

User's Guide to VelocityDB (Tuesday, September 14, 2021)

This guide compliments the [sample programs](#), [VelocityDB Quick Start](#), [VelocityGraph Quick Start](#) and the [API reference](#) provided on our site. Developers should review this in order to better understand how to a build a VelocityDB-integrated application.

Contents

Windows: Download and install a .NET development environment.....	5
Mac (and partly Linux): Download and install a .NET development environment.....	5
Open the VelocityDB repository in Visual Studio.....	6
Opening the samples solution, VelocityDB.sln	8
SampleData	9
Using VelocityDB and VelocityGraph NuGets	9
If not using our NuGets, manually add project reference to VelocityDB.dll	11
NuGet packages for solution.....	11
Add reference to System.Transactions	12
GitHub	13
Selecting the correct VelocityDB Session Class.....	13
Using database worker thread to speed up ingest/update of data.....	14
Concurrent access to database data.....	14
Optimistic locking versus Pessimistic locking	14
SessionPool class.....	15
Composite Object Identifier.....	15
DatabaseLocation.....	16
Moving/Copying Databases in a DatabaseLocation to a different Host/Directory	16
Page Compression.....	16
Encrypt Page data	17
Databases.....	17
Compacting Databases.....	18
VelocityDB license database	18
Replication	18
Storing Databases in the Cloud.....	18
Microsoft Azure.....	18
ServiceFabric Remoting.....	19
Accessing remote databases without using VelocityDBServer.....	19

- Pages 19
- Transactions 19
 - Try Catch blocks around all transactions 20
 - Why we need transaction for reads..... 20
 - Enabling recovery check for read transactions 20
 - Event subscription and notification 20
- How to enable persistent objects of some class..... 21
 - Implementing ISerializable..... 21
 - Collections using OptimizedPersistable.Equals and GetHashCode 22
- DateTime 22
- Database Schema 22
 - Register all types that you plan on persisting 23
 - If your application schema is using indexes..... 23
 - Fixed size class instances and limiting string size 24
 - Adding or removing field(s) from a class with existing objects in a database 25
 - Changing a field type without losing already persisted data 25
 - Renaming a persisted class or moving it to a different namespace 25
- VelocityGraph 26
 - Visualizing a VelocityGraph..... 26
- Persistent placement of objects 29
 - Best way to persist an object 29
 - Customizing object placement (most of you can skip this part)..... 30
 - Controlling placement of objects persisted by reachability 31
- Looking up objects 31
 - DO NOT reference persistent data using static variables 32
- Updating persistent objects..... 32
- Deleting (unpersisting) persistent objects 33
 - Referential integrity 33
- Collection Classes 34
 - List<T> vs VelocityDbList<T> 34
 - Avoid using Dictionary, HashSet and any other ISerializable classes 34
 - Using the provided BTree collections 35
 - BTreeMap<Key, Value> 36
- Indexes 36
 - Using a worker thread to add indexed objects to its indices..... 36

Class level index	36
Using a class level index	37
Index by a field	37
Using the index by field in a LINQ query	37
Changing indexing for a class after objects of that type already persisted	38
System.OutOfMemoryException	39
Limiting graph of objects in memory	40
Implementing your own classes with weak references	40
Using only weak references between objects	41
Lazy load of object references	41
Specifying depth to load at object open	41
Session caching of databases, pages and slots	41
Diagnostics	42
Handling exceptions thrown by VelocityDB.....	42
Database Manager	43
Starting Database Manager	44
Objects are initially lazy loaded	44
Browsing objects created by Baseball sample application	44
Validating Objects in your databases.....	46
Backing up (copy) all your database files.....	47
Database Schema Connections	48
Backup & Restore using Database Manager.....	49
Create Database.....	49
Create a backup Database Location.....	50
Create some persistent objects	52
Simulate loosing files in original DatabaseLocation.....	52
Restore these databases from backup DatabaseLocation.....	53
Restore a backup DatabaseLocation to a brand new directory.....	54
Using LINQPad to make VelocityDB LINQ queries/browsing.....	56
Issues with current LINQPad driver	65
Controlling the in memory page and object caching	65
Verifying all objects and references.....	65
Scalability	66
Database backup and restore	66
Backup.....	66

Restore	66
CopyAllDatabasesTo	66
ExportToCSV and ImportFromCsv.....	67
VelocityDbServer.exe.....	67
Changing the default SessionBase. BaseDatabasePath in a VelocityDbServer.....	68
Option to log all activity in VelocityDBServer	68
Changing the tcp/ip port number used when communication with a VelocityDBServer	68
Enabling Windows Authentication	68
VelocityDBCOREServer with http REST Api.....	69
Active connections to VelocityDBCOREServer	70
Viewing object.....	71
Updating object.....	71
Adding Object.....	74
Seetings for the VelocityDBCOREServer.....	75
Chrome Json Formatter	78
Why installation ends up in Program Files (x86) instead of Program Files?.....	78
.NET CORE	78
.NET 5 and .NET Standard 2.0	79
Universal Windows	79
Where to store databases with Universal Windows?.....	79
iOS	80
Android	80
Asp.Net Identity	80
Application Deployment and VelocityDB license check.....	80
Setting Up the sample Web Site (VelocityWeb) on a hosting web site (in this case GoDaddy).....	80
Transfer all the files to your hosting account	80
Login to your hosting provider to enable write access to a few of the directories in the application.....	81
Create an application root virtual directory for the new web application.....	83
Wait a few minutes then point your browser at your web application	84
If you transferred your application directory with databases then install your databases in their new loacftion.	85
If all is well, you are done, access the application and the databases!	85

Windows: Download and install a .NET development environment

If you don't already have, you need to download and install software that lets you edit, compile and debug .NET code. Some choices exist but for Windows development we recommend [Visual Studio Community 2019](#)(free) with all updates applied. The Professional or Enterprise versions can be even better, but they will cost you.

Mac (and partly Linux): Download and install a .NET development environment

If you don't already have, you need to download and install software that lets you edit, compile and debug .NET code. Some choices exist but for Mac development we recommend [Visual Studio](#).

Install a Git client, we like: <https://www.sourcetreeapp.com/>

Go to [GitHub](#).

In a Terminal window do:

Go to directory where you want VelocityDB code, for instance:

```
Last login: Thu Sep 13 15:07:20 on ttys006
```

```
Matss-Mac-mini:~ matspersson$ git clone https://github.com/VelocityDB/VelocityDB
```

```
Cloning into 'VelocityDB'...
```

```
remote: Counting objects: 15105, done.
```

```
remote: Compressing objects: 100% (276/276), done.
```

