BrightstarDB Documentation

Release 1.8.0

Kal Ahmed, Graham Moore

November 30, 2014

Contents

CHAPTER 1

Getting Started

Welcome to BrightstarDB, the NoSQL semantic web database for .NET. The documentation contains lots of examples and detailed information on all aspects of BrightstarDB. The following sections provide some gentle hints of where to look depending on what you are planning to do with BrightstarDB.

It's probably a good idea, no matter what you plan to use BrightstarDB for, to read the *Concepts* section and the *'Why BrightstarDB?'* section to understand the architecture and ideas behind the technology.

If you just want to see the simplest example of creating a BrightstarDB Entity Data Model then jump straight to the *Developer Quick Start*.

We hope you enjoy developing with BrightstarDB. Please consider joining our community of developers and users and share any questions or comments you may have.

1.1 Architect

If you are an architect considering using BrightstarDB then the *Concepts* section is important. Following that skimming over the different APIs will give you an overview of the different tools that developers can use to work with BrightstarDB. The other sections that provide a good overview of BrightstarDB's capabilities and features are the *API Documentation*, *Admin API* and *Polaris Management Tool* sections.

1.2 Data

If you are coming to BrightstarDB from an RDF perspective and want to work with RDF Data and SPARQL then the best place to start is the *Polaris Management Tool*. This shows how to create a new store without code, load in RDF data, and execute queries and update transactions. Other sections of interest will probably be *SPARQL Endpoint* and if you are writing code the *RDF Client API*.

1.3 Developer

BrightstarDB provides several layers of API that are aimed at specific development activities or scenarios. There are three main API levels, Entity Framework, Data Objects and RDF.

BrightstarDB Entity Framework & LINQ

The BrightstarDB Entity Framework is a powerful and simple to use technology to quickly build a typed .NET domain model that can persist object state into a BrightstarDB instance.To use this you create a set of .NET interfaces that define the data model. The BrightstarDB tooling takes these definitions and creates concrete implementing classes.

These classes can then be used in an application. The flexibility of the underlying storage makes evolving the model very easy and straight forward. BrightstarDB is optimized for associative data which provides a high performance when working with objects. As this is a fully typed domain model it also provides LINQ and OData support.

The main sections to see for developing .NET typed domain models are the *Developer Quick Start* section, the section on the BrightstarDB *Entity Framework*, and the *Entity Framework Samples*.

Data Objects & Dynamic

When working with data that may change shape at runtime, or when a fixed typed domain model is not required, the Data Object and Dynamic APIs provide a generic object layer on top of the RDF data. This layer provides abstractions that allow the developer to treat collections of triples as the state of a generic object. The sections *Data Object Layer* and *Dynamic API* provide documentation and examples of this APIs.

RDF & SPARQL

To work programmatically with RDF, SPARQL, and SPARQL see update the *RDF Client API* and *SPARQL Endpoint* sections.

Portable Class Library and Windows Store

If you are building a Windows Store application you can now make use of the Portable Class Library build of BrightstarDB. This build also supports targetting Silverlight 5 and Windows Phone 8. For more information please refer to *Developing Portable Apps*.

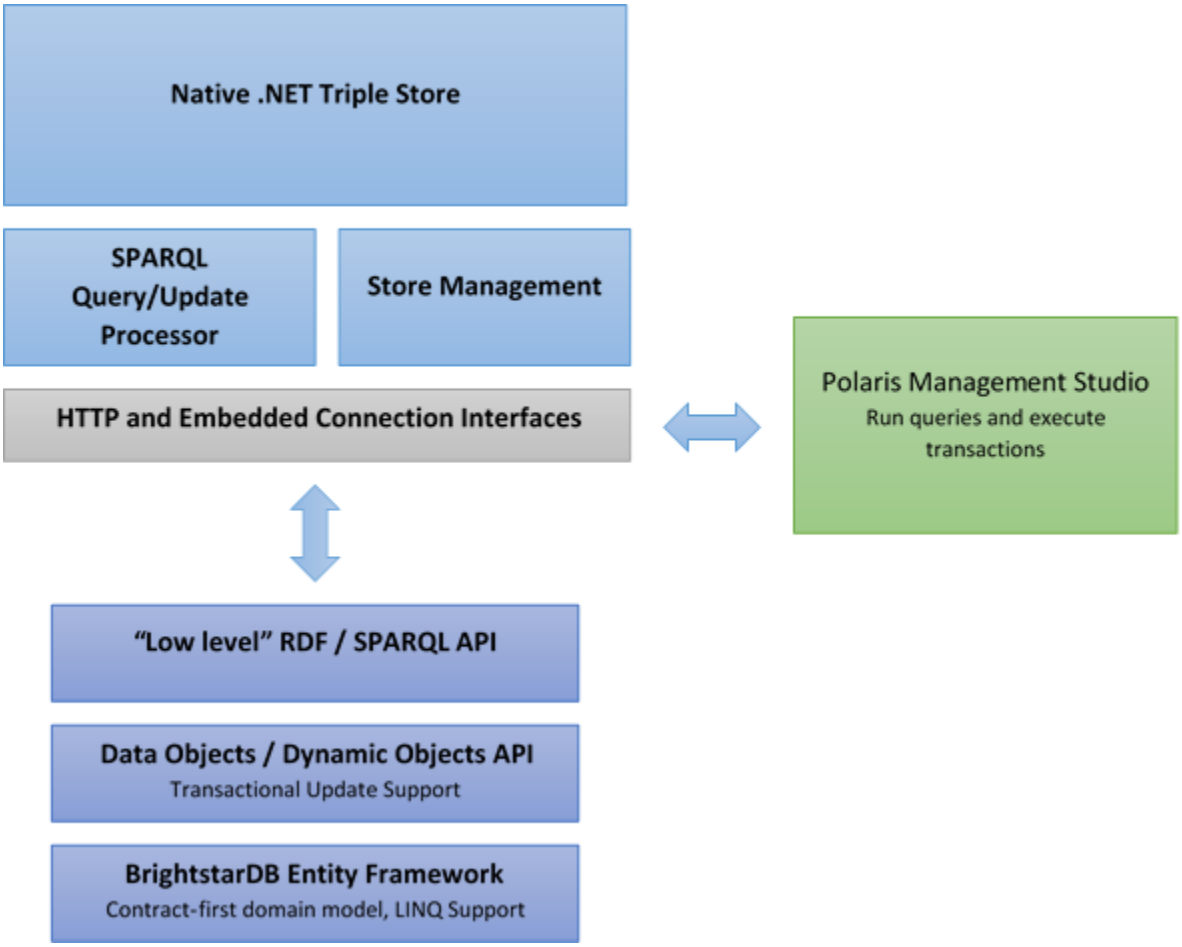
CHAPTER 2

Concepts

2.1 Architecture

BrightstarDB is a native .NET NoSQL semantic web database. It can be used as an embedded database or run as a service. When run as a service clients can connect using HTTP, TCP/IP or Named Pipes. While the core data model is RDF triples and the query language SPARQL BrightstarDB provides a code-first Entity Framework. The Entity Framework tools take .NET interfaces and generate concrete classes that persist their data in BrightstarDB. As well as the Entity Framework there is a low level *RDF API* for working with the underlying data. BrightstarDB (in the Enterprise and Server versions) also provides a management studio called *Polaris* for running queries and transactions against a BrightstarDB service.

The following diagram provides an overview of the BrightstarDB architecture.



2.2 Data Model

BrightstarDB supports the W3C RDF and SPARQL 1.1 Query and Update. standards, the data model stored is triples with a graph context (often this is called a quad store). The triple data structure is very powerful, especially for creating associative data models, merging data from many sources, and for giving unique persistent and global identity to 'things'.

A **triple** is defined as having three parts: A subject URI, a predicate URI, and an object value. The subject URI is the identifier for some thing. A person, company, product etc. The predicate is an identifier for a property type and the object can either be the identifier for another thing, or a literal value. Literal values can also have data types.

An example of a literal property assigned to some thing is:

```
<http://www.brightstardb.com/companies/brightstardb> <http://www.w3.org/2000/01/rdf-schema#label> "Br
```

and a connection between two entities is described:

```
<http://www.brightstardb.com/companies/brightstardb> <http://www.brightstardb.com/types/hasproduct> <
```

2.3 Storage Features

BrightstarDB is a write once, read many store (WORM). Modifications to data are appended to the end of the storage file, no data is ever overwritten. It employs a single writer, concurrent reader model. This supports concurrent read with no possibility of reading dirty data. Reads are not blocked while writes occur. The WORM store approach supports rollback or querying of the complete database at any transaction point. The store can be periodically coalesced to manage file size growth at the expense of removing previous transaction points.

2.4 Client APIs

There are three different code layers with which to access BrightstarDB. The first of these is the *RDF Client API*. This is a low level API that allows developers to insert and delete triples, and run SPARQL queries. The second API layer is the *Data Object Layer*. This provides the ability to treat a collection of triples with the same subject as a single unit and also provides support for RDF list structures and optimistic locking. The highest API layer is the *BrightstarDB Entity Framework*. BrightstarDB enables data-binding from items at the Data Object Layer to full .NET objects described by a programmer-defined interface. As well as storing object state BrightstarDB also allows developers to use LINQ expressions to query the data they have created.

2.5 Supported RDF Syntaxes

As BrightstarDB is built on the W3C RDF data model, we also provide the ability to import and export your data as RDF.

BrightstarDB supports a number of different RDF syntaxes for file-based import. This list of supported file formats applies both to import jobs created using the BrightstarDB API (see *RDF Client API* for details), and to file import using Polaris (see *Polaris Management Tool* for details). To determine the parser to be used, BrightstarDB checks the file extension, so it is important to use the correct file extension for the syntax you are importing. The supported syntaxes and their file extensions are listed in the table below as shown, BrightstarDB also supports reading from files that are compressed with the GZip compression method.

RDF Syntax	File Extension (uncompressed)	File Extension (GZip compressed)
NTriples	.nt	.nt.gz
NQuads	.nq	.nq.gz
RDF/XML	.rdf	.rdf.gz
Turtle	.ttl	.ttl.gz
RDF/JSON	.rj or .json	.rj.gz or .json.gz

CHAPTER 3

Why BrightstarDB?

BrightstarDB is a unique and powerful data storage technology for the .NET platform. It combines flexibility, scalability and performance while allowing applications to be created using tools developers are familiar with.

3.1 An Associative Model

All databases adopt some fundamental world view about how data is stored. Relational databases use tables, and document stores use documents. BrightstarDB has adopted a very flexible, associative data model based on the W3C RDF data model.

BrightstarDB uses the powerful and simple RDF graph data model to represent all the different kinds of models that are to be stored. The model is based on a concept of a triple. Each triple is the assignment of a property to an identified resource. This simple structure can be used to describe and represent data of any shape. This flexibility means that evolving systems, or creating systems that merge data together is very simple.

Few existing NoSQL databases offer a data model that understands, and automatically manages relationships between data entities. Most NoSQL databases require the application developer to take care of updating 'join' documents, or adding redundant data into 'document' representations, or storing extra data in a key value store. This makes many NoSQL databases not particularly good at dealing with many real word data models, such as social networks, or any graph like data structure.

3.2 Schema-less Data Store

The associative model used in BrightstarDB means data can be inserted into a BrightstarDB database without the need to define a traditional database schema. This further enhances flexibility and supports solution evolution which is a critical feature of modern software solutions.

While the schema-less data store enables data of any shape to be imported and linked together, application developers often need to work with a specific shape of data. BrightstarDB is unique in allowing application developers to map multiple .NET typed domain models over any BrightstarDB data store.

3.3 A Semantic Data Model

While many NoSQL databases are schema-less, few are inherently able to automatically merge together information about the same logical entity. BrightstarDB implements the W3C RDF data model. This is a directed graph data model that supports the merging of data from different sources without requiring any application intervention. All entities

are identified by a URI. This means that all properties assigned to that identifier can be seen to constitute a partial representation of that thing.

This unique property makes BrightstarDB ideal for building enterprise information integration solutions where there is a fundamental need to bring together data about a single entity from many different systems.

3.4 Automatic Data caching

Query results, and entity representations are cached to further improve performance for query intensive applications. Normally, data caching is done by applications but BrightstarDB provides this feature as a core capability.

3.5 Full Historical Capabilities

BrightstarDB uses a form of data storage that preserves full historical data at every transaction point. This allows applications to perform queries at any previous point in time, it ensures fully audit-able data and allows data stores to be returned to any previous state or snapshots taken at any point in time. This approach does increase the amount of disk space used, but BrightstarDB provides a feature to consolidate down to just the currently required data.

3.6 Developer Friendly Toolset

Most developers on .NET are accustomed to using objects and LINQ for building their applications. Database technologies that require a fundamental move away from this impose a large burden upon the developer. BrightstarDB provides a complete typed domain model interface to work with the data in the store. It adopts a unique position where the object model is an operational view onto the data. This means that many different object models can overlay the same semantic data model.

3.7 Native .NET Semantic Web Database

If you are working on .NET and want the power and flexibility of a semantic web data store. Then BrightstarDB is a great place to start. With support for the SPARQL query language and also the NTriples data format building semantic web based applications is simple and fun with BrightstarDB.

3.8 RDF is great for powering Object Oriented solutions

Objects are composed of properties, each property is either a literal value or a reference to another object. This creates a graph or related things with properties. ORM systems requires that tables are organised is specific ways to facilitate storing object state. Changes to either the object model or the relational schema often require a reciprocal change. RDF on the other hand can ideally be used to store both literal properties and object relationships and if the object model needs to change then new property value can be added as there is no fixed schema. Similalry, if additional RDF data is added to the store the object model can either ignore or make use of this data. In this way the object model is an operational, read/write, view of the RDF data.

CHAPTER 4

Developing With BrightstarDB

This section takes you through all of the basic principles of working with the BrightstarDB APIs.

BrightstarDB provides three different levels of API:

1. At the highest level the *Entity Framework* allows you to define your application data model in code. You can then use LINQ to query the data and simple operations on your application data entities to create, update and delete objects.
2. The *Data Object Layer* provides a simple abstract API for dealing with RDF resources, you can retrieve a resource and all its properties with a single call. This layer provides no direct query functionality, but it can be combined with the SPARQL query functionality provided by the RDF Client API. This layer also has a separate abstraction for use with *Dynamic Objects*.
3. The *RDF Client API* provides the lowest level interface to BrightstarDB allowing you to add or remove RDF triples and to execute SPARQL queries.

If you are new to BrightstarDB and to RDF, we recommend you start with the Entity Framework and take a walk through our *Developer Quick Start*. If you are already comfortable with RDF and SPARQL you may wish to start with the lower level APIs.

If you are the kind of person that just likes to dive straight into sample code, please take a moment to read about Running the BrightstarDB Samples first.

CHAPTER 5

Developer Quick Start

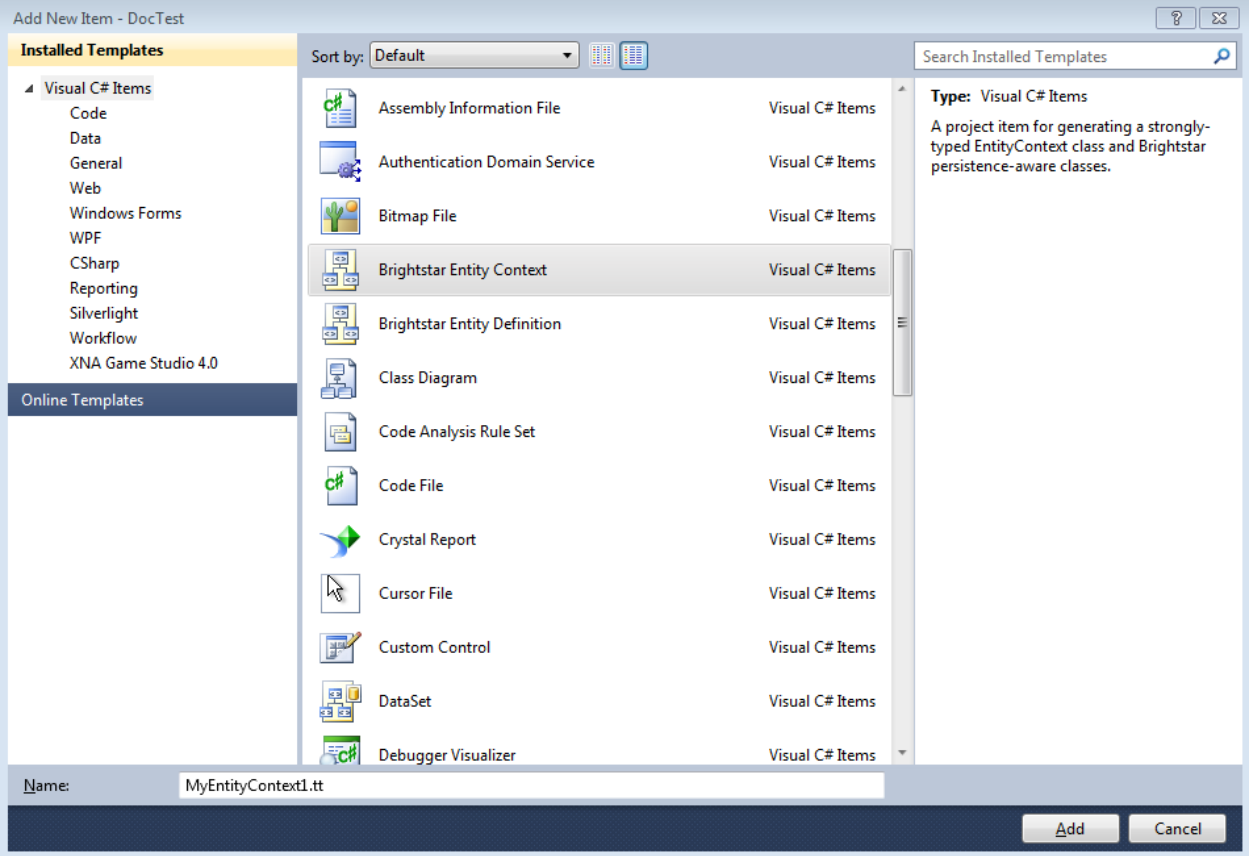
BrightstarDB is about giving developers a really powerful, quick and clean experience in defining and realizing persistent object systems on .NET. To achieve this BrightstarDB can use a set of interface definitions with simple annotations to generate a full LINQ capable object model that stores object state in a BrightstarDB instance. In this quick introduction we will show how to create a new data model in Visual Studio, create a new BrightstarDB store and populate it with data.

Note: The source code for this example can be found in [INSTALLDIR]\Samples\Embedded\EntityFramework\EntityFrameworkSampl

5.1 Create New Project

Create a new project in Visual Studio. For this example we chose a command line application. After creating the project ensure the build target is set to '.NET Framework 4' and that the Platform Target is set to 'Any CPU'

In the solution explorer right click and add a new item. Choose the 'Brightstar Entity Context' from the list.



The project will now show a new component has been added called "MyEntityContext.tt". On the project references right click and add references. Browse to the [INSTALLDIR]\SDK\net40 folder and include all the ".dll" files that are there.

5.2 Create the Model

In this sample we will create a data model that contains actors and films. An actor has a name and a date of birth. An actor can star in many films and each film has many actors. Films also have name property.

The BrightstarDB Entity Framework requires you to define the data model as a set of .NET interface definitions. You can either write these interfaces entirely by hand or you can use the Brightstar Entity Definition item template. Again, right-click on the solution item in the project explorer window and add a new item, this time from the displayed list choose Brightstar Entity Definition and change the name of the file to IActor.cs.

Add the following code to that file:

```
[Entity]
public interface IActor
{
  string Name { get; set; }
  DateTime DateOfBirth { get; set; }
  ICollection<IFilm> Films { get; set; }
}
```

Then add another Brightstar Entity Definition named IFilm.cs and include the following code:

```
[Entity]
public interface IFilm
{
  string Name { get; set; }

  [InverseProperty("Films")]
  ICollection<IActor> Actors { get; }
}
```

5.3 Generating the Context and Classes

A context is a manager for objects in a store. It provides an entry point for running LINQ queries and creating new objects. The context and implementing classes are automatically generated from the interface definitions. To create a context, right click on the MyEntityContext.tt file and select "Run custom tool". This updates the MyEntityContext.cs to contain the context class and also classes that implement the specified interfaces.

Note: The context is not automatically rebuilt on every build. After making a change to the interface definitions it is necessary to run the custom tool again.

5.4 Using the Context

The context can be used inside any .NET application or web service. The commented code below shows how to initialize a context and then use that context to create and persist data. It concludes by showing how to query the database using LINQ:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using BrightstarDB.Client;


namespace GettingStarted
{
    class Program
    {
        static void Main(string[] args)
        {

            // define a connection string
            const string connectionString = "type=embedded;storesdirectory=.\\;storename=Films";


            // if the store does not exist it will be automatically
            // created when a context is created
            var ctx = new MyEntityContext(connectionString);


            // create some films
            var bladeRunner = ctx.Films.Create();
            bladeRunner.Name = "BladeRunner";
```

```
            var starWars = ctx.Films.Create();
            starWars.Name = "Star Wars";


            // create some actors and connect them to films
            var ford = ctx.Actors.Create();
            ford.Name = "Harrison Ford";
            ford.DateOfBirth = new DateTime(1942, 7, 13);
            ford.Films.Add(starWars);
            ford.Films.Add(bladeRunner);


            var hamill = ctx.Actors.Create();
            hamill.Name = "Mark Hamill";
            hamill.DateOfBirth = new DateTime(1951, 9, 25);
            hamill.Films.Add(starWars);


            // save the data
            ctx.SaveChanges();


            // open a new context, not required
            ctx = new MyEntityContext(connectionString);


            // find an actor via LINQ
            ford = ctx.Actors.Where(a => a.Name.Equals("Harrison Ford")).FirstOrDefault();
            var dob = ford.DateOfBirth;


            // list his films
            var films = ford.Films;


            // get star wars
            var sw = films.Where(f => f.Name.Equals("Star Wars")).FirstOrDefault();


            // list actors in star wars
            foreach (var actor in sw.Actors)
            {
                var actorName = actor.Name;
                Console.WriteLine(actorName);
            }

            Console.ReadLine();
        }
    }
}
```

5.5 Optimistic Locking

Optimistic Locking is a way of handling concurrency control, meaning that multiple transactions can complete without affecting each other. If Optimistic Locking is turned on, then when a transaction tries to save data to the store, it first

checks that the underlying data has not been modified by a different transaction. If it finds that the data has been modified, then the transaction will fail to complete.

BrightstarDB has the option to turn on optimistic locking when connecting to the store. This is done by setting the enableOptimisticLocking flag when opening a context such as below:

```
ctx = new MyEntityContext(connectionString, true);
var newFilm = ctx.Films.Create();
ctx.SaveChanges();


var newFilmId = newFilm.Id;


//use optimistic locking when creating a new context
var ctx1 = new MyEntityContext(connectionString, true);
var ctx2 = new MyEntityContext(connectionString, true);


//create a film in the first context
var film1 = ctx1.Films.Where(f => f.Id.Equals(newFilmId)).FirstOrDefault();
Console.WriteLine("First context has film with ID '{0}'", film1.Id);
//create a film in the second context
var film2 = ctx2.Films.Where(f => f.Id.Equals(newFilmId)).FirstOrDefault();
Console.WriteLine("Second context has film with ID '{0}'", film2.Id);


//attempt to change the data from both contexts
film1.Name = "Raiders of the Lost Ark";
film2.Name = "American Graffiti";


//save the data to the store
try
{
  ctx1.SaveChanges();
  Console.WriteLine("Successfully updated the film to '{0}' in the store", film1.Name);
  ctx2.SaveChanges();
}
catch (Exception ex)
{
Console.WriteLine("Unable to save data to the store, as the underlying data has been modified.");
}

Console.ReadLine();
```

Note: Optimistic Locking can also be enabled in the configuration using the BrightstarDB.EnableOptimisticLocking application setting

5.6 Server Side Caching

When enabled, query results are stored on disk until an update is made. If the same query is executed, the cached result is returned. Cached results are stored in the Windows temporary folder, and deleted when an update is made to the store.

Server side caching is enabled by default, but can be disabled by adding the appSetting below to the application

configuration file:

```
<add key="BrightstarDB.EnableServerSideCaching" value="false" />
```

Note: Server side caching is not supported on BrightstarDB for Windows Phone 7.

5.7 What Next?

While this is just a short introduction it has covered a lot of how BrightstarDB works. The following sections provide some more conceptual details on how the store works, more details on the Entity Framework and how to work with BrightstarDB as a triple store.

CHAPTER 6

Connection Strings

BrightstarDB makes use of connection strings for accessing both embedded and remote BrightstarDB instances. The following section describes the different connection string properties.

Type : This property specifies the type of connection to create. Allowed values are:

Type	Description	Other Properties For Connection
embedded	Uses the embedded BrightstarDB server to directly open stores from the local file system.	StoresDirectory
rest	Uses HTTP(S) to connect to a BrightstarDB service.	Endpoint
dotNetRdf	Connects to another (non-BrightstarDB) store using DotNetRDF connectors	Configuration, StorageServer
sparql	Connects to another (non-BrightstarDB) store using SPARQL protocols	Query, Update

StoresDirectory : value is a file system path to the directory containing all BrightstarDB data. Only valid for use with **Type** set to *embedded*.

Endpoint : a URI that points to the service endpoint for the specified remote service. Only valid for connections with **Type** set to *rest*

StoreName : The name of a specific store to connect to. This property is only required when creating an EntityFramework connection or when creating a connection using the dotNetRdf connection type.

Configuration : The path to the RDF file that contains the configuration for the DotNetRDF connector. Only valid for connections with **Type** set to *dotNetRdf*. For more information please refer to the section :ref:Other_Stores

StorageServer : The URI of the resource in the DotNetRDF configuration file that configures the DotNetRDF storage server to be used for the connection. Only valid for connections with **Type** set to *dotNetRdf*. For more information please refer to the section :ref:Other_Stores

Query : The URI the SPARQL query endpoint to connect to. Only valid for connections with **Type** set to *sparql*.

Update: The URI of the SPARQL update endpoint to connect to. Only valid for connections with **Type** set to *sparql*. If this option is used in a connection string, then the **Query** property must also be provided.

OptimisticLocking: Specifies if optimistic locking should be enabled for the connection by default. Note that this setting can be overridden in code, allowing developers full control over whether or not optimistic locking is used. This option is only used by the *Data Object Layer* and *Entity Framework* and is currently not supported on connections of type *dotNetRDF*

UserName: Specifies the user name to use for authenticating with the server. A connection string with this property must also have a **Password** property for authentication to take place.

Password: Specifies the password to use for authenticating with the server. A connection string with this property must also have a **UserName** property for authentication to take place.

Note: You should never store credentials in a connection string as plain text. Instead your application should store the base connection string without the UserName and Password properties. It should then prompt the user to enter their credentials just before it creates the BrightstarDB client and append the UserName and Password properties to the base connection string.

The following are examples of connection strings. Property value pairs are separated by ';' and property names are case insensitive.:

```
// A connection to a BrightstarDB server running on localhost.
// The connection is configured with a default store to use for the Entity Framework
"type=rest;endpoint=http://localhost:8090/brightstar;storename=test"

// An embedded connection to the store named "test" in the directory c:\Brightstar
"type=embedded;storesdirectory=c:\brightstar;storename=test"

// An embedded connection to the stores contained in the directory c:\Brightstar
"Type=embedded;StoresDirectory=c:\Brightstar"

// A connection to one or more store providers configured in a DotNetRDF configuration file
"Type=dotnetrdf;configuration=c:\brightstar\dotNetRDFConfiguration.ttl"

// A connection to a storage server such as a Sesame server configured in a DotNetRDF configuration f
"Type=dotnetrdf;configuration=c:\brightstar\sesameConfiguration.ttl;storageServer=http://example.org/
// NOTE: It is also possible to use relative URIs (resolved against the base URI of the configuratio
"Type=dotnetrdf;configuration=c:\brightstar\sesameConfiguration.ttl;storageServer=#sesameServer"

// A read-write connection to a server with SPARQL query and SPARQL update endpoints
"Type=sparql;query=http://example.org/sparql;update=http://example.org/sparql-update"

// A read-only connection to a server with only a SPARQL query endpoint
"Type=sparql;query=http://example.org/sparql"
```

CHAPTER 7

Store Persistence Types

BrightstarDB supports two different file formats for storing its index information. The main difference between the two formats is the way in which modified pages of the index are written to the index file.

7.1 Append-Only

The Append-Only format means that BrightstarDB will write modified pages to the end of the index file. This approach has a number of benefits:

1. Writers never block readers, so any number of read operations (typically SPARQL queries) can be executed in parallel with updates to the index. Each reader accesses the store in the state that it was when their operation began.
2. Reads can access any previous state of the store. This is because the full history of updates to pages is maintained by the store.
3. Writes are faster - because they only append to the end of the file rather than needing to seek to a location within the file to be updated.

The down-side of this format is that the index file will grow not only as more data is added but also with every update operation applied to the store. BrightstarDB does provide a way to truncate a store to just its latest state, removing all the previous historical page states so this operation executed periodically can help to keep the file size under control.

In general the Append-Only format is recommended for most systems as long as disk space is not constrained.

7.2 Rewritable

The Rewriteable store format manages an active and a shadow copy of each page in the index. Writes are directed to the shadow copy while readers can access the current committed state of the store by reading from the active copy. On a commit, the shadow copy becomes the active and vice-versa. This approach keeps file size under control as changes to an index page are always written to one of the two copies of the page. However this format has some disadvantages compared to the append-only store.

1. Readers that take a long time to complete can get blocked by writers. In general if a reader completes in the time taken for a write to complete, the two operations can execute in parallel, however in the case that a reader requires access to the store across two successive reads, there is the potential that index pages could be modified. To avoid inconsistent results due to dirty reads, when a reader detects this it will automatically retry its current operation. This means that in stores where there are frequent, small updates readers can potentially be blocked for a long time as new writes keep forcing the read operation to be retried.

2. Write operations can be a bit slower - this is because pages are written to a fixed location within the index file, requiring a disk seek before each page write.

In general the Rewritable store format is recommended for embedded applications; for mobile devices that have space constraints to consider; or for server applications that are only required to support infrequent and/or large updates.

7.3 Specifying the Store Persistence Type

The persistence type to use for a store must be specified when the store is created and cannot be changed after the store has been created. The default persistence type is configured in the application configuration file for the application (or the web.config for web applications). To configure the default, you must add an entry to the appSetting section of the application configuration file with the key `BrightstarDB.PersistenceType` and the value `appendonly` for an Append-Only store or `rewrite` for a Rewriteable store (in both cases the values are case-insensitive).

It is also possible to override the default persistence type at runtime by calling the appropriate `CreateStore()` operation on the BrighstarDB service client API. If no default value is defined in the application configuration file and no override value is passed to the `CreateStore()` method, the default persistence type used by BrightstarDB is the Append-Only persistence type.

CHAPTER 8

Running The BrightstarDB Samples

All samples can be found in [INSTALLDIR]\Samples. Some samples are written to run against a local BrightstarDB service. These samples only need editing if you want to run them against BrightstarDB running on a different machine or running on a non-default port. This is achieved by altering the BrightstarDB.ConnectionString property in the web.config file of the sample.

CHAPTER 9

Entity Framework

The BrightstarDB Entity Framework is the main way of working with BrightstarDB instances. For those of you wanting to work with the underlying RDF directly please see the section on *RDF Client API*. BrightstarDB allows developers to define a data model using .NET interface definitions. BrightstarDB tools introspect these definitions to create concrete classes that can be used to create, and update persistent data. If you haven't read the *Getting Started* section then we recommend that you do. The sample provided there covers most of what is required for creating most data models. The following sections in the developer guide provide more in-depth explanation of how things work along with more complex examples.

9.1 Basics

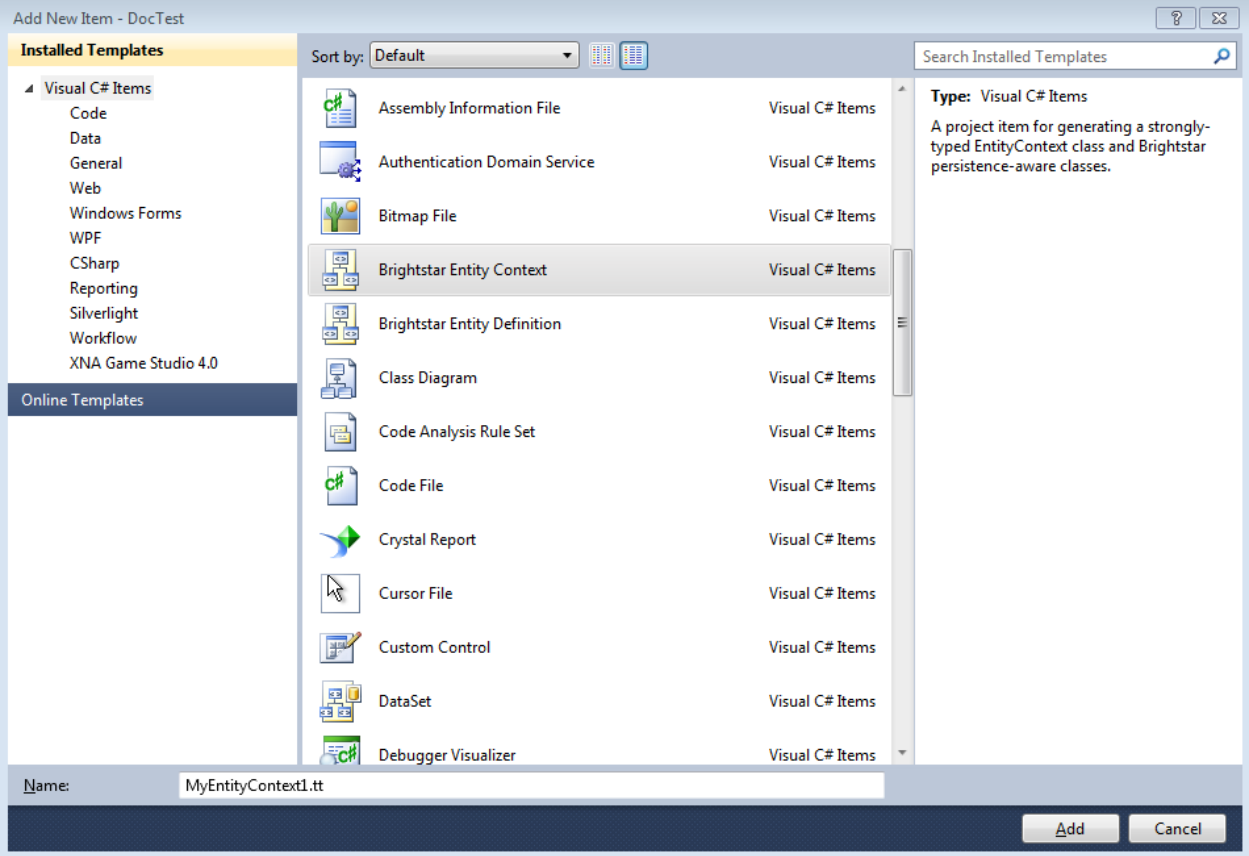
The BrightstarDB Entity Framework tooling is very simple to use. This guide shows how to get going, the rest of this section provides more in-depth information.

The process of using the Entity Framework is to:

1. Include the BrightstarDB Entity Context item into a project.
2. Define the interfaces for the data objects that should be persistent.
3. Run the custom tool on the Entity Context text template file.
4. Use the generated context to create, query or get and modify objects.

Including the BrightstarDB Entity Context

The **Brightstar Entity Context** is a text template that when run introspects the other code elements in the project and generates a number of classes and a context in a single file that can be found under the context file in Visual Studio. To add a new BrightstarEntityContext add a new item to the project. Locate the item in the list called Brightstar Entity Context, rename it if required, and add to the current project.



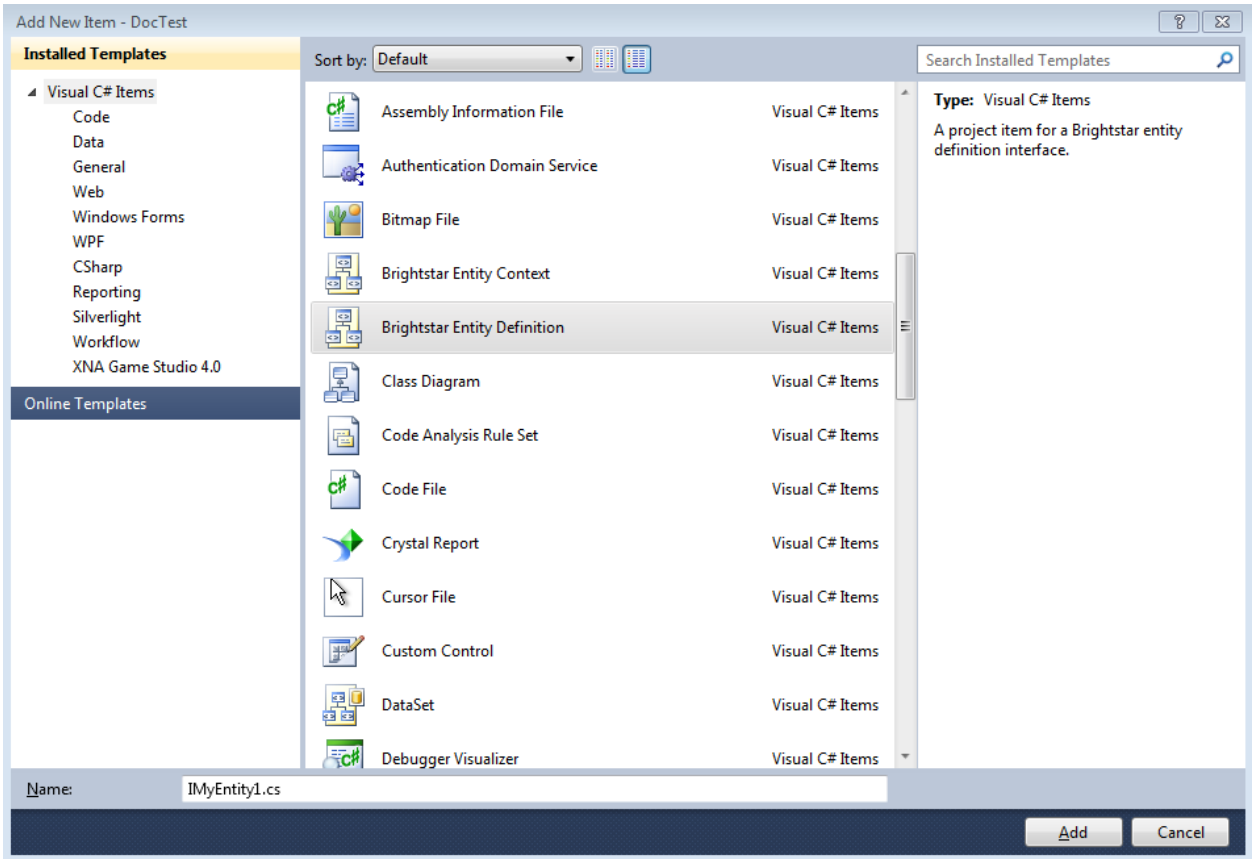
Define Interfaces

Interfaces are used to define a data model contract. Only interfaces marked with the `Entity` attribute will be processed by the text template. The following interfaces define a model that captures the idea of people working for an company.:

```
[Entity]
public interface IPerson
{
    string Name { get; set; }
    DateTime DateOfBirth { get; set; }
    string CV { get; set; }
    ICompany Employer { get; set; }
}

[Entity]
public interface ICompany
{
    string Name { get; set; }
    [InverseProperty("Employer")]
    ICollection<IPerson> Employees { get; }
}
```

Including a Brightstar Entity Definition Item

One quick way to include the outline of a BrightstarDB entity in a project is to right click on the project in the solution explorer and click **Add New Item**. Then select the **Brightstar Entity Definition** from the list and update the name.



This will add the following code file into the project.:

```
[Entity]
public interface IMyEntity1
{
    /// <summary>
    /// Get the persistent identifier for this entity
    /// </summary>
    string Id { get; }


    // TODO: Add other property references here
}
```

Run the MyEntityContext.tt Custom Tool

To ensure that the generated classes are up to date right click on the .tt file in the solution explorer and select **Run Custom Tool**. This will ensure that the all the annotated interfaces are turned into concrete classes.

Note: The custom tool is not run automatically on every rebuild so after changing an interface remember to run it.

Using a Context

A context can be thought of as a connection to a BrightstarDB instance. It provides access to the collections of domain objects defined by the interfaces. It also tracks all changes to objects and is responsible for executing queries and committing transactions.

A context can be opened with a connection string. If the store named does not exist it will be created. See the *connection strings* section for more information on allowed configurations. The following code opens a new context

connecting to an embedded store:

```
var dataContext = new MyEntityContext("Type=embedded;StoresDirectory=c:\\brightstardb;StoreName=test"
```

The context exposes a collection for each entity type defined. For the types we defined above the following collections are exposed on a context:

```
var people = dataContext.Persons;
var companies = dataContext.Companies;
```

Each of these collections are in fact IQueryable and as such support LINQ queries over the model. To get an entity by a given property the following can be used:

```
var brightstardb = dataContext.Companies.Where(
                      c => c.Name.Equals("BrightstarDB")).FirstOrDefault();
```

Once an entity has been retrieved it can be modified or related entities can be fetched:

```
// fetching employees
var employeesOfBrightstarDB = brightstardb.Employees;

// update the company
brightstardb.Name = "BrightstarDB";
```

New entities can be created either via the main collection; by using the `new` keyword and attaching the object to the context; or by passing the context into the constructor:

```
// creating a new entity via the context collection
var bob = dataContext.Persons.Create();
bob.Name = "bob";


// or created using new and attached to the context
var bob = new Person() { Name = "Bob" };
dataContext.Persons.Add(bob);

// or created using new and passing the context into the constructor
var bob = new Person(dataContext) { Name = "Bob" };
```

Once a new object has been created it can be used in relationships with other objects. The following adds a new person to the collection of employees. The same relationship could also have been created by setting the `Employer` property on the person:

```
// Adding a new relationship between entities
var bob = dataContext.Persons.Create();
bob.Name = "bob";
brightstardb.Employees.Add(bob);


// The relationship can also be defined from the 'other side'.
var bob = dataContext.Persons.Create();
bob.Name = "bob";
bob.Employer = brightstardb;

// You can also create relationships to previously constructed
// or retrieved objects in the constructor
var brightstardb = new Company(dataContext) { Name = "BrightstarDB" };
var bob = new Person(dataContext) {
                  Name = "Bob;
```

```
            Employer = brightstardb
        };
```

Saving the changes that have occurred is easily done by calling a method on the context:

```
dataContext.SaveChanges();
```

9.2 Annotations

The BrightstarDB entity framework relies on a few annotation types in order to accurately express a data model. This section describes the different annotations and how they should be used. The only required attribute annotation is Entity. All other attributes give different levels of control over how the object model is mapped to RDF.

9.2.1 TypeIdentifierPrefix Attribute

BrightstarDB makes use of URIs to identify class types and property types. These URI values can be added on each property but to improve clarity and avoid mistakes it is possible to configure a base URI that is then used by all attributes. It is also possible to define models that do not have this attribute set.

The type identifier prefix can be set in the AssemblyInfo.cs file. The example below shows how to set this configuration property:

```
[assembly: TypeIdentifierPrefix("http://www.mydomain.com/types/")]
```

9.2.2 Entity Attribute

The `Entity` attribute is used to indicate that the annotated interface should be included in the generated model. Optionally, a full URI or a URI postfix can be supplied that defines the identity of the class. The following examples show how to use the attribute. The example with just the value 'Person' uses a default prefix if one is not specified as described above:

```
// example 1.
[Entity]
public interface IPerson { ... }

// example 2.
[Entity("Person")]
public interface IPerson { ... }

// example 3.
[Entity("http://xmlns.com/foaf/0.1/Person")]
public interface IPerson { ... }
```

Example 3. above can be used to map .NET models onto existing RDF vocabularies. This allows the model to create data in a given vocabulary but it also allows models to be mapped onto existing RDF data.

9.2.3 Identity Property

The Identity property can be used to get and set the underlying identity of an Entity. The following example shows how this is defined:

```
// example 1.
[Entity("Person")]
public interface IPerson {
  string Id { get; }
}
```

No annotation is required. It is also acceptable for the property to be called `ID`, `{Type}Id` or `{Type}ID` where `{Type}` is the name of the type. E.g: `PersonId` or `PersonID`.

9.2.4 Identifier Attribute

Id property values are URIs, but in some cases it is necessary to work with simpler string values such as GUIDs or numeric values. To do this the Id property can be decorated with the identifier attribute. The identifier attribute requires a string property that is the identifier prefix - this can be specified either as a URI string or as {prefix}:{rest of URI} where {prefix} is a namespace prefix defined by the Namespace Declaration Attribute (see below):

```
// example 1.
[Entity("Person")]
public interface IPerson {
  [Identifier("http://www.mydomain.com/people/")]
  string Id { get; }
}

// example 2.
[Entity]
public interface ISkill {
  [Identifier("ex:skills#")]
  string Id {get;}
}
// NOTE: For the above to work there must be an assembly attribute declared like this:
[assembly:NamespaceDeclaration("ex", "http://example.org/")]
```

The `Identifier` attribute has additional arguments that enable you to specify a (composite) key for the type. For more information please refer to the section *Key Properties and Composite Keys*.

9.2.5 Property Inclusion

Any .NET property with a getter or setter is automatically included in the generated type, no attribute annotation is required for this:

```
// example 1.
[Entity("Person")]
public interface IPerson {
  string Id { get; }
  string Name { get; set; }
}
```

9.2.6 Property Exclusion

If you want BrightstarDB to ignore a property you can simply decorate it with an `[Ignore]` attribute:

```
[Entity("Person")]
public interface IPerson {
  string Id {get; }
```

```
  string Name { get; set; }

  [Ignore]
  int Salary {get;}
}
```

Note: Properties that are ignored in this way are not implemented in the partial class that BrightstarDB generates, so you will need to ensure that they are implemented in a partial class that you create.

Note: The `[Ignore]` attribute is not supported or required on *methods* defined in the interface as BrightstarDB does not implement interface methods - you are always required to provide method implementations in your own partial class.

9.2.7 Inverse Property Attribute

When two types reference each other via different properties that in fact reflect different sides of the same association then it is necessary to declare this explicitly. This can be done with the InverseProperty attribute. This attribute requires the name of the .NET property on the referencing type to be specified:

```
// example 1.
[Entity("Person")]
public interface IPerson {
  string Id { get; }
  ICompany Employer { get; set; }
}

[Entity("Company")]
public interface IPerson {
  string Id { get; }
  [InverseProperty("Employer")]
  ICollection<IPerson> Employees { get; set; }
}
```

The above example shows that the inverse of `Employees` is `Employer`. This means that if the `Employer` property on `P1` is set to `C1` then getting `C1.Employees` will return a collection containing `P1`.

9.2.8 Namespace Declaration Attribute

When using URIs in annotations it is cleaner if the complete URI doesn't need to be entered every time. To support this the NamespaceDeclaration assembly attribute can be used, many times if needed, to define namespace prefix mappings. The mapping takes a short string and the URI prefix to be used.

The attribute can be used to specify the prefixes required (typically assembly attributes are added to the AssemblyInfo.cs code file in the Properties folder of the project):

```
[assembly: NamespaceDeclaration("foaf", "http://xmlns.com/foaf/0.1/")]
```

Then these prefixes can be used in property or type annotation using the CURIE syntax of {prefix}:{rest of URI}:

```
[Entity("foaf:Person")]
public interface IPerson  { ... }
```

Namespace declarations defined in this way can also be retrieved programatically. The class `BrightstarDB.EntityFramework.NamespaceDeclarations` provides methods for retrieving these

declarations in a variety of formats:

```
// You can just iterate them as instances of
// BrightstarDB.EntityFramework.NamespaceDeclarationAttribute
foreach(var nsDecl in NamespaceDeclarations.ForAssembly(Assembly.GetExecutingAssembly()))
{
    // prefix is in nsDecl.Prefix
    // Namespace URI is in nsDecl.Reference
}

// Or you can retrieve them as a dictionary:
var dict = NamespaceDeclarations.ForAssembly(Assembly.GetExecutingAssembly());
foafUri = dict["foaf"];

// You can omit the Assembly parameter if you are calling from the assembly
// containing the delcarations.

