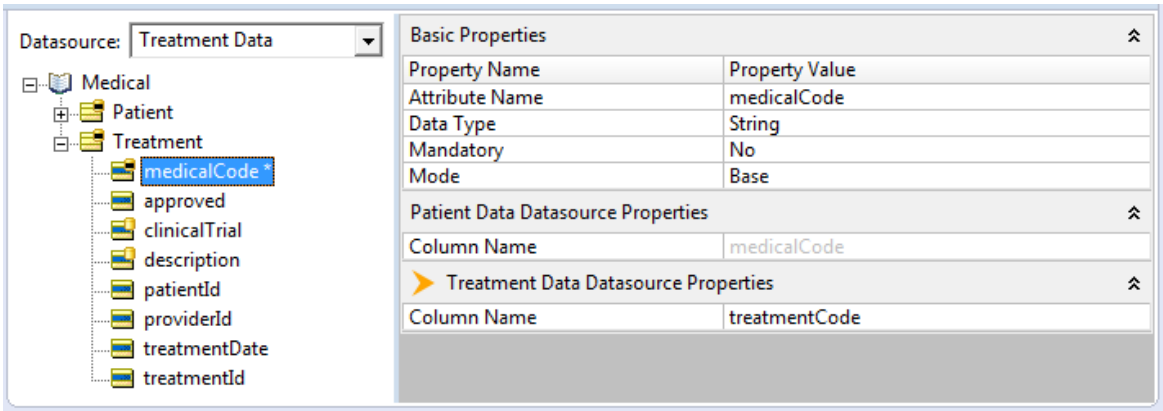
When the two Datasources defined in the Vocabulary text bring in their metadata, the `Treatment` Entity is shown to have bindings to both Datasources through the Attribute `medicalCode`.

Figure 6: Multiple Datasources bound to the Treatment Entity



The linkage enables each execution to take the `medicalCode` value in the request to access the corresponding `treatmentCode` and the data in its row in the other Datasource.

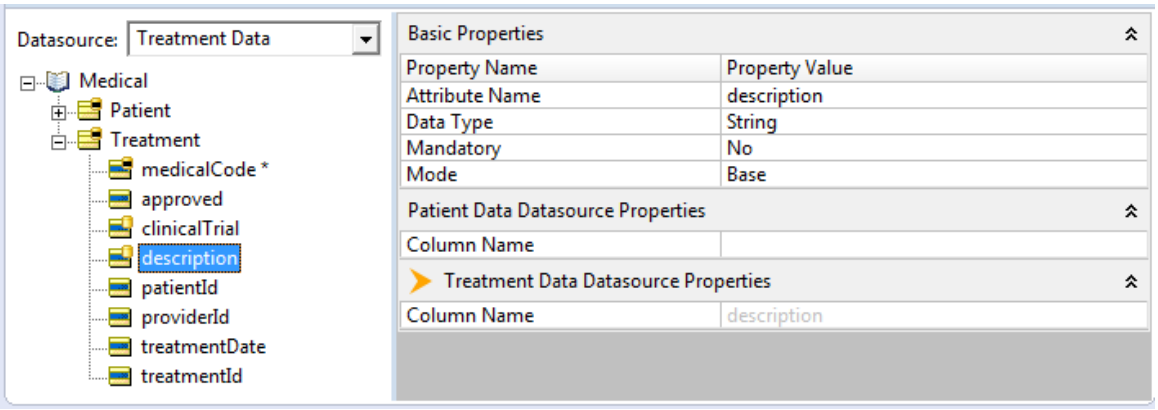
Figure 7: Multiple Datasources bound to the medicalCode Attribute



You can consolidate the data from tables and columns residing in different schemas into one intelligent vocabulary entity that can apply rules across the consolidated data. The rules modeler does not need to be concerned about how the data is sourced -- rules can be written to one, simplified semantical representation of the underlying database model.

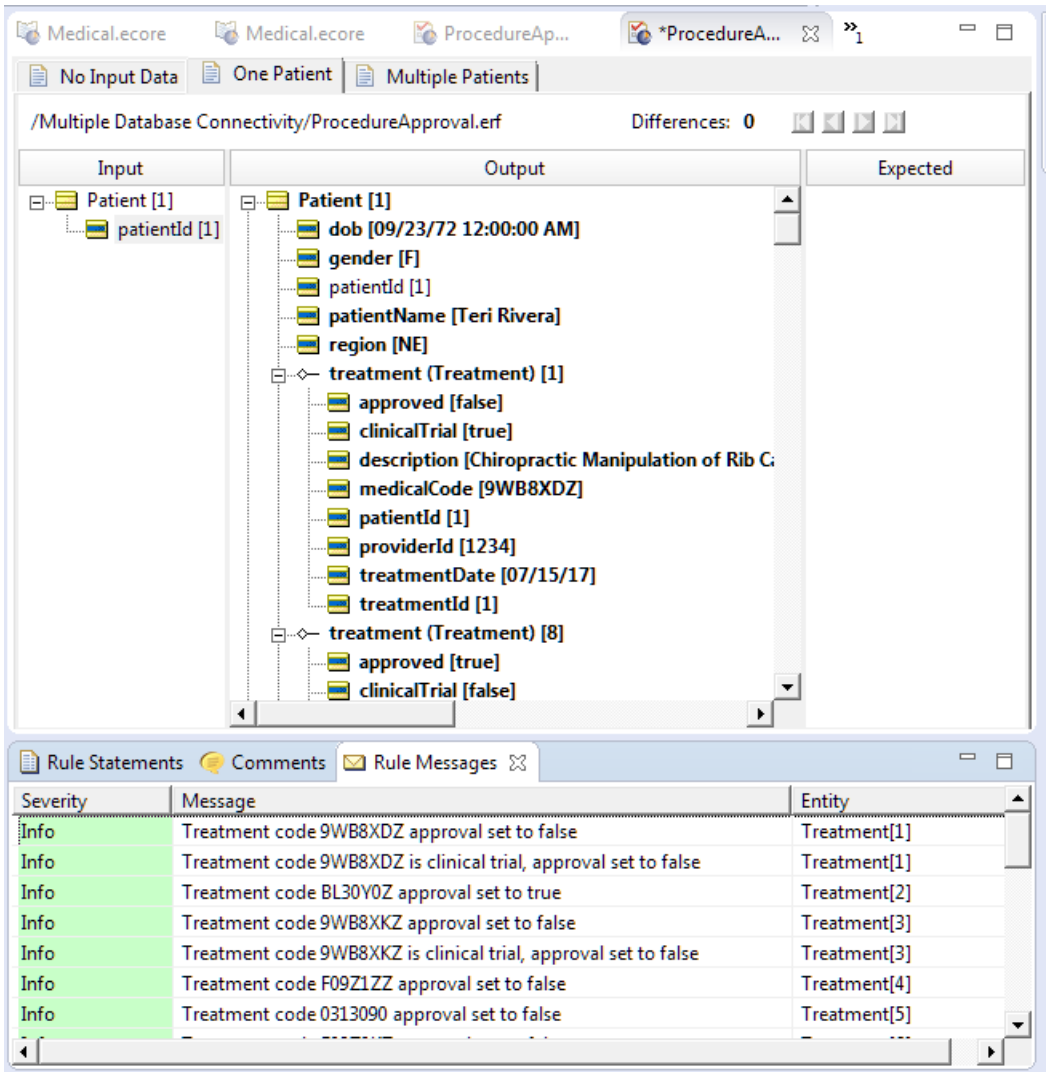
The `description` value is in grey indicating that *SmartMatch* found an unambiguous match of precisely the same name in the `CMSDetails` Datasource.

Figure 8: Lookup of description value from the related Datasource



When tests are run, both Datasources are connected to seamlessly provide the output of their combined data.

Figure 9: Ruletest that gets output from multiple Datasources

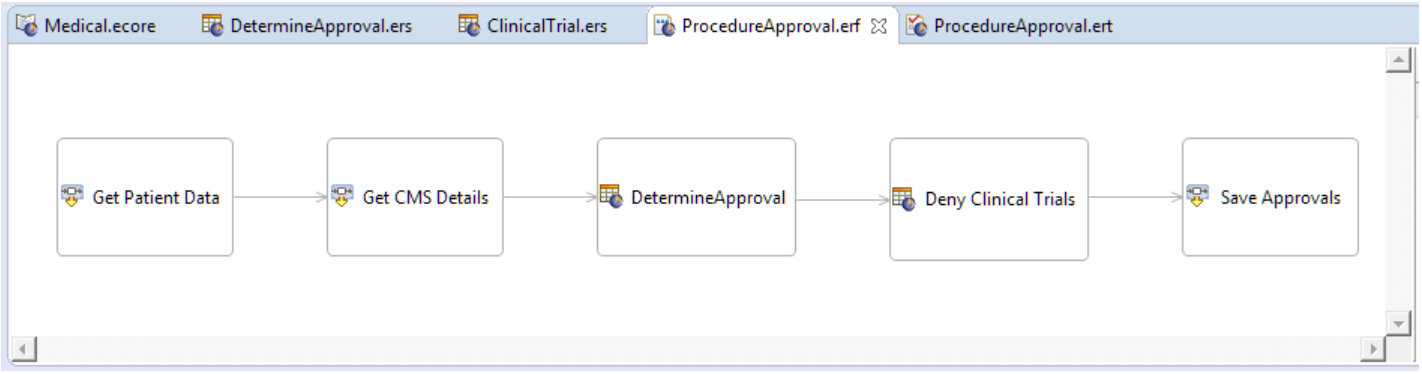


Use multiple database connections as Ruleflow service callouts

Note: The Ruleflow objects and their runtime properties are already defined in the sample.

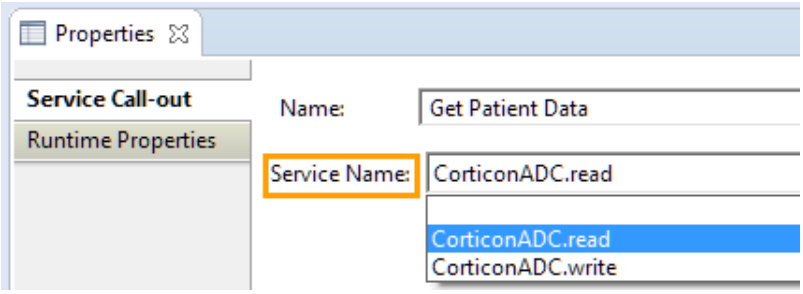
To use multiple ADC connections in service callouts:

1. In a Ruleflow where you want to use multiple ADC connections, create Service Call-out objects on the Ruleflow canvas. In this example, the Ruleflow is defined with the project's Rulesheet plus a Service Call-out to read and another one to write back to the database table after rule processing, as shown:

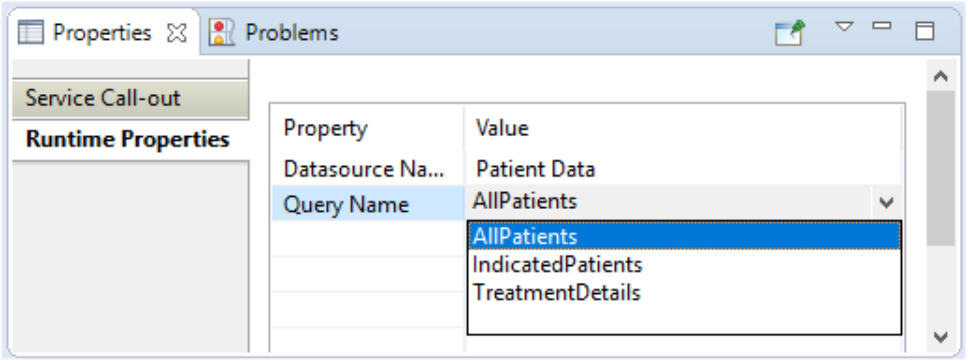


This sample points out that you can have two ADC connections that have read and write actions.

2. Click on the **Get Patient Data** object, and then, on the object's **Properties** tab.
3. On its **Service Call-out** tab, click the **Service Name** pulldown to select the service you want for this use, as shown here where the sample uses `CorticonADC.read` for this service, as illustrated:



4. On its **Runtime Properties** tab. Use the **Property** pulldown to:



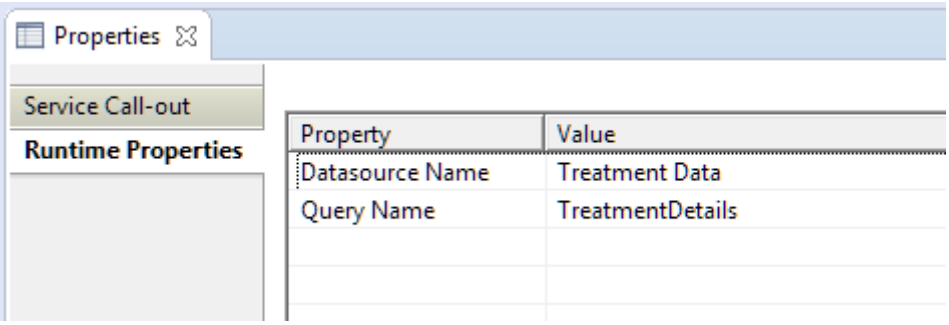
   a. Select **Datasource Name**, and then, for its value, use the dropdown menu to select the Datasource that corresponds to the name of the datasource to which you want to apply a query. For the ADC sample, choose **Patient Data**.

   b. Select **Query Name**, and then, for its value, use the dropdown menu to select the appropriate query. For the ADC sample, choose **AllPatients**. The listed values are initially imported from the Query Datasource, and are updated by the **QUERIES: Import** function on the **Query** tab.

Defining a Service Call-out to another database

Another Service Call-out object on the canvas can access another database:

1. Click on the **Get CMS Details** object, and then, on the object's **Properties** tab.
2. On its **Service Call-out** tab, click the **Service Name** pulldown to select the service you want for this use, as shown here where the sample again uses `CorticonADC.read` .
3. On its **Runtime Properties** tab. Use the **Property** pulldown:

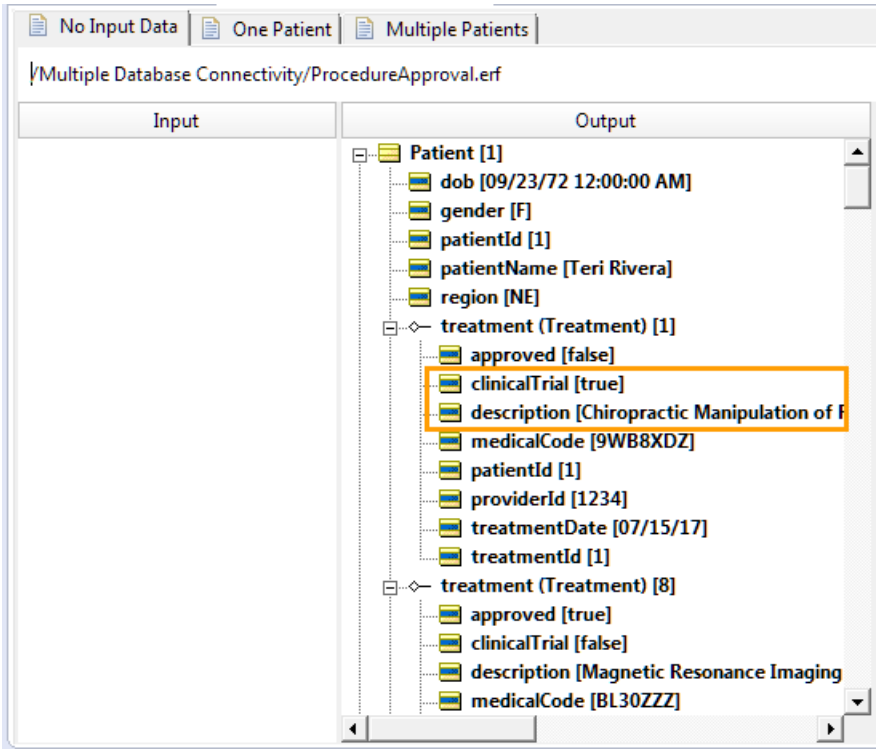


a. Select **Datasource Name**, and then, for its value, use the dropdown menu to select the Datasource that corresponds to the name of the appropriate ADC definition. For the ADC sample, enter **Treatment Data**.

b. Select **Query Name**, and then, for its value, use the dropdown menu to select the appropriate query. For the ADC sample, choose **TreatmentDetails**. The listed values are initially imported and are updated by the **QUERIES: Import** function on the **Query** tab.

Test the rules when reading from multiple databases

Open the sample's Ruletest, `ProcedureApproval.ert`. It opens on the Testsheet **No Input Data**. Click **Run** to compile and run the rules. The output shows patient and treatment data.



The highlighted treatment attributes differentiate this sample from the ADC sample. Here, information read from the first database provided the lookup value for the read in the second database. And the added Rulesheet simply states that treatments for clinical trials are not approved.

Run test with a different query

Change the Service Call-out's **Query Name** to `IndicatedPatients`. Then, return to the Ruletest. When you run the test for the **No Input Data** testsheet, you get no results. When you run the Testsheet for the **One Patient** and **Multiple Patients** Testsheets, you get exactly that one patient and the specified three patients.

The first part of this SQL statement is:

```
SELECT * FROM Patient WHERE patientId IN ({Patient.patientId})
```

The curly braces indicate tokens in the query that will be replaced by the data passed to Corticon -- in this case, selecting the patients to process.

Note: For more about the format of Corticon's queries, see How Corticon is expressed in SQL on page 163.

Test the rules when writing to multiple databases

In the database, the approval status is being evaluated but it is not being entered into the database, as shown:

treatmentId	approved	medicalCode	providerId	treatmentDate	patientId
1	*NULL*	9WB8XDZ	1234	2017-07-15	1
2	*NULL*	BL30Y0Z	5678	2017-08-01	4
3	*NULL*	9WB8XKZ	5678	2017-03-12	2
4	*NULL*	F09Z1ZZ	1234	2017-07-01	6
5	*NULL*	0313090	4321	2017-09-05	7
6	*NULL*	F09Z0KZ	1234	2017-09-28	2
7	*NULL*	BD41ZZZ	1234	2017-08-03	3
8	*NULL*	BL30ZZZ	4321	2017-06-12	1
9	*NULL*	B512ZZZ	8765	2017-10-30	8
10	*NULL*	F09Z0KZ	4321	2017-10-04	7

On the Ruleflow canvas, click on the `Save Approvals` Service Call-out on the canvas. Its Query Name is `UpdateTreatment` whose query SQL is:

```
UPDATE Treatment SET approved={Treatment.approved} WHERE
treatmentId={Treatment.treatmentId}
```

When you run the Testsheet, the approval values are written to the database for patient treatments, as shown:

treatmentId	approved	medicalCode	providerId	treatmentDate	patientId
1	False	9WB8XDZ	1234	2017-07-15	1
2	*NULL*	BL30Y0Z	5678	2017-08-01	4
3	False	9WB8XKZ	5678	2017-03-12	2
4	*NULL*	F09Z1ZZ	1234	2017-07-01	6
5	*NULL*	0313090	4321	2017-09-05	7
6	False	F09Z0KZ	1234	2017-09-28	2
7	True	BD41ZZZ	1234	2017-08-03	3
8	True	BL30ZZZ	4321	2017-06-12	1
9	*NULL*	B512ZZZ	8765	2017-10-30	8
10	*NULL*	F09Z0KZ	4321	2017-10-04	7

Where to now?

The flow of this document leads to Deploying projects that use data integration on page 89, an important preparation for Getting Started with Batch on page 93. Beyond that, the material focuses less on the samples by using various configurations to present advanced topics.

9

Getting Started with REST

Corticon's REST connectivity allows you to access data via REST services such that rule payloads can be enriched with REST data similar to database data. Corticon uses the Progress Data Direct Autonomous REST Connector to provide support for REST callouts. This allows you to access a REST service similar to the way you access a database. When integrated with Corticon, the Datasource can use mechanisms for smart data integration and Vocabulary mapping. Rules use these Service Callouts to retrieve data as steps in a Ruleflow, even allowing you to substitute payload attributes into REST URLs before execution.

For details, see the following topics:

- Overview of the Autonomous REST Connector
- Enable the project for REST SCO
- Define a Datasource connection for REST
- Create and map the REST schema
- Use REST data sources in a Ruleflow
- Test rules when importing from the REST Datasource
- Revise Connection and Service Call-out to retrieve data

Overview of the Autonomous REST Connector

The Progress DataDirect Autonomous REST Connector for JDBC is a driver that supports SQL read-only access to JSON-based REST API data sources. To support SQL access to REST services, the driver creates a relational map of the returned JSON data and translates SQL statements to REST API requests. The driver can either infer a map at the beginning of a session or can leverage a configuration REST file that allows you to modify and persist a map.

The use of the Autonomous REST Connector allows Corticon to present REST services as if they are relational databases. How you map a Corticon vocabulary to a REST service is the same as if you are mapping to a database. The only difference is in the configuration of the data source. Internally, the Autonomous REST Connector stores data in an in-memory database which Corticon access with SQL queries - the same as it does for database data sources.

This sample introduces the Autonomous REST Connector to retrieve reimbursement rates for medical procedure codes from a REST service. A procedure may have more than one rate where each rate has its range of effective dates.

To use REST services:

1. **Add a REST data source to your vocabulary** - Specify a REST URL, and then provide authentication credentials, and specify any parameters.
2. **Map your vocabulary to a REST Datasource** - In the Corticon vocabulary editor, map the entities, attributes, and associations that transform data retrieved from the REST Datasource.
3. **Add the REST Service callout to a Ruleflow** - In the Corticon Ruleflow editor, when you add a Service Call-out to a Ruleflow, you configure it to use the REST Datasource and identify the queries to be performed by selecting whether to do specific or bulk reads.

When all steps are completed you are ready to deploy your Ruleflow or test it in the Corticon tester. You can use multiple REST Services in a Ruleflow.

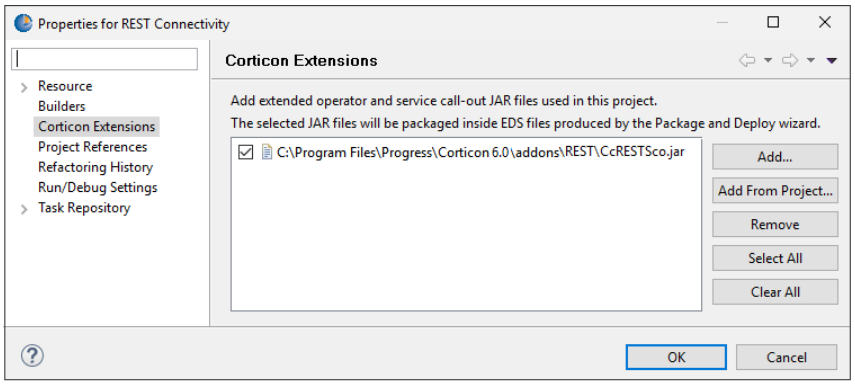
The following topics walk through the REST Connectivity sample. You are encouraged to follow along in Corticon Studio. After this sample has completed, advance to the Mixed Connectivity sample where relational database and REST Datasources work together seamlessly.

Enable the project for REST SCO

Note: To use a REST Datasource you need to add the Corticon REST service call-out jar file to your project. The sample has this jar file added as a dependency. This path is valid if your Corticon Studio installation path is the default, as illustrated. Otherwise, remove the default listed and add your path to the JAR file.

Each project that intends to use REST Datasources must include the Corticon REST service call-out jar file as a dependency. On Corticon Studio, do the following:

1. Right-click on the project name in the Project Explorer, and then choose **Properties**.
2. Click **Corticon Extensions**.
3. Click **Add**, and then locate and choose the file `[CORTICON_HOME]\addons\CcRESTSco.jar`, as illustrated:



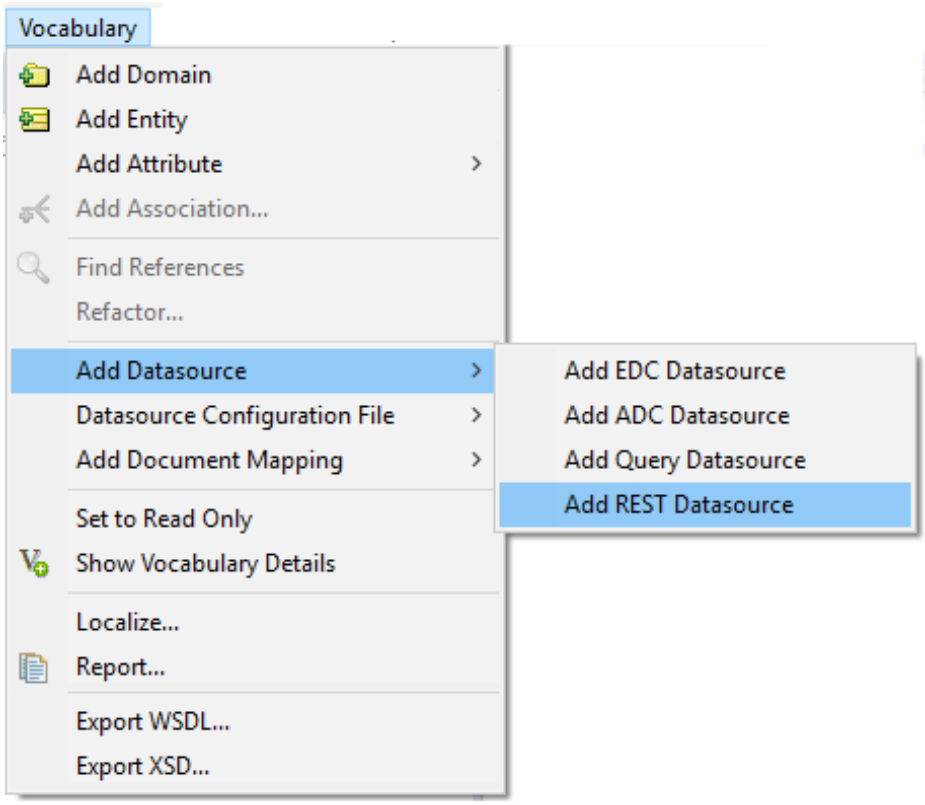
4. Click **OK**.

Note: If you installed Corticon to other than the default location, you'll notice that updating this preset value resolves the error shown in the project.

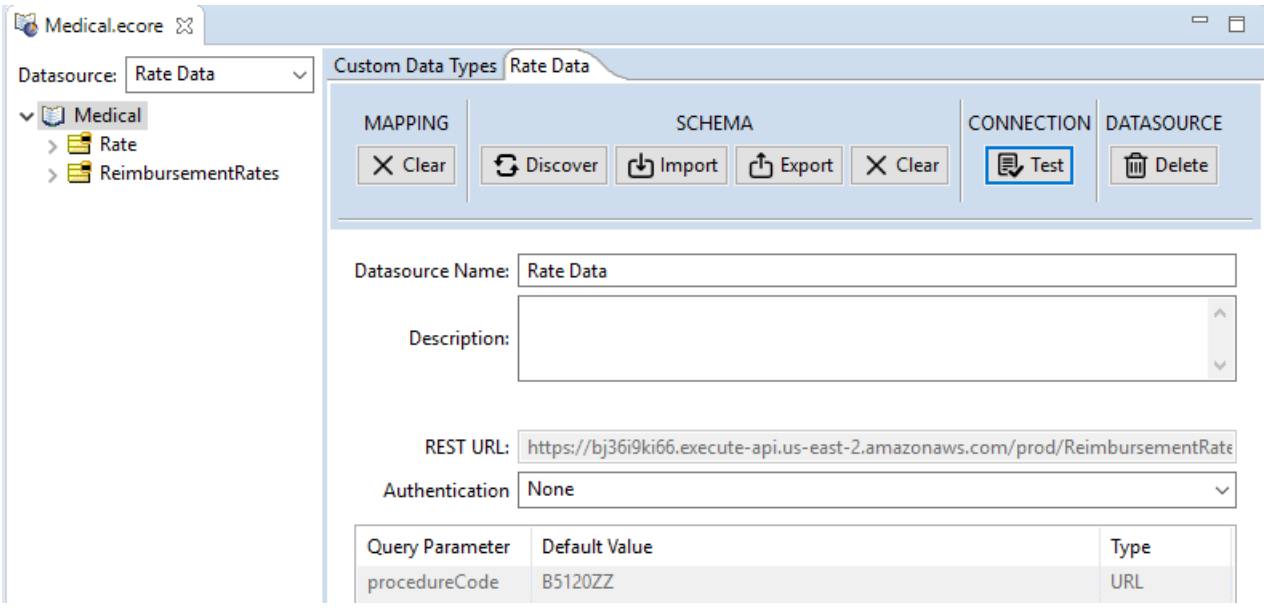
Define a Datasource connection for REST

Note: The REST Datasource connection is defined in the Vocabulary. It does not require authentication so, to test the connection, just click the **Test** button.

The REST Connectivity sample already has a REST data source defined in its vocabulary. To use your own REST data source you would first it to your vocabulary. To add a REST data source, select the Vocabulary command **Add Datasource > Add REST Datasource**, as shown:



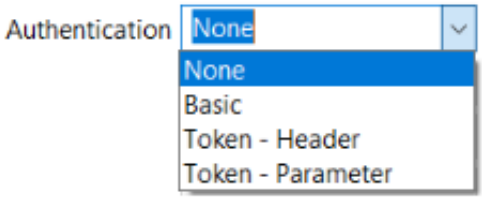
A **REST Service** tab is added to the root level of the Vocabulary. The REST sample renamed this Datasource to **Rate Data** which is now the name of its tab, as illustrated:



where:

- **Datasource Name**: The connection name that displays on the Vocabulary tab. This is also its name that will bind this REST Datasource connection to an instance of the service call-out in a Ruleflow. While you can change this name, it is a good idea to do so before you specify it in Ruleflow Service Call-out's **Service Name**.
- **Description**: Provide an informative description of the use of the Datasource you are adding.

- **REST URL**: The URL for the REST service. The protocols `http` and `https` are supported. If the URL includes a query, the parameters and their values are moved to the **Query Parameters** table.
- **Authentication**: There are four options for security on a REST Service connection:



  The default setting is `None`. When a REST service requires no authentication, it is appropriate for accessing an unsecured REST service. For example, government census data in the public domain. For information on the other options for security and authorization on REST Services, see Authentication on REST Service connections on page 168.

- **Query Parameters** and their respective **Default Values**. The query parameters on a REST URL when it was pasted in were transformed into Query Parameter/Default Value pairs. You can edit, add, and delete from the table as needed for your use case.

  When configuring usage of a REST data source in a Ruleflow you can identify entity attributes to substitute for query parameter values at runtime. This allows you to pass dynamic data values from the payload being processed, such as a "customer id", to your REST service. If you don't specify an entity attribute for a query parameter in the Ruleflow, the static value specified on the Datasource will be used at runtime.

Click the CONNECTION **Test** button. Confirm that you have a valid connection before proceeding to discover or import the schema, and then to refine the mapping.

Create and map the REST schema

To use a REST data source, Corticon needs to know how to map the JSON from the REST service to relational tables. The easiest way to do this is to **discover** the schema. The Autonomous REST Connector used by Corticon has the ability to query a REST service and infer a relational schema from the JSON returned.

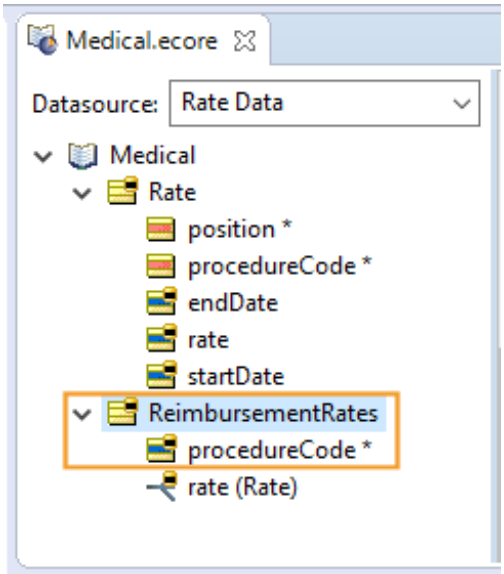
1. In Corticon Studio on the Vocabulary's Datasource tab, click SCHEMA: **Discover**. Corticon will query the REST service using the REST URL and query parameters defined on the data source. The JSON returned will be used to generate a schema for mapping JSON from the REST service to a relational representation. The metadata for this schema is added to the Vocabulary for the Entities (tables), Attributes (columns), and Associations (join expressions). If the imported tables and columns do not align with entities and their attributes, those values will require manual mapping.

   The schema for how to map the JSON from a REST service to a relational model is stored in the vocabulary. You can export this schema to a text file using **Export** then import it back to your vocabulary using **Import** after any modifications.

   Note: When using discovery to generate the schema, the JSON returned needs to be representative of the data which would be returned at runtime. Any data elements not represented, will not be in the generated schema and therefore not available for mapping to your vocabulary.

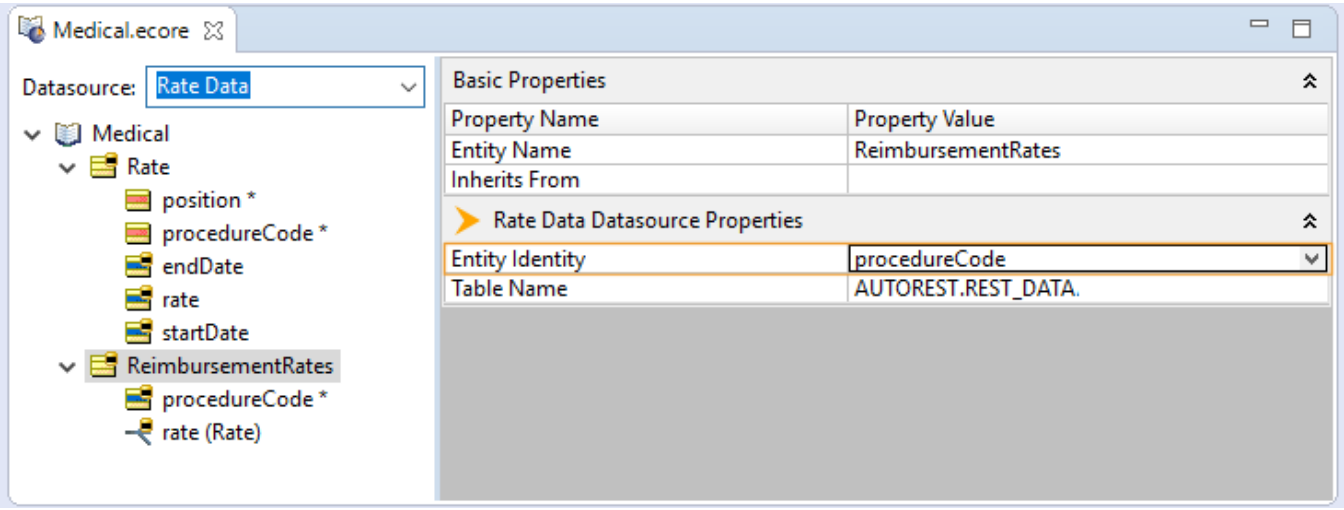
   ---

2. The sample has two entities `ReimbursementRates` and `Rate`, as illustrated:

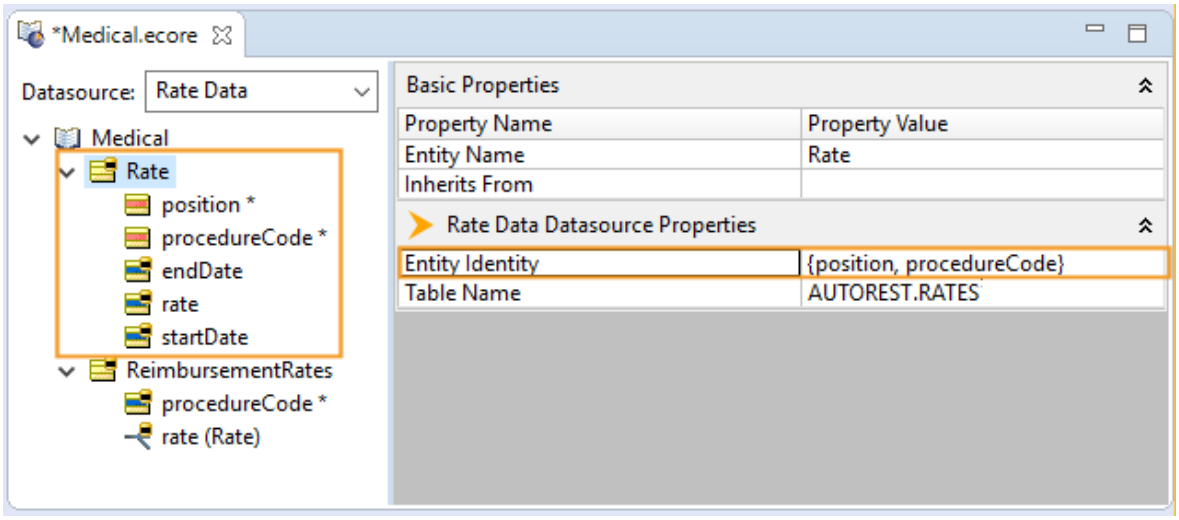


Each of these entities has already been mapped to the REST Datasource. Clicking on either entity will show the **Table Name** of the table the entity to which it is mapped.

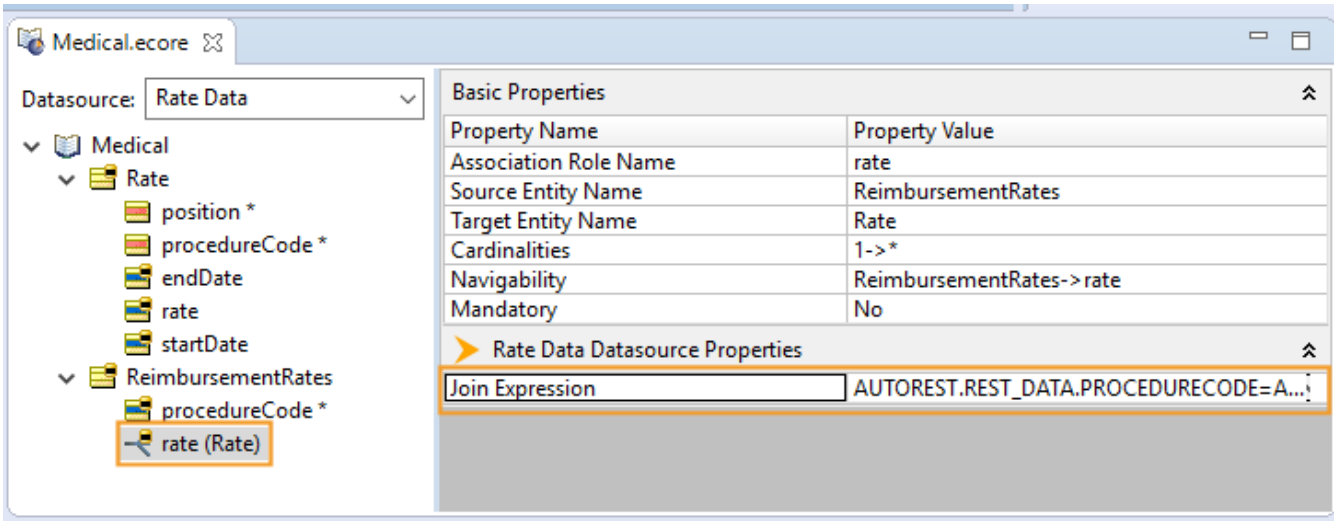
3. Specify the Primary Key (PK) in the table as the Entity Identity by clicking the **Entity Identity** Property Value to open its menu, and then choose from the listed attributes. Here there is only one attribute, `procedureCode`:



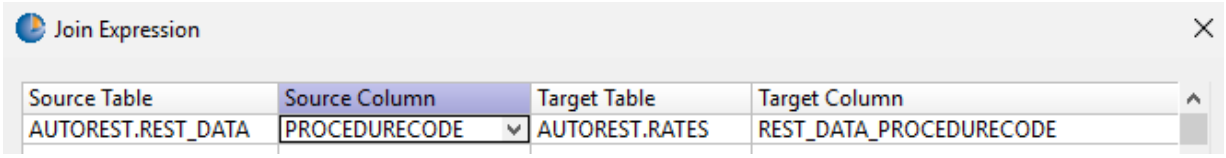
4. The sample's **Rate** entity is a JSON collection. The added attributes are added to uniquely identity the data's position inside the collection. Because there is no enforced uniqueness in the REST Datasource, two transient attributes are added to handle possible duplicates:



5. You specify its Primary Key (PK) in the database table as the Entity Identity by clicking the **Entity Identity** Property Value to open its menu, and then choosing both `position` and `procedureCode` from the listed attributes. The `position` attribute was created by the Autonomous REST Connector as a synthetic value that will ensure uniqueness of the primary key.
6. REST Data could have multiple rates that are an array of objects that Corticon will manage as a one-to-many association with the `ReimbursementRates procedureCode`



which is easier to see in the Join Expression dialog:



For more about REST data mapping, see Mapping REST Service metadata on page 173.