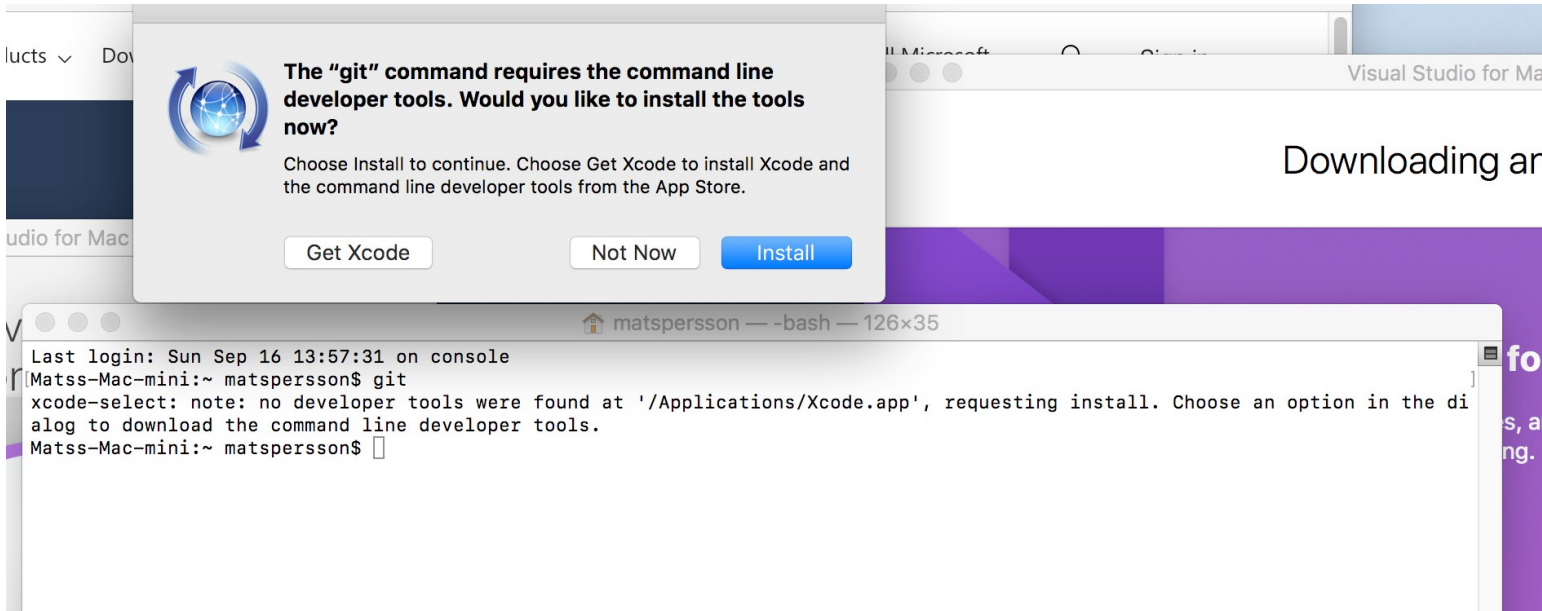
```
remote: Total 15105 (delta 418), reused 510 (delta 359), pack-reused 14462
```

```
Receiving objects: 100% (15105/15105), 4.95 MiB | 8.59 MiB/s, done.
```

```
Resolving deltas: 100% (13004/13004), done.
```