// You can get the declarations formatted for use in SPARQL...
// e.g. PREFIX foaf: <http://xmlns.com/foaf/0.1/>
sparqlPrefixes = NamespaceDeclarations.ForAssembly().AsSparql();

// ...or for use in Turtle (or TRiG)
// e.g. @prefix foaf: <http://xmlns.com/foaf/0.1/> .
turtlePrefixes = NamespaceDeclarations.ForAssembly().AsTurtle();
```

9.2.9 Property Type Attribute

While no decoration is required to include a property in a generated class, if the property is to be mapped onto an existing RDF vocabulary then the PropertyType attribute can be used to do this. The PropertyType attribute requires a string property that is either an absolute or relative URI. If it is a relative URI then it is appended to the URI defined by the TypeIdentifierPrefix attribute or the default base type URI. Again, prefixes defined by a NamespaceDeclaration attribute can also be used:

```
// Example 1. Explicit type declaration
[PropertyType("http://www.mydomain.com/types/name")]
string Name { get; set; }

// Example 2. Prefixed type declaration.
// The prefix must be declared with a NamespaceDeclaration attribute
[PropertyType("foaf:name")]
string Name { get; set; }


// Example 3. Where "name" is appended to the default namespace
// or the one specified by the TypeIdentifierPrefix in AssemblyInfo.cs.
[PropertyType("name")]
string Name { get; set; }
```

9.2.10 Inverse Property Type Attribute

Allows inverse properties to be mapped to a given RDF predicate type rather than a .NET property name. This is most useful when mapping existing RDF schemas to support the case where the .NET data-binding only requires the inverse of the RDF property:

```
// Example 1. The following states that the collection of employees
// is found by traversing the "http://www.mydomain.com/types/employer"
// predicate from instances of Person.
[InversePropertyType("http://www.mydomain.com/types/employer")]
ICollection<IPerson> Employees { get; set; }
```

9.2.11 Additional Custom Attributes

Any custom attributes added to the entity interface that are not in the BrightstarDB.EntityFramework namespace will be automatically copied through into the generated class. This allows you to easily make use of custom attributes for validation, property annotation and other purposes.

As an example, the following interface code:

```
[Entity("http://xmlns.com/foaf/0.1/Person")]
public interface IFoafPerson : IFoafAgent
{
    [Identifier("http://www.networkedplanet.com/people/")]
    string Id { get; }

    [PropertyType("http://xmlns.com/foaf/0.1/nick")]
    [DisplayName("Also Known As")]
    string Nickname { get; set; }

    [PropertyType("http://xmlns.com/foaf/0.1/name")]
    [Required]
    [CustomValidation(typeof(MyCustomValidator), "ValidateName",
                      ErrorMessage="Custom error message")]
    string Name { get; set; }
}
```

would result in this generated class code:

```
 public partial class FoafPerson : BrightstarEntityObject, IFoafPerson
 {
 public FoafPerson(BrightstarEntityContext context, IDataObject dataObject) : base(context, dataObje
 public FoafPerson() : base() { }
 public System.String Id { get {return GetIdentity(); } set { SetIdentity(value); } }
 #region Implementation of BrightstarDB.Tests.EntityFramework.IFoafPerson

 [System.ComponentModel.DisplayNameAttribute("Also Known As")]
 public System.String Nickname
 {
         get { return GetRelatedProperty<System.String>("Nickname"); }
         set { SetRelatedProperty("Nickname", value); }
 }

 [System.ComponentModel.DataAnnotations.RequiredAttribute]
 [System.ComponentModel.DataAnnotations.CustomValidationAttribute(typeof(MyCustomValidator),
   "ValidateName", ErrorMessage="Custom error message")]
 public System.String Name
 {
         get { return GetRelatedProperty<System.String>("Name"); }
         set { SetRelatedProperty("Name", value); }
 }

#endregion
```

```
}
```

It is also possible to add custom attributes to the generated entity class itself. Any custom attributes that are allowed on both classes and interfaces can be added to the entity interface and will be automatically copied through to the generated class in the same was as custom attributes on properties. However, if you need to use a custom attribute that is allowed on a class but not on an interface, then you must use the BrightstarDB.EntityFramework.ClassAttribute attribute. This custom attribute can be added to the entity interface and allows you to specify a different custom attribute that should be added to the generated entity class. When using this custom attribute you should ensure that you either import the namespace that contains the other custom attribute or reference the other custom attribute using its fully-qualified type name to ensure that the generated class code compiles successfully.

For example, the following interface code:

```
[Entity("http://xmlns.com/foaf/0.1/Person")]
[ClassAttribute("[System.ComponentModel.DisplayName(\\"Person\\")]")]
public interface IFoafPerson : IFoafAgent
{
  // ... interface definition here
}
```

would result in this generated class code:

```
[System.ComponentModel.DisplayName("Person")]
public partial class FoafPerson : BrightstarEntityObject, IFoafPerson
{
  // ... generated class code here
}
```

Note that the DisplayName custom attribute is referenced using its fully-qualified type name (`System.ComponentModel.DisplayName`), as the generated context code will not include a `using System.ComponentModel;` namespace import. Alternatively, this interface code would also generate class code that compiles correctly:

```
using System.ComponentModel;

[Entity("http://xmlns.com/foaf/0.1/Person")]
[ClassAttribute("[DisplayName(\\"Person\\")]")]
public interface IFoafPerson : IFoafAgent
{
  // ... interface definition here
}
```

9.3 Patterns

This section describes how to model common patterns using BrightstarDB Entity Framework. It covers how to define one-to-one, one-to-many, many-to-many and reflexive relationships.

Examples of these relationship patterns can be found in the *Tweetbox sample*.

9.3.1 One-to-One

Entities can have one-to-one relationships with other entities. An example of this would be the link between a user and a the authorization to another social networking site. The one-to-one relationship would be described in the interfaces as follows:

```
[Entity]
public interface IUser {
  ...
  ISocialNetworkAccount SocialNetworkAccount { get; set; }
  ...
}

[Entity]
public interface ISocialNetworkAccount {
  ...
  [InverseProperty("SocialNetworkAccount")]
          IUser TwitterAccount { get; set; }
  ...
}
```

9.3.2 One-to-Many

A User entity can be modeled to have a one-to-many relationship with a set of Tweet entities, by marking the properties in each interface as follows:

```
[Entity]
public interface ITweet {
  ...
  IUser Author { get; set; }
  ...
}

[Entity]
public interface IUser {
  ...
  [InverseProperty("Author")]
  ICollection<ITweet> Tweets { get; set; }
  ...
}
```

9.3.3 Many-to-Many

The Tweet entity can be modeled to have a set of zero or more Hash Tags. As any Hash Tag entity could be used in more than one Tweet, this uses a many-to-many relationship pattern:

```
[Entity]
public interface ITweet {
  ...
  ICollection<IHashTag> HashTags { get; set; }
  ...
}

[Entity]
public interface IHashTag {
  ...
  [InverseProperty("HashTags")]
  ICollection<ITweet> Tweets { get; set; }
  ...
}
```

9.3.4 Reflexive relationship

A reflexive relationship (that refers to itself) can be defined as in the example below:

```
[Entity]
public interface IUser {
  ...
  ICollection<IUser> Following { get; set; }

  [InverseProperty("Following")]
  ICollection<IUser> Followers { get; set; }
  ...
}
```

9.4 Behaviour

The classes generated by the BrightstarDB Entity Framework deal with data and data persistence. However, most applications require these classes to have behaviour. All generated classes are generated as .NET partial classes. This means that another file can contain additional method definitions. The following example shows how to add additional methods to a generated class.

Assume we have the following interface definition:

```
[Entity]
public interface IPerson {
  string Id { get; }
  string FirstName { get; set; }
  string LastName { get; set; }
}
```

To add custom behaviour the new method signature should first be added to the interface. The example below shows the same interface but with an added method signature to get a user's full name:

```
[Entity]
public interface IPerson {
  string Id { get; }
  string FirstName { get; set; }
  string LastName { get; set; }
  // new method signature
  string GetFullName();
}
```

After running the custom tool on the EntityContext.tt file there is a new class called Person. To add additional methods add a new .cs file to the project and add the following class declaration:

```
public partial class Person {
  public string GetFullName() {
    return FirstName + " " + LastName;
  }
}
```

The new partial class implements the additional method declaration and has access to all the data properties in the generated class.

9.5 Key Properties and Composite Keys

The *Identity Property* provides a simple means of accessing the key value of an entity, this key value is concatenated with the base URI string for the entity type to generate the full URI identifier of the RDF resource that is created for the entity. In many applications the exact key used is immaterial, and the default strategy of generating a GUID-based key works well. However in some cases it is desirable to have more control over the key assigned to an entity. For this purpose we provide a number of additional arguments on the `Identifier` attribute. These arguments allow you to specify that the key for an entity type is generated from one or more of its properties

9.5.1 Specifying Key Properties

The `KeyProperties` argument accepts an array of strings that name the properties of the entity that should be combined to create a key value for the entity. The value of the named properties will be concatenated in the order that they are named in the `KeyProperties` array, with a slash ('/') between values:

```
// An entity with a key generated from one of its properties.
[Entity]
public interface IBook {
    [Identifier("http://example.org/books/",
                KeyProperties=new [] {"Isbn"}]
    public string Id { get; }
    public string Isbn {get;set;}
}

// An entity with a composite key
[Entity]
public interface IWidget {
    [Identifier("http://widgets.org/",
        KeyProperties=new [] {"Manufacturer", "ProductCode"}]
    public string Id { get; }
    public string Manufacturer {get;set;}
    public string ProductCode  {get;set;}
}

// In use...
var book = context.Books.Create();
book.Isbn = "1234567890";
// book URI identifier will be http://example.org/books/1234567890

var widget = context.Widgets.Create();
widget.Manufacturer = "Acme";
widget.ProductCode = "Grommet"
// widget identifier will be http://widgets.org/Acme/Grommet
```

9.5.2 Key Separator

The `KeySeparator` argument of the `Identifier` attribute allows you to change the string used to concatenate multiple values into a single key:

```
// An entity with a composite key
[Entity]
public interface IWidget {
    [Identifier("http://widgets.org/",
        KeyProperties=new [] {"Manufacturer", "ProductCode"},
        KeySeparator="_"]
```

```
    public string Id { get; }
    public string Manufacturer {get;set;}
    public string ProductCode  {get;set;}
}

var widget = context.Widgets.Create();
widget.Manufacturer = "Acme";
widget.ProductCode = "Grommet"
// widget identifier will be http://widgets.org/Acme_Grommet
```

9.5.3 Key Converter

The values of the key properties are converted to a string by a class that implements the `BrightstarDB.EntityFramework.IKeyConverter` interface. The default implementation implements the following rules:

- Integer and decimal values are converted using the InvariantCulture (to eliminate culture-specific separators)
- Properties whose value is another entity will yield the key of that entity. That is the part of the URI identifier that follows the base URI string.
- Properties whose value is NULL are ignored.
- If all key properties are NULL, a NULL key will be generated, which will result in a `BrightstarDB.EntityFramework.EntityKeyRequiredException` being raised.
- The converted string value is URI-escaped using the .NET method `Uri.EscapeUriString(string)`.
- Multiple non-null values are concatenated using the separator specified by the KeySeparator property.

You can create your own key conversion rules by implementing the `IKeyConverter` interface and specifying the implementation type in the `KeyConverterType` argument of the `Identifier` attribute.

9.5.4 Hierarchical Key Pattern

Using the default key conversion rules it is possible to construct hierarchical identifier schemes:

```
[Entity]
public interface IHierarchicalKeyEntity
{
    [Identifier(BaseAddress = "http://example.org/",
     KeyProperties = new[]{"Parent", "Code"})]
    string Id { get; }

    IHierarchicalKeyEntity Parent { get; set; }
    string Code { get; set; }
}

// Example:
var parent = context.HierarchicalKeyEntities.Create();
parent.Code = "parent"; // URI will be http://example.org/parent

var child = context.HierarchicalKeyEntities.Create();
child.Parent = parent;
child.Code = "child"; // URI will be http://example.org/parent/child
```

Note: Although this example uses the same type of entity for both parent and child object, it is equally valid to use

different types of entity for parent and child.

9.5.5 Key Constraints

When using the Entity Framework with the BrightstarDB back-end, entities with key properties are treated as having a "class-unique key constraint". This means that it is not allowed to create an RDF resource with the same URI identifier and the same RDF type. This form of constraint means that it is possible for one resource to have multiple types, but it still ensures that for any given type all of its identifiers are unique.

The constraint is checked as part of the update transaction and if it fails a `BrightstarDB.EntityFramework.UniqueConstraintViolationException` will be raised. The constraint is also checked when creating new entities, but in this case the check is only against the entities currently loaded into the context - this allows your code to "fail fast" if a uniqueness violation occurs in the collection of entities loaded in the context.

Warning: Key constraints are not checked when using the Entity Framework with a DotNetRDF or generic SPARQL back-end, as the SPARQL UPDATE protocol does not allow for such transaction pre-conditions to be checked.

9.5.6 Changing Identifiers

With release 1.7 of BrightstarDB, it is now possible to alter the URI identifier of an entity. Currently this is only supported on entities that have generated keys and is achieved by modifying any of the properties that contribute to the key.

A change of identifier is handled by the Entity Framework as a deletion of all triples where the old identifier is the subject or object of the triple, followed by the creation of a new set of triples equivalent to the deleted set but with the old identifier replaced by the new identifier. Because the triples where the identifier is used as the object are updated, all "links" in the data set will be properly maintained when an identifier is modified in this way.

Warning: When using another entity ID as part of the composite key for an entity please be aware that currently the entity framework code does not automatically change the identifiers of all dependencies when a dependent ID property is changed. This is done to avoid a large amount of overhead in checking for ID dependencies in the data store when changes are saved. The supported use case is that the dependency ID (e.g. the ID of the parent entity) is not modified once it is used to construct other identifiers.

9.6 Optimistic Locking

The Entity Framework provides the option to enable optimistic locking when working with the store. Optimistic locking uses a well-known version number property (the property predicate URI is http://www.brightstardb.com/.well-known/model/version) to track the version number of an entity, when making an update to an entity the version number is used to determine if another client has concurrently updated the entity. If this is detected, it results in an exception of the type `BrightstarDB.Client.TransactionPreconditionsFailedException` being raised.

9.6.1 Enabling Optimistic Locking

Optimistic locking can be enabled either through the connection string (giving the user control over whether or not optimistic locking is enabled) or through code (giving the control to the programmer).

To enable optimistic locking in a connection string, simply add "optimisticLocking=true" to the connection string. e.g.

```
type=rest;endpoint=http://localhost:8090/brightstar;storeName=myStore;optimisticLocking=true
```

To enable optimistic locking from code, use the optional optimisticLocking parameter on the constructor of the context class e.g.:

```
var myContext = new MyEntityContext(connectionString, true);
```

Note: The programmatic setting always overrides the setting in the connection string - this gives the programmer final control over whether optimistic locking is used. The programmer can also prevent optimistic locking from being used by passing false as the value of the `optimisticLocking` parameter of the constructor of the context class.

9.6.2 Handling Optimistic Locking Errors

Optimistic locking errors only occur when the `SaveChanges()` method is called on the context class. The error is notified by raising an exception of the type `BrightstarDB.Client.TransactionPreconditionsFailedException`. When this exception is caught by your code, you have two basic options to choose from. You can apply each of these options separately to each object modified by your update.

1. Attempt the save again but first update the local context object with data from the server. This will save all the changes you have made EXCEPT for those that were detected on the server. This is the "store wins" scenario.
2. Attempt the save again, but first update only the version numbers of the local context object with data from the server. This will keep all the changes you have made, overwriting any concurrent changes that happened on the server. This is the "client wins" scenario.

To attempt the save again, you must first call the `Refresh()` method on the context object. This method takes two paramters - the first parameter specifies the mode for the refresh, this can either be RefreshMode.ClientWins or RefreshMode.StoreWins depending on the scenario to be applied. The second parameter is the entity or collection of entities to which the refresh is to be applied. You apply different refresh strategies to different entities within the same update if you wish. Once the conflicted entities are refreshed, you can then make a call to the `SaveChanges()` method of the context once more. The code sample below shows this in outline:

```
try
{
  myContext.SaveChanges();
}
catch(TransactionPreconditionsFailedException)
{
  // Refresh the conflicted object(s) - in this case with the StoreWins mode
  myContext.Refresh(RefreshMode.StoreWins, conflictedEntity);
  // Attempt the save again
  myContext.SaveChanges();
}
```

Note: On stores with a high degree of concurrent updates it is possible that the second call to `SaveChanges()` could also result in an optimistic locking error because objects have been further modified since the initial optimistic locking failure was reported. Production code for highly concurrent environments should be written to handle this possibility.

9.7 LINQ Restrictions

9.7.1 Supported LINQ Operators

The LINQ query processor in BrightstarDB has some restrictions, but supports the most commonly used core set of LINQ query methods. The following table lists the supported query methods. Unless otherwise noted the indexed variant of LINQ query methods are not supported.

Method	Notes
Any	Supported as first result operator. Not supported as second or subsequent result operator
All	Supported as first result operator. Not supported as second or subsequent result operator
Average	Supported as first result operator. Not supported as second or subsequent result operator.
Cast	Supported for casting between Entity Framework entity types only
Contains	Supported for literal values only
Count	Supported with or without a Boolean filter expression. Supported as first result operator. Not supported as second or subsequent result operator.
Distinct	Supported for literal values. For entities `Distinct()` is supported but only to eliminate duplicates of the same Id any override of .Equals on the entity class is not used.
First	Supported with or without a Boolean filter expression
LongCount	Supported with or without a Boolean filter expression. Supported as first result operator. Not supported as second or subsequent result operator.
Max	Supported as first result operator. Not supported as second or subsequent result operator.
Min	Supported as first result operator. Not supported as second or subsequent result operator.
OfType<TResult>	Supported only if `TResult` is an Entity Framework entity type
OrderBy	
OrderBy-Descending	
Select	
SelectMany	
Single	Supported with or without a Boolean filter expression
Single-OrDefault	Supported with or without a Boolean filter expression
Skip	
Sum	Supported as first result operator. Not supported as second or subsequent result operator.
Take	
ThenBy	
ThenByDe-scending	
Where	

9.7.2 Supported Class Methods and Properties

In general, the translation of LINQ to SPARQL cannot translate methods on .NET datatypes into functionally equivalent SPARQL. However we have implemented translation of a few commonly used String, Math and DateTime methods as listed in the following table.

The return values of these methods and properties can only be used in the filtering of queries and cannot be used to modify the return value. For example you can test that `foo.Name.ToLower().Equals("somestring")`, but you cannot return the value `foo.Name.ToLower()`.

.NET function	SPARQL Equivalent
String Functions	
p0.StartsWith(string s)	STRSTARTS(p0, s)
p0.StartsWith(string s, bool ignoreCase, CultureInfo culture)	REGEX(p0, "^" + s, "i") if ignoreCase is true; STRSTARTS(p0, s) if ignoreCase is false
p0.StartsWith(string s, StringComparison comparisonOptions)	REGEX(p0, "^" + s, "i") if comparisonOptions is StringComparison.CurrentCultureIgnoreCase, StringComparison.InvariantCultureIgnoreCase or StringComparison.OrdinalIgnoreCase; STRSTARTS(p0, s) otherwise
p0.EndsWith(string s)	STRENDS(p0, s)
p0.StartsWith(string s, bool ignoreCase, CultureInfo culture)	REGEX(p0, s + "$", "i") if ignoreCase is true; STRENDS(p0, s) if ignoreCase is false
p0.StartsWith(string s, StringComparison comparisonOptions)	REGEX(p0, s + "$", "i") if comparisonOptions is StringComparison.CurrentCultureIgnoreCase, StringComparison.InvariantCultureIgnoreCase or StringComparison.OrdinalIgnoreCase; STRENDS(p0, s) otherwise
p0.Length	STRLEN(p0)
p0.Substring(int start)	SUBSTR(p0, start)
p0.Substring(int start, int len)	SUBSTR(p0, start, end)
p0.ToUpper()	UCASE(p0)
p0.ToLower()	LCASE(p0)
Date Functions	
p0.Day	DAY(p0)
p0.Hour	HOURS(p0)
p0.Minute	MINUTES(p0)
p0.Month	MONTH(p0)
p0.Second	SECONDS(p0)
p0.Year	YEAR(p0)
Math Functions	
Math.Round(decimal d)	ROUND(d)
Math.Round(double d)	ROUND(d)
Math.Floor(decimal d)	FLOOR(d)
Math.Floor(double d)	FLOOR(d)
Math.Ceiling(decimal d)	CEIL(d)
Math.Ceiling(decimal d)	CEIL(d)
Regular Expressions	
Regex.IsMatch(string p0, string expression, RegexOptions options)	REGEX(p0, expression, flags) Flags are generated from the options parameter. The supported RegexOptions are IgnoreCase, Multiline, Singleline and IgnorePatternWhitespace (or any combination of these).

The static method `Regex.IsMatch()` is supported when used to filter on a string property in a LINQ query e.g.:

```
context.Persons.Where(p => Regex.IsMatch(p.Name, "^a.*e$", RegexOptions.IgnoreCase));
```

However, please note that the regular expression options that can be used is limited to a combination of `IgnoreCase`, `Multiline`, `Singleline` and `IgnorePatternWhitespace`.

9.8 Casting Entities

One of the nicest features of RDF is its flexibility - an RDF resource can be of multiple types and can support multiple (possibly conflicting) properties according to different schemas. It allows you to record different aspects of the same thing all at a single point in the data store. In OO programming however, we tend to prefer to separate out different representations of the same thing into different classes and to use those classes to encapsulate a specific model. So there is a tension between the freedom in RDF to record anything about any resource and the need in traditional OO programming to have a set of types and properties defined at compile time.

In BrightstarDB the way we handle is is to allow you to convert an entity from one type to any other entity type at runtime. This feature is provided by the `Become<T>()` method on the entity object. Calling `Become<T>()` on an entity has two effects:

1. It adds one or more RDF type statements to the resource so that it is now recorded as being an instance of the RDF class that the entity type `T` is mapped to. When `T` inherits from a base entity type both the RDF type for `T` and the RDF type for the base type is added.
2. It returns an instance of `T` which is bound to the same underlying DataObject as the entity you call `Become<T>()` on.

This feature gives you the ability to convert and extend resources at runtime with almost no overhead. You should note that `Become<T>()` does nothing to ensure that the resource conforms to the constraints that the type `T` might imply, so your code should be written to robustly handle missing properties.

Once you call `SaveChanges()` on the context, the new type statements (and any new properties you created) are committed to the store. You will now find the object can be accessed through the context entity set for `T`.

There is also an `Unbecome<T>()` method. This method can be used to remove RDF type statements from an entity so that it no longer appears in the collection of entities of type `T` on the context. Note that this does not remove the RDF type statements for super-types of `T`, but you can explicitly do this by making further calls to `Unbecome<T>()` with the appropriate super-types.

9.9 OData

The Open Data Protocol (OData) is an open web protocol for querying data. An OData provider can be added to BrightstarDB Entity Framework projects to allow OData consumers to query the underlying data in the store.

Note: *Identifier Attributes* must exist on any BrightstarDB entity interfaces in order to be processed by an OData consumer

---

For more details on how to add a BrightstarDB OData service to your projects, read *Adding Linked Data Support* in the MVC Nerd Dinner samples chapter

9.9.1 OData Restrictions

The OData v2 protocol implemented by BrightstarDB does not support properties that contain a collection of literal values. This means that BrightstarDB entity properties that are of type `ICollection<literal type>` are not supported. Any properties of this type will not be readable via the OData service.

An OData provider connected to the BrightstarDB Entity Framework as a few restrictions on how it can be queried.

Expand

- Second degree expansions are not currently supported. e.g. `Department('5598556a-671a-44f0-b176-502da62b3b2`

Filtering

- The arithmetic filter `Mod` is not supported
- The string filter functions `int indexof(string p0, string p1)`, `string trim(string p0)` and `trim(string p0, string p1)` are not supported.
- The type filter functions `bool IsOf(type p0)` and `bool IsOf(expression p0, type p1)` are not supported.

Format

Microsoft WCF Data Services do not currently support the `$format` query option. To return OData results formatted in JSON, the accept headers can be set in the web request sent to the OData service.

9.10 SavingChanges Event

The generated EntityFramework context class exposes an event, `SavingChanges`. This event is raised during the processing of the `SaveChanges()` method before any data is committed back to the Brightstar store. The event sender is the context class itself and in the event handler you can use the `TrackedObjects` property of the context class to iterate through all entities that the context class has retrieved from the BrightstarDB store. Entities expose an `IsModified` property which can be used to determine if the entity has been newly created or locally modified. The sample code below uses this to update a `Created` and `LastModified` timestamp on any entity that implements the `ITrackable` interface.:

```
private static void UpdateTrackables(object sender, EventArgs e)
{
  // This method is invoked by the context.
  // The sender object is the context itself
  var context = sender as MyEntityContext;


  // Iterate through just the tracked objects that implement the ITrackable interface
  foreach(var t in context.TrackedObjects
                  .Where(x=>x is ITrackable && x.IsModified)
                  .Cast<ITrackable>())
  {
    // If the Created property is not yet set, it will have DateTime.MinValue as its defaulft value
    // We can use this fact to determine if the Created property needs setting.
    if (t.Created == DateTime.MinValue) t.Created = DateTime.Now;

    // The LastModified property should always be updated
    t.LastModified = DateTime.Now;
  }
}
```

Note: The source code for this example can be found in [INSTALLDIR]\Samples\EntityFramework\EntityFrameworkSamples.sln

9.11 INotifyPropertyChanged and INotifyCollectionChanged Support

The classes generated by the Entity Framework provide support for tracking local changes. All generated entity classes implement the System.ComponentModel.INotifyPropertyChanged interface and fire a notification event any time a property with a single value is modified. All collections exposed by the generated classes implement the System.Collections.Specialized.INotifyCollectionChanged interface and fire a notification when an item is added to or removed from the collection or when the collection is reset.

There are a few points to note about using these features with the Entity Framework:

Firstly, although the generated classes implement the `INotifyPropertyChanged` interface, your code will typically use the interfaces. To attach a handler to the `PropertyChanged` event, you need an instance of `INotifyPropertyChanged` in your code. There are two ways to achieve this - either by casting or by adding `INotifyPropertyChanged` to your entity interface. If casting you will need to write code like this:

```
// Get an entity to listen to
var person = _context.Persons.Where(x=>x.Name.Equals("Fred")).FirstOrDefault();

// Attach the NotifyPropertyChanged event handler
(person as INotifyPropertyChanged).PropertyChanged += HandlePropertyChanged;
```

Alternatively it can be easier to simply add the `INotifyPropertyChanged` interface to your entity interface like this:

```
[Entity]
public interface IPerson : INotifyPropertyChanged
{
  // Property definitions go here
}
```

This enables you to then write code without the cast:

```
// Get an entity to listen to
var person = _context.Persons.Where(x=>x.Name.Equals("Fred")).FirstOrDefault();

// Attach the NotifyPropertyChanged event handler
person.PropertyChanged += HandlePropertyChanged;
```

When tracking changes to collections you should also be aware that the dynamically loaded nature of these collections means that sometimes it is not possible for the change tracking code to provide you with the object that was removed from a collection. This will typically happen when you have a collection one one entity that is the inverse of a collection or property on another entity. Updating the collection at one end will fire the CollectionChanged event on the inverse collection, but if the inverse collection is not yet loaded, the event will be raised as a NotifyCollectionChangedAction.Reset type event, rather than a NotifyCollectionChangedAction.Remove event. This is done to avoid the overhead of retrieving the removed object from the data store just for the purpose of raising the notification event.

Finally, please note that event handlers are attached only to the local entity objects, the handlers are not persisted when the context changes are saved and are not available to any new context's you create - these handlers are intended only for tracking changes made locally to properties in the context before a `SaveChanges()` is invoked. The properties are also useful for data binding in applications where you want the user interface to update as the properties are modified.

9.12 Graph Targeting

The Entity Framwork supports updating a specific named graph in the BrightstarDB store. The graph to be updated is specified when creating the context object using the following optional parameters in the context constructor:

- `updateGraph` : The identifier of the graph that new statements will be added to. Defaults to the BrightstarDB default graph (`http://www.brightstardb.com/.well-known/model/defaultgraph`)
- `defaultDataSet` : The identifier of the graphs that statements will be retrieved from. Defaults to all graphs in the store.
- `versionGraph` : The identifier of the graph that contains version information for optimistic locking. Defaults to the same graph as `updateGraph`.

Please refer to the section *Default Data Set* for more information about the default data set and its relationship to the `defaultDataSet`, `updateGraph`, and `versionGraph` parameters.

To create a context that reads properties from the default graph and adds properties to a specific graph (e.g. for recording the results of inferences), use the following:

```
// Set storeName, prefixes and inferredGraphUri here
    var context = new MyEntityContext(
      connectionString,
      enableOptimisticLocking,
      "http://example.org/graphs/graphToUpdate",
      new string[] { Constants.DefaultGraphUri },
      Constants.DefaultGraphUri);
```

Note: Note that you need to be careful when using optimistic locking to ensure that you are consistent about which graph manages the version information. We recommend that you either use the BrightstarDB default graph (as shown in the example above) or use another named graph seperate from the graphs that store the rest of the data (and define a constant for that graph URI).

9.12.1 LINQ and Graph Targeting

For LINQ queries to work, the triple that assigns the entity type must be in one of the graphs in the default data set or in the graph to be updated. This makes the Entity Framework a bit more difficult to use across multiple graphs. When writing an application that will regularly deal with different named graphs you may want to consider using the *Data Object Layer API* and SPARQL or the low-level RDF API for update operations.

CHAPTER 10

Entity Framework Samples

The following samples provide detailed information on how to build applications using BrightstarDB. If there are classes of applications for which you would like to see other tutorials please let us know.

10.1 Tweetbox

Note: The source code for this example can be found in [INSTALLDIR]\Samples\EntityFramework\EntityFrameworkSamples.sln

10.1.1 Overview

The TweetBox sample is a simple console application that shows the speed in which BrightstarDB can load content. The aim is not to create a Twitter style application, but to show how objects with various relationships to one another are loading quickly, in a structure that will be familiar to developers.

The model consists of 3 simple interfaces: `IUser`, `ITweet` and `IHashTag`. The relationships between the interfaces mimic the structure on Twitter, in that Users have a many to many relationship with other Users (or followers), and have a one to many relationship with Tweets. The tweets have a many to many relationship with Hashtags, as a Tweet can have zero or more Hashtags, and a Hashtag may appear in more than one Tweet.

10.1.2 The Interfaces

IUser

The IUser interface represents a user on Twitter, with simple string properties for the username, bio (profile text) and date of registration. The 'Following' property shows the list of users that this user follows, the other end of this relationship is shown in the 'Followers' property, this is marked with the 'InverseProperty' attribute to tell BrightstarDB that Followers is the other end of the Following relationship. The final property is a list of tweets that the user has authored, this is the other end of the relationship from the ITweet interface (described below):

```
[Entity]
public interface IUser
{
    string Id { get; }
    string Username { get; set; }
    string Bio { get; set; }
    DateTime DateRegistered { get; set; }
    ICollection<IUser> Following { get; set; }
    [InverseProperty("Following")]
```

```
    ICollection<IUser> Followers { get; set; }
    [InverseProperty("Author")]
    ICollection<ITweet> Tweets { get; set; }
}
```

ITweet

The ITweet interface represents a tweet on twitter, and has simple properties for the tweet content and the date and time it was published. The Tweet has an IUser property ('Author') to relate it to the user who wrote it (the other end of this relationship is described above). ITweet also contains a collection of Hashtags that appear in the tweet (described below):

```
[Entity]
public interface ITweet
{
    string Id { get; }
    string Content { get; set; }
    DateTime DatePublished { get; set; }
    IUser Author { get; set; }
    ICollection<IHashTag> HashTags { get; set; }
}
```

IHashTag

A hashtag is a keyword that is contained in a tweet. The same hashtag may appear in more than one tweet, and so the collection of Tweets is marked with the 'InverseProperty' attribute to show that it is the other end of the collection of HashTags in the ITweet interface:

```
[Entity]
public interface IHashTag
{
    string Id { get; }
    string Value { get; set; }
    [InverseProperty("HashTags")]
    ICollection<ITweet> Tweets { get; set; }
}
```

10.1.3 Initialising the BrightstarDB Context

The BrightstarDB context can be initialised using a connection string:

```
var connectionString = "Type=rest;endpoint=http://localhost:8090/brightstar;StoreName=Tweetbox";
var context = new TweetBoxContext(connectionString);
```

If you have added the connection string into the Config file:

```
<add key="BrightstarDB.ConnectionString" value="Type=rest;endpoint=http://localhost:8090/brightstar;S
```

then you can initialise the content with a simple:

```
var context = new TweetBoxContext();
```

For more information about connection strings, please read the *"Connection Strings"* topic.

10.1.4 Creating a new User entity

Method 1:

```
var jo = context.Users.Create();
jo.Username = "JoBloggs79";
jo.Bio = "A short sentence about this user";
jo.DateRegistered = DateTime.Now;
context.SaveChanges();
```

Method 2:

```
var jo = new User {
                Username = "JoBloggs79",
                Bio = "A short sentence about this user",
                DateRegistered = DateTime.Now
            };
context.Users.Add(jo);
context.SaveChanges();
```

10.1.5 Relationships between entities

The following code snippets show the creation of relationships between entities by simply setting properties.

Users to Users:

```
var trevor = context.Users.Create();
trevor.Username = "TrevorSims82";
trevor.Bio = "A short sentence about this user";
trevor.DateRegistered = DateTime.Now;
trevor.Following.Add(jo);
context.SaveChanges();
```

Tweets to Tweeter:

```
var tweet = context.Tweets.Create();
tweet.Content = "My first tweet";
tweet.DatePublished = DateTime.Now;
tweet.Tweeter = trevor;
context.SaveChanges();
```

Tweets to HashTags::

```
var nosql = context.HashTags.Where(ht => ht.Value.Equals("nosql").FirstOrDefault();
if (nosql == null)
{
    nosql = context.HashTags.Create();
    nosql.Value = "nosql";
}
var  brightstardb = context.HashTags.Where(ht => ht.Value.Equals("brightstardb").FirstOrDefault();
if (brightstardb == null)
{
    brightstardb = context.HashTags.Create();
    brightstardb.Value = "brightstardb";
}
var tweet2 = context.Tweets.Create();
tweet.Content = "New fast, scalable NoSQL database for the .NET platform";
tweet.HashTags.Add(nosql);
tweet.HashTags.Add(brightstar);
tweet.DatePublished = DateTime.Now;
tweet.Tweeter = trevor;
context.SaveChanges();
```

10.1.6 Fast creation, persistence and indexing of data

In order to show the speed at which objects can be created, persisted and index in BrightstarDB, the console application creates 100 users, each with 500 tweets. Each of those tweets has 2 hashtags (chosen from a set of 10,000 hash tags).

1. Creates 100 users
2. Creates 10,000 hashtags
3. Saves the users and hashtags to the database
4. Loops through the existing users and adds followers and tweets (each tweet has 2 random hashtags)
5. Saves the changes back to the store
6. Writes out the time taken to the console

10.2 MVC Nerd Dinner

Note: The source code for this example can be found in the solution [INSTALLDIR]\Samples\NerdDinner\BrightstarDB.Samples.NerdDinner.sln

To demonstrate the ease of using BrightstarDB with ASP.NET MVC, we will use the well-known "Nerd Dinner" tutorial used by .NET Developers when they first learn MVC. We won't recreate the full Nerd Dinner application, but just a portion of it, to show how to use BrightstarDB for code-first data persistence, and show how it not only matches the ease of creating applications from scratch, but surpasses Entity Framework by introducing pain free model changes (more on that later). The Brightstar.NerdDinner sample application shows a simple model layer, using ASP.NET MVC4 for the CRUD application and BrightstarDB for data storage. In later sections we will extend this basic functionality with support for linked data in the form of both OData and SPARQL query support and we will show how to use BrightstarDB as the basis for a .NET custom membership and role provider.

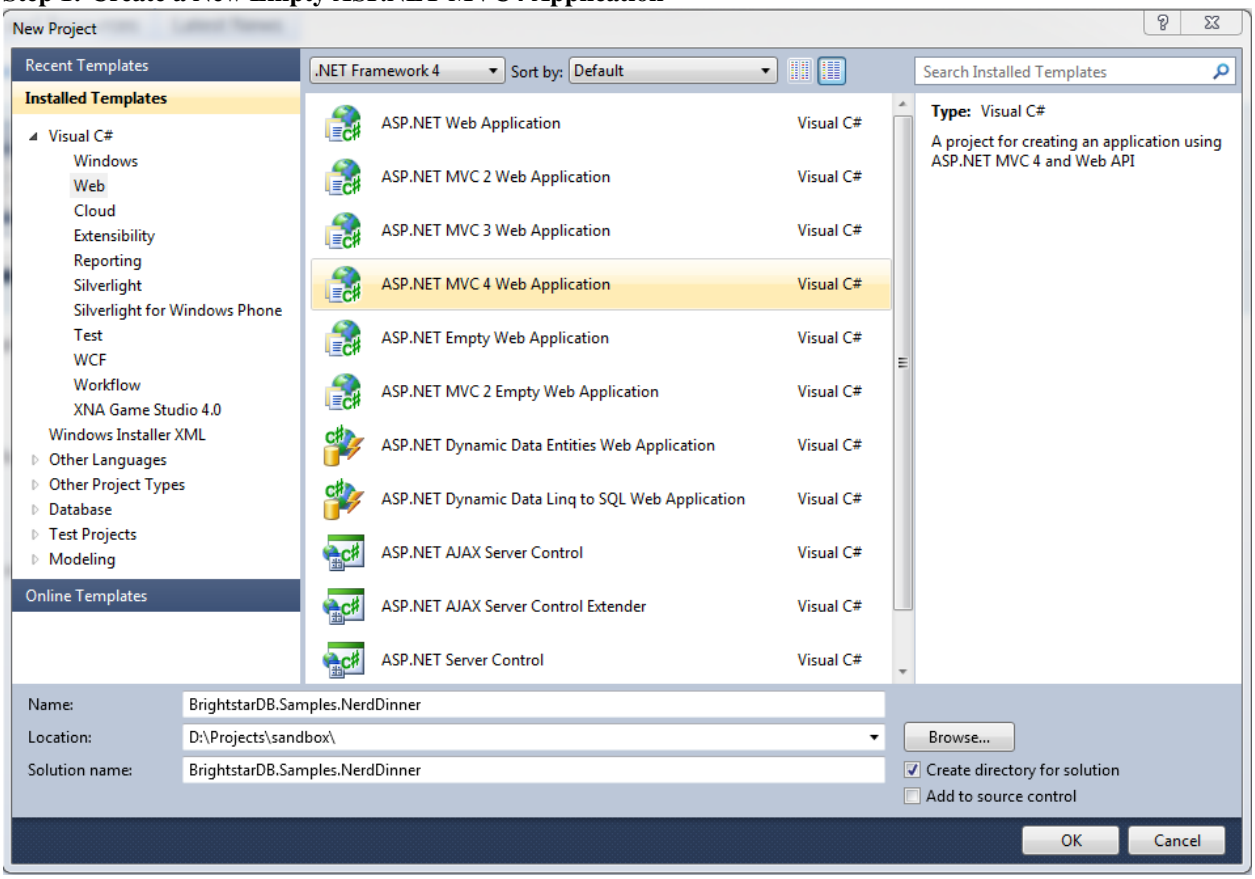
This tutorial is quite long, but is broken up into a number of separate sections each of which you can follow along with in code, or you can refer to the complete sample application which can be found in [INSTALLDIR]\Samples\NerdDinner.

- *Creating The Basic Data Model* - creates the initial application and code-first data model
- *Creating MVC Controllers and Views* - shows how easy it is to use this model with ASP.NET MVC4 to create web interfaces for create, update and delete (CRUD) operations.
- *Applying Model Changes* - shows how BrightstarDB handles changes to the code-first data model without data loss.
- *Adding A Custom Membership Provider* - describes how to build a ASP.NET custom membership provider that uses BrightstarDB to manage user account information.
- *Adding A Custom Role Provider* - builds on the custom membership provider to enable users to be assigned different roles and levels of access
- *Adding Linked Data Support* - extends the web application to provide a SPARQL and an ODATA query endpoint
- *Consuming OData In PowerPivot* - shows one way in which the OData endpoint can be used - enabling data to be retrieved into Excel.

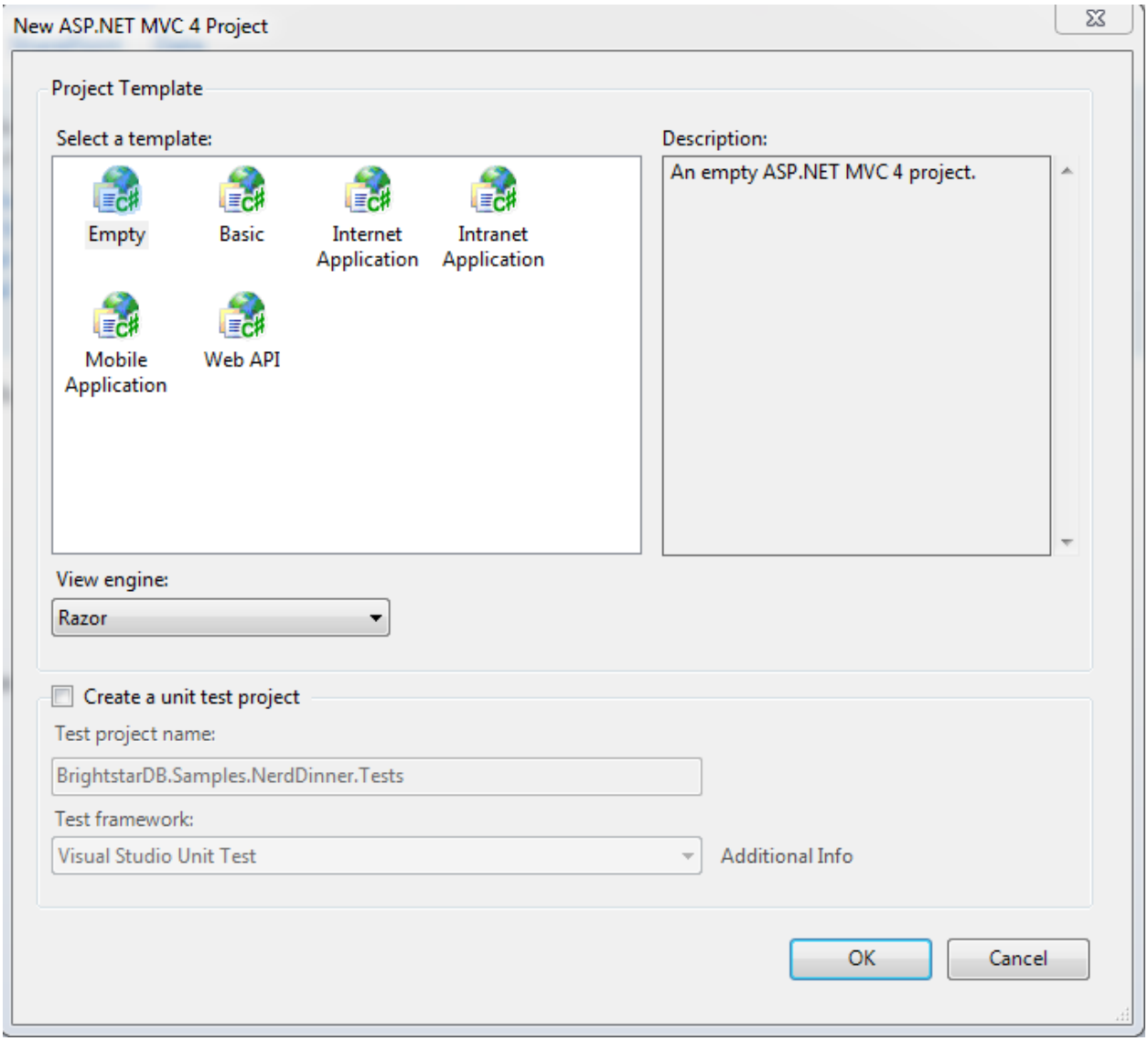
10.2.1 Creating The Basic Data Model

Creating the ASP.NET MVC4 Application.

Step 1: Create a New Empty ASP.NET MVC4 Application



Choose "ASP.NET MVC 4 Web Application" from the list of project types in Visual Studio. If you do not already have MVC 4 installed you can download it from http://www.asp.net/mvc/mvc4. You must also install the "Visual Web Developer" feature in Visual Studio to be able to open and work with MVC projects. Choose a name for your application (we are using BrightstarDB.Samples.NerdDinner), and then click OK. In the next dialog box, select "Empty" for the template type, this mean that the project will not be pre-filled with any default controllers, models or views so we can show every step in building the application. Choose "Razor" as the View Engine. Leave the "Create a unit test project" box unchecked, as for the purposes of this example project it is not needed.



Step 2: Add references to BrightstarDB

Add a reference in your project to the BrightstarDB DLL from the BrightstarDB SDK.

Step 3: Add a connection string to your BrightstarDB location

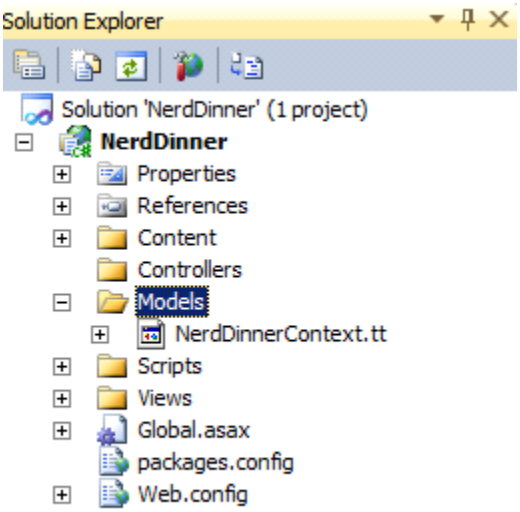
Open the web.config file in the root directory your new project, and add a connection string to the location of your BrightstarDB store. There is no setup required - you can name a store that does not exist and it will be created the first time that you try to connect to it from the application. The only thing you will need to ensure is that if you are using an HTTP, TCP or Named Pipe connection, the BrightstarDB service must be running:

```
<appSettings>
  ...
  <add key="BrightstarDB.ConnectionString"
       value="Type=rest;endpoint=http://localhost:8090/brightstar;StoreName=NerdDinner" />
  ...
</appSettings>
```

For more information about connection strings, please read the *"Connection Strings"* topic.

Step 4: Add the Brightstar Entity Context into your project

Select **Add > New Item** on the Models folder, and select **Brightstar Entity Context** from the Data category. Rename it to NerdDinnerContext.tt



Step 5: Creating the data model interfaces

BrightstarDB data models are defined by a number of standard .NET interfaces with certain attributes set. The NerdDinner model is very simple (especially for this tutorial) and only consists of a set of "Dinners" that refer to specific events that people can attend, and also a set of "RSVP"s that are used to track a person's interest in attending a dinner.

We create the two interfaces as shown below in the Models folder of our project.

IDinner.cs:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using BrightstarDB.EntityFramework;

namespace BrightstarDB.Samples.NerdDinner.Models
{
    [Entity]
    public interface IDinner
    {
        [Identifier("http://nerddinner.com/dinners/")]
        string Id { get; }


        [Required(ErrorMessage = "Please provide a title for the dinner")]
        string Title { get; set; }


        string Description { get; set; }


        [Display(Name = "Event Date")]
        [DataType(DataType.DateTime)]
        DateTime EventDate { get; set; }


        [Required(ErrorMessage = "The event must have an address.")]
```

```
        string Address { get; set; }


        [Required(ErrorMessage = "Please enter the name of the host of this event")]
        [Display(Name = "Host")]
        string HostedBy { get; set; }


        ICollection<IRSVP> RSVPs { get; set; }
    }
}
```

IRSVP.cs::

```
using System.ComponentModel.DataAnnotations;
using BrightstarDB.EntityFramework;

namespace BrightstarDB.Samples.NerdDinner.Models
{
    [Entity]
    public interface IRSVP
    {
        [Identifier("http://nerddinner.com/rsvps/")]
        string Id { get; }


        [Display(Name = "Email Address")]
        [Required(ErrorMessage = "Email address is required")]
        string AttendeeEmail { get; set; }


        [InverseProperty("RSVPs")]
        IDinner Dinner { get; set; }
    }
}
```

By default, BrightstarDB identifier properties are automatically generated URIs that are automatically. In order to work with simpler values for our entity Ids we decorate the Id property with an identifier attribute. This adds a prefix for BrightstarDB to use when generating and querying the entity identifiers and ensures that the actual value we get in the Id property is just the part of the URI that follows the prefix, which will be a simple GUID string.

In the IRSVP interface, we add an InverseProperty attribute to the Dinner property, and set it to the name of the .NET property on the referencing type ("RSVPs"). This shows that these two properties reflect different sides of the same association. In this case the association is a one-to-many relationship (one dinner can have many RSVPs), but BrightstarDB also supports many-to-many and many-to-one relationships using the same mechanism.

We can also add other attributes such as those from the `System.ComponentModel.DataAnnotations` namespace to provide additional hints for the MVC framework such as marking a property as required, providing an alternative display name for forms or specifying the way in which a property should be rendered. These additional attributes are automatically added to the classes generated by the BrightstarDB Entity Framework. For more information about BrightstarDB Entity Framework attributes and passing through additional attributes, please refer to the *Annotations* section of the *Entity Framework* documentation.