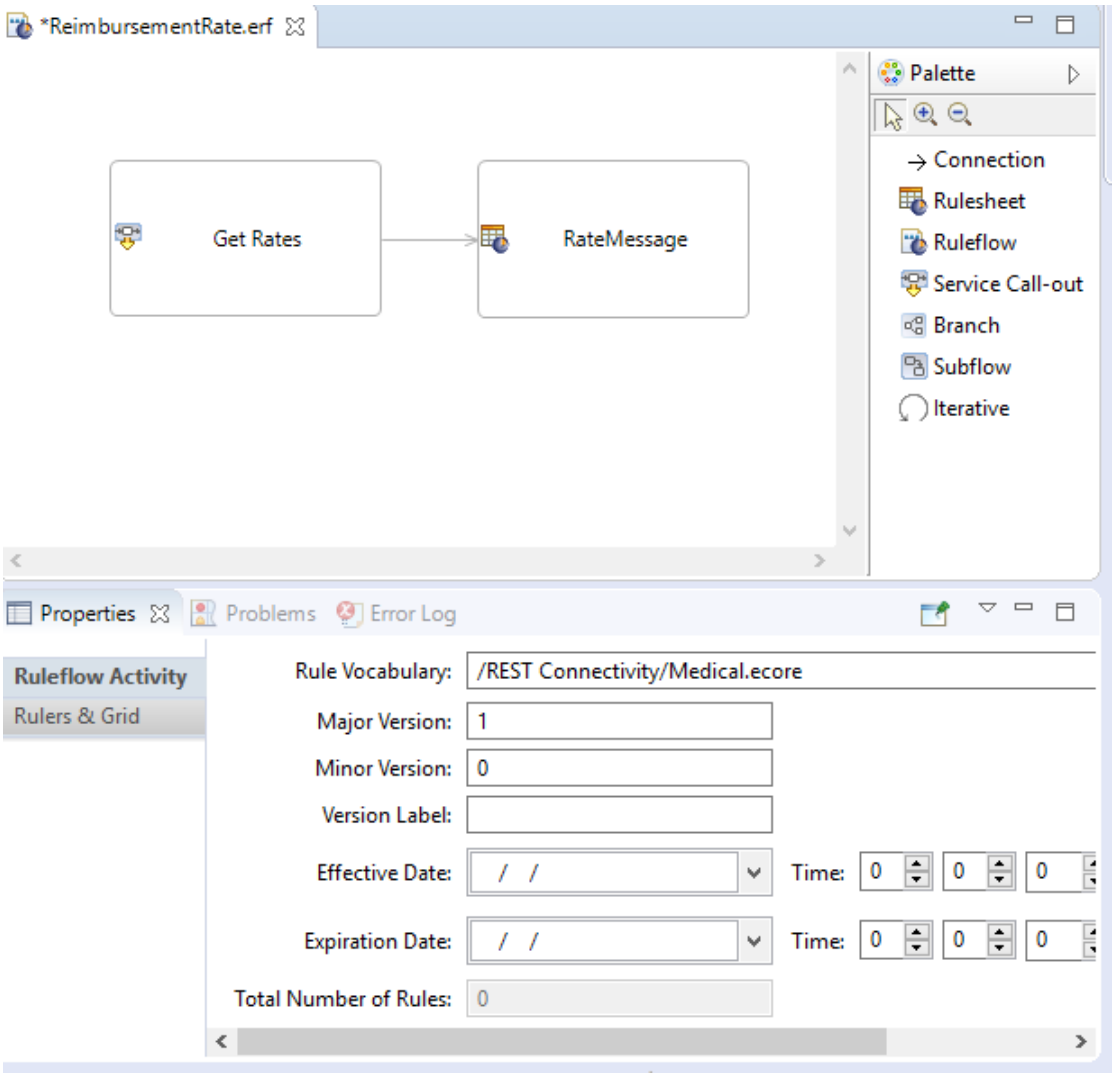
Use REST data sources in a Ruleflow

To use a REST data source, you need to add a call-out to it in your Ruleflow. The REST Connectivity sample already has it added. If using your own REST services, you would add call-outs where needed in your Ruleflow to access them. If you have multiple REST data sources, you can access those you need from your Ruleflow.

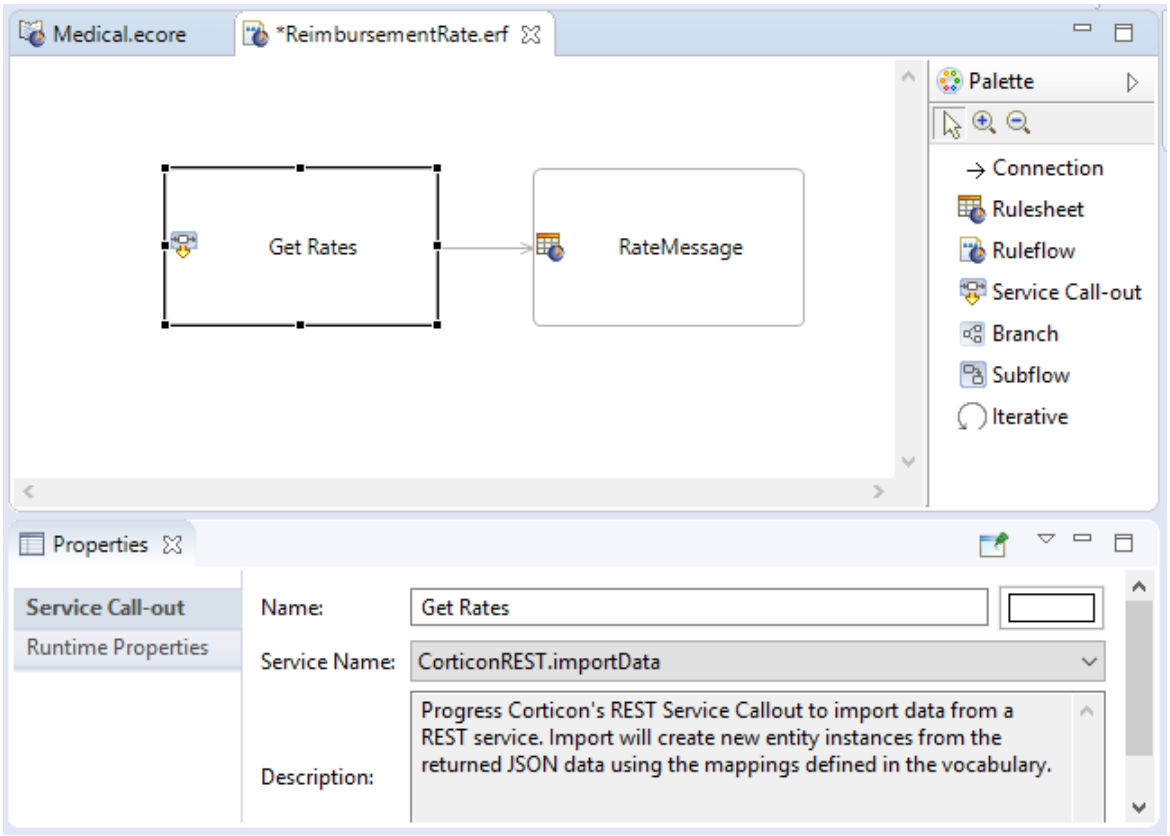
A Ruleflow could use several REST connectivity service callouts to enrich data before and after other processing steps. The sample uses one REST connectivity callout.

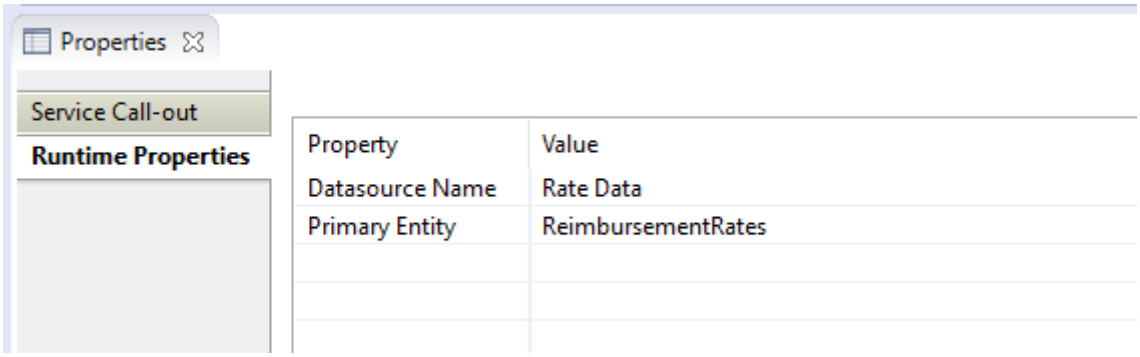
To use REST service callouts:

1. In a Ruleflow where you want to use a REST Service connection, add a Service Call-out to the Ruleflow canvas. In this example, the Ruleflow has a single call-out **Get Rates** to retrieve rate data from the REST service and a Rulesheet which produces rule messages from the rate data, as shown:



2. Click on the **Get Rates** object, and then, on the object's **Properties** tab.
3. On its **Service Call-out** tab, click the **Service Name** pulldown to select the read-only service you want for this use, `CorticonREST.importData` . The import will create new entity instances for the data returned. The alternative, retrieve, would update existing entity instances. The alternative, `CorticonREST.retrieveData`, will be used later in this sample run.
4. Click the **Runtime Properties** tab:



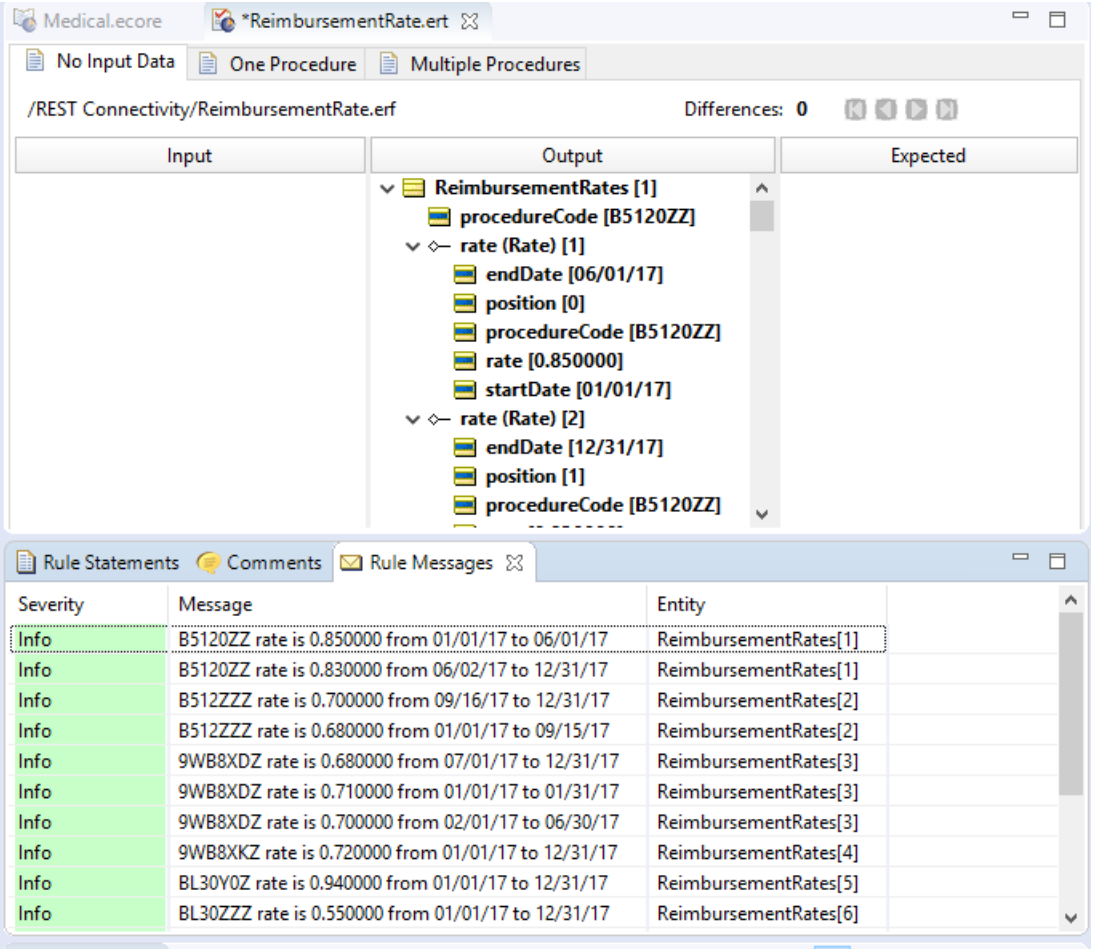


a. In the **Property** column, click on **Datasource Name**. Choose the Datasource that corresponds to the name of the appropriate REST Service definition. For the REST Service sample, choose **Rate Data**.

b. Select **Primary Entity**, and then, for its value, use the dropdown menu to select the appropriate Vocabulary Entity. For the REST Service sample, choose **ReimbursementRates**.

Test rules when importing from the REST Datasource

Open the sample's Ruletest `ReimbursementRate.ert`. This Ruletest has 3 Testsheets. The REST call-out in the Ruleflow is defined to do an **import** from the REST service. An import will create new entity instances for the data returned from the REST service. Select the **No Input Data** and run click **Run Test** to run it.



The Output for the Testsheet will contain a list reimbursement rates returned from the REST service. Where an **import** operation was performed, new entities were created for each reimbursement rate. To compare this to the JSON returned from the REST service, copy the REST URL from the Rate data source and paste it into your browser. The **One Procedure** and **Multiple Procedures** Testsheets have `procedureCodes` defined in the Input of the Testsheet. These Testsheets will be used in later sections to show how to perform a retrieve operation to get rates for specific procedure codes. If you run either of these Testsheets while the **Get Rates** call-out in the Ruleflow is configured to perform an import, you'll see the Output contains both the input data and new entities for all the rate data from the REST service.

Note: It is important to remember that an import will create new entity instances.

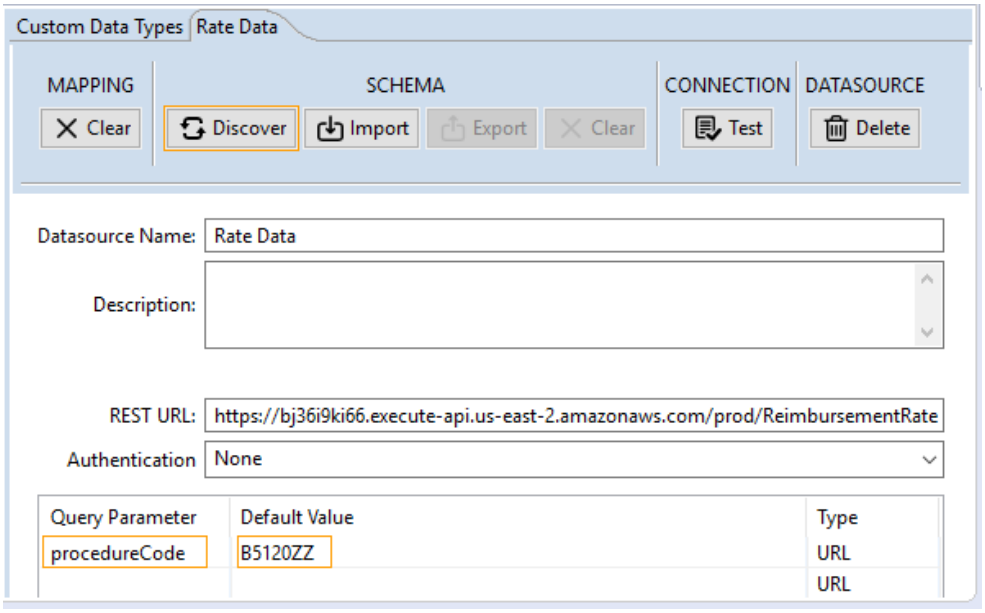
Revise Connection and Service Call-out to retrieve data

To update existing entities in the payload being processed by your rules, you need to change the REST call-out to perform a **retrieve operation**. When performing a retrieve operation your REST data source must have one or more query parameters to identify the instance of data required. For example, if your rules were processing mortgage applicants you might pass the social security number of an applicant to a REST service to get credit information about the applicant being processed. The REST service used by the REST Connectivity sample accepts the query parameter `procedureCode` to get rate information about a specific procedure. In this section you will convert this sample to perform a retrieve operation.

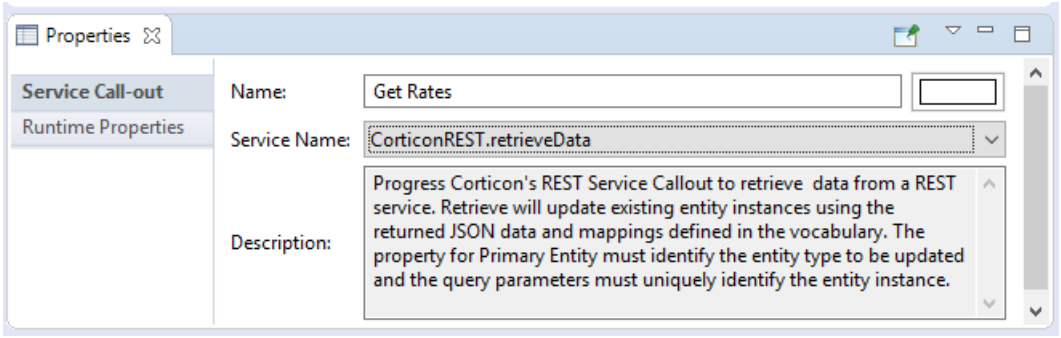
1. In the Vocabulary editor's root, click the **Rate Data** tab.
2. Click SCHEMA **Clear**.

Note: When changing the URL or adding query parameters you must first clear the existing schema.

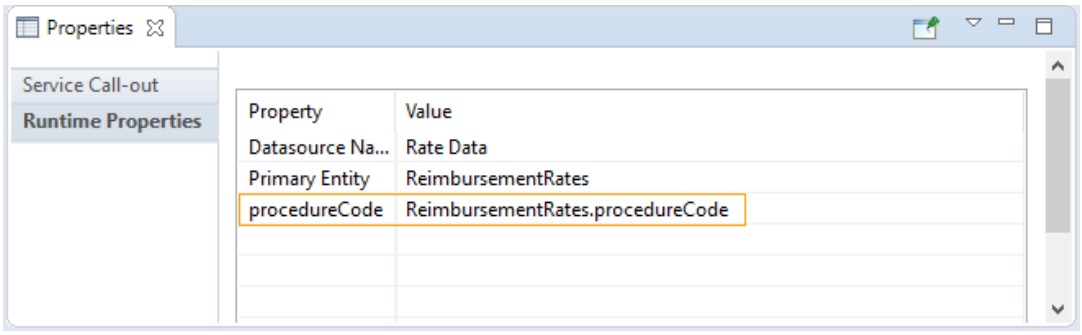
3. Enter the query parameter `procedureCode`, and then enter the value `B5120ZZ`, a known `procedureCode` value. This returns JSON representative of the JSON which could be returned at runtime.
4. Click SCHEMA **Discover**, as shown:



5. Open the Rulefllow editor, and then click the **Get Rates** object to access its **Properties** tab.
6. Click its **Service Call-out** tab, and then choose the **Service Name** `CorticonREST.retrieveData`:

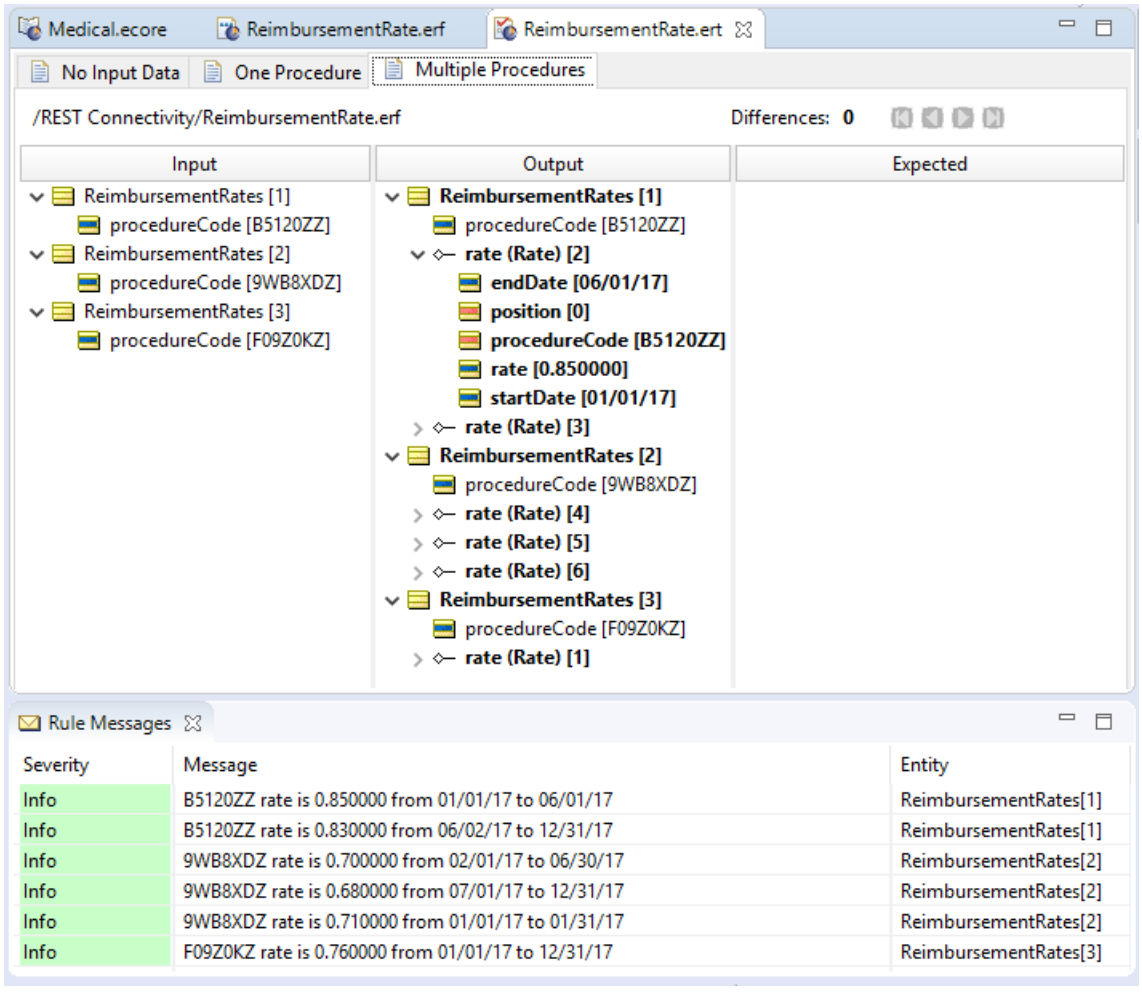


7. Click its **Runtime Properties** tab, and then add the **Property** `procedureCode` and the **Value** `ReimbursementRate.procedureCode`:



8. Open the Ruletest and run all tests.

The **Multiple Procedures** tab shows the specified procedureCodes and the related rates:



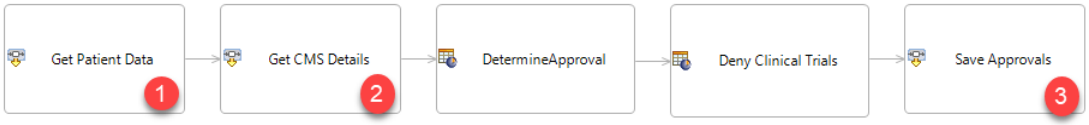
The `procedureCode` specified in **Input** where substituted as query parameter values and used to make calls to the REST service. For comparison, select the **No Input Data** Testsheet and click **Run Test**. No data shows in **Output**. This is because the `Get Rate Data` call-out is configured to do a retrieve, not an import, operation. Where no procedure codes where provided, there were no existing entities to update.

10

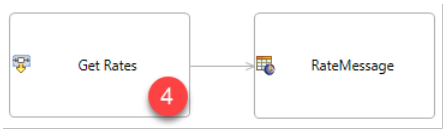
Mixing REST and database access

Often rules need to access data both in databases and via REST services. Corticon allows you to access both types of Datasources within a single Decision Service. This section expands on the **Multiple Database Connectivity** sample by adding the retrieval of reimbursement rates as done in the **REST Connectivity** sample, see these samples if not familiar with them.

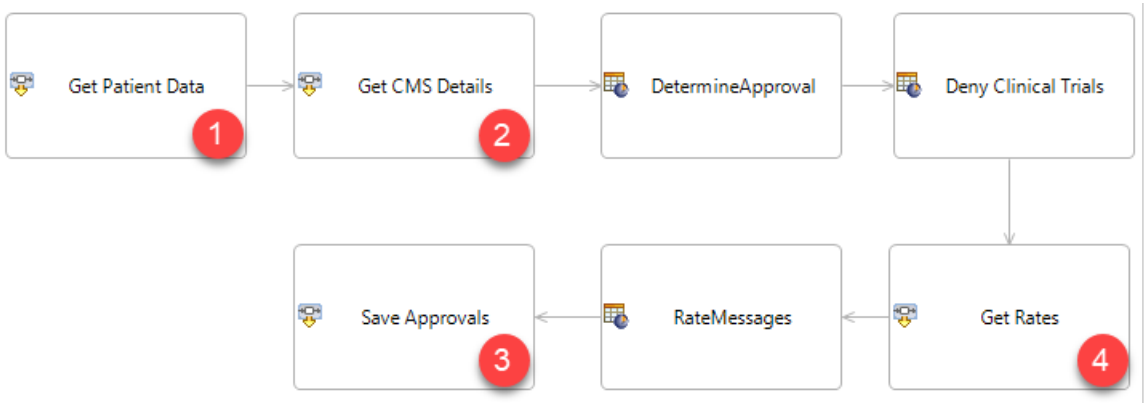
The Multiple Database Connectivity sample Ruleflow contains 2 steps to retrieve data from a database, (1) **Get Patient Data** and (2) **Get CMS Details,** and the step (3) **Save Approvals** to save approval decisions to a database.



The REST Connectivity sample ruleflow contains the step (4) **Get Rates** to retrieve reimbursement rates for a procedure code



The **Mixed Connectivity** sample combines these to create a ruleflow that accesses both databases and a REST service.



The steps in Mixed Connectivity are largely the same as in the other sample ruleflows with the one note being the Get Rates REST call-out has been configured to perform a retrieve operation – such that it updates existing entities

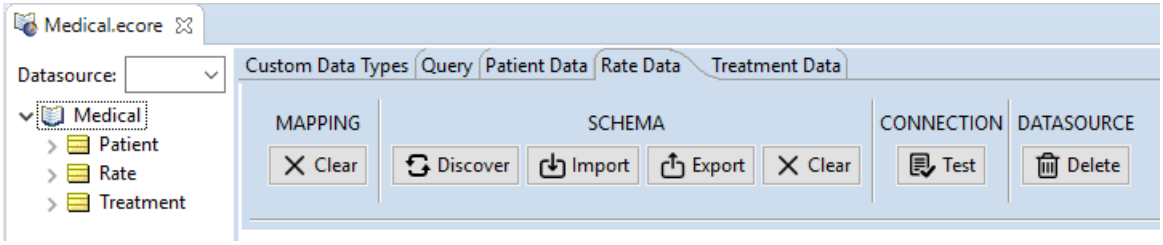
Data flow in the sample

In the ruleflow the interaction of call-outs is as follows:

- **Get Patient Data** retrieves information about a patient and the treatments they received from a database. Each treatment has a corresponding **medicalCode** identifying the treatment.
- **GET CMS Details** retrieves detailed information about a type of treatment from a second database by looking up the treatment using it's **medicalCode**
- **Get Rates** retrieves reimbursement rates for a type of treatment by querying a REST service, passing the medicalCode as a query parameter to get rates for a specific type of treatment. Note, the REST service takes the query parameter named **procedureCode**, the value of medicalCode is passed as the value for procedureCode.
- **Save Approvals** saves the decisions on which procedures are approved for each patient back to the patient database.

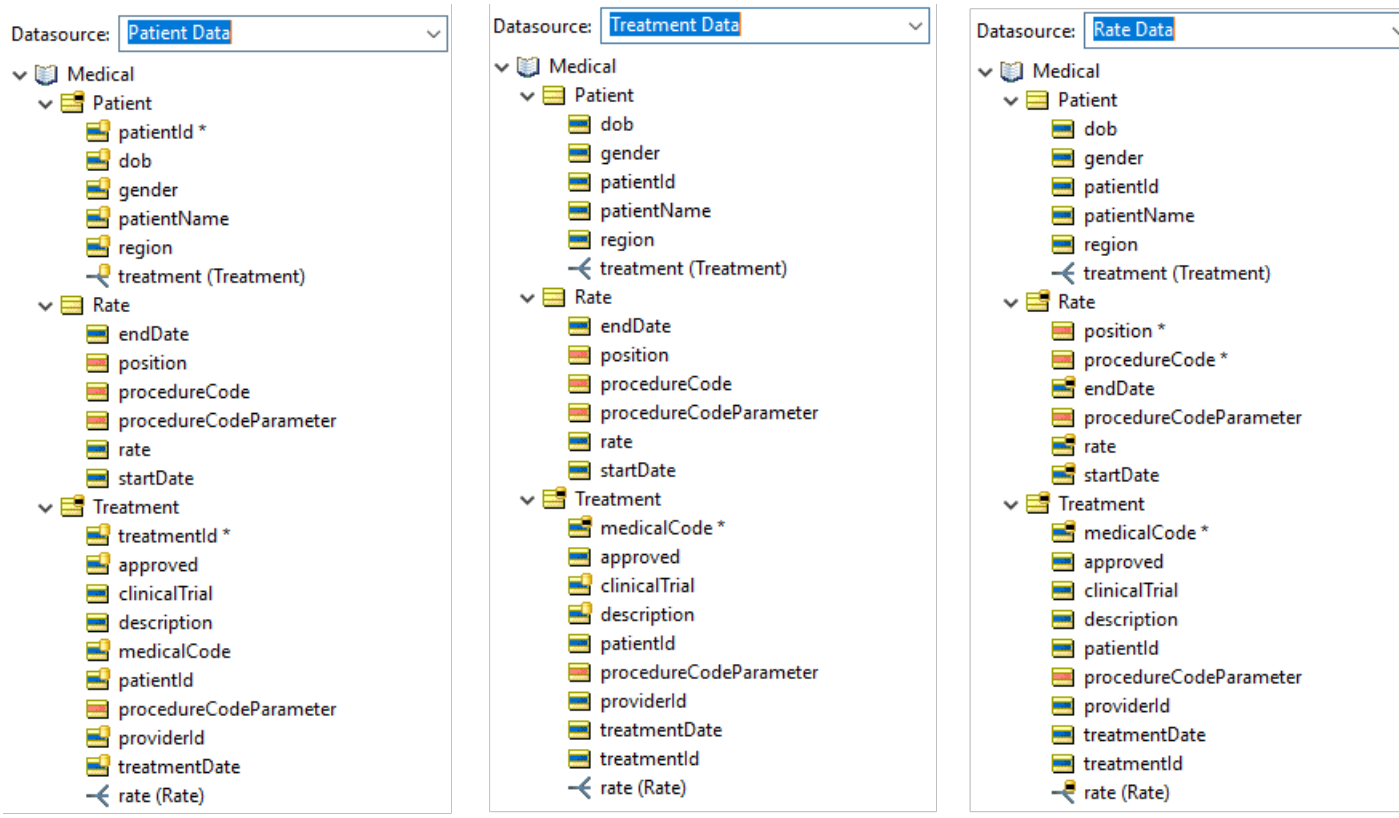
Explore the sample

1. Add the **Mixed Connectivity** sample to your workspace.
2. Open the vocabulary.

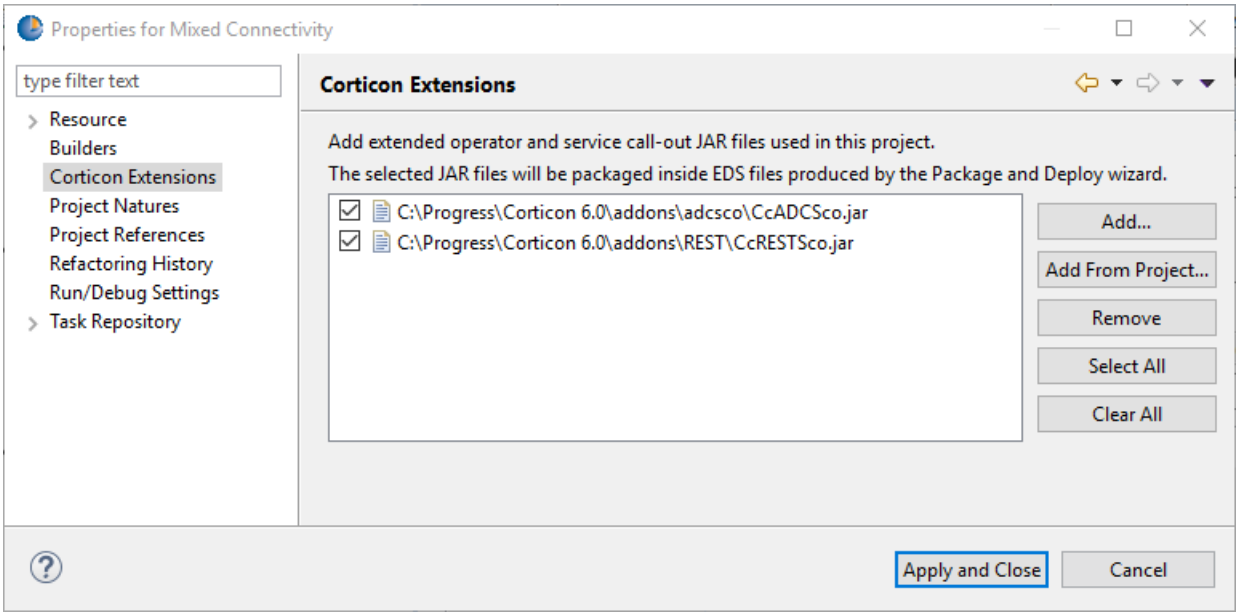


Note: Patient Data, Treatment Data and **Rate Data** Datasources are the same as in the earlier samples.

3. In the vocabulary tree, select each of these Datasources one at a time. This will show which entities and attributes are mapped to which Datasource. Corticon's data integration capabilities allow you to map a vocabulary to multiple Datasources.



4. Display the project properties, select Corticon Extensions.

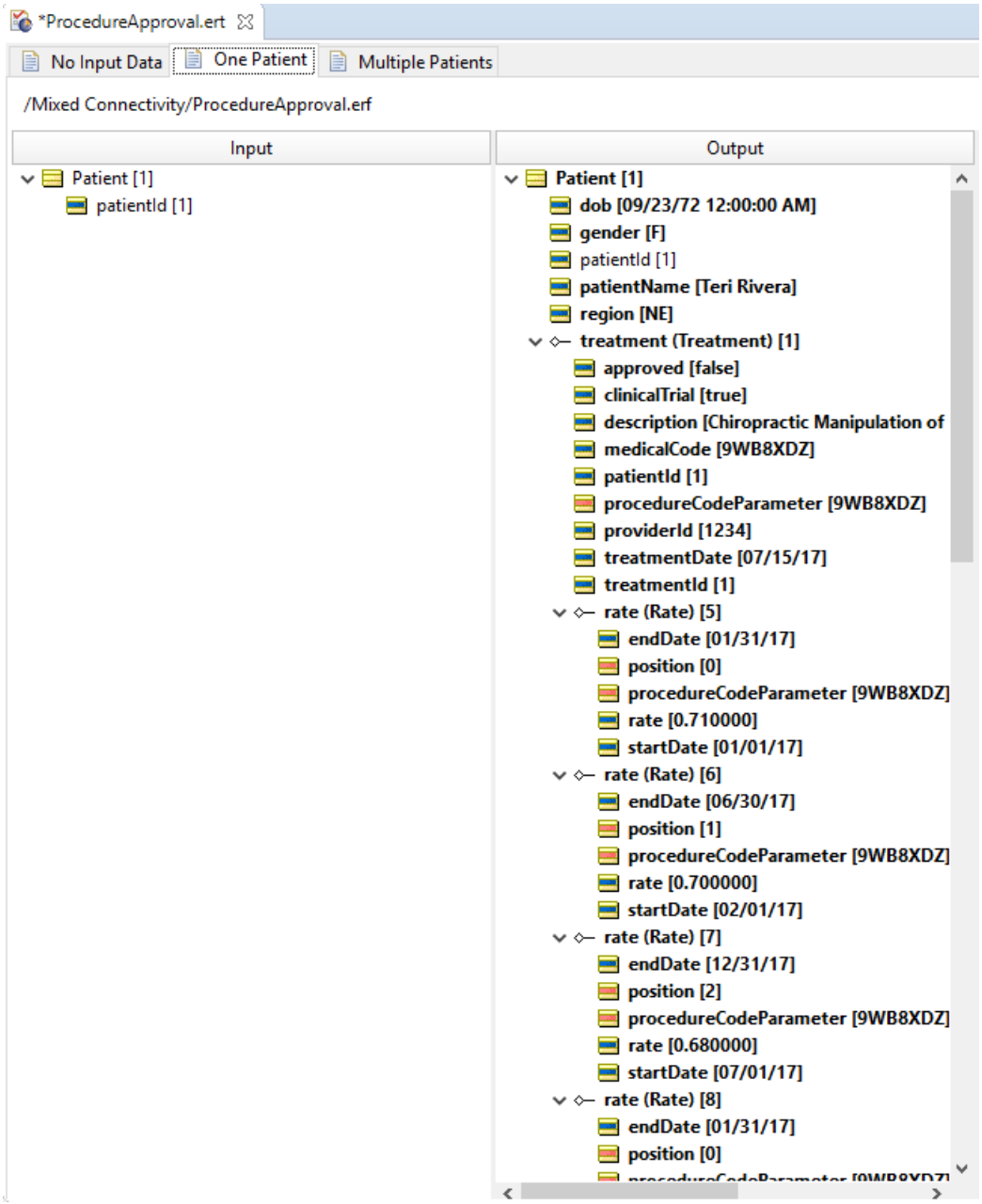


Note: This project uses both ADC and REST, the jar files for each are added to the project.

5. Open the Ruleflow and examine each callout to see to see the call-out performed and the Datasource used. Corticon allows a Datasource to be used multiple times in a Ruleflow. Each of the Datasources are used one or more times.

Run the sample

1. Open the Ruletest for the sample.
2. Open the **One Patient** Testsheet.
3. Click **Run Test.**



The Output shows the data retrieved from each Datasource and the results of the rule processing. Given just the patient ID, Corticon was able to retrieve data from multiple database and REST data sources and assemble it according to the vocabulary mappings to allow it to be processed by the rules.

11

Deploying projects that use data integration

The data integration samples you have worked with to this point in this guide have demonstrated and tested database connectivity entirely from the development environment. To move out of development and into production, you generate Decision Service files and corresponding Datasource Configuration files that will be positioned for servers to deploy them.

For details, see the following topics:

- Exporting the Datasource Configuration file
- Package a project in Corticon Studio for Corticon Server

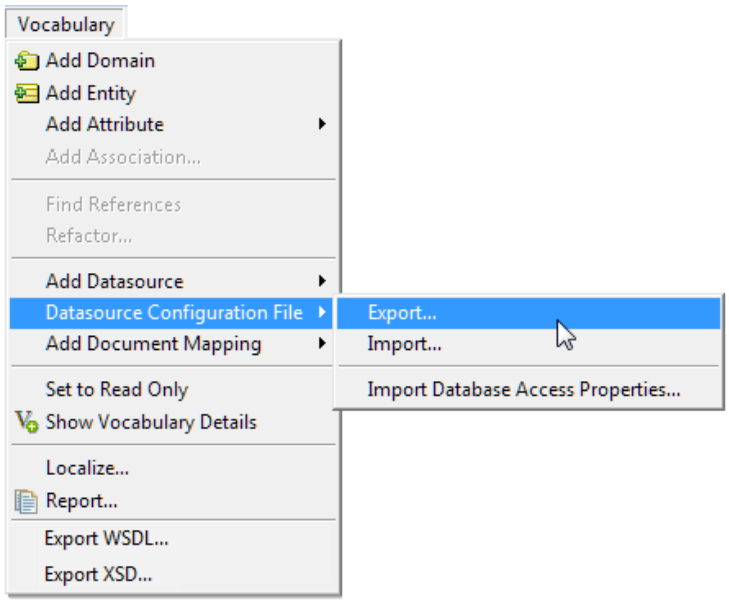
Exporting the Datasource Configuration file

When packaging a project on Corticon Studio for deployment as a Decision Service on a Corticon Server, a Datasource configuration file provides the database configurations and credentials that were used in the project.