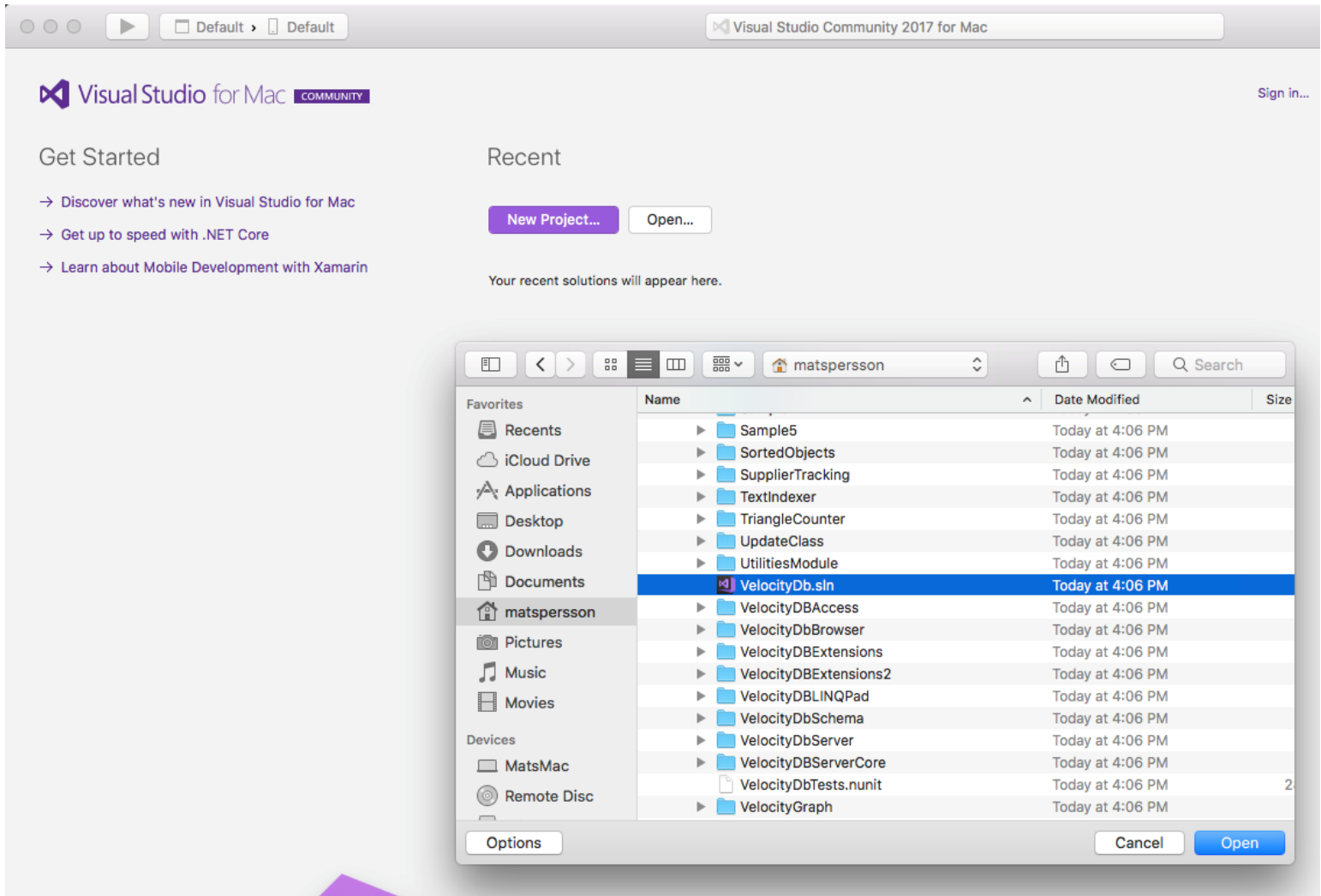
```
Matss-Mac-mini:~ matspersson$
```

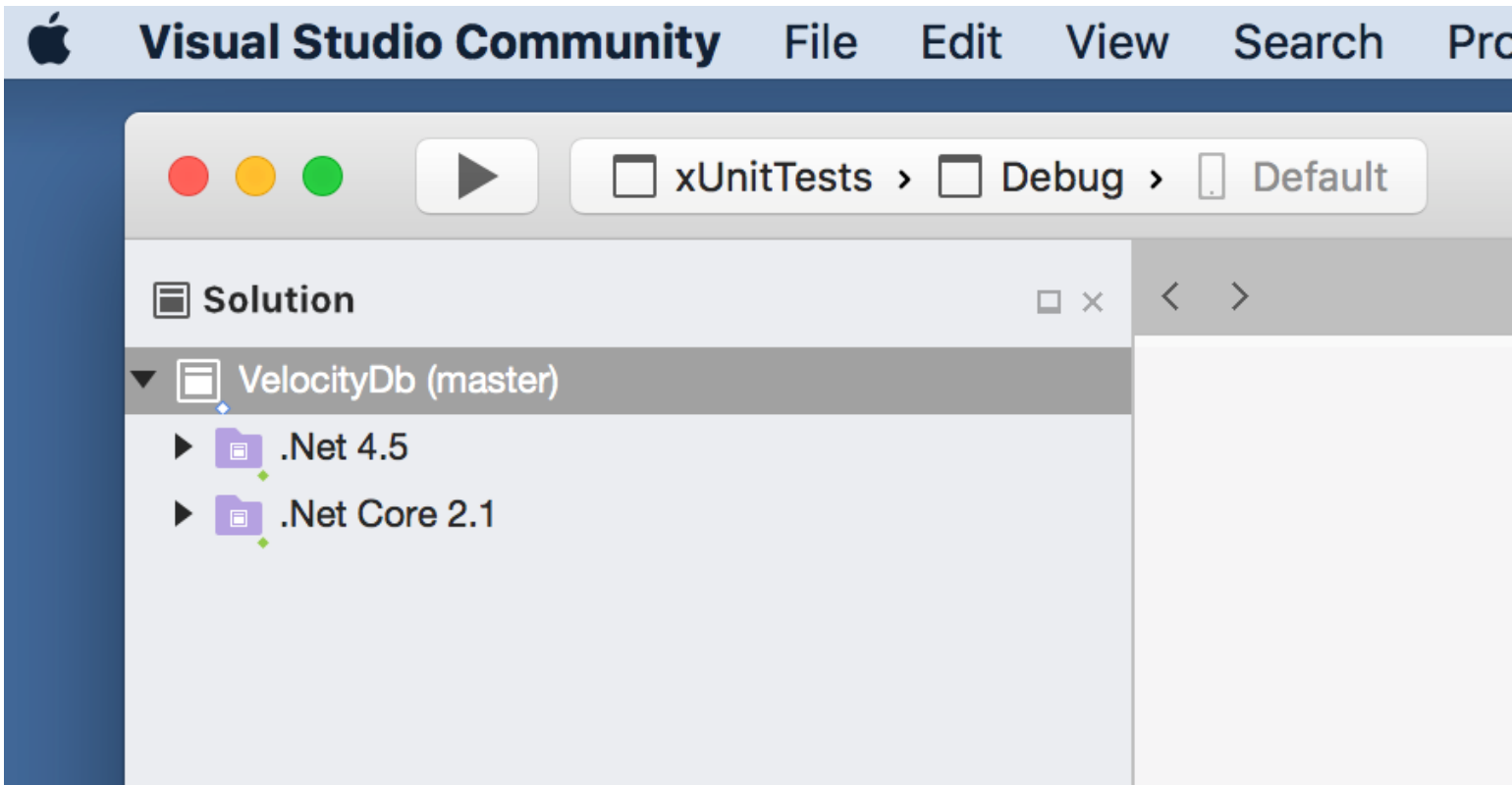
If git isn't already installed, you get



In such a case, choose to **Install** it, then try this again in a fresh terminal window.

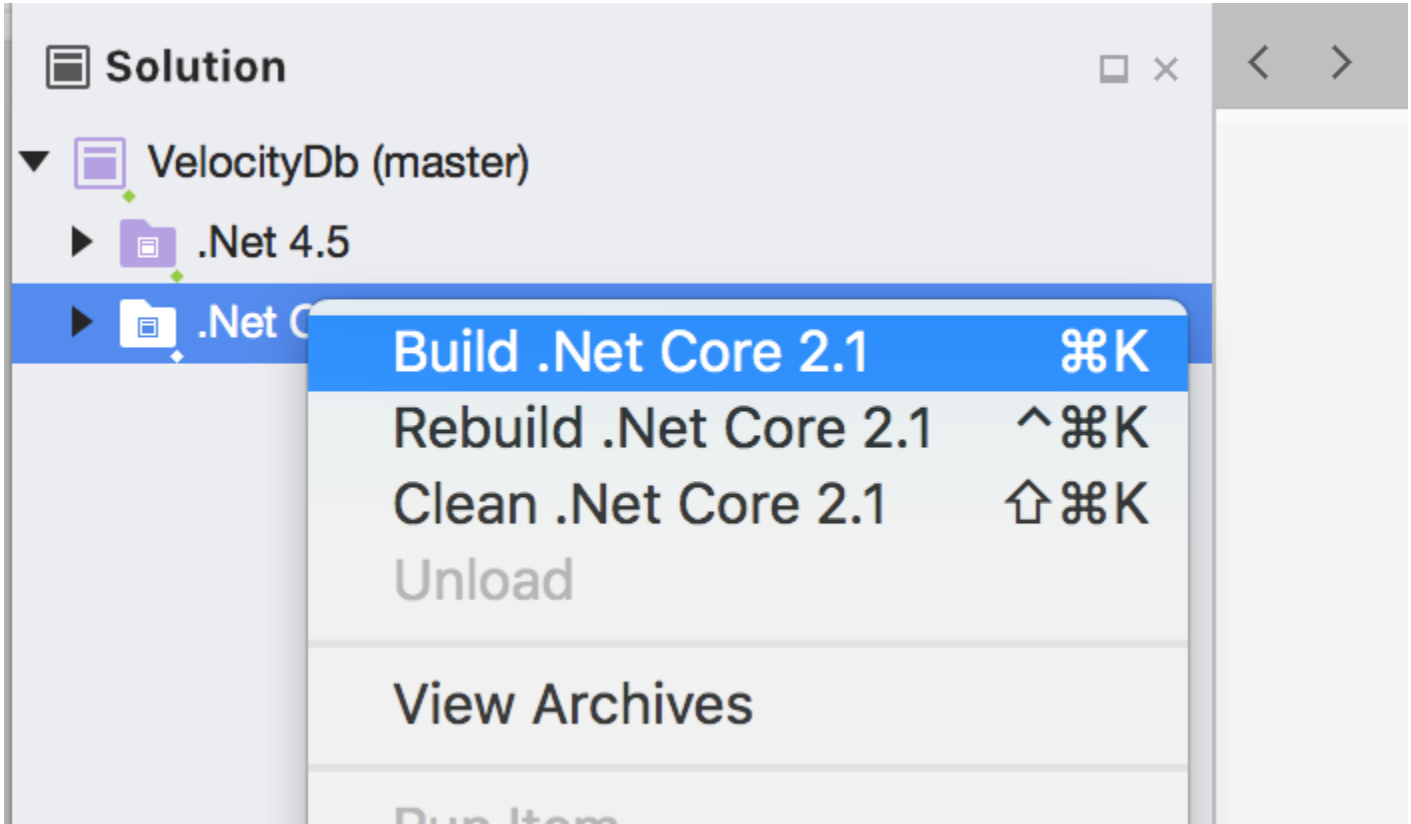
Open the VelocityDB repository in Visual Studio





This solution is shared with Windows. For Mac you can only run the projects in the .Net Core 3.1 folder.

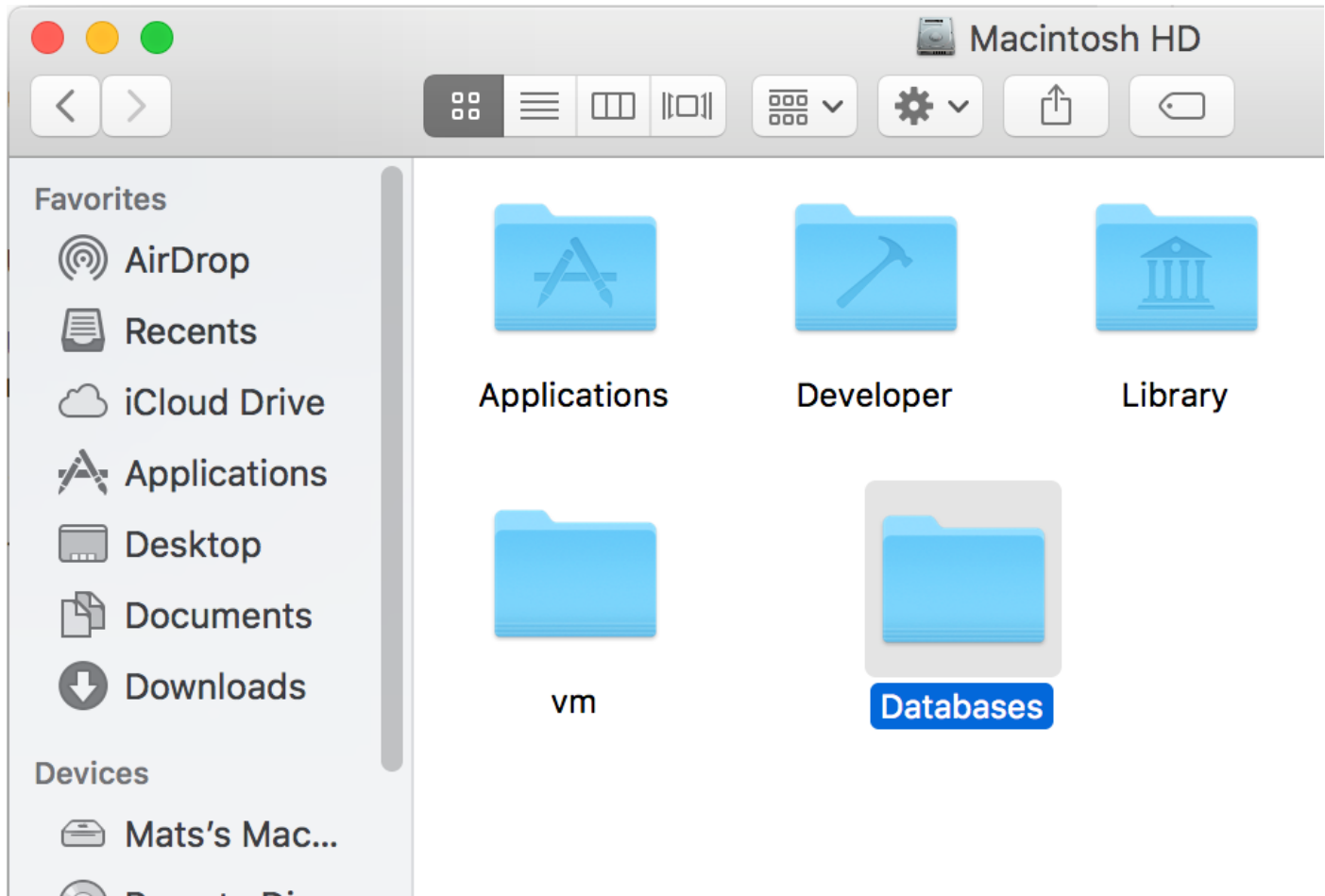
Select that folder and right click (*image needs to be updated to .NET Core 3.1*)



Build projects in this folder.