Step 6: Creating a context class to handle database persistence

Right click on the Brightstar Entity Context and select **Run Custom Tool**. This runs the text templating tool that updates the .cs file contained within the .tt file with the most up to date persistence code needed for your interfaces. Any time you modify the interfaces that define your data model, you should re-run the text template to regenerate the context code.

We now have the basic data model for our application completed and have generated the code for creating persistent entities that match our data model and storing them in BrightstarDB. In the next section we will see how to use this data model and context in creating screens in our MVC application.

10.2.2 Creating MVC Controllers And Views

In the previous section we created the skeleton MVC application and added to it a BrightstarDB data model for dinners and RSVPs. In this section we will start to flesh out the MVC application with some screens for data entry and display.

Create the Home Controller

Right click on the controller folder and select "Add > Controller". Name it "HomeController" and select "Controller with empty Read/Write Actions". This adds a Controller class to the folder, with empty actions for Index(), Details(), Create(), Edit() and Delete(). This will be the main controller for all our CRUD operations.

The basic MVC4 template for these operations makes a couple of assumptions that we need to correct. Firstly, the id parameter passed in to various operations is assumed to be an int; however our BrightstarDB entities use a string value for their Id, so we must change the int id parameters to string id on the Details, Edit and Delete actions. Secondly, by default the HttpPost actions for the Create and Edit actions accept FormCollection parameters, but because we have a data model available it is easier to work with the entity class, so we will change these methods to accept our data model's classes as parameters rather than FormCollection and let the MVC framework handle the data binding for us - for the Delete action it does not really matter as we are not concerned with the value posted back by that action in this sample application.

Before we start editing the Actions, we add the following line to the HomeController class:

```
public class HomeController : Controller
{
        NerdDinnerContext _nerdDinners = new NerdDinnerContext();
...
}
```

This ensures that any action invoked on the controller can access the BrightstarDB entity framework context.

Index

This view will show a list of all dinners in the system, it's a simple case of using LINQ to return a list of all dinners::

```
public ActionResult Index()
{
    var dinners = from d in _nerdDinners.Dinners
                  select d;
    return View(dinners.ToList());
}
```

Details

This view shows all the details of a particular dinner, so we use LINQ again to query the store for a dinner with a particular Id. Note that we have changed the type of the id parameter from int to string. The LINQ query here uses FirstOrDefault() which means that if there is no dinner with the specified ID, we will get a null value returned by the query. If that is the case, we return the user to a "404" view to display a "Not found" message in the browser, otherwise we return the default Details view.:

```
public ActionResult Details(string id)
{
    var dinner = _nerdDinners.Dinners.FirstOrDefault(d => d.Id.Equals(id));
    return dinner == null ? View("404") : View(dinner);
}
```

Edit

The controller has two methods to deal with the Edit action, the first handles a get request and is similar to the Details method above, but the view loads the property values into a form ready to be edited. As with the previous method, the type of the id parameter has been changed to string:

```
public ActionResult Edit(string id)
{
    var dinner = _nerdDinners.Dinners.Where(d => d.Id.Equals(id)).FirstOrDefault();
    return dinner == null ? View("404") : View(dinner);
}
```

The method that accept the HttpPost that is sent back after a user clicks "Save" on the view, deals with updating the property values in the store. Note that rather than receiving the id and FormsCollection parameters provided by the default scaffolding, we change this method to receive a Dinner object. The Dinner class is generated by the BrightstarDB Entity Framework from our IDinner data model interface and the MVC framework can automatically data bind the values in the edit form to a new Dinner instance before invoking our Edit method. This automatic data binding makes the code to save the edited dinner much simpler and shorter - we just need to attach the Dinner object to the _nerdDinners context and then call SaveChanges() on the context to persist the updated entity:

```
[HttpPost]
public ActionResult Edit(Dinner dinner)
{
    if(ModelState.IsValid)
    {
        dinner.Context = _nerdDinners;
        _nerdDinners.SaveChanges();
        return RedirectToAction("Index");
    }
    return View();
}
```

Create

Like the Edit method, Create displays a form on the initial view, and then accepts the HttpPost that gets sent back after a user clicks "Save". To make things slight easier for the user we are pre-filling the "EventDate" property with a date one week in the future:

```
public ActionResult Create()
{
   var dinner = new Dinner {EventDate = DateTime.Now.AddDays(7)};
   return View(dinner);
}
```

When the user has entered the rest of the dinner details, we add the Dinner object to the Dinners collection in the context and then call SaveChanges():

```
[HttpPost]
public ActionResult Create(Dinner dinner)
{
    if(ModelState.IsValid)
    {
        _nerdDinners.Dinners.Add(dinner);
        _nerdDinners.SaveChanges();
        return RedirectToAction("Index");
    }
    return View();
}
```

Delete

The first stage of the Delete method displays the details of the dinner about to be deleted to the user for confirmation:

```
public ActionResult Delete(string id)
{
    var dinner = _nerdDinners.Dinners.Where(d => d.Id.Equals(id)).FirstOrDefault();
    return dinner == null ? View("404") : View(dinner);
}
```

When the user has confirmed the object is Deleted from the store:

```
[HttpPost, ActionName("Delete")]
public ActionResult DeleteConfirmed(string id, FormCollection collection)
{
    var dinner = _nerdDinners.Dinners.FirstOrDefault(d => d.Id.Equals(id));
    if (dinner != null)
    {
        _nerdDinners.DeleteObject(dinner);
        _nerdDinners.SaveChanges();
    }
    return RedirectToAction("Index");
}
```

Adding views

Now that we have filled in the logic for the actions, we can proceed to create the necessary views. These views will make use of the Microsoft JQuery Unobtrusive Validation nuget package. You can install this package through the GUI Nuget package manager or using the NuGet console command:

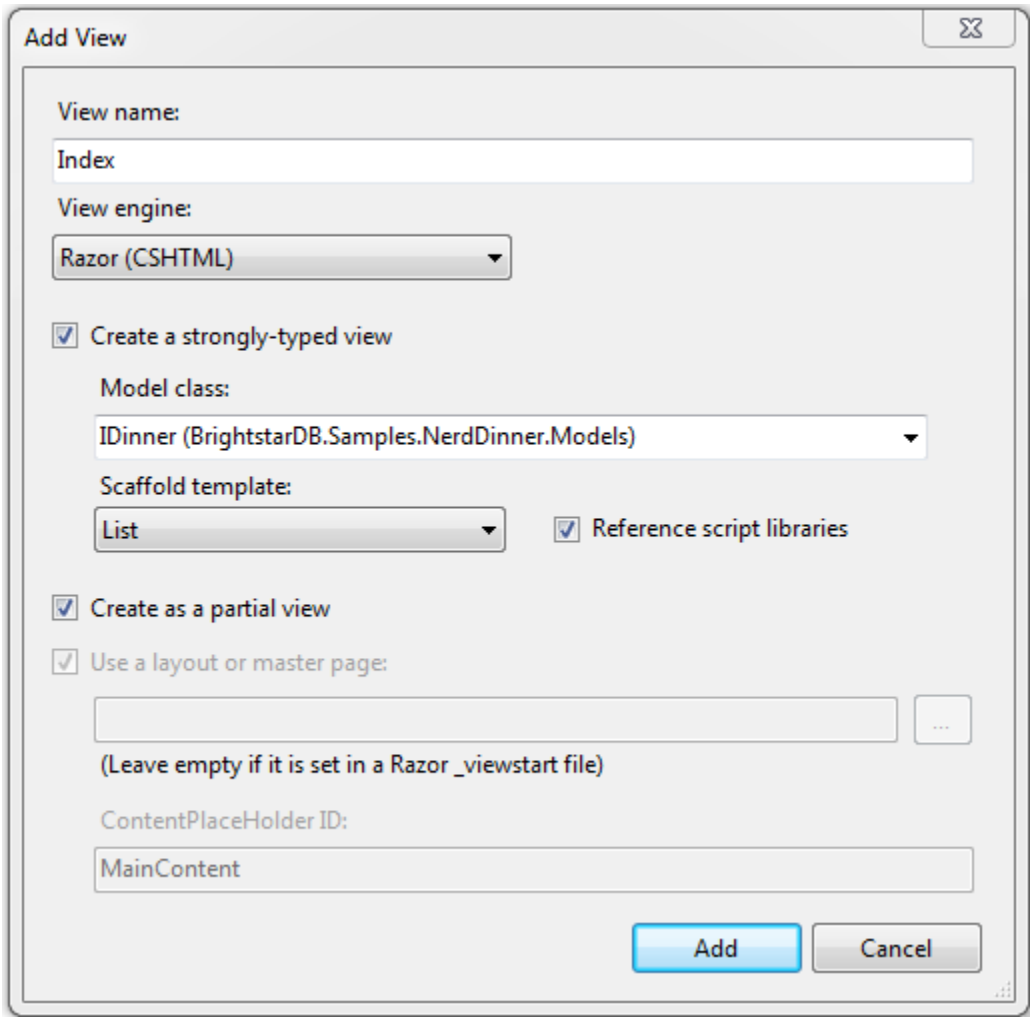
```
PM> install-package Microsoft.jQuery.Unobtrusive.Validation
```

This will also install the jQuery and jQuery.Validation packages that are dependencies.

Before creating specific views, we can create a common look and feel for these views by creating a _ViewStart.cshtml and a shared _Layout.cshtml. This approach also makes the Razor for the individual views simpler and easier to manage. Please refer to the sample solution for the content of these files and the 404 view that is displayed when a URL specifies an ID that cannot be resolved.

All of the views for the Home controller need to go in the Home folder under the Views folder - if it does not exist yet, create the Home folder within the Views folder of the MVC solution. Then, to Add a view, right click on the “Home” folder within “Views” and select “Add > View”. For each view we create a strongly-typed view with the appropriate scaffold template and create it as a partial view.

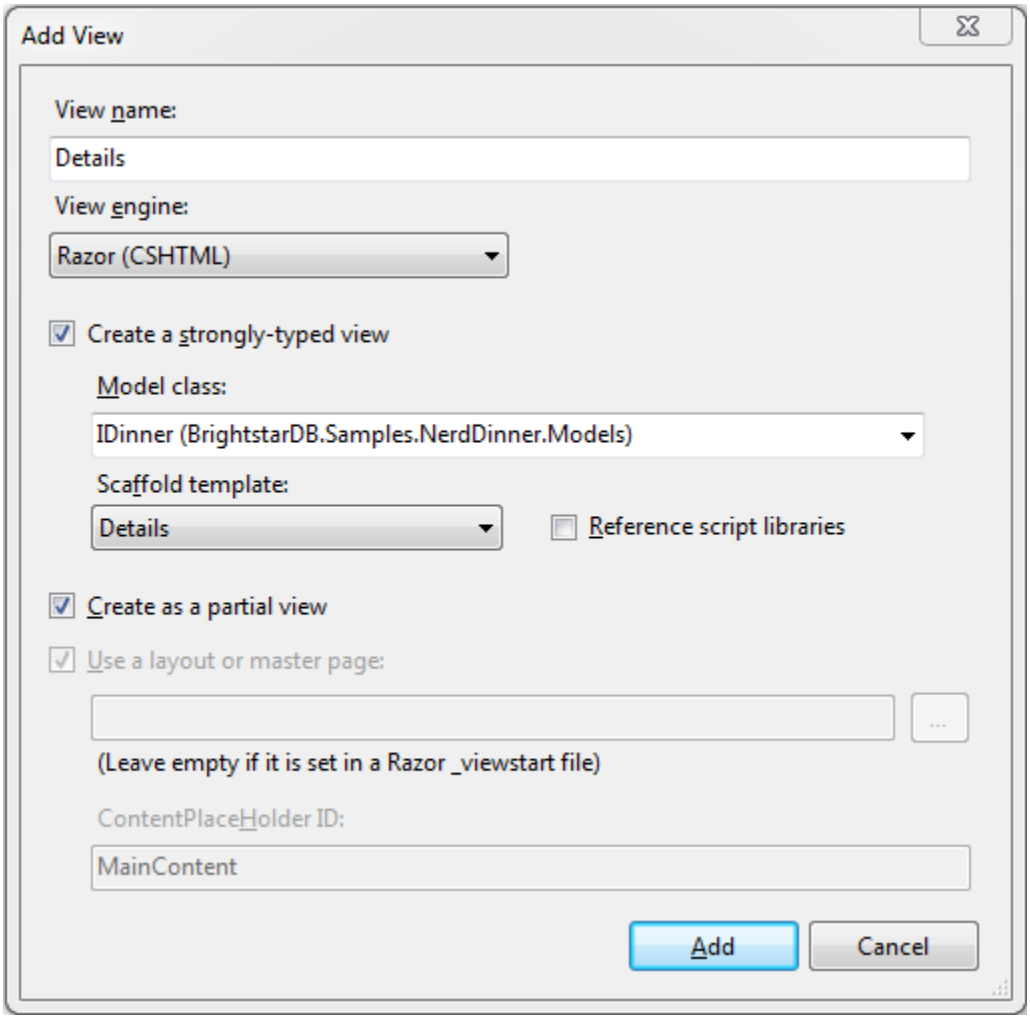
The Index View uses a List template, and the IDinner model:



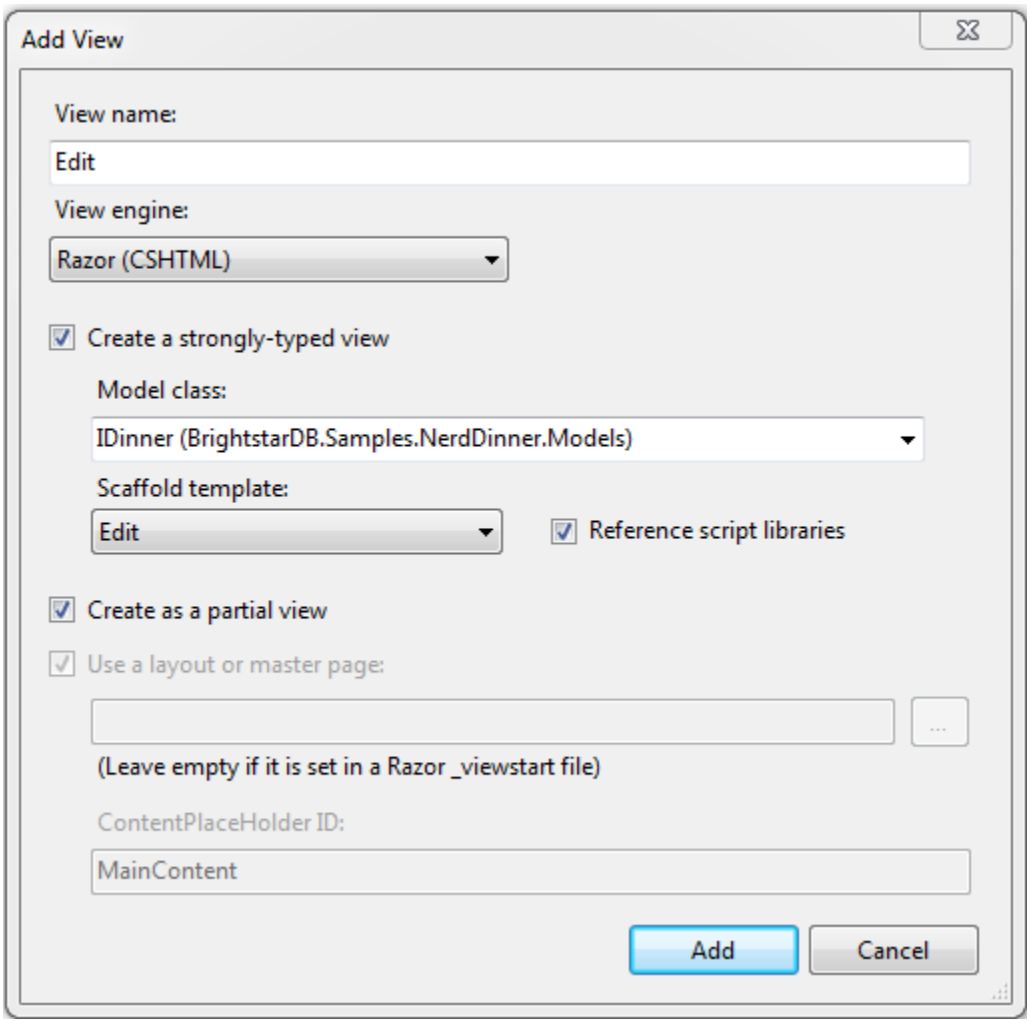
Note: If the IDinner type is not displayed in the "Model class" drop-down list, this may be because Visual Studio is not aware of the type yet - to fix this, you must save and compile the solution before trying to add views.

Note: If you get an error from Visual Studio when trying to add this view, please see this blog post for a possible solution.

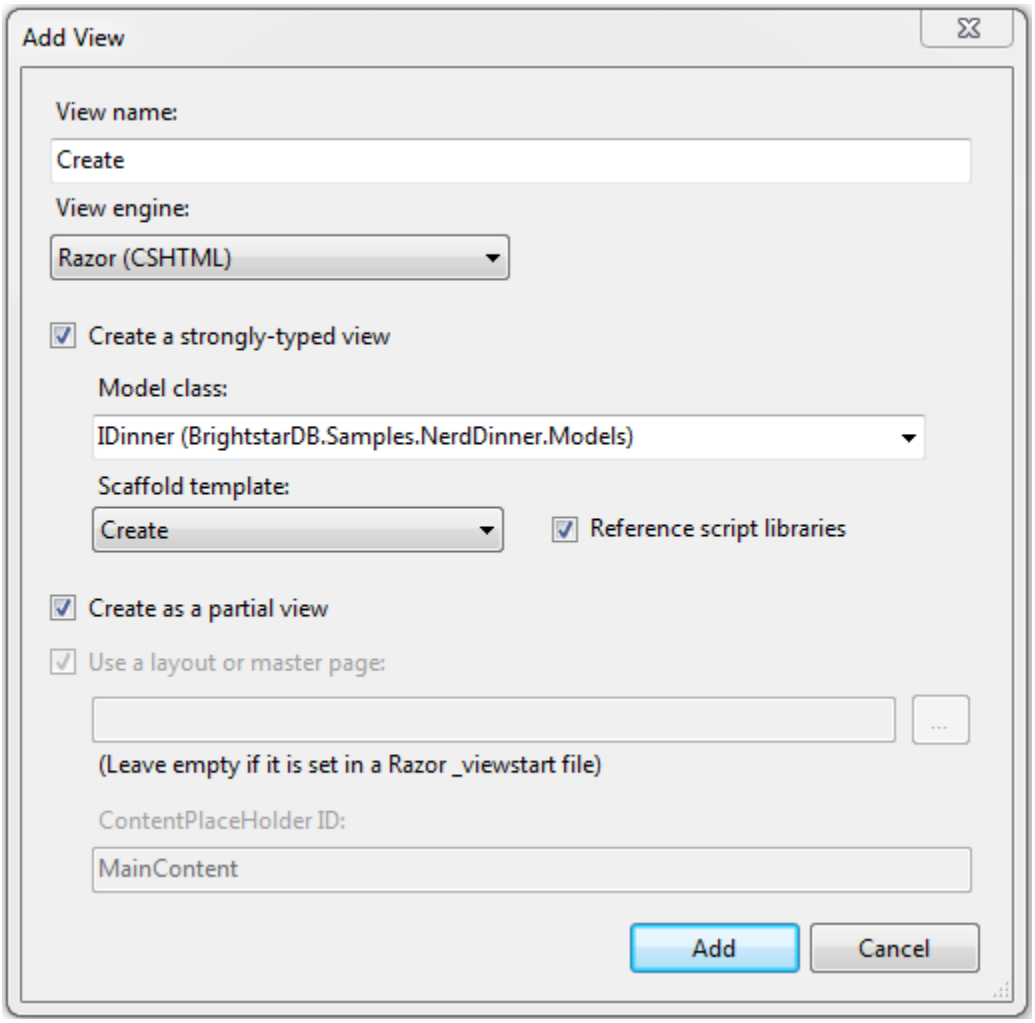
The Details View uses the Details template:



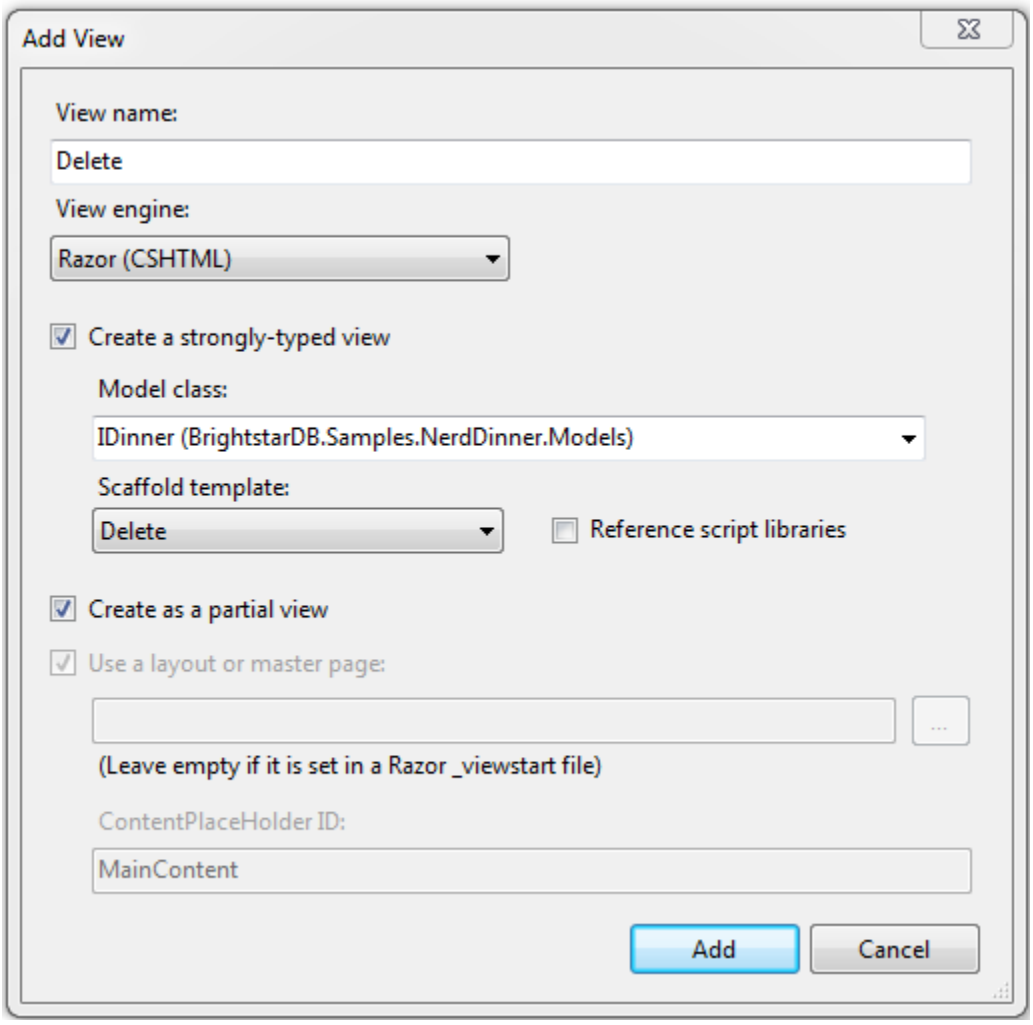
The Edit View uses the Edit template and also includes script library references. You may want to modify the reference to the jquery-1.7.1.min.js script from the generated template to point to the version of jQuery installed by the validation NuGet package (this is jquery-1.4.4.min.js at the time of writing).



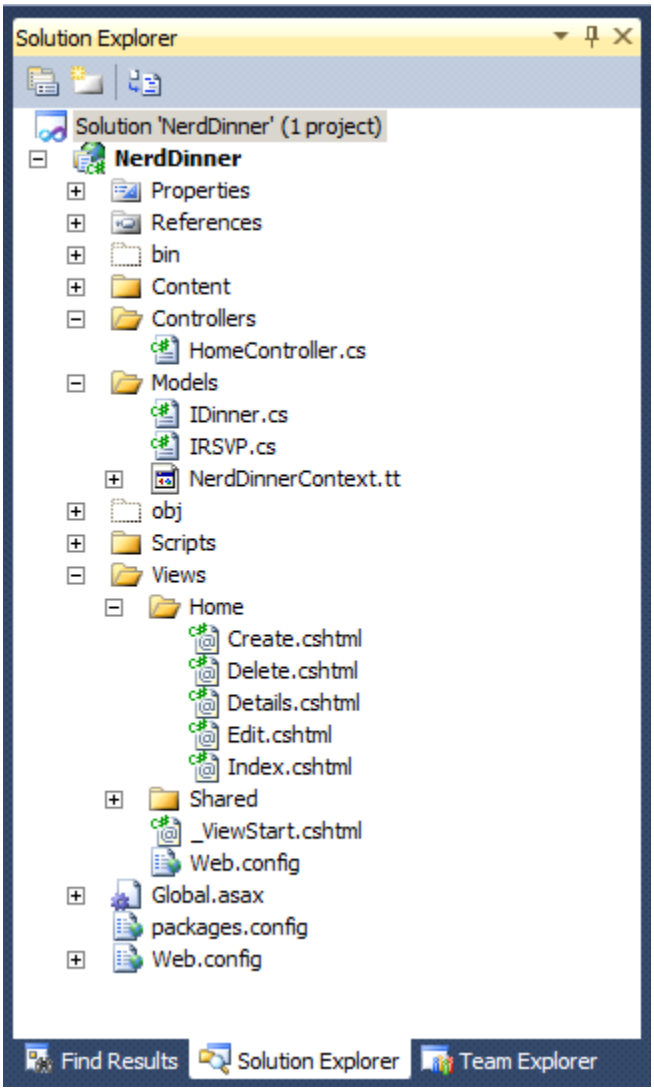
The Create View uses the Create template and again includes the script library references, which you should modify in the same way as you did for the Edit view.



The Delete view uses the Delete template:

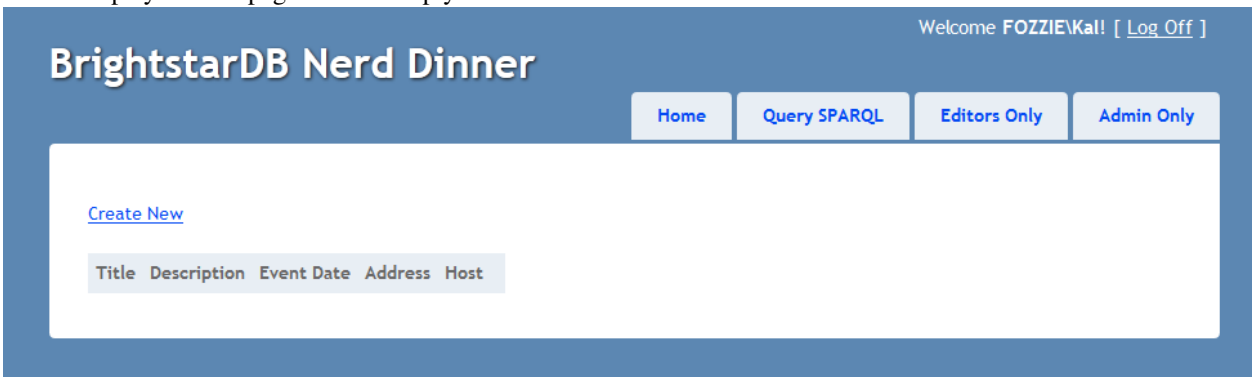


Adding strongly typed views in this way pre-populates the HTML with tables, forms and text where needed to display information and gather data from the user.



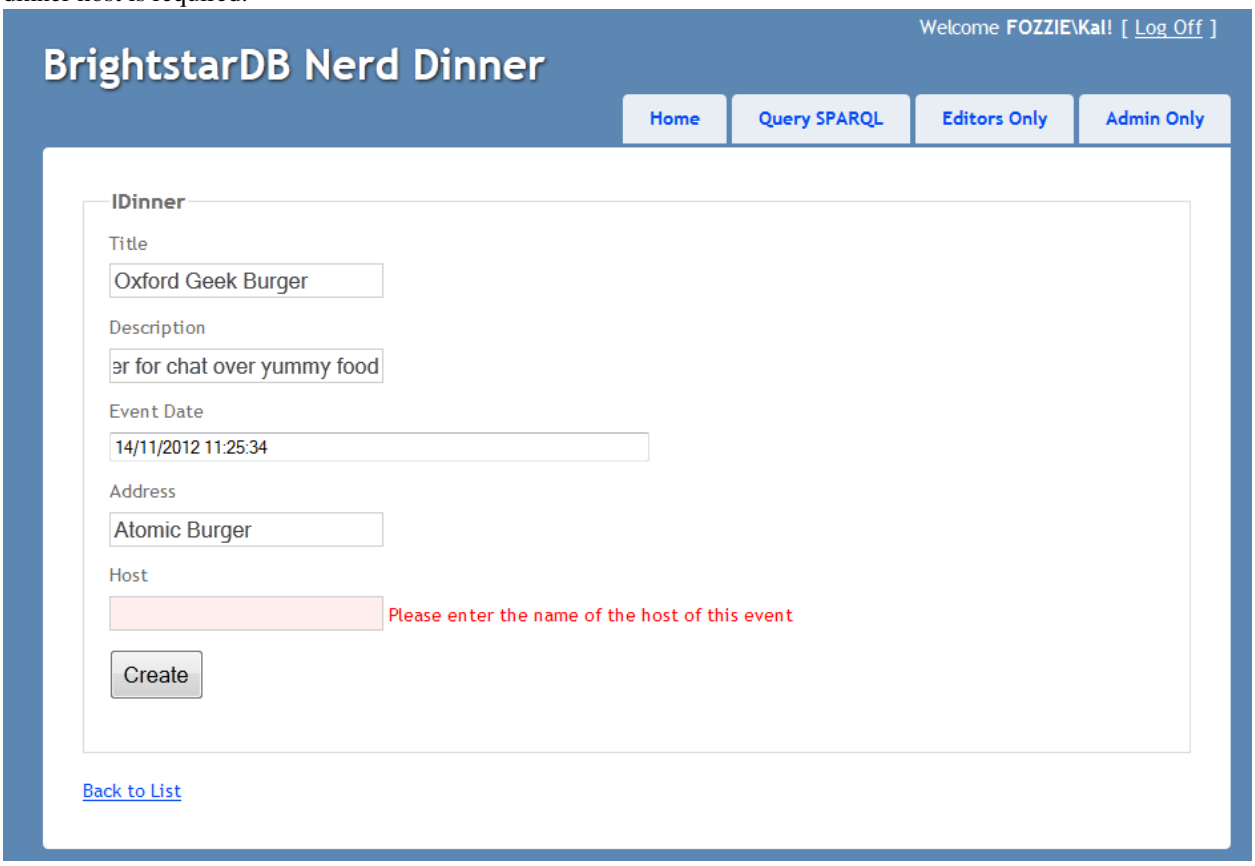
Review Site

We have now implemented all of the code we need to write within our Controller and Views to implement the Dinner listing and Dinner creation functionality within our web application. Running the web application for the first time should display a home page with an empty list of dinners:

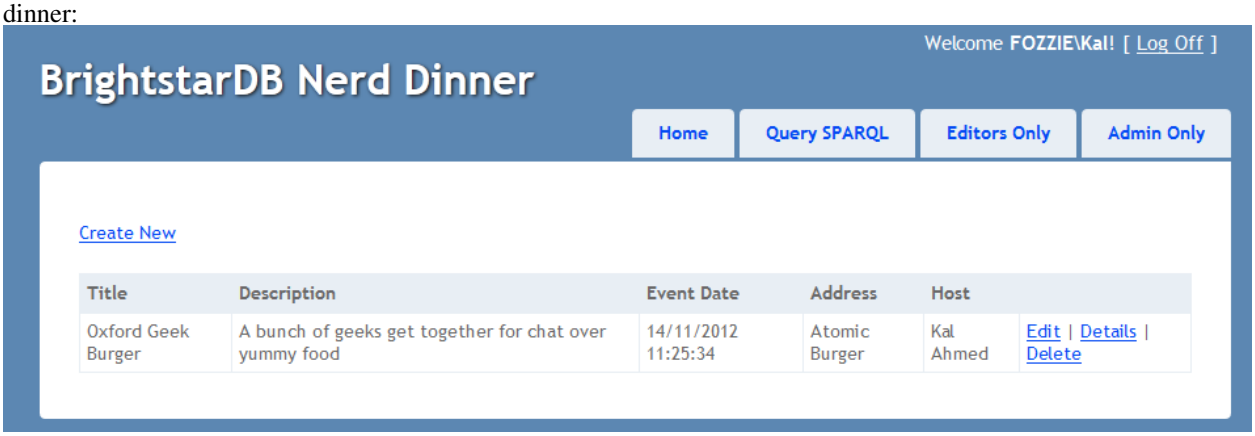


Clicking on the Create New link takes you to the form for entering the details for a new dinner. Note that this form

supports some basic validation through the annotation attributes we added to the model. For example the name of the dinner host is required:



Once a dinner is created it shows up in the list on the home page from where you can view details, edit or delete the dinner:



However, we still have no way of registering attendees! To do that we need to add another action that will allow us to create an RSVP and attach it to a dinner.

Create the AddAttendee Action

Like the Create, Edit and Delete actions, AddAttendee will be an action with two parts to it. The first part of the action, invoked by an HTTP GET (a normal link) will display a form in which the user can enter the email address they want to use for the RSVP. The second part of the action will handle the HTTP POST generated by that form when the user

submits it - this part will use the details in the form to create a new RSVP entity and connect it to the correct event. The action will be created in the Home controller, so new methods will be added to HomeController.cs.

This is the code for the first part of AddAttendee action - it is a similar pattern that we have seen else where. We retrieve the dinner entity by its ID and pass it through to the view so we can show the user some details about the dinner they have chosen to attend:

```
public ActionResult AddAttendee(string id)
{
    var dinner = _nerdDinners.Dinners.FirstOrDefault(x => x.Id.Equals(id));
    ViewBag.Dinner = dinner;
    return dinner == null ? View("404") : View();
}
```

The view invoked by this action needs to be added to the Views/Home folder as AddAttendee.cshtml. Create a new view, named AddAttendee and strongly typed using the IDinner type but choose the Empty scaffold and check "Create as partial view" and then edit the .cshtml file like this:

```
@model BrightstarDB.Samples.NerdDinner.Models.IRSVP

<h3>Join A Dinner</h3>
<p>To join the dinner @ViewBag.Dinner.Title on @ViewBag.Dinner.EventDate.ToLongDateString(),
   enter your email address below and click RSVP.</p>

@using(@Html.BeginForm("AddAttendee", "Home")) {
    @Html.ValidationSummary(true)
    @Html.Hidden("DinnerId", ViewBag.Dinner.Id as string)
    <div class="editor-label">@Html.LabelFor(m=>m.AttendeeEmail)</div>
    <div class="editor-field">
        @Html.EditorFor(m=>m.AttendeeEmail)
        @Html.ValidationMessageFor(m=>m.AttendeeEmail)
    </div>
    <p><input type="submit" value="Register"/></p>
}
<div>
    @Html.ActionLink("Back To List", "Index")
</div>
```

Note the use of a hidden field in the form that carries the Dinner ID so that when we handle the POST we know which dinner to connect the response to.

This is the code to handle the second part of the action:

```
[HttpPost]
public ActionResult AddAttendee(FormCollection form)
{
    if (ModelState.IsValid)
    {
        var rsvpDinnerId = form["DinnerId"];
        var dinner = _nerdDinners.Dinners.FirstOrDefault(d => d.Id.Equals(rsvpDinnerId));
        if (dinner != null)
        {
            var rsvp= new RSVP{AttendeeEmail = form["AttendeeEmail"], Dinner = dinner};
            _nerdDinners.RSVPs.Add(rsvp);
            _nerdDinners.SaveChanges();
            return RedirectToAction("Details", new {id = rsvp.Dinner.Id});
        }
    }
    return View();
}
```

Here we do not use the MVC framework to data-bind the form values to an RSVP object because it will attempt to put the ID from the URL (which is the dinner ID) into the Id field of the RSVP, which is not what we want. Instead we just get the FormCollection to allow us to retrieve the form values. The code retrieves the DinnerId from the form and uses that to get the IDinner entity from BrightstarDB. A new RSVP entity is then created using the AttendeeEmail value from the form and the dinner entity just found. The RSVP is then added to the BrightstarDB RSVPs collection and SaveChanges() is called to persist it. Finally the user is returned to the details page for the dinner.

Next, we modify the Details view so that it shows all attendees of a dinner. This is the updated CSHTML for the Details view:

```
@model BrightstarDB.Samples.NerdDinner.Models.IDinner

<fieldset>
    <legend>IDinner</legend>

    <div class="display-label">
         @Html.DisplayNameFor(model => model.Title)
    </div>
    <div class="display-field">
        @Html.DisplayFor(model => model.Title)
    </div>

    <div class="display-label">
         @Html.DisplayNameFor(model => model.Description)
    </div>
    <div class="display-field">
        @Html.DisplayFor(model => model.Description)
    </div>

    <div class="display-label">
         @Html.DisplayNameFor(model => model.EventDate)
    </div>
    <div class="display-field">
        @Html.DisplayFor(model => model.EventDate)
    </div>

    <div class="display-label">
         @Html.DisplayNameFor(model => model.Address)
    </div>
    <div class="display-field">
        @Html.DisplayFor(model => model.Address)
    </div>

    <div class="display-label">
         @Html.DisplayNameFor(model => model.HostedBy)
    </div>
    <div class="display-field">
        @Html.DisplayFor(model => model.HostedBy)
    </div>

    <div class="display-label">
        @Html.DisplayNameFor(model=>model.RSVPs)
    </div>
    <div class="display-field">
        @if (Model.RSVPs != null)
        {
            <ul>
                @foreach (var r in Model.RSVPs)
                {
```

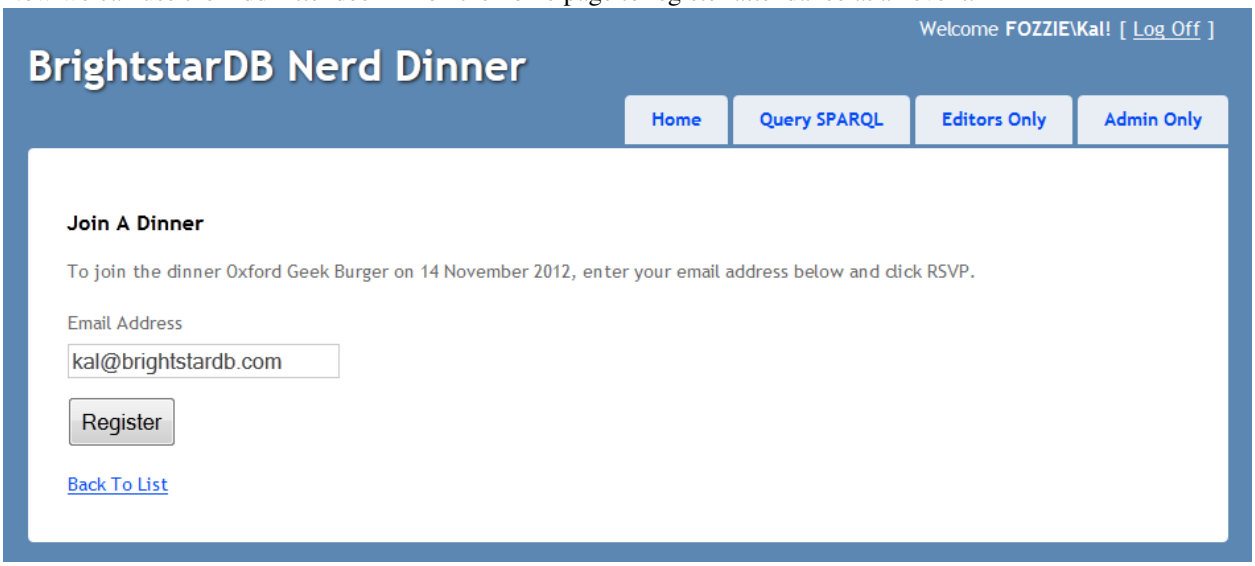
```
                    <li>@r.AttendeeEmail</li>
                }
            </ul>
        }
    </div>
</fieldset>
<p>
    @Html.ActionLink("Edit", "Edit", new { id=Model.Id }) |
    @Html.ActionLink("Back to List", "Index")
</p>
```

Finally we modify the Index view to add an Add Attendee action link to each row in the table. This is the updated CSHTML for the Index view:

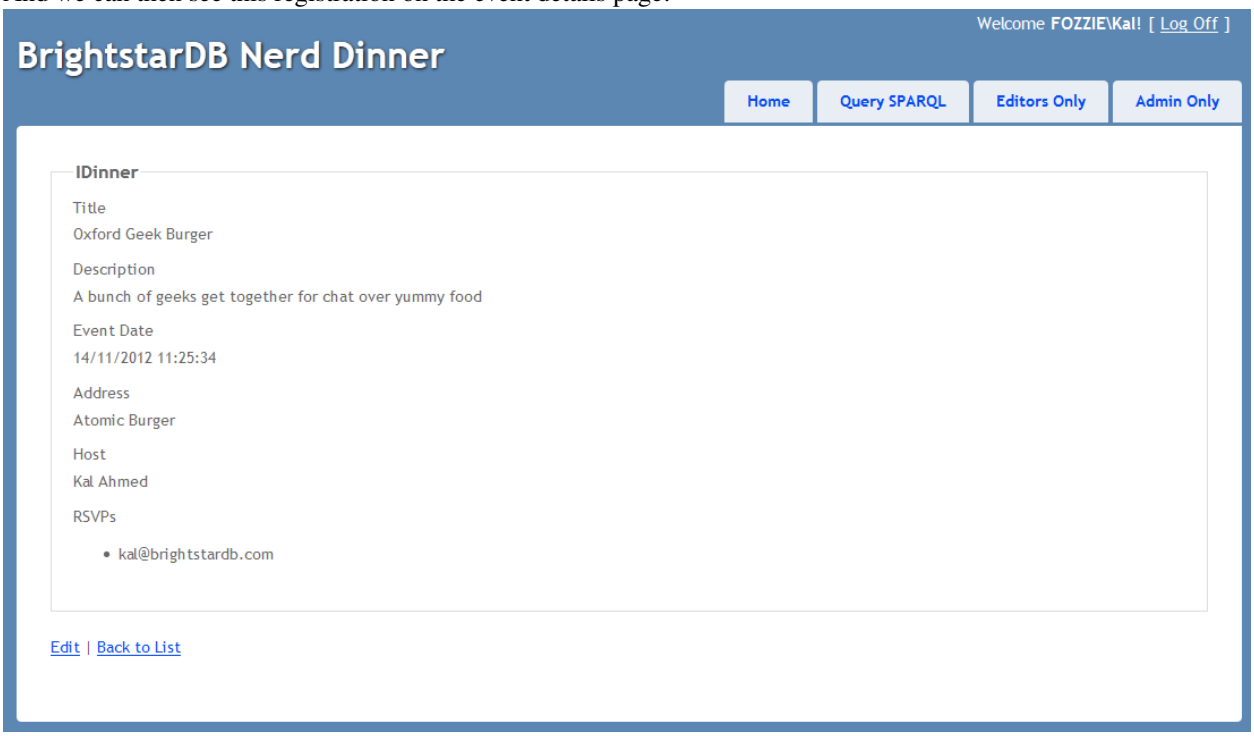
```
@model IEnumerable<BrightstarDB.Samples.NerdDinner.Models.IDinner>

<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table>
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.Title)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Description)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.EventDate)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Address)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.HostedBy)
        </th>
        <th></th>
    </tr>

@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Title)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Description)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.EventDate)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Address)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.HostedBy)
        </td>
        <td>
            @Html.ActionLink("Add Attendee", "AddAttendee", new { id=item.Id }) |
```

```
            @Html.ActionLink("Edit", "Edit", new { id=item.Id }) |
            @Html.ActionLink("Details", "Details", new { id=item.Id }) |
            @Html.ActionLink("Delete", "Delete", new { id=item.Id })
        </td>
    </tr>
}

</table>
```

Now we can use the Add Attendee link on the home page to register attendance at an event:



And we can then see this registration on the event details page:



10.2.3 Applying Model Changes

Change during development happens and many times, changes impact the persistent data model. Fortunately it is easy to modify the persistent data model with BrightstarDB.

As an example we are going to add the requirement for dinners to have a specific City field (perhaps to allow grouping of dinners by the city the occur in for example).

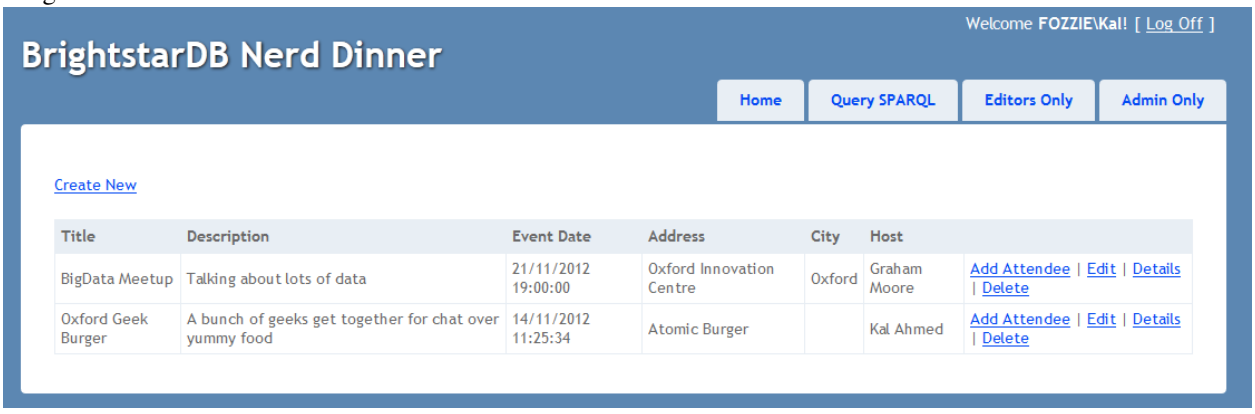
The first step is to modify the IDinner interface to add a City property:

```
[Entity]
public interface IDinner
{
    [Identifer("http://nerddinner.com/dinners#")]
    string Id { get; }
    string Title { get; set; }
    string Description { get; set; }
    DateTime EventDate { get; set; }
    string Address { get; set; }
    string City { get; set; }
    string HostedBy { get; set; }
    ICollection<IRSVP> RSVPs { get; set; }
}
```

Because this change modifies an entity interface, we need to ensure that the generated context classes are also updated. To update the context, right click on the NerdDinnerContext.tt and select "Run Custom Tool"

That is all that needs to be done from a BrightstarDB point of view! The City property is now assignable on all new and existing Dinner entities and you can write LINQ queries that make use of the City property. Of course, there are still a couple of things that need to change in our web interface. Open the Index, Create, Delete, Details and Edit views to add the new City property to the HTML so that you will be able to view and amend its data - the existing HTML in each of these views should provide you with the examples you need.

Note that if you create a new dinner, you will be required to enter a City, but existing dinners will not have a city assigned:



If you use a query to find or group dinners by their city, those dinners that have no value for the city will not be returned by the query, and of course if you try to edit one of those dinners, then you will be required to provide a value for the City field.

10.2.4 Adding a Custom Membership Provider

Custom Membership Providers are a quick and straightforward way of managing membership information when you wish to store that membership data in a data source that is not supported by the membership providers included within

the .NET framework. Often developers will need to implement custom membership providers even when storing the data in a supported data source, because the schema of that membership information differs from that in the default providers.

In this topic we are going to add a Custom Membership Provider to the Nerd Dinner sample so that users can register and login.

Adding the Custom Membership Provider and login Entity

1. Add a new class to your project and name it BrightstarMembershipProvider.cs
2. Make the class extend System.Web.Security.MembershipProvider. This is the abstract class that all ASP.NET membership providers must inherit from.
3. Right click on the MembershipProvider class name and choose "Implement abstract class" from the context menu, this automatically creates all the override methods that your custom class can implement.
4. Add a new interface to the Models directory and name it INerdDinnerLogin.cs
5. Add the [Entity] attribute to the interface, and add the properties shown below:
6. The Id property is decorated with the Identifier attribute to allow us to work with simpler string values rather than the full URI that is generated by BrightstarDB (for more information, please read the Entity Framework Documentation).

```
[Entity]
public interface INerdDinnerLogin
{
   [Identifier("http://nerddinner.com/logins/")]
   string Id { get; }
   string Username { get; set; }
   string Password { get; set; }
   string PasswordSalt { get; set; }
   string Email { get; set; }
   string Comments { get; set; }
   DateTime CreatedDate { get; set; }
   DateTime LastActive { get; set; }
   DateTime LastLoginDate { get; set; }
   bool IsActivated { get; set; }
   bool IsLockedOut { get; set; }
   DateTime LastLockedOutDate { get; set; }
   string LastLockedOutReason { get; set; }
   int? LoginAttempts { get; set; }
}
```

To update the Brightstar Entity Context, right click on the NerdDinnerContext.tt file and select "Run Custom Tool" from the context menu.