Note: When you are running and testing in Corticon Studio, the Datasource configuration and mapping information is saved in the project's Vocabulary file. It does not need a Datasource Configuration file.

To generate a Datasource Configuration file:

1. With the project's Vocabulary open in its editor, select **Vocabulary > Datasource Configuration File > Export**, as shown:



2. All the defined connections to external data sources are packaged into a single file XML, typically named, `datasources.xml`. The file content might look like this:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<decisionService>
    <datasources>
        <database useKerberosAuthentication="true" name="ADC">
            <connection-url>jdbc:progress:sqlserver://localhost:1433;
                            databaseName=Cargo</connection-url>
            <database-driver>com.corticon.database.id.MsSql2014</database-driver>
        </database>
        <edc useKerberosAuthentication="false" description="" name="EDC">
            <connection-url>jdbc:progress:sqlserver://localhost:1433;
                            databaseName=PatientRecords</connection-url>
            <database-driver>com.corticon.database.id.MsSql2014</database-driver>
            <password>062046016058035029061039110</password>
            <username>061046</username>
            <properties/>
        </edc>
    </datasources>
</decisionService>
```

When `useKerberosAuthentication="true"`, neither the `username` or `password` parameter are included.

You can specify a preferred name and location for the file, although colocating it with its Decision Service file, or within its related project folder is a good idea.

Note: Because data integration carries the potential for data loss or corruption due to unintended updates, it is a good idea to use a test instance of a database whenever testing database-enabled Rulesheets and Ruleflows from Studio. Then, if unintended changes or deletions are made during rule execution, only test database instances have been changed, not production databases. Even when using test instances, you may want to restrict the ability to read and update connected databases to those users who understand the possible impact. For other rule modelers without a solid understanding of databases, you may want to provide them with read-only access.

There are several techniques for deployment, as described in the section *"Packaging and deploying Decision Services" in the Deployment section*. This section will focus on one, *"Using Studio to compile and deploy Decision Services" in the Deployment section*.

Managing user access on Corticon Server

Typically, enterprises constrain developers to appropriate database products. As data integration carries the potential for data loss or corruption due to unintended updates, developers are typically limited a test instance of a database when testing from Studio. If unintended changes or deletions are made during rule execution, then only test database instances have been changed. When deploying to Corticon Server, it is a good practice to have servers reserved for developer integration testing to the test databases through the user-acceptance test phase.

Managing database connections on Corticon Server

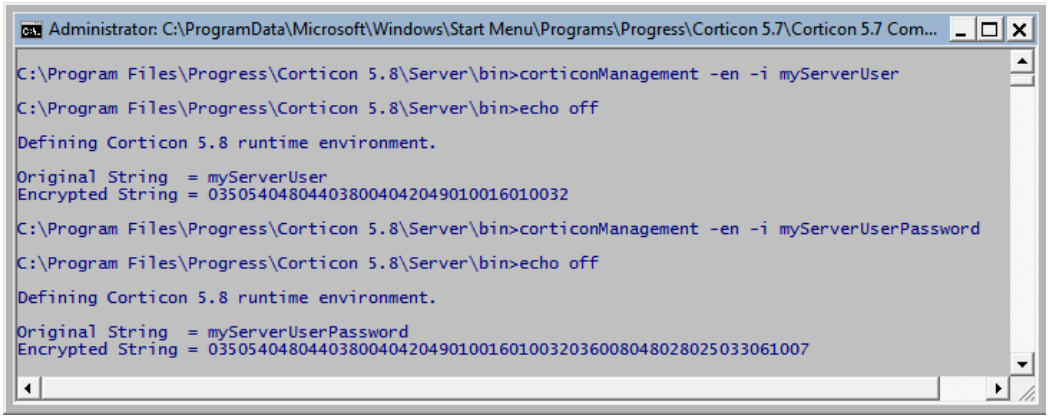
The handoff to production administrators will typically recast the Datasource configurations to the pre-production server locations and credentials followed by validation tests. Production might require another adjustment of the database configuration.

However, notice that the username and password values are very different from the credentials that were entered. These values were encrypted when the database access file was created and will be decrypted when they are implemented in a decision service. You can use the following utility to encrypt the required credentials.

Encrypting database credentials defined in the Datasource Configuration File

To use Corticon's proprietary encryption algorithm to encrypt credentials:

1. Obtain the credentials you want for each of the Datasources. Use those values to replace *myServerUser* and *myServerUserPassword* in the following procedure.
2. In a Corticon Server installation, open a Command Window at `[CORTICON_HOME]\Server\bin\`.
3. Type `corticonManagement -en -i` *myServerUser*. An encrypted String for the *username* is output. Then type `corticonManagement -en -i` *myServerUserPassword*. An encrypted String is output. The procedure looks like this:



4. Copy the encrypted String to the appropriate Datasource in the Datasource configuration file and enter it as the value for the username. Then similarly copy the encrypted password String as the value for the user's password.
5. Colocate the revised Datasource configuration file with the appropriate instance of the Decision Service on the Corticon Server.

Note: When using CDD deployment, the database access properties file is identified within the CDD file. When using the Web Console or APIs for deployment you specify the file at the time of deployment.

Using the Datasource XSD file

Every Corticon Studio and Corticon Server installs an XSD file that specifies how to formally describe the elements in Datasource configuration XML files. You can use this file to verify the content in your documents. The file `datasourceConfig.xsd` is located at `[CORTICON_HOME]/Studio/lib`.

Package a project in Corticon Studio for Corticon Server

There are several techniques for deployment, as described in the section *"Packaging and deploying Decision Services" in the Deployment section*. This section will focus on one, *"Using Studio to compile and deploy Decision Services" in the Deployment topics*.

To make the project into a Decision Service and stage it to a server:

1. In Corticon Studio's Project Explorer, right-click on **ADC Database Connectivity**, and then choose **Package and Deploy Decision Services**. In its dialog, choose **Package and save for later deployment**, and then save it as `ProcedureApproval_v1.1.eds` where the server will be able to access it.

   (See *"Using Studio to compile and deploy Decision Services" in the Deployment Guide*.)

2. In the ADC project's Vocabulary, choose the menu command **Vocabulary > Datasource Configuration File > Export**, and then save it as `ADC_Sample_Config.xml`, typically colocated with the Decision Service file you just created.

   (See Exporting the Datasource Configuration file on page 89)

12

Getting Started with Batch

Batch processing is a server-based function that provides additional power to external data source functionality. Elevating an external data source solution to a batch job means that Corticon Server can take several `patientIds` from the database's `Patient` table at once and pass them in as a set to the Decision Service for processing – a very efficient way to perform processing – where the Decision Service retrieves additional data for each request from the Datasource for rule conditions and to enrich the record in a relational database with results. You get greater control over queries and insert statements that are used. This is beneficial when you need finer control for performance or need to retrieve large amounts of data.

How batch processing works - Fetching the transaction identifying data from the underlying data source that will be injected into the rules engine – *seed data retrieval* – takes place outside the Decision Service. As such, Decision Service requests that are usually individual transactions are instead fetched in chunks for the rules engine, and then dispersed across multiple processing threads to concurrently process the incoming requests. Batch processing produces no return payload per request – however, the result of each rule processing is persisted in a relational Datasource.

The Datasource for a batch processing is typically to a relational database through an ADC or EDC connection. You can use a REST service to read data that enhances the data; however, REST latency could compromise your batch processing's performance requirements. ***Try it!*** Use the Mixed Connectivity sample in a batch scenario. For each batch item, all data to drive the sample comes from a database, while the Ruleflow calls a REST service to get the rate data.

Note:

For this sample, you must have:

1. Completed the steps in Getting Started with ADC on page 37 that:
   a. Run the scripts `patient` and `adc` that set up the database with the schema and data.
   b. Imported the metadata into the Vocabulary.
   c. Set the Ruleflow's **Get Patient Data** query to `IndicatedPatients`.
2. Packaged and saved the Decision Service file and its Datasource Configuration file where the server will be able to access them.
3. Installed Corticon Server with the server and Web Console options, and then started the server.

To load the batch sample:

In Corticon Studio, choose the menu item **Help > Samples**. Select the Advanced Sample **Batch Rule Processing**, and then click **Done**. Follow the **Import** dialog to bring the sample into your workspace. The batch sample is just two SQL queries that will be used later in this topic, yet that step completes setup.

Note: For details on defining and running batch processes, see the section *"Batch Configurations" in the Web Console Guide.*

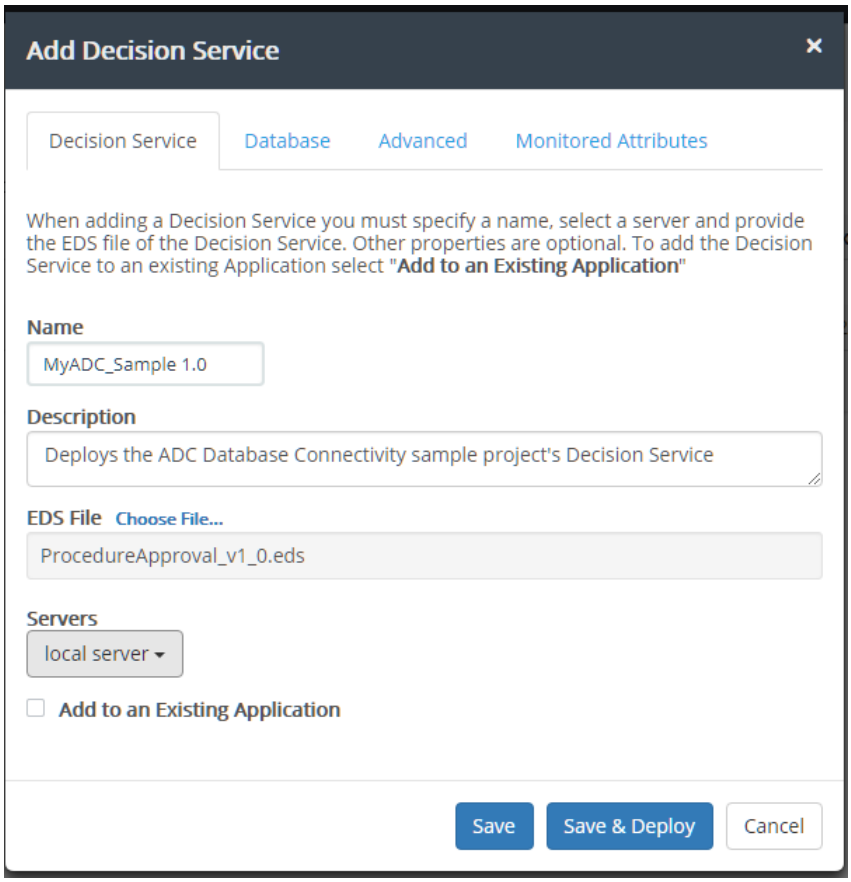
Batch configuration depends on the Server where the Web Console connects to have access to::

- A Decision Service that has Ruleflow Service Callouts that reference the databases and queries
- The Decision Service project's Datasource Configuration
- `BATCH_READ` queries loaded in the query service database

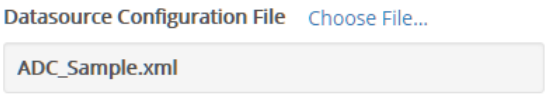
To run the batch sample:

1. In your browser, connect to the Web Console, typically `http://localhost:8850/corticon`, using the default credentials `Admin/Admin`.
2. In the Web Console, choose **Decision Services**, and then click **Add Decision Service** for just a single Decision Service. In the dialog box:
   a. Name your service. For example, `myADC_Sample 1.0`.
   b. Choose the Decision Service file you created.
   c. Choose its server location.

The settings will look similar to this:



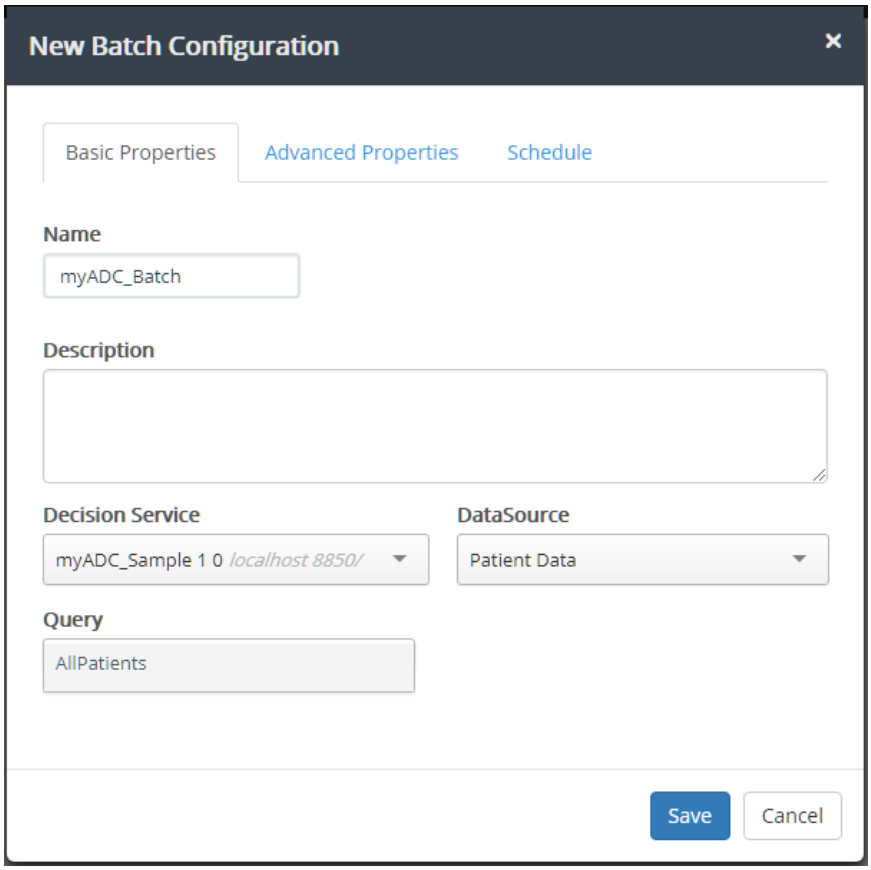
d. On the **Database** tab, choose your Datasource Configuration File.



e. Click **Save and Deploy**.

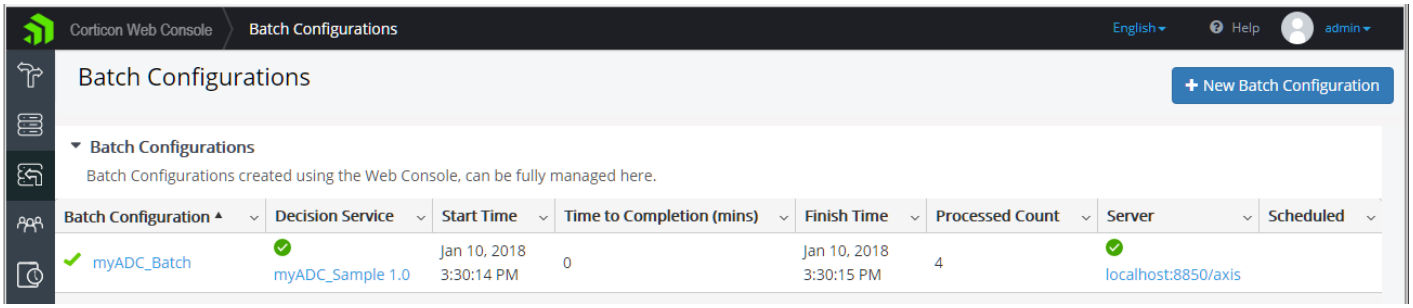
3. Choose **Batch Configurations**, and then click **New Batch Configuration**. In the dialog box:
   a. Name the batch configuration. For example, `myADC_Batch`.
   b. Choose your deployed Decision Service.
   c. Choose its Datasource. This is a specific database name within the Datasource configuration file. (See Exporting the Datasource Configuration file on page 89)
   d. Choose the **AllPatients** query.

The settings will look similar to this:

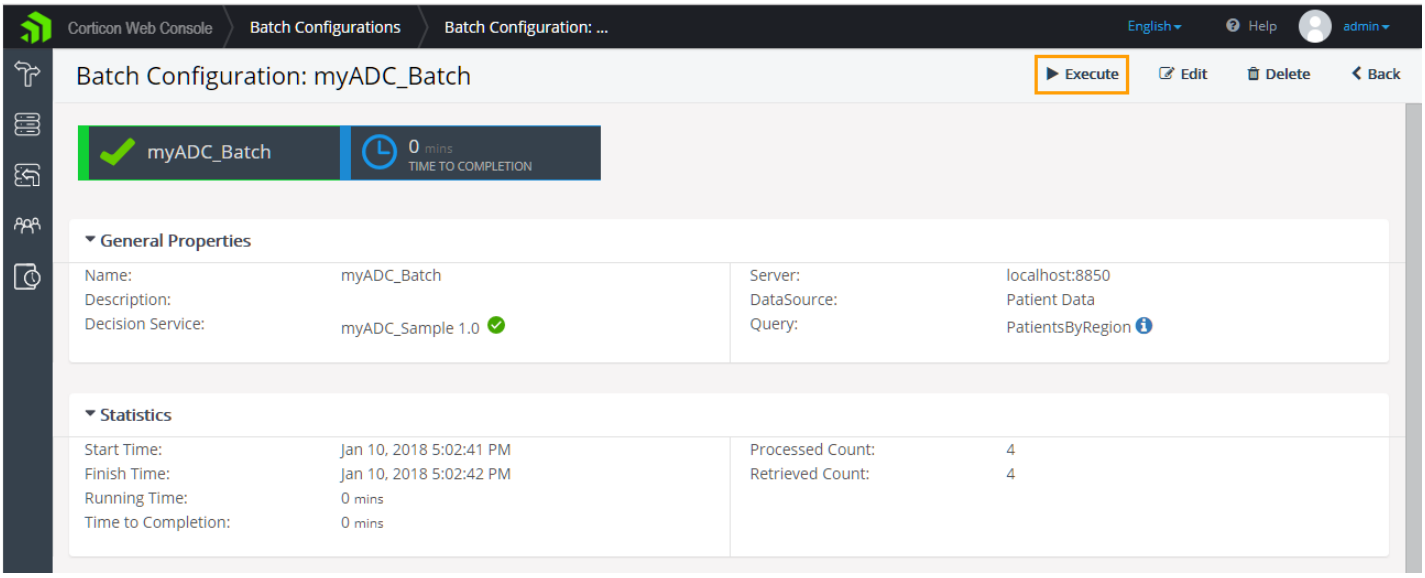


e. Save the batch configuration.

4. The Batch Configuration page lists the new configuration:



Run the batch job by clicking on its Batch Configuration name, and then clicking **Execute**:



5. On the Batch Configuration page detail page you can see the Processed Count each time you run it.
6. Test that the database is getting updates by looking at the **Treatment** table's `Approved` column.

Note: Clearing sample writes - To reset the approval values written to the database **Treatment** table to *NULL*, run the Batch sample's `clear_approved` script in your DB management tool.

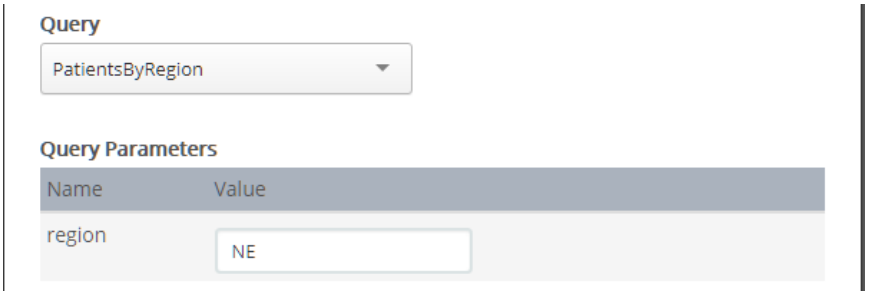
Parameters in queries

Batch configurations can access queries that are defined to request parameter values. The `BATCH_READ` query `PatientsByRegion` runs the query:

```
SELECT patientId FROM Patient WHERE region IN ({Patient.region})
```

To run a parameterized batch job:

1. Create batch configuration named **MyADC_Batch_Regional**. Select the Decision Service you deployed, its Datasource, and the query **PatientsByRegion**. That adds another field to accept the values of your **Query Parameters**. Enter the value `NE`. The settings will look similar to this:



2. Save and then run this batch job. When you inspect the Treatment table you see that only patients from the `NE` region have been processed.

Note: For more about the format of Corticon's queries, see How Corticon is expressed in SQL on page 163.

Creating additional sample records in the database

As the sample set of patient/treatment records is very small, the efficiency of rapid batch processing can be difficult to observe. A SQL utility is provided that will generate `PATIENT_COUNT` additional records for your tests. It is a good idea to also adjust the `FIRST_PATIENT_ID` and `FIRST_TREATMENT_ID` so that they are not overwriting each time you execute the utility

To create large data sets and run large batch tests:

1. Open the script in the Batch sample's `generate_patients` script in your DB management tool.
2. Change the values for:

```
SET @PATIENT_COUNT=1000
SET @FIRST_PATIENT_ID=1000
SET @FIRST_TREATMENT_ID=1000
```

   These set the number of patients you want generated, and the starting ids for patients and treatments.

3. Run the batch job with this newly generated data, and then look at the Decision Service page to see the counters and charts.

13

A closer look at how Corticon relates to Datasources

All Corticon's data integration techniques enable sophisticated interaction with database architectures.

For details, see the following topics:

- SmartMatching of Vocabularies to databases
- Validation of names against SQL keywords and database restrictions
- Support for catalogs and schemas
- Fully-qualified table names
- Filtering catalogs and schemas
- Support for database views
- Associations as join expressions

SmartMatching of Vocabularies to databases

Corticon's Vocabulary binds to database metadata, and then stores the database connection and database metadata (tables, columns, primary keys, and foreign keys) that Corticon loads into its working memory as CDOs for use in rule execution. Corticon attempts to infer the **'SmartMatch'** for database table names, column names and related information such as the association join expressions. When a value is inferred this way, that value is not stored in the model; rather, the system dynamically infers the derived value whenever the Vocabulary Properties table is refreshed.

You can override the inferred value by choosing an explicit value from a drop-down list, or by entering a value manually. The explicitly-specified value is displayed in black font if it exists, otherwise it displays in orange. Corticon displays inferred properties in light gray font to distinguish them from explicitly-specified values.

You should favor inferred values whenever possible, because these values are automatically updated as database metadata evolves.

Table 1: Corticon inference rules for SmartMatching

Vocabulary property	Database element	Inference rules for mapping metadata to the Vocabulary
`Entity`	`Table`	Derived from table metadata. The first table located in database metadata that matches the entity name (ignoring case) is chosen. This matching process ignores catalog, schema and domains. The inferred value is displayed as a fully-qualified name including catalog and schema, if applicable.
`Attribute`	`Column`	Derived from first column in database metadata that matches the attribute name (ignoring case). For this purpose, the Table Name (whether explicitly-specified or inferred) is used.
`Association`	`Join Expression`	Complex derivation algorithm involving table data, column data, primary key and foreign key definitions. The algorithm attempts to find the best matching join expression that defines the relationships between database columns, typically along the lines of foreign keys.

Validation of names against SQL keywords and database restrictions

Commercial databases, such as Microsoft SQL Server and Oracle, use specific words for defining, manipulating, and accessing databases. These reserved keywords are part of the grammar used to parse and understand statements. Do not use database reserved words for Corticon Entity, Attribute, and Association names when creating the schema in Corticon. Your database support pages list reserved words -- for example, SQL Server 2012 -- that you should review as you prepare your Vocabulary for enterprise data connection.

Corticon makes a best-effort to validate the names against the SQL keywords against the database restrictions for column and table naming (such as length of a table name), and then validates generated column names (such as Foreign Key (FK) columns) against SQL keywords and table/column name restrictions.

Note: It is good practice to ensure that database columns not have hyphens, spaces or other special characters (even though some databases and SQL parsers allow them). The generally accepted valid values are all alphanumeric characters and the underscore character. It is a plus to use all-lowercase names to avoid platform case inconsistencies. For more information on Corticon's accepted names, see the topic *"Vocabulary node naming restrictions" in the Quick Reference Guide.*

Support for catalogs and schemas

Catalogs and schemas refer to the organization of data within relational databases. Data is contained in *tables*, tables are grouped into *schemas*, and then schemas are grouped into *catalogs*. The concepts of *schemas* and *catalogs* are defined in the SQL 92 standard yet are not implemented in all RDBMS brands, and, even then, not consistent in their meaning.

For example, in SQL Server, tables are grouped by owner and catalogs are called *databases*. In that case, a list of database names is filtered by a *Catalog filter*, and a list of table owners is filtered by a *Schema filter*. The owner of all tables is typically the database administrator, so if you do not know the actual owner name, select '**dbo**' (under SQL Server or Sybase), or the actual name of the database administrator.

Note: The term *schema*, as used in Corticon's **Import Database Metadata** feature, does not refer to the 'schema objects' that the mapping tool manipulates.

Fully-qualified table names

Whenever table names appear in properties, Corticon uses fully-qualified names; thus, a table name may consist of up to three nodes separated by periods. The JDBC specification allows for up to three levels of qualification for a table name -- Catalog, Schema, and Table.

For databases that support all three levels of qualification, table names take the form:

```
<catalog>.<schema>.<table>
```

Microsoft SQL Server uses all three levels of qualification. For example, `PatientRecords.dbo.Patient`

Others, such as Oracle, do not use Catalog Name, using only schema and table. For example, `corticon.Patient`. Corticon can infer which levels of qualification are applicable by checking for null values in database metadata.

Filtering catalogs and schemas

Once a database connection to a running database instance is established, the Database Metadata can be imported into the Vocabulary by clicking the METADATA **Import** button on the database tab. Then, the **class:table** property of each Entity is populated with a list of the fully qualified names (`catalog.schema.table` or `schema.table`) of all tables in the database.

Typically, it is a good idea to filter the metadata to specific database catalogs and/or schema.

Note: Whether or not these settings actually do filter the list depends, in part, on the type of database used and/or the JDBC driver used. For example, Oracle JDBC drivers do not honor these filters. However, the **Import Database Metadata** feature does apply a second layer of filtering beyond the JDBC driver to minimize the amount of metadata imported.

The ability to map entities to fully qualified table names makes it possible to map a single Vocabulary to more than one catalog/schema provided they are all accessible through the same JDBC connection.

When tables are generated to the database, they will use the default Catalog/Schema unless you specify otherwise.

Support for database views

Many RDBMS brands support *views*, a virtual table that is essentially a stored query. Your database administrator might have set up views to:

- Combine (JOIN) columns from multiple tables into a single virtual table that can be queried
- Partition a large table into multiple virtual tables
- Aggregate and perform calculations on raw data
- Simplify data enrichment

It is common practice to constrain staff users to accessing *only* views in their database connection credentials.

Corticon's Enterprise Data Connector supports mapping a Vocabulary to an RDBMS view.

Using Associations

When Corticon Entities are mapped to View tables that were created without any `WHERE` clause in the Select statement (in other words, Corticon filters are NOT applied), Associations (in a View table) are not required as the Entities mapped to the View tables with no Join Expressions in the Vocabulary returns the expected results that include the Association.

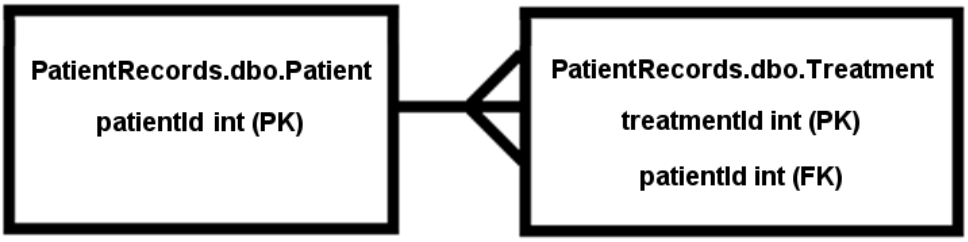
Note: When Entities are mapped to View tables that were created *with* a `WHERE` clause in the Select statement (in other words, Corticon filters *are* applied), results are incorrect: Associations are required even when there is a View table for the Join Expressions. Attempts to map the View tables to the Entities in the Vocabulary will generate validation warnings for lost Join Expressions. A Join Expression currently cannot be mapped to its related View tables.

Associations as join expressions

Each association in a Corticon Vocabulary will have a join expression that is used to establish the relationships between matching columns in the database. The syntax is similar to the SQL `WHERE` clause and are illustrated here by examples.

One to Many Association with Single Primary Keys

The samples in this guide have a bidirectional one-to-many relationship between tables:

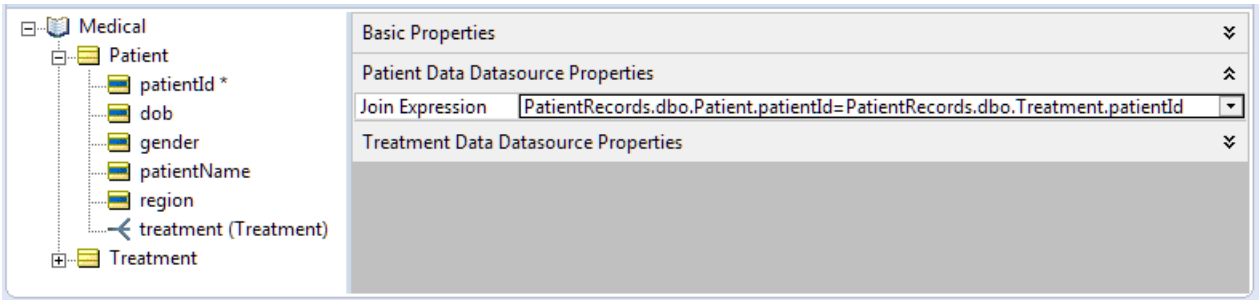


`PatientRecords.dbo.Patient` has the integer primary key `patientId`, and `PatientRecords.dbo.Treatment` has `treatmentId` as its primary key. `PatientRecords.dbo.Treatment.patientId` is a foreign key that "points" to primary key `PatientRecords.dbo.Patient.patientId`. In such case the join expressions would be as follows:

Vocabulary Association	Join Expression
`Patient.treatment`	`PatientRecords.dbo.Patient.patientId = PatientRecords.dbo.Treatment.patientId`
`Treatment.patient`	`PatientRecords.dbo.Treatment.patientId = PatientRecords.dbo.PatientId.patientId`

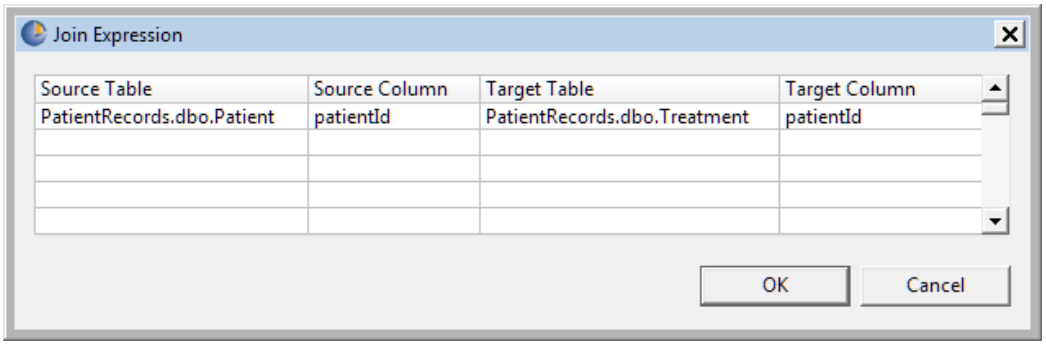
Note that in a bidirectional association, the two join expressions are mirror images of one another. Unlike ANSI SQL, the order of operands in the join expression is significant.

In Corticon Studio, the sample association mapping in the data integration samples is:



A closer look at the expression shows the correct inferred join expression for the `Patient`:

Clicking opens the **Join Expression** dialog for the connection, as shown:



If you want to revise the expression, each field entry area opens a dropdown menu for that field where you can choose a value that is in the scope of the connection, or clear the value. You can also just click in the box and enter a value.

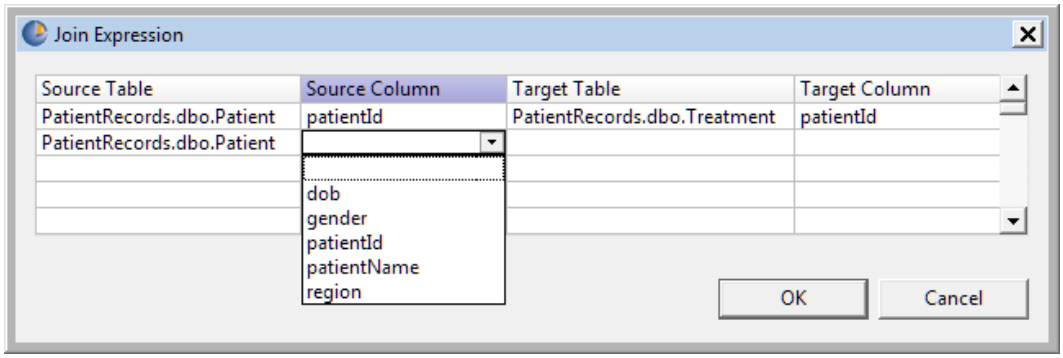
One to Many Association with Multiple Primary Keys

Consider the sample as having a multi-column primary key. All key columns must be specified in the join expression; in such case, the join expression becomes a set.

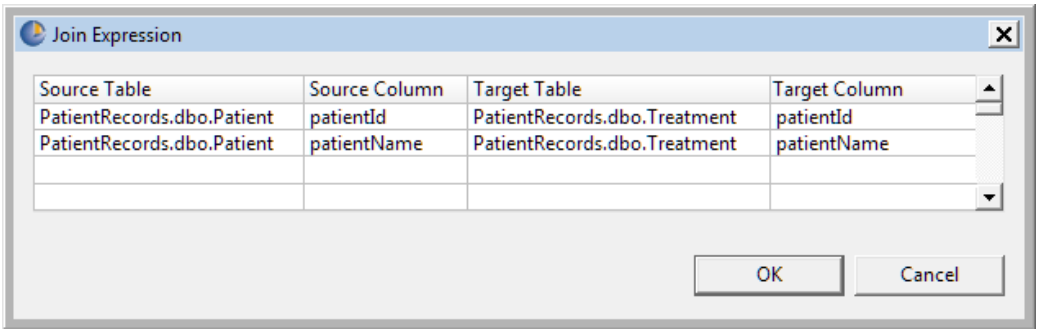
This is a one-to-many, bidirectional association between `PatientRecords.dbo.Patient` and `PatientRecords.dbo.Treatment`, where both have multi-column primary keys (`patientId, patientName, treatmentId, PatientCode`), and `PatientRecords.dbo.Treatment` also has multi-column foreign key (`Treatment.patientId, Treatment.patientName`). The join expressions would be as follows:

Vocabulary Association	Join Expression
`Patient.treatment`	`{ PatientRecords.dbo.Patient.patientId = PatientRecords.dbo.Treatment.patientId, PatientRecords.dbo.Patient.patientName = PatientRecords.dbo.Treatment.patientName }`
`Treatment.patient`	`{ PatientRecords.dbo.Treatment.patientId = PatientRecords.dbo.Patient.patientId, PatientRecords.dbo.Treatment.patientName = PatientRecords.dbo.Patient.patientName }`

Note the braces surrounding the comma-separated relational expressions, denoting that in this case, the join expressions are sets. To extend the `Patient` association to enter the multiple keys, add another line in the **Join Expression** dialog box, as shown:



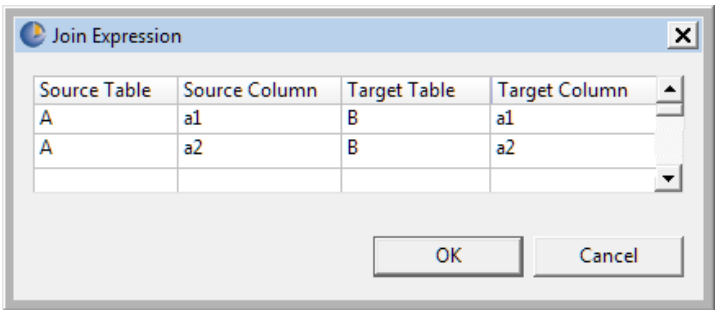
In this case, the join expression was extended as shown:



When you click OK, the expression is constructed as shown, a bit hard to read yet exactly as described:

```
{PatientRecords.dbo.Patient.patientId=PatientRecords.dbo.Treatment.patientId, PatientRecords.dbo.Patient.patientName=PatientRecords.dbo.Treatment.patientName}
```

Let's create the same construct using simple tokens:



The resulting join expression is:

```
{A.a1=B.a1, A.a2=B.a2}
```

The braces surround the comma-separated relational expressions: The join expressions are sets.

Best effort at inferring Join Expressions

Because join expressions are cumbersome to enter, it is crucial that Corticon have the best possible logic for automatically inferring them from metadata. For one-to-many associations, the join expression can frequently be inferred from primary and foreign key metadata, assuming that the entities can be successfully mapped to particular tables, and the foreign key relationships between those tables are properly declared. Exceptions to this rule include:

- Unary one-to-one associations (that is, *self-joins*), where it is impossible to infer which side of the association corresponds to the primary or foreign key
- Unary many-to-many associations, where it is impossible to infer which of the join table foreign keys should be used for each side of the association
- Tables that have multiple foreign key relationships between them with different meanings for each.

Corticon recognizes when it is not possible to unambiguously infer the proper join expression, and allow the user to choose from a set (drop-down list) of choices.

Corticon infers the join expressions in all cardinalities.

14

Advanced EDC Topics

This section describes advanced setup and operational functions of Corticon's Enterprise Data Connector.

For details, see the following topics:

- Setting EDC Vocabulary properties
- Mapping and validating EDC database metadata
- Setting additional EDC Datasource connection properties
- How data from an EDC Datasource integrates into rule output
- Using caches in EDC for entities and queries
- Metadata for Datastore Identity in XML and JSON Payloads
- Relational database concepts in the Enterprise Data Connector
- How EDC handles transactions and exceptions

Setting EDC Vocabulary properties

Note: Database, Datasource, Datastore - These terms are used throughout this document. **Database** is an external installation of a relational database management system that provides structured tables and accepts connections. **Datasource** is Corticon's term for a database that has a connection defined and tested through a Vocabulary. **Datastore** is generic term for an external repository; however, in Corticon's EDC settings, the term refers to a table in a Datasource that is mapped to an Entity in Corticon.

Editing Entity EDC properties

When an EDC Datasource has been added to the Vocabulary, its Entity properties for database interaction are displayed.

Note: The basic and document mapping Entity properties are discussed in *"Entity nodes: Adding and editing entities and their properties" in the Quick Reference Guide.*

Table 2: Enterprise Data Connector (EDC) Entity Properties

Property	Description	Applicability	Values and Defaults
Entity Identity	Specifies which attributes (if any) act as its Entity's primary key.	-	Choose multiple attributes on the pulldown list by opening the list then holding `Ctrl` while clicking the selections.
Datastore Persistent	Indicates whether this entity will be database bound.	Required.	`Yes, No`, defaults to `No`.
Table Name	Name of database table, chosen from a drop-down list of all database table names, fully-qualified with catalog and schema if applicable.	Optional, only active when the entity is **Datastore Persistent**	Value selected. When not specified, the system infers the best matching table from database metadata.

Property	Description	Applicability	Values and Defaults
Datastore Caching	Caching technique, chosen from a drop-down list. Indicates whether instances of this entity are subject to caching.	Optional, only active when the entity is **Datastore Persistent**.	Values are: • **No Cache** or blank (default) - Disable caching. • **Read Only** - Caches data that is never updated • **Read/Write** - Caches data that is sometimes updated while maintaining the semantics of "read committed" isolation level. If the database is set to "repeatable read," this concurrency strategy almost maintains the semantics. Repeatable read isolation is compromised in the case of concurrent writes. • **Nonstrict Read/Write** - Caches data that is sometimes updated without ever locking the cache. If concurrent access to an item is possible, this concurrency strategy makes no guarantee that the item returned from the cache is the latest version available in the database.
Identity Strategy	Strategy to generate unique identity for this entity.	Enabled when Entity Identity is not specified and DataStore Persistent is set to `Yes`	`Native, Table, Identity Sequence, UUID` (These are described in the following table.)
Identity Column Name	Name of the identity column, chosen from a drop-down list consisting of all column names associated with the table.	Enabled when Entity Identity is *unspecified*.	System attempts to create a match as *`entityName`*`_ID`.

Property	Description	Applicability	Values and Defaults
Identity Sequence	The fully-qualified name of the sequence to be used.	Applicable when Entity Identity is unspecified and Identity Strategy is **Sequence**.	System attempts to create a match as *entityName*_SEQUENCE.
Identity Table Name	The fully-qualified name of the identity table to be used, chosen from a drop-down list of all table names and sequence names.	Applicable when Entity Identity is unspecified and Identity Strategy is **Table**.	When not specified, the value defaults to SEQUENCE_TABLE.
Identity Table Name Column Name	The name of the column in the identity table that is used as the key (the name of the entity). Choose from a drop-down list of all columns in the table selected in the Table Name field.	Applicable when Entity Identity is unspecified and Identity Strategy is **Table**.	When not specified, the Name column name of the *Identity Table* defaults to SEQUENCE_NAME with data type (String).
Identity Table Value Column Name	The name of the column that holds the identity value. Chosen from a drop-down list of all columns in the table selected in the Table Name field.	Applicable when Entity Identity is unspecified and Identity Strategy is **Table**.	When not specified, the Value column of the *Identity Table* defaults to NEXT_VAL with data type (Big Integer).
Version Strategy	Strategy to control optimistic concurrency.	Optional. Tells Hibernate how to handle table locking. See the Hibernate Developer's Guide for more information.	Version Number, Timestamp
Version Column Name	Name of the column that contains the version number or the timestamp.	Applicable and required when Version Strategy is specified.	The specified column name is created when you update the database schema.

Note: Identity and strategy concepts are general relational database concepts. Refer to your RDBMS brand's documentation for more information, especially the identity strategies that are specific to certain brands. Also see *"Identity strategies"* and *"Advantages of using Identity Strategy rather than Sequence Strategy" in the Data Integration Guide* .

Table 3: Description of the Identity Strategy values

Strategy	Description
Native	Allows database to choose best possible identity strategy.

Strategy	Description
`Table`	Uses a database table, whose name is specified in **Identity Table Name**. This table will have two columns: a name column and a value column. (Previously referenced as "increment".)
`Identity`	Uses the native identity capability in DB2, SQL Server (the column is defined as an "identity" column in the database schema).
`Sequence`	Uses sequence capability in DB2, PostgreSQL and Oracle.
`UUID`	Generates 128-bit UUID string of 32 hex digits. (Previously referenced as "uuid-hex".)

Note: Databases mentioned in the Identity Strategy table do not imply that they are currently supported RDBMS brands. For the current list of supported RDBMS brands, access the web location Progress Corticon 6.0 - Supported Platforms Matrix.

When a database Datasource (ADC) has been added to the Vocabulary, its Entity properties for database interaction are displayed. These properties and their usage are discussed in the *Data Integration Guide*.

Table 4: Database Datasource Entity Properties

Property	Description	Values	Applicability
Table Name	Name of the database Table, chosen from a drop-down list consisting of all column names associated with the Entity Table Name.		Required.

Editing Attribute EDC properties

When an EDC Datasource has been added to the Vocabulary, its Attribute properties for database interaction are displayed.

Note: The basic and document mapping Attribute properties are discussed in *"Attribute nodes: Adding and editing attributes and their properties" in the Quick Reference Guide.*

Table 5: Enterprise Data Connector (EDC) Attribute Properties

Property	Description	Values	Applicability
Column Name	Name of the database column, chosen from a drop-down list consisting of all column names associated with the Entity Table Name (see Entity Properties).		Optional, if not specified, system will infer best match from database metadata.

Property	Description	Values	Applicability
Value Strategy	Strategy to use to generate unique value for this column.	`Native, Table, Identity, Sequence, UUID`	Optional. Only enabled if attribute is an element of the Entity Identity set. Only value strategies that make sense with respect to the attribute data type will be presented in the drop-down list.
Value Sequence	The fully-qualified name of the sequence to be used.		Only enabled if attribute is an element of the Entity Identity set and Identity Strategy is Sequence. Required if enabled.
Value Table Name	The fully-qualified name of the identity table to be used, chosen from a drop-down list of all table names and sequence names.		Only enabled if attribute is an element of the Entity Identity set and Identity Strategy is Table. Optional. If not specified, the value will default to `SEQUENCE_TABLE`.
Value Table Name Column Name	The name of the column in the identity table that is used as the key (this column will contain the name of the entity). Chosen from a drop-down list of all columns in the table selected in the Table Name field.		Only enabled if attribute is an element of the Entity Identity set and Identity Strategy is Table. If not specified the value will default to `SEQUENCE_NAME (String)`.
Value Table Value Column Name	The name of the column which holds the identity value. Chosen from a drop-down list of all columns in the table selected in the Table Name field.		Only enabled if attribute is an element of the Entity Identity set and Identity Strategy is Table. If not specified the value will default to `NEXT_VAL (Big Integer)`.

Table 6: Description of the Value Strategy values

Strategy	Description
`Native`	Allows database to choose best possible value strategy.
`Table`	Uses a database table, whose name is specified in **Identity Table Name**. This table will have two columns: a name column and a value column. (Previously referenced as "increment".)

Strategy	Description
`Identity`	Uses the native identity capability in DB2, SQL Server (the column is defined as an "identity" column in the database schema).
`Sequence`	Uses sequence capability in DB2, PostgreSQL and Oracle.
`UUID`	Generates 128-bit UUID string of 32 hex digits. (Previously referenced as "uuid-hex".)

Note: Databases mentioned in the Value Strategy table do not imply that they are currently supported RDBMS brands. For the current list of supported RDBMS brands, access the web location Progress Corticon6.0 - Supported Platforms Matrix.

When a database Datasource (ADC) has been added to the Vocabulary, its Attribute properties for database interaction are displayed. These properties and their usage are discussed in the *Data Integration Guide*.

Table 7: Database Datasource Attribute Properties

Property	Description	Values	Applicability
Column Name	Name of the database column, chosen from a drop-down list consisting of all column names associated with the Entity Table Name (see Entity Properties).		Required.

Importing possible values of an attribute from database tables

A database connection can also provide designers and testers of Rulesheets and Ruletests with lists of enumerations, also known as *possible values*. While these lists can be created and maintained by hand on the Custom Data Types tab of a Vocabulary, you can retrieve lists from the connected database.

Consider the general behavior of enumerations, especially when retrieving labels and values from a database:

- There can be only one instance of any label and any value in the list, whether created manually or imported. An exception will make the Vocabulary invalid. The database retrieval will work as expected but you will have to groom the results to make the lists valid. You can get optimal results when your database source prevents duplicates in the table columns you are using for your values or label-value pairs.
- If you chose a label in a Rulesheet and that label is no longer available after an update, an error will occur. Any Rulesheet expressions that refer to the defunct label will be flagged as invalid. You must update the Rulesheet expressions to correct the problem.
- If you chose a label in a Rulesheet and that label takes on a different value after an update, the current value is what is evaluated.
- The value assigned - whether directly or as the label's value - at the time of deployment does not change thereafter on the server.

It is good practice to ensure that the data types of the retrieved values in the database are consistent with the Custom Data Type, and then extend the corresponding base data value in the attribute.

Procedures

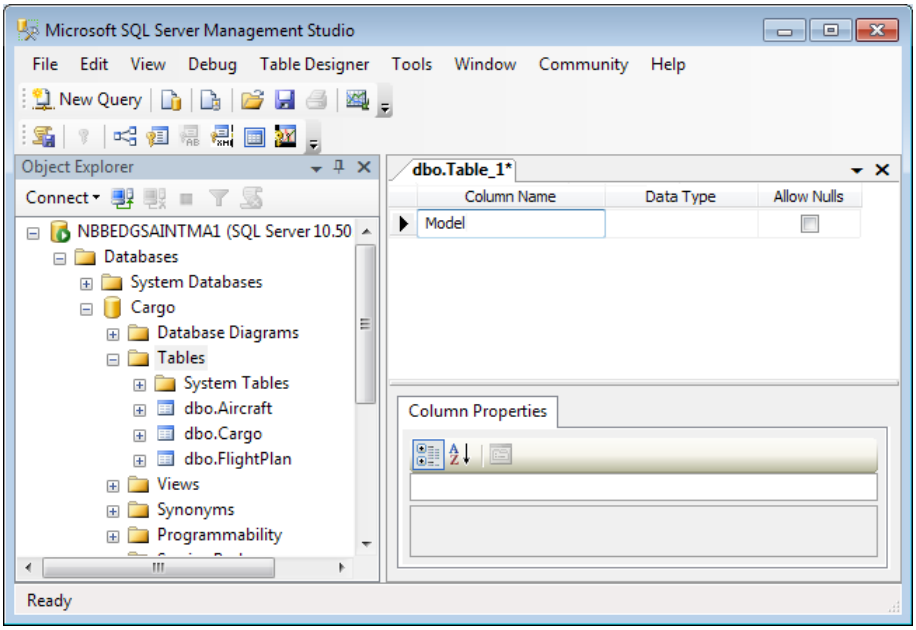
The steps to implement custom data types retrieved from a database are, in summary, as follows:

- A - Create or locate the database table and columns you want to retrieve.
- B - Verify the database connectivity, and then import its metadata.
- C - Define the Custom Data Type lookup information.
- D - Import the enumeration elements.
- E - Check the lists for duplicates.
- F - Set the Data Type of appropriate attributes to the Custom Data Type.
- G - Verify that the list functions correctly.

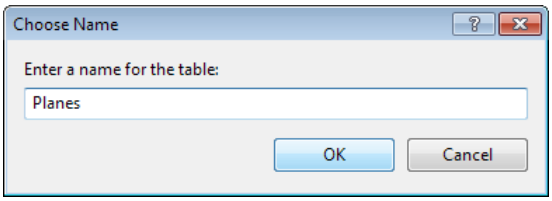
A - Create or locate the database table and columns you want to retrieve.

You need to add two tables to the SQL Server database to demonstrate both value-only and label+value enumerations:

1. Start the SQL Server Management Studio, and then expand the tree for **Databases : Cargo : Tables**. Right-click on **Tables** and choose **New Table**. Enter `Model` as the only column name, as shown:



2. Choose the menu command **File > Save Table_1**, enter the name `Planes`, and then click **OK**.

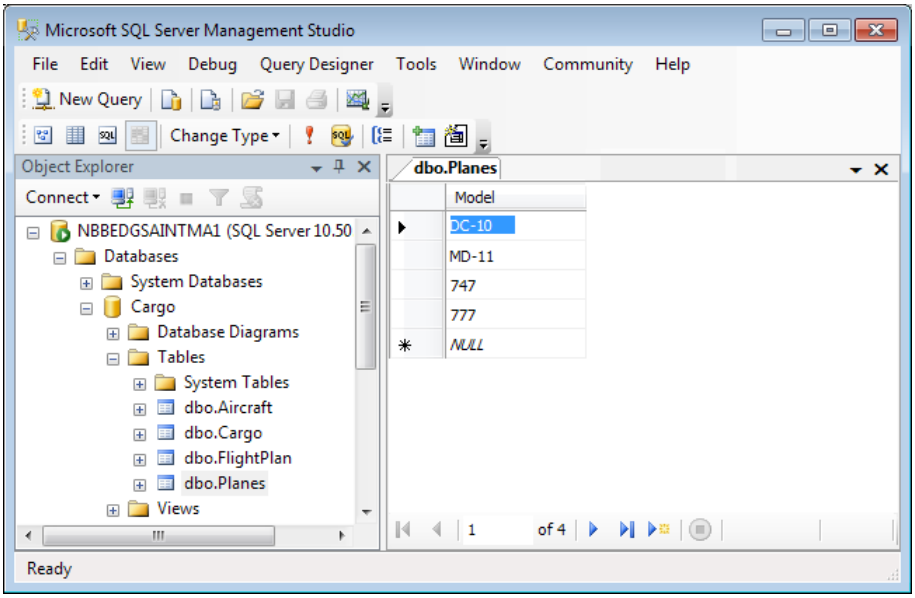


3. Create another table, now with two columns named `planeCarrier` and `planeID`, saving it as `Carrier`.

4. Click **New Query**, copy/paste the following text, and then click **Execute**.

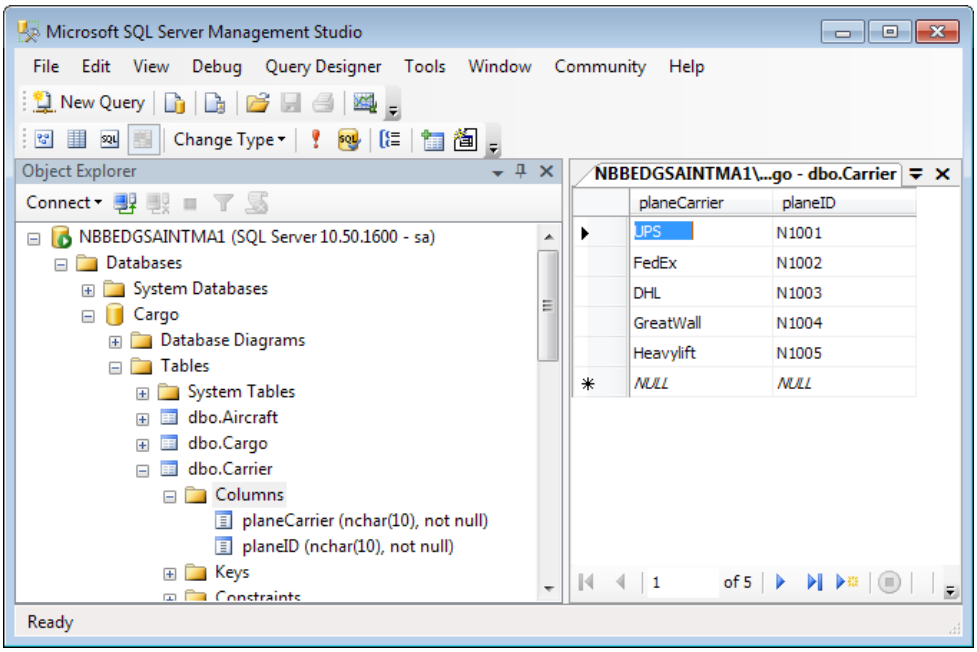
```
INSERT INTO Cargo.dbo.Planes (Model) VALUES ('DC-10');
INSERT INTO Cargo.dbo.Planes (Model) VALUES ('MD-11');
INSERT INTO Cargo.dbo.Planes (Model) VALUES ('747');
INSERT INTO Cargo.dbo.Planes (Model) VALUES ('777');
INSERT INTO Cargo.dbo.Carrier (planeCarrier,planeID) VALUES ('UPS','N1001');
INSERT INTO Cargo.dbo.Carrier (planeCarrier,planeID) VALUES ('FedEx','N1002');
INSERT INTO Cargo.dbo.Carrier (planeCarrier,planeID) VALUES ('DHL','N1003');
INSERT INTO Cargo.dbo.Carrier (planeCarrier,planeID) VALUES ('GreatWall','N1004');
INSERT INTO Cargo.dbo.Carrier (planeCarrier,planeID) VALUES ('Heavylift','N1005');
```

5. In the tree, right-click on **dbo.Planes**, and then choose **Edit Top 200 Rows**.



The **Planes** data is as we intended. It is ready for our use in the Corticon Studio.

6. Similarly, right-click on **dbo.Carrier**, and then choose **Edit Top 200 Rows**.

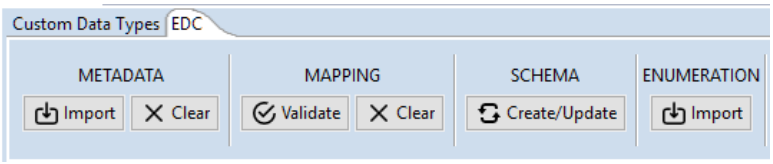


The **Carrier** data is as we intended. It is ready for our use in the Corticon Studio.

B - Verify the database connectivity, and then import its metadata.

We want to bring the information about the table definitions into the Studio:

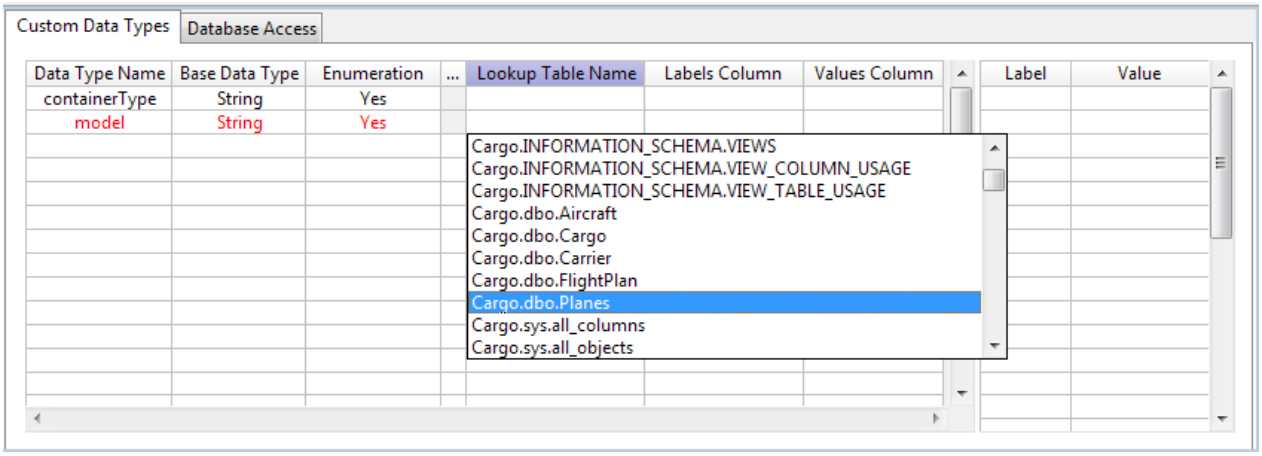
1. In Corticon Studio, confirm that you have the same good connection you achieved in Getting Started with EDC on page 27
2. With `Cargo.ecore` open its editor, select the Vocabulary root, and the click METADATA **Import**, as illustrated:



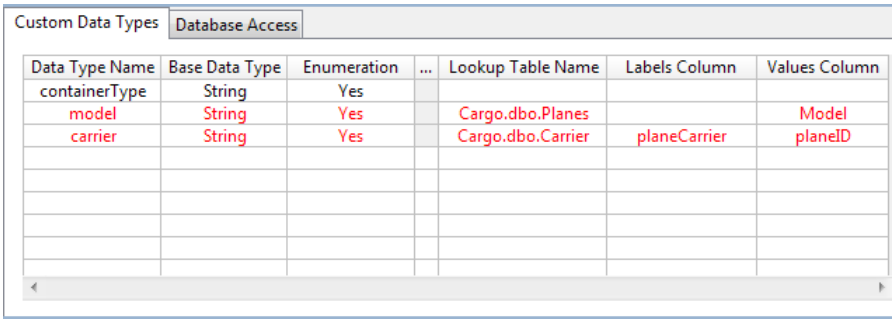
C - Define the Custom Data Type lookup information.

We now can specify how we want to use the data and then bind it to the appropriate database table and columns:

1. Click on `Cargo` to get to its top level, and then select the **Custom Data Types** tab.
2. Click on the next empty row, enter `model` as the Data Type Name, select `String` as the Data Type, and `Yes` as the Enumeration.
3. Click on the Lookup column in the row to expose its dropdown, and then choose `Cargo.dbo.Planes` that we imported in the database metadata.



4. We are using a values-only lookup, click on the row's Values Column to select its one database column, `Model`:
5. For the other table, click on the next empty row, enter `carrier` as the Data Type Name, select `String` as the Data Type, and `Yes` as the Enumeration.
6. Click on the Lookup Table Name in the row to expose its dropdown, and then choose `Cargo.dbo.Carrier` that we imported in the database metadata.
7. We are using a label-values lookup, so click on the row's Labels Column to select planeCarrier, and then in the Values Column to select `planeID`:

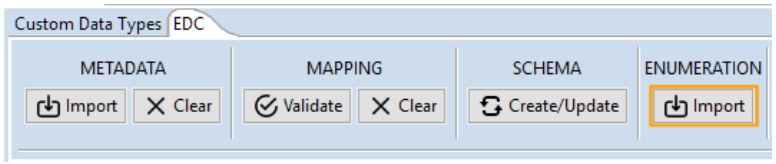


Everything we have entered is red! That's because Studio has no data for either of these enumeration sets.

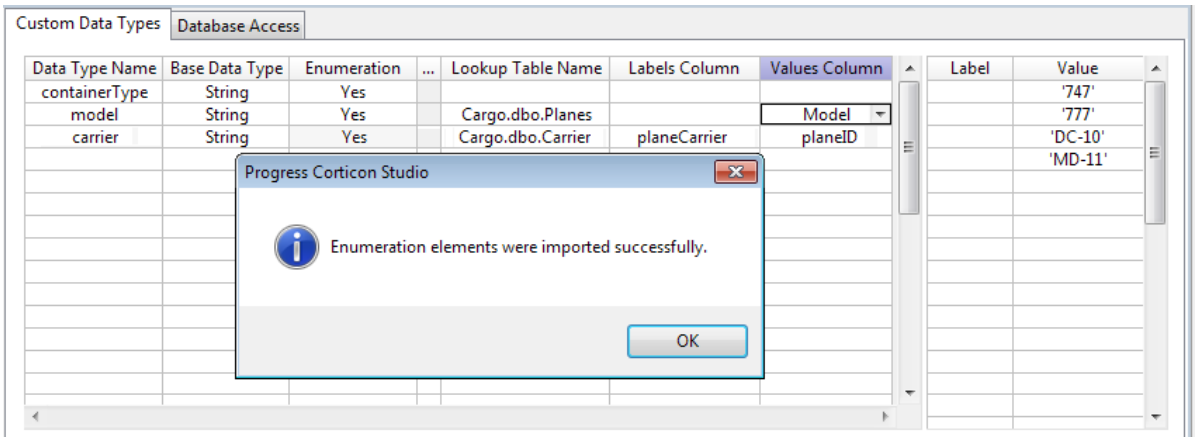
D - Import the enumeration elements.

Once you have defined the database table and columns you want, you can retrieve the data:

1. Choose the menu command **Database Access > Import Enumeration Elements**, as shown:



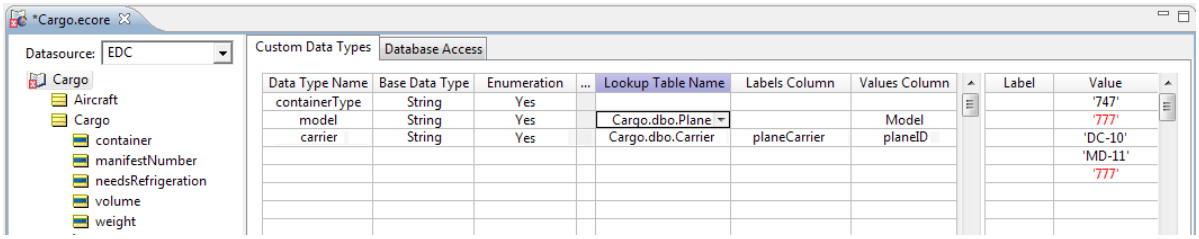
2. The retrieved values are displayed in the associated Labels and Values window to the right, as shown for the `model`:



E - Check the lists for duplicates.

Unless you enforced uniqueness in the source database. To demonstrate what happens, we'll add an existing value to the `model` enumerations.

1. In the Values retrieved column, enter a new value that is already there, such as `777`, as shown:



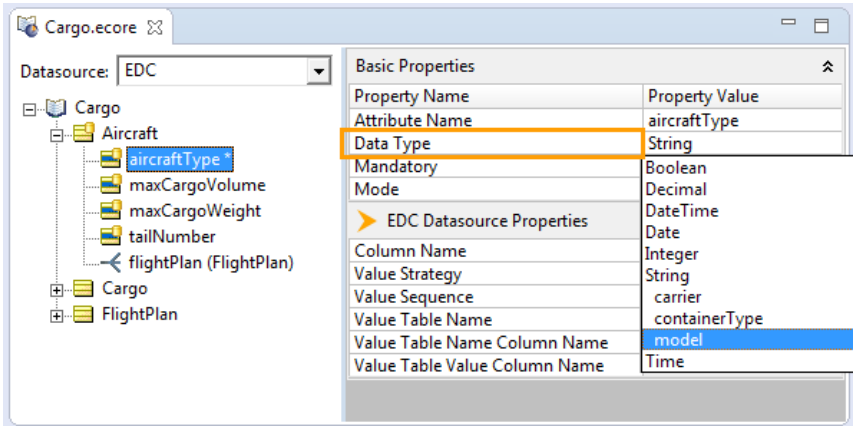
   The duplicates are both highlighted in red, and the `Cargo.ecore` file is marked as being in an error state.

2. Remove the line (or change it to something unique) and the Vocabulary is again valid.

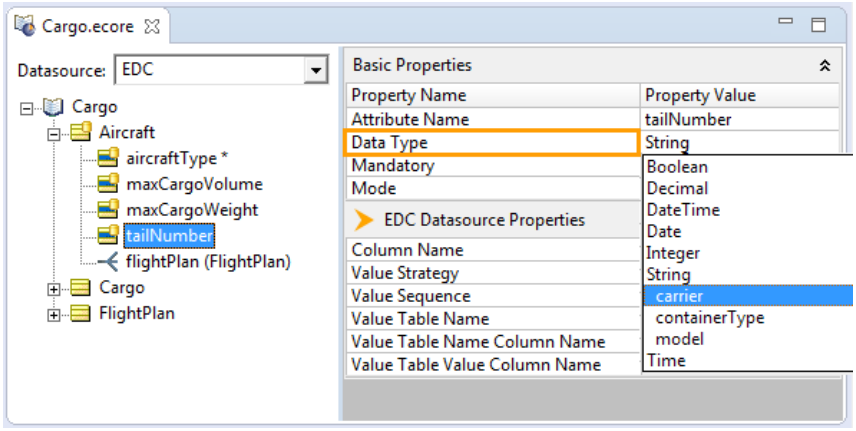
F - Set the Data Type of appropriate attributes to the Custom Data Type.

With our enumeration lists imported from the database and verified as free of duplicate labels or values, we can link them to the attributes that will use them:

1. Aircraft.aircraftType:



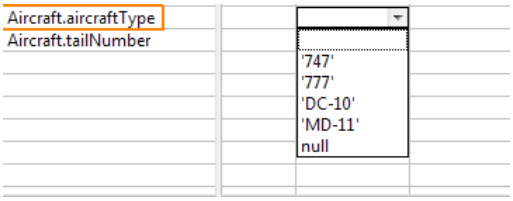
2. Aircraft.tailNumber:



G - Verify that the list functions correctly.

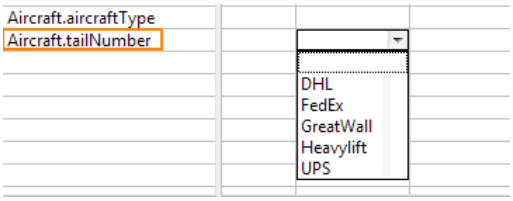
To verify that the lists perform as expected, use them in a Rulesheet or Ruletest :

1. In a Rulesheet Actions area, enter two new lines, one with the attribute syntax `Aircraft.aircraftType` and the other with `Aircraft.tailNumber`, as shown:
2. Click on the `aircraftType` where it intersects with column 1, as shown:



The pulldown displays our imported values, as well as blank and `null`.

3. Click on the `tailNumber` where it intersects with column 1, as shown:



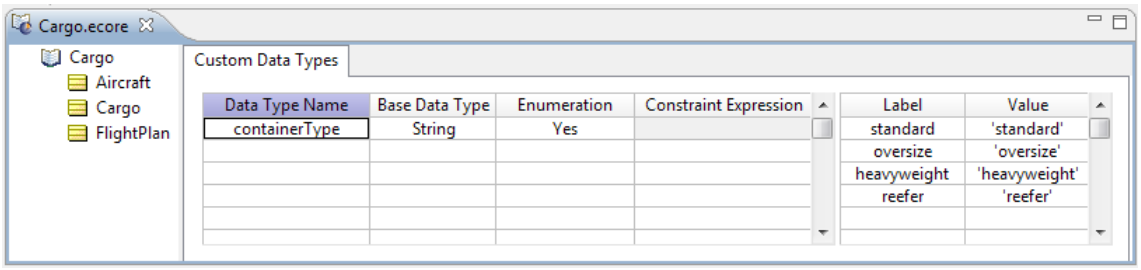
The pulldown displays our imported label, as well as blank. The label is a place holder for its value.

Note: For more information about enumerations and retrieving values from databases, see:

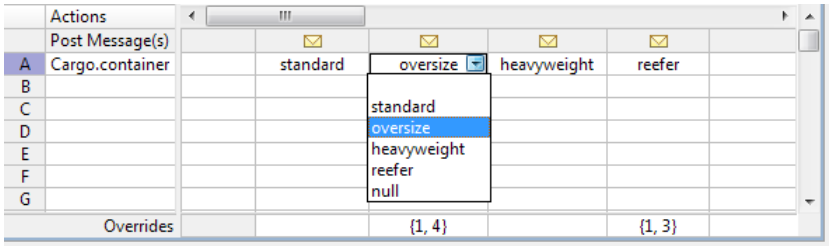
- *"Enumerated values" in the Quick Reference Guide.*
- *"Enumerations retrieved from a database" in the Rule Modeling Guide.*

Enumerated values

You can define lists of values that are the set of allowable values associated with a Vocabulary attribute. In the *Basic Tutorial*, you saw how we could delimit the options for a `containerType` by defining labels and their respective values:

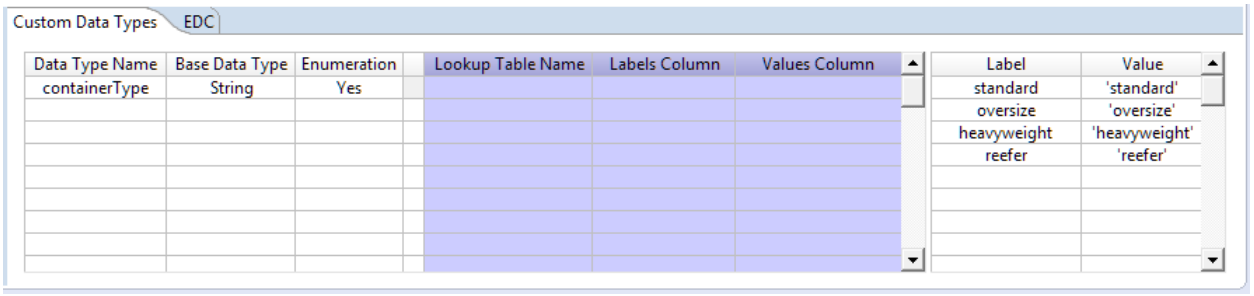


Then, when you are in the Rulesheet, the defined values -- as well as `null` and blank -- were offered.



Importing enumerated values from a database

When you use the Enterprise Data Connector and establish connection to a database, additional functionality is added to the **DataTypes** tab:



You can specify a column within a table of the connected database to retrieve and import the name and values (or just the values) to populate the selections to the specified attribute.

Note: For more information about enumerations and retrieving values from databases, see:

- *"Enumerations defined in the Vocabulary" in the Rule Modeling Guide*
- *"Enumerations retrieved from a database" in the Rule Modeling Guide*
- *"Importing an attribute's possible values from database tables" in the Data Integration Guide*
- *"Mapping database tables to Vocabulary Entities" in the Data Integration Guide*

Editing Association EDC properties

When an EDC Datasource has been added to the Vocabulary, its Association properties for database interaction are displayed.

Note: The basic and document mapping Association properties are discussed in *"Association nodes: Adding and editing associations and their properties" in the Quick Reference Guide.*

Table 8: Enterprise Data Connector (EDC) Attribute Properties

Property	Description	Applicability
Join Expression	Expression that defines the relationships between foreign key columns in the database	Required for all database-mapped associations. Inferred in most instances from database metadata (exceptions: unary associations and certain many-to-many associations).

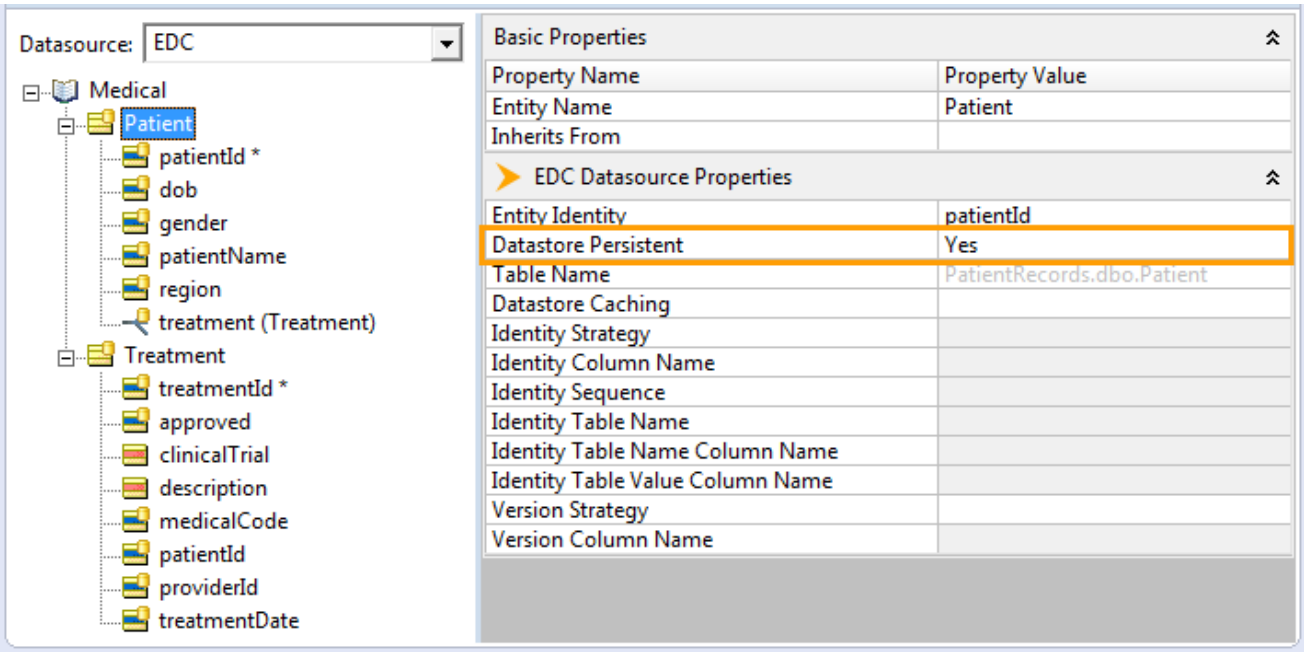
Mapping and validating EDC database metadata

Mapping data between a Corticon Vocabulary and relational database is not always perfect. When there are issues, you need to review the mappings to resolve incomplete or conflicting mapping data.

Mapping EDC database tables to Vocabulary Entities

Not all Vocabulary entities must be mapped to corresponding database tables - only those entities whose attribute values need to be persisted in the external database should be mapped. Those entities not mapped should have their **Datastore Persistent** property set to `No`. Mapped entities must have their **Datastore Persistent** property set to `Yes`, as shown circled in orange in the following figure:

Figure 10: Automatic Mapping of Vocabulary Entity

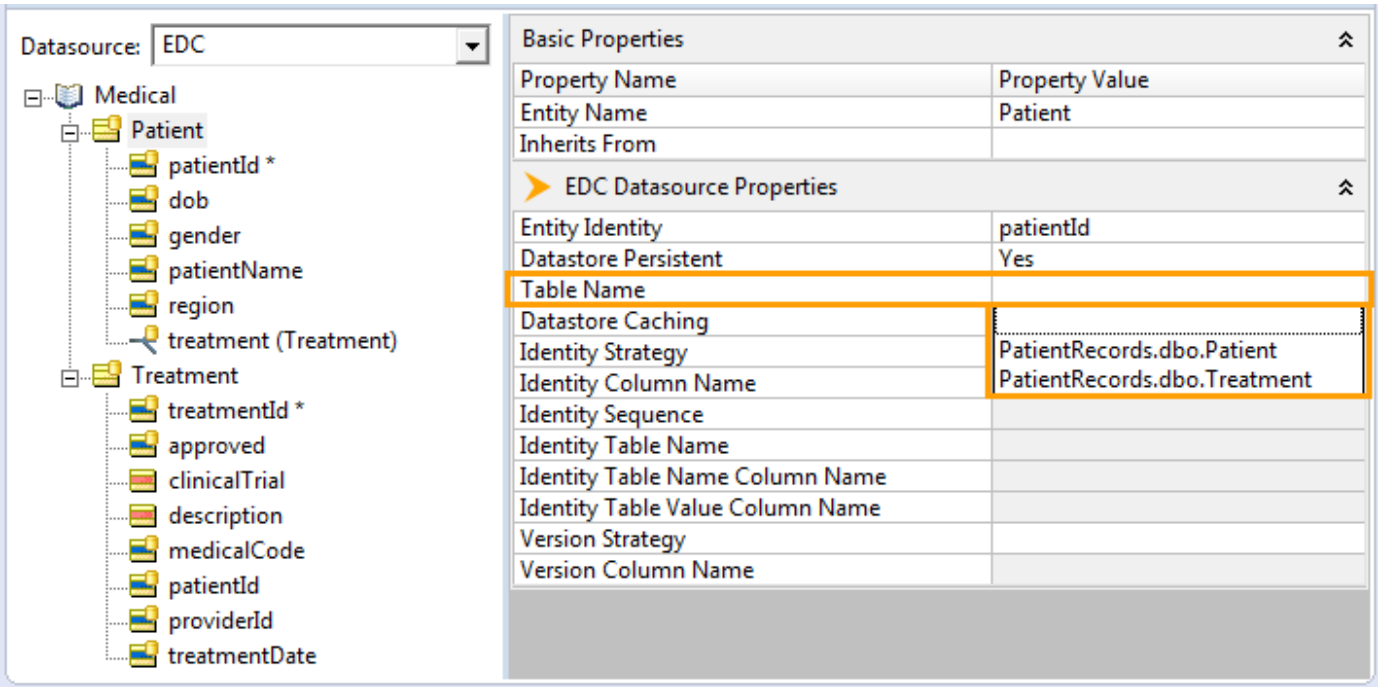


It is also possible for an external database to contain tables or fields not mapped to Vocabulary entities and attributes - these terms are simply excluded from the Vocabulary.

Assume that database metadata containing a table named `Patient` was imported. Because the table's name spelling matches the name of entity `Patient`, the **Table Name** field was mapped automatically. Automatic mappings are shown in light gray, as shown above. Also, note that the primary key of table `Patient` is a column named `patientId`. The Vocabulary Editor detects the primary key and determines that the property **Entity Identity** should be mapped to attribute `patientId`.

If the automatic mapping feature fails to detect a match for any reason (different spellings, for example), then you must make the mapping manually. In the **Table Name** field, use the drop-down list to select the appropriate database table to map, as shown:

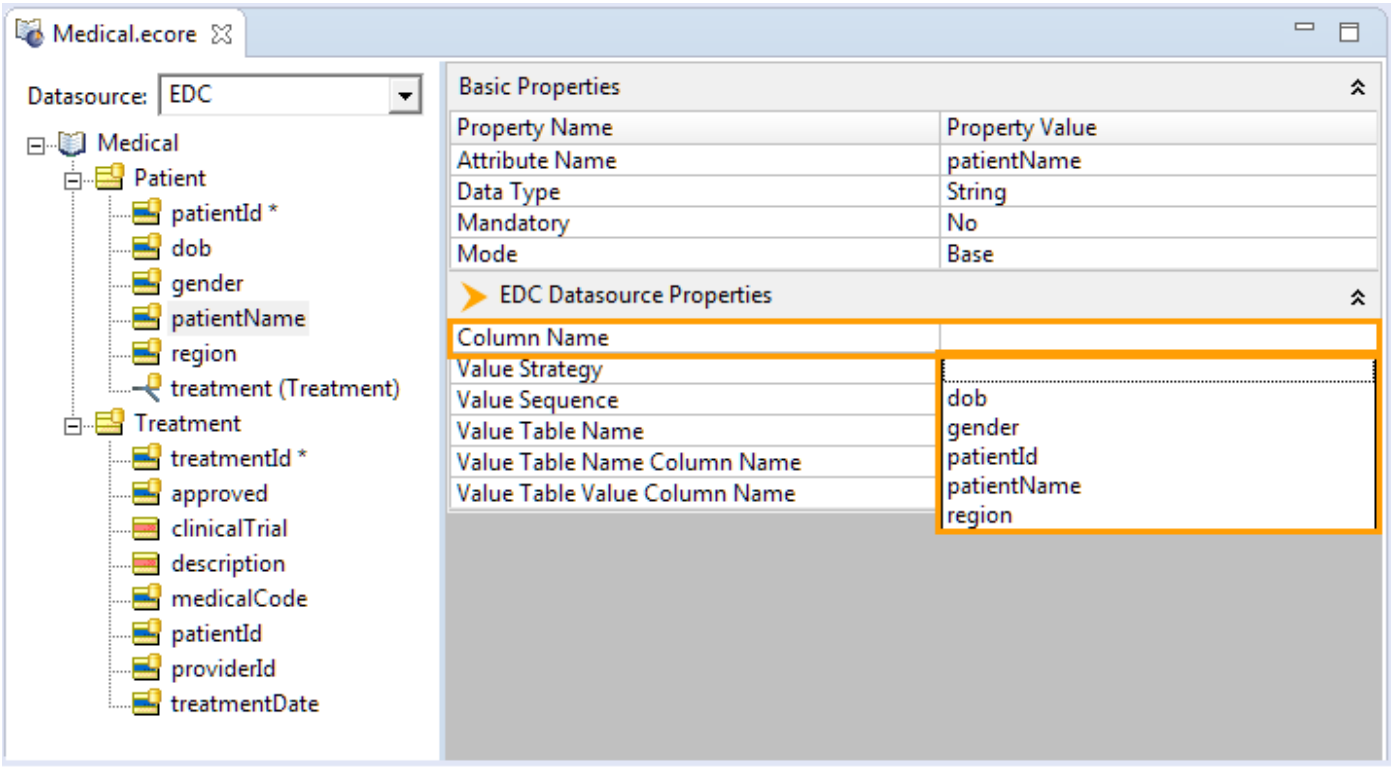
Figure 11: Manual Mapping of Vocabulary Entity



Mapping EDC database fields (columns) to Vocabulary Attributes

Automatic mapping of attributes works the same as entities. If an automatic match is not made by the system, then select the appropriate field name from the drop-down in **field:column** property, as shown:

Figure 12: Manual Mapping of Vocabulary Attribute



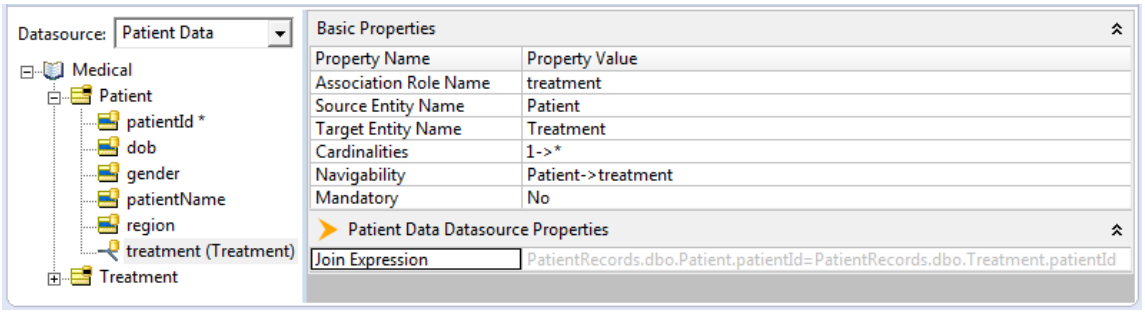
Note: Handling data in a `CHAR` database column

The database column type `CHAR` has a constant length. When a Corticon string attribute is mapped to such a column, the string retrieved from the database always has the length that is specified in the database definition. When a string shorter than the specified length is assigned to the attribute, the database adds spaces at the end of the string before storing it in the database. When the attribute is retrieved from the database, the value returns with the padded spaces at the end of the string.

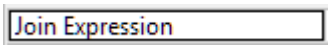
If this is not the intended behavior, change the database type for the column from `CHAR` to variable-length character data type. If the database schema cannot be changed, either use a `trimSpace` operator to strip the trailing spaces from the returned attribute value or redefine the query string to allow for its full length including added spaces.

Mapping EDC database relationships to Vocabulary Associations

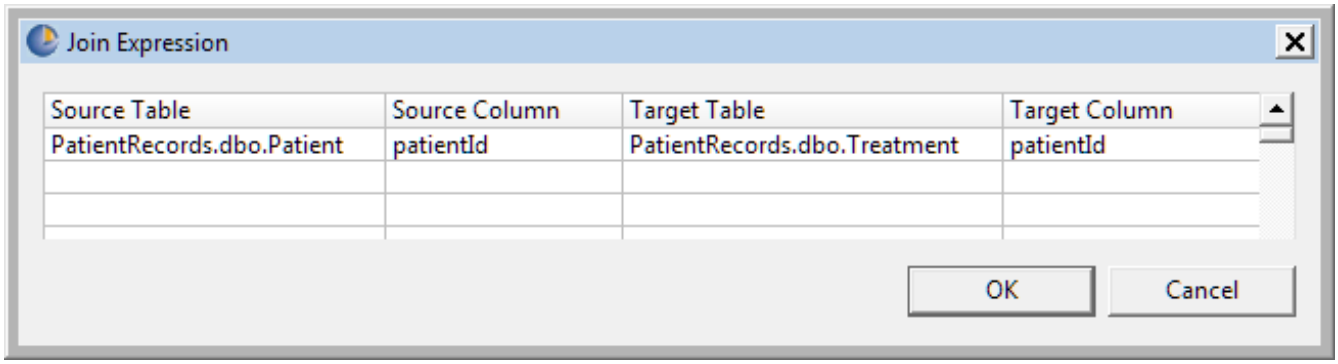
Automatic mapping of associations works substantially the same as entities. However, rather than entry text boxes and pulldowns for mappings, a more visual approach is provided. If an automatic match is made by the system, it is displayed in grey as shown:



If you want to revise the join expression, click on the **Join Expression** Property Name, as shown:



The **Join Expression** dialog box opens with a deconstruction of the join expression, as shown:

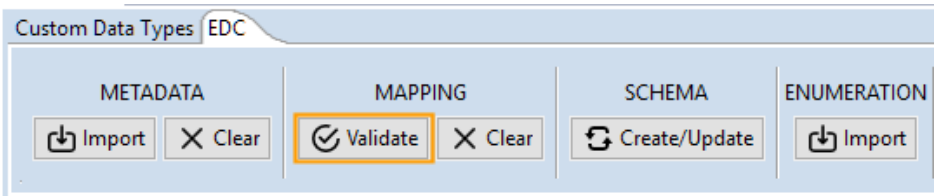


Use the pulldown lists in each column to refine the join expression. You can add lines to define complex join expressions where appropriate. As all revised join expressions are not validated, they are always displayed in black.

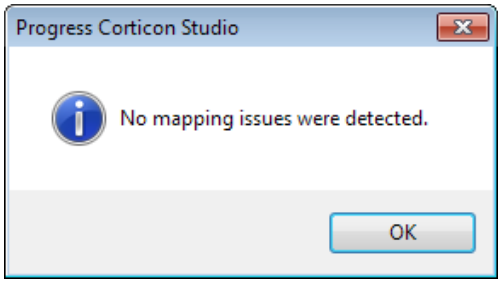
For more information and examples of complex joins, see Associations as join expressions on page 102

Validating EDC database mappings

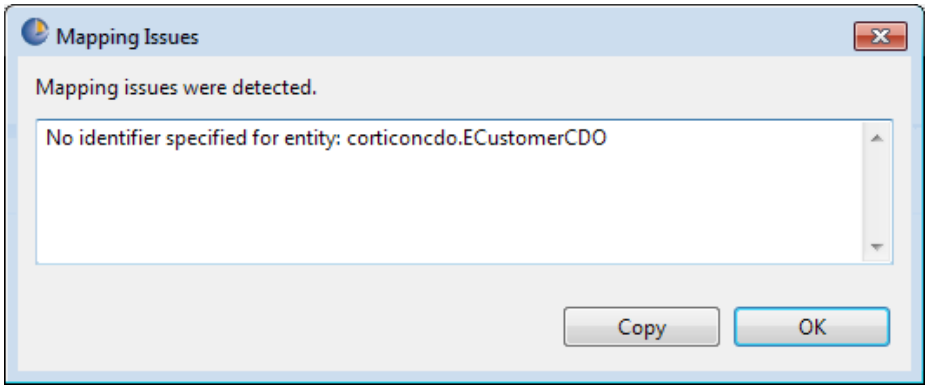
Once the Vocabulary has been mapped (either automatically or manually) to the imported database metadata, the mappings must be verified by clicking the MAPPING **Validate** button on the Datasource panel, as shown:



If all the mappings are valid, then a confirmation window opens:



If anything in the mappings does not validate, then a list of problems is generated:



These problems must be corrected before the Decision Service can be deployed.

Note: For a more detailed discussion of validation, see Types of mapping validation and validation errors on page 125

Writing rules that access data through the EDC connection

With the EDC connection established, mapped, and validated, you can proceed to the Rule Modeling Guide section *"Writing rules to access external data"* to create and test rules that use the EDC connection.

Types of mapping validation and validation errors

When EDC is enabled, the Vocabulary elements - Entity, Attribute, and Association - each have additional properties that can be entered by the user or inferred from database metadata. Corticon EDC validates these Vocabulary-to-Database mappings and displays error conditions in a window. There are three aspects to the database validation function:

Dynamic Validation

Corticon Studio validates against imported database metadata as property values change in a Vocabulary. For example, for a database-persistent entity, if you specify a table name that does not exist in the database metadata, the system posts a validation message in the **Problems View**. Dynamic validation uses internal Corticon algorithms to attempt validation without accessing Hibernate which would connect to the Datasource to do full validation. Studio creates other error and warning validation messages depending on the severity of the issue detected, such as:

- Data type mismatch between an attribute and a column that cannot be coerced.
- Property values are explicitly contradicted by database metadata.
- You select a property value, then re-import metadata, only to find that the selected value no longer exists in the database schema.

Warnings are also created for "soft" errors, such as:

- If you designate a Vocabulary entity as datastore persistent, the system is unable to infer which database table best matches the entity name, dynamic validation issues a warning message.
- If the system is unable to unambiguously determine the join expression for a given association, the association is flagged as a warning until you select one of the allowable values.

Note: Dynamic validation is always performed against the imported copy of database metadata. You must ensure that metadata is imported into the Vocabulary whenever the database schema is modified.

On-Demand Validation

In addition to dynamic validation, Corticon Studio provides the Datasource action **Validate Mappings**, as illustrated in Validating EDC database mappings on page 124, so that you can validate the Vocabulary, as a whole, against the database schema. Unlike dynamic validation, on-demand validation is performed against the actual schema so it is considered the definitive test of Vocabulary mappings.

Internally, the system performs on-demand validation by building annotated Corticon Data Objects (CDOs) from Vocabulary metadata, then asking Hibernate to evaluate the readiness of those CDOs with respect to the database schema. If Hibernate "blesses" the CDOs, the system displays a message box indicating that the Vocabulary mappings are valid, and that the Vocabulary is considered fit-for-use in a Decision Service. If Hibernate detects any errors, the system presents the errors to the user in a scrolling dialog window. The on-demand validation action simply presents raw information returned from Hibernate with no additional transformations or interpretation.