Currently, you will probably see one build error. We are still trying to figure out how to fix that one.

By default, databases will be stored in a sub folder to /Databases so create this folder before running any sample project.

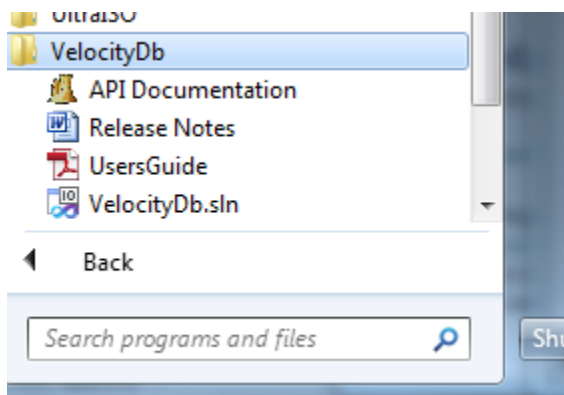


Some of these samples and Database Manager are WPF applications and these can currently not run with .NET Core. They will run with Windows and .NET Core 3 but not on a Mac.

Opening the samples solution, VelocityDB.sln

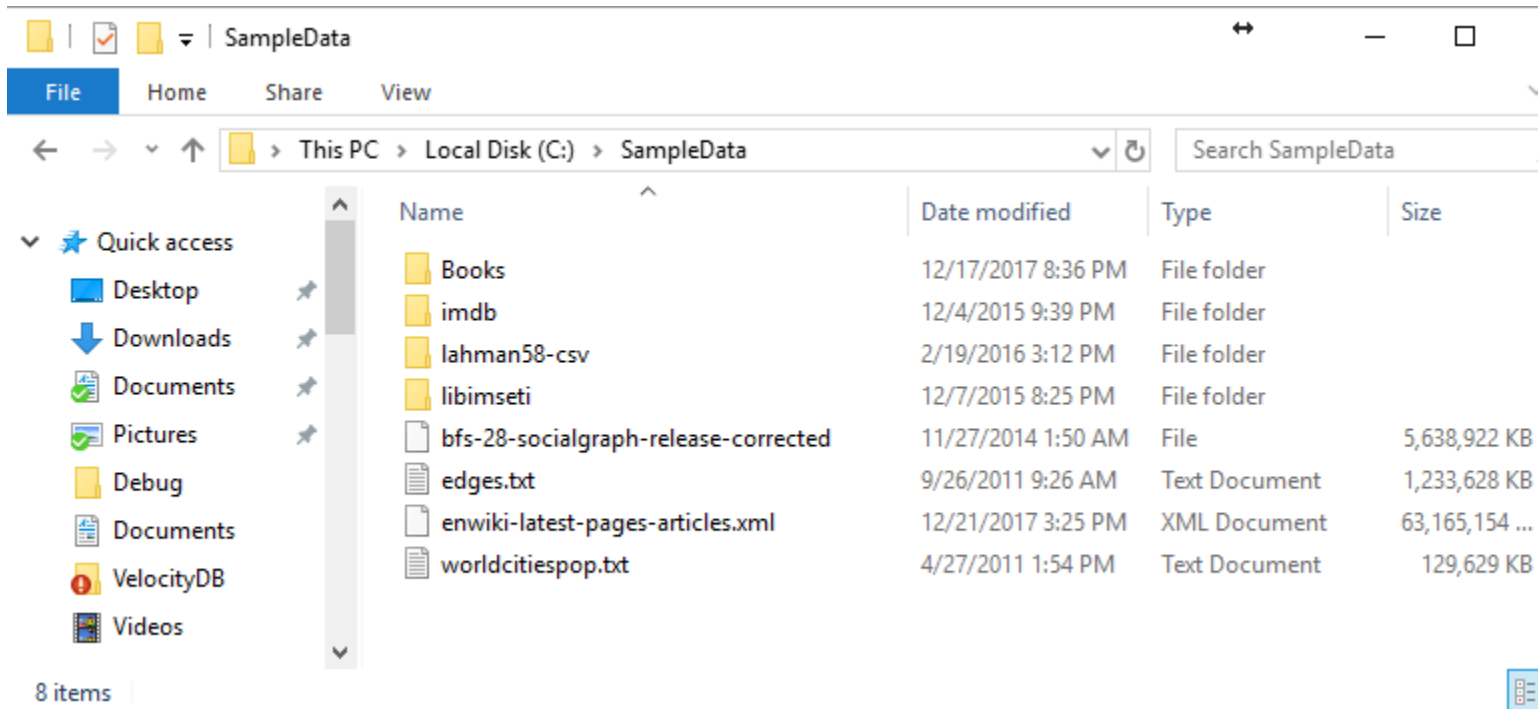
Open %USERPROFILE%\My Documents\VelocityDB\VelocityDB.sln

You can also start it by using the shortcut in the programs start menu.



SampleData

Many of the sample projects use data files. We expect these files to be in folder c:\SampleData



Download a zip file with this SampleData folder [here](#).

If your C drive isn't the best location for these large files then create the sample folder on another drive and create a link from C:\SampleData to this location using the mklink command.