Configuring the application to use the Brightstar Membership Provider

To configure your web application to use this custom Membership Provider, we simply need to change the configuration values in the Web.config file in the root directory of the application. Change the membership node contained within the <system.web> to the snippet below:

```
<membership defaultProvider="BrightstarMembershipProvider">
  <providers>
    <clear/>
    <add name="BrightstarMembershipProvider"
```

```
        type="BrightstarDB.Samples.NerdDinner.BrightstarMembershipProvider, BrightStarDB.Samples.Ner
        enablePasswordReset="true"
        maxInvalidPasswordAttempts="5"
        minRequiredPasswordLength="6"
        minRequiredNonalphanumericCharacters="0"
        passwordAttemptWindow="10"
        applicationName="/" />
  </providers>
</membership>
```

Note that if the name of your project is not BrightstarDB.Samples.NerdDinner, you will have to change the type="" attribute to the correct full type reference.

We must also change the authentication method for the web application to Forms authentication. This is done by adding the following inside the <system.web> section of the Web.config file:

```
<authentication mode="Forms"/>
```

If after making these changes you see an error message like this in the browser:

```
Parser Error Message: It is an error to use a section registered as
allowDefinition='MachineToApplication' beyond application level.  This error can be caused by
a virtual directory not being configured as an application in IIS.
```

The most likely problem is that you have added the <membership> and <authentication> tags into the Web.config file contained in the Views folder. These configuration elements must ONLY go in the Web.config file located in the project's root directory.

Adding functionality to the Custom Membership Provider

Note: For the purpose of keeping this example simple, we will leave some of these methods to throw `System.NotImplementedException`, but you can add in whatever logic suits your business requirements once you have the basic functionality up and running.

The full code for the `BrightstarMembershipProvider.cs` is given below, but can be broken down as follows:

Initialization

We add an `Initialize()` method along with a `GetConfigValue()` helper method to handle retrieving the configuration values from *Web.config*, and setting default values if it is unable to retrieve a value.

Private helper methods

We add three more helper methods: `CreateSalt()` and `CreatePasswordHash()` to help us with user passwords, and `ConvertLoginToMembershipUser()` to return a built in .NET MembershipUser object when given the BrightstarDB `INerdDinnerLogin` entity.

CreateUser()

The `CreateUser()` method is used when a user registers on our site, the first part of this code validates based on the configuration settings (such as whether an email must be unique) and then creates a NerdDinnerLogin entity, adds it to the NerdDinnerContext and saves the changes to the BrightstarDB store.

GetUser()

The `GetUser()` method simply looks up a login in the BrightstarDB store, and returns a .NET MembershipUser object with the help of the `ConvertLoginToMembershipUser()` method mentioned above.

GetUserNameByEmail()

The `GetUserNameByEmail()` method is similar to the `GetUser()` method but looks up by email rather than username. It's used by the `CreateUser()` method if the configuration settings specify that new users must have unique emails.

ValidateUser()

The `ValidateUser()` method is used when a user logs in to our web application. The login is looked up in the BrightstarDB store by username, and then the password is checked. If the checks pass successfully then it returns a true value which enables the user to successfully login.

```
using System;
using System.Collections.Specialized;
using System.Linq;
using System.Security.Cryptography;
using System.Web.Security;
using BrightstarDB.Samples.NerdDinner.Models;


namespace BrightstarDB.Samples.NerdDinner
{
    public class BrightstarMembershipProvider : MembershipProvider
    {


        #region Configuration and Initialization


        private string _applicationName;
        private const bool _requiresUniqueEmail = true;
        private int _maxInvalidPasswordAttempts;
        private int _passwordAttemptWindow;
        private int _minRequiredPasswordLength;
        private int _minRequiredNonalphanumericCharacters;
        private bool _enablePasswordReset;
        private string _passwordStrengthRegularExpression;
        private MembershipPasswordFormat _passwordFormat = MembershipPasswordFormat.Hashed;


        private string GetConfigValue(string configValue, string defaultValue)
        {
            if (string.IsNullOrEmpty(configValue))
                return defaultValue;


            return configValue;
        }


        public override void Initialize(string name, NameValueCollection config)
        {
            if (config == null) throw new ArgumentNullException("config");


            if (string.IsNullOrEmpty(name)) name = "BrightstarMembershipProvider";


            if (String.IsNullOrEmpty(config["description"]))
            {
                config.Remove("description");
```

```
            config.Add("description", "BrightstarDB Membership Provider");
        }


        base.Initialize(name, config);


        _applicationName = GetConfigValue(config["applicationName"],
                        System.Web.Hosting.HostingEnvironment.ApplicationVirtualPath);
        _maxInvalidPasswordAttempts = Convert.ToInt32(
                        GetConfigValue(config["maxInvalidPasswordAttempts"], "10"));
        _passwordAttemptWindow = Convert.ToInt32(
                        GetConfigValue(config["passwordAttemptWindow"], "10"));
        _minRequiredNonalphanumericCharacters = Convert.ToInt32(
                        GetConfigValue(config["minRequiredNonalphanumericCharacters"],
                                       "1"));
        _minRequiredPasswordLength = Convert.ToInt32(
                        GetConfigValue(config["minRequiredPasswordLength"], "6"));
        _enablePasswordReset = Convert.ToBoolean(
                        GetConfigValue(config["enablePasswordReset"], "true"));
        _passwordStrengthRegularExpression = Convert.ToString(
                         GetConfigValue(config["passwordStrengthRegularExpression"], ""));


    }

    #endregion


    #region Properties


    public override string ApplicationName
    {
        get { return _applicationName; }
        set { _applicationName = value; }
    }


    public override int MaxInvalidPasswordAttempts
    {
        get { return _maxInvalidPasswordAttempts; }
    }


    public override int MinRequiredNonAlphanumericCharacters
    {
        get { return _minRequiredNonalphanumericCharacters; }
    }


    public override int MinRequiredPasswordLength
    {
        get { return _minRequiredPasswordLength; }
    }


    public override int PasswordAttemptWindow
```

```
{
    get { return _passwordAttemptWindow; }
}


public override MembershipPasswordFormat PasswordFormat
{
    get { return _passwordFormat; }
}


public override string PasswordStrengthRegularExpression
{
    get { return _passwordStrengthRegularExpression; }
}


public override bool RequiresUniqueEmail
{
    get { return _requiresUniqueEmail; }
}
#endregion


#region Private Methods


private static string CreateSalt()
{
    var rng = new RNGCryptoServiceProvider();
    var buffer = new byte[32];
    rng.GetBytes(buffer);
    return Convert.ToBase64String(buffer);
}


private static string CreatePasswordHash(string password, string salt)
{
    var snp = string.Concat(password, salt);
    var hashed = FormsAuthentication.HashPasswordForStoringInConfigFile(snp, "sha1");
    return hashed;


}

/// <summary>
/// This helper method returns a .NET MembershipUser object generated from the
/// supplied BrightstarDB entity
/// </summary>
private static MembershipUser ConvertLoginToMembershipUser(INerdDinnerLogin login)
{
    if (login == null) return null;
    var user = new MembershipUser("BrightstarMembershipProvider",
        login.Username, login.Id, login.Email,
        "", "", login.IsActivated, login.IsLockedOut,
        login.CreatedDate, login.LastLoginDate,
        login.LastActive, DateTime.UtcNow, login.LastLockedOutDate);
    return user;
```

```
}

#endregion

public override MembershipUser CreateUser(
                                    string username,
                                                                        string passwor
                                                                        string email,
                                                                        string passwor
                                                                        string passwor
                                                                        bool isApprove
                                                                        object provide
                                                                        out Membership
{
    var args = new ValidatePasswordEventArgs(email, password, true);

    OnValidatingPassword(args);

    if (args.Cancel)
    {
        status = MembershipCreateStatus.InvalidPassword;
        return null;
    }

    if (string.IsNullOrEmpty(email))
    {
        status = MembershipCreateStatus.InvalidEmail;
        return null;
    }

    if (string.IsNullOrEmpty(password))
    {
        status = MembershipCreateStatus.InvalidPassword;
        return null;
    }

    if (RequiresUniqueEmail && GetUserNameByEmail(email) != "")
    {
        status = MembershipCreateStatus.DuplicateEmail;
        return null;
    }

    var u = GetUser(username, false);

    try
    {
        if (u == null)
        {
            var salt = CreateSalt();

            //Create a new NerdDinnerLogin entity and set the properties
            var login = new NerdDinnerLogin
            {
                Username = username,
                Email = email,
                PasswordSalt = salt,
```

```
                Password = CreatePasswordHash(password, salt),
                CreatedDate = DateTime.UtcNow,
                IsActivated = true,
                IsLockedOut = false,
                LastLockedOutDate = DateTime.UtcNow,
                LastLoginDate = DateTime.UtcNow,
                LastActive = DateTime.UtcNow
            };

            //Create a context using the connection string in the Web.Config
            var context = new NerdDinnerContext();

            //Add the entity to the context
            context.NerdDinnerLogins.Add(login);

            //Save the changes to the BrightstarDB store
            context.SaveChanges();

            status = MembershipCreateStatus.Success;
            return GetUser(username, true /*online*/);
        }
    }
    catch (Exception)
    {
        status = MembershipCreateStatus.ProviderError;
        return null;
    }


    status = MembershipCreateStatus.DuplicateUserName;
    return null;
}


public override MembershipUser GetUser(string username, bool userIsOnline)
{
    if (string.IsNullOrEmpty(username)) return null;
    //Create a context using the connection string in Web.config
    var context = new NerdDinnerContext();
    //Query the store for a NerdDinnerLogin that matches the supplied username
    var login = context.NerdDinnerLogins.Where(l =>
                          l.Username.Equals(username)).FirstOrDefault();
    if (login == null) return null;
    if(userIsOnline)
    {
        // if the call states that the user is online, update the LastActive property
        // of the NerdDinnerLogin
        login.LastActive = DateTime.UtcNow;
        context.SaveChanges();
    }
    return ConvertLoginToMembershipUser(login);
}


public override string GetUserNameByEmail(string email)
{
    if (string.IsNullOrEmpty(email)) return "";
    //Create a context using the connection string in Web.config
```

```
            var context = new NerdDinnerContext();
            //Query the store for a NerdDinnerLogin that matches the supplied username
            var login = context.NerdDinnerLogins.Where(l =>
                                    l.Email.Equals(email)).FirstOrDefault();
            if (login == null) return string.Empty;
            return login.Username;
        }

        public override bool ValidateUser(string username, string password)
        {
            //Create a context using the connection string set in Web.config
            var context = new NerdDinnerContext();
            //Query the store for a NerdDinnerLogin matching the supplied username
            var logins = context.NerdDinnerLogins.Where(l => l.Username.Equals(username));
            if (logins.Count() == 1)
            {
                //Ensure that only a single login matches the supplied username
                var login = logins.First();
                // Check the properties on the NerdDinnerLogin to ensure the user account is
                // activated and not locked out
                if (login.IsLockedOut || !login.IsActivated) return false;
                // Validate the password of the NerdDinnerLogin against the supplied password
                var validatePassword = login.Password == CreatePasswordHash(password, login.PasswordS
                if (!validatePassword)
                {
                    //return validation failure
                    return false;
                }
                //return validation success
                return true;
            }
            return false;
        }


        #region MembershipProvider properties and methods not implemented for this tutorial
...
        #endregion

    }
}
```

Extending the MVC application

All the models, views and controllers needed to implement the logic logic are generated automatically when creating a new MVC4 Web Application if the option for "Internet Application" is selected. However, if you are following this tutorial through from the beginning you will need to add this infrastructure by hand. The infrastructure includes:

- An AccountController class with ActionResult methods for logging in, logging out and registering (in `AccountController.cs` in the Controllers folder).
- `AccountModels.cs` which contains classes for LogonModel and RegisterModel (in the Models folder).
- LogOn, Register, ChangePassword and ChangePasswordSuccess views that use the models to display form fields and validate input from the user (in the Views/Account folder).
- A _LogOnPartial view that is used in the main _Layout view to display a login link, or the username if the user is logged in (in the Views/Shared folder).

Note: These files can be found in [INSTALLDIR]\Samples\NerdDinner\BrightstarDB.Samples.NerdDinner

The details of the contents of these files is beyond the scope of this tutorial, however the infrastructure is all designed to work with the configured Membership Provider for the web application - in our case the `BrightstarMembershipProvider` class we have just created.

The AccountController created here has some dependencies on the Custom Role Provider discussed in the next section. You will need to complete the steps in the next section before you will be able to successfully register a user in the web application.

Summary

In this tutorial we have walked through some simple steps to use a Custom Membership Provider to allow BrightstarDB to handle the authentication of users on your MVC3 Web Application.

For simplicity, we have kept the same structure of membership information as we would find in a default provider, but you can expand on this sample to include extra membership information by simply adding more properties to the BrightstarDB entity.

10.2.5 Adding a Custom Role Provider

As with Custom Membership Providers, Custom Role Providers allow developers to use role management within application when either the role information is stored in a data source other than that supported by the default providers, or the role information is managed in a schema which differs from that set out in the default providers.

In this topic we are going to add a Custom Role Provider to the Nerd Dinner sample so that we can restrict certain areas from users who are not members of the appropriate role.

Adding the Custom Role Provider

1. Add the following line to the INerdDinnerLogin interface's properties:

   ```
   ICollection<string> Roles { get; set; }
   ```

2. To update the context classes, right click on the NerdDinnerContext.tt file and select "Run Custom Tool" from the context menu.
3. Add a new class to your project and name it BrightstarRoleProvider.cs
4. Make this new class inherit from the RoleProvider class (System.Web.Security namespace)
5. Right click on the RoleProvider class name and choose "Implement abstract class" from the context menu, this automatically creates all the override methods that your custom class can implement.

Configuring the application to use the Brightstar Membership Provider

To configure your web application to use the Custom Role Provider, add the following to your Web.config, inside the <system.web> section:

```
<roleManager  enabled="true" defaultProvider="BrightstarRoleProvider">
  <providers>
    <clear/>
    <add name="BrightstarRoleProvider"
         type="BrightstarDB.Samples.NerdDinner.BrightstarRoleProvider" applicationName="/" />
  </providers>
</roleManager>
```

To set up the default login path for the web application, replace the <authentication> element in the Web.config file with the following:

```
<authentication mode="Forms">
  <forms loginUrl="/Account/LogOn"/>
</authentication>
```

Adding functionality to the Custom Role Provider

The full code for the `BrightstarRoleProvider.cs` is given below, but can be broken down as follows:

Initialization

We add an `Initialize()` method along with a `GetConfigValue()` helper method to handle retrieving the configuration values from Web.config, and setting default values if it is unable to retrieve a value.

GetRolesForUser()

This method returns the contents of the Roles collection that we added to the INerdDinnerLogin entity as a string array.

AddUsersToRoles()

This method loops through the usernames and role names supplied, and looks up the logins from the BrightstarDB store. When found, the role names are added to the Roles collection for that login.

RemoveUsersFromRoles()

This method loops through the usernames and role names supplied, and looks up the logins from the BrightstarDB store. When found, the role names are removed from the Roles collection for that login.

IsUserInRole()

The BrightstarDB store is searched for the login who matches the supplied username, and then a true or false is passed back depending on whether the role name was found in that login's Role collection. If the login is inactive or locked out for any reason, then a false value is passed back.

GetUsersInRole()

BrightstarDB is queried for all logins that contain the supplied role name in their Roles collection.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Security;
using BrightstarDB.Samples.NerdDinner.Models;


namespace BrightstarDB.Samples.NerdDinner
{
    public class BrightstarRoleProvider : RoleProvider
    {
        #region Initialization

        private string _applicationName;


        private static string GetConfigValue(string configValue, string defaultValue)
        {
            if (string.IsNullOrEmpty(configValue))
```

```
        return defaultValue;


    return configValue;
}


public override void Initialize(string name,
                  System.Collections.Specialized.NameValueCollection config)
{
    if (config == null) throw new ArgumentNullException("config");


    if (string.IsNullOrEmpty(name)) name = "NerdDinnerRoleProvider";


    if (String.IsNullOrEmpty(config["description"]))
    {
        config.Remove("description");
        config.Add("description", "Nerd Dinner Membership Provider");
    }
    base.Initialize(name, config);
    _applicationName = GetConfigValue(config["applicationName"],
                System.Web.Hosting.HostingEnvironment.ApplicationVirtualPath);
}

#endregion


/// <summary>
/// Gets a list of the roles that a specified user is in for the configured
/// applicationName.
/// </summary>
/// <returns>
/// A string array containing the names of all the roles that the specified user is
/// in for the configured applicationName.
/// </returns>
/// <param name="username">The user to return a list of roles for.</param>
public override string[] GetRolesForUser(string username)
{
    if (string.IsNullOrEmpty(username)) throw new ArgumentNullException("username");
    //create a new BrightstarDB context using the values in Web.config
    var context = new NerdDinnerContext();
    //find a match for the username
    var login = context.NerdDinnerLogins.Where(l =>
                                    l.Username.Equals(username)).FirstOrDefault();
    if (login == null) return null;
    //return the Roles collection
    return login.Roles.ToArray();
}


/// <summary>
/// Adds the specified user names to the specified roles for the configured
/// applicationName.
/// </summary>
/// <param name="usernames">
///   A string array of user names to be added to the specified roles.
```

```
/// </param>
/// <param name="roleNames">
///  A string array of the role names to add the specified user names to.
/// </param>
public override void AddUsersToRoles(string[] usernames, string[] roleNames)
{
    //create a new BrightstarDB context using the values in Web.config
    var context = new NerdDinnerContext();
    foreach (var username in usernames)
    {
        //find the match for the username
        var login = context.NerdDinnerLogins.Where(l =>
                            l.Username.Equals(username)).FirstOrDefault();
        if (login == null) continue;
        foreach (var role in roleNames)
        {
            // if the Roles collection of the login does not already contain the
            // role, then add it
            if (login.Roles.Contains(role)) continue;
            login.Roles.Add(role);
        }
    }
    context.SaveChanges();
}


/// <summary>
/// Removes the specified user names from the specified roles for the configured
/// applicationName.
/// </summary>
/// <param name="usernames">
///  A string array of user names to be removed from the specified roles.
/// </param>
/// <param name="roleNames">
///  A string array of role names to remove the specified user names from.
/// </param>
public override void RemoveUsersFromRoles(string[] usernames, string[] roleNames)
{
    //create a new BrightstarDB context using the values in Web.config
    var context = new NerdDinnerContext();
    foreach (var username in usernames)
    {
        //find the match for the username
        var login = context.NerdDinnerLogins.Where(l =>
                                    l.Username.Equals(username)).FirstOrDefault();
        if (login == null) continue;
        foreach (var role in roleNames)
        {
            //if the Roles collection of the login contains the role, then remove it
            if (!login.Roles.Contains(role)) continue;
            login.Roles.Remove(role);
        }
    }
    context.SaveChanges();
}


/// <summary>
```

```
/// Gets a value indicating whether the specified user is in the specified role for
/// the configured applicationName.
/// </summary>
/// <returns>
/// true if the specified user is in the specified role for the configured
/// applicationName; otherwise, false.
/// </returns>
/// <param name="username">The username to search for.</param>
/// <param name="roleName">The role to search in.</param>
public override bool IsUserInRole(string username, string roleName)
{
    try
    {
        //create a new BrightstarDB context using the values in Web.config
        var context = new NerdDinnerContext();
        //find a match for the username
        var login = context.NerdDinnerLogins.Where(l =>
                                    l.Username.Equals(username)).FirstOrDefault();
        if (login == null || login.IsLockedOut || !login.IsActivated)
        {
            // no match or inactive automatically returns false
            return false;
        }
        // if the Roles collection of the login contains the role we are checking
        // for, return true
        return login.Roles.Contains(roleName.ToLower());
    }
    catch (Exception)
    {
        return false;
    }
}


/// <summary>
/// Gets a list of users in the specified role for the configured applicationName.
/// </summary>
/// <returns>
/// A string array containing the names of all the users who are members of the
/// specified role for the configured applicationName.
/// </returns>
/// <param name="roleName">The name of the role to get the list of users for.</param>
public override string[] GetUsersInRole(string roleName)
{
    if (string.IsNullOrEmpty(roleName)) throw new ArgumentNullException("roleName");
    //create a new BrightstarDB context using the values in Web.config
    var context = new NerdDinnerContext();
    //search for all logins who have the supplied roleName in their Roles collection
    var usersInRole = context.NerdDinnerLogins.Where(l =>
                l.Roles.Contains(roleName.ToLower())).Select(l => l.Username).ToList();
    return usersInRole.ToArray();
}

/// <summary>
/// Gets a value indicating whether the specified role name already exists in the
/// role data source for the configured applicationName.
/// </summary>
/// <returns>
```

```
/// true if the role name already exists in the data source for the configured
/// applicationName; otherwise, false.
/// </returns>
/// <param name="roleName">The name of the role to search for in the data source.</param>
public override bool RoleExists(string roleName)
{
    //for the purpose of the sample the roles are hard coded
    return roleName.Equals("admin") ||
           roleName.Equals("editor") ||
           roleName.Equals("standard");
}

/// <summary>
/// Gets a list of all the roles for the configured applicationName.
/// </summary>
/// <returns>
/// A string array containing the names of all the roles stored in the data source
/// for the configured applicationName.
/// </returns>
public override string[] GetAllRoles()
{
    //for the purpose of the sample the roles are hard coded
    return new string[] { "admin", "editor", "standard" };
}


/// <summary>
/// Gets an array of user names in a role where the user name contains the specified
/// user name to match.
/// </summary>
/// <returns>
/// A string array containing the names of all the users where the user name matches
/// <paramref name="usernameToMatch"/> and the user is a member of the specified role.
/// </returns>
/// <param name="roleName">The role to search in.</param>
/// <param name="usernameToMatch">The user name to search for.</param>
public override string[] FindUsersInRole(string roleName, string usernameToMatch)
{
    if (string.IsNullOrEmpty(roleName)) {
        throw new ArgumentNullException("roleName");
    }
    if (string.IsNullOrEmpty(usernameToMatch)) {
        throw new ArgumentNullException("usernameToMatch");
    }

    var allUsersInRole = GetUsersInRole(roleName);
    if (allUsersInRole == null || allUsersInRole.Count() < 1) {
        return new string[] { "" };
    }
    var match = (from u in allUsersInRole where u.Equals(usernameToMatch) select u);
    return match.ToArray();
}


#region Properties


/// <summary>
```

```
        /// Gets or sets the name of the application to store and retrieve role information for.
        /// </summary>
        /// <returns>
        /// The name of the application to store and retrieve role information for.
        /// </returns>
        public override string ApplicationName
        {
            get { return _applicationName; }
            set { _applicationName = value; }
        }


        #endregion


        #region Not Implemented Methods

        /// <summary>
        /// Adds a new role to the data source for the configured applicationName.
        /// </summary>
        /// <param name="roleName">The name of the role to create.</param>
        public override void CreateRole(string roleName)
        {
            //for the purpose of the sample the roles are hard coded
            throw new NotImplementedException();
        }


        /// <summary>
        /// Removes a role from the data source for the configured applicationName.
        /// </summary>
        /// <returns>
        /// true if the role was successfully deleted; otherwise, false.
        /// </returns>
        /// <param name="roleName">The name of the role to delete.</param>
        /// <param name="throwOnPopulatedRole">If true, throw an exception if <paramref name="roleNa
        /// one or more members and do not delete <paramref name="roleName"/>.</param>
        public override bool DeleteRole(string roleName, bool throwOnPopulatedRole)
        {
            //for the purpose of the sample the roles are hard coded
            throw new NotImplementedException();
        }

        #endregion
    }
}
```

Adding Secure Sections to the Website

To display the functionality of the new Custom Role Provider, add 2 new ViewResult methods to the Home Controller. Notice that the [Authorize] MVC attribute has been added to each of the methods to restrict access to users in those roles only.

```
[Authorize(Roles = "editor")]
public ViewResult SecureEditorSection()
{
    return View();
```

```
}


[Authorize(Roles = "admin")]
public ViewResult SecureAdminSection()
{
    return View();
}
```

Right click on the View() methods, and select "Add View" for each. This automatically adds the SecureEditorSection.cshtml and SecureAdminSection.cshtml files to the Home view folder.

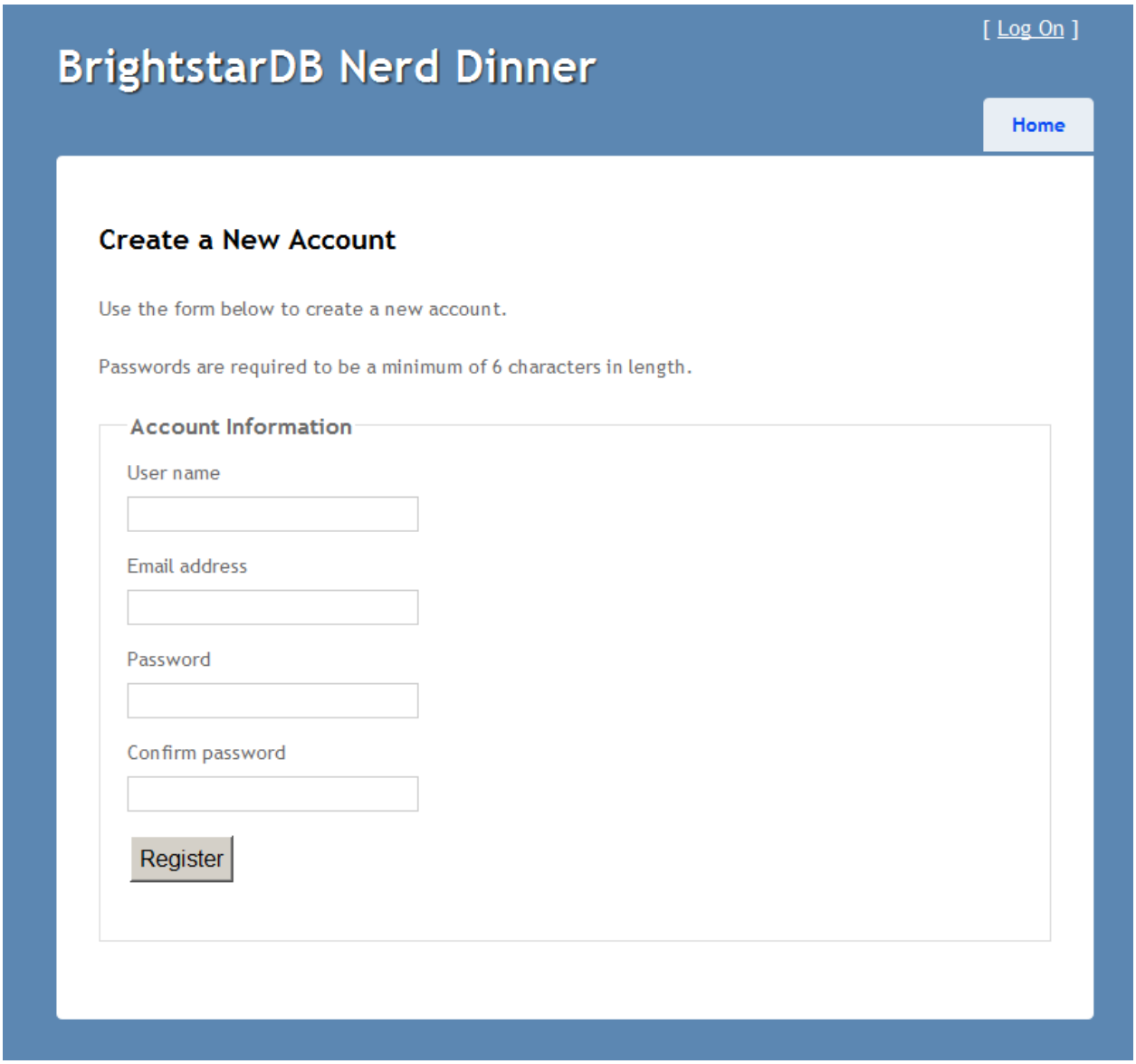
To be able to navigate to these sections, open the file Views/Shared/_Layout.cshtml and add two new action links to the main navigation menu:

```
<div id="menucontainer">
  <ul id="menu">
    <li>@Html.ActionLink("Home", "Index", "Home")</li>
    <li>@Html.ActionLink("Query SPARQL", "Index", "Sparql")</li>
    <li>@Html.ActionLink("Editors Only", "SecureEditorSection", "Home")</li>
    <li>@Html.ActionLink("Admin Only", "SecureAdminSection", "Home")</li>
  </ul>
</div>
```

In a real world application, you would manage roles within your own administration section, but for the purpose of this sample we are going with an overly simplistic way of adding a user to a role.

Running the Application

Press F5 to run the application. You will notice a [Log On] link in the top right hand corner of the screen. You can navigate to the registration page via the logon page.

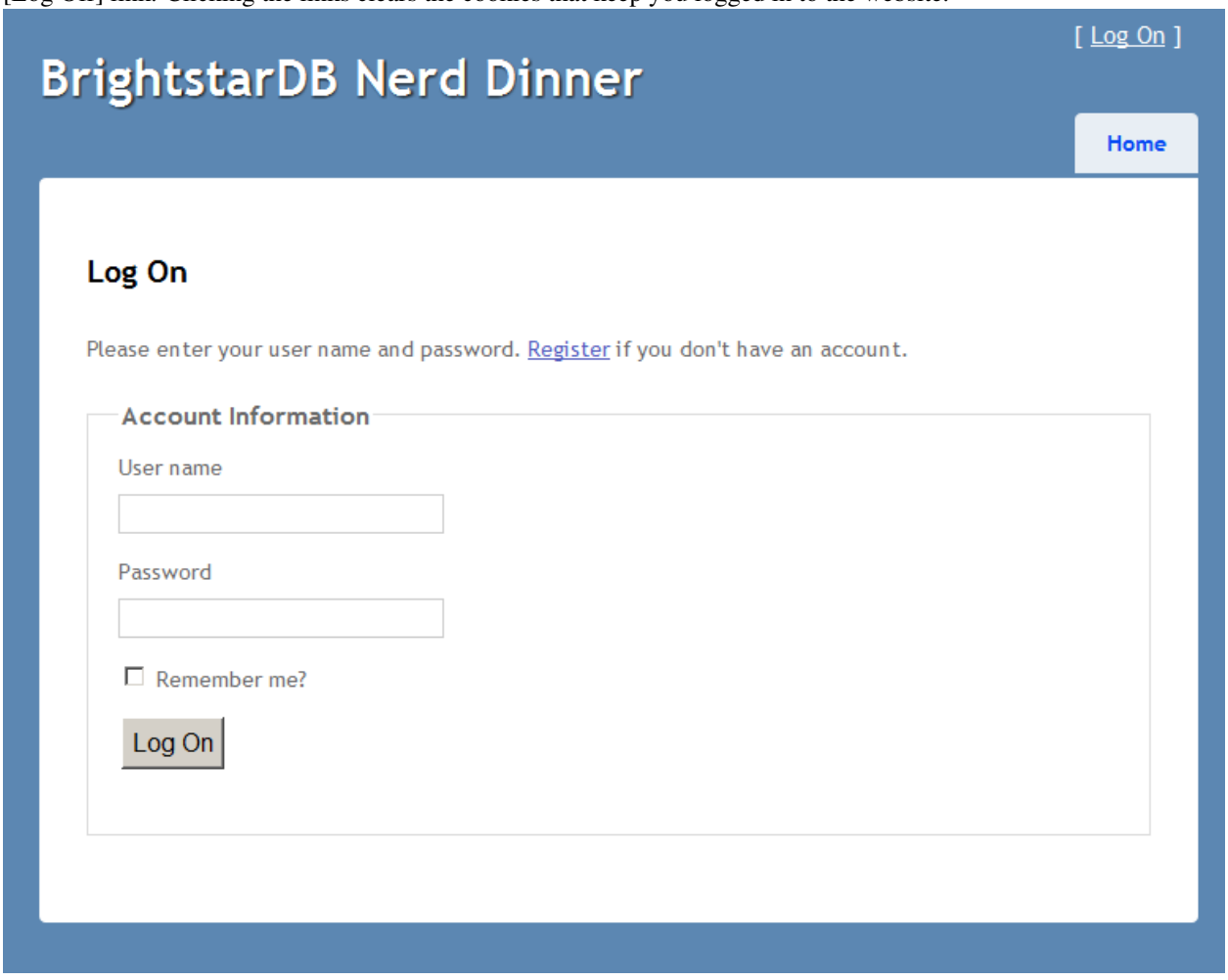


Register

Choosing a username, email and password will create a login entity for you in the BrightstarDB store, and automatically log you in.

Logged In

The partial view that contains the login link code recognizes that you are logged in and displays your username and a [Log Off] link. Clicking the links clears the cookies that keep you logged in to the website.



Log On

You can log on again at any time by entering your username and password.

Role Authorization

Clicking on the navigation links to "Secure Editor Section" will allow access to that view. Whereas the "Secure Admin Section" will not pass authorization - by default MVC redirects the user to the login view.

10.2.6 Adding Linked Data Support

As data on the web becomes more predominant, it is becoming increasingly important to be able to expose the underlying data of a web application in some way that is easy for external applications to consume. While many web applications choose to expose bespoke APIs, these are difficult for developers to use because each API has its own data structures and calls to access data. However there are two well supported standards for publishing data on the web - OData and SPARQL.

OData is an open standard, originally created by Microsoft, that provides a framework for exposing a collection of entities as data accessible by URIs and represented in ATOM feeds. SPARQL is a standard from the W3C for querying an RDF data store. Because BrightstarDB is, under the hood, an RDF data store adding SPARQL support is pretty straightforward; and because the BrightstarDB Entity Framework provides a set of entity classes, it is also very easy to create an OData endpoint.

In this section we will show how to add these different forms of Linked Data to your web application.

Create a SPARQL Action

The standard way of interfacing to a SPARQL endpoint is to either use an HTTP GET with a ?query= parameter that carries the SPARQL query as a string; or to use an HTTP POST which has a form encoded in the POST request with a query field in it. For this example we will do the latter as it is easiest to show and test with a browser-based API. We will create a query action at /sparql, and include a form that allows a SPARQL query to be submitted through the browser. To do this we need to create a new Controller to handle the /sparql URL.

Right-click on the Controllers folder and choose Add > Controller. In the dialog that is displayed, change the controller name to `SparqlController`, and choose the **Empty MVC Controller** template option from the drop-down list.

Edit the `SparqlController.cs` file to add the following two methods to the class:

```
public ViewResult Index()
{
    return View();
}

[HttpPost]
[ValidateInput(false)]
public ActionResult Index(string query)
{
    if (String.IsNullOrEmpty(query))
    {
        return View("Error");
    }
    var client = BrightstarService.GetClient();
    var results = client.ExecuteQuery("NerdDinner", query);
    return new FileStreamResult(results, "application/xml; charset=utf-16");
}
```

The first method just displays a form that will allow a user to enter a SPARQL query. The second method handles a POST operation and extracts the SPARQL query and executes it, returning the results to the browser directly as an XML data stream.

Create a new folder under Views called "Sparql" and add a new View to the Views\Sparql with the name Index.cshtml. This view simply displays a form with a large enough text box to allow a query to be entered:

```
<h2>SPARQL</h2>


@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)

    <p>Enter your SPARQL query in the text box below:</p>


    @Html.TextArea("query",
                   "SELECT ?d WHERE {?d a <http://brightstardb.com/namespaces/default/Dinner>}",
                   10, 50, null)
    <p>
        <input type="submit" value="Query" />
    </p>
}
```

Now you can compile and run the web application again and click on the Query SPARQL link at the top of the page (or simply navigate to the /sparql address for the web application). As this is a normal browser HTTP GET, you will see the form rendered by the first of the two action methods. By default this contains a SPARQL query that should work nicely against the NerdDinner entity model, returning the URI identifiers of all Dinner entities in the BrightstarDB data store.

Clicking on the Query button submits the form, simulating an HTTP POST from an external application. The results are returned as raw XML, which will be formatted and displayed depending on which browser you use and your browser settings (the screenshot below is from a Firefox browser window).

This XML file does not appear to have any style information associated with it. The document tree is shown below.

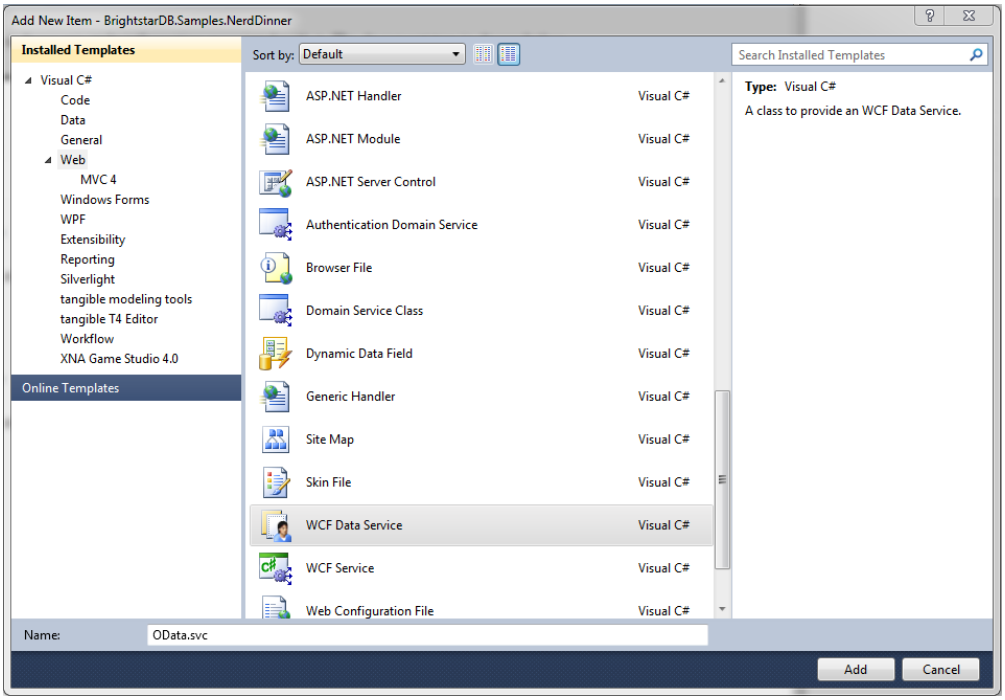

```
- <sparql>
  - <head>
      <variable name="d"/>
    </head>
  - <results>
    - <result>
      - <binding name="d">
        - <uri>
            http://nerddinner.com/dinners/30d5d7fe-c06f-4a87-902f-7acf607ca727
          </uri>
        </binding>
      </result>
    - <result>
      - <binding name="d">
        - <uri>
            http://nerddinner.com/dinners/4012ef60-ceab-4c66-a4f7-6c35e80372eb
          </uri>
        </binding>
      </result>
    </results>
  </sparql>
```


Creating an OData Provider

The Open Data Protocol (OData) is an open web protocol for querying and updating data. An OData provider can be added to BrightstarDB Entity Framework projects to allow OData consumers to query the underlying data.

The following steps describe how to create an OData provider to an existing project (in this example we add to the NerdDinner MVC Web Application project).

1. Right-click on the project in the Solution Explorer and select **Add New Item**. In the dialog that is displayed click on Web, and select WCF Data Service. Rename this to `OData.svc` and click **Add**.



2. Change the class inheritance from DataService to `EntityDataService`, and add the name of the BrightstarEntityContext to the type argument.

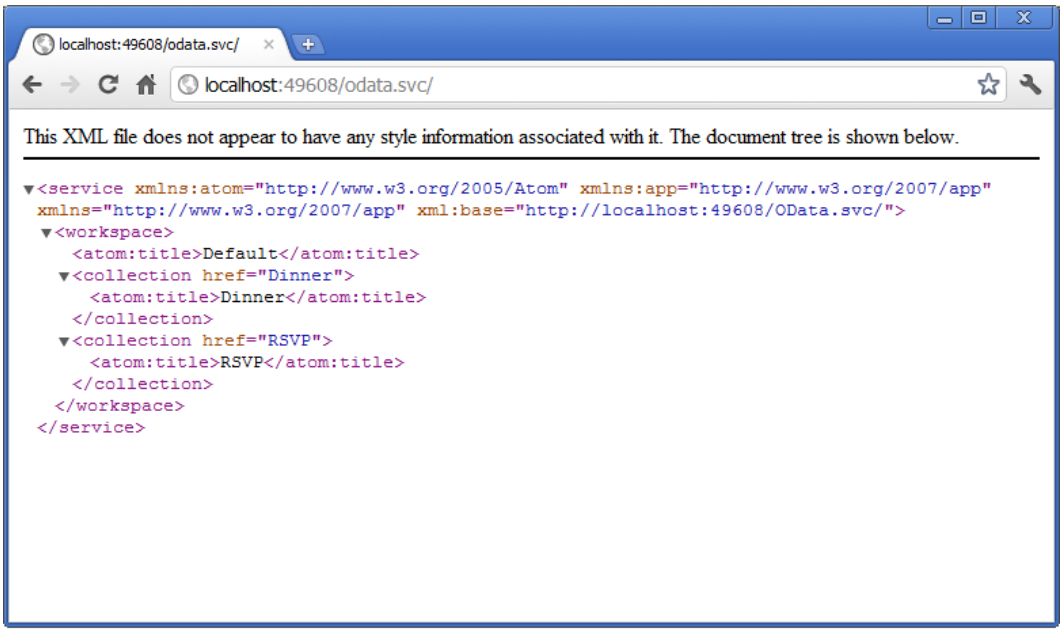
3. Edit the body of the method with the following configuration settings:

```
public class OData : EntityDataService<NerdDinnerContext>
{
  // This method is called only once to initialize service-wide policies.
  public static void InitializeService(DataServiceConfiguration config)
  {
    config.SetEntitySetAccessRule("*", EntitySetRights.AllRead);
    config.SetEntitySetAccessRule("NerdDinnerLogin", EntitySetRights.None);
    config.SetServiceOperationAccessRule("*", ServiceOperationRights.All);
    config.DataServiceBehavior.MaxProtocolVersion = DataServiceProtocolVersion.V2;
  }
}
```

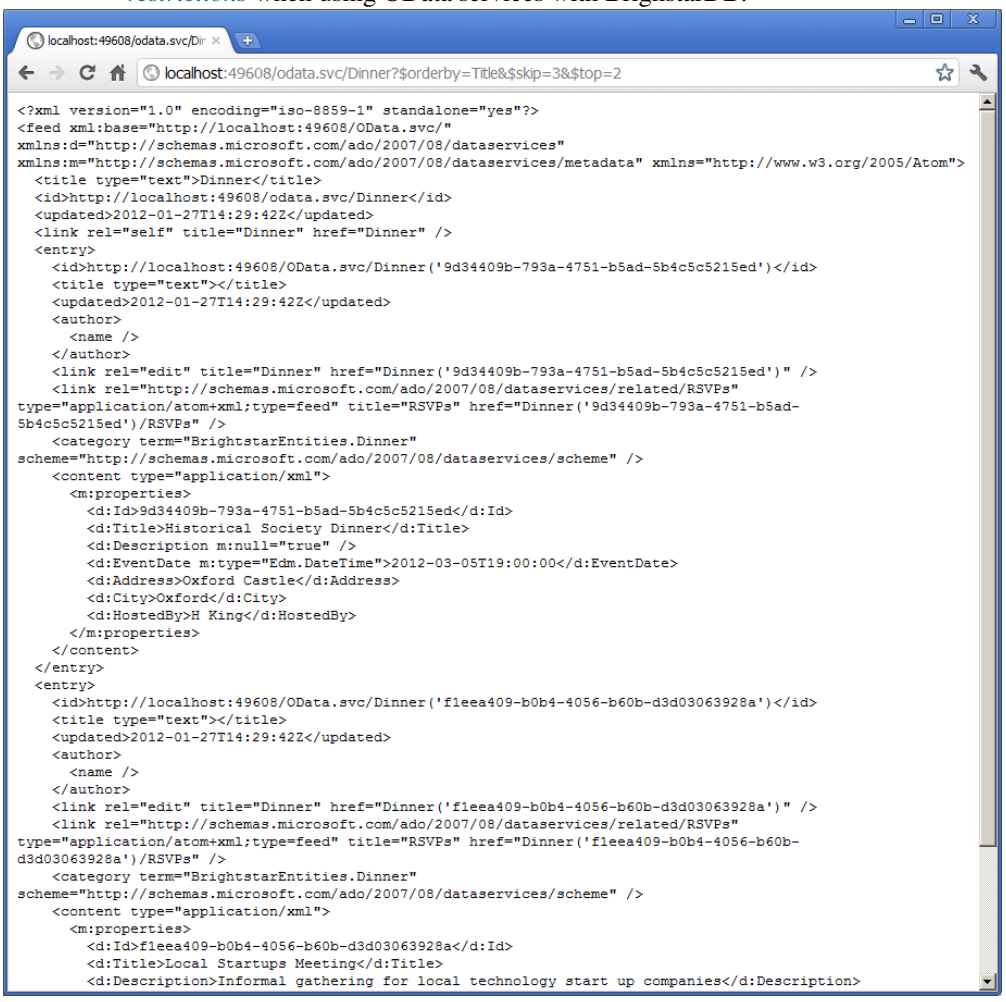
Note: The NerdDinnerLogin set has been given EntitySetRights of None. This hides the set (which contains sensitive login information) from the OData service

---

4. Rebuild and run the project. Browse to /OData.svc and you will see the standard OData metadata page displaying the entity sets from BrightstarDB



5. The OData service can now be queried using the standard OData conventions. There are a *few restrictions* when using OData services with BrighstarDB.

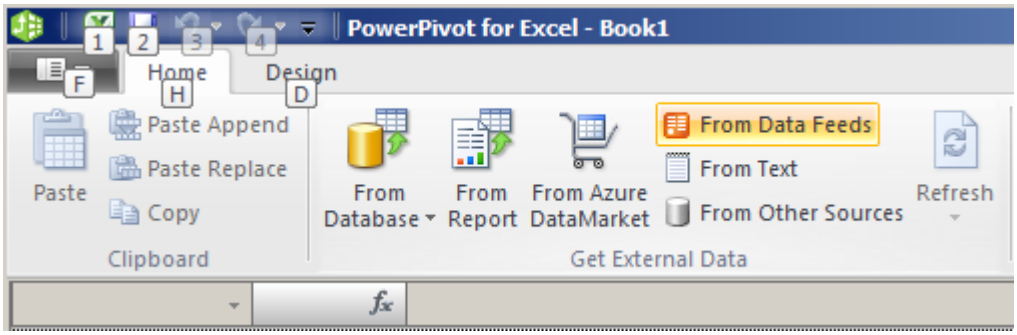


10.2.7 Consuming OData in PowerPivot

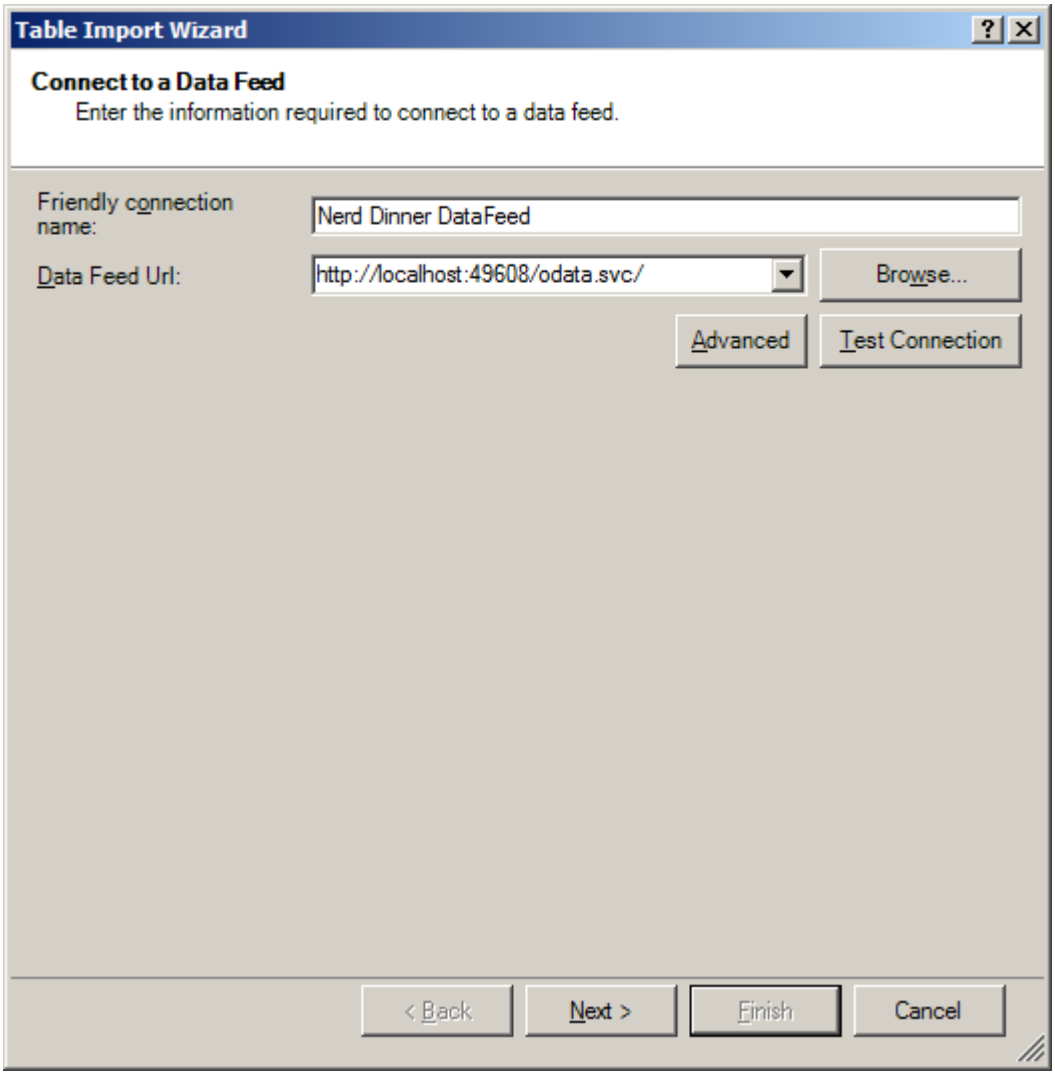
The data in BrighstarDB can be consumed by various OData consumers. In this topic we look at consuming the data using PowerPivot (a list of recommended OData consumers can be found odata.org/consumers).

To consume OData from BrightstarDB in PowerPivot:

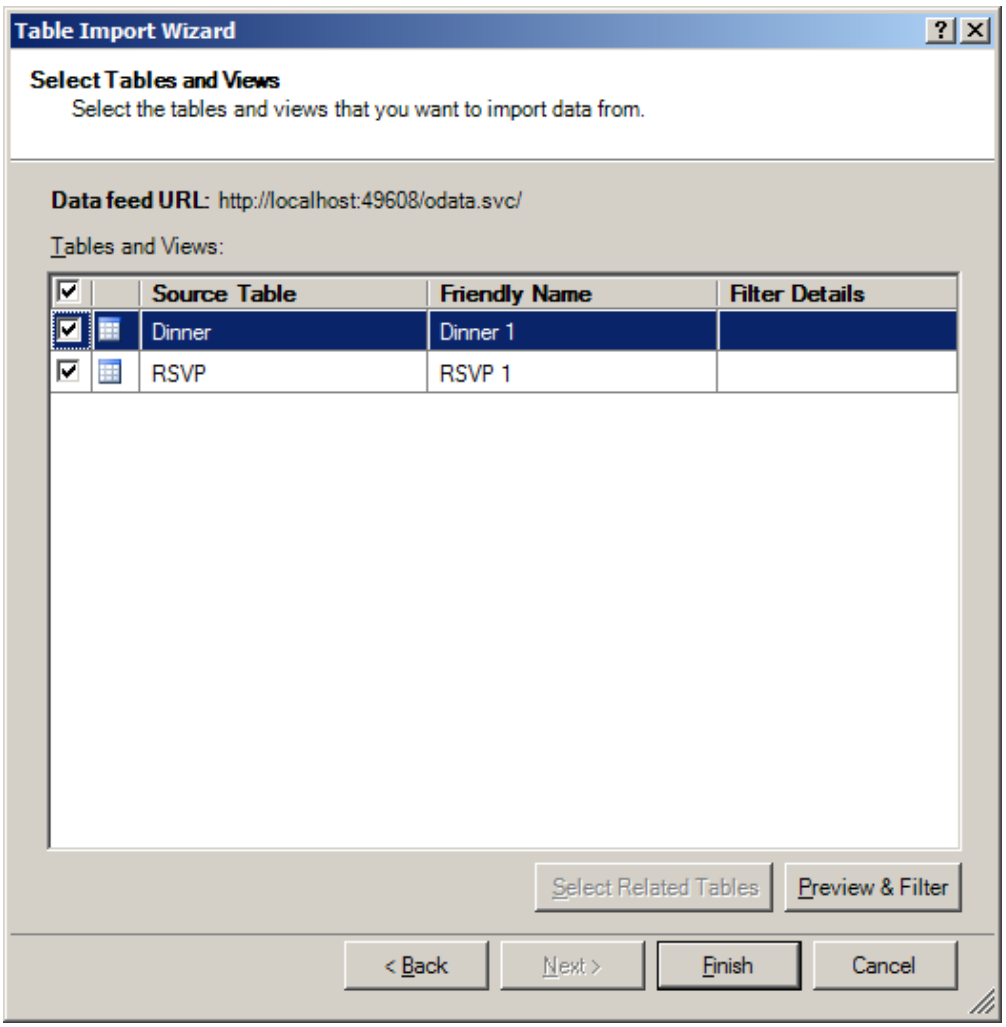
1. Open Excel, click the PowerPivot tab and open the PowerPivot window. If you do not have PowerPivot installed, you can download it from powerpivot.com
2. To consume data from BrightstarDB, click the **From Data Feeds** button in the **Get External Data** section:



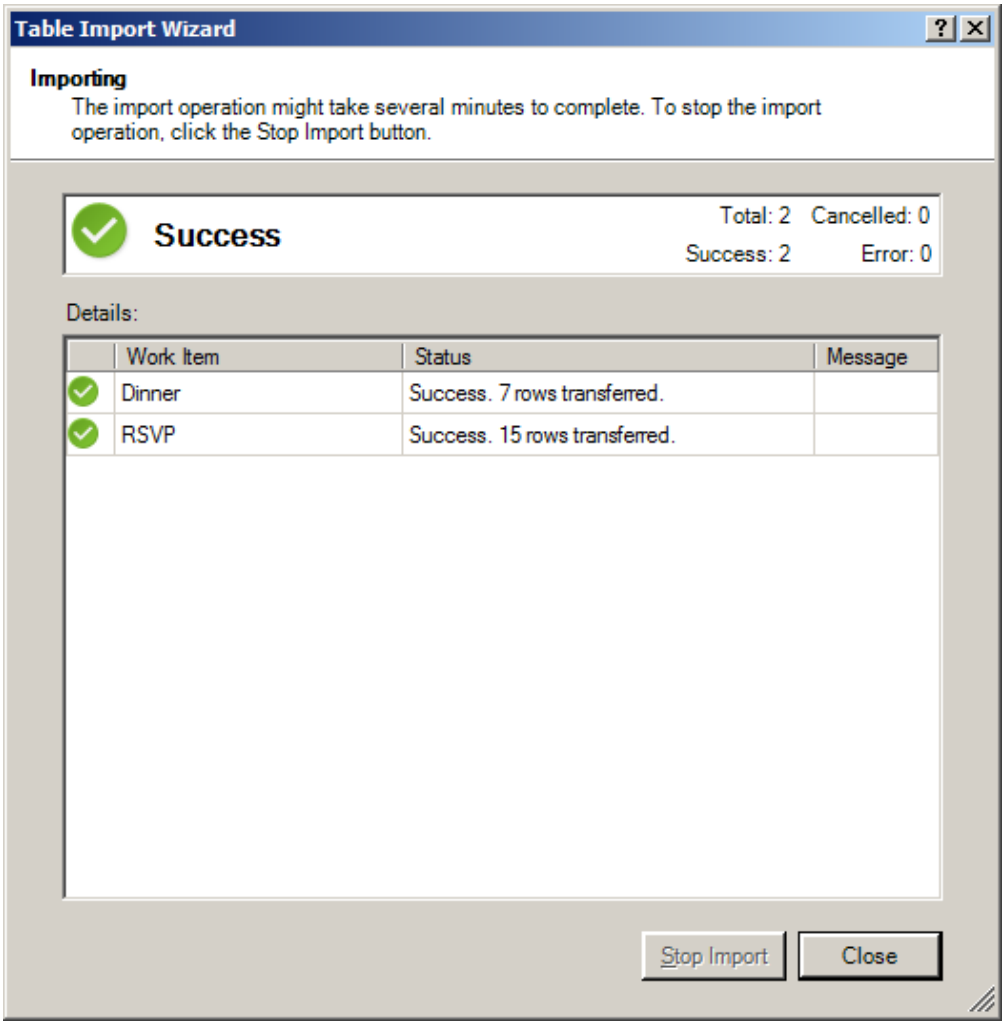
3. Add a name for your feed, and enter the URL of the OData service file for your BrightstarDB application.



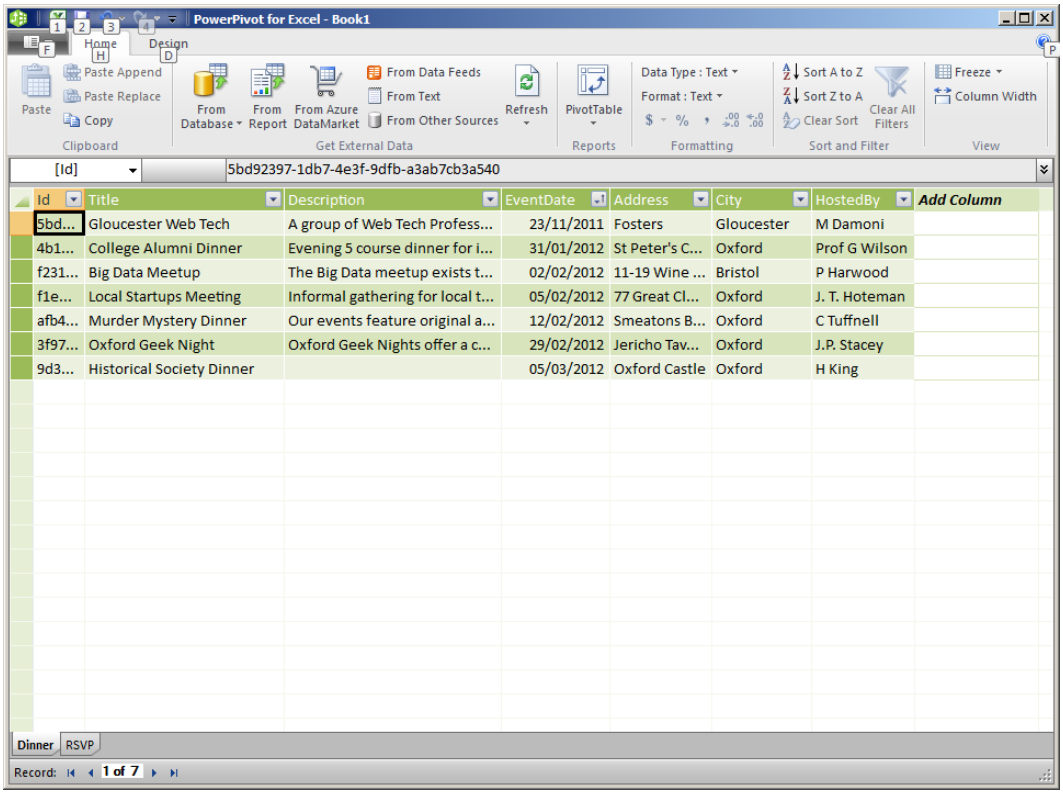
4. Click **Test Connection** to make sure that you can connect to your OData service and then click **Next**



1. Select the sets that you wish to consume and click **Finish**



1. This then shows all the data that is consumed from the OData service in the PowerPivot window. When any data is added or edited in the BrightstarDB store, the data in the PowerPivot windows can be updated by clicking the **Refresh** button.



10.3 Mapping to Existing RDF Data

Note: The source code for this example can be found in [INSTALLDIR]\Samples\EntityFramework\EntityFrameworkSamples.sln

One of the things that makes BrightstarDB unique is the ability to map multiple object models onto the same data and to map an object model onto existing RDF data. An example of this could be when some contact data in the RDF FOAF vocabulary is imported into BrightstarDB and an application wants to make use of that data. Using the BrightstarDB annotations it is possible to map object classes and properties to existing types and property types.

10.3.1 The following FOAF RDF triples are added to the data store.

```
<http://www.brightstardb.com/people/david> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http:/
<http://www.brightstardb.com/people/david> <http://xmlns.com/foaf/0.1/nick> "David" .
<http://www.brightstardb.com/people/david> <http://xmlns.com/foaf/0.1/name> "David Summers" .
<http://www.brightstardb.com/people/david> <http://xmlns.com/foaf/0.1/Organization> "Microsoft" .
<http://www.brightstardb.com/people/simon> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http:/
<http://www.brightstardb.com/people/simon> <http://xmlns.com/foaf/0.1/nick> "Simon" .
<http://www.brightstardb.com/people/simon> <http://xmlns.com/foaf/0.1/name> "Simon Williamson" .
<http://www.brightstardb.com/people/simon> <http://xmlns.com/foaf/0.1/Organization> "Microsoft" .
<http://www.brightstardb.com/people/simon> <http://xmlns.com/foaf/0.1/knows> <http://www.brightstardb
```

Triples can be loaded into the BrightStarDB using the following code::

```
var triples = new StringBuilder();
triples.AppendLine(@"<http://www.brightstardb.com/people/simon> <http://www.w3.org/1999/02/22-rdf-syn
triples.AppendLine(@"<http://www.brightstardb.com/people/simon> <http://xmlns.com/foaf/0.1/nick> ""Si
```

```
triples.AppendLine(@"<http://www.brightstardb.com/people/simon> <http://xmlns.com/foaf/0.1/name> ""Si
triples.AppendLine(@"<http://www.brightstardb.com/people/simon> <http://xmlns.com/foaf/0.1/Organizati
triples.AppendLine(@"<http://www.brightstardb.com/people/simon> <http://xmlns.com/foaf/0.1/knows> <ht
client.ExecuteTransaction(storeName, null, triples.ToString());
```

10.3.2 Defining Mappings

To access this data from the Entity Framework, we need to define the mappings between the RDF predictates and the properties on an object that represents an entity in the store.

The properties are marked up with the PropertyType attribute of the RDF predicate. If the property "Name" should match the predicate `http://xmlns.com/foaf/0.1/name`, we add the attribute `[PropertyType("http://xmlns.com/foaf/0.1/name")]`.

We can add a `NamespaceDeclaration` assembly attribute to the project's AssemblyInfo.cs file to shorten the URIs used in the attributes. The NamespaceDeclaration attribute allows us to define a short code for a URI prefix. For example:

```
[assembly: NamespaceDeclaration("foaf", "http://xmlns.com/foaf/0.1/")]
```

With this `NamespaceDeclaration` attribute in the project, the `PropertyType` attribute can be shortened to `[PropertyType("foaf:name")]`

The RDF example given above would be mapped to an entity as given below::

```
[Entity("http://xmlns.com/foaf/0.1/Person")]
public interface IPerson
{
    [Identifier("http://www.brightstardb.com/people/")]
    string Id { get; }

    [PropertyType("foaf:nick")]
    string Nickname { get; set; }

    [PropertyType("foaf:name")]
    string Name { get; set; }

    [PropertyType("foaf:Organization")]
    string Organisation { get; set; }

    [PropertyType("foaf:knows")]
    ICollection<IPerson> Knows { get; set; }

    [InversePropertyType("foaf:knows")]
    ICollection<IPerson> KnownBy { get; set; }
}
```

Adding the `[Identifier("http://www.brightstardb.com/people/")]` to the ID of the interface, means that when we can query and retrieve the Id without the entire prefix

10.3.3 Example

Once there is RDF data in the store, and an interface that maps an entity to the RDF data, the data can then be accessed easy using the Entity Framework by using the correct connection string to directly access the store.

```
var connectionString = "Type=rest;endpoint=http://localhost:8090/brightstar;StoreName=Foaf";
var context = new FoafContext(connectionString);
```

If you have added the connection string into the Config file:

```
<add key="BrightstarDB.ConnectionString"
     value="Type=rest;endpoint=http://localhost:8090/brightstar;StoreName=Foaf" />
```

Then you can initialise the content with a simple:

```
var context = new FoafContext();
```

For more information about connection strings, please read the *"Connection Strings" topic*

The code below connects to the store to access all the people in the RDF data, it then writes their name and place of employment, along with all the people they know or are known by.

```
var context = new FoafContext(connectionString);
var people = context.Persons.ToList();
var count = people.Count;
Console.WriteLine(@"{0} people found in raw RDF data", count);
Console.WriteLine();
foreach(var person in people)
{
    var knows = new List<IPerson>();
    knows.AddRange(person.Knows);
    knows.AddRange(person.KnownBy);


    Console.WriteLine(@"{0} ({1}), works at {2}", person.Name, person.Nickname, person.Organisation);
    Console.WriteLine(knows.Count == 1 ? string.Format(@"{0} knows 1 other person", person.Nickname)
                      : string.Format(@"{0} knows {1} other people", person.Nickname, knows.Count));
    foreach(var other in knows)
    {
        Console.WriteLine(@"    {0} at {1}", other.Name, other.Organisation);
    }
    Console.WriteLine();
}
```

CHAPTER 11

Advanced Entity Framework

The BrightstarDB Entity Framework has a number of built in extension points to enable developers to integrate their own custom SPARQL queries; to override the SPARQL query and update protocols used; or to change the way local C# types and properties are mapped to RDF types and properties.

11.1 Custom Queries

To Be Completed

11.2 Custom SPARQL Protocol

To Be completed

11.3 Custom Type Mappings

To be completed

CHAPTER 12

Data Object Layer

The Data Object Layer is a simple generic object wrapper for the underlying RDF data in any BrightstarDB store.

Data Objects are lightweight wrappers around sets of RDF triples in the underlying BrightstarDB store. They allow the developer to interact with the RDF data without requiring all information to be sent in N-Triple format.

For more information about the RDF layer of BrightstarDB, please read the *RDF Client API* section.

12.1 Creating a Data Object Context

The `IDataObjectContext` interface provides the methods for accessing BrightstarDB stores through the Data Object Layer. You can use this interface to list the available stores, to open existing stores and to create or delete stores. The following example shows how to create a new context using a connection string:

```
var context = BrightstarService.GetDataObjectContext("Type=rest;endpoint=http://localhost:8090/bright
```

The connection string defines the type of service you are connecting to. For the Data Object Context, the connection can be to an embedded instance of BrightstarDB; a connection to a BrightstarDB server over the HTTP REST interface; or a connection to another store using a DotNetRDF storage connector. For more information about connection strings, please refer to the section *Connection Strings*.

12.2 Using the IDataObjectContext

Once you have an `IDataObjectContext`, a new store can be creating using the `CreateStore` method:

```
IDataObjectStore myStore = context.CreateStore("MyStore");
```

`CreateStore` also accepts a number of optional parameters which are described in later sections.

Deleting a store is also straight forward - you just pass in the name of the store to be deleted:

```
context.DeleteStore("MyStore");
```

To check if a store with a particular name already exists, use the `DoesStoreExist()` method:

```
// Create MyStore if it doesn't already exist
if (!context.DoesStoreExist("MyStore")) {
  context.CreateStore("MyStore");
}
```

To open an existing store, use the `OpenStore()` method. This method will throw an exception if the named store does not exist, so it is a good idea to test for this first:

```
IDataObjectStore myStore;
if (context.DoesStoreExist("MyStore")) {
  myStore = context.OpenStore("MyStore");
}
```

12.3 Working With Data Objects

Data Objects can be created using the `MakeDataObject()` method on the IDataObjectStore:

```
var fred = store.MakeDataObject("http://example.org/people/fred");
```

The objects can be created by passing in a well formed URI as the identity, if no identity is given then one is automatically generated for it and can be accessed via its `Identity` property. A data object can be retrieved from the store using its URI identifier:

```
var fred = store.GetDataObject("http://example.org/people/fred");
```

If BrightstarDB does not hold any information about a given URI, then a data object is created for it and passed back. When the developer adds properties to it and saves it, the identity will be automatically added to BrightstarDB.

Note: `GetDataObject()` will never return a null object. The data object consists of all the information that is held in BrightstarDB for a particular identity.

To set the value of a single property, use the `SetProperty()` method. The method requires an IDataObject instance that defines the type of the property being added, so this needs to be created first.:

```
var name = store.MakeDataObject("http://xmlns.com/foaf/0.1/name");
fred.SetProperty(name, "Fred Evans");
```

There is also a short-hand version that takes care of creating the IDataObject for the type, so the following is equivalent to the previous two-line example:

```
fred.SetProperty("http://xmlns.com/foaf/0.1/name", "Fred Evans");
```

Calling `SetProperty()` a second time will overwrite the previous value of the property:

```
fred.SetProperty("http://xmlns.com/foaf/0.1/name", "Fred Q. Evans");
```

If you want to add multiple properties of the same type use the `AddProperty()` method instead of `SetProperty()`:

```
var mbox = store.MakeDataObject("http://xmlns.com/foaf/0.1/mbox");
fred.AddProperty(mbox, "fred@example.com");
fred.AddProperty(mbox, "fred.evans@example.com");
```

A property value can either be a literal primitive type (supported C# primitive types are string, bool, DateTime, Date, double, int, float, long, byte, decimal, short, ubyte, ushort, uint, ulong, char and byte[]), or another IDataObject instance:

```
var alice = store.MakeDataObject("http://example.org/people/alice");
var knows = store.MakeDataObject("http://xmlns.com/foaf/0.1/knows");
fred.AddProperty(knows, alice);
```

There is also a short-hand function for setting the RDF type property for a data object:

```
var person = store.MakeDataObject("http://xmlns.com/foaf/0.1/Person");
fred.SetType(person);
```

A property can be removed from a data object using the `RemoveProperty()` method:

```
fred.RemoveProperty(mbox, "fred@example.com");
```

`RemoveProperty()` will only remove a property that matches exactly by type and value (and language code if specified). Alternatively to remove all properties of a given type, use the `RemovePropertiesOfType()` method:

```
fred.RemovePropertiesOfType(mbox);
```

All of these methods for adding/remove properties and setting a type return the data object itself, allowing the calls to be chained:

```
fred.SetType(person)
    .SetProperty(name, "Fred Q. Evans")
    .AddProperty(mbox, "fred@example.org")
    .AddProperty(knows, alice);
```

Adding and removing properties and changing the type simply adds and removes triples from the set of locally managed triples for the data object.

To retrieve property values use either the `GetPropertyValues()` method to retrieve an enumerator over all of the property values for a specific property type; or use the `GetPropertyValue()` method that returns just the first value of a specific type. These methods both take either an `IDataObject` instance or a string to identify the property type:

```
// Get a data object for the property type we are intersted in
var name = store.MakeDataObject("http://xmlns.com/foaf/0.1/name");
// Write all names of fred
foreach (var n in fred.GetPropertyValues(name)) {
    Console.WriteLine(n);
}
// Write just one mbox of fred
Console.WriteLine(fred.GetProperty("http://xmlns.com/foaf/0.1/mbox));
```

To determine what properties a data object has, use the `GetPropertyTypes()` method to enumerate over the distinct types of property that a data object has. This can be useful for grouping together properties by type or for exploring / displaying data with an unknown schema behind it:

```
Console.WriteLine("Properties of Fred:");
foreach(var propertyType in fred.GetPropertyTypes()) {
    Console.WriteLine("\t" + propertyType.Identity + ":");
    foreach(var propertyValue in fred.GetPropertyValues(propertyType)) {
        Console.WriteLine("\t\t" + propertyValue);
    }
}
```

A data object can be deleted using the `Delete()` method on the data object itself:

```
var fred = store.GetDataObject("http://example.org/people/fred");
fred.Delete();
```

This will remove all triples describing that data object from the store when changes are saved.

Updates such as new properties, new objects and deletions are all tracked by the IDataObjectStore locally and are only applied to the BrightstarDB store when you call the `SaveChanges()` method on the store. `SaveChanges()` saves your changes in a single transaction, so either all updates will be applied to the store or the transaction will fail and none of the updates will be applied.

12.4 Namespace Mappings

Namespace mappings are sets of simple string prefixes for URIs, enabling the developer to use identities that have been shortened to use the prefixes.

For example, the mapping:

```
{"people", "http://example.org/people/"}
```

Means that the short string "people:fred" will be expanded to the full identity string "http://example.org/people/fred"

These mappings are passed through as a dictionary to the OpenStore() method on the context:

```
_namespaceMappings = new Dictionary<string, string>()
                         {
                             {"people", "http://example.org/people/"},
                             {"skills", "http://example.org/skills/"},
                             {"schema", "http://example.org/schema/"}
                         };
store = context.OpenStore(storeName, _namespaceMappings);
```

Note: It is best practise to set up a static dictionary within your class or configuration

12.5 Querying data using SPARQL

BrightstarDB supports SPARQL 1.1 for querying the data in the store. These queries can be executed via the Data Object store using the `ExecuteSparql()` method.

The SparqlResult returned has the results of the SPARQL query in the ResultDocument property which is an XML document formatted according to the SPARQL XML Query Results Format. The BrightstarDB libraries provide some helpful extension methods for accessing the contents of a SPARQL XML results document:

```
var query = "SELECT ?skill WHERE { " +
            "<http://example.org/people/fred> <http://example.org/schemas/person/skill> ?skill " +
            "}";
var sparqlResult = store.ExecuteSparql(query);
foreach (var sparqlResultRow in sparqlResult.ResultDocument.SparqlResultRows())
{
    var val = sparqlResultRow.GetColumnValue("skill");
    Console.WriteLine("Skill is " + val);
}
```

12.6 Binding SPARQL Results To Data Objects

When a SPARQL query has been written to return a single variable binding, it can be passed to the `BindDataObjectsWithSparql()` method. This executes the SPARQL query, and then binds each URI in the results to a data object, and passes back the enumeration of these instances:

```
var skillsQuery = "SELECT ?skill WHERE {?skill a <http://example.org/schemas/skill>}";
var allSkills = store.BindDataObjectsWithSparql(skillsQuery).ToList();
foreach (var s in allSkills)
{
    Console.WriteLine("Skill is " + s.Identity);
}
```

12.7 Optimistic Locking in the Data Object Layer

The Data Object Layer provides a basic level of optimistic locking support using the conditional update support provided by the RDF Client API and a special version property that gets assigned to data objects. Optimistic locking is enabled in one of two ways. The first option is to enable optimistic locking in the connection string used to create the `IDataObjectContext`:

```
var context = BrightstarService.GetDataObjectContext(
                  "type=rest;endpoint=http://localhost:8090/brightstar;optimisticLocking=true");
```

The other option is to enable optimistic locking in the `OpenStore()` or `CreateStore()` method used to retrieve the IDataObjectStore instance from the IDataObjectContext:

```
var store = context.OpenStore("MyStore", optimisticLockingEnabled:true);
```

Note: The optimisticLockingEnabled parameter of `OpenStore()` and `CreateStore()` is optional. If it is omitted, then the setting in the connection string for the IDataObjectContext is used. If it is specified, it always overrides the setting in the connection string.

With optimistic locking enabled, the Data Object Layer checks for the presence of a special version property on every object it retrieves (the property predicate URI is `http://www.brightstardb.com/.well-known/model/version`). If this property is present, its value defines the current version number of the property. If the property is not present, the object is recorded as being currently unversioned. On save, the Data Object Layer uses the current version number of all versioned data objects as the set of preconditions for the update, if any of these objects have had their version number property modified on the server, the precondition will fail and the update will not be applied. Also as part of the save, the Data Object Layer updates the version number of all versioned data objects and creates a new version number for all unversioned data objects.

When an concurrent modification is detected, this is notified to your code by a `TransactionPreconditionsFailedException` being raised. In your code you should catch this exception and handle the error. The `IDataObjectStore` interface provides a `Refresh()` method that implements two common approaches to handling this status. The `Refresh()` method takes two parameters: a data object instance and a `RefreshMode` parameter that specifies how the object is to be updated. `RefreshMode.StoreWins` overwrites any local modifications made to the object with the updated values held on the server. `RefreshMode.ClientWins` works the other way around, keeping the local changes and updating the version number for the locally tracked object so that the next time `SaveChanges()` is attempted the local changes will overwrite those held on the server. To find which objects need refreshing, the `IDataObjectStore` provides the `TrackedObjects` property that returns an enumerator over all the objects currently tracked by the store. Each IDataObject instance provides an `IsModified` property that is set to true if the store has some local changes for that object.

12.8 Graph Targeting in the Data Object API

You can use the Data Object API to update a specific named graph in the BrightstarDB store. Each time you open a store you can specify the following optional parameters:

- **`updateGraph`** [The identifier of the graph that new statements will be added to.] For connections to a BrightstarDB server, this defaults to the BrightstarDB default graph (`http://www.brightstardb.com/.well-known/model/defaultgraph`) For connections through the DotNetRDF connectors, the default graph will be store and service dependent.
- **`defaultDataSet`** [The identifier of the graphs that statements will be retrieved from.] For connections to a BrightstarDB server, this defaults to all graphs in the store. For connections through the DotNetRDF

connectors, the default data set will be store and service dependent.

- **versionGraph** [The identifier of the graph that contains version information for] optimistic locking. Defaults to the same graph as `updateGraph`.

These are passed as additional optional parameters to the `IDataObjectContext.OpenStore()` method.

To create a store that reads properties from the default graph and adds properties to a specific graph (e.g. for recording the results of inferences), use the following:

```
// Set storeName, prefixes and inferredGraphUri here
var store = context.OpenStore(storeName, prefixes, updateGraph:inferredGraphUri,
                              defaultDataSet: new[] {Constants.DefaultGraphUri},
                                                    versionGraph:Constants.DefaultGraphUri)
```

Note: Note that you need to be careful when using optimistic locking to ensure that you are consistent about which graph manages the version information. We recommend that you either use the BrightstarDB default graph (as shown in the example above) or use another named graph separate from the graphs that store the rest of the data (and define a constant for that graph URI).

To create a store that reads only the inferred properties use code like this:

```
// Set storeName, prefixes and inferredGraphUri here
var store = context.OpenStore(storeName, prefixes, updateGraph:inferredGraphUri,
                              defaultDataSet: new[] {inferredGraphUri},
                                                    versionGraph:Constants.DefaultGraphUri)
```

When creating a new store using the `IDataObjectContext.CreateStore()` method the `updateGraph` and `versionGraph` options can be specified, but the `defaultDataSet` parameter is not available as a new store will not have any graphs. In this case the store returned will read from and write to the graph specified by the `updateGraph` parameter.

12.8.1 Default Data Set

The `defaultDataSet` parameter can be used to list the URIs of the graphs that should be queried by the `IDataObjectStore` returned by the method. In SPARQL parlance, this set of graphs is known as the *dataset*. If an update graph or version graph is specified then those graph URIs will also be added to the data set.

In the special case that `updateGraph`, `versionGraph` and `defaultDataSet` are all NULL (or not specified in the call to `OpenStore`), and the connection being opened is a connection to a BrightstarDB store the default data set will be set to cover all of the graphs in the BrightstarDB store.

When connecting to other stores using the DotNetRDF connectors, the default data set will be defined by the server unless the `defaultDataSet` parameter is explicitly set.

12.8.2 Graph Targeting and Deletions

The `RemoveProperty()` and `SetProperty()` methods can both cause triples to be deleted from the store. In this case the triples are removed from both the graph specified by `updateGraph` and all the graphs specified in the `defaultDataSet` (or all graphs in the store if the `defaultDataSet` is not specified or is set to null).

Similarly if you call the `Delete` method on a DataObject, the triples that have the DataObject as subject or object will be deleted from the `updateGraph` and all graphs in the `defaultDataSet`.

CHAPTER 13

Dynamic API

The Dynamic API is a thin layer on top of the data object layer. It is designed to further ease the use of .NET with RDF data and to provide a model for persisting data in systems that make use of the .NET dynamic keyword. The .NET dynamic keyword and dynamic runtime allow properties of objects to be set at runtime without any prior class definition.

The following example shows how dynamics work in general. Both 'Name' and 'Type' are resolved at runtime. In this case they are simply stored as properties in the ExpandoObject.

```
dynamic d = new ExpandoObject();
d.Name = "Brightstar";
d.Type = "Product";
```

13.1 Dynamic Context

A dynamic context can be used to create dynamic objects whose state is persisted as RDF in BrightstarDB. To use the dynamic context a normal DataObjectContext must be created first. From the context a store can be created or opened. The store provides methods to create and fetch dynamic objects.

```
var dataObjectContext = BrightstarService.GetDataObjectContext();
// create a dynamic context
var dynaContext = new BrightstarDynamicContext(dataObjectContext);
// create a new store
var storeId = "DynamicSample" + Guid.NewGuid().ToString();
var dynaStore = dynaContext.CreateStore(storeId);
```

13.2 Dynamic Object

The dynamic object is a wrapper around the `IDataObject`. When a dynamic property is set this is translated into an update to the `IDataObject` and in turn into RDF. By default the name of the property is appended to the default namespace. By using namespace mappings any RDF vocabulary can be mapped. To use a namespace mapping, you must access / update a property whose name is the namespace prefix followed by __ (two underscore characters) followed by the suffix part of the URI. For example `object.foaf__name`.

If the value of the property is set to be a list of values then multiple triples are created, one for each value.

```
dynamic brightstar = dynaStore.MakeNewObject();
brightstar.name = "BrightstarDB";
```

```
// setting a list of values creates multiple properties on the object.
brightstar.rdfs__label = new[] { "BrightstarDB", "NoSQL Database" };

dynamic product = dynaStore.MakeNewObject();
// objects are connected together in the same way
brightstar.rdfs__type = product;
```

13.3 Saving Changes

The data updated in a context is persisted when `SaveChanges()` is called on the store object.:

```
dynaStore.SaveChanges();
```

13.4 Loading Data

After opening a store dynamic objects can be loaded via the `GetDataObject()` method or the `BindObjectsWithSparql()` method.

```
dynaStore = dynaContext.OpenStore(storeId);

// Retrieve a single object by its identifier
var object = dynaStore.GetDataObject(aUri);

// Use a SPARQL query that returns a single variable to return a collection of dynamic objects
var objects = dynaStore.BindObjectsWithSparql("select distinct ?dy where { ?dy ?p ?o }");
```

13.5 Sample Program

Note: The source code for this example can be found in [INSTALLDIR]\Samples\Dynamic\Dynamic.sln

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using BrightstarDB.Dynamic;
using BrightstarDB.Client;


namespace DynamicSamples
{
    public class Program
    {
        /// <summary>
        /// Assumes BrightstarDB is running as a service and exposing the
        /// default endpoint at http://localhost:8090/brightstar
        /// </summary>
        /// <param name="args"></param>
        static void Main(string[] args)
        {
            // gets a new BrightstarDB DataObjectContext
```

```
var dataObjectContext = BrightstarService.GetDataObjectContext();


// create a dynamic context
var dynaContext = new BrightstarDynamicContext(dataObjectContext);


// open a new store
var storeId = "DynamicSample" + Guid.NewGuid().ToString();
var dynaStore = dynaContext.CreateStore(storeId);


// create some dynamic objects.
dynamic brightstar = dynaStore.MakeNewObject();
dynamic product = dynaStore.MakeNewObject();


// set some properties
brightstar.name = "BrightstarDB";
product.rdfs__label = "Product";
var id = brightstar.Identity;


// use namespace mapping (RDF and RDFS are defined by default)
// Assigning a list creates repeated RDF properties.
brightstar.rdfs__label = new[] { "BrightstarDB", "NoSQL Database" };


// objects are connected together in the same way
brightstar.rdfs__type = product;


dynaStore.SaveChanges();


// open store and read some data
dynaStore = dynaContext.OpenStore(storeId);
brightstar = dynaStore.GetDataObject(brightstar.Identity);


// property values are ALWAYS collections.
var name = brightstar.name.FirstOrDefault();
Console.WriteLine("Name = " + name);


// property can also be accessed by index
var nameByIndex = brightstar.name[0];
Console.WriteLine("Name = " + nameByIndex);


// they can be enumerated without a cast
foreach (var l in brightstar.rdfs__label)
{
    Console.WriteLine("Label = " + l);
}


// object relationships are navigated in the same way
```

```
            var p = brightstar.rdfs__type.FirstOrDefault();
            Console.WriteLine(p.rdfs__label.FirstOrDefault());


            // dynamic objects can also be loaded via sparql
            dynaStore = dynaContext.OpenStore(storeId);
            var objects = dynaStore.BindObjectsWithSparql(
                                "select distinct ?dy where { ?dy ?p ?o }");
            foreach (var obj in objects)
            {
                Console.WriteLine(obj.rdfs__label[0]);
            }

            Console.ReadLine();
        }
    }
}
```

CHAPTER 14

RDF Client API

The RDF Client API provides a simple set of methods for creating and deleting stores, executing transactions and running queries. It should be used when the application needs to deal directly with RDF data. An RDF Client can connect to an embedded store or remotely to a running BrightstarDB instance.

14.1 Creating a client

The BrightstarService class provides a number of static methods that can be used to create a new client. The most general one takes a connection string as a parameter and returns a client object. The client implements the `BrightstarDB.IBrightstarService` interface.

The following example shows how to create a new service client using a connection string:

```
var client = BrightstarService.GetClient(
                  "Type=rest;endpoint=http://localhost:8090/brightstar;");
```

For more information about connection strings, please read the *"Connection Strings" topic*

14.2 Creating a Store

A new store can be creating using the following code:

```
string storeName = "Store_" + Guid.NewGuid();
client.CreateStore(storeName);
```

14.3 Deleting a Store

Deleting a store is also straight forward:

```
client.DeleteStore(storeName);
```

14.4 Jobs and IJobInfo

In BrightstarDB, many operations are executed as jobs. A job is simply an asynchronous task that is processed by the BrightstarDB server. BrightstarDB maintains a queue of jobs for each store and each store will process its jobs one at a time.

In the API, the methods that run as jobs all return an IJobInfo result. This interface defines a number of properties that can be used to check the status of a job.

Property Name	Description
JobId	The unique identifier for the job. This can be used with the `GetJobInfo` method to retrieve updates about the job as it is processed.
Label	A user-provided label for the job. This label can be set when the job is created.
JobPending	A boolean flag that is true if the job is currently queued for execution.
JobStarted	A boolean flag that is true if the job is currently being executed.
JobCompleted-WithErrors	A boolean flag that is true if the job has failed.
JobCompletedOk	A boolean flag that is true if the job has completed successfully.
QueuedTime	The date/time when the job was queued to be processed
StartTime	The date/time when the job entered processing.
EndTime	The date/time when the job completed processing.
ExceptionInfo	If an error occurred, this property exposed the detailed exception information.

Typically calling one of these methods simply queues the job for update and returns an IJobInfo structure straight away (the exceptions are the `ExecuteTransaction` and `ExecuteUpdate` methods which by default will only return when the job has completed successfully or failed).

You can monitor the progress of a job by making a call to the `GetJobInfo` method on the client. There are two variants of `GetJobInfo` the first takes a store name and a job ID and returns the status of that specific job. The second takes a store name, an offset and a length and returns the status of the jobs in that portion of the queue.

Note: Job status is not persisted by a store. This means that a server is restarted for any reason, all queued jobs and job information is lost. Additionally, job status records for completed (or failed) jobs may be periodically culled from the queue.

Therefore it is possible for `GetJobInfo` to fail to find the details of a previously submitted job in some circumstances.

14.5 Transactional Update

BrightstarDB supports a transactional update model that allows you to group together a collection of triples to remove and triples to add as a single atomic operation, that will either succeed and modify the store, or if it fails will leave the store unmodified.

A transaction is defined by creating a new instance of the `BrightstarDB.Client.UpdateTransactionData` class and setting its properties. The transaction is then executed by passing the `UpdateTransactionData` instance to the `ExecuteTransaction()` method on the client:

```
var transactionData = new UpdateTransactionData();

// ... set properties of transactionData here...

var jobInfo = client.ExecuteTransaction(storeName, transactionData);
```

By default the method will block until the job completes processing (either successfully or with errors). You can then check the value of the `IJobInfo` object returned for the job status and any exception details. Alternatively, you can pass `false` for the optional `waitForCompletion` parameter and the update job will be queued and the `IJobInfo` object returned straight away that you can then monitor asynchronously from your code. To provide a custom label for the job, you can pass the label string in to the optional `label` parameter.

14.5.1 Inserting Data

Data is added to the store by specifying the data to be added in N-Triples or N-Quads format on the `InsertData` property of the `UpdateTransactionData` class. Each triple or quad must be on a single line with no line breaks, a good way to do this is to use a `StringBuilder` and then using `AppendLine()` for each triple.

```
var addTriples = new StringBuilder();
addTriples.AppendLine("<http://www.brightstardb.com/products/brightstar> <http://www.brightstardb.com
addTriples.AppendLine("<http://www.brightstardb.com/products/brightstar> <http://www.brightstardb.com
addTriples.AppendLine("<http://www.brightstardb.com/products/brightstar> <http://www.brightstardb.com
addTriples.AppendLine("<http://www.brightstardb.com/products/brightstar> <http://www.brightstardb.com

var transactionData = new UpdateTransactionData { InsertData = addTriples };
```

The `ExecuteTransaction()` method is used to insert the data into the store:

```
var jobInfo = client.ExecuteTransaction(storeName, transactionData);
```

14.5.2 Deleting Data

Deletion is done by defining a pattern that should matching the triples to be deleted. The following example deletes the triple that asserts that BrightstarDB is in the product category NoSQL:

```
var deletePatterns = "<http://www.brightstardb.com/products/brightstar> <http://www.brightstardb.com/
var transactionData = new UpdateTransactionData { DeletePatterns = deletePatterns };
client.ExecuteTransaction(storeName, transactionData);
```

The identifier `http://www.brightstardb.com/.well-known/model/wildcard` is a wildcard match for any value, so the following example deletes all triples that have a subject of `http://www.brightstardb.com/products/brightstar` and a predicate of `http://www.brightstardb.com/schemas/product/category`:

```
var deletePatterns = "<http://www.brightstardb.com/products/brightstar> <http://www.brightstardb.com/
var transactionData = new UpdateTransactionData { DeletePatterns = deletePatterns };
var jobInfo = client.ExecuteTransaction(storeName, transactionData);
```

Note: The string `http://www.brightstardb.com/.well-known/model/wildcard` is also defined as the constant string `BrightstarDB.Constants.WildcardUri`.

14.5.3 Conditional Updates

The execution of a transaction can be made conditional on certain triples existing in the store. This is done by specifying the triples or triple patterns to be matched on the `ExistencePreconditions` property of the `UpdateTransactionData` class.

The following example updates the `productCode` property of a resource only if its current value is `640`:

```
var preconditions = new StringBuilder();
preconditions.AppendLine("<http://www.brightstardb.com/products/brightstar> <http://www.brightstardb
var deletes = new StringBuilder();
deletes.AppendLine("<http://www.brightstardb.com/products/brightstar> <http://www.brightstardb.com/sc
var inserts = new StringBuilder();
inserts.AppendLine("<http://www.brightstardb.com/products/brightstar> <http://www.brightstardb.com/sc
var transactionData = new UpdateTransactionData {
      ExistencePreconditions = preconditions.ToString(),
```

```
      DeletePatterns = deletes.ToString(),
      InsertData = inserts.ToString() };
client.ExecuteTransaction(storeName, transactionData);
```

When a transaction contains condition triples, every triple specified in the preconditions must exist in the store before the transaction is applied. If one or more triples specified in the preconditions are not matched, the update will not be applied.

In addition to being able to specify triple patterns that must exist in the store, it is also possible to specify patterns that MUST NOT exist before the update is applied. As with the existence preconditions, a failure

The following example adds a `productCode` property to a resource, only if the resource currently does not have a `productCode` property:

```
var preconditions = new StringBuilder();
preconditions.AppendLine("<http://www.brightstardb.com/products/brightstar> <http://www.brightstardb
var inserts = new StringBuilder();
inserts.AppendLine("<http://www.brightstardb.com/products/brightstar> <http://www.brightstardb.com/sc
var transactionData = new UpdateTransactionData {
    NonexistencePreconditions = preconditions.ToString(),
    InsertData = inserts.ToString() };
client.ExecuteTransaction(storeName, transactionData);
```

Existence and non-existence preconditions may both be specified on a transaction, both are checked before applying the update.

14.6 Data Types

In the code above we used simple triples to add a string literal object to a subject, such as:

```
<http://www.brightstardb.com/products/brightstar> <http://www.brightstardb.com/schemas/product/name>
```

Other data types can be specified for the object of a triple by using the ^^ syntax:

```
<http://www.brightstardb.com/products/brightstar> <http://www.brightstardb.com/schemas/product/produc
<http://www.brightstardb.com/products/brightstar> <http://www.brightstardb.com/schemas/product/releas
<http://www.brightstardb.com/products/brightstar> <http://www.brightstardb.com/schemas/product/cost>
```

14.7 Updating Graphs

The `ExecuteTransaction()` method on the `IBrightstarService` interface accepts a parameter that defines the default graph URI. When this parameters is specified, all precondition triples are tested against that graph; all delete triple patterns are applied to that graph; and all addition triples are added to that graph:

```
// This code update the graph http://example.org/graph1
client.ExecuteTransaction(storeName, preconditions, deletePatterns, additions, "http://example.org/gr
```

In addition, the format that is parsed for preconditions, delete patterns and additions allows you to mix N-Triples and N-Quads formats together. N-Quads are simply N-Triples with a fourth URI identifier on the end that specifies the graph to be updated. When an N-Quad is encountered, its graph URI overrides that passed into the `ExecuteTransaction()` method. For example, in the following code, one triple is added to the graph "http://example.org/graphs/alice" and the other is added to the default graph (because no value is specified in the call to `ExecuteTransaction()`:

```
var txn1Adds = new StringBuilder();
txn1Adds.AppendLine(
    @"<http://example.org/people/alice> <http://xmlns.com/foaf/0.1/name> ""Alice"" <http://example.or
txn1Adds.AppendLine(@"<http://example.org/people/bob> <http://xmlns.com/foaf/0.1/name> ""Bob"" .");
var result = client.ExecuteTransaction(storeName, null, null, txn1Adds.ToString());
```

Note: The wildcard URI is also supported for the graph URI in delete patterns, allowing you to delete matching patterns from all graphs in the BrightstarDB store.

14.8 Querying data using SPARQL

BrightstarDB supports SPARQL 1.1 for querying the data in the store. A simple query on the N-Triples above that returns all categories that the subject called "Brightstar DB" is connected to would look like this:

```
var query = "SELECT ?category WHERE { " +
        "<http://www.brightstardb.com/products/brightstar> <http://www.brightstardb.com/schemas/produ
        "}";
```

This string query can then be used by the `ExecuteQuery()` method to create an XDocument from the SPARQL results (See SPARQL XML Query Results Format for format of the XML document returned).

```
var result = XDocument.Load(client.ExecuteQuery(storeName, query));
```

BrightstarDB also supports several different formats for SPARQL results. The default format is XML, but you can also add a `BrightstarDB.SparqlResultsFormat` parameter to the `ExecuteQuery` method to control the format and encoding of the results set. For example:

```
var jsonResult = client.ExecuteQuery(storeName, query, SparqlResultsFormat.Json);
```

By default results are returned using UTF-8 encoding; however you can override this default behaviour by using the `WithEncoding()` method on the `SparqlResultsFormat` class. The `WithEncoding()` method takes a `System.Text.Encoding` instance and returns a `SparqlResultsFormat` instance that will ask for that specific encoding:

```
var unicodeXmlResult = client.ExecuteQuery(
                          storeName, query,
                          SparqlResultsFormat.Xml.WithEncoding(Encoding.Unicode));
```

SPARQL queries that use the CONSTRUCT or DESCRIBE keywords return an RDF graph rather than a SPARQL results set. By default results are returned as RDF/XML using a UTF-8 format, but this can also be overridden by passing in an `BrightstarDB.RdfFormat` value for the `graphFormat` parameters:

```
var ntriplesResult = client.ExecuteQuery(
                        storeName, query, // where query is a CONSTRUCT or DESCRIBE
                        graphFormat:RdfFormat.NTriples);
```

14.9 Querying Graphs

By default a SPARQL query will be executed against the default graph in the BrightstarDB store (that is, the graph in the store whose identifier is `http://www.brightstardb.com/.well-known/model/defaultgraph`). In SPARQL terms, this means that the default graph of the dataset consists of just the default graph in the store. You can override this default behaviour by passing the identifier of one or more graphs to the `ExecuteQuery()` method. There are two overrides of `ExecuteQuery()` that allow this. One accepts a single graph identifier as a

string parameter, the other accepts multiple graph identifiers as an IEnumerable<string> parameter. The three different approaches are shown below:

```
// Execute query using the store's default graph as the default graph
var result = client.ExecuteQuery(storeName, query);

// Execute query using the graph http://example.org/graphs/1 as
// the default graph
var result = client.ExecuteQuery(storeName, query,
                                 "http://example.org/graphs/1");

// Execute query using the graphs http://example.org/graphs/1 and
// http://example.org/graphs/2 as the default graph
var result = client.ExecuteQuery(storeName, query,
                                 new string[] {
                                                                    "http://example.org/graphs/1",
                                                                        "http://example.org/graphs/2"
```

Note: It is also possible to use the FROM and FROM NAMED keywords in the SPARQL query to define both the default graph and the named graphs used in your query.