Validation at Deployment

Corticon Server leverages on-demand validation functionality whenever a decision service is deployed. If Corticon Server detects a problem, it throws an exception and prevents deployment.

Setting additional EDC Datasource connection properties

There are additional properties you might want to set for an EDC Datasource connection.

Note: It is a good practice to test your connection before and after changing additional properties.

Connection Pooling

Corticon uses C3P0, an open source JDBC connection pooling product, for connection pooling to Hibernate. The following properties might help tune connection pooling.

The following properties let you tune connection pooling:

Table 9: Settable C3P0 properties and their default value

Property Name	Default value	Comment
`hibernate.c3p0.min_size`	1	Minimum number of Connections a pool will maintain at any given time.
`hibernate.c3p0.max_size`	100	Maximum number of Connections a pool will maintain at any given time.

Property Name	Default value	Comment
hibernate.c3p0.timeout	1800	Number of seconds a Connection will remain pooled but unused before being discarded. Zero sets idle connections to never expire.
hibernate.c3p0.max_statements	50	Size of C3P0's PreparedStatement cache. Enter zero (0) to turn statement caching off. Then--depending on the alternative connection pooling mechanism requirements--you might need to declare required JAR and configuration files on the classpath.

You can bypass the use of C3P0 for connection pooling by setting the **Property** name hibernate.use.c3p0.connection_pool to the value false.

Note:

For more information about C3P0 and its use with Hibernate, see their *JDBC3 Connection and Statement Pooling* page at http://www.mchange.com/projects/c3p0/index.html#appendix_d.

Corticon has no recommendations for adjusting the properties in the Hibernate product. Refer to their web location for details. Then consult with Progress Corticon Support to note the behaviors you are attempting to adjust before making changes.

Database Time Zone

When your application stores date/time values in the database, you might need to set the following property:

com.corticon.edc.dateTimezone. This property pertains to only the DateTime data type, and lets you declare how DateTime values are expressed in the database:

Value	Purpose
JDK_DEFAULT_TIMEZONE	Declares that date/time values will be expressed in the Java Virtual Machine (JVM) time zone. Use this setting if your date/time values are expressed in “local” time.
UTC	Declares that date/time values will be expressed in GMT. This setting is typical for internet applications that are used across time zones.
America/Los_Angeles	Declares that date/time values will be expressed in America/Los Angeles time.
Europe/Paris	Declares that date/time values will be expressed in Europe/Paris time.
GMT+01:00	Declares that date/time values will be expressed in time zone GMT plus one hour.

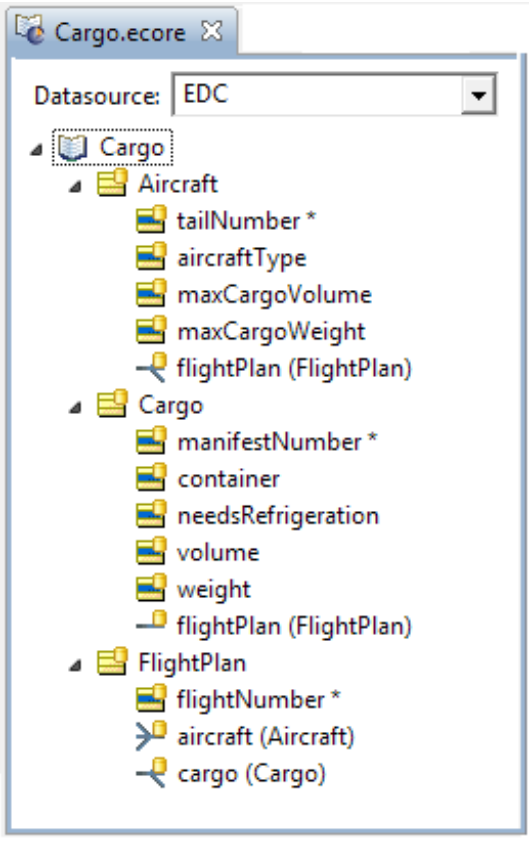
Set your override values in the **Property** table of the Vocabulary editor's **EDC** tab, as illustrated:

Property Name	Property Value
hibernate.c3p0.min.size	5
com.corticon.edc.dataTimezone	UTC

How data from an EDC Datasource integrates into rule output

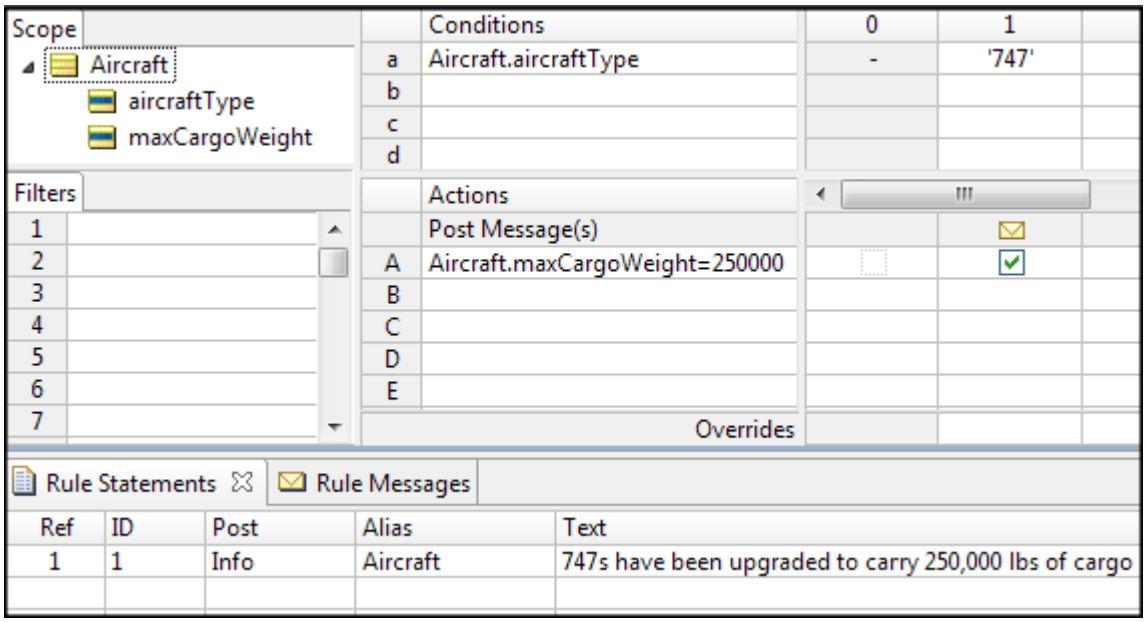
An EDC connector enables interaction with its connected database, its Datasource, to read and write data from rule executions. Without database connectivity, Decision Service execution takes data in the request payload, modifies it through rules, and then returns the data in the response. When EDC is used, the Datasource can enrich the data in the request, and can store result in the database. The following sections show separately the effects in read-only and read-update scenarios. Included in these examples are variations that use the **Extend to Database** feature to further enrich results.

To enable adequate complexity, the scenarios use data provided and the familiar `Cargo.ecore` Vocabulary:.



The sample Rulesheet is defined as shown:

Figure 13: Sample Rulesheet for EDC database examples



The Datasource data in this section was established in a Microsoft SQL Server 2014 installation as described in the topic *"Quick Steps for setting up the Cargo sample" in the Rule Modeling Guide.*

When Datasource access is Read Only

In **Read Only** mode, data may be retrieved from the database in order to provide the inputs necessary to execute the rules. But the results of the rules won't be written back to the database – hence, read-only.

Open the project's Ruletest, and then set the menu option **Ruletest > Testsheet > Database Access > Read Only**.

The variations that will be explored in Read Only mode are:

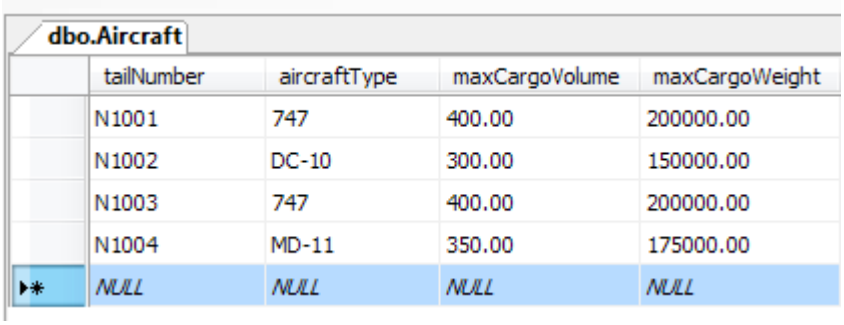
- Payload contains a record new to the database, and the entity is not extended to database on page 130
- Payload contains a record new to the database, and the entity is extended to database on page 131
- Payload contains existing database record on page 132
- Payload contains existing database record, but with changes on page 133

Finally, the section Effect of rule execution on the database on page 134 shows that the read-only functions did not change the database but perhaps they should have.

Payload contains a record new to the database, and the entity is not extended to database

Let's look at a Studio Test with an Input Ruletest (simulating a request payload) containing a record not present in the database. The initial database table `dbo.Aircraft` is as shown:

Figure 14: Initial state of database table `Aircraft`

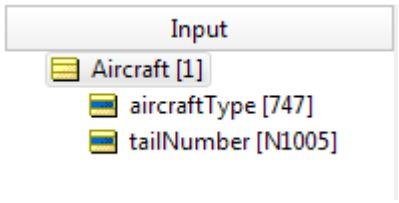


dbo.Aircraft

	tailNumber	aircraftType	maxCargoVolume	maxCargoWeight
	N1001	747	400.00	200000.00
	N1002	DC-10	300.00	150000.00
	N1003	747	400.00	200000.00
	N1004	MD-11	350.00	175000.00
▸*	NULL	NULL	NULL	NULL

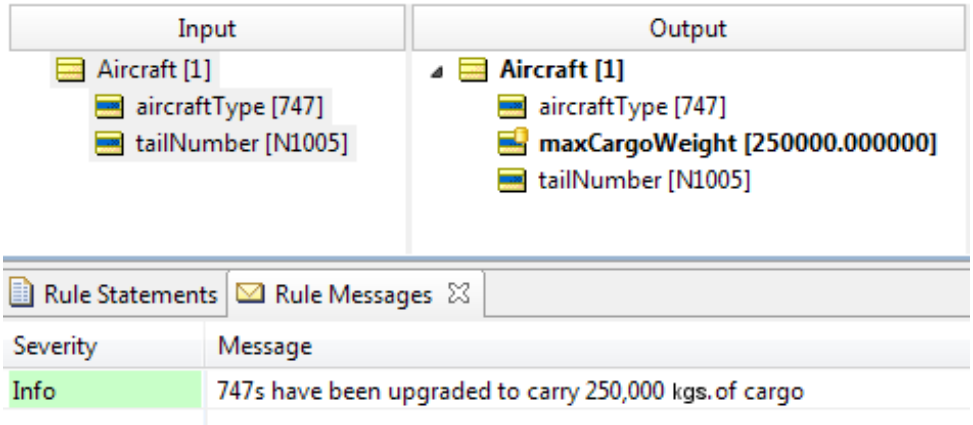
And the Studio Input Ruletest is as shown in the following figure.

Figure 15: Input Ruletest Testsheet with new record, in Read Only mode



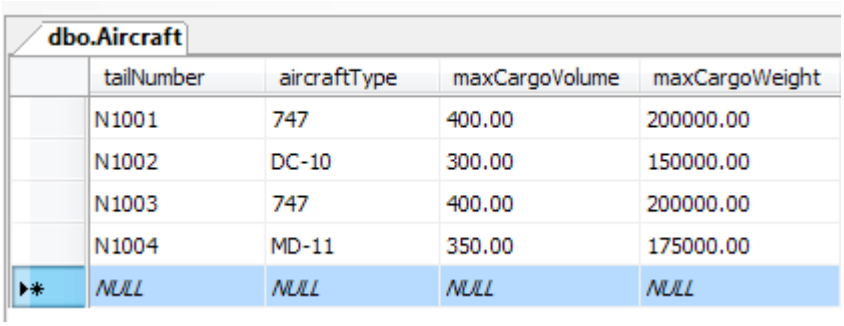
We know from our Vocabulary that `tailNumber` is the primary key for the `Aircraft` entity. We also know by examining the `Aircraft` table that this particular set of input data is not present in our database, which only contains aircraft records with `tailNumber` values `N1001` through `N1004`. So when we execute this Test, the Studio performs a query using the `tailNumber` as unique identifier. No such record is present in the table so all the data required by the rule must be present in the Input Ruletest. Fortunately, in this case, the required `aircraftType` data is present, and the rule fires, as shown:

Figure 16: Results Ruletest with new record



Again, since EDC is **Read Only** for this test, no database updates are made and the end start of the `AIRCRAFT` table, as shown, is the same as the original state:

Figure 17: Final state of database table `Aircraft`

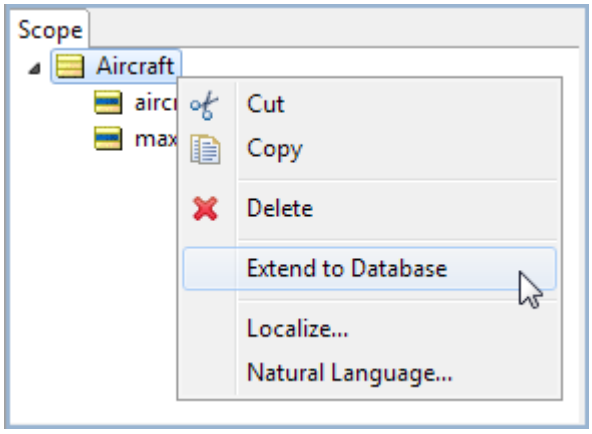


dbo.Aircraft

	tailNumber	aircraftType	maxCargoVolume	maxCargoWeight
	N1001	747	400.00	200000.00
	N1002	DC-10	300.00	150000.00
	N1003	747	400.00	200000.00
	N1004	MD-11	350.00	175000.00
*	NULL	NULL	NULL	NULL

Payload contains a record new to the database, and the entity is extended to database

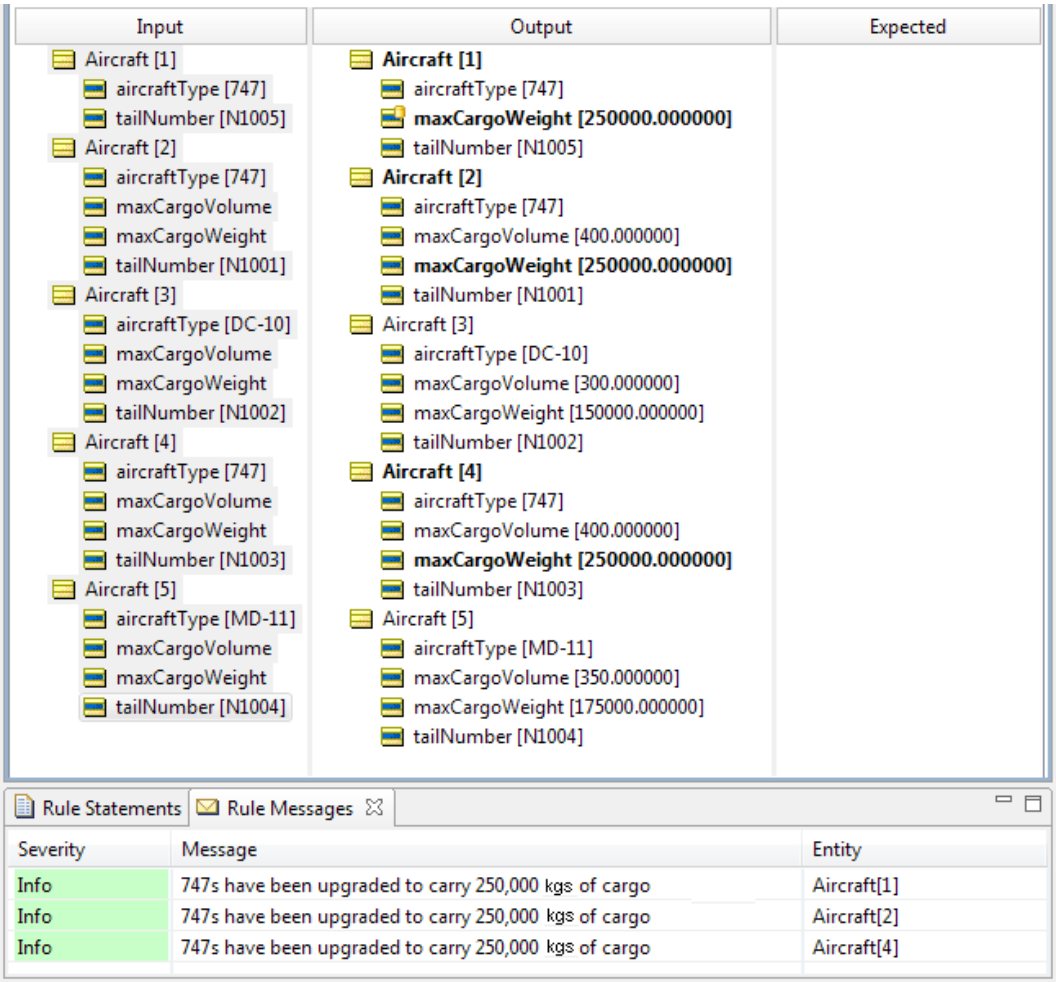
This scenario assumes the rule shown in Sample Rulesheet for Synchronization Examples makes use of an alias extended to the database. By placing the `Aircraft` Entity in the Scope of Rulesheet, we can right-click on `Aircraft` and then choose **Extend to Database** as shown:



See the *Rule Modeling Guide* chapter "Writing Rules to Access External Data" for more information about this setting. In that guide, you might want to learn about "Optimizing Aggregations that Extend to Database" which pushes these collection operations onto the database.

When our sample rule uses an alias extended to the database instead of the root-level entity shown in Sample Rulesheet for Synchronization Examples, different behavior is observed. When an Input Ruletest or request payload contains data not present in the database, as in test case `N1005` above, and the database access mode is **Read-Only**, then the rule engine dynamically re-synchronizes or "re-binds" with *only those records in the database table*. When this re-synchronization occurs, any data not present in the database table (like `N1005`) is excluded from working memory and not processed by the rules using that alias. The Results Ruletest is shown in the following figure. Notice that the Aircraft `N1005` was not processed by the rule even though, as a `747`, it satisfies the condition.

Figure 18: Results Ruletest showing rebinding

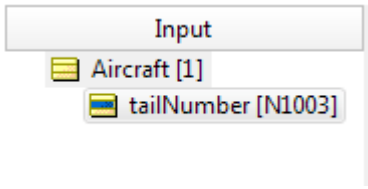


Payload contains existing database record

Now, let's change our input data so that it contains a record in the database. As we can see in the following figure, the value of `tailNumber` in the Input Ruletest has been changed to `N1003`. Also, the value of `aircraftType` has been deleted. By deleting the value of `aircraftType` from the Input Ruletest, rule execution is depending on successful data retrieval because the Input Ruletest no longer contains enough data for the rule to execute. Data retrieval is this rule's "last chance" – if no data is retrieved, then the rule simply won't fire.

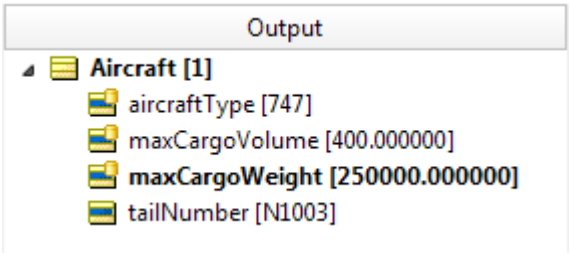
Fortunately, a record with this value exists in the database table, so when the Test is executed, a query to the database successfully retrieves the necessary data.

Figure 19: Ruletest input with existing record



The Results Ruletest, as shown below, confirms that data retrieval was performed.

Figure 20: Ruletest output with existing record



And, finding that the aircraft with `tailNumber=N1003` was in fact a `747`, the rule fired. But as before, no updates have been made to the database because this Test still uses Read-Only mode. The final database state is as shown:

Figure 21: Final state of database table `Aircraft`

dbo.Aircraft

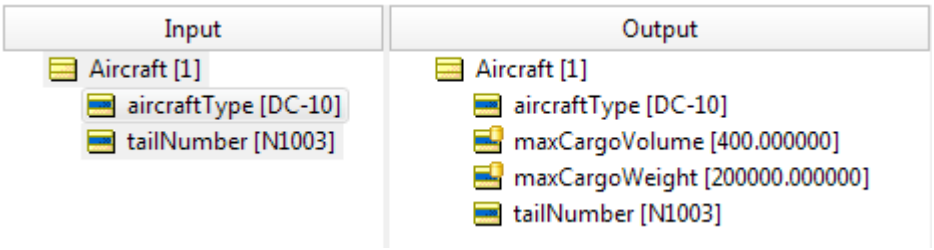
tailNumber	aircraftType	maxCargoVolume	maxCargoWeight
N1001	747	400.00	200000.00
N1002	DC-10	300.00	150000.00
N1003	747	400.00	200000.00
N1004	MD-11	350.00	175000.00
NULL	*NULL*	*NULL*	*NULL*

Payload contains existing database record, but with changes

What happens when, for a given record, the request payload and database record don't match? For example, look carefully at the Input Ruletest below. In the database, the record corresponding to tailNumber `N1003` has an `aircraftType` value of `747`. But the `aircraftType` attribute in the Input Ruletest has a value of `DC-10`. How is this mismatch handled?

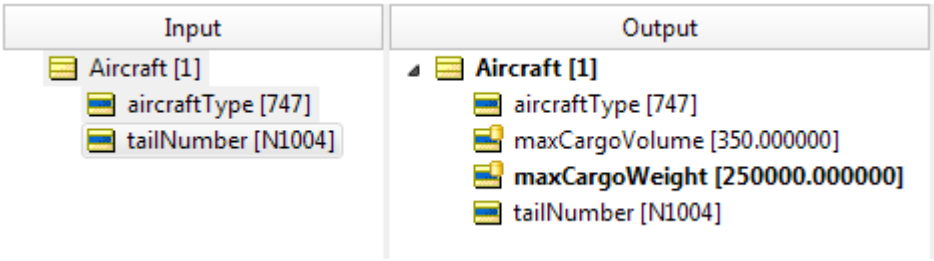
Studio still performs a query to the database because it has the necessary key information in the provided `tailNumber`. When the query returns with an `aircraftType` of `747`, the Synchronization algorithm decides that the data in the Input Ruletest has priority over the retrieved data – for the purposes of working memory (which is what the rules use during processing), the data in the Input Ruletest is treated as “more recent” than the data from the table. The state of `aircraftType` in working memory remains `DC-10`, and therefore the condition of the rule is not satisfied and the rule does not fire. Even though the database record defines the aircraft with `tailNumber` of `N1003` as a `747`, this is not good enough to fire the rule. The other piece of retrieved data, `maxCargoWeight`, is accepted into working memory and is inserted into attribute `maxCargoWeight` in the results Ruletest upon completion of rule execution, as shown on the right side of the following figure:

Figure 22: Ruletest with existing record but different aircraft



Let’s modify the scenario slightly. Look at the next Input Ruletest, as shown on the left side off the following image. It contains an `aircraftType` attribute value of `747`, but the `AIRCRAFTTYPE` value in the `AIRCRAFT` table of the database (for this value of `TAILNUMBER`) is `MD-11`. How is data synchronized in this case?

Figure 23: Ruletest with existing record and same aircraft



Once again, when a data mismatch is encountered, the data in the Input Ruletest (simulating the request payload) is given higher priority than the data retrieved from the database. Furthermore, the data in the Input Ruletest satisfies the rule, so it fires and causes `maxCargoWeight` to receive a value of `250000`, as shown on the right side of the figure above.

Effect of rule execution on the database

In several of the examples above, the state of data post-rule execution differs from that in the database. In Results Ruletest with Existing Record and Results Ruletest with Existing Record, rule execution produced a `maxCargoWeight` of `250000`, yet the database values remained `200000`. The application architect and integrator must be aware of this and ensure that additional data synchronization is performed by another application layer, if necessary. When Corticon Studio and Server are configured for **Read Only** data access, data contained in the response payload may not match the data in the mapped database.

When Datasource access is Read/Update

In Read/Update mode, Decision Services can update the database so that data changes made by rules are persisted. That avoids the problem of post-rule execution data mismatch experienced in **Read Only**, but must be used carefully (especially when testing from Studio!) since *rules will be writing to the database*.

Open the project's Ruletest, and then set the menu option **Ruletest > Testsheet > Database Access > Read/Update**.

The variations that will be explored in Read/Update mode are:

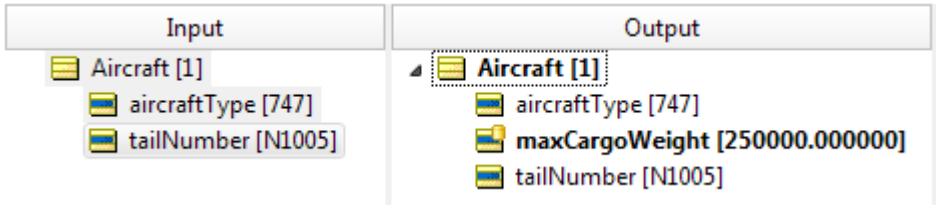
- Payload contains a new record not in the database on page 135
- Payload contains existing database record on page 136
- Payload contains existing database record, but with changes on page 136

Payload contains a new record not in the database

Once again, the Studio Ruletest Input is shown in the following figure.

As before, no such record is present in the table so all the data required by the rule must be present in the Input section. Fortunately, in this case, the required `aircraftType` data is present, and the rule fires, as shown:

Figure 24: Ruletest with new record



Since the EDC mode is **Read/Update**, a database update is made and the end state of the `Aircraft` table, shown below, is different from its original state.

Figure 25: Final state of database table Aircraft

dbo.Aircraft

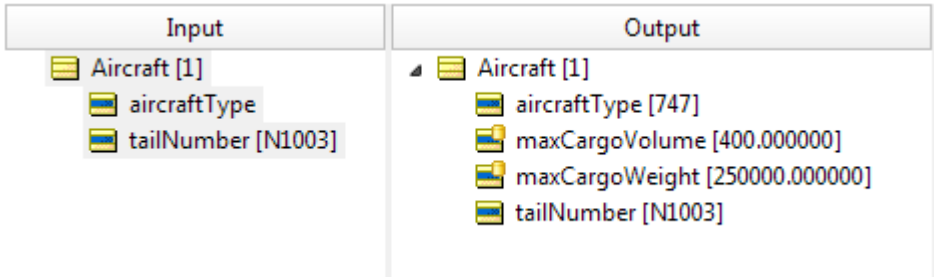
tailNumber	aircraftType	maxCargoVolume	maxCargoWeight
N1001	747	400.00	250000.00
N1002	DC-10	300.00	150000.00
N1003	747	400.00	250000.00
N1004	MD-11	350.00	175000.00
N1005	747	*NULL*	250000.00
NULL	*NULL*	*NULL*	*NULL*

We can see that the database and the Ruletest Results (simulating the response payload) contain identical data for the record processed by the rule – no post-execution synchronization problems exist.

Payload contains existing database record

Now, let's revisit the Input Ruletest shown in Input Ruletest with Existing Record. Setting this Test to **Read/Update** mode, it appears as shown:

Figure 26: Ruletest with existing record



The Output section of the Ruletest confirms that data retrieval was performed. And, finding the retrieved aircraft was (and still is) a `747`, the rule fired.

Unlike the **Read-Only** example, the database has been updated with the new `maxCargoWeight` data. The final database state is as shown:

Figure 27: Final state of database table `Aircraft`

dbo.Aircraft

tailNumber	aircraftType	maxCargoVolume	maxCargoWeight
N1001	747	400.00	250000.00
N1002	DC-10	300.00	150000.00
N1003	747	400.00	250000.00
N1004	MD-11	350.00	175000.00
N1005	747	*NULL*	250000.00
NULL	*NULL*	*NULL*	*NULL*

Payload contains existing database record, but with changes

To better illustrate how the following examples affect the database when run in **Read/Update** mode, we will return the database's `Aircraft` table to its original state, as shown:

Figure 28: Original state of database table `Aircraft`

dbo.Aircraft

tailNumber	aircraftType	maxCargoVolume	maxCargoWeight
N1001	747	400.00	200000.00
N1002	DC-10	300.00	150000.00
N1003	747	400.00	200000.00
N1004	MD-11	350.00	175000.00
NULL	*NULL*	*NULL*	*NULL*

When the following Ruletest is executed, we know from our experience with **Read Only** mode that the rule will not fire. However, notice in Final State of Database Table `Aircraft` that the database record has been updated with the `aircraftType` value (`DC-10`) present in working memory when rule execution ended. And since the value of `aircraftType` in working memory came from the Input Ruletest (having priority over the original database field), that's what's written back to the database when execution is complete. The final state of the data in the database matches that in the Results Ruletest upon completion of rule execution, as shown in the Results Ruletest:

Figure 29: Ruletest with existing record

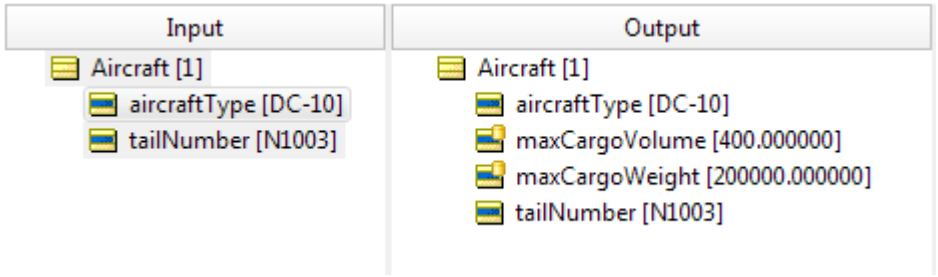


Figure 30: Final state of database table `Aircraft`

dbo.Aircraft

	tailNumber	aircraftType	maxCargoVolume	maxCargoWeight
	N1001	747	400.00	250000.00
	N1002	DC-10	300.00	150000.00
▶	N1003	DC-10	400.00	200000.00
	N1004	MD-11	350.00	175000.00
*	*NULL*	*NULL*	*NULL*	*NULL*

As before, let's modify the scenario slightly. The Ruletest Input shown in the next figure now contains an aircraft record that has an `aircraftType` value of `747`, but the `aircraftType` value in the database's `Aircraft` table (for this `tailNumber`) is `MD-11`. Let's see what happens to the database upon Test execution:

Figure 31: Ruletest with existing record

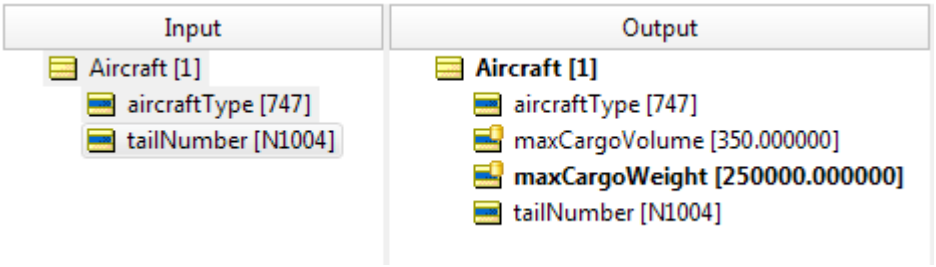
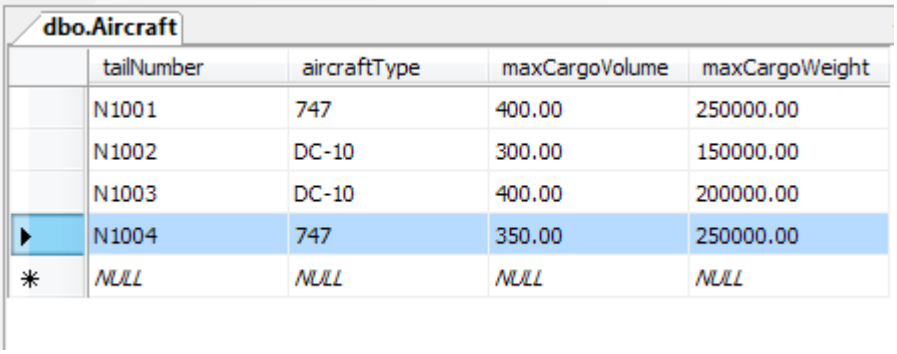


Figure 32: Final state of database table `Aircraft`



tailNumber	aircraftType	maxCargoVolume	maxCargoWeight
N1001	747	400.00	250000.00
N1002	DC-10	300.00	150000.00
N1003	DC-10	400.00	200000.00
N1004	747	350.00	250000.00
NULL	*NULL*	*NULL*	*NULL*

Once again, when a data mismatch is encountered, the data in the Input Ruletest (simulating the request payload) is given higher priority than the data retrieved from the database. Furthermore, the data in the Input Ruletest satisfies the rule, so it fires and causes `maxCargoWeight` to receive a value of `250000`, as shown above. Unlike before, however, the new `maxCargoWeight` value is updated in the database.

Using caches in EDC for entities and queries

As you move Decision Services that use an EDC database connection toward production, you might want to get the performance benefits of caching entities and queries used in Decision Services. Even though the first cache use has some performance overhead, subsequent references that use that cached content will see excellent performance.

There are two ways caching can be used in Corticon rules:

1. **Entity cache** is appropriate when common database-enabled entities are used in the input messages sent to a Decision Service.

   - *Case for using entity caching:* Consider a Decision Service that expedites Shipping Requests. The Decision Service might receive a Shipment Request for an Order entity that has a one-to-many association with Customer entities. When the Decision Service receives an Order entity, it would query the database to get the associated Customer entities (for example, the Decision Service needs the customer's address to estimate delivery lead time). When using an entity cache, the Customers could be queried once, and that data used in expediting other Order entities.

2. **Query cache** optimizes lookup (query) of database data, such as when a cached entity is extended to the database in a Rulesheet. A Query Cache is read-only -- it should not be expected to receive updates from the rules. If an update is attempted on an entity contained in a query cache, an exception occurs. A Query

Cache is **optimistic** -- that means that updates from outside of the Decision Service will not modify or invalidate the cache contents.

- *Case for using query caching:* A Decision Service that prices online orders using a query cache could query state sales tax rates (or VAT rates) once, and then use that data when calculating the price of all orders it receives.
- *Case for using query caching:* Consider a Decision Service that prices insurance policies. With a query cache, it could query an actuarial table once, and then use the results in pricing multiple insurance policies.

First, you specify the caching settings in the Vocabulary and Rulesheets, and then enable caching in tests and deployed Decision Services.

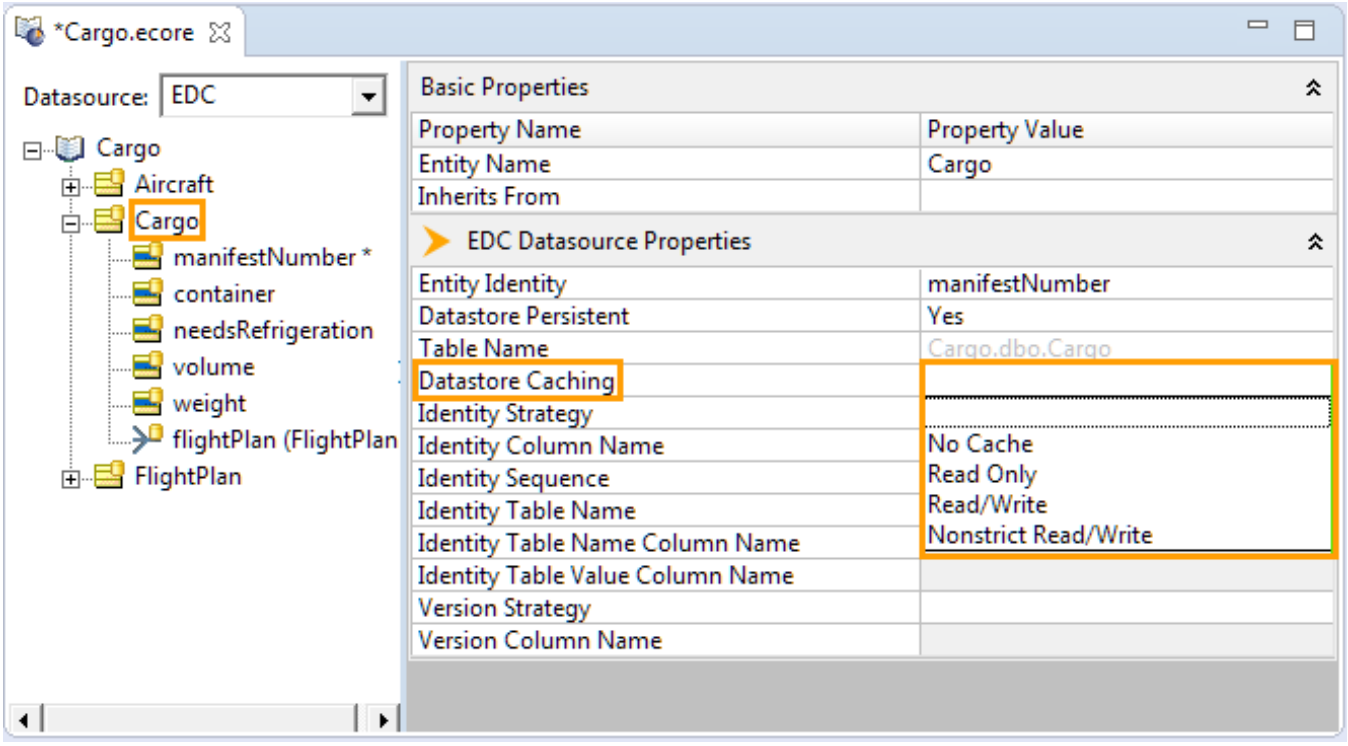
Specifying caching on Vocabularies and Rulesheets

Setting caching on datastore persistent Entities in the Vocabulary

Database caching is a feature of an EDC Datasource connection, enabling both Entity caching and Query caching.

To set Entity caching in a Vocabulary:

1. Identify the Vocabulary entities that you want cached.
2. Edit the Vocabulary.
3. Confirm (or set) each entity's **Datastore Persistent** property to `Yes`
4. Choose the preferred **Datastore Caching** value:
   - `No Cache` or blank (default) - Disable caching.
   - `Read Only` - Caches data that is never updated. This strategy works well for unchanging reference data that might need to occasionally be flushed and repopulated. For example, countries of the world.
   - `Read/Write` - Caches data that is sometimes updated while maintaining the semantics of "read committed" isolation level. If the database is set to "repeatable read," this concurrency strategy almost maintains the semantics. Repeatable read isolation is compromised in the case of concurrent writes. See https://sqlperformance.com/2014/04/t-sql-queries/the-repeatable-read-isolation-level for a discussion of this functionality.
   - `Nonstrict Read/Write` - Caches data that is sometimes updated without ever locking the cache. If concurrent access to an item is possible, this concurrency strategy makes no guarantee that the item returned from the cache is the latest version available in the database. This works well for data that changes and must be committed, but it does not guarantee exclusivity or consistency (and so avoids the associated performance costs). This strategy allows more than one transaction to simultaneously write to the same entity and is intended for applications able to tolerate caches that may at times be out of sync with the database.



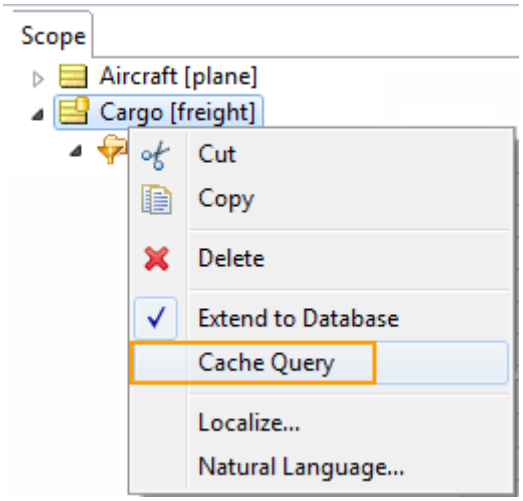
Set query caching on Entities in a Rulesheet

In a Rulesheet, caching can be set on an Entity and on a database filter. You can use either or both.

Note: Query caching is independent of entity caching's **Datastore Caching** setting.

To use query caching on an entity in a Rulesheet:

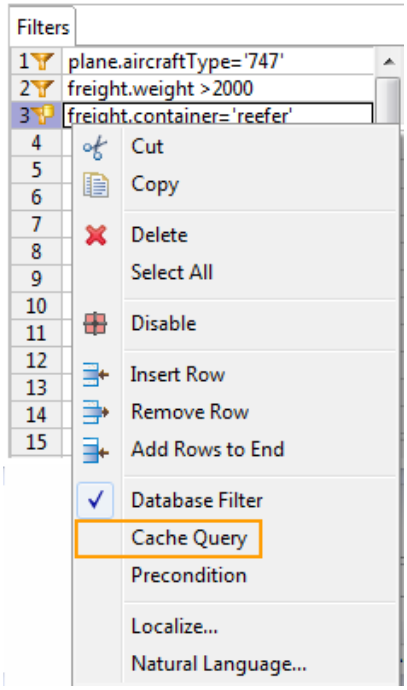
1. In a Rulesheet, choose **Advanced** view to show the **Scope** tab. Datastore-persistent entities have a database decoration.
2. Right-click on a datastore-persistent entity, and then choose **Extend to Database**.
3. Right-click again on the same datastore-persistent entity, and then choose **Cache Query**, as shown:



Set query caching on database filters in a Rulesheet

To use query caching on a database filter in a Rulesheet:

1. In a Rulesheet, choose **Advanced** view to show the **Scope** tab. Datastore-persistent entities have a database decoration.
2. On the **Filters** tab, right-click on a filter that references an entity extended to database, and then choose **Database Filter**. The filter is decorated with a database symbol.
3. Right-click on that filter again, and then choose to **Cache Query**, as shown:



Settings for EDC caching

The cache settings described in Specifying caching on Vocabularies and Rulesheets on page 139 can be combined to achieve your caching goals.

Legend:

- **Vocabulary** - Set caching on datastore persistent Entities in the Vocabulary
- **Scope** - Set query caching on Entities in a Rulesheet
- **Filter** - Set query caching on database filters in a Rulesheet

Settings	Description
[X] Vocabulary [ ] Scope [ ] Filter	Operates only on the entities in the Decision Service request payload. Corticon will retrieve all missing attribute data from the database (or from its entity cache if data already exists) for the requested entity instance(s). If these entities instances are associated with other entities, then these associated entities will also be placed in the entity cache.
[X] Vocabulary [X] Scope [ ] Filter	The incoming entity in the request payload will add to the entity cache, whereas the query cache will be populated with all records from the mapped database table (if no query filter is defined). **NOTE:** Typically not set on an entity. Instead, set either entity or query cache on the entity depending on the application scenario.
[X] Vocabulary [X] Scope [X] Filter	The incoming entity in the request payload will add to the entity cache, where the query cache will be populated with the filtered records from the mapped database table (defined by the filter criteria). **NOTE:** Typically not set on an entity. Instead, set either entity or query cache on the entity depending on the application scenario.

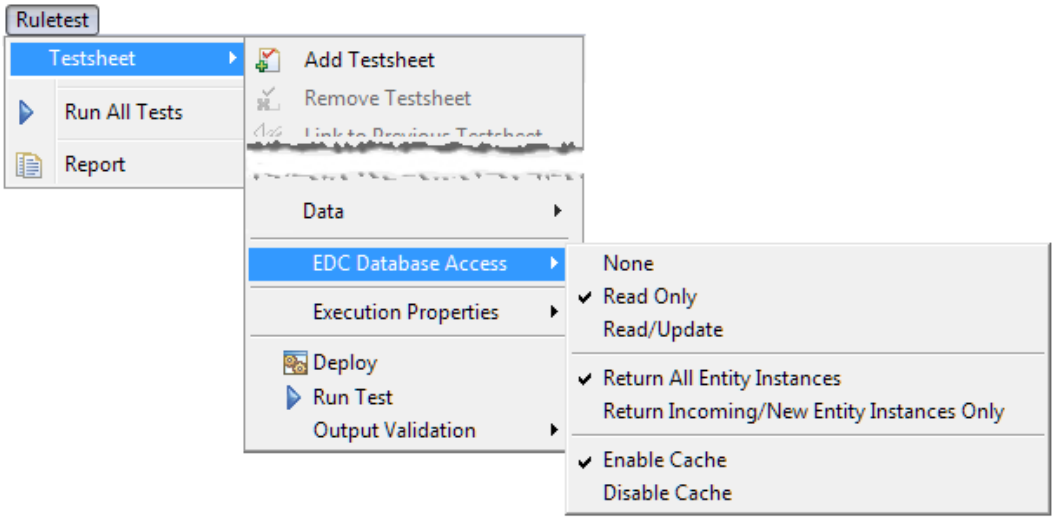
Working with database caches

Corticon's EDC provides functionality for enhanced database caching at runtime. Its cache is temporary data that duplicates data located in a database so that it can be repeatedly accessed with minimal costs in terms of time and resources. If an application must be certain not to get stale data, then it should not use caching. Caching is best used for reference data such as tax or actuarial tables.

What gets cached is based on settings in a project's Vocabulary and Rulesheets. Ruletests and deployed Decision Services let you choose to enable the requested caching. The first cache usage takes some overhead to establish the cache so that subsequent test runs get the benefit of very fast performance. When Studio or Server restarts, its in-memory cache(s) and on-disk cache files are cleared.

Testing caching on a Studio Ruletest to Run in Studio

Once you have Rulesheets and a Vocabulary that are prepared for database caching, choosing to enable cache on the Studio will perform the caching functions in the Studio's space. To enable caching on the Ruletest, choose the **Ruletest** menu command **Testsheet > EDC Database Access > Enable Cache**, as shown:

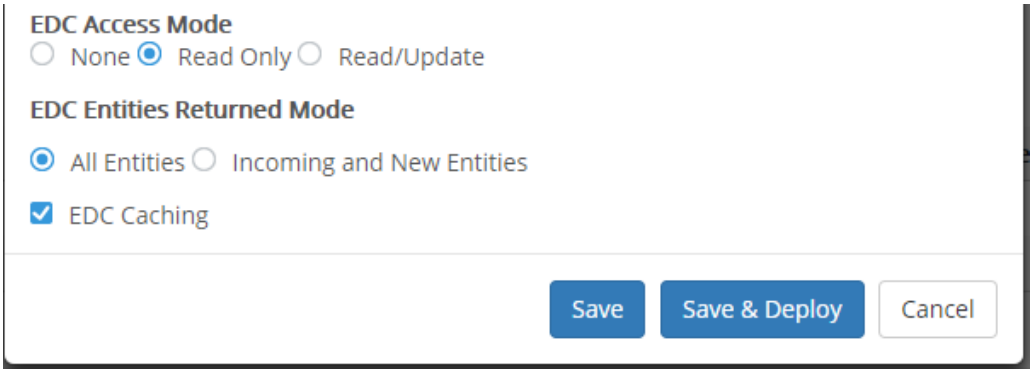


When you run the Ruletest in Studio, you can observe its performance against your Input. There are no local files that are user modifiable.

Executing a Studio Ruletest against a deployed Decision Service

You can run your Ruletests against the Decision Service deployed on Corticon Server where you can tune the cache configuration of each Decision Service instance. The optimal way to manage a cache-enabled Decision Service is as follows:

1. In Studio, package the Decision Service and its Datasource Configuration file in a location that is accessible from the Web Console user's machine. When you deploy a Decision Service to a Server together with its Datasource Configuration file, all the Vocabulary and Rulesheet cache choices that you specified are packaged in the Decision Service.
2. In a web browser, connect to the Web Console that manages the server where you will deploy the Decision Service–perhaps a production-quality machine reserved for testing.
3. In the Web Console, add a Decision Service. Locate the EDS file, and then on the **Database** tab, locate the Datasource Configuration file.
4. Choose the EDC database settings that were on the Ruletest, as shown:



5. Click **Save and Deploy**.

Note: Once deployed, you can run Ruletests in Studio by changing the Test Subject to Run against Server, and then choosing your deployed Decision Service. However, the **EDC Database Access** settings on the Ruletest are ignored. Instead, use the corresponding options on the deployed Decision Service through the Web Console.

If you want to tune the cache configuration, see Modifying a cache configuration on page 145.

Important: Turning caching on or off - If you want to enable or disable caching on a deployed Decision Service, the mechanisms of caching require that you undeploy and delete the Decision Service, and then add and deploy the Decision Service again with the cache enablement setting you want.

Cache files and configuration on Corticon Server

On Corticon Servers:

- Each Decision Service maintains its own cache, and cached data is never shared between Decision Services. Undeploying a Decision Service immediately clears its cache in memory and on disk.
- Each Decision Service records its configuration in its properties file, `[CORTICON_WORK_DIR]/etc/ehcache_<DSName>_v<M.m>.xml` where *<DSName>_v<M.m>* is the named and versioned Decision Service. For example, `[CORTICON_WORK_DIR]\etc\ehcache_Cargo_v0.16.xml`.

Properties in a cache configuration

The first run of the Decision Service on a Corticon Server creates its cache configuration file. The default properties and values for the deployed Decision Service `ehcache_Cargo_v0.16.eds` are in its configuration file `ehcache_Cargo_v0.16.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<ehcache>
  <diskStore path="[CORTICON_WORK_DIR]\Server\etc\Cargo_v0.16.xml" />
  <defaultCache
      overflowToDisk="true"
  timeToLiveSeconds="120"
  timeToIdleSeconds="120"
  eternal="false"
  maxElementsInMemory="1000" />
</ehcache>
```