Open a Cmd window and type

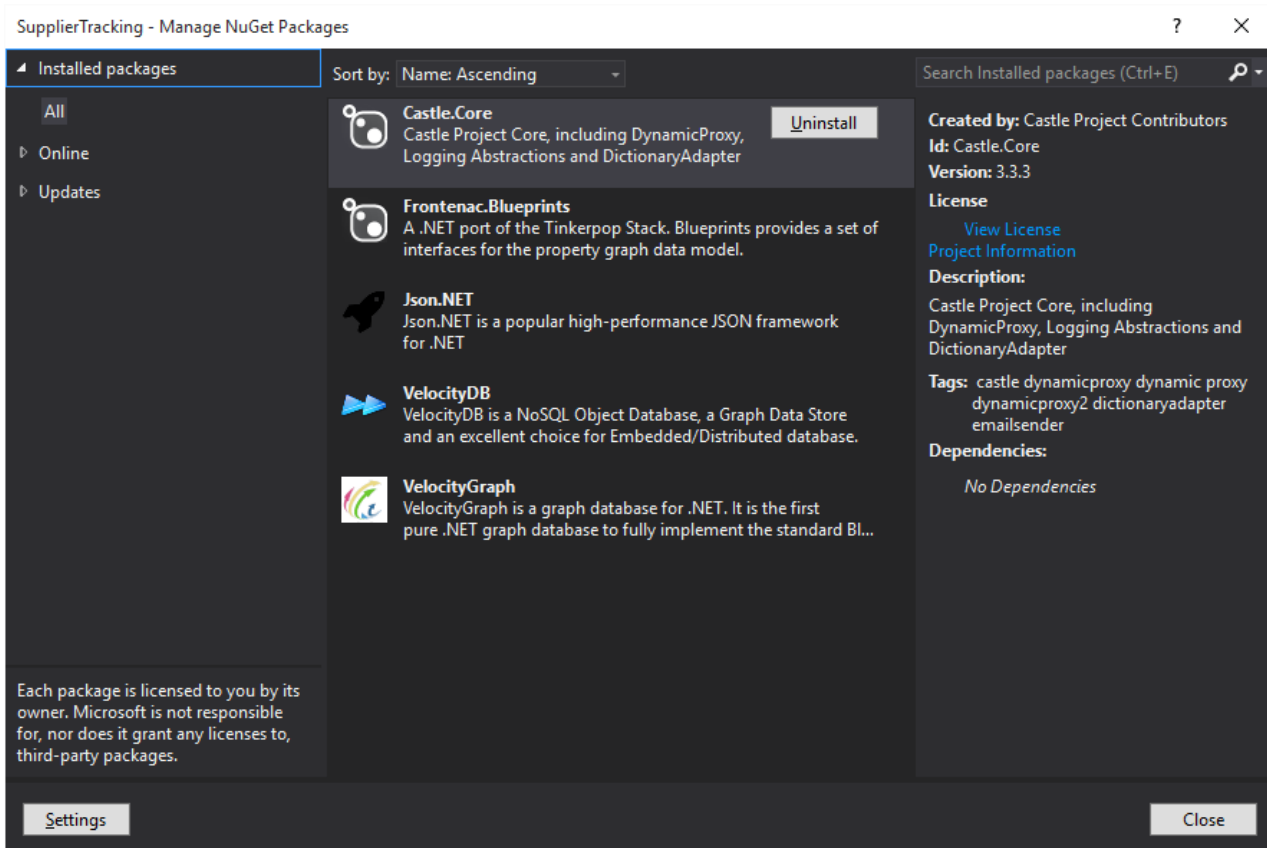
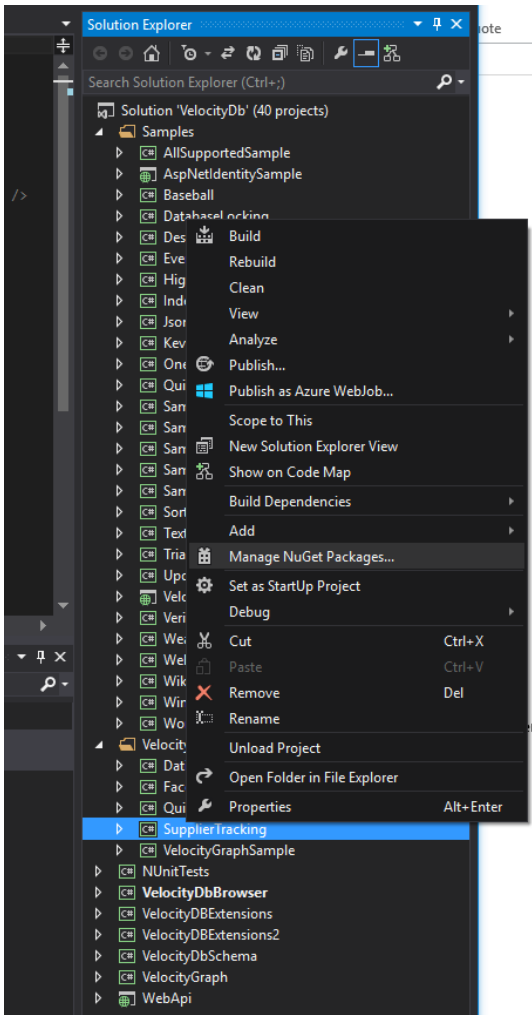
```
[C:\>mklink /D SampleData F:\SampleData
```

Symbolic link created for SampleData <====> F:\SampleData

You may also want to do the same for C:\Databases

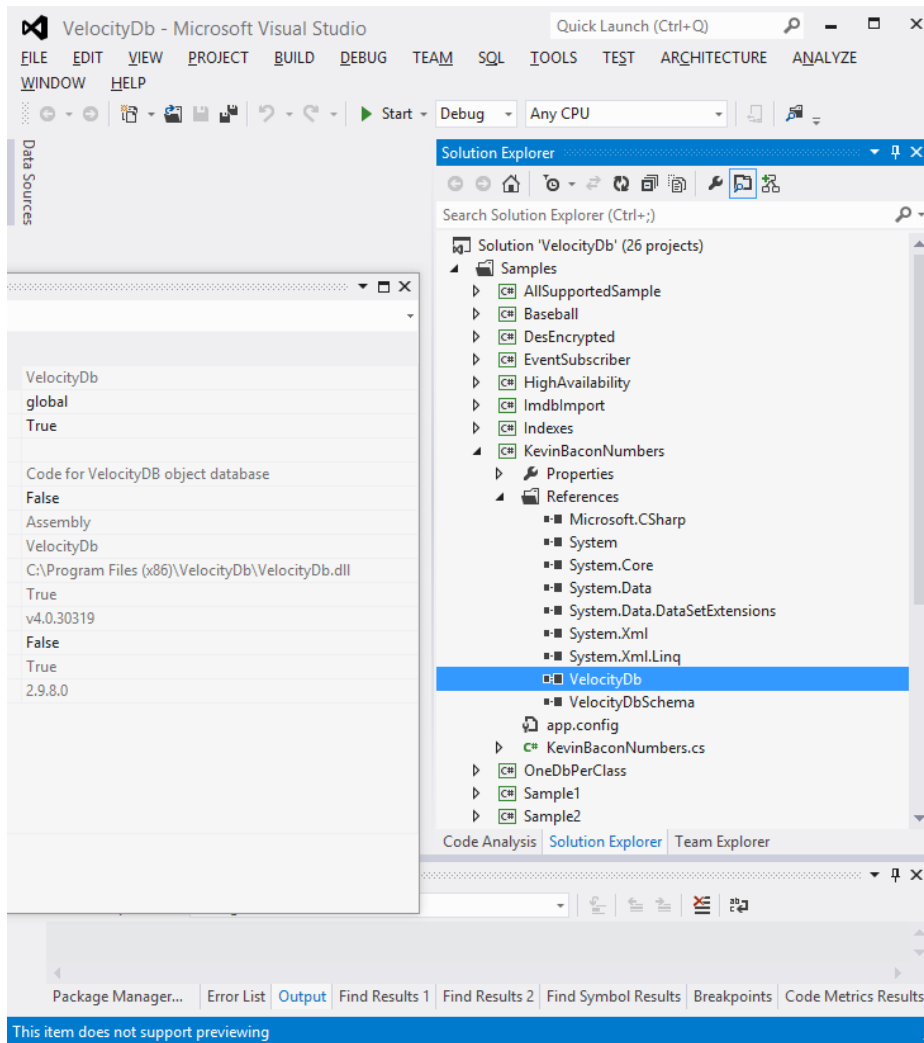
Using VelocityDB and VelocityGraph NuGets

This is the recommended way to add a reference to our DLLs. Right click on a project, like SupplierTracking, and select "Manage NuGet Packages..."



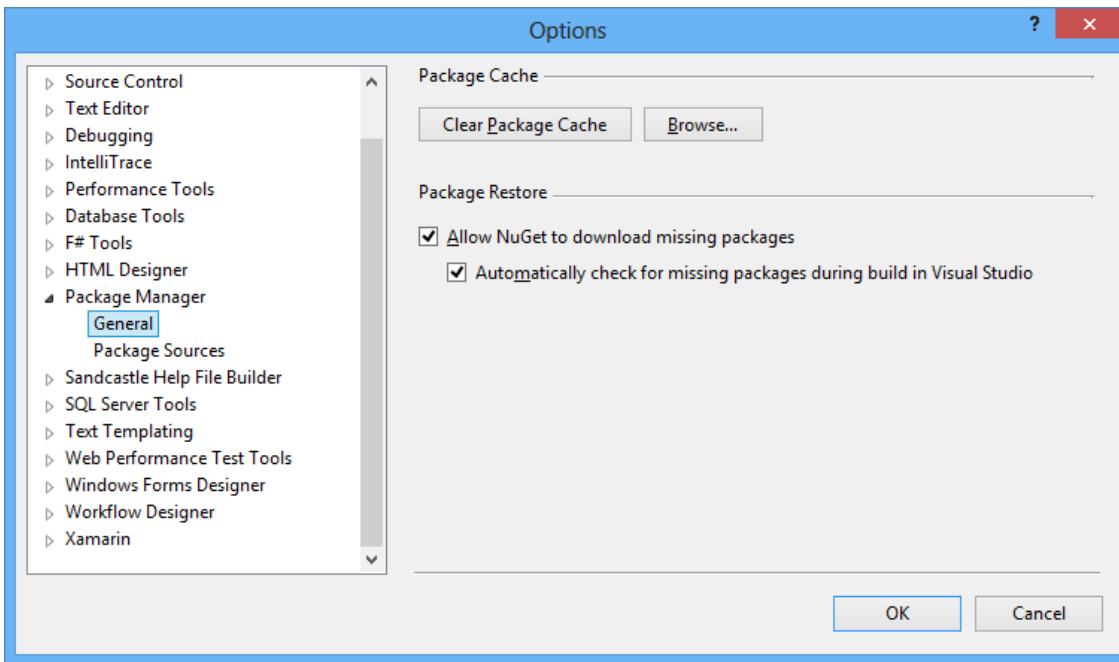
If not using our NuGets, manually add project reference to VelocityDB.dll