14.10 Using extension methods

To make working with the resulting XDocument easier there exist a number of extensions methods. The method SparqlResultRows() returns an enumeration of XElement instances where each XElement represents a single result row in the SPARQL results.

The GetColumnValue() method takes the name of the SPARQL result column and returns the value as a string. There are also methods to test if the object is a literal, and to retrieve the data type and language code.

```
foreach (var sparqlResultRow in result.SparqlResultRows())
{
   var val = sparqlResultRow.GetColumnValue("category");
   Console.WriteLine("Category is " + val);
}
```

14.11 Update data using SPARQL

BrightstarDB supports SPARQL 1.1 Update for updating data in the store. An update operation is submitted to BrightstarDB as a job. By default the call to ExecuteUpdate() will block until the update job completes:

```
IJobInfo jobInfo = _client.ExecuteUpdate(storeName, updateExpression);
```

In this case, the resulting IJobInfo object will describe the final state of the update job. However, you can also run the job asynchronously by passing false for the optional waitForCompletion parameter:

```
IJobInfo jobInfo = _client.ExecuteUpdate(storeName, updateExpression, false);
```

In this case the resulting IJobInfo object will describe the current state of the update job and you can use calls to GetJobInfo() to poll the job for its current status.

14.12 Data Imports

To support the loading of large data sets BrightstarDB provides an import function. Before invoking the import function via the client API the data to be imported should be copied into a folder called 'import'. The 'import' folder should be located in the folder containing the BrigtstarDB store data folders. After a default installation the stores folder is [INSTALLDIR]\Data, thus the import folder should be [INSTALLDIR]\Data\import. For information about the RDF syntaxes that BrightstarDB supports for import, please refer to *Supported RDF Syntaxes*.

With the data copied into the folder the following client method can be called. The parameter is the name of the file that was copied into the import folder. You can optionally specify the default graph for the data to be imported into, and the format that the data is in. Import is executed asynchronously - the client will return an `IJobInfo` instance to allow you to monitor the progress of the job. See *Jobs and IJobInfo* for more information about managing asynchronous jobs.

Examples:

```
// Import the NTriples data into the default graph in "mystore"
var importJob = client.StartImport("mystore", "data.nt");

// Import the RDF/XML data into a specific graph in "mystore"
var importJob = client.StartImport("mystore", "data.rdf", "http://example.org/graphs/1", RdfFormat.Rd
```

14.13 Data Exports

BrightstarDB also supports the bulk export of triples from a store. You can choose either to export a single graph or all of the graphs in the store. You can choose to export in the formats listed in the table below.

For Single Graph	For Full Store
NTriples RDF/XML	NQuads

The exported data will be written to a file contained in the BrightstarDB import folder (the same path as used for data import). Export is executed asynchronously - the client will return an `IJobInfo` instance to allow you to monitor the progress of the job. See *Jobs and IJobInfo* for more information about managing asynchronous jobs.

Examples:

```
// Export all graphs in "mystore" in default NQuads format
var exportAllJob = client.StartExport("mystore", "data.nq");

// Export a single graph in RDF/XML format
var exportGraphJob = client.StartExport("mystore", "data.rdf",
                 "http://example.org/graph/1", RdfFormat.RdfXml);
```

14.14 Introduction To N-Triples

N-Triples is a consistent and simple way to encode RDF triples. N-Triples is a line based format. Each N-Triples line encodes one RDF triple. Each line consists of the subject (a URI), followed by whitespace, the predicate (a URI), more whitespace, and then the object (a URI or literal) followed by a dot and a new line. The encoding of the literal may include a datatype or language code as well as the value. URIs are enclosed in '<' '>' brackets. The formal definition of the N-Triples format can be found here.

The following are examples of N-Triples data:

```
# simple literal property
<http://www.brightstardb.com/products/brightstar> <http://www.brightstardb.com/schemas/product/name>


# relationship between two resources
<http://www.brightstardb.com/products/brightstar> <http://www.brightstardb.com/schemas/product/catego


# A property with an integer value
<http://www.brightstardb.com/products/brightstar> <http://www.brightstardb.com/schemas/product/produc

# A property with a date/time value
<http://www.brightstardb.com/products/brightstar> <http://www.brightstardb.com/schemas/product/releas

# A property with a decimal value
<http://www.brightstardb.com/products/brightstar> <http://www.brightstardb.com/schemas/product/cost>
```

Allowed Data Types

Data types are defined in terms of an identifier. Common data types use the XML Schema identifiers. The prefix of these is `http://www.w3.org/2001/XMLSchema#`. The common primitive datatypes are defined in the XML Schema specification.

14.15 Introduction To SPARQL

BrightstarDB is a triple store that implements the RDF and SPARQL standards. SPARQL, pronounced "sparkle", is the query language developer by the W3C for querying RDF data. SPARQL primarily uses pattern matching as a query mechanism. A short example follows:

```
PREFIX ont: <http://www.brightstardb.com/schemas/>
SELECT ?name ?description WHERE {
  ?product a ont:Product .
  ?product ont:name ?name .
  ?product ont:description ?description .
}
```

This query is asking for all the names and descriptions of all products in the store.

SELECT is used to specify which bound variables should appear in the result set. The result of this query is a table with two columns, one called "name" and the other "description".

The PREFIX notation is used so that the query itself is more readable. Full URIs can be used in the query. When included in the query directly URIs are enclosed by '<' and '>'.

Variables are specified with the '?' character in front of the variable name.

In the above example if a product did not have a description then it would not appear in the results even if it had a name. If the intended result was to retrieve all products with their name and the description if it existed then the OPTIONAL keyword can be used.

```
PREFIX ont: <http://www.brightstardb.com/schemas/>
SELECT ?name ?description WHERE {
  ?product a ont:Product .
  ?product ont:name ?name .

  OPTIONAL {
    ?product ont:description ?description .
```

```
    }
}
```

For more information on SPARQL 1.1 and more tutorials the following resources are worth reading.

1. SPARQL 1.1 Query Language
2. Introduction to RDF Query with SPARQL Tutorial

CHAPTER 15

Admin API

In addition to the APIs already covered for updating and querying stores, there are a number of useful administration APIs also provided by BrightstarDB. A Visual Studio solution file containing some sample applications that use these APIs can be found in [INSTALLDIR]/Samples/StoreAdmin.

15.1 Jobs

When a new job is passed to a store, the job information is added to a queue. Jobs are queued and executed in the order they are received. Through the BrightstarDB APIs you can retrieve that list of jobs and monitor the state of a given job.

15.1.1 IJobInfo

Job information retrieved from BrightstarDB is represented by an instance of the `BrightstarDB.Client.IJobInfo` interface. This interface exposes the following properties:

- `JobId`: The unique identifier for the job.
- `Label`: An optional user-friendly name for the job. The label is set by passing it in with the optional `label` parameter on methods that start a job.
- `JobPending`: A boolean flag. If true, the job is in the queue but has not yet been executed.
- `JobStarted`: A boolean flag. If true, the job is in the queue and is currently being executed.
- `JobCompletedWithErrors`: A boolean flag. If true, the processing of the job completed but the job itself failed for some reason. More information can be found by examining the `StatusMessage` and `ExceptionInfo` properties
- `JobCompletedOk`: A boolean flag. If true the job has completed processing successfully.
- `StatusMessage`: The current job status message. For some long-running jobs such as RDF import, this message will be updated as the job runs. For other types of job the status message may only be updated on completion or failure of the job.
- `ExceptionInfo`: If an error is raised internally as a job is run, the exception information wil be recorded in this property. The value is a `BrightstarDB.Dto.ExceptionDetailObject` which provides access to the exception type, message and any inner exceptions.
- `QueuedTime`: The date/time when the job was queued to be processed.
- `StartTime`: The date/time when the job started processing.

- `EndTime`: The date/time when the job completed processing.

Note: Timestamps are all provided in UTC and are serialized with a resolution of 1 second.

15.1.2 Retrieving the Jobs List

The method to retrieve a list of jobs from a store is `GetJobInfo(string storeName, int skip, int take)`. The `storeName` parameter specifies the name of the store to retrieve job information from. The `skip` and `take` parameters can be used for paging long lists of jobs. The return values is and enumerable of *IJobInfo* instances.

Note: The list of jobs maintained by the BrightstarDB server is not persistent. This means that the jobs list is reset whenever the server gets restarted so if you were to retrieve the list of jobs immediately after starting the server you would get an empty list.

15.1.3 Monitoring Individual Jobs

The methods in the BrightstarDB API that queue long running jobs all return an instance of the *IJobInfo* interface. To check on the status of a job, you can use the method `GetJobInfo(string storeName, string jobId)`. The `storeName` parameter is the name of the store that the job runs against and `jobId` is the unique identifier for the job (which is provided in the `JobId` property of the *IJobInfo* object). The return value of this method is an *IJobInfo* instance that represents the current state of the job.

Monitoring the status of a job is then a question of simply polling the server by calling the `GetJobInfo(string,string)` method until either the `JobCompletedOk` or `JobCompletedWithErrors` property on the returned *IJobInfo* instance gets set to true.

When polling status in this way you should be aware of the following:

1. Polling for status does require some (fairly minimal) server resources, so you should avoid polling in a very tight loop.
2. If the server gets reset before your job has a chance to execute, the job information will be lost and a `BrightstarClientException` will get thrown. In this case your code should either notify the user of the failure or you could opt to simply resubmit the job.

Note: Job IDs are assigned by the server using GUIDs so even if the server gets reset it is not possible to end up monitoring a different job with the same JobId.

15.2 Commit Points

Note: Commit Points are a feature that is only available with the Append-Only store persistence type. If you are accessing a store that uses the Rewrite persistence type, operations on a Commit Points are not supported and will raise a BrightstarClientException if an attempt is made to query against or revert to a previous Commit Point.

Each time a transaction is committed to a BrightstarDB store, a new commit point is written. Unlike a traditional database log file, a commit point provides a complete snapshot of the state of the BrightstarDB store immediately after the commit took place. This means that it is possible to query the BrightstarDB store as it existed at some previous point in time. It is also possible to revert the store to a previous commit

point, but in keeping with the BrightstarDB architecture, this operation doesn't actually delete the commit points that followed, but instead makes a new commit point which duplicates the commit point selected for the revert.

15.2.1 Retrieving Commit Points

The method to retrieve a list of commit points from a store is `GetCommitPoints()` on the `IBrightstarService` interface. There are two versions of this method. The first takes a store name and skip and take parameters to define a subrange of commit points to retrieve, the second adds a date/time range in the form of two date time parameters to allow more specific selection of a particular commit point range. The code below shows an example of using the first of these methods:

```
// Create a client - the connection string used is configured in the App.config file.
var client = BrightstarService.GetClient();
foreach(var commitPointInfo in client.GetCommitPoints(storeName, 0, 10))
{
   // Do something with each commit point
}
```

To avoid operations that return potentially very large results sets, the server will not return more than 100 commit points at a time, attempting to set the take parameter higher than 100 will result in an `ArgumentException` being raised.

The structures returned by the `GetCommitPoints()` method implement the `ICommitPointInfo` interface, this interface provides access to the following properties:

`StoreName` the name of the store that the commit point is associated with.

`Id` the commit point identifier. This identifier is unique amongst all commit points in the same store.

`CommitTime` the UTC date/time when the commit was made.

`JobId` the GUID identifier of the transaction job that resulted in the commit. The value of this property may be Guid.Empty for operations that were not associated with a transaction job (e.g initial store creation).

15.2.2 Querying A Commit Point

To execute a SPARQL query against a particular commit point of a store, use the overload of the `ExecuteQuery()` method that takes an `ICommitPointInfo` parameter rather than a store name string parameter:

```
var resultsStream = client.ExecuteQuery(commitPointInfo, sparqlQuery);
```

The resulting stream can be processed in exactly the same way as if you had queried the current state of the store.

15.3 Reverting The Store

Reverting the store takes a copy of an old commit point and pushes it to the top of the commit point list for the store. Queries and updates are then applied to the store as normal, and the data modified by commit points since the reverted one is effectively hidden.

This operation does not delete the commit points added since the reverted one, those commit points are still there as long as a Coalesce operation is not performed, meaning that it is possible to "re-revert" the store to its state before the revert was applied. The method to revert a store is also on the `IBrightstarService` interface and is shown below:

```
var client = BrightstarService.GetClient();
ICommitPointInfo commitPointInfo = ... ; // Code to get the commit point we want to revert to
client.RevertToCommitPoint(storeName, commitPointInfo); // Reverts the store
```

15.4 Consolidating The Store

Over time the size of the BrightstarDB store will grow. Each separate commit adds new data to the store, even if the commit deletes triples from the store the commit itself will extend the store file. The `ConsolidateStore()` operation enables the BrightstarDB store to be compressed, removing all commit point history. The operation rewrites the store data file to a shadow file and then replaces the existing data file with the new compressed data file and updates the master file. The consolidate operation blocks new writers, but allows readers to continue accessing the data file up until the shadow file is prepared. The code required to start a consolidate operation is shown below:

```
var client = BrightstarService.GetClient();
var consolidateJob = client.ConsolidateStore(storeName);
```

This method submits the consolidate operation to the store as a long-running job. Because this operation may take some time to complete the call does not block, but instead returns an `IJobInfo` structure which can be used to monitor the job. The code below shows a typical loop for monitoring the consolidate job:

```
while (!(consolidateJob.JobCompletedOk || consolidateJob.JobCompletedWithErrors))
{
   System.Threading.Thread.Sleep(500);
   consolidateJob = client.GetJobInfo(storeName, consolidateJob.JobId);
}
```

15.5 Creating Store Snapshots

From version 1.4, BrightstarDB now provides an API to allow you to create an independent snapshot of a store. A snapshot is an entirely separate store that contains a consolidated version of the data in the source store. You can use snapshots for a number of purposes, for example creating replicas for query or branching the data in a store to allow two different parallel modifications to the data.

The API for creating a store snapshot is quite simple:

```
var snapshotJob = client.CreateSnapshot(sourceStoreName, targetStoreName,
    persistenceType, commitPoint);
```

The `sourceStoreName` and `targetStoreName` parameters name the source for the snapshot and the store that will be created by the snapshot respectively. The store named by `targetStoreName` must not exist (the method will not overwrite existing stores). The `persistenceType` parameter can be one of `PersistenceType.AppendOnly` or `PersistenceType.Rewrite` and specifies the type of persistence used by the target store. The target store can use a different persistence type to the source store. The commitPointId parameter is optional. If it is not specified or if you pass null, the snapshot will be created from the most recent commit of the source store. If you want to create a snapshot from a previous commit of the source store, you can pass the `ICommitPointInfo` instance for that commit.

..note:

A snapshot can be created from a previous commit point only if the source store persistence type is `PersistenceType.AppendOnly`

15.6 Store Statistics

From version 1.4, BrightstarDB can now optionally maintain some basic triple-count statistics. The statistics kept are the total number of triples in the store, and the total number of triples for each distinct predicate. Statistics can be maintained automatically by the store or updated using an API call. As with transaction logs, BrightstarDB will maintain historical stats, allowing you to analyse the changes in a store over time if you wish.

15.6.1 Retrieving Statistics

The API provides two methods for retrieving statistics. To retrieve just the most recently generated statistics you can use code like this:

```
var client = BrightstarService.GetClient();
var stats = client.GetStatistics(storeName);
```

This method will return an `IStoreStatistics` instance which represents the most recent statistics for the store. The `IStoreStatistics` interface defines the following properties:

- CommitId and CommitTimestamp: The identifier and timestamp of the database commit that the statistics relate to. This information enables you to relate statistics to a commit point.
- TotalTripleCount: The total number of triples in the store
- PredicateTripleCounts: A dictionary of entries in which the key is a predicate URI and the value is the count of the number of triples using that predicate in the store.

If you want to analyse the changes in statistics over a period of time, there is an alternate method that retrieves multiple statistics records in one call:

```
DateTime fromDate = DateTime.UtcNow.Subtract(Timespan.FromDays(10));
DateTime toDate = DateTime.UtcNow();
IEnumerable<IStoreStatistics> allStats =
    client.GetStatistics(storeName, fromDate, toDate, 0, 100);
```

As you can see from the example above, this method takes a date range allowing you to select the period in time you want stats for. The final two parameters are a skip and take that is applied to the list of statistics after the date range filter. A BrightstarDB server will not return more than 100 statistics records at a time, so if your date range covers a period with more statistics in it than this you will need to make multiple calls using the skip and take parameters for paging.

15.6.2 Updating Statistics

Statistics can be updated automatically by the store if it is configured to do so (see the next section for details). However you can also use the API to request an update of the statistics. Statistics updates are processed as a long running job as for large stores the process may take some time:

```
IJobInfo statsUpdateJob = client.UpdateStatistics(storeName);
```

This method call will queue the update job and return a structure that you can use to poll until the job is completed (or you can simply call the method in a fire-and-forget manner).

15.6.3 Automatic Update of Statistics

The BrighstarDB server process can automatically update statistics. This is done by periodically queuing a job to update statistics. The period between updates is controlled by two configuration settings in the application configuration file for your BrightstarDB service (or other BrightstarDB application if you are using the embedded store).

The setting `BrightstarDB.StatsUpdate.Timespan` specifies the minimum number of seconds that must pass between executions of the statistics update job.

The setting `BrightstarDB.StatsUpdate.TransactionCount` specifies the minimum number of other transaction or update jobs that must be queued between executions of the statistics update job.

These conditions are only checked after a job is placed in the queue, so during quiet periods when there is no activity statistics will not be unnecessarily updated. Both conditions have to be met before a statistics update job will be queued. Normally it makes sense to set both of these properties to a non-zero value to ensure that both sufficient time has passed and sufficient changes have been made to the store to justify the overhead of running a statistics update. However, you can set either one of these properties to zero (which is the default value) to only take account of the other. Setting both of these configuration properties to zero (or leaving them out of the configuration file) results in automatic statistics update being disabled.

CHAPTER 16

Concurrency and Multi-threading

This section covers the use of BrightstarDB in a variety of concurrent-access and multi-threading scenarios and describes BrightstarDB's

16.1 Concurrent Access to Stores

A BrightstarDB service of type "embedded" assumes that it has sole access to the store data files and the stores directory. You should not attempt to run two embedded instances of BrightstarDB concurrently with the same stores directory. If you want multiple applications to concurrently access the same collection of BrightstarDB stores you should instead run a BrightstarDB service that provides access to the store and then change you applications to use the "rest" connection string and connect to the server.

On a single store, BrightstarDB supports single-threaded writes and multi-threaded reads. Write operations are serialized (and are executed in the order that they are received), with read operations being executed in parallel with the writes. The isolation level for reads is "read committed" - in other words a read will see the state of the last successful commit of the store, even if a write is in progress or if a write starts while the read is being executed.

Warning: The current re-writeable store implementation is not structured to hold on to commit points while reads are being executed. If a single read operation spans multiple write operations, the commit point that the read is using will be removed from the store. If this happens, the read request is automatically retried using the latest commit point.
This scenario never occurs with the append-only store implementation as that store structure is designed to keep all previous commits.

16.2 Thread-safety

All implementations of IBrightstarService are thread-safe, this means you can use the low-level *RDF API* safely in a multi-threaded application. However, the IDataObjectContext, IDataObjectStore and the Entity Framework contexts are not. Multi-threaded applications that use either the *Data Objects API* or the *Entity Framework* should ensure that each thread uses its own context and store instance for these API calls.

CHAPTER 17

Developing Portable Apps

BrightstarDB provides support for a restricted set of platforms through the Windows Portable Class library. The set of platforms has been restricted to ensure that all of the supported platforms support the complete set of BrightstarDB features, including the *Data Object Layer* and the *Entity Framework* as well as the *RDF Client API*.

17.1 Supported Platforms

The BrightstarDB Portable Class Library supports the following platforms:

- .NET 4.5
- Silverlight 5
- Windows Phone 8
- Windows Store Apps
- Android
- iOS

17.2 Including BrightstarDB In Your Project

The BrightstarDB Portable Class Library is split into two parts; a core library and a platform-specific extension library for each supported platform. The extension library provides file-system access methods for the specific platform. In most cases both DLLs are required.

17.2.1 Using BrightstarDB from NuGet

If you are writing a Portable Library DLL, you need only include the BrightstarDB or BrightstarDBLibs package. Your DLL will have to target the same platforms that BrightstarDB supports (see above) or a sub-set of them.

If you are writing an application, you need to include either BrightstarDB or BrightstarDBLibs for the core library and then you must also add the BrightstarDB.Platform package which will install the correct extension library for your application platform.

17.2.2 Using BrightstarDB from Source

The main portable class library build file is the file srcportableportable.sln. Building this solution file will build the Portable Class Library and all of the platform-specific extension libraries. You should then include the BrightstarDB.Portable.DLL and one of the following extension DLLs:

- BrightstarDB.Portable.Desktop.DLL for .NET 4.5 applications
- BrightstarDB.Portable.Phone.DLL for Windows Phone 8 applications
- BrightstarDB.Portable.Silverlight.DLL for Silverlight 5 applications
- BrightstarDB.Portable.Store.DLL for Windows store applications.
- BrightstarDB.Portable.Android for Android applications.
- BrightstarDB.Portable.iOS for iOS applications.

Alternatively you can include just the relevant project files in your application solution and build them as part of your application build.

17.3 API Changes

There are some minor differences between the APIs provided by the Portable Class Library build and the other builds of BrightstarDB.

1. The IBrightstarService.ExecuteTransaction and IBrightstarService.ExecuteUpdate methods do not support the optional *waitForCompletion* parameter. These methods will always return without waiting for completion of the transaction / update and you must write code to monitor the job until it completes.
2. The configuration options detailed in :ref:<BrightstarDB_Configuration_Options> are not supported as there is no common interface for accessing application configuration information. Instead you can set the static properties exposed by the *BrightstarDB.Configuration* class at run-time (see the API documentation for details).

17.4 Platform Notes

Due to the differences in storage model, the different platforms behave slightly differently in where they expect / allow BrightstarDB stores to be created and accessed from.

17.4.1 Desktop

Paths to BrightstarDB stores are resolved relative to the applications working directory. It is possible to create / read stores in any file location accessible to the user that the application runs as.

17.4.2 Phone and Silverlight

All BrightstarDB stores are created in Isolated storage under the user-scoped storage for the application (the store returned by *IsolatedStorageFile.GetUserStoreForApplication()*). Any path you specify for a store location will be resolved relative to this isolated storage root. It is not possible to create or access a BrightstarDB store under any other location.

17.4.3 Windows Store

For Windows Store applications paths are resolved relative to the user-scoped local storage folder for the application (the folder returned by *ApplicationData.Current.LocalFolder*). It is not possible to create or access a BrightstarDB store under any other location.

17.4.4 Android

Android support is in the early stages of development and should be considered experimental.

Please note the following when using BrightstarDB from within an Android application.

1. The package targets Android API Level 10.
2. Due to limited resources the code has only been tested on the Google emulators. It has not yet been tested on any Android hardware.
3. The REST client is not tested and not supported yet.
4. Ensure that the StoresDirectory property of your embedded client connection string specifies a path that your application can write to. The persistence layer used will use the System.IO classes in Mono, not IsolatedStorage, so you need to be careful to provide a path that Android will allow your application to read from and write to (including creating subdirectories and files).
5. As there is no easy way to use app.config from any PCL application, we recommend that you explicitly set the BrightstarDB.Configuration class properties when your application starts up.
6. Query is not currently optimized for devices with small amounts of memory. SPARQL queries can vary quite widely in their runtime memory footprint depending both on how the query is written and on the size of data being queried. We plan on addressing the amount of memory used by SPARQL query processing in a future release.

OK, that is a lot of caveats, but we would really welcome one or two brave souls trying this out in a test Android application and giving us some feedback.

17.4.5 iOS

iOS support is in the early stages of development and should be considered experimental.

Please note the following when using BrightstarDB from within an Android application.

1. The code has only been tested on the Apple iOS emulators. It has not yet been tested on any iOS hardware.
2. The REST client is not tested and not supported yet.
3. Ensure that the StoresDirectory property of your embedded client connection string specifies a path that your application can write to. The persistence layer used will use the System.IO classes in Mono, so you need to be careful to provide a path that Android will allow your application to read from and write to (including creating subdirectories and files). We recommend using a sub-folder within the Library folder for your app.
4. As there is no easy way to use app.config from any PCL application, we recommend that you explicitly set the BrightstarDB.Configuration class properties when your application starts up.
5. Query is not currently optimized for devices with small amounts of memory. SPARQL queries can vary quite widely in their runtime memory footprint depending both on how the query is written and on the size of data being queried. We plan on addressing the amount of memory used by SPARQL query processing in a future release.

17.5 BrightstarDB Database Portability

All builds of BrightstarDB use exactly the same binary format for their data files. This means that a BrightstarDB store created on any of the supported platforms can be successfully opened and even updated on any other platform as long as all of the files are copied retaining the original folder structure.

17.6 Acknowledgment

We would like to thank Xamarin for providing the BrightstarDB project with a license for their Xamarin.Android and Xamarin.iOS products - without them we wouldn't be able to continue to develop and support those branches branch of the code!

CHAPTER 18

Connecting to Other Stores

BrightstarDB provides a set of high-level APIs for working with RDF data that we think are useful regardless of what underlying store you used to manage the RDF data. For that reason we have done some work to enable the use of the *Data Object Layer*, *Dynamic API* and *Entity Framework* with stores other than BrightstarDB.

If your store provides SPARQL 1.1 query and update endpoints that implement the SPARQL 1.1 protocol specification you can use a SPARQL connection string. For other stores you can use DotNetRDF's configuration syntax to configure a connection to the store.

18.1 Store Requirements

To use this functionality in BrightstarDB, the store must support query using SPARQL 1.1 Query and update using SPARQL 1.1 Update. The store must also have a connector available for it in DotNetRDF, or support SPARQL 1.1 Protocol.

A number of commercial and non-commercial stores are supported by DotNetRDF (for a list please refer to the DotNetRDF documentation).

18.2 Configuration and Connection Strings

The connection to your store must be configured by providing an RDF file that contains the configuration information. We use the DotNetRDF Configuration API and load the configuration from a file whose path you specify in the connection string (refer to *Connection Strings* for more details).

In the DotNetRDF configuration file you need to either configure one or more Storage Providers or a **single** Storage Server (at the time of writing the configuration for these has not been documented in the DotNetRDF project).

18.2.1 Using Storage Providers

This approach provides a flexible way to make one or more RDF data stores accessible via the BrightstarDB APIs. You must create a DotNetRDF configuration file that contains the configuration for each of the stores you want to access. Each store must be configured as a DotNetRDF StorageProvider.

Each configuration you create should have a URI identifier assigned to it (so don't use a blank node for the configuration in the configuration file). The full URI of the configuration resource is used as the store name in calls to DoesStoreExist() or OpenStore(). For a shorter store name it is also possible to use a relative URIs - these will be resolved against the base URI of the configuration graph.

The connection string you use for BrightstarDB is then just:

```
type=dotnetrdf;configuration={configuration_file_path}
```

where `configuration_file_path` is the full path to the DotNetRDF configuration file.

18.2.2 Using A StorageServer

DotNetRDF supports connections to Sesame and to Stardog servers that manage multiple stores. These connections must be configured as a DotNetRDF StorageServer. In this case, the list of stores is managed by the storage server so you don't need to write a separate configuration for each individual store on the server.

The configuration you create must have a URI identifier assigned to it. The full URI of this configuration resource is used in the connection string.

The connection string you would use for BrightstarDB in this scenario follows this template:

```
type=dotnetrdf;configuration={config_file_path};storageServer={config_uri}
```

where `config_file_path` is the full path to the DotNetRDF configuration file, and `config_uri` is the URI identifier of the configuration resource for the storage server.

18.2.3 Using SPARQL endpoints

If the data store you want to connect to supports SPARQL 1.1 Query and Update and the SPARQL 1.1 Protocol specification, then you can create a connection that will use the SPARQL query and update endpoints directly. The template for this type of connection string is:

```
type=sparql;query={query_endpoint_uri};update={update_endpoint_uri}
```

where `query_endpoint_uri` is the URI of the SPARQL query endpoint for the server and `update_endpoint_uri` is the URI of the SPARQL update endpoint.

You can omit the `update=` part of the connection string, in which case the connection will be a read-only one and calls to `SaveChanges()` will result in a runtime exception.

If credentials are required to access the server, these can be passed in using optional `userName=` and `password=` parameters:

```
type=sparql;query=...;update=...;userName=joe;password=secret123
```

18.3 Differences to BrightstarDB Connections

We have tried to keep the differences between using BrightstarDB and using another store through the high-level APIs to a minimum. However as there are many differences between different store implementations, so there are a few points of potential difference to be aware of:

1. **Default dataset and update graph.** If not overridden in code, the default dataset for a BrightstarDB connection is all graphs in the store; for another store the default dataset is defined by the server. Similarly if not overridden in code, the default graph for updates on a BrightstarDB connection is the BrightstarDB default graph (a graph with the URI `http://www.brightstardb.com/.well-known/model/defaultgraph`); for another store, the default graph for updates is defined by the server.

   To minimize confusion it is a good idea to always explicitly specify the update graph and default data set when your code may connect to stores other than BrightstarDB and to ensure that the update graph is included in the default data set.

2. **Optimistic locking** This is currently unsupported for connections to stores other than BrightstarDB as its implementation depends on functionality not available in SPARQL 1.1 protocol.
3. **Transactional Updating** This is highly dependent on the way in which the store's SPARQL update implementation works. The code will send a set of SPARQL update commands in a single request to the store. If the store does not implement the processing such that the multiple updates are handled in a single transaction, then it will be possible to end up with partially completed updates. It is worth checking with the documentation for your store / endpoint to see what transactional guarantees it makes for SPARQL Update requests.

18.4 Example Configurations

18.4.1 Connecting over SPARQL Protocol

DotNetRDF configuration file (dotNetRdf.config.ttl):

```
@prefix dnr: <http://www.dotnetrdf.org/configuration#> .
@prefix : <http://example.org/configuration#> .

:sparqlQuery a dnr:SparqlQueryEndpoint ;
    dnr:type "VDS.RDF.Query.SparqlRemoteEndpoint" ;
    dnr:queryEndpointUri <http://example.org/sparql> .

:sparqlUpdate a dnr:SparqlUpdateEndpoint ;
    dnr:type "VDS.RDF.Update.SparqlRemoteUpdateEndpoint" ;
    dnr:updateEndpointUri <http://example.org/update> .
```

connection string:

```
type=dotnetrdf;configuration=c:\path\to\dotNetRdf.config.ttl;query=http://example.org/configuration#s
```

18.4.2 Connecting to a Fuseki Server

DotNetRDF configuration file (dotNetRdf.config.ttl):

```
@prefix dnr: <http://www.dotnetrdf.org/configuration#>
@prefix : <http://example.org/configuration#>

:fuseki a dnr:StorageProvider ;
    dnr:type "VDS.RDF.Storage.FusekiConnector" ;
    dnr:server "http://fuseki.example.org/dataset/data" .
```

connection string:: type=dotnetrdf;configuration=c:pathtodotNetRdf.config.ttl;store=http://example.org/configuration#fuseki

TBD: More examples

CHAPTER 19

API Documentation

The API documentation for the full set of classes and methods in the latest release of BrightstarDB can be found in the BrightstarDB API Docs online or in the BrightstarDB_API.chm file that can be found in the Docs directory of your installation.

CHAPTER 20

BrightstarDB Security

This section covers the topic of BrightstarDB server security from multiple viewpoints. The security features of BrightstarDB are basic but designed to be customizable to fit with different schemes of user authentication and authorization.

20.1 Access Control

Note: Access controls for BrightstarDB services is a work in progress. Previous releases had no form of access control and there is much work to complete to reach the desired state. Rather than wait until everything is all completed, this release provides the framework for access controls and future releases will build on that framework to deliver incremental increases in functionality. Comments and suggestions for improvements in this area are most welcome.

20.1.1 Store Permissions

BrightstarDB is secured at the store level. A user that has read access to a store has read access to all the data in the store. A user with the required update privileges can update or delete any of the triples in the store. The permissions for a user on a store can be any combination of the following:

None The user has no permissions on the store and can perform no operations on it at all

Read The user has permission to perform SPARQL queries on the store

Export The user can run an export job to retrieve a dump of the RDF contained in the store

ViewHistory The user can view the commit and transaction history of the store

SparqlUpdate The user can post updates to the store using the SPARQL update protocol

TransactionUpdate The user can post updates to the store using the BrightstarDB transactional update protocol

Admin The user can re-execute previous transactions; revert the store to a previous transaction; and delete the store

WithGrant The user can grant permissions on this store to other users

All A combination of all of the above permissions

20.1.2 System Permissions

In addition to permissions on individual stores, users can also be assigned permissions on the BrightstarDB server as a whole. These permissions control only the ability to list, create and delete stores. The system permissions for a user can be any combination of the following:

None The user has no system permissions. This level denies even the listing of the stores currently available on the server.

ListStores The user can list the stores available on the server. Note that the listing is not currently filtered by store access permissions, so the user will see all stores regardless of whether or not they have any permission to access the stores.

CreateStore The user can create new stores on the server.

Admin The user can delete stores from the server regardless of whether they have permissions to administer the individual stores themselves.

All A combination of all the above permissions.

20.2 Authentication

User authentication is the responsibility of the host application for the BrightstarDB service. There are several different approaches which can be taken to user authentication for a REST-based API and the structure of BrightstarDB enables you to plug or leverage the form of authentication that works best for your solution.

20.2.1 Credential-based Authentication

If the BrightstarDB service is hosted under IIS, you can use IIS Basic Authentication or Windows Authentication to protect the service. This requires that the client provides credentials and that those credentials are checked for each request made. If the credentials are valid, the user identity for the request will be set to the identity associated with the credentials. If the credentials are invalid, the request will be rejected without further processing.

20.2.2 Shared Secret Authentication

An alternative to id/password credentials is to use a shared secret key mechanism. The BrightstarDB server and client share a secret key which is used to sign requests. The key is provided to the client by some mechanism outside of the BrightstarDB service itself (e.g. you might email the key or provide a separate web endpoint for requesting and providing keys). Each secret key is associated with an account ID. The requestor includes their account ID in the request and then signs the content of the request using their secret key. The server checks for the account ID, and validates the signature on the request using the same key. If the signature is valid, the identity for the request is set to the identity associated with the account ID. If the signature is not valid, the request is rejected without further processing.

Note: Currently this form of authentication is not yet implemented on the server. It is planned to add support for this in a future release and to provide a simple service for managing account/secret pairs in a BrightstarDB store so that it is easy to integrate key generation and management into an existing site.

20.3 Authorization

BrightstarDB has an extensible solution for the task of determining the precise permissions of a specific user. Permission Providers are classes that are responsible for returning the permission flags for a user.

Store Permission Providers determine the permissions for a given user (or the anonymous user) on a given store. System Permission Providers determine the permissions for a given user on the BrightstarDB server.

Possible means of determining the permissions for a user include:

Fixed Permission Levels All users have the same level of access to all stores. A variation of this specifies on set of permissions for authenticated users and another set of permissions for anonymous users.

Statically Configured Permission Levels Users are assigned permissions from a master list of permissions. This master list might be kept in a file or in a BrightstarDB store. Either way the permissions list needs to be manually updated when new stores are created or users are added to or removed from the system.

Alternatively permissions can be statically assigned to roles. Authenticated users are associated with one or more roles and receive permissions based on adding together all the permissions of all of their roles. This requires that the authentication system be capable of returning a set of roles for an authenticated user.

Dynamically Configured Permission Levels Users or roles are assigned permissions from a master list of permissions kept in a BrightstarDB store. These permissions can be updated through the BrightstarDB Admin API.

Note: Currently only support for Fixed Permission Levels is implemented. Support for the other forms of authentication will be added in forthcoming releases.

CHAPTER 21

Running BrightstarDB

BrightstarDB can be used as an embedded database or accessed via HTTP(S) as a RESTful web service. The REST service can be hosted in a number of different ways or it can be run directly from the command-line.

21.1 Namespace Reservation

The BrightstarDB server requires permission from the Windows system to start listening for connections on an HTTP port. This permission must be granted to the user that the service runs as. When the BrightstarDB server is run as a service, this will be the service user. When the BrightstarDB server is run from a command line, it will be the user who starts the command line shell.

To grant users the permission to listen for connections on a particular endpoint you must run the `http add urlacl` command in command prompt with elevated (Administrator) permissions.

If you use the default port and path for the BrightstarDB service, the following command will grant all users the required permissions to start the service:

```
netsh http add urlacl url=http://+:8090/brightstar/ user=Everyone
```

note: The BrightstarDB installer will automatically make the required reservation for running the BrightstarDB server as a Windows service using the default port (8090) and path (/brightstar/)

note: If you chose to host BrightstarDB in IIS or another web application host then the URL reservation will not be required as IIS (or the other host application) should manage this on your behalf.

21.2 Running BrightstarDB as a Windows Service

The installer will create a windows service called "BrightstarDB". This exposes a RESTful HTTP service endpoint that can be used to access the database. The configuration for this service can be found in *BrightstarService.exe.config* in the *[INSTALLDIR]\Service* folder.

21.3 Running BrightstarDB as an Application

Running the service as an application rather than a Windows service can be done by running the *BrightstarService.exe* located in the *[INSTALLDIR]\Service* folder. The configuration from the *BrightstarService.exe.config* file is used by the service when it starts up. However, some properties can also be overridden using command line parameters passed to the service. The format of the command-line is as follows:

```
BrightstarService.exe [options]
```

Where `options` are:

/c, /ConnectionString Provides the connection string used by the service to access the BrightstarDB stores. Typically this connection string should be an **embedded** connection string, but it is not a requirement. If this option is specified on the command-line it overrides any setting contained in the application configuration file. If this option is not specified on the command-line then a value MUST be provided in the the application configuration file.

/r, /RootPath Specifies the full file path to the directory containing the *Views* and *assets* folder for the service. The default path used is the path to the directory containing the BrightstarService.exe file itself. This should only need to be overridden in development environments where it can be used to serve views/assets directly from the source folders rather than from the bin directory.

/u, /ServiceUri Specifies the base URI path that the service will listen on for connections. This parameter can be repeated multiple times to create a service that will listen on multiple endpoints. The default value is "http://localhost:8090/brightstar/"

21.4 Running BrightstarDB In IIS

BrightstarDB can be hosted as a .NET 4.0 web application in IIS. If you have installed BrightstarDB from the installer, you will find a pre-built version of the web application in the *INSTALLDIR\webapp* directory.

You will need to ensure that the application pool that the web application runs under has the necessary privileges to access the directory where the BrightstarDB stores are kept. It is strongly advised that this directory should be outside the directory structure used for the IIS website itself.

For a step-by-step guide please refer to *Running BrightstarDB in IIS*

21.5 Running BrightstarDB in Docker

From the 1.8 release we now provide pre-built Docker images to run the BrightstarDB service. Docker is an open platform for developers and sysadmins to build, ship and run distributed applications, whether on laptops, data center VMs, or the cloud.

The BrightstarDB Docker images are built on the most recent Ubuntu LTS and the most recent Mono stable release. The Dockerfile and other configuration files can be found in our Docker repository on GitHub where you will also find important information about how to configure and run the Docker images.

21.6 BrightstarDB Service Configuration

The BrightstarDB server can also be configured from its application configuration file (or web.config when hosted in IIS). This is achieved through a custom configuration section which must be registered. This custom configuration section grants far more control over the configuration of the service than the command line parameters and is the recommended way of configuring the BrightstarDB service.

The sample below shows a skeleton application configuration file with just the BrightstarDB configuration shown:

```
<configuration>
  <configSections>
    <section name="brightstarService" type="BrightstarDB.Server.Modules.BrightstarServiceConfiguratio
  </configSections>
```

```
  <brightstarService connectionString="type=embedded;StoresDirectory=c:\brightstar">
    <storePermissions>
      <passAll anonPermissions="All"/>
    </storePermissions>
    <systemPermissions>
      <passAll anonPermissions="All"/>
    </systemPermissions>
  </brightstarService>

</configuration>
```

Note that the configuration section must first be registered in the *configSections* element so that the correct handler is invoked. The section itself consists of the following elements and attributes:

brightstarService This is the root element for the configuration. It supports a number of attributes (documented below) and contains one or zero *storePermissions* elements and one or zero *systemPermissions* elements.

brightstarService/@connectionString This attribute specifies the connection string that the BrightstarDB service will use to connect to the stores it serves. The attribute value must be a valid BrightstarDB connection string. Typically the connection type will be embedded, but this is not required. See the section *Connection Strings* for more information about the format of BrightstarDB connection strings.

storePermissions This element is the root element for configuring the way that the BrightstarDB service manages store access permissions. See *Configuring Store Permissions* for more details.

systemPermissions This element is the root element for configuring the way that the BrightstarDB service manages system access permissions.

21.6.1 Configuring Store Permissions

When a user attempts to read or write data in a BrightstarDB store, the Store Permissions for that user are checked to ensure that the user has the required privileges. Store Permissions for a user are provided by a Store Permissions Provider, and a user may have different permissions for each store on the BrightstarDB server. For more information about Store Permissions and providers please refer to the *Store Permissions* section of the *BrightstarDB Security* documentation.

The permissions that a user has are provided to the BrightstarDB service by one or more configured *Store Permission Providers*. The following providers are available "out of the box":

Fallback Provider This provider grants all users (authenticated or anonymous) a specific set of permissions. It is meant to be used in conjunction with a Combined Permissions Provider and some other providers. The configuration element for a Fallback Provider is:

```
<fallback authenticated="[Flags]" anonymous="[Flags]"/>
```

where `[Flags]` is one or more of the store permissions levels. Multiple values must be separated by the comma (,) character (e.g. "Read,Export"). The `anonymous` attribute can be ommitted, in which case anonymous users will be granted no store permissions.

Combined Permissions Provider This provider wraps two other providers and grants a user the combination of all permissions granted by the two child providers. You can use this to combine a custom permissions provider and a Fallback or Pass All provider to provide a backstop set of permissions when your custom provider doesn't grant any at all. The configuration element for a Combined Permissions Provider is:

```
<combine>[child providers]</combine>
```

where `[child providers]` is exactly two XML elements one for each of the child permission providers.

Static Provider This provider uses a fixed configuration that maps users or claims to permissions. The configuration element for a Static Permissions Provider is:

```
<static>
    <store name="{storeName}">
        <user name="{userName}" permissions="[Flags]" /> *
        <claim name="{claimName}" permissions="[Flags]" /> *
    </store> *
</static>
```

where `storeName` is the name of the store that the permissions are granted on, `userName` and `claimName` are the names of a specific user or a claim that a user holds respectively, and `[Flags]` is one or more store permission levels.

Depending on the user validation you use, the claim names may be specific claims about a user's identity (e.g. their email address) or about their group membership (e.g. group names) or both.

Any number of `store` elements may appear inside the `static` element, and any number of `user` and `claim` elements may appear inside the `store` element (in any order).

21.6.2 Configuring System Permissions

System Permissions control the access of users to list, create and manage BrightstarDB stores. There is one set of System Permissions for a user on the BrightstarDB server. For more information about System Permissions please refer to the *System Permissions* section of the *BrightstarDB Security* documentation.

The permissions that a user has are provided to the BrightstarDB service by one or more configured *System Permission Providers*. The following providers are available "out of the box":

Fallback Provider This provider grants all users (authenticated or anonymous) a specific set of permissions. It is meant to be used in conjunction with a Combined Permissions Provider and some other providers. The configuration element for a Fallback Provider is:

```
<fallback authenticated="[Flags]" anonymous="[Flags]" />
```

where `[Flags]` is one or more of the system permissions levels. Multiple values must be separated by the comma (,) character (e.g. "ListStores,CreateStore"). The `anonymous` attribute may be omitted in which case anonymous users will be granted no system permissions.

Combined Permissions Provider This provider wraps two other providers and grants a user the combination of all permissions granted by the two child providers. You can use this to combine a custom permissions provider and a Fallback or Pass All provider to provide a backstop set of permissions when your custom provider doesn't grant any at all. The configuration element for a Combined Permissions Provider is:

```
<combine>[child providers]</combine>
```

where `[child providers]` is exactly two XML elements one for each of the child permission providers.

Static Provider This provider uses a fixed configuration that maps users or claims to permissions. The configuration element for a Static Permissions Provider is:

```
<static>
    <user name="{userName}" permissions="[Flags]" /> *
    <claim name="{claimName}" permissions="[Flags]" /> *
</static>
```

where `userName` and `claimName` are the names of a specific user or a claim that a user holds respectively, and `[Flags]` is one or more system permission levels.

Depending on the user validation you use, the claim names may be specific claims about a user's identity (e.g. their email address) or about their group membership (e.g. group names) or both.

Any number of `user` and `claim` elements may appear inside the `static` element (in any order).

21.6.3 Configuring Authentication

Authentication is the process by which the server determines a user identity for an incoming request. BrightstarDB has been developed to give as much flexibility as possible over how the server authenticates a user, without (we hope!) making it to complicated to configure.

Authentication is a service that is implemented by an Authentication Provider. You can attach multiple Authentication Providers to the BrightstarDB server and each one will attempt to determine the user identity from an incoming request. If none of the attached Authentication Providers can determine the user identity, then the request is processed as if the user were an anonymous user.

The list of Authentication Providers for the server are configured by adding an `authenticationProviders` element inside the `brightstarService` element of the configuration file. The `authenticationProviders` element has the following content:

```
<authenticationProviders>
    <add type="{Provider Type Reference}"/> *
</authenticationProviders>
```

where `Provider Type Reference` is the full class and assembly reference for the authentication provider class to be used. An Authentication Provider class must implement the `BrightstarDB.Server.Modules.Authentication.IAuthenticationProvider` interface and it must also have a default no-args constructor. The `add` element used to add the provider is passed to the provider instance after it is constructed so depending on the provider implementation you may be allowed/required to add more configuration elements inside the `add` element. Check the documentation for the individual provider types below.

BrightstarDB provides the following implementations "out of the box":

NullAuthenticationProvider Type Reference: `BrightstarDB.Server.Modules.Authentication.NullAuthent`
`BrightstarDB.Server.Modules`

This provider does no authentication at all, so it is probably of very little interest!

BasicAuthenticationProvider Type Reference: `BrightstarDB.Server.Modules.Authentication.BasicAuther`
`BrightstarDB.Server.Modules`

This provider authenticates a user by their credentials being passed using HTTP Basic Authentication. It uses NancyFX's Basic Authentication Module, which accepts a custom validator class which implements the logic that takes the user name and password provided and determines the user identity. This requires some additional configuration, so the configuration for this provider follows this pattern:

```
<add type="BrightstarDB.Server.Modules.Authentication.BasicAuthenticationProvider,
          BrightstarDB.Server.Modules">
    <validator type="{Validator Type Reference}"/>
```

```
    <realm>{Authentication Realm}</realm> ?
</add>
```

Where `Validator Type Reference` is the full class and assembly reference for the validator class. A validator must implement the `Nancy.Authentication.Basic.IUserValidator` interface, which has a single method called Validate that receives the user name and password that the user entered and returns an IUserIdentity instance (or null if the username/password pair was not valid).

BrightstarDB provides the following "out of the box" validators:

MembershipValidator Type Reference: `BrightstarDB.Server.AspNet.Authentication, BrightstarDB.Server.AspNet`

This provider uses the ASP.NET Membership and Roles framework to validate the user identity. To use this provider you must also configure at least a Membership Provider for the server and optionally a Role Provider. The validator will create a user identity where the validated user name from the request is mapped to the user name of the generated user identity, and the roles that the user is in are mapped to claims on the generated user identity.

An example ASP.NET-based BrightstarDB service is available in the source code for you to see how all these pieces hang together (src\core\BrightstarDB.Server.AspNet.Secured).

Note: Please note that at present there are no validator implementations available for BrightstarDB running as a Windows Service. The Membership and Role providers bring in a dependency on ASP.NET that is not suitable for a Windows Service. A future release will address this deficit, but for now if you want user authentication you will have to run the ASP.NET implementation of the BrightstarDB server.