where:

- `diskStore path` is the location where overflows to disk are written.
- `overflowToDisk` sets whether elements can overflow to disk when the in-memory cache has reached the `maxElementsInMemory` limit.
- `timeToLiveSeconds` is the maximum number of seconds an element can exist in the cache regardless of use. The element expires at this limit and will no longer be returned from the cache. If the value is 0, no TTL eviction takes place (infinite lifetime).
- `timeToIdleSeconds` is the maximum number of seconds an element can exist in the cache without being accessed. The element expires at this limit and will no longer be returned from the cache. If the value is 0, no TTL eviction takes place (infinite lifetime).
- `eternal` sets whether elements are eternal. When eternal is `true`, timeouts are ignored and elements are never expired.
- `maxElementsInMemory` is the maximum number of objects that will be created in memory. When set to 0, there is no limit.

Modifying a cache configuration

If you want to modify any of the cache configuration properties for a Decision Service deployed on Corticon Server, you need to follow these steps for each Decision Service instance, as illustrated for `ehcache_Cargo_v0.16.xml`:

1. Run the deployed Decision service with its cache enabled to create its default configuration file in `etc`.
2. Edit the file to specify your preferred property values and then save it.
3. Add the folder and the explicit filename, in this case `etc\ehcache_Cargo_v0.16.xml`, to the server classpath.
4. Edit the Decision Service's deployed Datasource Configuration file to add the location of the configuration file relative to the classpath as a property, as illustrated:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<decisionService>
    <datasources>
        <edc useForQueryService="false" description="" name="EDC">
            <connection-url>jdbc:progress:sqlserver://localhost:1433;
                            databaseName=PatientRecords</connection-url>
            <database-driver>com.corticon.database.id.MsSql2014</database-driver>
            <password>062046016058035029061039110</password>
            <username>061046</username>
            <properties>
                <property name="net.sf.ehcache.configurationResourceName"
                          value="..\..\..\..\..\etc\ehcache_Cargo_v0.16.xml"/>
            </properties>

        </edc>
    </datasources>
</decisionService>
```

   where `value` is the appropriate relative location.

5. Restart the Server to apply the changes.

Note: For more information about the settings and behaviors of Corticon's advanced EDC caching, see the Ehcache 2.4.3 documentation.

Metadata for Datastore Identity in XML and JSON Payloads

When Element attributes have extra information at the Entity Element level, data such as the datastore identity requires special handling as metadata because it is not an attribute in the Vocabulary. It is invalid to declare the datastore identity as an Element, as shown:

```
<?xml version="1.0" encoding="UTF-8"?>
<CorticonRequest decisionServiceName="MyDS">
  <WorkDocuments>
    <TestEntity1 id="TestEntity1_id_1">
      <databaseid>1</databaseid>   this is incorrect
      <testBoolean xsi:nil="true" />
      <testDate xsi:nil="true" />
      <testDateTime xsi:nil="true" />
      <testDecimal xsi:nil="true" />
      <testInteger xsi:nil="true" />
      <testString xsi:nil="true" />
      <testTime xsi:nil="true" />
```

```
    </TestEntity1>
  </WorkDocuments>
</CorticonRequest>
```

Adding Datastore Identity to an XML Payload

For an XML payload, `databaseid` is placed inside the Element, as shown:

```
<?xml version="1.0" encoding="UTF-8"?>
<CorticonRequest decisionServiceName="MyDS">
  <WorkDocuments>
    <TestEntity1 databaseid="1" id="TestEntity1_id_1">
      <testBoolean xsi:nil="true" />
      <testDate xsi:nil="true" />
      <testDateTime xsi:nil="true" />
      <testDecimal xsi:nil="true" />
      <testInteger xsi:nil="true" />
      <testString xsi:nil="true" />
      <testTime xsi:nil="true" />
    </TestEntity1>
  </WorkDocuments>
</CorticonRequest>
```

Adding Datastore Identity to a JSON Payload

In JSON formatting, the `#datastore_id` is placed in the `__metadata` section of the Entity, as shown:

```
{"name":"MyDS",
    "Objects": [{
        "testDate": null,
        "testDecimal": null,
        "testDateTime": null,
        "testString": null,
        "testBoolean": null,
        "testInteger": null,
        "testTime": null,
        "__metadata": {
            "#id": "TestEntity1_id_1",
            "#type": "TestEntity1",
            "#datastore_id": "1"
        }
    }]
}
```

For more information about datastore identity, see the topic Identity strategies on page 147.

Relational database concepts in the Enterprise Data Connector

Corticon's Enterprise Data Connector integrates its Decision Services with implementations of the relational database model.

Note: Identity and strategy concepts are general relational database concepts. Refer to your RDBMS brand's documentation for more information, especially the identity strategies that are specific to certain brands.

Identity strategies

Because EDC allows Studio and Server to dynamically query an external database during Rulesheet/Decision Service execution, the Vocabulary must contain the necessary key and identity information to allow Studio and Server to access the specific data required. There are two identity types which may be selected for each Vocabulary entity: application and datastore.

Application Identity

With application identity, the field(s) of a given table's primary key are present as attributes of the Vocabulary entity. As a result, application identity normally means that the table's primary key field(s) have some business meaning themselves; otherwise they wouldn't be part of the Vocabulary. The `Cargo` sample illustrates entities using application identities. In the case of entity `Aircraft`, the unique identifier (primary key) is `tailNumber`. In the database metadata, `tailNumber` is the designated primary key field. The presence in the Vocabulary of a matching attribute named `tailNumber` informs the auto-mapper that this particular entity must be application identity.

Datastore Identity

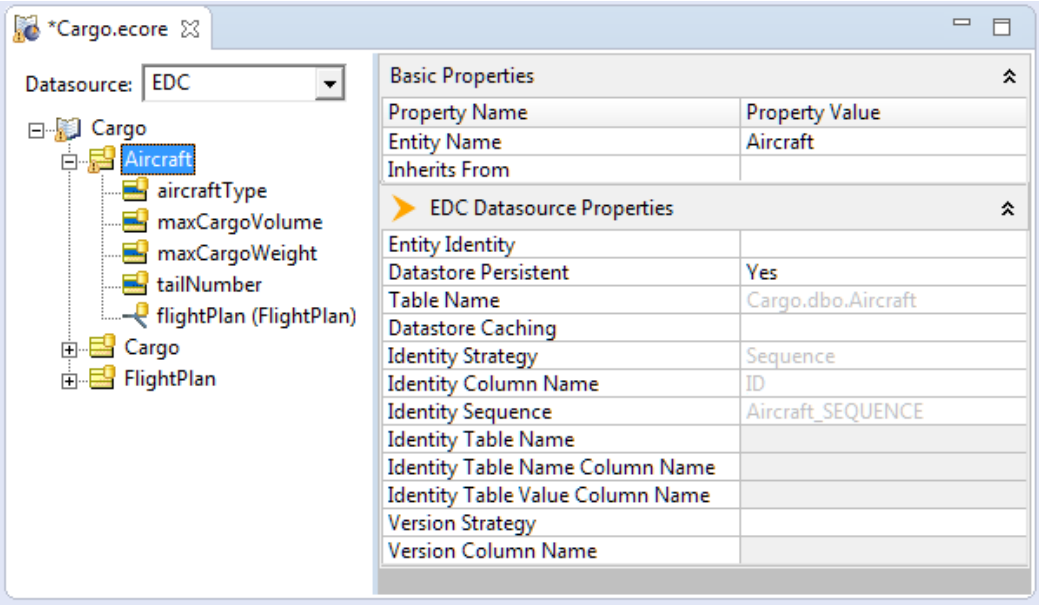
A Vocabulary entity uses datastore identity when it does not have an attribute that matches the database table's primary key field(s) . The table's primary key is effectively a *surrogate key* which really has no business meaning. If the designated primary key fields in the imported database metadata are not present as attributes in the Vocabulary entity, then the Vocabulary Editor will assume datastore identity and insert the table's primary key field(s) in the **datastore-identity:column** property.

We have modified our `Aircraft` table slightly to change the primary key. Previously, we assumed that `tailNumber` was the unique identifier for each `Aircraft` record – in other words, every aircraft must have a tail number and no two can have the same one. Let's assume now that this is no longer the case – perhaps `tailNumber` is optional (perhaps aircraft based in some countries don't require one?) or we somehow acquired two aircraft with the same `tailNumber`. So instead of `tailNumber`, we adopt a surrogate key for this table named `Aircraft_ID` that will always be non-null and unique. And since this field has no real business meaning (and we never expect to write rules with it), it isn't included in the Vocabulary.

Note: We can get to this state by clearing the database metadata, and then -- in the database - clearing (or deleting/recreating) the database. When we create the database schema again, the entity identities are all defaulted to datastore identities.

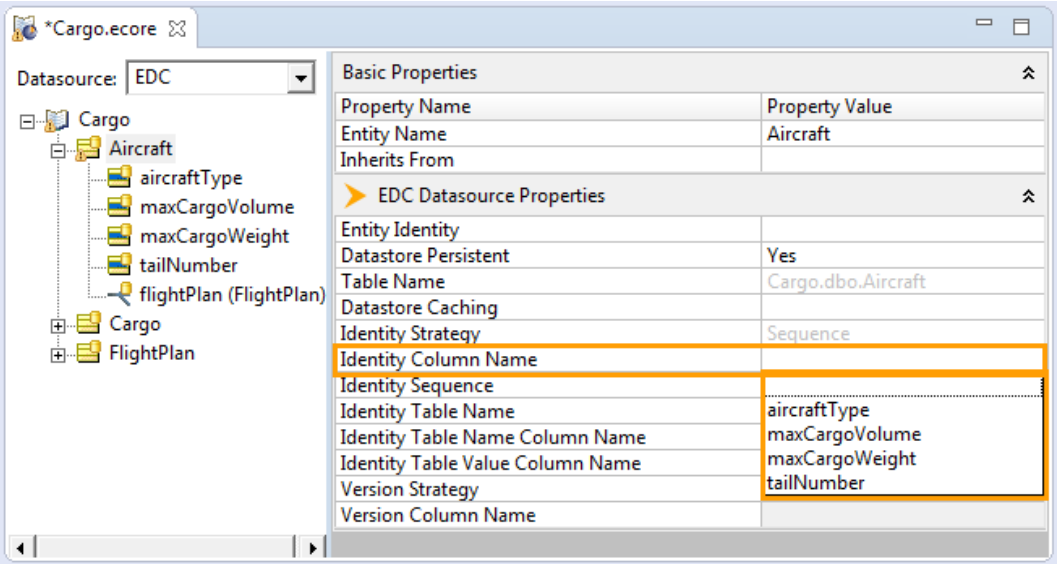
When the auto-mapper updated the schema, the Entity Identity was set to a `NULL`, and set the primary key field(s) in **Identity Column Name** as `ID`, as shown:

Figure 33: Automatic Mapping of Datastore Identity Column



If the auto-mapper does not detect the correct primary key in the metadata, we may need to manually select the field from the drop-down list, as shown:

Figure 34: Manual Mapping of Datastore Identity Column



By choosing datastore identity we are delegating the process of identity generation to Hibernate. That does not mean that we cannot control how it does this. The Vocabulary Editor offers the following ways to generate the identities:

- **Native** - Lets Hibernate choose the appropriate method for the underlying database. This usually means a Sequence in the RDBMS. Depending on the RDBMS you use, a sequence may require the addition of a sequence object or generator in the database.
- **Table** - Uses a table in the datastore with one row per table, storing the latest max id.
- **Identity** - Uses *identity* (Requires *identity* support in the underlying database.)

- **Sequence** - Uses *sequence* (Requires *sequence* support in the underlying database.)
- **UUID** - A UUID-style hexadecimal identity.

All of these strategies are database-neutral except for *sequence*. It is generally recommended that identity strategy be adopted for Vocabularies that are used to generate the database. When mapping to an existing database either *identity* or *sequence* strategies are typically used, depending on the database design.

Note:

These generators can be used for both datastore and application identities. The datastore identity is always using a strategy; if not explicitly set by the user, a default strategy is used. The application identity does not have a default strategy.

All strategies are using the integer data type with the exception of UUID which is using a string data type. If the type of the application identity attribute type does not fit the selected value strategy (for application identity), you get an alert.

For examples of proper syntax for datastore identities in query payloads, see the topic Metadata for Datastore Identity in XML and JSON Payloads on page 145

For a detailed discussion of this subject, refer to The Hibernate community documentation, section 5.1.2.2: Identifier generator.

Advantages of using Identity Strategy rather than Sequence Strategy

EDC offers options for assigning primary keys. For SQL Server databases, you might want an Identity strategy. For an Oracle database, you might choose a Sequence strategy. Consider the following points when deciding whether to use *identity* strategy or *sequence* strategy:

- When using the **Create/Update Database Schema** function in the Vocabulary, the sequences are generated automatically and tied to the **table id** fields on the database side. On the other hand, when using *sequence* strategy, the sequences are not generated during the **Create/Update Database Schema** process. If Corticon, at runtime, attempts to access a sequence and finds it missing, it will try to create it on the fly. But such a dynamic creation of sequences is tricky and does not always work properly.
- Using *identity* strategy should result in better performance when inserting a large number of records into the database. This is simply because the database I/O is cut in half since there is no need to retrieve the next unique id from the database prior to adding a new record.
- Using *sequence* strategy tends to not be compatible with read-only database access which may result in runtime exceptions.
- Using *identity* strategy makes a Vocabulary more portable across databases since not all databases support sequences.

Hibernate supports Sequence strategy for all databases; in a case where the database does not support it -- such as SQL Server -- Hibernate emulates it. However, in a case where the database does not support Identity strategy -- such as Oracle -- there is no emulation. This makes Sequence more portable.

Key assignments

Key designations occur automatically once an entity identity has been defined in the Vocabulary Editor.

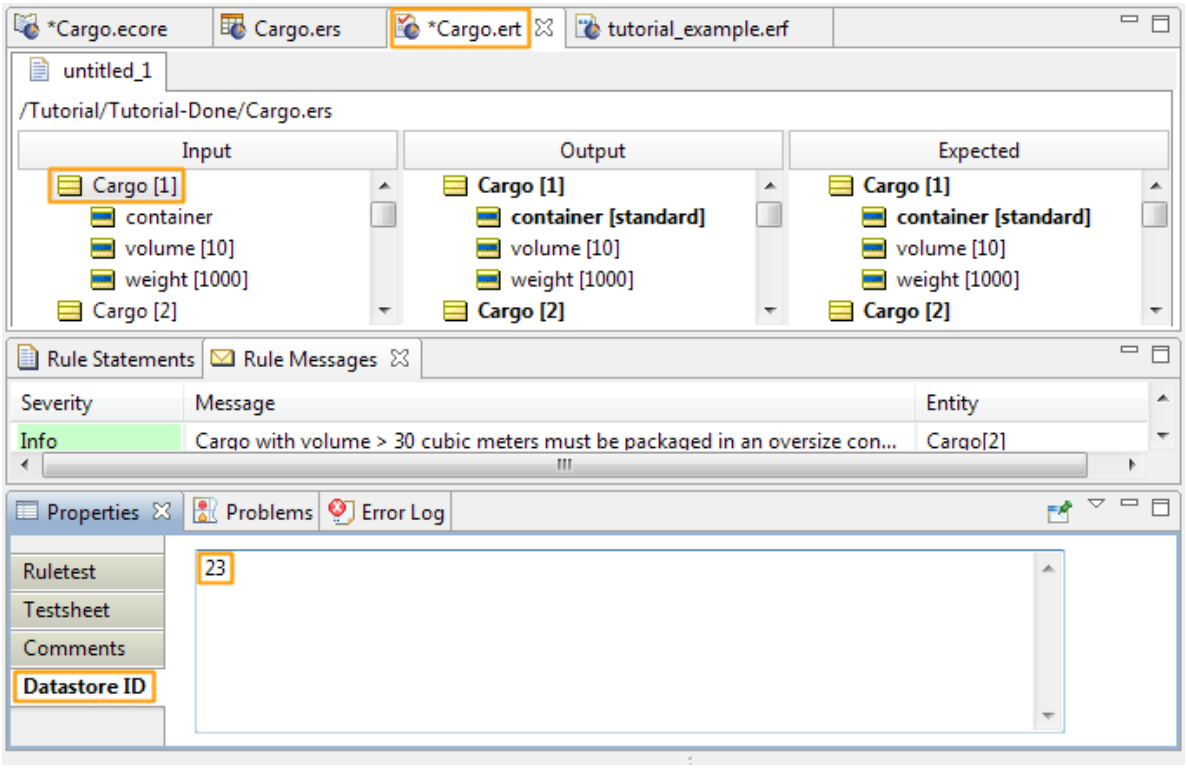
Primary Key

If the chosen (or auto-mapped) entity identity appears in the Vocabulary as an attribute (see Application identity), then that attribute receives an asterisk character to the right of its node in the Vocabulary's TreeView. Attributes with asterisks are part of the entity's primary key as shown in Automatic Mapping of Vocabulary Entity.

If the chosen (or auto-mapped) entity identity does **not** appear in the Vocabulary as an attribute see Datastore identity), then no attribute receives an asterisk character. None of the attributes in the Vocabulary are part of the entity's primary key, as shown in Automatic Mapping of Datastore Identity Column. This causes complications when testing and invoking Decision Services with connected databases. If no primary key is visible in the Vocabulary, then how do we indicate in an unambiguous way the specific records(s) to be used by the Decision Service?

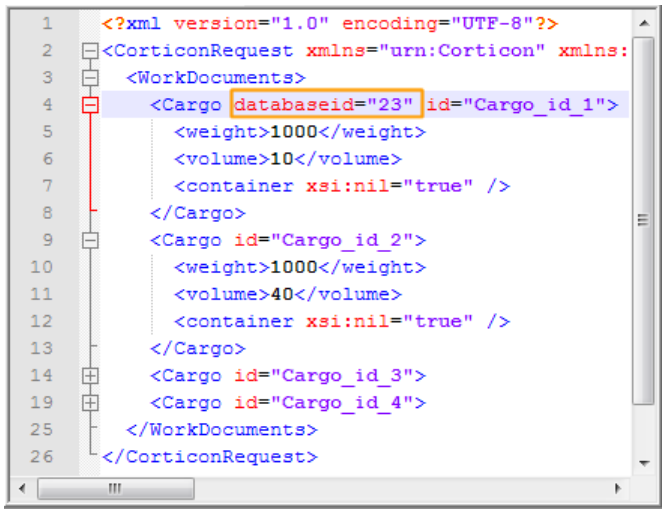
In the Studio Test, an entity using Datastore identity has its key set in the entity's **Properties** window. The following figure shows that the Ruletest was chosen. Right-clicking on first Cargo entity and choosing **Properties** on the menu opened the **Properties** tab where the **Datastore ID** side tab was selected. The value `23` was entered for the test:

Figure 35: Setting the Identity for Entities Using Datastore Identity



When we export the Ruletest to XML (**Ruletest > Testsheet > Data > Output > Export Response XML**) illustrates how this Database ID appears in the XML message. In the following figure, we see how the **Database ID** value is included in the XML as an attribute (an XML attribute, not a Vocabulary attribute). Your XML toolset and client may need to insert this data into a `CorticonRequest` message.

Figure 36: Datastore Identity inside the XML Request



Foreign Key

Foreign key relationships between database tables are represented in the Vocabulary via association mappings. As we see in Mapping EDC database relationships to Vocabulary Associations on page 123, the association mappings are entered (or auto-mapped) in the **Join Expression** field.

Composite Key

Multiple keys may be selected (if not auto-mapped) by choosing the **Select All** option, or by holding the **Control** key while clicking on all the items you want on the **Entity Identity** drop-down. If multiple selections are made, then all Vocabulary attributes will have asterisk characters to indicate that they are part of the primary key.

Conditional entities

Although all database properties will unconditionally be displayed, their applicability and enablement is often dependent upon the values of other properties.

Universally, EDC properties are applicable only for entities whose Datastore Persistent flags are set to Yes. For entities that are not datastore-persistent, all EDC properties for that entity, including EDC properties belonging to the entity's attributes and associations, will be disabled.

For datastore-persistent entities, fields that are applicable will be enabled and editable, while fields that are not applicable will be disabled and will have a light-gray background. The applicability of fields will change dynamically based on the values of other fields.

Generally, fields which are not applicable in a given context will be disabled; however, any values that were previously entered into those fields will be preserved notwithstanding their lack of applicability, even if the field itself is disabled. Specific rules governing applicability are detailed in Entity Properties, Attribute Properties and Association Properties below.

Dependent tables

Sometimes the existence of a record in one table is dependent upon the existence of another record in a related table. For example, a `Person` table may be related to a `Car` table (one-to-many). A car may exist in the `Car` table independent of any entry in the `Person` table. In other words, a car record does not require a related person – a physical object exists on its own. Likewise, a person record could exist without an associated car (the person might not own a car). These two tables are independent, even though a relationship/association exists between them.

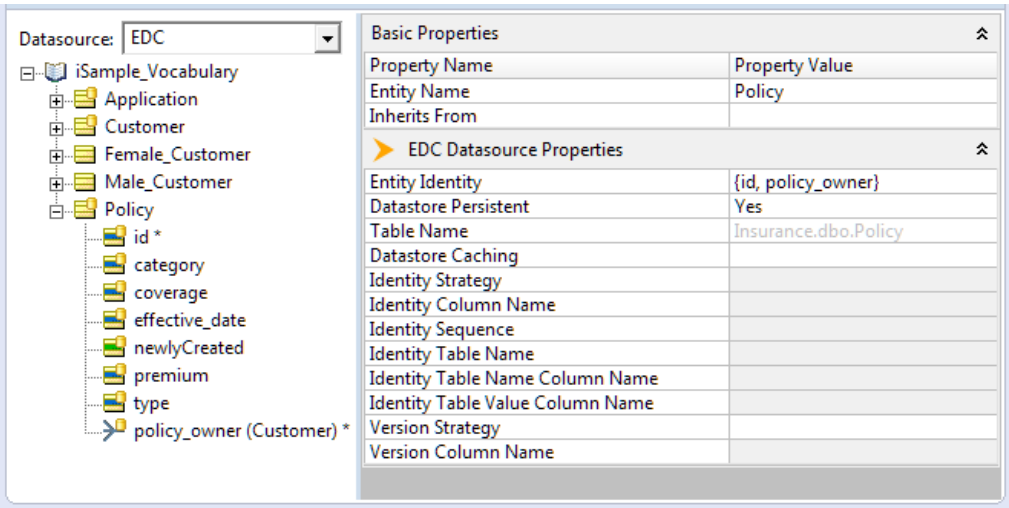
Some tables are not independent. Take `Customer` and `Policy` tables – if each policy record must have a person to whom the policy is "attached," we say the `Policy` table is dependent upon the `Customer` table. A person may or may not have a policy, but each policy must have a person.

Dependency normally comes into play when records are being removed from a table. In the first example, removing a person record has no effect on the associated car record. Although the person may no longer function as the car's owner, the car itself continues to exist. A car doesn't automatically vanish just because a person dies. On the other hand, removing a person *should* remove all associated policies. A person who switches insurance companies (and is deleted from its database) can expect his previous company to cancel and delete his old policies, too.

A Dependent table normally contains as part of its primary key the foreign key of the independent table. Since a Corticon Vocabulary represents a foreign key relationship as a **Join Expression** in the association mapping (see Mapping EDC database relationships to Vocabulary Associations on page 123), a dependent entity will have a composite key with the association name participating in the key.

As we can see in the following figure, the composite key contains both `id`, which is the application identity for the `Policy` entity and `policy_owner`, which is the association between `Customer` and `Policy` entities. This indicates that `Policy` is a dependent table, and that removing a `Customer` record will also remove all associated policy records.

Figure 37: Primary Key of a Dependent Table Includes the Role Name



How EDC handles transactions and exceptions

Here are a few points to note about Corticon's Enterprise Data Connector:

- Each Decision Service call is one database transaction. Transactions are not per operation, per Rulesheet, or per Ruleflow. Corticon does not currently provide for configuration of transaction management.

- The default transaction isolation level in Corticon EDC is the same as the default transaction isolation level of the database to which it is connected.
- When an exception occurs, the database transaction is rolled back, and the database reverts to the same state as before the Decision Service was called.

15

Advanced ADC Topics

This section describes advanced information on schemas, requirements, and SQL scripts for using ADC data sources.

For details, see the following topics:

- Mapping ADC database metadata
- Configuring ADC
- How Corticon is expressed in SQL
- Tips and techniques in SQL data integration

Mapping ADC database metadata

For rules in a Decision Service to read or write to a database, Vocabulary elements used in the rules must map to elements in the database. After you have imported the database metadata from an ADC Datasource, map each Vocabulary entity, attribute, and association to the appropriate table, column, and join expression.

Mapping data between a Corticon Vocabulary and an ADC relational database is not always perfect. When there are issues, you need to review the mappings to resolve incomplete or conflicting mapping data.

Smart matching will infer precisely matched table name and entities, and then seek column names that match attributes in each matched table. Join expressions are inferred from matched tables and columns.

Note: Primary Key - Each database table's primary key is not inferred as the Vocabulary's Entity Identity in each Entity. These values must be set manually.

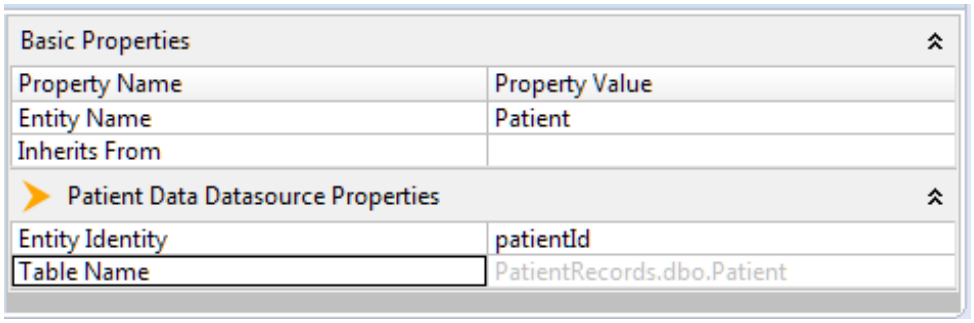
Note: When you are using an EDC datasource, its entity identity and decorations on the vocabulary icons are not relevant to the ADC datasource. ADC adds no decorations to the icons.

Mapping ADC database tables to Vocabulary Entities

Not all Vocabulary entities must be mapped to corresponding database tables - only those entities whose attribute values need to interact with the external database should be mapped.

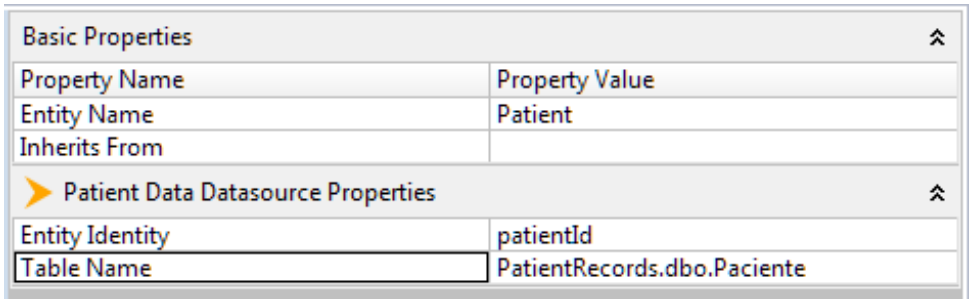
In this example, database metadata containing a table named `Patient` was imported. Because the table's name spelling matches the name of the entity `Patient`, the **Table Name** field was mapped automatically, and displayed in light gray, as shown:

Figure 38: Smart match mapping of Vocabulary Entity



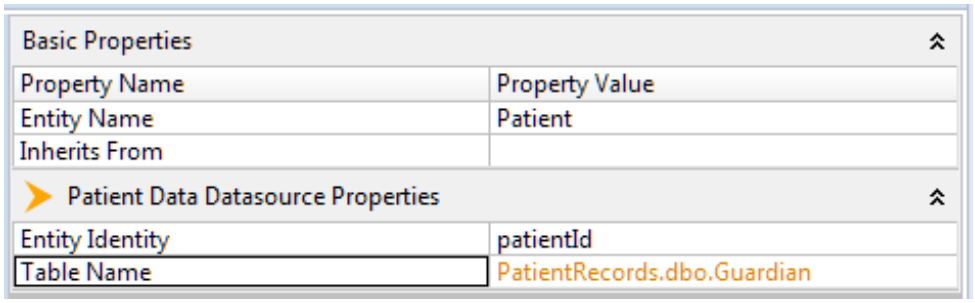
If the automatic mapping feature fails to detect a match for any reason (different spellings, for example), then you must make the mapping manually. In the **Table Name** field, use the drop-down list to select the appropriate database table to map. The selection is displayed in black, as shown:

Figure 39: Manual Mapping of Valid Vocabulary Entity



If you choose to type in the name of a table and that table is not in the database's metadata, the property value displays in orange, as shown:

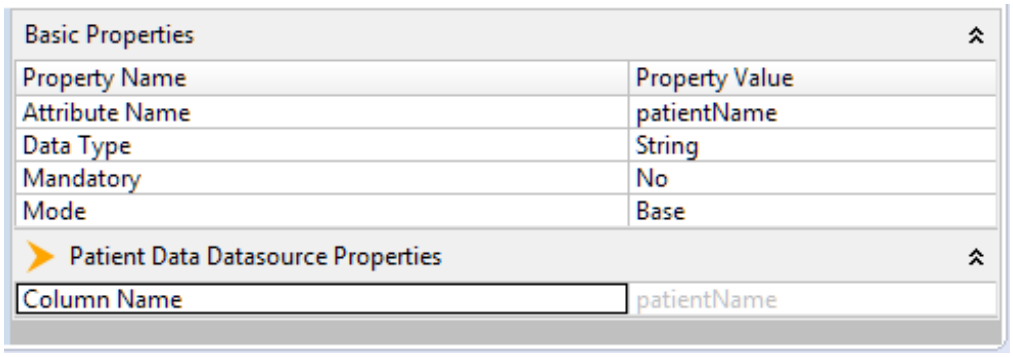
Figure 40: Manual Mapping of Invalid Vocabulary Entity



Mapping ADC database fields to Vocabulary Attributes

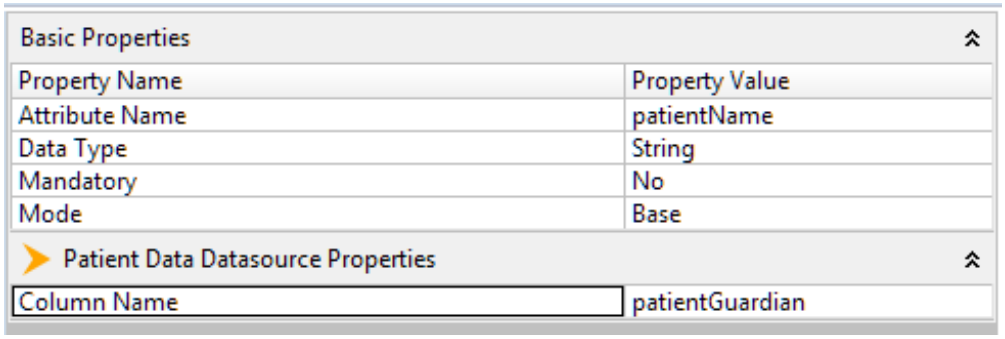
Mapping of attributes is similar to the way it works for entities. A smart match displays in grey, as shown:

Figure 41: Smart match mapping of Vocabulary Attribute



If an automatic match is not made by the system, then select the appropriate field name from the drop-down in **field:column** property, as shown:

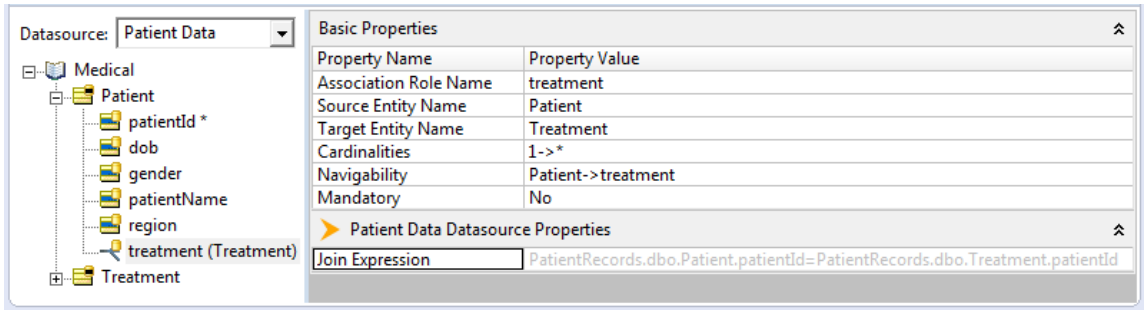
Figure 42: Manual Mapping of Vocabulary Attribute



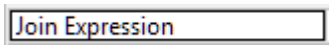
A preferred, valid name displays in black. If you enter a non-existent column name or select a name assigned to another column, the value displays in orange.

Mapping ADC database relationships to Vocabulary Associations

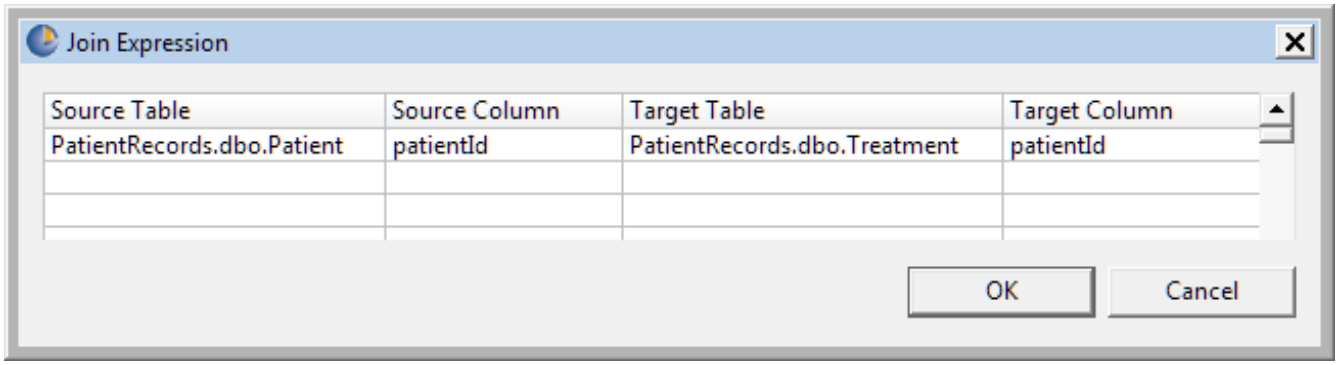
Automatic mapping of associations works substantially the same as entities. However, rather than entry text boxes and pulldowns for mappings, a more visual approach is provided. If an automatic match is made by the system, it is displayed in grey as shown:



If you want to revise the join expression, click on the **Join Expression** Property Name, as shown:



The **Join Expression** dialog box opens with a deconstruction of the join expression, as shown:



Use the pulldown lists in each column to refine the join expression. You can add lines to define complex join expressions where appropriate. As all revised join expressions are not validated, they are always displayed in black.

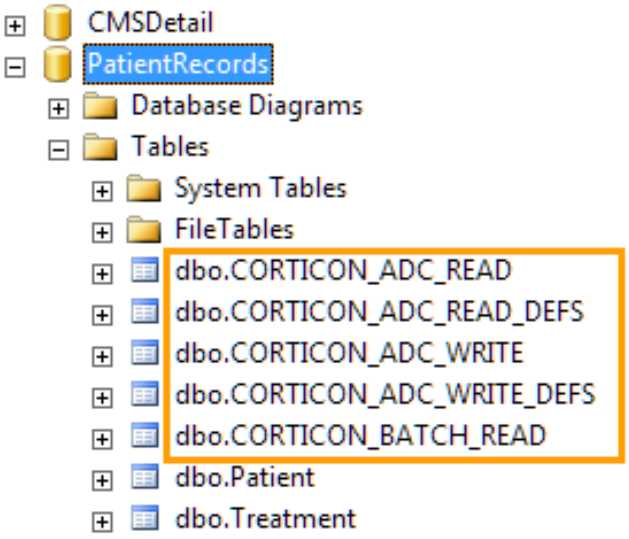
Note: The join expression is used by ADC to form associations in memory. The join expression is parsed by ADC -- it is not sent to the database server as part of a query.

For more information and examples of complex joins, see Associations as join expressions on page 102

Configuring ADC

The queries used in the data integration samples are stored in several tables in the database declared as the one to use for the query service:

Figure 43: Query Tables in SQL Server



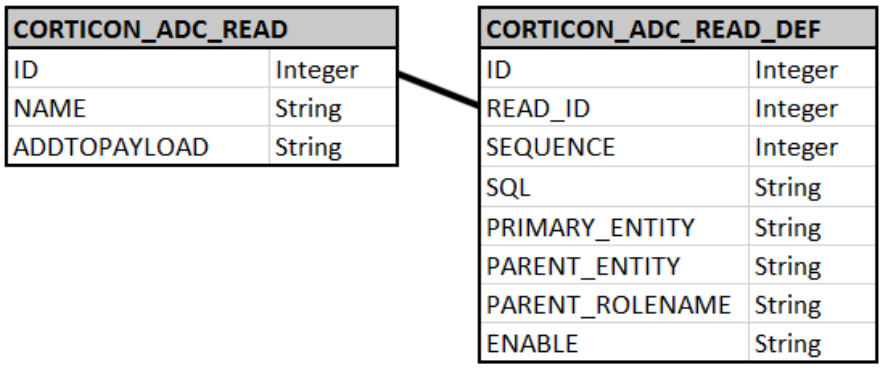
Note: As ADC is set to find specific names, the table and column names must not be modified.

Configuring ADC reads

The database schema that ADC reads use is illustrated in the following diagram.

A core operation that ADC performs is retrieving data using the `CORTICON_ADC_READ` table. Each `CORTICON_ADC_READ` row instance can use a different Datasource.

Figure 44: Database Schema for Corticon ADC Read Service Callouts



Note: The schema notes in the following tables are brief. For more details about a column name, see Configuration details on page 162

Table 10: `CORTICON_ADC_READ` Table

Column Name : DataType	Note
`ID` : Integer	The **Primary Key** for the Table which then gets propagated down to each `CORTICON_ADC_READ_DEFS` record.
`NAME` : String	A logical name that you want to associate with this `CORTICON_ADC_READ`. This is the name that will be specified inside of each Service Call-out's **Runtime Properties** tab for the appropriate **Query Name**.
`ADDTOPAYLOAD` : String (`true` or `false`)	Controls whether all data retrieved from the read will be added to the response payload. If this value is null or any value other than `true`, the default value is `false`.

Table 11: `CORTICON_ADC_READ_DEFS` Table

The `CORTICON_ADC_READ_DEFS` and `CORTICON_ADC_WRITE_DEFS` Tables are the key Tables for ADC. They contain the most pertinent information that ADC needs to perform its duties.

Column Name : DataType	Note
`ID` : Integer	The **Primary Key** for the Table.
`READ_ID` : Integer (required)	Foreign Key back to `CORTICON_ADC_READ.ID` column. There can be many `CORTICON_ADC_READ_DEFS` associated with a `CORTICON_ADC_READ` record.

Column Name : DataType	Note
SEQUENCE : Integer (required)	The integer value that specifies the order of execution of each CORTICON_ADC_READ_DEFS within a given CORTICON_ADC_READ_ID.
SQL : String (required)	An SQL Statement, a template to be used for the current CORTICON_ADC_READ_DEFS operation.
PRIMARY_ENTITY : String (required)	The Corticon Entity to which the SQL statement will map.
PARENT_ENTITY : String and PARENT_ROLENAME : String (optional)	The values needed to create an Association between the Parent Entity (PARENT_ENTITY) to the Target Entity (PRIMARY_ENTITY) through Association Role Name (PARENT_ROLENAME).
ENABLE : String (true or false)	Suppresses or allows the CORTICON_ADC_READ_DEFS to execute. If this value is null or any value other than false, the default value is true.

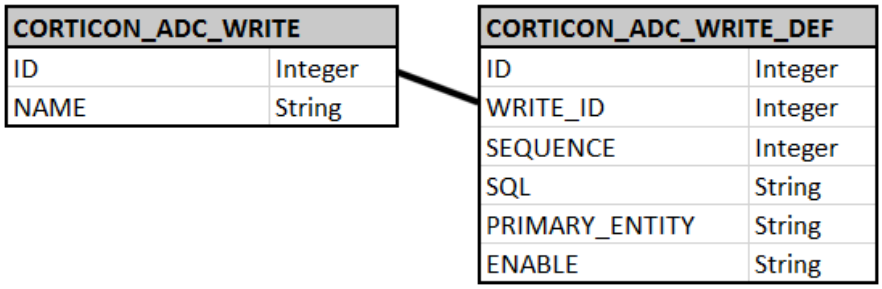
Note: For more about the format of Corticon's queries, see How Corticon is expressed in SQL on page 163.

Configuring ADC writes

The database schema that ADC writes use is illustrated in the following diagram.

A core operation that ADC performs is updating data using the CORTICON_ADC_WRITE table. Each CORTICON_ADC_WRITE row instance can use a different Datasource.

Figure 45: Database Schema for Corticon ADC Write Service Callouts



Note: The schema notes in the following tables are brief. For more details about a column name, see Configuration details on page 162

Table 12: CORTICON_ADC_WRITE Table

Column Name : DataType	Note
ID : Integer	The **Primary Key** for the Table which then gets propagated down to each CORTICON_ADC_WRITE_DEFS record.
NAME : String	The logical name to associate with this CORTICON_ADC_WRITE. This is the name that will be specified in a Ruleflow Service Call-out's **Runtime Properties** tab as the **Query Name**.

Table 13: CORTICON_ADC_WRITE_DEFS Table

The CORTICON_ADC_READ_DEFS and CORTICON_ADC_WRITE_DEFS Tables are the key Tables for ADC. These Tables contain the most pertinent information for the ADC to perform its duties.

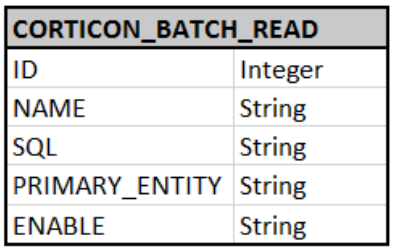
Column Name : DataType	Note
ID : Integer	The **Primary Key** for the Table
WRITE_ID : Integer	Foreign Key back to CORTICON_ADC_WRITE.ID column. There can be many CORTICON_ADC_WRITE_DEFS associated with a CORTICON_ADC_WRITE record.
SEQUENCE : Integer	The integer value that specifies the order of execution of each CORTICON_ADC_WRITE_DEFS within a given CORTICON_ADC_WRITE_ID when several CORTICON_ADC_WRITE_DEFS are associated with a CORTICON_ADC_WRITE record.
SQL : String	SQL Statement used as a template for this CORTICON_ADC_WRITE_DEFS operation.
PRIMARY_ENTITY : String	The Entity name that will be used to look up all instances of this Entity type from working memory in which variable substitution will be applied to the SQL statement to create one INSERT or UPDATE statement per Entity instance.
ENABLE : String (true or false)	Suppresses or allows the CORTICON_ADC_WRITE_DEFS to execute. If this value is null or any value other than false, the default value is true.

Note: For more about the format of Corticon's queries, see How Corticon is expressed in SQL on page 163.

Configuring batch

The database schema that Batch configurations uses is illustrated in the following diagram.

Figure 46: Database Schema for Corticon Batch Reads



Note: The schema notes in the following tables are brief. For more details about a column name, see Configuration details on page 162

Table 14: `CORTICON_BATCH_READ` Table

The `CORTICON_BATCH_READ` table describes a batch query.

Column Name : DataType	Note
`ID` : Integer	The **Primary Key** for the Table.
`NAME` : String	The name of this batch read operation.
`SQL` : String	The SQL statement that is a batch operation associated with this Decision Service.
`PRIMARY_ENTITY` : String (required)	The Corticon Entity to which the SQL statement will map.
`ENABLE` : String (`true` or `false`)	Suppresses or allows the `BATCH_READ` to execute. If this value is null or any value other than `false`, the default value is `true`.

Note: For more about the format of Corticon's queries, see How Corticon is expressed in SQL on page 163.

Configuration details

Several schema details note that they use **Variable Substitution**, Corticon's technique for in-memory data to dynamically create SQL statements using a template, as discussed in How Corticon is expressed in SQL on page 163

Columns described in Corticon query schemas are detailed as follows:

- **`Sequence`** - Several `CORTICON_ADC_READ_DEFS` or `CORTICON_ADC_WRITE_DEFS` might be associated with a `CORTICON_ADC_READ` or `CORTICON_ADC_WRITE` record. The values are typically strictly sequenced ascending values such as (1,4,6,7). If a value is not unique ,such as (1,4,4,7), the

CORTICON_ADC_READ_DEFS or CORTICON_ADC_WRITE_DEFS might not fire in the same way in the next execution.

- **SQL** - In SQL READ Statements, you can incorporate a complex WHERE clause using Corticon's Variable Substitution, so that can specify values in the SQL that will be replaced with corresponding values based on what is currently in the working memory of the execution.
- **SQL** - In SQL WRITE Statements, you can use Variable Substitution to add Primary Entity values directly into the SQL. Since the structure of an INSERT and UPDATE statement are different from a SELECT statement, Variable Substitution does not aggregate all values to create one SQL statement – instead, it will use the SQL as a template to create a SQL statement for each Primary Entity instance.
- **PRIMARY_ENTITY** - The results from the SQL Statement need to be converted to map to Corticon Entities. ADC does not automatically *create* a new instance of the Entity in memory. First, it will determine if that Entity is already in memory, and--if it is not already in memory--a new Entity instance of type (PRIMARY_ENTITY) will be created, using ICcDataObjectManager.createEntity(<PRIMARY_ENTITY>. Then, the Column Values for that Row will be added into that new instance of the Entity. Duplication of Entity instances is prevented when the rules engine checks to see whether that Entity instance is already in memory. This is done by comparing each in-memory Entity instance's "Entity Identity" values with the values retrieved for that row. If the instance already exists, then it will use that instance, and then merge the Column Values into that Entity instance.
- **PARENT_ENTITY** and **PARENT_ROLENAME** -

  The Association's Join Expression is critical to the mapping of Associations between the PARENT_ENTITY and the PRIMARY_ENTITY. ADC parses the Join Expression to determine which Attributes in the Parent Entity need to match which Attributes in the Primary Entity. For each Primary Entity retrieved, an algorithm is used to match these values between two different Entities. If there is a match, the Primary Entity is added to the Parent Entity's Association as defined by the Parent Role Name.

- **Enable** - For testing purposes, you may want to test some CORTICON_ADC_READ_DEFS or CORTICON_ADC_WRITE_DEFS out of all the ones associated with the CORTICON_ADC_READ or CORTICON_ADC_WRITE. You can add all your CORTICON_ADC_READ_DEFS and CORTICON_ADC_WRITE_DEFS and then incrementally expand the retrieval, while testing each step.

How Corticon is expressed in SQL

SQL Queries provide tremendous power to the access, retrieval, and management of data records. If you have done the ADC and Batch samples, you have used SQL queries and saw how using a different one causes different processing.

Variable Substitution is Corticon's technique for in-memory data to dynamically create SQL statements with the SQL value as a template in the tables CORTICON_ADC_READ_DEFS, CORTICON_ADC_WRITE_DEFS, and CORTICON_BATCH_READ. This is expressed in statements as a value in curly braces, like this: {Patient.patientId}.

For example, the sample batch queries are:

```
SELECT patientId from Patient
SELECT patientId from Patient WHERE region IN ({Patient.region})
```

The first SELECT query selects all patients. The second uses a parameter that will match on a specified region value in a Patient record.

You might want to create a query that uses multiple parameters such as:

```
SELECT patientId FROM Patient WHERE region IN ({Patient.region}) AND gender IN
({Patient.gender})
```

that would be specified in the Web Console batch configuration like this:

Query Parameters

Name	Value
Patient.region	
Patient.gender	

The READ and WRITE queries allow for multiple statements by exposing values in the `CORTICON_ADC_READ` and `CORTICON_ADC_WRITE` tables that link to a corresponding `CORTICON_ADC_READ_DEFS` and `CORTICON_ADC_WRITE_DEFS` table for the sequence of steps for the SQL statements.

Here, the sample's Ruleflow property for the Service Call-out's chose the `CorticonADC.read` service and the Query name **IndicatedPatients**. That referenced `ID=2` in the `CORTICON_ADC_READ` table:

ID	NAME	ADD_TO_PAYLOAD
1	AllPatients	true
2	IndicatedPatients	true
3	TreatmentDetails	true

That called to `CORTICON_ADC_READ_DEFS` with `READ_ID = 2` to perform its `SEQUENCE` of steps 1 and 2:

ID	READ_ID	SEQ...	SQL	PRIMARY_ENTITY	PARENT_ENTITY	PARENT_R
1	1	1	SELECT * FROM Patient	Patient	*NULL*	*NULL*
2	1	2	SELECT * FROM Treatment WHERE patientId IN ({Patient.patientId})	Treatment	Patient	treatment
3	2	1	SELECT * FROM Patient WHERE patientId IN ({Patient.patientId})	Patient	*NULL*	*NULL*
4	2	2	SELECT * FROM Treatment WHERE patientId IN ({Patient.patientId})	Treatment	Patient	treatment
5	3	1	SELECT * FROM TreatmentDetails WHERE treatmentCode IN ({Treatment.medicalCode})	Treatment	*NULL*	*NULL*
NULL	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*

Tips and techniques in SQL data integration

The following sections provide insights into techniques and behaviors you might find useful:

- Use of an IN ( ) instead of comparison operators in WHERE clause on page 164
- Inserting or updating multiple rows into specific database table(s) on page 165
- Multiple ADC instances can be added to one or many Ruleflows on page 165
- Each ADC task can use a different Datasource on page 165
- Information when execution fails on page 166

Use of an IN ( ) instead of comparison operators in WHERE clause

Use an `IN ( )` clause instead of an = sign in your `WHERE` clause. They mean the same thing; however, the `IN ( )` clause can handle multiple values, while the = sign can only handle one value.

Consider here are three A Entities in memory. That means there are three values for { A.id }. In the following SQL note that the one with the IN ( ) is valid while the = sign is not:

```
Select * from Patients where patientId IN ( 1, 2, 3 ) Valid
Select * from Patients where patientId = 1, 2, 3      Invalid
```

You cannot use an IN clause with <, <=, >, and =>. To prevent invalid SQL through variable substitution with <, <=, >, and =>, there can only be one instance of the Entity in working memory.

Inserting or updating multiple rows into specific database table(s)

When a Ruleflow establishes an ADC Service Call-out using the CorticonADC.write, ADC uses the metadata inside CORTICON_ADC_WRITE, and CORTICON_ADC_WRITE_DEFS tables to determine which Entities in the Vocabulary will be used to insert into which database table.

The core Table that contains the data about which Entity or Entities will be inserted into the Database is in the CORTICON_ADC_WRITE_DEFS table. This section describes how the SEQUENCE, SQL, PRIMARY_NAME are used in one or multiple CORTICON_ADC_WRITE_DEFS to insert multiple records into the intended table.

Much like the CORTICON_ADC_READ_DEFS' SEQUENCE field, the CORTICON_ADC_WRITE_DEFS' SEQUENCE field determines in which order the CORTICON_ADC_WRITE_DEFS will fire. For each CORTICON_ADC_WRITE_DEFS' SQL, there is a PRIMARY_ENTITY, which is used to create individual Insert Statements to be used by the database.

Variable substitution is used to substitute the PRIMARY_ENTITY values into the SQL Statement.

Example:

```
SQL = UPDATE Treatment SET approved={Treatment.approved}
             WHERE treatmentId={Treatment.treatmentId}
PRIMARY_ENTITY = Treatment
```

For every instance of Treatment in memory a new SQL Statement will get created using those values inside the Treatment instance.

The user controls the SQL statement, and can customize an INSERT SQL to match the Identity Strategy appropriate for a particular Database:

- In Oracle, Database Sequences are used to set the Primary Keys. You need to create your own Database Sequence and add that Sequence Name to the SQL statement.
- In SQL Server, you can just set your Table to use Identity strategy to populate the Primary Key.

Note: Because you have control over the SQL, you can inject Database Functions directly in the SQL that are unrelated to Corticon, such as a sysdate function.

Multiple ADC instances can be added to one or many Ruleflows

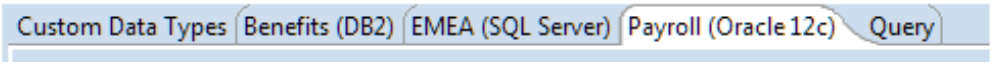
There is no restriction on how many ADC instances you can have in a Ruleflow. Its position on the Ruleflow canvas is based on your use case. When retrieving extra data that is only needed in certain cases, you can put an ADC instance inside a Branch that will only fire under certain conditions. Similarly, you can control whether a Ruleflow execution writes and where it writes..

Each instance of the ADC works independently to do what it is assigned to do.

Each ADC task can use a different Datasource

Each instance of an ADC can call any CORTICON_ADC_READ or CORTICON_ADC_WRITE operation, and, for each CORTICON_ADC_READ and CORTICON_ADC_WRITE, there is a Datasource configuration.

In the following illustration, the root level of the Vocabulary shows tabs for the connections to four datasources:



The Query Datasource is shared by all ADC Datasources.

Information when execution fails

Various errors can occur during the execution of the ADC. Some common issues are:

- `CORTICON_ADC_READName` or `CORTICON_ADC_WRITEName` does not exist.
- Bad SQL statement, possibly due to variable substitution issues.
- Bad Join Statement definition for an association.
- Failed to connect to the Datasource.

Whatever the type of error, execution will not only stop on the service callout, but for the entire execution. If there is an issue in the service callout, then current working memory could be incomplete or corrupted. Either way, the safest play is to stop all execution.

An entry is made in the Corticon Log with the Exception, and a `CcRuleMessage -> Violation` message added to the Response.

16

Advanced REST Datasource Topics

This section describes advanced information on connection requirements, and mapping a REST data source, and then filtering what will return to the Corticon rules engine.

The open style of REST data sources can present daunting and cryptic information. The authors of a well-formed REST API provide guidance to their users that:

- Describe its authentication and, if needed, where to get credentials.
- Documentation in HTML that describe usage, access, and constraints
- A schema of the data types, columns with unique data appropriate as keys, and relations between columns

A good example of a well-formed and presented REST API, see Open Weather Map

Note: For more information about the Progress DataDirect Autonomous REST Connector for JDBC, see topics in its online help at https://documentation.progress.com/output/DataDirect/jdbcautoresthelp.

For details, see the following topics:

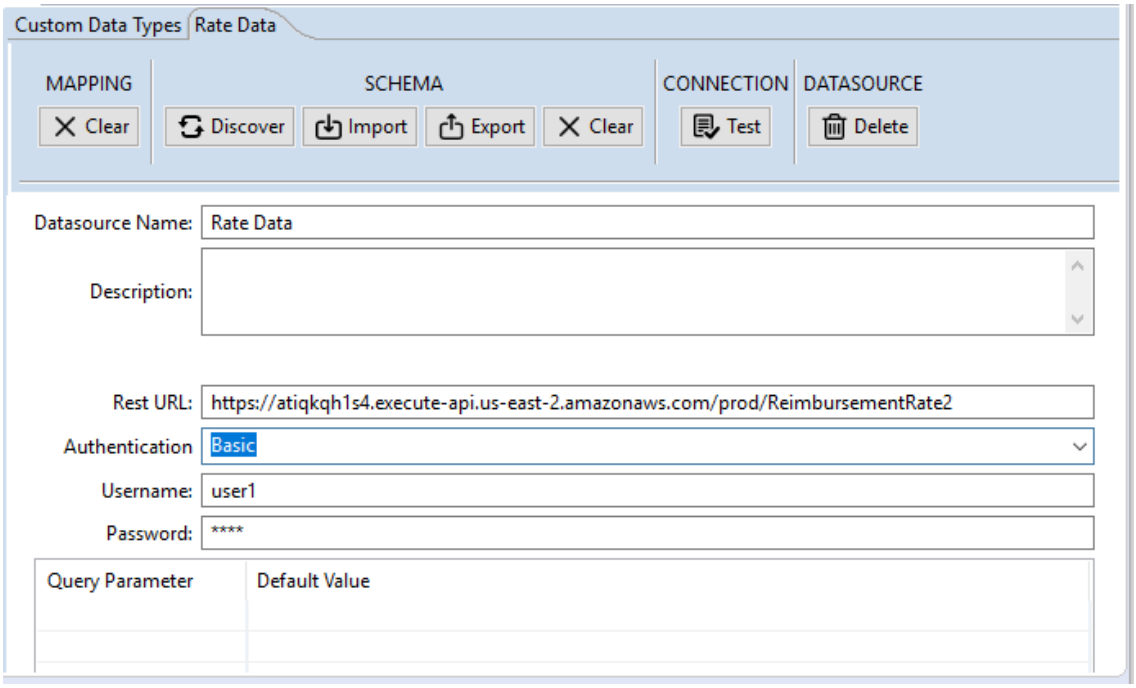
- Authentication on REST Service connections
- Parameters on REST Service connections
- Import REST Datasource metadata into a Vocabulary
- Mapping REST Service metadata

Authentication on REST Service connections

When you choose to create a REST Datasource, a new REST Service tab is created that has authentication set to *none*. You need to adjust the authentication parameter if you are provided credentials for basic or token authentication for the connection. The configuration of each of these is:

- **BASIC Authentication**: Credentials are used to access the REST service as a configured user associated with the REST Datasource, and then these credentials are used for all calls to the REST endpoint.

  Figure 47: REST Connectivity sample using Basic authentication



  The **Username** identifies a user value in the REST Datasource, and its **Password**. The sample Datasource has `pwd1` for `user1`. The credentials are encrypted when they are exported for deployment. When you connect to a URL of a REST API that requires basic authentication, you then add the credentials to the connection definition, as illustrated:

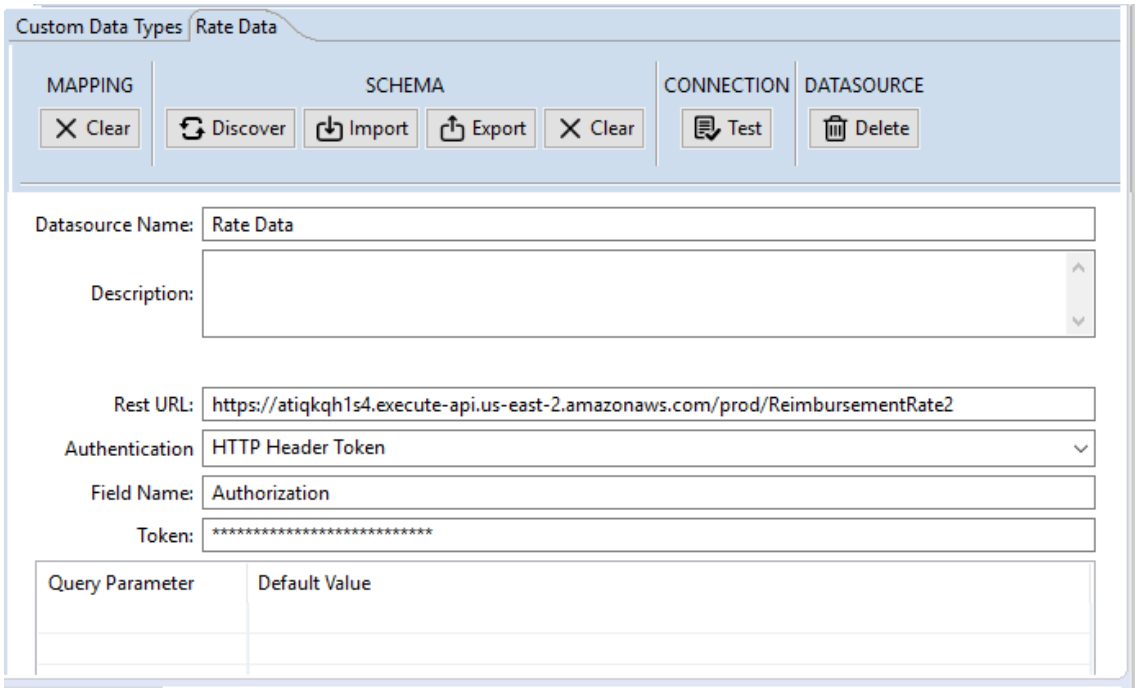
Figure 48: Datasource Configuration file of REST Connectivity sample using Basic authentication