All sample projects should have a reference to VelocityDB.dll. The path used to VelocityDB.dll is C:\Program Files (x86)\VelocityDb\VelocityDB.dll, if you windows directory isn't C: or the reference is broken then you need to remove each project reference to VelocityDB.dll and add a new one using the path to it in your installation.



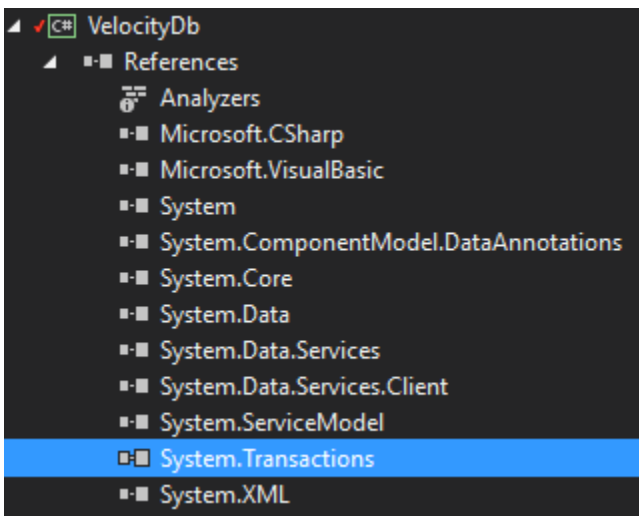
NuGet packages for solution

A few of the samples including VelocityGraph project uses 3rd party NuGet libraries. These libraries are not part of the installation but will be downloaded automatically when you attempt to build such a project. To make this happen you need to allow NuGet to download missing packages. If it still does not download (firewall blocking?) then you may need to manually install the missing NuGets.



Add reference to System.Transactions

VelocityDB now supports this type of distributed transactions. A transaction can now be shared between SQL Server and VelocityDB (and other resources/database systems). To make this work, any client using VelocityDB sessions must add a reference to [System.Transactions](#) (except for NET.CORE and UniversalWindows which do not support System.Transactions).



Sample code using this type of distributed transaction is included in NUnit Tests. Here is part of it.

```
[Test]
public void GermanString()
{
    UInt64 id = 0;
    using (SessionNoServer session = new SessionNoServer(s_systemDir))
    {
        using (var trans = new TransactionScope())
        {
            session.BeginUpdate();
            VelocityDbSchema.Person person = new VelocityDbSchema.Person();
            person.LastName = "Med vänliga hälsningar";
        }
    }
}
```

```

    id = session.Persist(person);
    trans.Complete();
}
}

```

GitHub

If you prefer not to use our installer and instead want to build our extensions, drivers, server and samples from the source code as in our [GitHub repository](#) then you need to manually first install

1. [Microsoft Sync Framework](#) (used by our extensions project VelocityDBExtensions2)
2. [LinqPad5](#) (used by our LinqPad driver)

Clone the repository: <https://github.com/VelocityDB/VelocityDB.git>

Selecting the correct VelocityDB Session Class

The most important class for users of VelocityDB is the *Session* class which contains the **Transaction Control API**, the **Persistence API**, the **Data Cache API** and more. VelocityDB provides three session types and does not limit usage. Your application can utilize all of them as necessary:

- **ServerClientSession** - Used for distributed databases or when clients are hosted remotely.

```

// initial DatabaseLocation directory and hostname
using (ServerClientSession session = new ServerClientSession("c:\\Databases", "DbServer"))
{
    session.BeginRead();
    // your code here
    session.Commit();
}

```

- **SessionNoServer** - Client and data are on the same host (unless it is a web application)

```

using (SessionNoServer session = new SessionNoServer("c:\\Databases"))
{
    session.BeginRead();
    // your code here
    session.Commit();
}

```

SessionNoServerShared - Client and data are on the same host (unless it is a web application) with use of pages and databases thread safe (other objects only partially) and by default VelocityDB adds some threading. One thread handles all index updates and another thread handles object encoding and page writes. You can optionally turn of the page write thread by a property setting in the session.

`session.WriteToDiskInSeperateDatabaseThreads = false`, the index update thread can also be disabled (but must be enabled if page write threads are) by setting `session.AddToIndexInSeperateThread = false`;

Having these worker threads active can dramatically improve update performance BUT at this time it may not due to the overhead of the Monitor locks. However, more could be parallelized, but it requires complicated object level thread locks. Eventually, we will probably merge in the worker thread functionality into SessionNoServer and eliminate SessionNoServerShared.

```

using (SessionNoServerShared session = new SessionNoServerShared ("c:\\Databases"))
{
    session.BeginRead();
    // your code here
    session.Commit();
}

```

The session class [ServerClientSession](#) is appropriate if the application will distribute data and/or clients across multiple hosts (where the clients are not just clients of a web site). Otherwise, [SessionNoServer](#) or [SessionNoServerShared](#) are appropriate. Of the two, the best choice is dependent upon the architecture of the application.

Additional benefits of using [ServerClientSession](#)

- ✓ Granularity of locking is page instead of database (file).
- ✓ Backup feature option
- ✓ Shared cache for all users (on server side)
- ✓ [Deadlock](#) detection (when pessimistic locking is used, with optimistic locking deadlocks don't happen)
- ✓ Change event subscription and notification

Benefits of using [SessionNoServer](#) or [SessionNoServerShared](#)

- ✓ No server installation required
- ✓ More stable, less can go wrong
- ✓ Can perform better with local files.

[Our video](#) talking about database concurrency control may help you decide what session to use.

Use [SessionNoServerShared](#) when the application must share a client-side cache between multiple threads. This may be the case for a web site that has limited RAM resources while also having a large amount of persistent data to manage.

It is recommended that a session is reused for multiple transactions since that will provide some caching benefits and also avoids some setup time, especially with [ServerClientSession](#).

DO NOT pass objects between session instances. Once you read an object from a database, that object belongs to the session used to read it. Do not attempt to read an object using one session and the update it using another session. This will not work as expected and we may not detect it so it will fail silently.

Using database worker thread to speed up ingest/update of data

By default, starting in VelocityDB 4.6, each database will have a worker thread responsible for taking updated objects and writing these to disk. This is currently only available when using [SessionNoServerShared](#) session class. An application can turn this threading off by setting the session property

`session.WriteToDiskInSeperateDatabaseThreads = false;` For now, ~~we recommend using [SessionNoServer](#) over [SessionNoServerShared](#) due a few remaining issues in [SessionNoServerShared](#) that may end up as exceptions being thrown.~~

Concurrent access to database data

[SessionNoServer](#) and [ServerClientSession](#) are not thread safe so don't use these with multi-threaded code. Be careful not to declare database access code async as it introduces possible multi-threading. [SessionNoServerShared](#) is thread safe but only at object, page and database access level. Complex objects such as [BTreeSet](#) may still not be fully thread safe with update transactions. We recommend using a single [SessionNoServerShared](#) for all read only access and a [SessionPool](#) session for update transactions. See [Issues.aspx.cs](#) as an example of how to use it.