21.6.4 Additional Configuration Options

A number of other aspects of BrightstarDB service operations can be configured by adding values to the `appSettings` section of the application configuration file. These are:

- `BrightstarDB.LogLevel` - configures the level of detail that is logged by the BrightstarDB application. The valid options are ERROR, INFO, WARN, DEBUG, and ALL. For more information about logging and configuring where logs are written please refer to the section *Logging*. For Windows Phone 7.1 this setting is fixed as ERROR and cannot be overridden.
- `BrightstarDB.TxnFlushTripleCount` - specifies a batch size for importing large sets of triples. At the end of each batch BrightstarDB will perform housekeeping tasks to try to ensure a lower memory footprint. The default value is 10,000 on .NET 4.0. For applications that run on larger, more capable hardware (with available memory of 4GB or more) the value can usually be increased to 50,000 or even 100,000 - but it is worth testing the configured value before committing to it in deployment. For Windows Phone 7.1 this value is fixed as 1,000 and cannot be overridden.
- `BrightstarDB.PageCacheSize` - specifies the amount of memory in MB to be used by the BrightstarDB store page cache. This setting applies only to applications that open a BrightstarDB store as the cache is used to cache pages of data from the data.bs and resources.bs data files. The default value is 2048 on .NET 4.0 and 4 on Windows Phone 7.1. Note that this memory is not all allocated on startup so actual memory usage by the application may initially be lower than this value.
- `BrightstarDB.ResourceCacheLimit` - specifies the number of resource entries to keep cached for each open store. Default values are 1,000,000 on .NET 4.0 and 10,000 on Windows Phone.
- `BrightstarDB.EnableQueryCache` - specifies whether or not the application should cache the results of SPARQL queries. Allowed values are "true" or "false" and the setting defaults to "true". Query caching is only available on .NET 4.0 so this setting has no effect on Windows Phone 7.1

- `BrightstarDB.QueryCacheDirectory` - specifies the folder location where cached results are stored.
- `BrightstarDB.QueryCacheMemory` - specifies the amount of memory in MB to be used by the SPARQL query cache. The default value is 256.
- `BrightstarDB.QueryCacheDisk` - specifies the amount of disk space (in MB) to be used by the SPARQL query cache. The default value is 2048. The disk space used will be in a subdirectory under the location specified by the BrightstarDB.StoreLocation configuration property.
- `BrightstarDB.PersistenceType` - specifies the default type of persistence used for the main BrightstarDB index files. Allowed values are "appendonly" or "rewrite" (values are case-insensitive). For more information about the store persistence types please refer to the section *Store Persistence Types*.
- `BrightstarDB.StatsUpdate.Timespan` - specifies the minimum number of seconds that must pass between automatic update of store statistics.
- `BrightstarDB.StatsUpdate.TransactionCount` - specifies the minimum number of transactions that must occur between automatic update of store statistics.

21.6.5 Example Server Configuration

The sample below shows all the BrightstarDB options with usage comments.

```
<?xml version="1.0"?>
<configuration>
  <configSections>
    <!-- This configuration section is required to configure server security -->
    <section name="brightstarService" type="BrightstarDB.Server.Modules.BrightstarServiceConfiguratio
    <!-- This configuration section is required only for advanced configuration options
         such as page-cache warmup -->
    <section name="brightstar" type="BrightstarDB.Config.BrightstarConfigurationSectionHandler, Brigh
  </configSections>

  <appSettings>

    <!-- The logging level for the server. -->
    <add key="BrightstarDB.LogLevel" value="ALL" />

    <!-- Indicates the number of triples in a transaction to process before doing a partial commit.
         Larger numbers require more machine memory but result in faster transaction processing. -->
    <add key="BrightstarDB.TxnFlushTripleCount" value="100000" />

    <!-- Specifies the maximum amount of memory (in MB) to use for page caching. -->
    <add key="BrightstarDB.PageCacheSize" value="2048" />

    <!-- Enable (true) or disable (false) the caching of SPARQL query results -->
    <add key-"BrightstarDB.EnableQueryCache" value="true" />

    <!-- The amount of memory to use for the SPARQL query cache -->
    <add key="BrightstarDB.QueryCacheMemory" value="512" />

    <!-- The amount of disk space (in MB) to use for the SPARQL query cache. This only applies to ser
    <add key="BrightstarDB.QueryCacheDisk" value="2048" />

    <!-- The default store index persistence type -->
    <add key="BrightstarDB.PersistenceType" value="AppendOnly" />

  </appSettings>
```

```
  <!-- Core BrightstarDB service configuration -->
  <brightstarService connectionString="type=embedded;StoresDirectory=c:\brightstar">

    <!-- Store Permissions Provider. -->
    <storePermissions>
      <!-- WARNING: This configuration Grants full access to all users -->
      <passAll anonPermissions="All"/>
    </storePermissions>

    <!-- System Permissions Provider -->
    <systemPermissions>
      <!-- WARNING: This configuration Grants full access to all users -->
      <passAll anonPermissions="All"/>
    </systemPermissions>

  </brightstarService>

  <brightstar>

    <!-- Enable page-cache warmup -->
    <preloadPages enabled="true" />

  </brightstar>

</configuration>
```

21.7 Configuring Caching

BrightstarDB provides facilities for caching the results of SPARQL queries both in memory and to disk. Caching complex SPARQL queries or queries that potentially return large numbers of results can provide a significant performance improvement. Caching is controlled through a combination of settings in the application configuration file (the web.config for web apps, or the .exe.config for other executables).

AppSetting Key Default Value Description BrightstarDB.EnableQueryCache false Boolean value (“true” or “false”) that specifies if the system should cache the result of SPARQL queries. BrightstarDB.QueryCacheMemory 256 The size in MB of the in-memory query cache. BrightstarDB.QueryCacheDirectory <undefined> The path to the directory to be used for the disk cache. If left undefined, then the behaviour depends on whether the BrightstarDB.StoreLocation setting is provided. If it is, then a disk cache will be created in the _bscache subdirectory of the StoreLocation, otherwise disk caching will be disabled. BrightstarDB.QueryCacheDiskSpace 2048 The size in MB of the disk cache.

21.7.1 Example Caching Configurations

To cache in the _bscache subdirectory of a fixed store location (a good choice for server applications), it is necessary only to enable caching and ensure that the store location is specified in the configuration file:

```
<configuration>
  <appSettings>
    <add key="BrightstarDB.EnableQueryCache" value="true" />
    <!-- disk cache will be written to the directory d:\brightstar\_bscache -->
    <add key="BrightstarDB.StoreLocation" value="d:\brightstar\" />
  </appSettings>
</configuration>
```

To cache in some other location (e.g. a fast disk dedicated to caching):

```
<configuration>
  <configSections>
    <section name="brightstarService" type="BrightstarDB.Server.Modules.BrightstarServiceConfiguratio
  </configSections>
  <appSettings>
    <add key="BrightstarDB.EnableQueryCache" value="true" />
    <add key="BrightstarDB.StoreLocation" value="d:\brightstar\" />


    <!-- Cache on a different disk from the B* stores to maximize disk throughput.
         Disk cache will be written to the directory e:\bscache -->
    <add key="BrightstarDB.QueryCacheDirectory" value="e:\bscache\"/>


    <!-- Allow disk cache to grow to up to 200GB in size -->
    <add key="BrightstarDB.QueryCacheDiskSpace" value="204800" />
  </appSettings>
</configuration>
```

This sample has no disk cache because there is no valid location for the cache to be created:

```
<configuration>
  <appSettings>
    <add key="BrightstarDB.EnableQueryCache" value="true" />
    <!-- 1GB in-memory cache -->
    <add key="BrightstarDB.QueryCacheMemory" value=1024"/>


    <!-- This property is not used because there is no
          BrightstarDB.QueryCacheDirectory or
          BrightstarDB.StoreLocation setting defined. -->
    <add key="BrightstarDB.QueryCacheDiskSpace" value="204800" />


  </appSettings>
</configuration>
```

21.8 Configuring Logging

BrightstarDB uses the .NET diagnostics infrastructure for logging. This provides a good deal of runtime flexibility over what messages are logged and how/where they are logged. All logging performed by BrightstarDB is written to a TraceSource named "BrightstarDB".

The default configuration for this trace source depends on whether or not the *BrightstarDB.StoreLocation* configuration setting is provided in the application configuration file. If this setting is provided then the BrightstarDB trace source will be automatically configured to write to a log.txt file contained in the directory specified as the store location. By default the trace source is set to log Information level messages and above.

Other logging options can be configured by entries in the <system.diagnostics> section of the application configuration file.

To log all messages (including debug messages), you can modify the TraceSource's *switchLevel* as follows:

```
<system.diagnostics>
  <sources>
    <source name="BrightstarDB" switchValue="Verbose"/>
```

```
  </sources>
</system.diagnostics>
```

Equally you can use other switchValue settings to reduce the amount of logging performed by BrightstarDB.

21.9 Preloading Stores

The BrightstarDB server can be configured to automatically preload the active pages from one or more stores into the in-memory page-cache. Preloading the pages trades-off a slightly longer server start-up time for a reduced time to respond to the first incoming request. By default preloading is disabled and pages will be pulled into the cache on an as-needed basis.

21.9.1 Configuring Basic Preloading

As preloading is concerned with populating the BrightstarDB store page cache, it can only be enabled on a BrightstarDB server that is using an embedded connection to a store directory. Basic preloading will fill the cache with pages from all stores in the store directory in an equal ratio, so if there are 10 stores in the directory, each will be allowed to use up to 10% of the available cache. Basic preloading proceeds in order of store size (from smallest to largest store based on their data file sizes), so if smaller stores do not use up their full allocation of pages, the remaining space can be shared amongst the remaining larger stores as they are pre-loaded.

To enable basic preloading, the following needs to be added to the `brightstar` element in the server application (or web) configuration file:

```
<preloadPages enabled="true" />
```

21.9.2 Advanced Preloading

Basic preloading is a simple strategy that makes the assumption that all stores in a directory are equally important - each is preloaded to the same extent. In some cases as an administrator you may want to prioritize some stores over others.

To allow for this you can assign one or more stores a cache ratio number. This number specifies the relative amount of page cache space to be assigned to the store, so a store with a cache ratio of 3 gets 3x the pages that a store with a cache ratio of 1 is assigned, and 1.5x the pages that a store with a cache ratio of 2. By default all stores have a cache ratio of 1 assigned, but you can also set this default to 0.

To configure advanced preloading you add a `store` element child to the `preloadPages` element as shown here:

```
<preloadPages enabled="true">
    <store name="storeA" cacheRatio="4" />
    <store name="storeB" cacheRatio="2" />
</preloadPages>
```

To understand how cache ratios work, imagine that the server using this configuration is actually serving 4 stores, storeA, storeB, storeC and storeD, and that the server is configured with a page cache size of 2048M As the default cache ratio for a store is 1, the effective ratios for the stores are:

Store Name	Cache Ratio
storeA	4
storeB	2
storeC	1
storeD	1

The sum of those ratios is (4+2+1+1) = 8. So storeC and storeD are assigned one-eighth of the page cache, storeB is assigned one-quarter and storeA one-half, making the assigned page cache preload sizes:

Store Name	Cache Ratio	Preload Size
storeA	4	1024M
storeB	2	512M
storeC	1	256M
storeD	1	256M

It is also possible to change the default cache ratio assigned to stores that are not explicitly configured by adding a `defaultCacheRatio` attribute to the `preloadPages` element:

```
<preloadPages enabled="true" defaultCacheRatio="2">
    <store name="storeA" cacheRatio="4" />
    <store name="storeB" cacheRatio="2" />
</preloadPages>
```

The configuration above changes the cache preload sizes for the stores as follows:

Store Name	Cache Ratio	Preload Size
storeA	4	819.2M
storeB	2	409.6M
storeC	2	409.6M
storeD	2	409.6M

It is also possible to use the `defaultCacheRatio` to disable preloading for stores that are not explicitly named, by setting the default ratio to zero:

```
<preloadPages enabled="true" defaultCacheRatio="0">
    <store name="storeA" cacheRatio="4" />
    <store name="storeB" cacheRatio="2" />
</preloadPages>
```

This leads the the following preloaded cache sizes:

Store Name	Cache Ratio	Preload Size
storeA	4	1365.3M
storeB	2	682.7M
storeC	0	0M
storeD	0	0M

CHAPTER 22

SPARQL Endpoint

The BrightstarDB service supports query and update as specified in the SPARQL 1.1 W3C recommendation. Each BrightstarDB store has its own endpoints for query and for update.

22.1 Usage

The SPARQL service accepts both query and update operations. With the BrightstarDB service is accessible at {service}, the following URI patterns are supported:

Query

```
GET {service}/{storename}/sparql?query={query expression}
```

Will execute the query provided as the query parameter value against the store indicated.

```
POST {service}/{storename}/sparql
```

Will look for the query as an unencoded value in the request body.

Update

```
POST {service}/{storename}/update
```

Will execute the update provided as the value in the request body.

For full details on these protocols, please refer to the SPARQL 1.1 Protocol recommendation.

Note: The SPARQL 1.1 Graph Store Protocol is not implemented at this time.

CHAPTER 23

Polaris Management Tool

Polaris is a Windows desktop application that allows a user to manage various aspects of local and remote BrightstarDB servers. Using Polaris you can:

- Create and delete stores on the server
- Import N-Triples or N-Quads files into a store
- Run a SPARQL query against a store
- Run an update transaction against a store

Polaris is optionally installed as part of the BrightstarDB installer, if it is not initially installed, it can be installed later by re-running the installer and selecting the appropriate option.

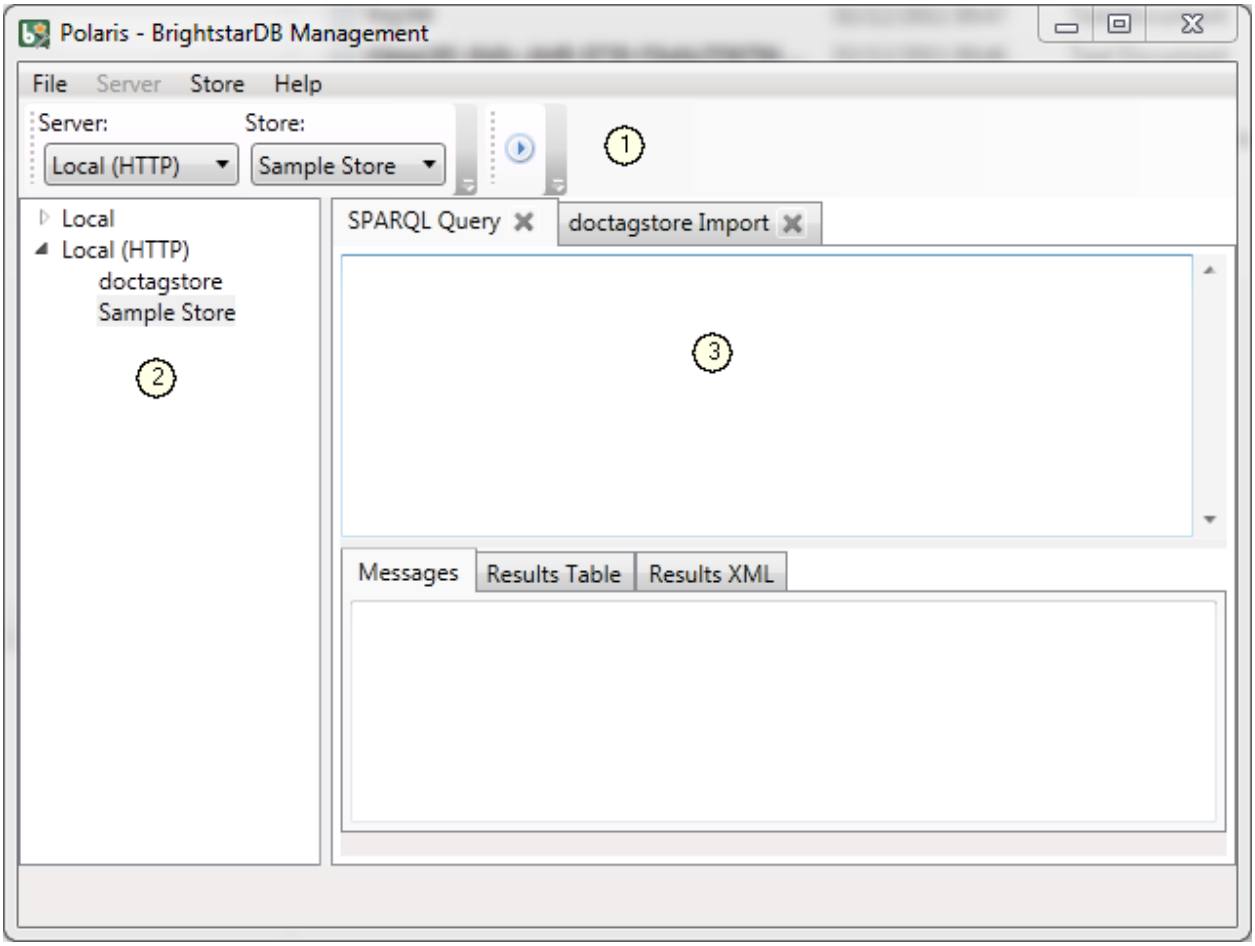
23.1 Running Polaris

Polaris can be run by clicking on its short-cut, which can be found inside the folder BrightstarDB on the Start Menu. Alternatively it can be run from the command-line. To run from the command-line, run the BrightstarDB.Polaris.exe executable. This executable can be found in [INSTALLDIR]ToolsPolaris. The executable accepts the following command line parameters:

Parameter	Description
/log:{log file name} [/verbose]	With the /log: option specified on the command-line, Polaris will write logging information to the file named after the colon (:) character. The optional /verbose flag will ensure that more verbose logging information is also written to this file.

23.2 Polaris Interface Overview

The Polaris user interface consists of three areas as shown in the screenshot below.

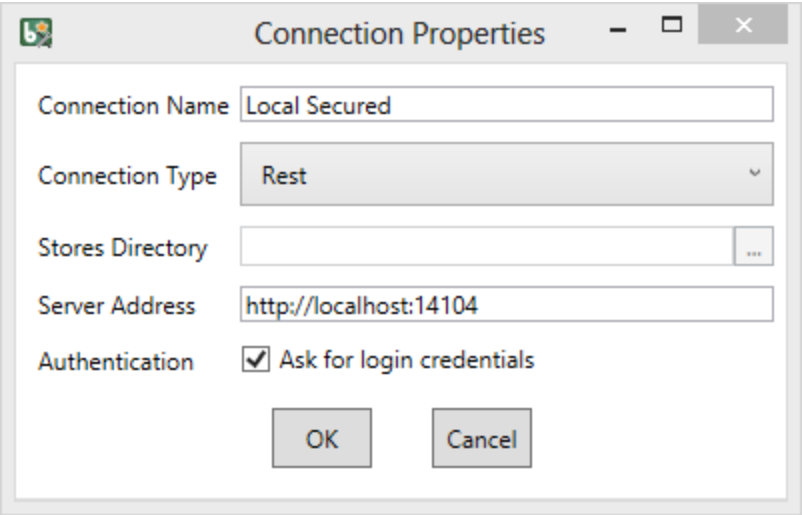


The areas of the interface are:

1. The Menu Area contains the Polaris menu items and, depending on the current tab selected in the Tab Area may also display a toolbar.
2. The Server List contains a list of all the servers that Polaris has been configured with connection strings for. If Polaris is able to establish a connection to the server, the list of stores on that server can be viewed or hidden by clicking on the toggle button to the left of the server name.
3. The Tab Area contains tabs for running SPARQL queries or transaction updates against a store.

23.3 Configuring and Managing Connections

To configure Polaris with a new connection, click on File > Connect... to bring up the Connection Properties dialog as shown in the screenshot below.



The fields of this dialog should be filled out as follows:

- **Connection Name:** Enter a memorable name for this connection - this is the name that will be displayed in the Polaris interface.
- **Connection Type:** Choose the protocol to use to connect to the server. This may be one of:
  - **Embedded: Select this option to connect directly to the store data** files. This is only recommended when the data files are accessible on a local disk and should not be used to access data files that any other process (such as a BrightstarDB server) could be attempting to access at the same time.
  - REST: Connect to the server using the HTTP or HTTPS protocol. This is the option to use to connect to a remote server.
- **Stores Directory:** This property is required only for the Embedded connection type. Specify the full path to the directory that contains the BrightstarDB server's store folders.
- **Server Address:** This property is required only for the Rest connection type. Specify the full URL for the BrightstarDB service. For example, to connect to a BrightstarDB service running on the default port (8090) on the local machine you would enter `http://localhost:8090/brightstar` as the address.
- **Authentication:** Check the box if the connection you are using requires a user name and password. You will be prompted for this information after the connection is added.

When you select the Rest connection type from the drop-down list, the dialog will automatically populate with the default settings for making a connection to a local BrightstarDB server. You can modify the server name and/or the other settings to make a connection to a remote server or to a server with a non-default port setup.

As you make changes in this dialog any validation error messages will be displayed in a red area beneath the input fields.

When you click OK, Polaris will attempt to contact the server using the information you have provided, if contact is established then a list of all stores hosted on that server will be retrieved and displayed under the server name in the Server List area. If you checked the box that indicates that the connection requires authentication, you will be prompted to enter your user name and password in a separate dialog. If contact cannot be established for some reason, an error dialog will display the details of the problem encountered.

To remove a connection from the list, select the server name in the Server List area and click on Server > Remove Server From List, or right-click on the server name and select Remove Server From List from the popup menu. You will be prompted to confirm this operation before the server is removed from the list.

To edit an existing connection, select the server name in the Server List area and click on Server > Edit Connection, or right-click on the server name and select "Edit" from the popup menu. The Connection Properties dialog will be

displayed allowing you to edit the parameters used for the connection.

If for some reason a connection cannot be established to a server, the message "Could not establish connection" will be displayed next to the server name in the Server List. To attempt to reconnect to the server, select the server from the list and click on Server > Refresh.

The connections you add to Polaris are stored in a configuration file under your local AppData folder and they will be automatically saved when you add/remove a connection.

23.3.1 Authentication

Any REST connection to a BrightstarDB server can be configured to require authentication. When you configure a new connection in Polaris you can specify that the connection requires authentication information. This information will be prompted for when you initally add the connection, but your user name and password is **NOT** stored in the configuration file. Instead, the next time you start Polaris you will see that connection shows an error in connecting (probably with a message saying 401 Authentication Required, which is the HTTP protocol message that Polaris receives from the server). In this case, simply right-click on the connection and select "Refresh" from the popup menu - you should then be prompted for your user name and password. If this does not happen, it probably means that the connection was added to Polaris without the Authentication box checked.

To force authentication on a connection that does not currently require it, simply right-click on the connection, select "Edit" from the popup menu and check the Authentication box in the dialog displayed and click OK. You should then be prompted for your user name and password, and this update to the connection configuration will be stored for future sessions.

If you want to authenticate as multiple different users, or if you want to have an authenticated connection and an unauthenticated connection to the same server, simply add the same connection details to Polaris multiple times.

Note: At present it is not possible to associate a user name with a connection. You will always be required to enter both your user name and password.

23.4 Managing Stores

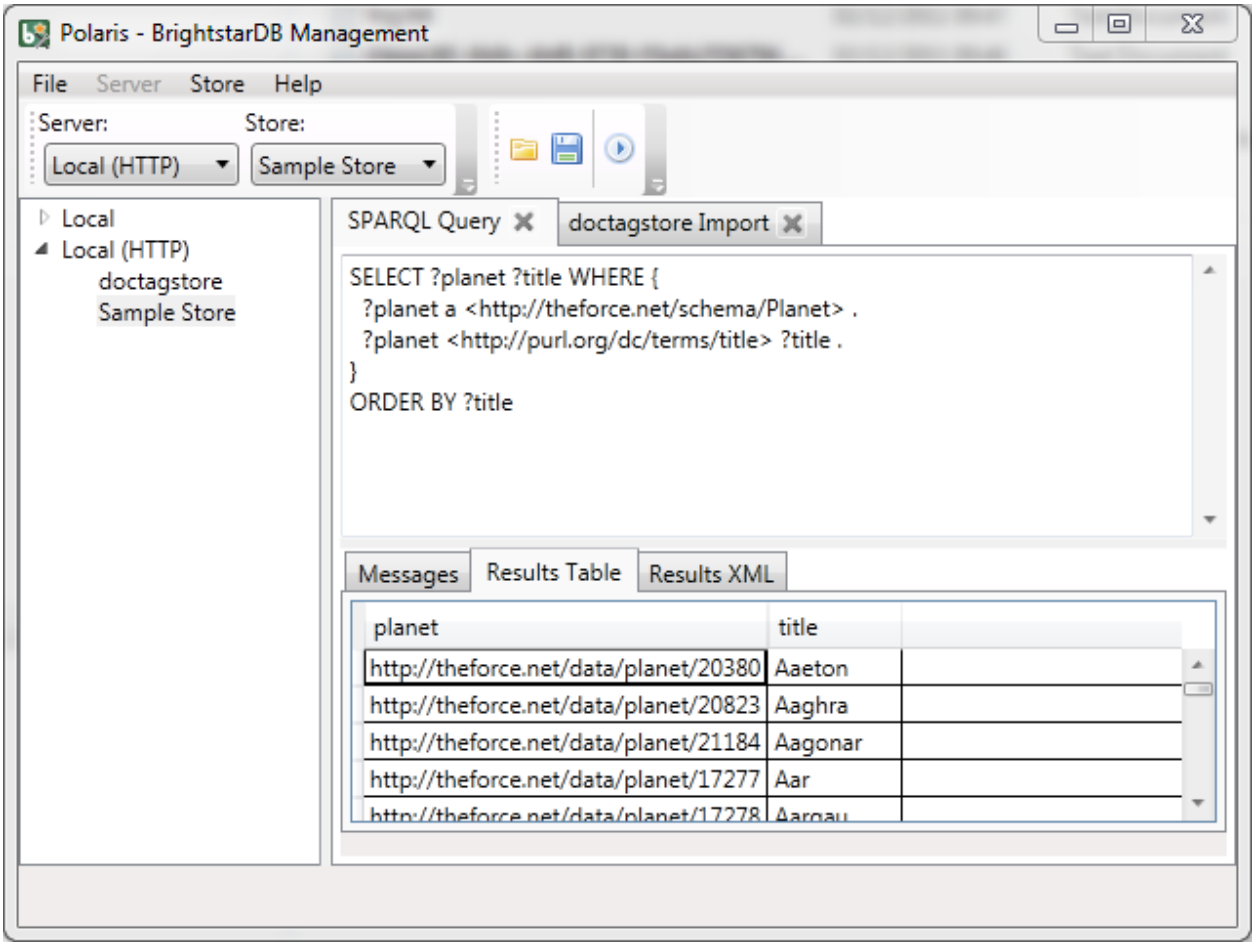
To add a new store to a server, select the server from the Server List area and then click on Server > New Store.., or right-click on the server and select New Store from the popup menu. In the dialog box that is displayed, enter the name of the store. A default GUID-based name is generated for you, but changing this to a more meaningful name will probably be useful for you and other users of the server. The new store will be added to the end of the list of stores for the server in the Server List area.

To delete a store from a server, select the store from the Server List area and then click on Store > Delete, or right-click on the store and select Delete. You will asked to confirm the operation before it is completed.

Removing a store from a server deletes the entire contents of the store from the server. It is not possible to undo this operation once it is confirmed.

23.5 Running SPARQL Queries

Polaris allows users to write SPARQL queries and execute them against a BrightstarDB store. To create a query, select the store you wish to run the query against and then click on Store > New > SPARQL Query, or right click on the store and select New > SPARQL Query from the popup menu. This will add a new SPARQL Query tab to the Tab area. The interface is shown in the screenshot below.



The toolbars added to the Menu area allow you to change the store that the query will execute against by selecting the server and the store from the drop-down lists. The query is executed either by pressing the F5 key or by clicking on the button in the tool bar.

The tab itself is divided into a top area where you can write your SPARQL query and a lower area which displays messages and results when a query is executed. If part of the text in this area is selected when the query is run, then only the selected text will be passed to BrightstarDB. A query that results in SPARQL bindings (typically a SELECT query) will display results in a tabular format in the Results Table tab. All queries will also display their results in the Results XML tab.

Note: For more details about the SPARQL query language please refer to *Introduction To SPARQL*.

23.6 Saving SPARQL Queries

You can save SPARQL queries entered in Polaris to use in later sessions. To save a query, select the tab that contains the query you want to save and then click on the button. By default your queries will be saved to a folder named "SPARQL Queries" inside your "My Documents" folder - if this folder does not already exist, you will be prompted to allow Polaris to create it for you (if you choose not to allow this, you can choose a different location to save queries to). Saved queries are stored with a ".sq" extension.

To load a saved query, open a new SPARQL Query tab or select an existing one and then click on the button. A

file dialog will appear allowing you to select the query to be loaded.

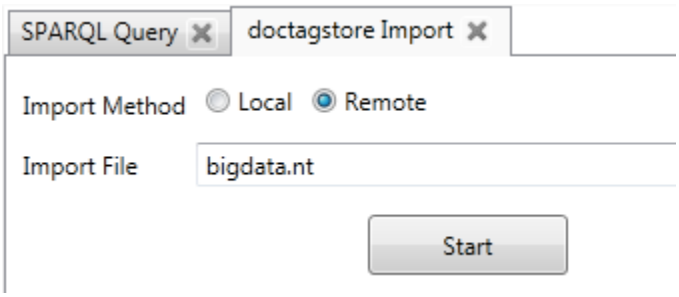
23.7 Importing Data

Polaris allows users to import RDF data from files into an existing BrightstarDB store. Polaris supports two modes of data import: Remote and Local. A Remote import specifies the name of a file that is located in a specific directory on the target server and submits a job for that file to be imported into the store. A Local import specifies the name of a file that is accessible to Polaris, processes it locally and then creates a job to add the data contained in that file to the target server. Remote import allows for much more efficient loading of very large data sets but it requires that the data file(s) should first be copied onto the server.

Note: For details about the RDF syntaxes that are supported by BrightstarDB and Polaris, please refer to *Supported RDF Syntaxes*.

To run a Remote import:

1. Ensure that the file to be imported is copied into the Import folder located directly under the stores directory of the server. When connecting to a server via HTTP, TCP or Named Pipes, the import directory is located in the directory on the server where the stores are located (typically [INSTALLDIR]Data). When connecting to an embedded store, the import directory should be created in the directory specified for the embedded store. If this directory does not exist it should be created. You should also ensure that the user that the BrightstarDB service has sufficient privileges to be able to read the files to be imported.
2. From the Polaris interface, create a new import task by selecting the store the data is to be imported into and then clicking Store > New > Import Job, or by right-clicking on the store and selecting New > Import Job from the popup menu.
3. In the interface that is displayed, change the Import Method radio button selection to Remote and enter the name of the file to be imported. Do not specify the path to the file, just the file name - the server will only look for this file in its Import directory.
4. Click on the Start button to submit the job to the server.
5. Once the job is submitted, the interface will track the job progress, but you can at any time exit Polaris and the job will continue to run on the server.

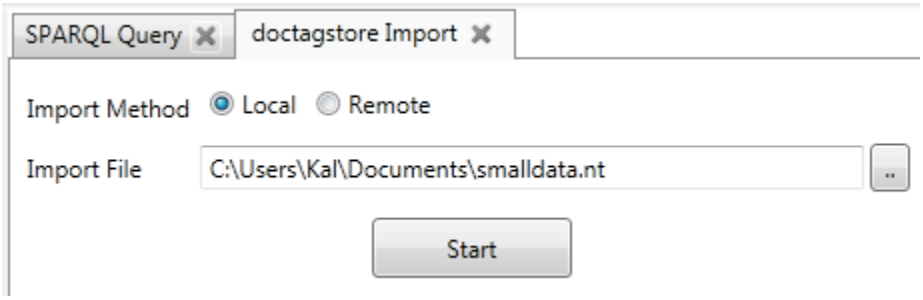


To run a Local import:

1. From the Polaris interface, create a new import task by selecting the store the data is to be imported into and clicking Store > New > Import Job.
2. In the interface that is displayed, ensure the Import Method is set to Local and enter the full path to the file to be imported - you can use the .. button to launch a file browser to locate the file.
3. Click on the Start button.

4. Polaris will attempt to parse the contents of the file and create a new job to submit the data found in the file to the server.
5. Once the job is submitted, the interface will track the job progress, but you can at any time exit Polaris and the job will continue to run on the server.

Note: Local import is not recommended for large data files. If the file you try to import exceeds 50MB in size a warning will be displayed - you may still continue with the import, but you may experience better performance if you copy the data file to the server's import folder and use a Remote import instead. This even applies to the case where the server connection type is Embedded.

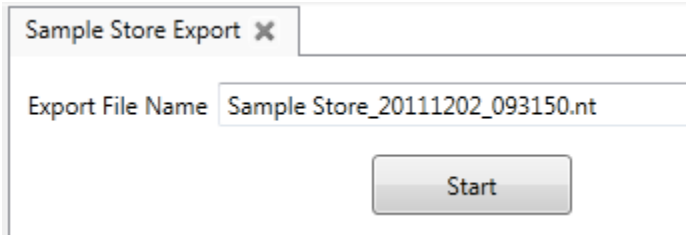


23.8 Exporting Data

You can export all of the RDF data contained in a BrightstarDB store using Polaris. For performance and network considerations, data export is limited to working as a remote job - the export request is submitted as a long-running job and the data is written to a specific directory on the target server.

To run an export:

1. From the Polaris interface, create a new export task by selecting the store that the data is to be exported from and then clicking Store > New > Export Job, or by right-clicking on the store and selecting New > Export Job from the popup menu.
2. In the interface that is displayed, a default name for the export file is generated based on the store name and the current date/time. You can modify this file name if you wish.
3. Click on the Start button to submit the job to the server.
4. Once the job is submitted, the interface will track the job progress. For connections other than a local embedded connection, you can exit Polaris and the job will continue to run on the server.
5. Once the job is completed, the exported data will be found in the Import folder located directly under the stores directory of the server.



23.9 Running Update Transactions

An update transaction allows you to specify the triples to delete from and add to a store. Deletions are always processed before additions, allowing you to effectively replace or update property values by issuing a delete and an add in the same transaction.

The triples to be deleted are specified using N-Triples syntax with one extension. The special symbol <*> can be used in place of a URI or literal value to specify a wildcard match so:

```
<http://example.org/people/alice> <http://xmlns.org/foaf/0.1/name> <*>
```

Would remove all FOAF name properties from the resource http://example.org/people/alice equally, the following can be used to remove all properties from the resource:

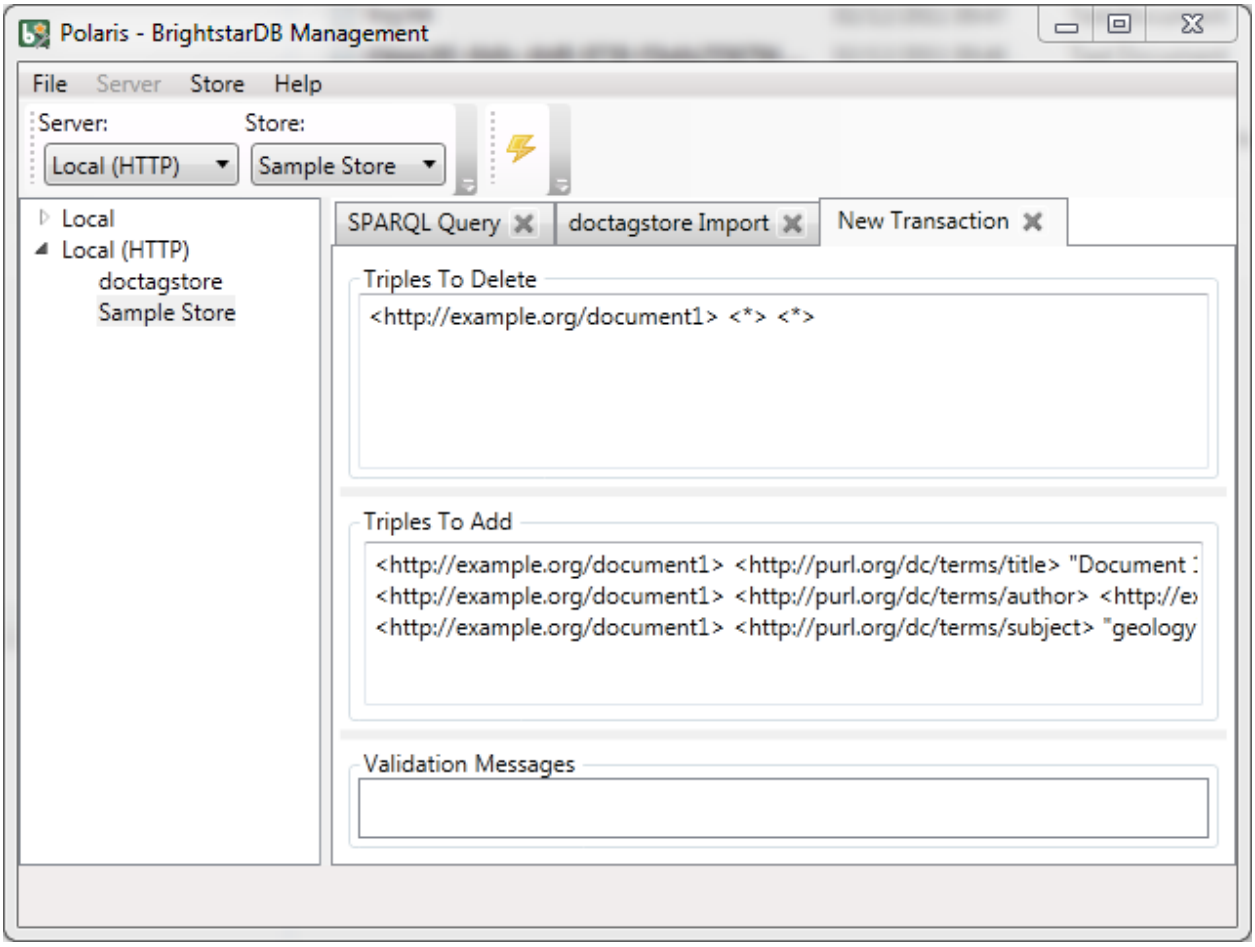
```
<http://example.org/people/alice> <*> <*>
```

The triples to be added are also specified using N-Triples syntax, but in this case the wildcard symbol is not supported.

Note: For a quick introduction to the N-Triples syntax please refer to *Introduction To NTriples*

To run an update transaction:

1. From the Polaris interface, create a new update task by selecting the store the update is to be executed against and clicking Store > New > Transaction, or by right clicking on the store and selecting New > Transaction from the popup menu.
2. In the interface that is displayed, enter the triple patterns to delete and the triples to add into the relevant boxes.
3. To run the transaction click on the  icon in the tool bar.
4. A dialog box will display the outcome of the transaction.



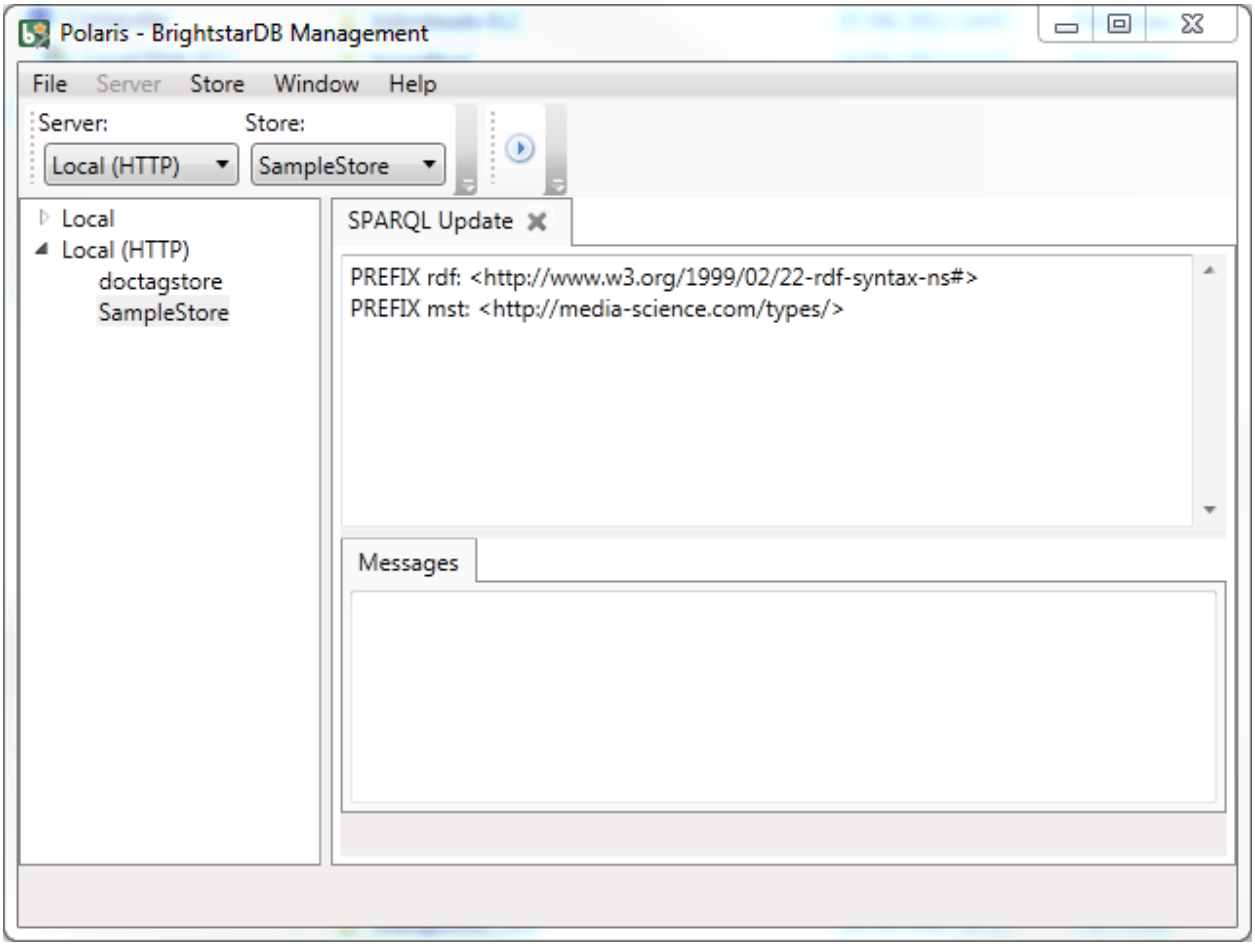
Note: You can run the same transaction against a different store by changing the selected server and store in the drop-down lists in the toolbar area.

23.10 Running SPARQL Update Transactions

The SPARQL Update support in BrightstarDB allows you to selectively update, add or delete data in a BrightstarDB store in a transaction. BrightstarDB supports the SPARQL 1.1 Update language.

To run an update transaction:

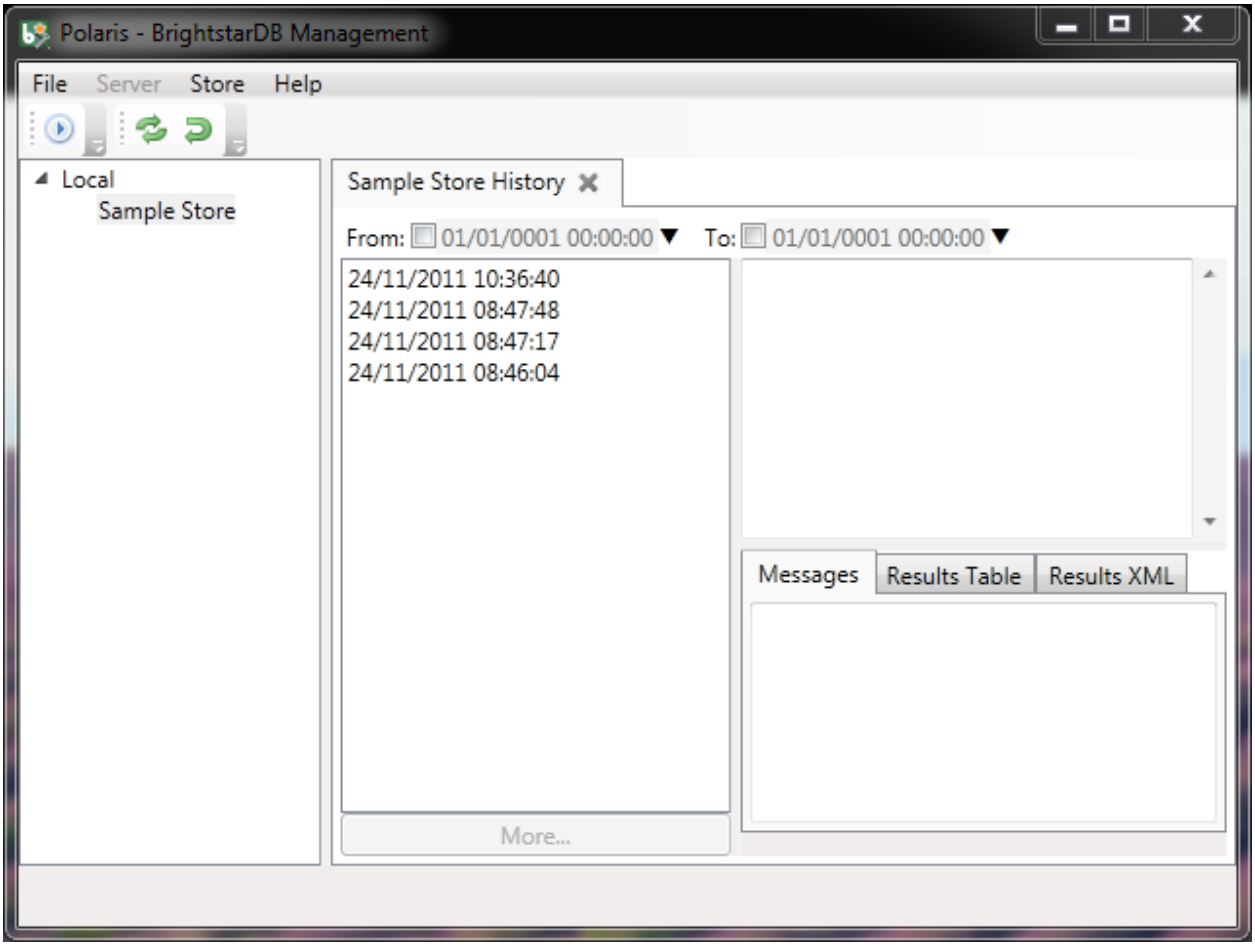
1. From the Polaris interface, create a new SPARQL Update task by selecting the store the update is to be executed against and clicking Store > New > SPARQL Update, or by right clicking on the store and selecting New > SPARQL Update from the popup menu.
2. In the interface that is displayed, enter the SPARQL Update request into the upper text box.
3. To run the transaction click on the  icon in the tool bar.
4. The results of the operation will be displayed in the lower text area.



Note: You can run the same transaction against a different store by changing the selected server and store in the drop-down lists in the toolbar area.

23.11 Managing Store History

Polaris provides the ability to view all the previous states of a BrightstarDB store and to query the store as it existed at any previous point in time. You can also "revert" the store to a previous state. These operations can be performed using the Store History View. To access this view, select the store in the Server List area on the left and click on Store > New > History View, or right-click on the store and select New > History View from the popup menu. This will add a new history view tab to the window as shown in the screenshot below.



The tab content is divided into two panes. The left-hand pane shows a list of the historical commit points for the store as the date/time when the store update was committed. By default this panel lists the 20 most recent commits, however you can use the fields at the top of the panel to restrict the date range. The black arrow next to each date/time field allows you to pick a date, and any of the fields in the picker can be altered by clicking on the field and using the up and down arrows on the keyboard or the mouse wheel. When retrieving commit points from the store, the server returns a maximum of 100 commit points in one go, if there are more than 100 commit points in the date range, the "More..." button is enabled to allow you to retrieve the next 100 from the server. You can refresh the commit list by clicking on the .. image:: Images/polaris_refreshbutton.png, this will clear the current list of commit points and the current date filters and re-run the query to retrieve the latest 20 commit points from the server.

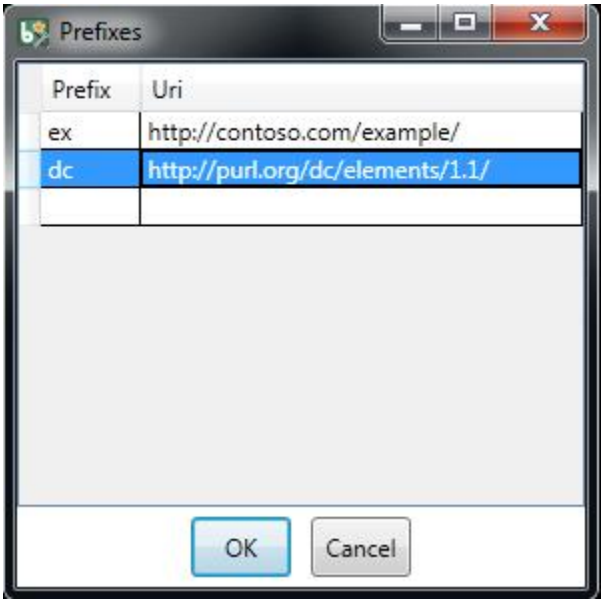
The right-hand panel allows you to write a SPARQL query and execute it against the store. With no commit point selected on the left, the query is executed against the store in its current state. However, once you select a commit point, the query is executed against that commit point. To run the SPARQL query click on the  button in the tool bar.

If you wish to revert the store to a previous state, you can do this by selecting the commit point you want to revert to and clicking on the  button in the toolbar. You will be prompted to confirm this action before it is applied to the store. This action creates a new commit point that points back to the store as it exited at the selected commit point - it does not delete or remove the changes made since that commit point. When you revert the store in this way, the list of commit points and the date filters are cleared and the latest 20 commit points are retrieved from the server again.