```
1  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2  <decisionService>
3      <datasources>
4          <rest authentication-type="BASIC" name="Rate Data">
5              <connection-url>https://atiqkqh1s4.execute-api.us-east-2.amazonaws.com/prod/ReimbursementRate2<
6              <password>062056007120</password>
7              <username>059060006059101</username>
8          </rest>
9      </datasources>
10 </decisionService>
11
```

- **Token Authentication**: A static string Token value can be associated with a Corticon REST Datasource. The user obtains an appropriate token from the REST service, and then saves it in the `datasource.xml` file. It is then used for all calls to the REST endpoint. A Token can be passed inside an HTTP Header or as a parameter on a URL. The REST Service must declare which token authentication mechanism it will use.

- **HTTP Header Token** - Set the property that specifies the name of the HTTP header used for authentication. Input as **Field Name**, typically defaulted to `Authorization` in the Datasource and its **Token** (in this example `Bearer 12345678901234567890`), as shown:

Figure 49: REST Connectivity sample using Token authentication by header



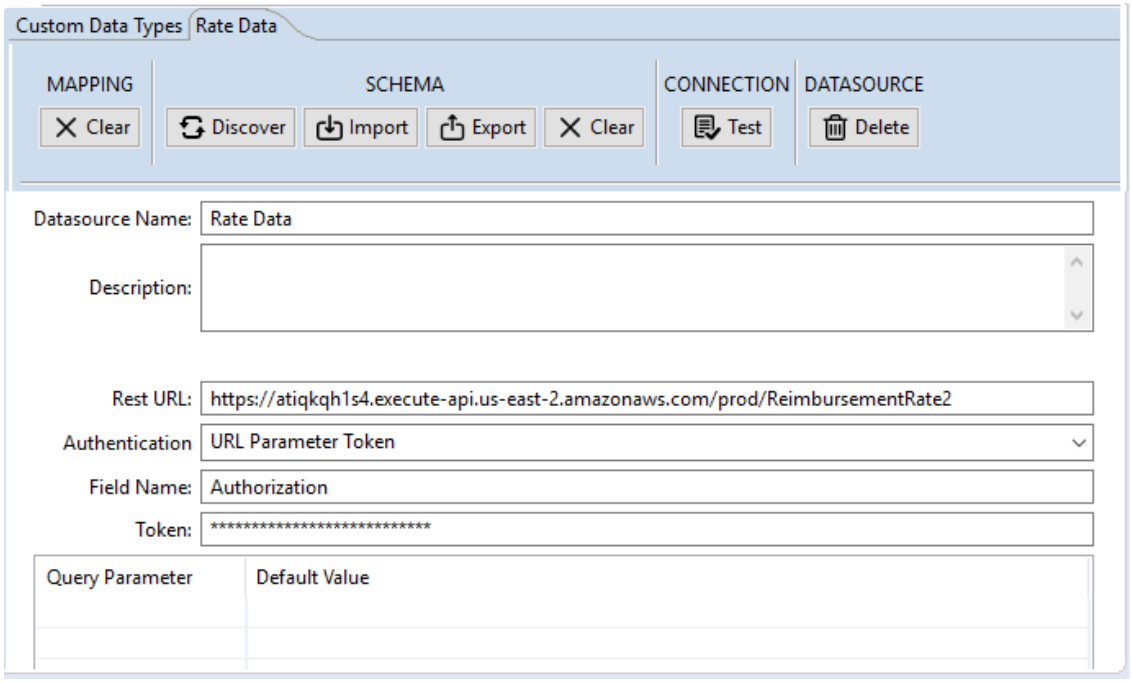
The credentials are encrypted when they are exported for deployment, as illustrated:

Figure 50: Datasource Configuration file of REST Connectivity sample using HTTP Header Token authentication

```
1  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2  <decisionService>
3      <datasources>
4          <rest authentication-type="HTTPHEADER" field-name="Authorization" name="Rate Data">
5              <connection-url>https://atiqkqh1s4.execute-api.us-east-2.amazonaws.com/prod/ReimbursementRat
6              <token>012042002059049000111114109080091103066094123086094127125080125097068120123102083</to
7          </rest>
8      </datasources>
9  </decisionService>
10
```

- **URL Parameter Token**- Set the property that specifies the URL parameter that will pass the security token. Requires a Field Name, typically defaulted to Authorization in the Datasource and its Token (in this example `Bearer 12345678901234567890`), as shown:

Figure 51: REST Connectivity sample using URL ParameterToken authentication



The credentials are encrypted when they are exported for deployment, as illustrated:

Figure 52: Datasource Configuration file of REST Connectivity sample using URL Parameter Token authentication

```
1  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2  <decisionService>
3      <datasources>
4          <rest authentication-type="URLPARAMETER" field-name="Authorization" name="Rate Data">
5              <connection-url>https://atiqkqh1s4.execute-api.us-east-2.amazonaws.com/prod/ReimbursementRat
6              <token>012042002059049000111114109080091103066094123086094127125080125097068120123102083</to
7          </rest>
8      </datasources>
9  </decisionService>
10
```

Parameters on REST Service connections

Parameters on REST Services can be either URL parameters or path parameters. :

1. Service using query parameter for specifying the procedure code:
   - The base URL is https://bj36i9ki66.execute-api.us-east-2.amazonaws.com/prod/ReimbursementRate
   - For example, to access a specific procedure code: https://bj36i9ki66.execute-api.us-east-2.amazonaws.com/prod/ReimbursementRate?procedureCode=B5120ZZ

   The API Gateway entry point is: `ReimbursementRate-API`
2. Service using path parameter for specifying the procedure code:

- The base URL is https://qknpo561c0.execute-api.us-east-2.amazonaws.com/prod/ReimbursementRate/
- For example, to access a specific procedure code: https://qknpo561c0.execute-api.us-east-2.amazonaws.com/prod/ReimbursementRate/0313090

Note: Parameters that have a default value are in the Ruleflow. Parameters that do not have values are taken to be static parameter options that are not intended to take values. Therefore you cannot override the values, since it is not expected to take a value.

Description

The REST Datasource Property Table provides a column for parameter type:

URL www.beachreport.com

Name	Value	Type
state	NH	Path
beach	Hampton	Path
date	20190112	Query

The sample listed has 2 path parameters that would result in the following connection URL: www.beachreport.com/NH/Hampton The date parameter would be added as a standard URL parameter (?date=20190112).

Import REST Datasource metadata into a Vocabulary

Where relational databases have formal schemas, keys, and datatypes, REST datasources have many variations. In a REST data source, JSON-formatted data might have a JSON map that describes the structure of the data, but often mapping the columns in a REST datasources requires manual intervention to define primary keys and relationships of nested objects and arrays. Those variations are discussed in the topic Mapping REST Service metadata on page 173.

Note: Using the sample: To load the REST Connectivity sample, choose the menu item **Help > Samples**. Select **REST Connectivity**, and then click **Done**. Follow the Import dialog to bring the sample into your workspace. The REST Datasource is predefined in the sample to specify the data types, table and column names, and the join.

Let's get one set of rates from our sample REST Datasource by entering its URL with one parameter in a browser:

```
https://bj36i9ki66.execute-api.us-east-2.amazonaws.com/prod/ReimbursementRate?procedureCode=B5120ZZ
```

That returns:

```
{
                                    "results": [
                                    {
                                    "procedureCode": "B5120ZZ",
                                    "rates": [
                                    {
                                    "startDate": "2017-1-1",
                                    "endDate": "2017-6-1",
                                    "rate": 0.85
                                    },
                                    {
                                    "startDate": "2017-6-2",
                                    "endDate": "2017-12-31",
                                    "rate": 0.83
                                    }
                                    ]
                                    }
                                    ]
                                    }
```

The URL connected to the REST server that enables query filtering such that the connection returned only the results that matched the parameter value.

The sample has some complexity. You can see that the REST source data has two rates for a `procedureCode`. You have to consider then that there could be many rates for one code, differentiated by applicable dates. The transformation to a relational way of thinking looks like this:

Corticon Entity association	*Treatment:medicalCode*	*Equivalent to relational database as...*
Objects in REST Datasource ***Rate Data***		
Database implied	***Column***	
Rate	*(synthetic increment)**	position (PK, varchar(1), not null)
Rate	*(PROCEDURECODE)**	procedureCode (PK, varchar(1), not null
Rate	startDate	startDate (date, null)
Rate	endDate	endDate (date, null)
Rate	rate	rate (float, null)

This pattern shows that the REST data was interpreted as two tables related through the `procedureCode`. To ensure uniqueness for the primary key an incrementing integer value is added to key. Now it will be easier to define the REST Datasource in a Corticon Vocabulary. For more information, see Advanced REST Datasource Topics on page 167.

When a REST Service exposes a schema, its metadata can be imported into Corticon Studio to refine and complete the mappings between the Vocabulary and the metadata. The REST Service connection will make best-efforts to discover the REST schema. You can edit the schema definitions and tune the mapping of the REST data structure to the Vocabulary.

In the Vocabulary editor with a REST Service connection established, select the Vocabulary root, and then select the tab of the Datasource connection metadata you want to import.

As REST Datasource metadata is imported into a Vocabulary, the Vocabulary Editor's automatic mapping feature attempts to find the *smart match* for each piece of metadata. An entity will be auto-mapped to a table if the two names are spelled the same way, regardless of case.

As REST data sources are not as strictly defined as relational databases, the mapping of REST Datasources will likely require manual intervention to establish the primary keys and associations in the REST metadata. For more information, see the advanced topics Mapping REST Service metadata on page 173 and Defining associations in REST Service metadata on page 180

Mapping REST Service metadata

Introduction

A REST Datasource can present very straightforward data, such as a one-dimensional table and an obvious primary key. Then again, it might have implicit structure of data and types that need to be clearly defined to support SQL queries.

The mapping of the sample JSON document produced one parent table and one child table. In the parent table, the object `procedureCode` is viewed as a relational column. Nested objects are also flattened into relational columns; however, column names are formed by concatenating the name of the parent and nested objects, which are joined by an underscore character. For example, the `PROCEDURECODE_RATE` column contains the values of the rate objects that are nested in the `rates` object. The primary key is determined by the first field detected in the document, `procedureCode`.

The following snippet is the results String with a request URL that is filtering for just the `procedureCode B5120ZZ`:

```
1  {"results":[{"procedureCode":"B5120ZZ","rates":[{"startDate"
     :"2018-1-1","endDate":"2018-6-1","rate":0.85},{"startDate"
     :"2018-6-2","endDate":"2018-12-31","rate":0.8}]}]}
```

When JSON formatting is applied, it is easier to see the data structure. Note that the `results` table corresponds to the top-level entities in the JSON and the `rates` table to the individual rates for procedures. The `procedureCode` is evaluated by the Autonomous REST Connector and discovered as unique, so it is set as the key. The JSON data is mapped into this schema:

```
{
  "results": [
    {
      "procedureCode": "B5120ZZ",
      "rates": [
        {
          "startDate": "2018-1-1",
          "endDate": "2018-6-1",
          "rate": 0.85
        },
        {
          "startDate": "2018-6-2",
          "endDate": "2018-12-31",
          "rate": 0.8
        }
      ]
    }
  ]
}
```

But there are two effective date ranges and rates. There is an implicit association of `0` to `n` rates for each `procedureCode`. The Autonomous REST Connector creates synthetic key fields that are added to the `rates` table. When JSON is viewed as nodes, the two rates are distinguished by the integer incrementor that becomes the synthetic key field, as illustrated:

You are ensured a unique primary key (PK) by melding the incremental value with `procedureCode` for a `rate` – it links a row in the `rate` table to a row in the `results` table.

Note: The URL that is the target for the import of REST metadata must return JSON representing unique keys that will be mapped to your vocabulary. If it does not, the `position` field will be added to the generated schema to uniquely identify instances.

When JSON is viewed as nodes, the ability to distinguish the two rates with an integer incrementor, as illustrated:

Corticon Entity association	*Treatment:medicalCode*	*Equivalent to relational database as...*
Objects in REST Datasource ***Rate Data***		
Database implied	***Column***	
Rate	*(synthetic increment)**	position (PK, varchar(1), not null)
Rate	*(PROCEDURECODE)**	procedureCode (PK, varchar(1), not null
Rate	startDate	startDate (date, null)
Rate	endDate	endDate (date, null)
Rate	rate	rate (float, null)

Export a discovered schema

The Autonomous REST Connector schema discovery mechanism generates a schema for a REST service. You can make changes or add your preferred schema. Once you have fully defined a REST connection, it will be saved in the Vocabulary so it does not need to be recreated on each use. If you want to manually edit a REST schema to, for example, tweak a data type, click SCHEMA **Export** to export it to a text file. In that circumstance, you must specify the schema file when configuring a REST Datasource. Here is the exported schema from the REST sample:

```
{
    "REST_DATA":{
        "#path":[
            "
            https://bj36i9ki66.execute-api.us-east-2.amazonaws.com/
            prod/ReimbursementRate /results"
        ],
        "procedureCode":{
            "#type":"VarChar(64),#key",
            "#default":"B5120ZZ",
            "#eq":"procedureCode"
        },
        "rates[1]":{
            "startDate":"Date",
            "endDate":"Date",
            "rate":"Double"
        }
    }
}
```

Here is the subtly different exported schema from REST in the Mixed Connectivity sample:

```
{
    "REST_DATA":{
        "#path":[
            "
            https://bj36i9ki66.execute-api.us-east-2.amazonaws.com/
            prod/ReimbursementRate /results"
        ],
        "procedureCode":"VarChar(64),#key",
        "rates[12]":{
            "startDate":"Date",
            "endDate":"Date",
            "rate":"Double"
        },
        "procedurecode":{
            "#type":"VarChar(64),#key",
            "#virtual":true,
            "#default":"B5120ZZ",
            "#eq":"procedurecode"
        }
    }
}
```

Corticon does not provide any instructions on manipulation of schema files. See topics in DataDirect's Autonomous REST Connector online help at https://documentation.progress.com/output/DataDirect/jdbcautoresthelp.

Once you have saved your updated schema file, click SCHEMA **Import** to apply it.

Define your preferred schema

The REST SCO currently depends on the Autonomous REST Connector schema discovery mechanism to generate the schema for a REST service. The Autonomous REST Connector design accounts for this by allowing users to supply a REST schema file. This is an alternative to the schema discovery approach. Corticon needs to allow users to specify the schema file when configuring a REST Datasource. When specified, the schema file would be supplied to Autonomous REST Connector and schema discovery not performed.

The schema files are text files but they can be complex. Corticon will not provide any mechanisms to simplify the creation of schema files - this is the purview of the Autonomous REST Connector, whose roadmap includes provisions for tooling to aid in the creation of schema files. Users are referred to Autonomous REST Connector documentation and tooling for creation of schema files - Corticon will just provide for the import of the files.

Requirements

When configuring a REST Datasource, you have the option of supplying the schema file.

The schema is imported and stored as part of the vocabulary, similar to how it is done when schema discovery is performed. Because of its close relationship with the vocabulary, the schema is not stored as a separate file.

Steps in mapping REST Service metadata

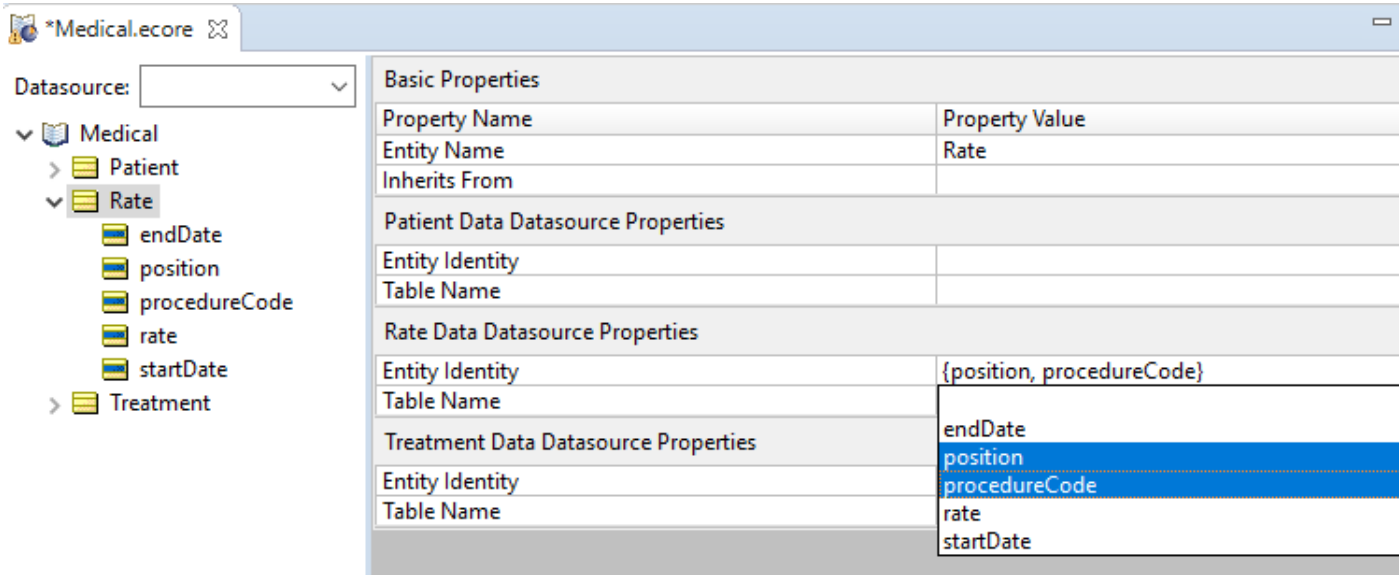
Note: You can set the REST Datasource in the Mixed Connectivity sample to enable you to experience the hand crafting of the REST mapping process. With the Mixed Connectivity sample in your workspace, choose the top-level Vocabulary tab for the **Rate** data, and then click MAPPING **Clear** and METADATA **Clear**.

To map a REST Datasource into a Corticon Vocabulary:

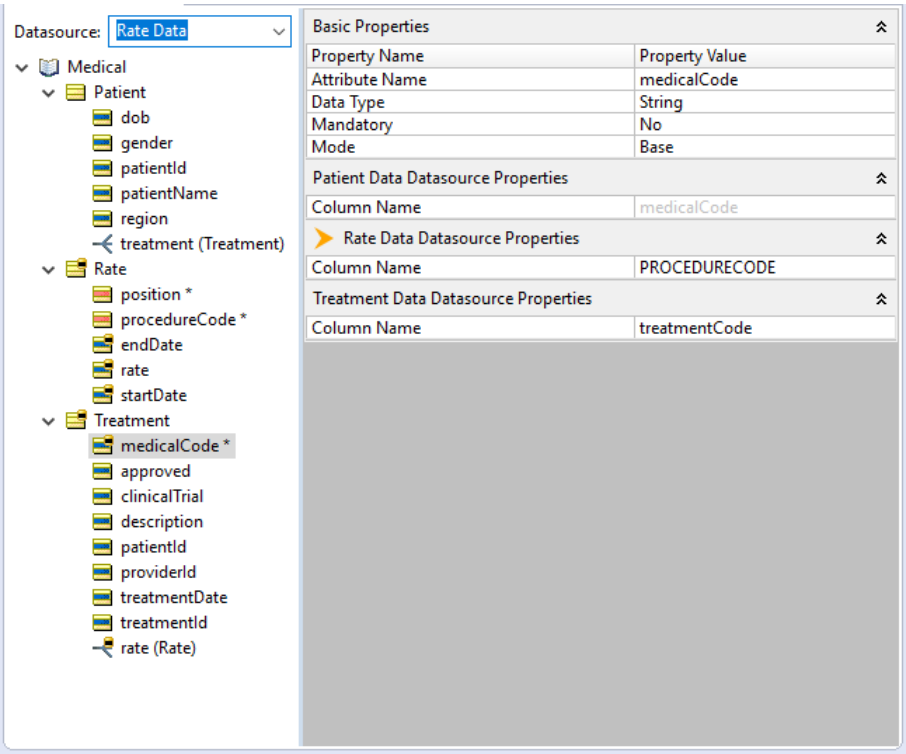
1. Create a Corticon Vocabulary entity and attributes that will provide a formal structure to the data. The Autonomous REST Connector processor sets up a column name for each data label and infers its data types from the value.
2. Set the Rate table name, RATE
3. Specify the column names and data types of the attributes, as follows:

a	b	c
`position`	`POSITION`	`Integer`
`endDate`	`ENDDATE`	`Date`
`rate`	`RATE`	`Decimal`
`procedureCode`	`PROCEDURECODE`	`String`
`startDate`	`STARTDATE`	`Date`

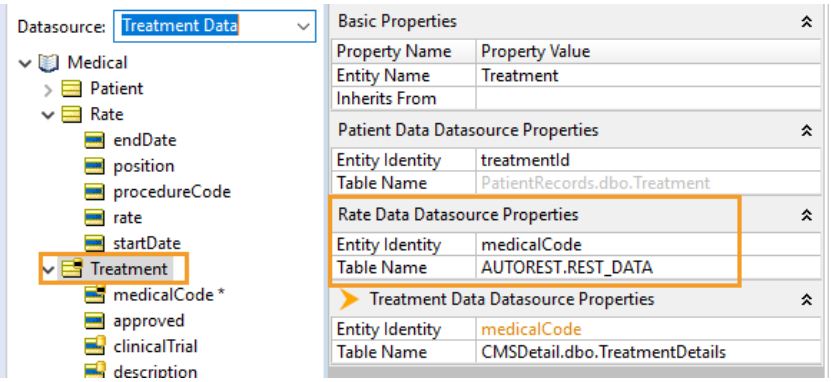
4. Click on the name of the REST entity to display its properties. The processor might have also created a synthetic field, a sequential integer that ensures uniqueness. On the **Rate Data** Datasource, pulldown to choose `AUTOREST.REST_DATA`.
5. Then choose the Entity Identity pulldown to select the compound key as shown:



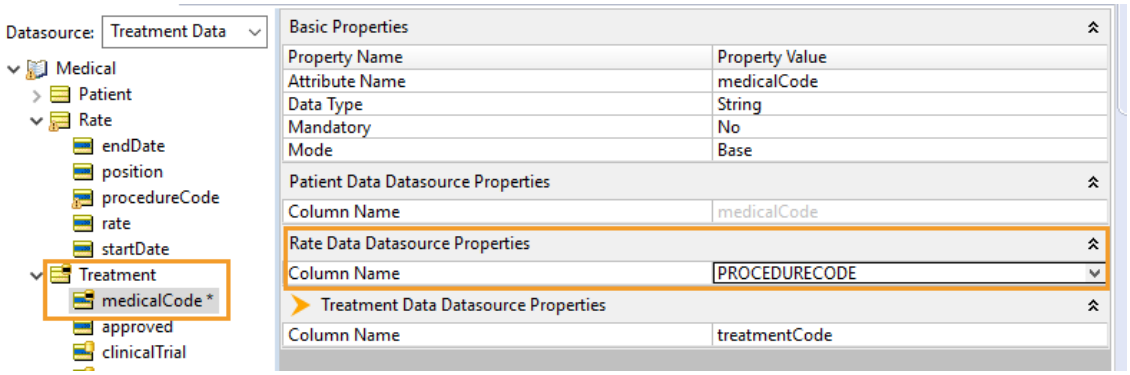
6. Define the link element on **Rate Data** Datasource. Pulldown to choose `AUTOREST.REST_DATA`.

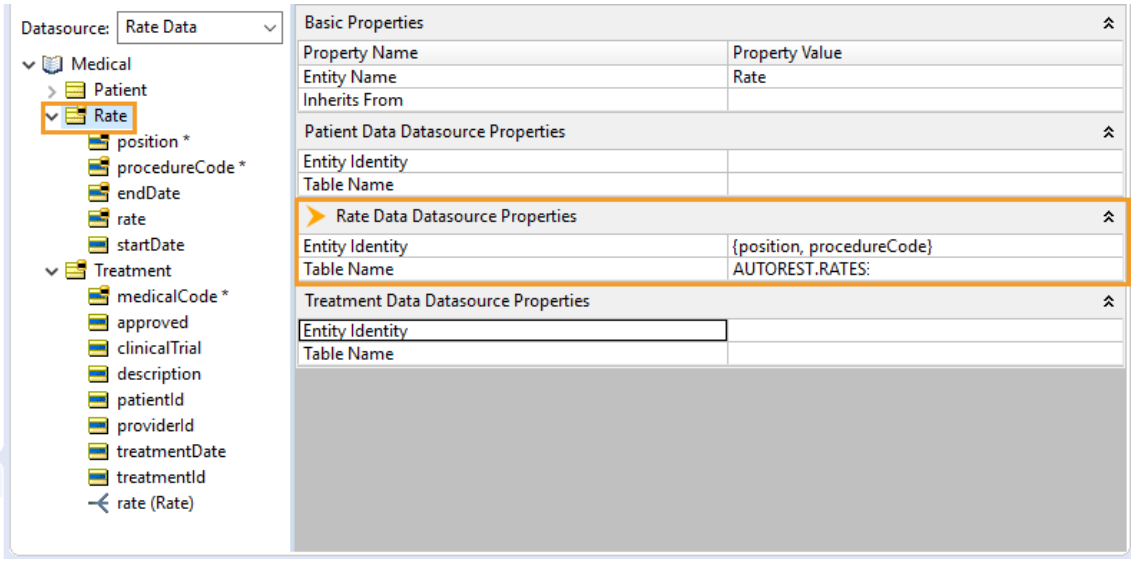


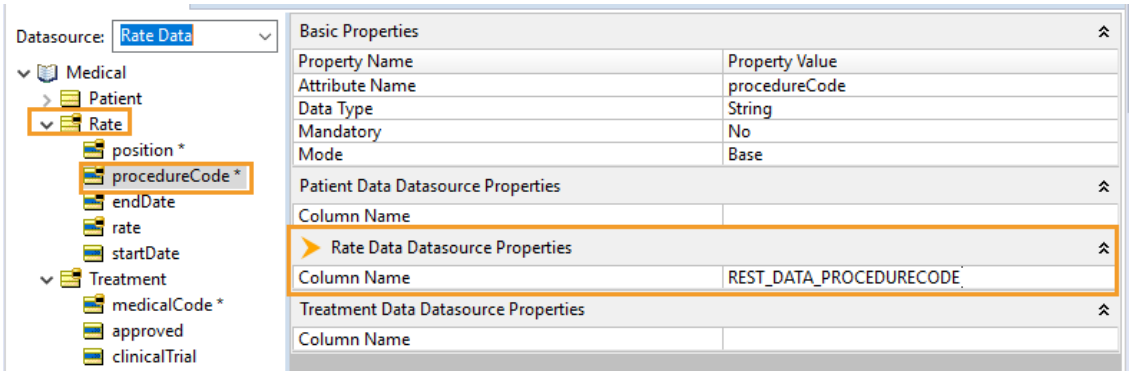
Then choose Entity Identity pulldown to select `medicalCode` as shown:



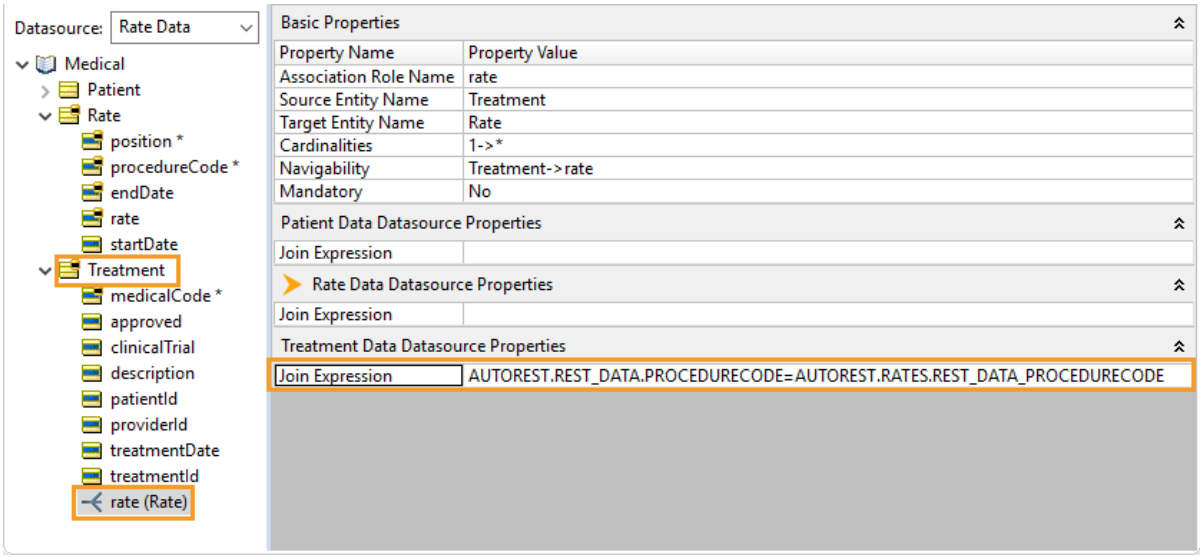
7. The Treatment `medicalCode` maps to the `PROCEDURECODE` of the REST data:





8. 

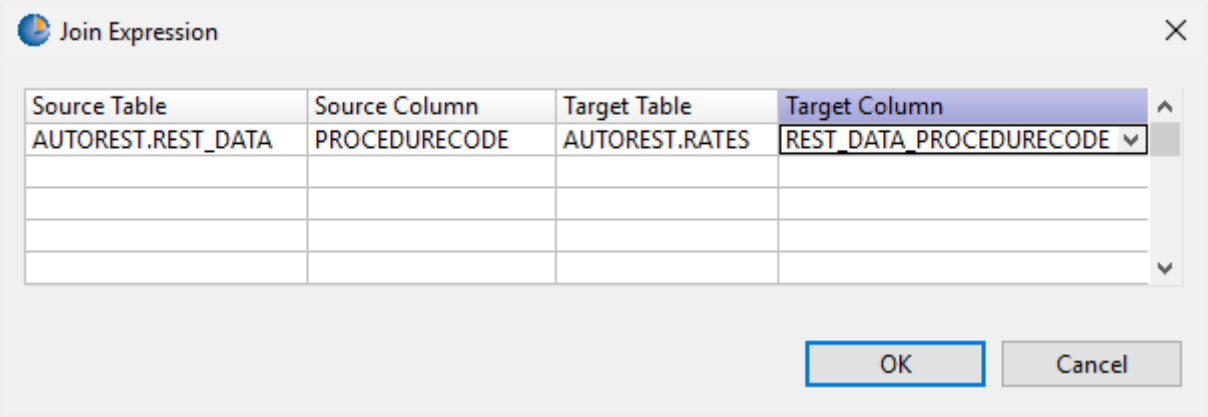
9. The elements of the join are all defined. Open the Treatment's rate association by clicking on the label **Join Expression**. In the dialog box that opens



10. The association from the `rate` entity to the `treatment` entity is defined as:

Property Name	Property Value
Association Role Name	rate
Source Entity Name	Treatment
Target Entity Name	Rate
Cardinalities	1->*
Navigability	Treatment->rate
Mandatory	No

For this sample, that join expression is as shown:



Defining associations in REST Service metadata

The JSON data returned from a REST datasource can be, and often is, hierarchical. There are some special concerns when mapping this hierarchical data in Studio.

In order to properly map associations (which are essentially joins between 2 tables), the Entity Identity must be set for the Entity to specify which attributes are part of the primary key. Additionally, the join expression for the association must be set to define which attributes are used to create the join.

The REST Services driver automatically creates the primary key for the tables. In most cases the primary key is the first field in the JSON for that object, unless there is some reason that it cannot be used as a primary key, or a different field was determined to be a better fit. When the data contains arrays of objects, the Autonomous REST Connector creates additional fields named "POSITION", and then adds them to the primary key. This field indicates where the object was found in the array.

If parameters are also mapped, then the parameter column is also added to the primary key, so these columns must also have corresponding attributes in the Entity.

In summary, the primary key or Entity Identity for an Entity will consist of:

- At least one attribute that was determined to be unique across all of the elements of the array
- For arrays, a position field to specify the location in the array, if a unique field could not be identified.
- Any URL parameter columns

For associated Entities there will be additional elements in the primary key or Entity Identity :

- The ID of the root element that this element has as a parent
- The position field of the parent object (if it was determined to be in an array
- The position field of the parent's parent recursively back up to the root table (REST_DATA).

For example, if we have the following structure:

Vehicle

Id

→Devices

Id

→ Radios

Id

The structure that the Autonomous REST Connector would create would be this:

Vehicle

*Id

→Devices

*position

*VehicleID

Id

→ Radios

*DevicesPosition

*position

*VehiclePosition

Id

As you can see, the more nested the structure, the more complex the primary keys get for the lower levels, as the keys from the previous levels all have to be maintained at each level.

The potentially confusing point is that the position columns are added by the Autonomous REST Connector and will have to have corresponding attributes in the vocabulary. These attributes do not exist in the JSON document and will need to be manually added during modeling (they may be set to transient if desired).

When there is a name conflict, or a conflict with a reserved word, Autonomous REST Connector will post-pend a "_" to the name, or a "_<number>". This may make mapping difficult, as it may be hard to tell which is which.

A

Supported RDBMS brands and features for Corticon EDC

Corticon EDC can connect a Vocabulary to an instance of a supported database. All data sources support **Import/Clear Database Metadata** and **Validate Mappings**. Some Corticon EDC features are not supported in certain supported databases. Data manipulations and database startup functions that might be required to ensure error-free interaction between Corticon EDC and a database are noted.

Note: The mapping of database columns to a Corticon Vocabulary through SQL might experience problems when database columns have hyphens, spaces or other special characters (even though some databases and SQL parsers allow them). The generally accepted valid values are all alphanumeric characters and the underscore character. It is a plus to use all-lowercase names to avoid platform case inconsistencies. For more information on Corticon's accepted names, see the topic *"Vocabulary node naming restrictions" in the Quick Reference Guide*.

The following table summarizes the features:

Database	Import Vocabulary	Create Schema	DB Rows: Read/Update	DB Rows: Read-only	Import Metadata	Import Enumerations	Extend to database
IBM DB2	-	Yes	Yes	Yes	Yes	Yes	Yes
Microsoft SQL Server	-	Yes	Yes	Yes	Yes	Yes	Yes
MySQL	-	Yes	Yes	Yes	Yes	Yes	Yes
Oracle Database	-	Yes	Yes	Yes	Yes	Yes	Yes

Database	Import Vocabulary	Create Schema	DB Rows: Read/Update	DB Rows: Read-only	Import Metadata	Import Enumerations	Extend to database
Postgres	-	Yes	Yes	Yes	Yes	Yes	Yes
Progress OpenEdge	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Progress OpenAccess	Yes	-	Yes	Yes	Yes	Yes	Yes

Note: The feature of importing database metadata will infer associations when the information (foreign keys) is available in the data source's metadata.

For the current list of supported data sources and versions, access the web location Progress Corticon 6.0 - Supported Platforms Matrix.

For details, see the following topics:

- IBM DB2
- Microsoft SQL Server
- MySQL
- Postgres
- Oracle Database
- Progress OpenEdge
- Progress OpenAccess

IBM DB2

Vocabulary: Database Connection definition

- **Database URL** - Default port: 50000
- **Username and Password** - DB2 credentials.
- **Catalog filter** - Yes.
- **Schema filter** - Yes.
- **Additional properties** - None required.

Vocabulary: Database Access actions

- **Create/Update Database Schema** - Yes.
- **Import Enumeration Elements** - Yes.
- **Export Database Access Properties** - Yes.

Rulesheet: Set an alias to Extend to Database - Yes.

Ruletest: Database Access options

- **Read Only** - Yes.
- **Read/Update** - Yes.
- **Return All Entity Instances** - Yes.
- **Return Incoming/New Entity Instances Only** - Yes.

Other requirements and considerations:

- None.

Microsoft SQL Server

Vocabulary: Database Connection definition

- **Database URL** - Default port: 1433
- **Username and Password** - SQL Server credentials.
- **Catalog filter** - Yes.
- **Schema filter** - Yes.
- **Additional properties** - None required.

Vocabulary: Database Access actions

- **Create/Update Database Schema** - Yes.
- **Import Enumeration Elements** - Yes.
- **Export Database Access Properties** - Yes.

Rulesheet: Set an alias to Extend to Database - Yes.

Ruletest: Database Access options

- **Read Only** - Yes.
- **Read/Update** - Yes.
- **Return All Entity Instances** - Yes.
- **Return Incoming/New Entity Instances Only** - Yes.

Other requirements and considerations:

- None.

MySQL

Vocabulary: Database Connection definition

- **Database URL** - Default port: 3306
- **Username and Password** - Refer to MySQL documentation or your database administrator.
- **Catalog filter** - No.
- **Schema filter** - No.
- **Additional properties** - None required.

Vocabulary: Database Access actions

- **Create/Update Database Schema** - Yes.
- **Import Enumeration Elements** - Yes.
- **Export Database Access Properties** - Yes.

Rulesheet: Set an alias to Extend to Database - Yes.

Ruletest: Database Access options

- **Read Only** - Yes.
- **Read/Update** - Yes.
- **Return All Entity Instances** - Yes.
- **Return Incoming/New Entity Instances Only** - Yes.

Other requirements and considerations:

- While MySQL presents databases as catalogs, the Database Connection definition cannot apply a filter to those catalogs.

  .

Postgres

Vocabulary: Database Connection definition

- **Database URL** - Default port: 5432
- **Username and Password** - Postgres credentials.
- **Catalog filter** - Yes.
- **Schema filter** - Yes.
- **Additional properties** - None required.

Vocabulary: Database Access actions

- **Create/Update Database Schema** - Yes.
- **Import Enumeration Elements** - Yes.
- **Export Database Access Properties** - Yes.

Rulesheet: Set an alias to Extend to Database - Yes.

Ruletest: Database Access options

- **Read Only** - Yes.
- **Read/Update** - Yes.
- **Return All Entity Instances** - Yes.
- **Return Incoming/New Entity Instances Only** - Yes.

Other requirements and considerations:

- None.

Oracle Database

Vocabulary: Database Connection definition

- **Database URL** - Default port: 1521
- **Username and Password** - Oracle DB credentials.
- **Catalog filter** - No.
- **Schema filter** - Yes.
- **Additional properties** - None required.

Vocabulary: Database Access actions

- **Create/Update Database Schema** - Yes.
- **Import Enumeration Elements** - Yes.
- **Export Database Access Properties** - Yes.

Rulesheet: Set an alias to Extend to Database - Yes.

Ruletest: Database Access options

- **Read Only** - Yes.
- **Read/Update** - Yes.
- **Return All Entity Instances** - Yes.
- **Return Incoming/New Entity Instances Only** - Yes.

Other requirements and considerations:

- None.

Progress OpenEdge

Vocabulary: Database Connection definition

- **Database URL** - Default port: 5566
- **Username and Password** - OpenEdge credentials.
- **Catalog filter** - Yes.
- **Schema filter** - Yes.
- **Additional properties** - None required.

Vocabulary: Database Access actions

- **Create/Update Database Schema** - Yes. Only for Vocabulary elements that were not created through a BRVD import.
- **Import Enumeration Elements** - Yes.
- **Export Database Access Properties** - Yes.

Rulesheet: Set an alias to Extend to Database - Yes.

Ruletest: Database Access options

- **Read Only** - Yes.
- **Read/Update** - Yes.
- **Return All Entity Instances** - Yes.
- **Return Incoming/New Entity Instances Only** - Yes.

Other requirements and considerations:

- **Importing a file to create a Vocabulary** - Import requires a `.brvd` file created in OpenEdge (see Progress OpenEdge documentation for details.) The import function into Corticon is described in *"Importing an OpenEdge Business Rules Vocabulary Definition (BRVD) file" in the Quick Reference Guide.*
- **Startup of OE server** - It is recommended that you start the OpenEdge database server with the following parameters within the `Proenv` window, shown here with values used in a test environment:

  `proserve` *`db_name`* `-n 65 -Mn 20 -Mpb 4 -Ma 20 -Mi 3 -S` *`port_number`*

  where:

  - *`db_name`* is the database name
  - *`port_number`* is the port number
  - Other OpenEdge parameters as described in OpenEdge Database Server parameters.

Progress OpenAccess

Vocabulary: Database Connection definition

- **Database URL** - Default port: 19991. Must include `ServerDataSource=OA_OpenEdgeAppServer`
- **Username and Password** - OpenAccess credentials.
- **Catalog filter** - Yes.
- **Schema filter** - Yes.
- **Additional properties** - None required.

Vocabulary: Database Access actions

- **Create/Update Database Schema** - No.
- **Import Enumeration Elements** - Yes.
- **Export Database Access Properties** - Yes.

Rulesheet: Set an alias to Extend to Database - Yes.

Ruletest: Database Access options

- **Read Only** - Yes.
- **Read/Update** - Yes.
- **Return All Entity Instances** - Yes.
- **Return Incoming/New Entity Instances Only** - Yes.

Other requirements and considerations:

- **Importing a file to create a Vocabulary** - Import requires a `.brvd` file created in OpenEdge (see Progress OpenEdge documentation for details.) The import function into Corticon is described in *"Importing an OpenEdge Business Rules Vocabulary Definition (BRVD) file" in the Quick Reference Guide.*
- See the OpenEdge documentation for Progress OpenAccess for constraints and additional information.