Optimistic locking versus Pessimistic locking

By default VelocityDB uses optimistic locking. Pessimistic locking can be turned on by a session constructor parameter. With optimistic locking, reads are always possible except for uncommitted new databases and multiple updaters are

allowed but only the first writer will succeed, the other writers of the same page ([ServerClientSession](#)) or database ([SessionNoServer](#)) will get an optimistic locking exception. Once you decide using optimistic/pessimistic concurrency control, stick with your choice. **Do not mix** sessions using optimistic concurrency control with sessions using pessimistic concurrency control. If your application often try to update the same database/page concurrently, you are better off using **pessimistic** locking as it will wait for a lock to be released and then proceed to do the updates successfully in each concurrent transaction unless a [deadlock](#) is detected.

SessionPool class

Use this class when you have frequent database requests coming in from multiple clients possibly simultaneously, i.e. a web application serving multiple clients. With [SessionPool](#), you will reuse a set of sessions. With reuse comes a cache of databases, pages and objects. Having the cached data makes access to data faster compared to starting with a brand new fresh session each time. Keep the number of sessions allocated for the pool small to reduce memory usage, we think 3 sessions should be enough in most cases. If more than the set maximum sessions are requested from [SessionPool](#) then a temporary new session will created and then disposed after usage so that memory usage is reduced. It is important that your code frees the session back into the pool after each usage.

```
const int numberOfSessions = 5;
SessionPool pool = new SessionPool(numberOfSessions, () => new SessionNoServer(systemDir));
int sessionId = -1;
SessionBase session = null;
try
{
    session = pool.GetSession(out sessionId);
    session.BeginUpdate();
    for (int i = 0; i < 1000; i++)
    {
        Man man = new Man();
        session.Persist(man);
    }
    session.Commit();
}
catch (Exception e)
{
    if (session != null)
        session.Abort();
    Console.WriteLine(e.Message);
    throw e;
}
finally
{
    pool.FreeSession(sessionId, session);
}
```

Composite Object Identifier

All normal VelocityDB persistent objects have an associated composite object identifier. It is encoded as a [UInt64](#) with three composite parts; a database number (upper 32 bits), a page number and a slot number. The [Id](#) property returns an objects encoded object identifier and the [Oid](#) property returns the decoded object identifier as the struct [Oid](#). A reference to a persistent object is persistently stored as an object identifier, it is normally a [UInt64](#) but it can also be using a short object identifier, a [UInt32](#), when the reference is to another object within the same database. The decoded short reference as a struct is [OidShort](#). Use the special [OidShort](#) collection classes and tag object references with the attribute [\[UseOidShort\]](#) as in:

```
[Serializable]
[UseOidShort]
```

```
internal class Recovery : OptimizedPersistable
```

and for a specific member:

```
[UseOidShort]  
public VelocityDbListOidShort<FreeSpace> theArray;
```

DatabaseLocation

This is a directory on some host. The initial `DatabaseLocation` is created when you create your first persistent object. You specify the directory when you create the session class. You can create additional database locations like:

```
using (ServerClientSession session = new ServerClientSession(systemDir, Dns.GetHostName()))  
{  
    session.BeginUpdate();  
    DatabaseLocation otherLocation = new DatabaseLocation(Dns.GetHostName(), location2Dir,  
locationStartDbNum, locationEndDbNum, session, true, 0);  
    otherLocation = session.NewLocation(otherLocation);  
    session.Commit();  
}
```

You need to commit the initial `DatabaseLocation` before other sessions (clients) can access it.

Moving/Copying Databases in a DatabaseLocation to a different Host/Directory

If you only have a single directory for your set of connected databases, you may wonder why we need to update anything. The reason is that some usage scenarios may use one hundred or more database locations in a single set of databases. That is why we maintain a catalog of database locations in 2.odb.

First move your database files to desired host and directory, then do like:

```
using (var session = new SessionNoServer("CompanyBootLocation"))  
{ // NO longer required starting in version 10.1, we do this automatically when we detect a new default/bootup  
location in an update transaction  
    session.RelocateDefaultDatabaseLocation(); // update default database location without first starting  
a transaction  
}  
  
// other locations you will have to update yourself  
using (var session = new SessionNoServer("CompanyBootLocation"))  
{  
    session.BeginUpdate();  
    session.RelocateDatabaseLocationFor(session.DatabaseNumberOf(typeof(InsuranceCompany)),  
SessionBase.LocalHost, "InsuranceCompanies");  
    session.Commit();  
}
```

Page Compression

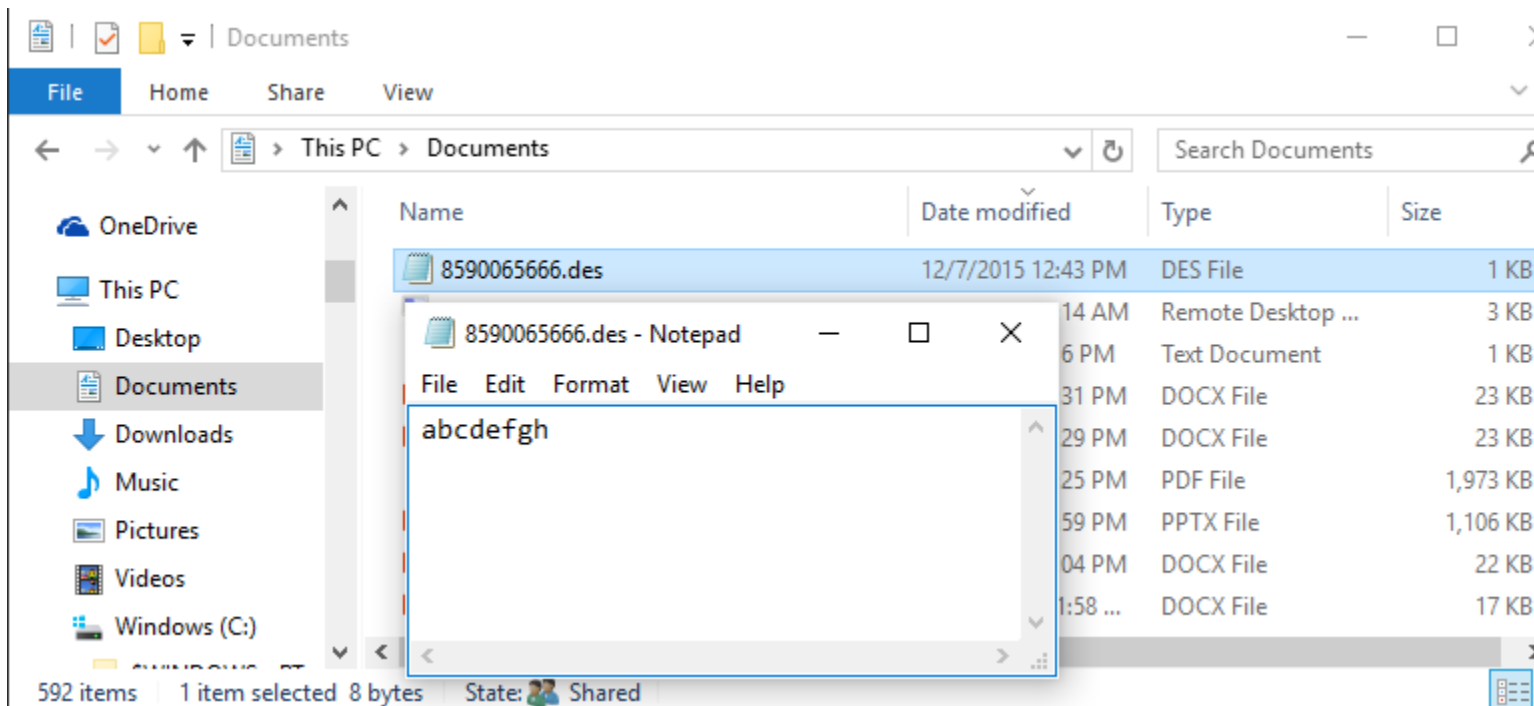
Page compression is now by default turned off. You can turn it on by setting the constructor parameter when you create a `DatabaseLocation`.

The initial/default `DatabaseLocation` is created when you run your first update transaction with a specified directory that does not already contain databases 0, 1, and 2 (0.odb, 1.odb, and 2.odb).