23.12 Defining and Using Prefixes

As it can be cumbersome and slow to have to continually type in long URI strings, Polaris provides functionality to allow you to map the namespace URIs you most commonly use to shorter prefixes. These prefixes can be used both in SPARQL queries and in transactions.

To manage the prefixes defined in Polaris click on File > Settings > Prefixes. This displays the prefixes dialog, which will initially be empty. You can add a new prefix by entering a prefix string and URI in the next empty row. To delete a prefix, click on the row and press the Delete key. You can also modify a prefix or URI by selecting the text and typing directly into the text box.



Once a prefix is defined it will automatically be added to the start of any new SPARQL query you create as PREFIX declarations, and can then be used in the normal way that any PREFIX declaration in SPARQL can be used. Prefixes can also be used in transactions so instead of typing a full URI you can type the prefix followed by a colon and then the rest of the URI, the prefix and the colon are replaced by the URI specified in the prefixes dialog. For example if you map the prefix string "ex" to "http://contoso.com/example/", and dc to "http://purl.org/dc/elements/1.1/" then the following NTriple in a transaction:

```
<http://contoso.com/example/1234> <http://purl.org/dc/elements/1.1/title> "This is an example" .
```

can be re-written more compactly as:

```
<ex:1234> <dc:title> "This is an example"
```

Note: Unlike SPARQL, the < > markers are still REQUIRED around each `prefix:restOfUri` string.

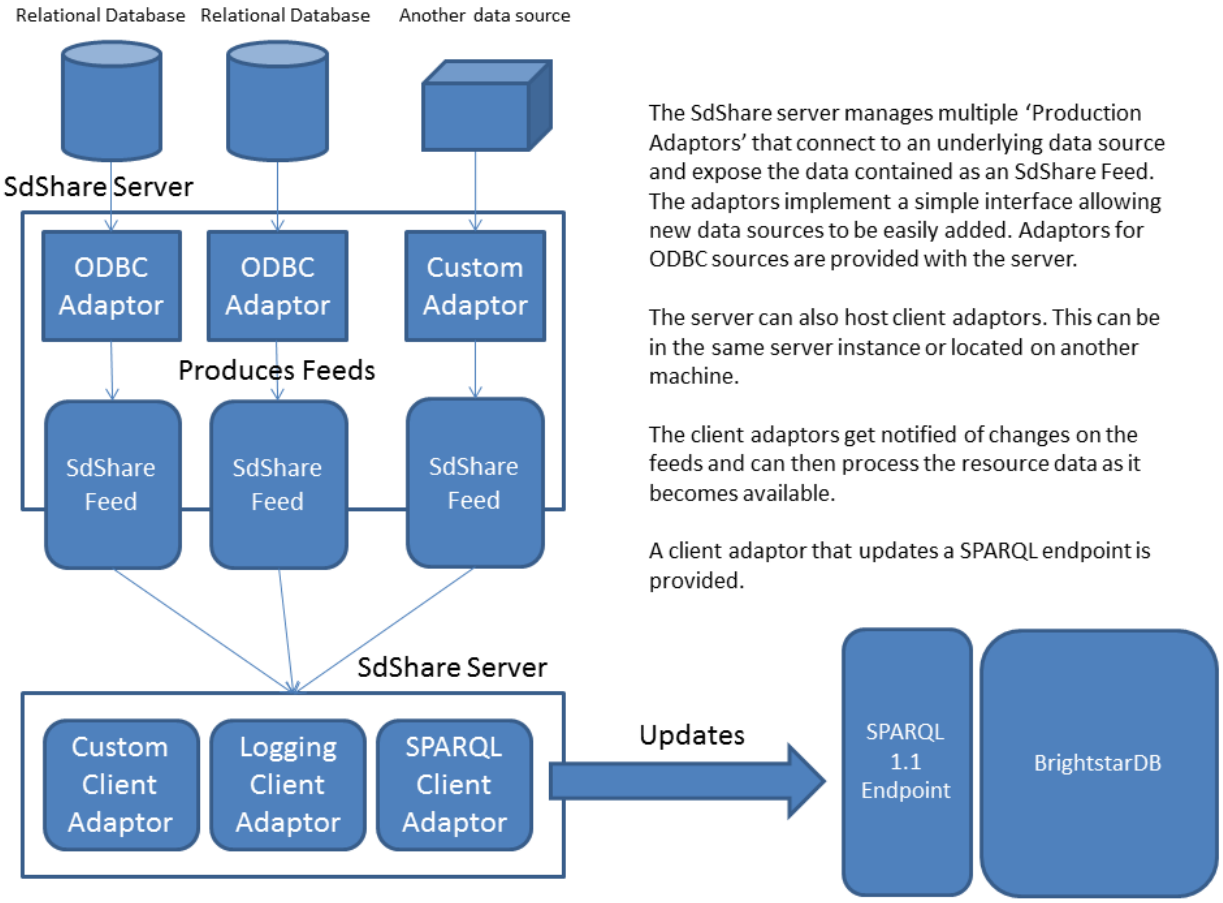
CHAPTER 24

SdShare Server

The BrightstarDB SdShare Server is designed to be used to expose RDF data from existing data sources. The data produced can be easily consumed into a BrightstarDB instance or any SPARQL compliant data store. The server has a pluggable architecture to allow any data source to be exposed in accordance with the latest SDShare specification (SDShare), it comes with configurable components for ODBC enabled databases.

The SdShare Server provides two main features, firstly it exposes existing data sources as feeds of data that comply with the SdShare specification.Second, it runs a client service that can consume and process valid SdShare feeds. Both the producer and consumer services offer a pluggable framework to support different data sources and data destinations. In addition, a data source adaptor is provided for exposing data via any ODBC compliant database and a client component is also provided that can send updates from an SdShare feed to any SPARQL 1.1 compliant endpoint and BrightstarDB instance.

The following diagram shows the server architecture.



The SdShare server manages multiple 'Production Adaptors' that connect to an underlying data source and expose the data contained as an SdShare Feed. The adaptors implement a simple interface allowing new data sources to be easily added. Adaptors for ODBC sources are provided with the server.

The server can also host client adaptors. This can be in the same server instance or located on another machine.

The client adaptors get notified of changes on the feeds and can then process the resource data as it becomes available.

A client adaptor that updates a SPARQL endpoint is provided.

CHAPTER 25

Building BrightstarDB

This section will take you through the steps necessary to build BrightstarDB from source.

25.1 Prerequisites

Before you can build BrightstarDB you need to install the following tools.

1. **Visual Studio 2013**

   You can use the Professional or Ultimate editions to build everything.

   TBD: Check what can be built with VS 2013 Express

2. **MSBuild Community Tasks**

   You must install this from the MSI installer which can be found at http://code.google.com/p/msbuildtasks/downloads/list. The MSI installer will install the MSBuild Community Tasks extension in the right place for it to be found by our build scripts.

3. **OPTIONAL: WiX**

   WiX is required only if you plan to build the installer packages for BrightstarDB. You can download WiX from http://wixtoolset.org/. The build is currently based on version 3.5 of WiX, but it is recommended to build with the latest version of WiX (3.7 at the time of writing).

4. **OPTIONAL: Xamarin.Android, Xamarin.iOS**

   Xamarin.Android is required only if you plan to build the Android package for BrightstarDB, and Xamarin.iOS is needed to build the iOS package. Please read *Developing Portable Apps* first!

25.2 Getting The Source

The source code for BrightstarDB is kept on GitHub and requires Git to retrieve it. The easiest way to use Git on Windows is to get the GitHub for Windows application from http://windows.github.com/. Alternatively you can download the Git installation package from http://git-scm.com/. If you do not want to install Git and are happy to simply work with a snapshot of the code, the GitHub website offers ZIP file packages of the source tree.

Branches

The BrightstarDB source code is organized into multiple branches. The most important ones are **develop** and **master**.

The **develop** branch is the latest development version of the source code. Most of the time the code on **develop** should be stable in as much as it compiles and the unit tests all pass. However, occasionally this is not the case.

The **master** branch only gets updated when a new release is ready, so the head of the **master** branch will be the source code for the last release.

Branches with the name **release/X.X** contain the source code for the named release.

Branches with the name **feature/XXX** contain work in progress and should be regarded as unstable.

Cloning With GitHub For Windows

To retrieve a clone of the Git repository simply go to https://github.com/BrightstarDB/BrightstarDB and on the right-hand side of the page you will see a button labelled "Clone in Desktop". Click on that button to launch GitHub for Windows and start the process of cloning the repository. Once you have the cloned repository you can then use the GitHub for Windows GUI to select the branch you want to work with.

Cloning With Git

To clone over HTTPS use the repository URL https://github.com/BrightstarDB/BrightstarDB.git To clone over SSH use *git@github.com:BrightstarDB/BrightstarDB.git*. Note that cloning over SSH requires that you have an SSH key set up with GitHub.

Downloading a source ZIP

You can download the source code on a given branch as a ZIP file if you want to avoid using Git. To do this, go to https://github.com/BrightstarDB/BrightstarDB and select the branch you want to download from the drop-down box. Then use the 'Download ZIP' button to retrieve the source.

25.3 MSBuild build.proj

The quickest and simplest way to build BrightstarDB is to use the build.proj MSBuild script. This script is found in the top-level directory of the BrightstarDB source.

The script uses the following properties:

Configuration The project configuration to be built. Can be either 'Debug' or 'Release'. Defaults to 'Debug'.

NUnitPath The path to your NUnit bin directory. This is used when running the unit tests. Defaults to *C:Program Files (x86)NUnit 2.6.2bin*

You can either override these properties on the command-line using /p:{Property}={Value} switches or you can edit the build.proj file (the properties are defined at the top of the file).

The MSBuild script has the following targets:

BuildAndTest Build Core, Mobile, Portable and Tools and run the unit tests. This is the default target that will be run if you don't specify a /t:{Target} switch on the command-line.

BuildCore Performs a clean build of the core .NET 4.0 library only.

BuildMobile Performs a clean build of the Windows Phone library only.

BuildTools Performs a clean build of the Polaris tool.

Test Run Core and Portable unit tests

TestCore Run Core unit tests only

TestPortable Run Portable unit tests only

Note: The `build.proj` script is provided to make it easy to locally build and test BrightstarDB. It does not contain targets for building release packages. The process for building a full release is a little more involved and requires more pre-requisites to be installed. This is documented below.

25.4 Building The Core Solution

The core BrightstarDB solution can be found at `src\core\BrighstarDB.sln`. This solution will build BrightstarDB's .NET 4 assemblies as well as the BrightstarDB service components including the Windows service wrapper.

The BrightstarDB solution uses a some NuGet packages which are not stored in the Git repository, so the first time you open the solution you will need to restore the missing packages. To do this, right-click on the solution in the Solution Explorer window in Visual Studio and select **Manage NuGet Packages for Solution...**. In the dialog that opens you should see a message prompting you to restore the missing NuGet packages.

Once the NuGet packages are restored you can build the entire solution either from within Visual Studio or from the command-line using the MSBuild tool.

25.5 Running the Unit Tests

The core solution's unit tests are all written using the NUnit framework. The easiest way to run all the unit tests is to use the unit test project file from the command prompt. To do this, open a Visual Studio command prompt and cd to the `src\core` directory under the BrightstarDB source. Then run the unit tests with:

```
msbuild unittests.proj
```

25.6 Building the Portable Class Libraries

The source code for the Portable Class Library and the platform-specific assemblies are all contained in `src\portable`. There are three separate solution files.

- portable.sln - this builds the core PCL assembly and the Desktop, Windows Phone, Silverlight and Windows Store platform assemblies.
- android.sln - this solution builds the core PCL assembly and the Android platform assembly only.
- ios.sln - this solution builds the core PCL assembly and the iOS platform assembly only.

All three solutions are intended for use in Visual Studio 2013. It has not been possible to make these solutions build under MonoDevelop / Xamarin Studio due to some of the features used in the .csproj files.

To build the Android libraries from source you will require an installation of Xamarin.Android at Indie level or above. Unfortunately once BrightstarDB is included the built application size will exceed the maximum supported by the Free version of Xamarin.Android.

To build the iOS libraries from source you will require an installation of Xamarin.iOS. This configuration has not been tested in the free version of Xamarin.iOS.

As with the core solution, the portable class library solution has some NuGet dependencies which need to be downloaded. Follow the same steps outlined above for the core solution to download and install the dependencies before trying to build this solution from the command line.

This solution also requires that you have a Windows 8 developer license installed. You should be prompted by to retrieve and install this license if necessary when you first open the solution file in Visual Studio.

25.7 Building The Tools

The `src\tools` directory contains a number of command-line and GUI tools including the Polaris management console. Each subdirectory contains its own Visual Studio solution file. As with the core solution, NuGet packages may need to be restored, so when opening the solution file for the first time right-click on the solution in the Solution Explorer window and select **Manage NuGet Packages for Solution...** and if necessary follow the prompt to download an install missing NuGet packages.

25.8 Building The Documentation

Documentation for BrightstarDB is in two separate parts.

Developers Guide / User Manual

The developer and user manual (this document) is maintained as RestructuredText files and uses Sphinx to build.

Details on getting and using Sphinx can be found at http://sphinx-doc.org/. Sphinx is a Python based tool so it also requires a Python installation on your machine. You may just find it easier to get the pre-built documentation from http://brightstardb.readthedocs.org/

API Documentation

The API documentation is generated using Sandcastle Help File Builder. You can get the installer for SHFB from http://shfb.codeplex.com/. The .shfbproj file for the documentation is at `doc/api/BrightstarDB.shfbproj`. To build the documentation using this project file you must first build the Core in the Debug configuration.

25.9 Building Installation and NuGet Packages

An MSBuild project is provided to compile and build a complete release package for BrightstarDB. This project can be found at `installer\installers.proj`. The project will build all of the libraries and documentation and then make MSI and NuGet packages.

Note: Building the full installer solution requires all the pre-requisites listed above to be installed. It also requires that you have first restored NuGet dependencies in both the core solution and the tools solution as described in the sections above.

25.10 Building Under Mono

Please see *Building From Source* in the section *Using BrightstarDB Under Mono*

CHAPTER 26

Using BrightstarDB Under Mono

This section covers how to use the BrightstarDB libraries and server in a Mono environment as well as how to build BrightstarDB from source using Mono.

> **Warning:** The use of BrightstarDB under Mono should be considered to be in alpha status. It is certainly not yet ready for production use.

We would welcome any bug reports and especially reproducible errors in BrightstarDB under Mono to help us improve the stability of the code on this platform.

26.1 Using BrightstarDB Libraries

The best way to use BrightstarDB libraries in your Mono application is to retrieve the packages via NuGet. There are details about using NuGet under mono in the NuGet FAQ.

The .NET 4.0 version of BrightstarDB should work correctly under the latest version of Mono (3.2.4 at the time of writing). It will probably not work under older versions of Mono.

26.2 Building From Source

You can also build the core BrightstarDB libraries under Mono. Refer to *Getting The Source* for information about getting hold of BrighstarDB source code.

Note: You will need to build from the source code for release 1.5 or from the develop branch in order to have the Mono build scripts.

The build scripts for mono are contained in the `mono` directory at the top level of the project. The build scripts use NuGet to retrieve dependency package and a version of NuGet.exe is included in the BrightstarDB repository to assist in that. However, NuGet requires some SSL certificates to be registered otherwise download will fail. Before trying to build under Linux please execute the following commands:

```
sudo mozroots --import --machine --sync
sudo certmgr -ssl -m https://go.microsoft.com
sudo certmgr -ssl -m https://nugetgallery.blob.core.windows.net
sudo certmgr -ssl -m https://nuget.org
```

Once this is completed, it should be possible to build BrightstarDB simply by executing the `build.sh` script from within the `mono` directory. The resulting binaries can then be found in `mono/build`. At the time of writing the build script builds the following items:

- `BrightstarDB.dll` : the core BrightstarDB library
- `BrightstarDB.Server.Modules.dll`: the core BrightstarDB server library
- `service` : a directory containing the BrightstarDB server runner and all of its dependencies

26.3 Running a BrightstarDB Server

26.3.1 Self-Hosted

Assuming that you have built BrightstarDB from source as described above, the server can be run from within `mono/build/service`.

> **Warning:** Before you run the service for the first time you must edit the *BrightstarService.exe.config* file in `mono/build/service` as this file is copied out of the Windows build and so contains DOS path names. You need to edit the path for the log file (in the *system.diagnostics* section) and the *storesDirectory* path in the connection string specified in the *brightstarService* section.

To start the server simply run the following:

```
mono BrightstarService.exe
```

The service will start listening on port 8090 at the path /brightstar. So from a local machine you can access the service from a browser pointed at http://localhost:8090/brightstar.

TBD: Document how to run the BrightstarDB server in Apache using mod_mono

TBD: Document how to secure the BrightstarDB server under Mono

26.4 Unit Tests

The unit tests for Mono can be run using the `test.sh` script in to `mono` directory. At present these tests do not successfully run to completion, but this is being worked on.

CHAPTER 27

Running BrightstarDB in IIS

It is possible to run BrightstarDB as a web application under IIS. This can be useful if you want to integrate BrightstarDB services into an existing website or to make use of IIS-specific features such as authentication.

27.1 Installation and Configuration

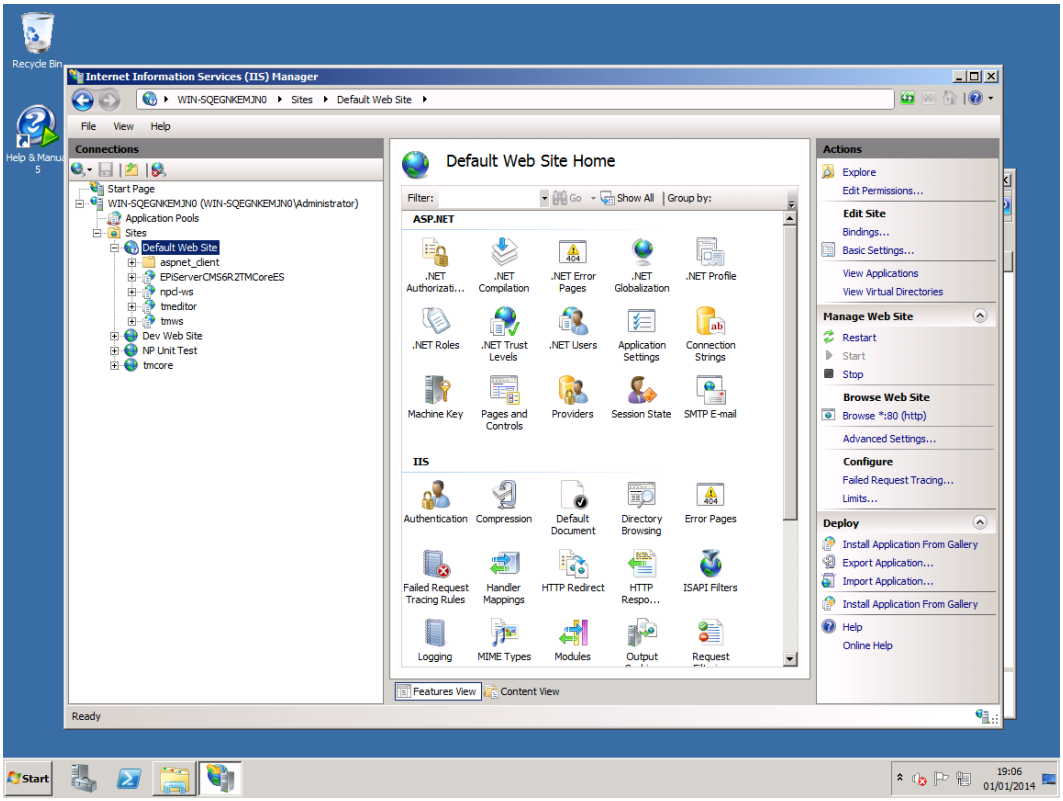
The BrightstarDB service is a NancyFX application and so you can refer to the NancyFX wiki page Hosting Nancy with ASP.NET and other pages of the NancyFX wiki for in-depth details. However, we present here a simple way to get started using IIS to host BrightstarDB.

1. **Install BrightstarDB**

   The best option is to install BrightstarDB from the .EXE installer as this will create the web application directory for you. The rest of this short guide assumes that you have used the installer and have installed in the default location of *C:\Program Files\BrightstarDB* - if you installed somewhere else the paths you use will be different.

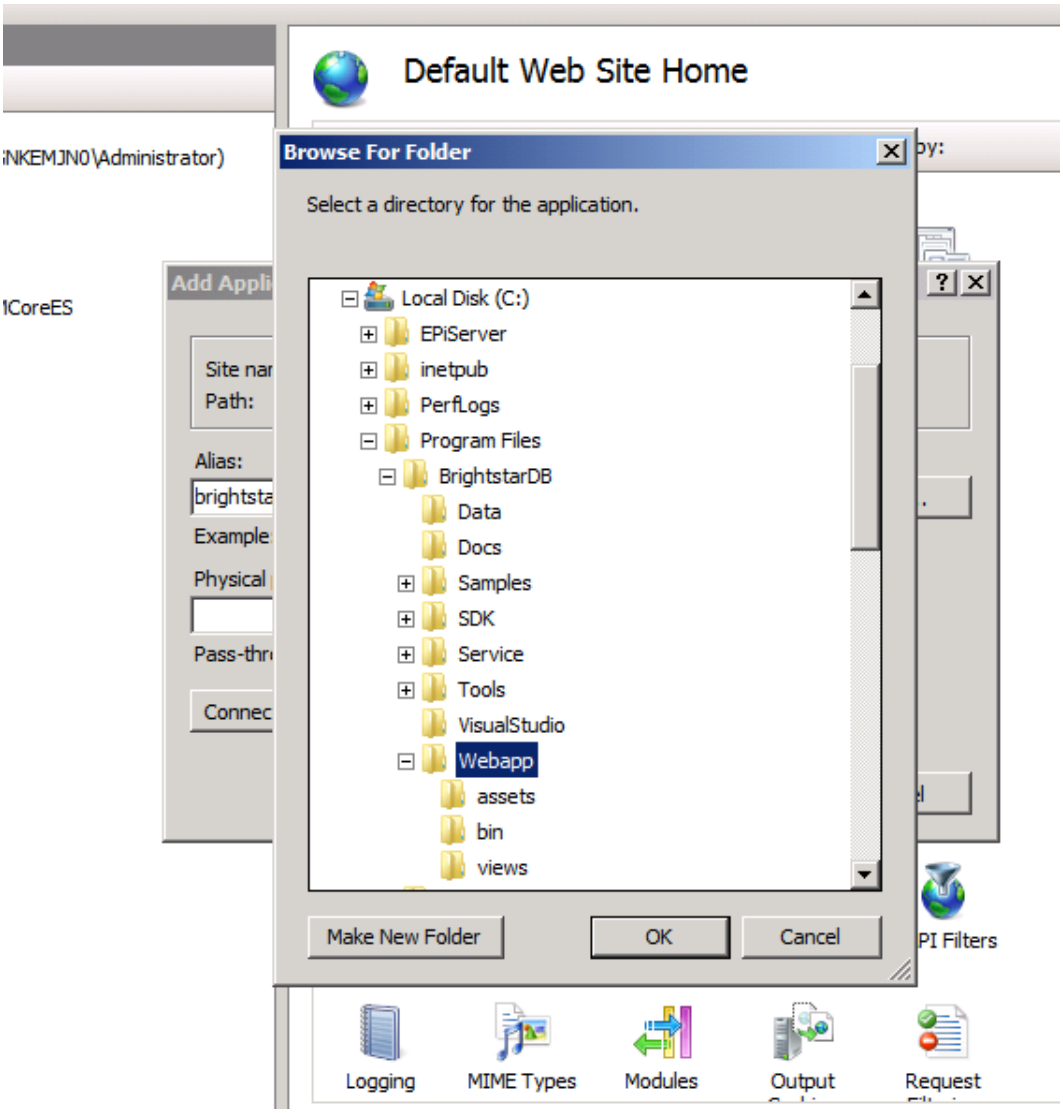
2. **Create a website in IIS**

   You can skip this step if you are planning to add BrightstarDB to an existing site. In this example we are going to add BrightstarDB to the default website which, as you can see from the screenshot already hosts several other web applications.



3. **Add an application to the website**

   Right-click on the website and select “Add Application...”. In the dialog that comes up enter the alias for the application (in this example the application alias is “brightstardb”, but you can choose some other alias if you prefer). To set the Physical Path click on the ”..” button, browse to *C:\Program Files\BrightstarDB\Webapp* and click OK to choose that folder.



For the application pool choose an existing application pool that runs .NET Framework version 4.0 with Pipeline mode: Integrated. By default IIS7 has an app pool named ASP.NET 4.0 which has this configuration, but you may want to or need to choose another app pool or create a new app pool for running BrightstarDB. In any case, you should remember the name of the application pool you create and the identity that the application pool runs under.

4. **(OPTIONAL) Change data directory**

   The data directory is the location where BrightstarDB stores its files. This should be a directory outside the IIS application folders. By default the BrightstarDB web application is configured to use the *Data* folder under the location where BrightstarDB is installed (e.g. *C:\Program Files\BrightstarDB\Data*). To change this to a different location, open the *web.config* file in the web application directory and locate the line:

   ```
   <brightstarService connectionString="type=embedded;storesDirectory=c:\Program Files\brightst
   ```

   and change the path to the directory you want to use.

Warning: If you are running BrightstarDB both as a Windows Service and as an IIS application, the two applications MUST NOT use an embedded connection to the same stores directory. If you want to have the IIS web application share the same data as the Windows service, then change the connection string for the web application to use a REST connection to the Windows service (or vice-versa):

```
<brightstarService connectionString="type=rest;endpoint=http://localhost:8090/brightstar">
```

5. **Set Directory Access Privileges**

   The final step is to ensure that the application pool that runs the BrightstarDB web application has the permissions required to create and delete files and directories under the data directory. To do this:

   (a) Use Windows Explorer to locate the data directory (if it does not already exist, create it).

   (b) Now right-click on the folder for the data directory and select "Properties".

   (c) Go to the "Security" tab, and click on "Edit...".

   (d) In the dialog that is displayed click on the button labelled "Add..."

   (e) Enter the identity of the application pool that is running the BrightstarDB web application. If the application pool is set to a local user identity, enter the name of the user here. If the application pool is set to use a domain user identity, enter the name of the user as DOMAINUser Name. If the application pool is set to use AppPoolIdentity, enter the name of the user as IIS AppPoolApp Pool Name.

   (f) Click "OK" and then in the Permissions dialog check the box for "Full control" under the "Allow" column for the user you just added to the permissions. The result should be something like shown in the screenshot below.

   (g) Click "OK" to exist the Permissions dialog and "OK" again to exist the Properties dialog.

6. **Browse the Site**

   You should now be able to connect to your site and the BrightstarDB application underneath it. By default the application name will be part of the path, so if you installed the application with the alias "brightstardb" under a server accessible as http://localhost/, then the URL for the BrightstarDB service will be http://localhost/brightstardb/

CHAPTER 28

What's New

This section gives a brief outline of what is new / changed in each official release of BrightstarDB. Where there are breaking changes, that require either data migration or code changes in client code, these are marked with **BREAKING**. New features are marked with NEW and fixes for issues are marked with FIX.

28.1 BrightstarDB 1.8 Release

- NEW: EntityFramework now supports GUID properties.
- **NEW: EntityFramework now has an [Ignore] attribute which can be used to decorate interface properties** that are not to be implemented by the generated EF class. See the *guide to EF Annotations* for more information.
- **NEW: Added a constructor option to generated EF entity classes that allows property initialisation in the constructor. Tha** the suggestion.
- **NEW: Added some basic logging support for Android and iOS PCL builds. These builds now log diagnostic messages whe** and the BrightstarDB logging subsystem can be initialized with a local file name to generate persistent log files in Release configuration.
- NEW: It is now possible to iterate the distinct predicates of a data object using the GetPropertyTypes method.
- FIX: Fix for Polaris crash when attempting to process a query containing a syntax error.
- FIX: Fixed NuGet packaging to remove an obsolete reference to Windows Phone 8. WP8 (and 8.1) are still both supported but as PCL profiles.
- **FIX: Performance fix for full cache scenarios. When an attempt to evict items out of a full cache results in no items being** process will not be repeated again for another minute to allow for any current update transactions that have locked pages in the cache to complete. This can avoid a lot of unnecessary cache scans when a large update transaction is being processed. Thanks to CyborgDE for the bug report.

28.2 BrightstarDB 1.7 Release

- **BREAKING: BrightstarDB no longer supports Windows Phone 7 development. Due to changes in the** libraries that we use there is now only a Portable Class Library build available which targets .NET 4.5, Windows Phone 8, Silverlight 5, Windows Store apps and Android. iOS support is in the pipeline.
- **NEW: EXPERIMENTAL support has been added for using DotNetRDFs virtual nodes query facility.** This feature can improve query performance by reducing the number of times that RDF resource values

need to be looked up. There are still some bugs left to be ironed out in this feature so it should not be used in production. To enable this feature set BrightstarDB.Configuration.EnableVirtualizedQueries to true.

- **NEW: Added support for non-existence preconditions on transactional updates. This precondition** fails if one or more of the specified triples already exists in the store prior to executing the update. See *Transactional Update*.
- **NEW: Added support for generated and composite keys for entities. See *Key Properties and Composite Keys*.** This includes a new type-based unique constraint check for entities with generated or composite keys.
- NEW: RDF/XML is now supported as an export format.
- **NEW: It is now possible to retrieve an IEntitySet from the Entity Framework context using the EntitySet<T>()** method on the context object. Thanks to NZ_Dig for the contribution.
- **FIX: Fixed the way that the BrightstarDB Entity Framework handles the case where the same RDF property has** a domain or range of multiple classes. The collections provided by Entity Framework now filter to exclude resources which are not of the expected type rather than trying to coerce the resources into the expected type. This leads to more consistent OO behaviour. Thanks to NZ_Dig for the bug report.
- **FIX: Added guard statements to PCL implementation of ConcurrentQueue<T> to avoid InvalidOperationExceptions** being raised and then immediately handled in the case of an empty queue being accessed.
- **FIX: Major overhaul of the BinaryFilePageStore (the basis of the rewrite store type). This fixes a number of** issues found under the PCL build and also introduces support for background writing of page updates to improve update performance. Thanks to CyborgDE for the bug report.
- **FIX: Replaced polling loop with proper synchronized handling of job status changes in the embedded store** implementation. Thanks to CyborgDE for the fix.
- FIX: A number of fixes to the JS used in the browser interface to the BrightstarDB server.
- FIX: Reinstated logging for the BrightstarDB service.
- FIX: Removed dependency on external System.Threading.Tasks DLL
- NEW: Jobs are now given a default name if one is not specified when they are created.

28.3 BrightstarDB 1.6.2 Release

- **FIX: Fixed an error in the LRU cache implementation that could corrupt the cache during import / update operations.** Thanks to pcoppney for the bug report.
- FIX: Fixed version number specified in the setup bootstrapper and reported when looking at the installed programs under Windows.

28.4 BrightstarDB 1.6.1 Release

- FIX: Restored default logging configuration for BrightstarDB service
- **FIX: Fix for wildcard delete patterns in a transaction processed against a SPARQL endpoint.** Thanks to feugen24 for the bug report and suggested fix.
- **FIX: SPARQL endpoint connection strings now default the store name to "sparql". Thanks to** feugen24 for raising the bug report.
- FIX: Fixed sample projects included in the MSI installer. Thanks to aleblanc70 for the bug report.

- **NEW: Added platform-specific default configuration settings and removed dependency on** third-party System.Threading.Tasks.dll from Windows Phone build.

28.5 BrightstarDB 1.6 Release

- NEW: Added experimental support for Android.
- **NEW: Jobs created through the API can now be assigned a user-defined title string, this will be displayed / returned** when the jobs are listed.
- NEW: Entity Framework internals allow better constructor injection of configuration parameters.
- **NEW: Entity Framework will now "eagerly" load the triples for entities returned by a LINQ query in a wider number of** circumstances, including paged and sorted LINQ queries.
- **NEW: Added a utility class to the API for retrieving the namespace prefix declarations used by entity classes and** formatting them for custom SPARQL queries or Turtle files.
- **NEW: Export job now has an additional optional parameter to specify the export format. Currently only NTriples and NQ** are supported but this will be extended to support other export syntaxes in future releases.
- **NEW: Added support to the BrightstarDB server for using ASP.NET membership and role providers to secure access to th** and its stores. For more information please refer to the section *Configuring Authentication*.
- **BREAKING: The connection string syntax for connections to generic SPARQL endpoints and to other RDF stores via dot** has been changed. Please refer to the section *Connection Strings* for more information.
- FIX: Fix for bug in reading back through multiple entries in the store statistics log.
- FIX: Fixed the New Job form in the browser interface for the BrightstarDB server so that it properly resets on page load.
- FIX: Fixed the New Job form to allow Import and Export jobs to be created without requiring a Graph URI.
- FIX: Fix for concurrency bug in Background Page Writer - with thanks to Michael Schulte for the bug report and suggested fix.

28.6 BrightstarDB 1.5.3 Release

- FIX: Fixes a packaging issue with the Polaris tool in the 1.5.2 release.

28.7 BrightstarDB 1.5.2 Release

- FIX: Fixed a regression bug in the SPARQL query template for the browser interface to the BrightstarDB server.
- FIX: Added missing sizing parameters to the SPARQL results text box in the browser interface.
- FIX: Fixed browser interface for SPARQL queries to not report an error when the form is initially loaded.

28.8 BrightstarDB 1.5.1 Release

- FIX: Fixed the default connection string used in the NerdDinner sample.
- NEW: Installer now supports installing the VS extensions into VS2013 Professional edition and above.

- NEW: Overhaul of the SPARQL query APIs to allow the specification of both SPARQL results format and RDF graph format. This allows RDF formats other than RDF/XML to be returned by CONSTRUCT and DESCRIBE queries. For more information please refer to *Querying data using SPARQL*
- NEW: Added an override for GetJobInfo to list the jobs recently queued or executed for a store. Refer to *Jobs* for more information.

28.9 BrightstarDB 1.5 Release

- **BREAKING** : The WCF server has been replaced with an HTTP server with a full RESTful API. Connection strings of type `http`, `tcp` and `namedpipe` are no longer supported and should be replaced with a connection string of type `rest` to connect to the HTTP server. The new HTTP server can be run under IIS or as a Windows Service and the distribution includes both of these configuration options. For more information please refer to *Running BrightstarDB*. The configuration for the server has also been changed to enable more complex configuration options. The new configuration structure is detailed in *Running BrightstarDB*. Please note when upgrading from a previous release of BrightstarDB you may have to manually edit the server configuration file as an existing configuration file cannot be overwritten if it was locally modified.
- **BREAKING**: The SDShare server has been removed from the BrightstarDB package. This component is now managed in a separate Github repository (https://github.com/BrightstarDB/SDShare)
- **BREAKING**: RDF literal values without an explicit datatype are now exposed through the Data Objects and Entity Framework APIs as instances of the type `BrightstarDB.Rdf.PlainLiteral` rather than as `System.String`. This change has been made to better enable the APIs to deal with RDF literals with language tags. This update allows both dynamic objects and Entity Framework interfaces to have properties typed as `BrightstarDB.Rdf.PlainLiteral` (or an `ICollection<BrightstarDB.Rdf.PlainLiteral>`). The LINQ to SPARQL implementation has also been updated to support this type. However, this change may be **BREAKING** for some uses of the API. In particular when using either the dynamic objects API or the SPARQL results set `XElement` extension methods, the object returned for an RDF plain literal result will now be a `BrightstarDB.Rdf.PlainLiteral` instance rather than a string. The fix for this breaking change is to call `.ToString()` on the `PlainLiteral` instance. e.g:

```
// This comparison will always return false as the object returned by
// GetColumnValue is a BrightstarDB.Rdf.PlainLiteral
bool isFoo = resultRow.GetColumnValue("o").Equals("foo");

// To fix this breaking change insert .ToString() like this:
bool isActuallyFoo = resultRow.GetColumn("o").ToString().Equals("foo");

// Or for a more explicit comparison
bool isLiteralFoo = resultRow.GetColumn("o").Equals(new PlainLiteral("foo"));
```

- NEW: Job information now includes date/time when the job was queued, started processing and completed processing.
- NEW: BrightstarDB installer now includes both 32-bit and 64-bit versions and will install into `C:\Program Files\` on 64-bit platforms.
- NEW: Added shell scripts for building BrightstarDB under mono.
- NEW: BrightstarDB Entity Framework and Data Objects APIs can now connect to stores other than BrightstarDB. This includes the ability to use the Entity Framework and DataObjects APIs with generic SPARQL 1.1 Query and Update endpoints, as well as the ability to use these APIs with other stores supported by DotNetRDF. For more information please refer to *Connecting to Other Stores*
- FIX: Fixed incorrect handling of \ escape sequences in the N-Triples and N-Quads parsers.

- FIX: BrightstarDB now uses NuGet to provide the DotNetRDF library rather than using a local copy of the assemblies.

28.10 BrightstarDB 1.4 Release

- NEW: Stores can now extract and persist basic triple count statistics. See *Store Statistics* for more information.
- NEW: Stores can now be cloned into a new snapshot store. For stores using the append-only storage mechanism, a snapshot can be created from any previous commit point. See *Creating Store Snapshots* for more information
- NEW: Added support for System.Uri typed properties in Entity Framework. Thanks to github user jhashemi for the suggestion.
- NEW: Portable class library build. Refer to *Developing Portable Apps* for more information.
- NEW: Dynamic objects and Entity Framework APIs now support named graphs.
- FIX: Reduced memory usage for BTree's by half.
- FIX: Fixed a memory leak in the page cache code that prevented expired pages from being released to the garbage collector.
- FIX: Fixed the resource ID and resource caches to support a (configurable) limit on the number of entries cached.
- FIX: Fixed error in deleting an entity from the same entity framework context in which it was originally created. Thanks to github user cmerat for the report.
- FIX: Fixed EntityFramework code to clean up InverseProperty collections correctly. Thanks to BrightstarDB user Alan for the bug report.
- FIX: Fixed EntityFramework text template code for matching class names in generic collection properties. Thanks to github user Xsan-21 for the bug report.
- FIX: Fix for Polaris hanging when trying to process a GZipped NTriples file.

28.11 BrightstarDB 1.3 Release

- NEW: First official open source release. All documentation and examples updated to remove references to commercial licensing and license protection code. Build updated to remove dependencies on third-party commercial tools
- NEW: The ExecuteTransaction method now supports specifying a target graph.
- NEW: The ExecuteQuery Method now supports specifying the default graph of the SPARQL dataset.
- FIX: Disabled profiling code that was eating up significant amounts of memory during long running imports. Profiling can now be enabled globally by calling Logging.EnableProfiling(true);

28.12 BrightstarDB 1.2 Release

- NEW: Collection properties on entities now support compiling LINQ queries to SPARQL. This can be achieved by using the AsQueryable() method on the collection. e.g. myEntity.RelatedItems.AsQueryable()..../// LINQ query follows

- NEW: Interface and property annotations are now copied from the entity interface to the entity class by the code generator. This applies only to annotations that are not in the BrightstarDB namespace. For interface annotations, only those annotations that are also applicable to classes can be copied through to the generated class. For more information please refer to the section *Annotations* in the *Entity Framework* API documentation.
- NEW: BrightstarDB now supports XML, JSON, CSV and TSV (tab-separated values) as SPARQL reults formats. You can specify the format you want using the optional SparqlResultsFormat parameter on the ExecuteQuery methods. The SPARQL service samples has been updated to select the appropriate results format depending on the requested content type.
- NEW: BrightstarDB generated entity classes now implement the System.ComponentModel.INotifyPropertyChanged interface and fire a notification event any time a property with a single value is modified. All collections exposed by the generated classes now implement the System.Collections.Specialized.INotifyCollectionChanged interface and fire a notification when an item is added to or removed from the collection or when the collection is reset. For more information please refer to the section *INotifyPropertyChanged and INotifyCollectionChanged Support*.

28.13 BrightstarDB 1.1 Release

- FIX: Entity Framework code generation now supports multiple levels of inheritance on interfaces.
- NEW: Polaris now supports editing the server connection details
- NEW: Installer now adds the BrightstarDB item templates for EntityContext and Entity to VS2012 Professional and above. VS2010 and VS2010 Express are also still supported. Please note that VS2012 Express editions are not supported at this time.

28.14 BrightstarDB 1.0 Release

- NEW: Added support for executing SPARQL Update commands to *Polaris*
- FIX: A few minor bug fixes

28.15 BrightstarDB 1.0 Release Candidate

This release introduces a BREAKING file format change. If you are upgrading from a previous version of BrightstarDB and you wish to retain the data in a store, you should export all data from that store before performing the upgrade and then after the upgrade delete and recreate the store and import the exported data.

- BREAKING: Store file format is significantly different from previous versions - please read the warning information above carefully BEFORE upgrading.
- NEW: Store now supports a file format that reduces index file growth rate

28.16 BrightstarDB 1.0 Public Beta Refresh

This release introduces some BREAKING API changes (but data store format is unaffected, so only your code needs to be modified). If you are upgrading from a previous release, please read the following carefully - in particular note the BREAKING changes that are introduced in this release.

- BREAKING: All API namespaces have now changed from NetworkedPlanet.Brightstar.* to BrightstarDB.*. Custom code will require modification and recompilation
- BREAKING: The only DLL now required for the .NET 4.0 SDK is BrightstarDB.dll.
- BREAKING: Entity sets exposed by the generated Entity Framework context class are now typed by the implementation class rather than the entity interface class. Code written on top of the Entity Framework will need to be refactored to use the interface rather than the concrete class or to cast the return values to the concrete class where necessary. Note, this reverses the change made in the Public Beta release.
- BREAKING: The default installation directory and by extension the default data store directory has changed from C:Program Files (x86)NetworkedPlanetBrightstar to C:Program Files (x86)BrightstarDB. If using the default data directory path, after upgrading you should manually copy the contents of C:Program Files(x86)NetworkedPlanetBrightstarData to C:Program Files (x86)BrightstarDBData.
- NEW: Added support for binding BrightstarDB data objects to .NET dynamic objects. For more information please refer to the section *Dynamic API*.
- NEW: Added an optional SPARQL endpoint implementation that runs in IIS allowing BrightstarDB to be exposed as a SPARQL 1.1 endpoint. For more information please refer to the *SPARQL Endpoint* section of the documentation.
- NEW: The BrightstarService service executable now supports specifying the base directory, HTTP and TCP ports and named pipe that the service listens on as command-line parameters
- NEW: The BrightstarDB API has been extended to add support for importing / exporting named graphs and for executing a transaction against a named graph.
- NEW: Added support for SPARQL 1.1
- NEW: Added support for SPARQL UPDATE
- NEW: SPARQL support now includes support for querying named graphs.
- NEW: EntityFramework now supports the use of enum property types (including Flags and Nullable enum types)
- NEW: EntityFramework now surfaces an event that is invoked immediately before changes are saved to the store. For more information please see the section *SavingChanges Event*.
- FIX: The XML Schema "date" datatype (`http://www.w3.org/2001/XMLSchema#date`) is now recognized and mapped to a System.DateTime value by EntityFramework.
- NEW: Added support for the LINQ .All() filter operator.
- FIX: The WCF service mode for the BrightstarDB service now supports concurrent requests.
- FIX: Several bug fixes for LINQ to SPARQL query generation
- NEW: BrightstarDB now supports import of a number of additional RDF syntaxes as documented in the section *Supported RDF Syntaxes*.

28.17 BrightstarDB Public Beta

- FIX: Several performance fixes and the introduction of configurable client and server-side caching have significantly improved the speed of SPARQL and LINQ queries. For information about configuring caching please refer to the section *Caching*.
- NEW: BrightstarDB Entity Framework now adds support for creating an OData provider. For more information please see the *OData* section of the *Entity Framework* API documentation.

- NEW: LINQ-to-SPARQL now has support for a number of additional String functions. For details please refer to the section *LINQ Restrictions*.
- NEW: Optimistic locking support has been added to the *Data Object Layer* and *Entity Framework*.
- BREAKING: Entity sets exposed by the generated Entity Framework context class are now typed by the entity interface rather than the generated implementation class. Code written on top of the Entity Framework will need to be refactored to use the interface rather than the concrete class or to cast the return values to the concrete class where necessary.
- NEW: Logging is now performed through the standard .NET tracing framework, removing the dependency on Log4Net. Please refer to the section *Logging* for more information.
- NEW: Polaris now supports saving SPARQL queries between sessions and configuring commonly used URI prefixes to make it quicker and easier to write SPARQL queries and transactions. These features are documented in the section *Polaris Management Tool*.

28.18 BrightstarDB Developer Preview Refresh

- BREAKING: A number of changes and improvements to data file format means that databases created with the initial Developer Preview cannot be used with the Developer Preview Refresh.
- NEW: Windows Phone 7.1 support. It is now possible to create applications that target Windows Phone OS 7.1 with BrightstarDB. Databases are portable between the desktop / server and the mobile version of BrightstarDB.
- NEW: The *Data Object Layer* is now publicly exposed and documented for developers to use as a mid-point between the low-level RDF Client API and the data-binding provided by the Entity Framework.
- BREAKING: Replaced the use of Log4Net with standard Microsoft tracing. This provides more easily configurable logging and tracing functionality.
- NEW: Polaris now provides the ability to view the previous states of a BrightstarDB store, run queries against them, and revert the database to a previous state if required.
- NEW: Polaris now provides keyboard shortcuts for menu items and a right-click context menu on the store list.
- FIX: The range of native datatypes supported by the EntityFramework has been greatly expanded.
- FIX: The scope of LINQ support by EntityFramework is now better documented,
- NEW: EntityFramework now supports String.StartsWith, String.EndsWith and Regex.IsMatch methods for string filtering in LINQ queries.
- NEW: BrightstarDB now provides support for conditional update. This functionality is used to provide optimistic locking support for the Data Object Layer and EntityFramework.
- NEW: NerdDinner sample now includes examples of a .NET MembershipProvider and RoleProvider implemented on BrightstarDB.
- NEW: EntityFramework now supports properties that are an ICollection<T> of native types such as string, int etc.
- BREAKING: The GetColumnValue extension method on XDocument now returns a typed object rather than a string whenever the bound variable's datatype is a recognized XML Schema datatype.
- FIX: EntityFramework now supports inheritance on Entity interfaces.
- FIX: The service contract for the BrightstarDB WCF service now has a proper URI: http://www.networkedplanet.com/schemas/brightstar.
- BREAKING: ICommitPointInfo and ITransactionInfo interfaces have been significantly reworked to provide better history information for BrightstarDB stores.

- FIX: SPARQL results XML document generated by the Brightstar service now escapes all reserved XML characters in the binding values.
- FIX: Added an optimization for the SPARQL query generated by LINQ expressions that simply retrieve an entity by its identifier.
- NEW: Added more documentation and samples, especially for Windows Phone 7 applications and the *Admin APIs*.

CHAPTER 29

Known Issues

29.1 SPARQL Queries

When using the less-than (<) symbol in SPARQL queries, it is necessary to put spaces between the symbol and the rest of the query to avoid a parser error. For example the following query will fail with a parser error::

```
SELECT ?p ?s WHERE { ?p a <http://example.org/schema/person> . ?p <http://example.org/schema/salary>
```

but the same query written as shown below will be processed correctly.:

```
SELECT ?p ?s WHERE { ?p a <http://example.org/schema/person> . ?p <http://example.org/schema/salary>
```

29.2 Entity Framework Tooling

'_' underscore characters are not allowed in the names of the namespace(s) containing the interfaces that are to be generated into entity classes.

Currently only the following versions of Visual Studio are provisioned with the Entity Framework item templates through the installer:

- Visual Studio C# Express 2010
- Visual Studio 2010 Professional and above
- Visual Studio 2012 Professional and above

To create an entity context class in other versions of Visual Studio, we recommend that you copy the .tt file from one of the Entity Framework samples into your own project. You may rename the file if you wish as long as you retain the .tt file extension.

29.3 OData Functions

The filter function 'replace' is not supported.

29.4 Avoid HTML Named Entities in String Values

Using HTML named entities in string values that are not also valid XML named entities will result in errors when parsing the SPARQL results if these string values are included in the results set. Examples of such entities are £

for a pound-symbol, © for a copyright symbol etc. It is best to avoid this situation by converting all HTML named entities to their numeric entity form before storing them in BrightstarDB (e.g. £ instead of £). A full list of HTML named entities and their numeric equivalents for HTML 4 can be found at http://www.w3.org/TR/WD-html40-970708/sgml/entities.html.

CHAPTER 30

Getting Support

If you need support while working with BrightstarDB there are two primary channels for asking for help. All BrightstarDB users are invited to join our Codeplex Discussion Forum. In this forum you can ask questions and see the latest postings from the BrightstarDB team.

You can also optionally purchase a support contract from NetworkedPlanet Limited. Support contracts last for a full year and provide you with email support from the BrightstarDB team, as well as priority bug-fixes and product enhancements. For more information please email NetworkedPlanet Limited.

CHAPTER 31

Indices and Tables

- *search*