If you want page compression turned on for this `DatabaseLocation`, set `SessionBase.DefaultCompressPages` to true first. This static variable is also used when you create your own `DatabaseLocation` and not specifying the `compressPages` constructor parameter.

Encrypt Page data

By default page data is not encrypted. You can enable Des encryption when you create a `DatabaseLocation`. Our sample application [DesEncrypted](#) shows how to do it. You can also use [DatabaseManager](#) to make it happen. Des encryption requires an 8 character (8 bytes) key. Once you start using Des encryption, this key is stored in a file within the active Users Documents folder. Filename is based on Id of the `DatabaseLocation`.



This file needs to be copied to all Users Documents folder for access to such encrypted pages. **DO NOT** change the key after you have persisted pages with Des encryption.

We can provide other encryption mechanisms on request and we can also rework API such that custom encryption methods can be used.

Databases

A database corresponds to a file within a `DatabaseLocation`. The file name of a `Database` is `<database number>.odb`. When you create your first persistent data, three system databases are created:

- **0.odb**
Contains a log of update transactions and the recovery mechanism data.
- **1.odb**
Contains the schema objects
- **2.odb**
Contains the `DatabaseLocation` objects.

These system databases must be committed by a session before other sessions can use them. This is true for any new database; a database must be committed before other sessions can access it.

A new uncommitted `Database` is named `<database number>.new` and an uncommitted deleted `Database` is named `<database number>.de1`.

A **Database** can be created explicitly using session API or implicitly by placing a new persistent object with database part of the object identifier corresponding to an unallocated database number.

Compacting Databases

Database pages uses versioning so that a page can be updated in one transaction and prior committed state of that page can be read by other transactions. After updating pages there may be available space within a Database file. This is because when a Database page is updated, it is not written back to the same location in the file, a new version of the page is created somewhere else in the file. A new version of a page may be smaller/larger than the prior version. Space for a new page version is allocated from a best fit free area. If no free area large enough is available, then the database file is extended and the page is allocated at the end of the file. This versioning can create up to two versions of each page in a database. By calling `SessionBase.Compact()` this extra database space is reclaimed and pages are physically ordered in page number order. This may improve performance. `ServerClientSession` currently does not implement `Compact()` so for now use `SessionNoServer` when calling `Compact()`. Call `Compact()` outside the scope of any transaction.

Notice! Backup your database files before attempting a `Compact()` as it is a potentially very large update to your database structures. Avoid attempting to make other updates to the databases while running `Compact()`.

VelocityDB license database

Download your VelocityDB license database file from <http://www.velocitydb.com/Secure/License.aspx>

The license database file is named **4.odb**. Copy this file to all database directories used for the system databases 1.odb ... 9.odb. This is the directory you specify when creating the session instance. Some of the sample applications provided with the download will fail without a license database. If a license database is missing when a license check is happening, VelocityDB will copy a license database from your "Downloads" directory if such a file exists and use it for the license check. A license check does not require an active internet connection. VelocityDB never tries to talk to any other host as part of the license check.

Replication

Databases in the initial `DatabaseLocation` that starts with database id 0 can be replicated to multiple directories on multiple hosts. This enables high availability, if one replica isn't available then another available one is used if available. Under normal operation, all changes are applied to all replicas. If a replica is found to be out of sync, it is refreshed from one of the replicas that is up to date. Using replication is optional and is activated by using a special version of the `ServerClientSession` constructor as in:

```
alternateSystemBoot = new List<ReplicaInfo> { new ReplicaInfo { Path = "Replica1" }, new ReplicaInfo { Path = "Replica2" }, new ReplicaInfo { Path = "Replica3", Host = s_systemHost2 } };  
using (var session = new ServerClientSession(alternateSystemBoot))
```

Replicas can be added/removed by changing the `List<ReplicaInfo>` constructor parameter.

As of February 2019, this is a new feature with some limitations, we will incrementally update the replication code with automatic fault tolerance until it's perfected, handling all cases and is rock solid. Please help us with ideas and test cases for how to get there.

Storing Databases in the Cloud

Microsoft Azure

It is very easy to store databases in the cloud with replication, backup and safe access using Microsoft Azure File storage. Microsoft provide free trials of Azure. To store databases on Azure servers, all you need to do is to use a file share.

See description [here](#).

Once you have mounted your Azure directory as a local drive such as z:, you can start using it for reading and updating Azure hosted storage. We also started work on an AzureSession class based on SessionNoServer as a direct way to access Azure hosted databases. The code for this is in our download as part of VelocityDbExtensions project file name AzureSession.cs. It currently isn't fully working due to challenges with Azure Stream that only can be read only or update only, required explicit Flush() and file resizing. In any case the shared drive solution is more transparent and have less restriction so use it for now.

```
Example: net use z: \\samples.file.core.windows.net\logs /u:samples<storage-account-key>
```

ServiceFabric Remoting

Microsoft now supports a micro service technology they named "[ServiceFabric](#)". It is a very cool option that lets you deploy services on your local computer, an own server or in the cloud using Microsoft Azure. It lets you communicate in many ways between client and server but the coolest/easiest way is by using [ServiceFabric Remoting](#). All you do is define an interface and then you implement the interface in the service fabric service. Clients just instantiate the interface by a proxy (one line statement) and then the service becomes available as if it was API within the client process. Very nice! On the server (service) side you do not need to use the VelocityDBServer, you can use the embedded client sessions instead (SessionNoServer and/or SessionNoServerShared).

Accessing remote databases without using VelocityDBServer

If remote server is within a Windows network, [UNC path](#) to databases can be used. Easiest way to do it is by setting SessionBase.BaseDatabasePath, i.e.

```
static readonly string s_systemDir = "UncPath"; // appended to SessionBase.BaseDatabasePath
static int Main(string[] args)
{
    SessionBase.BaseDatabasePath = @"\\FindPriceBuy\BenchmarkDatabases";
    for (int i = 0; i < 5; i++)
        using (var session = new SessionNoServer(s_systemDir))
        {
            Console.WriteLine($"Running with databases in directory: {session.SystemDirectory}");
        }
}
```

Pages

A VelocityDB page can contain one or more persistent objects. The size of a [Page](#) can vary dynamically. A page is stored within a Database file. Each [Page](#) has a [PageInfo](#) header that contains information about a page. A Page can optionally be encrypted and/or compressed.

Transactions

All interaction with databases and persistent objects require an active transaction. With VelocityDB we provide two kinds of transactions; update and read only. With an update transaction, you are permitted to update and add persistent data. With a read only transaction, an exception will be thrown by VelocityDB if you try to update persistent data. Only one concurrent transaction per session is permitted. A transaction is started and committed by API on the session classes. An application may examine in memory persistent objects without being in a transaction but an exception will be thrown if any persistent operation is requested like reading a page from a database.

```
public virtual void BeginRead(bool doRecoveryCheck = true)
public virtual void BeginUpdate()
public virtual void Commit(bool doRecoveryCheck = true)
public virtual void Abort()
```

Try Catch blocks around all transactions

It is particularly important to add this around update transactions. If you don't add it around an update transaction then you could end up corrupting your data. You should always abort the active transaction if you get an exception.

```
static int Main(string[] args)
{
    using (SessionNoServer session = new SessionNoServer(systemDir))
    {
        Console.WriteLine("Running with databases in directory: " + session.SystemDirectory);
        try
        {
            session.BeginUpdate();
            Company company = new Company();
            company.Name = "MyCompany";
            session.Persist(company);
            Employee employee1 = new Employee();
            employee1.Employer = company;
            employee1.FirstName = "John";
            employee1.LastName = "Walter";
            session.Persist(employee1);
            session.Commit();
        }
        catch (Exception ex)
        {
            Trace.WriteLine(ex.Message);
            session.Abort();
        }
    }
    Retrieve();
    return 0;
}
```

Why we need transaction for reads

With optimistic locking option (the default) there is no locking reason for a transaction when only reading objects. If the other locking model is used, pessimistic locking, then read only transactions are needed because they define the scope of read locks. A session constructor parameter is used for requesting optimistic or pessimistic locking model. Another reason we need read only transaction is cache management and validation. Each Database, Page and Object is cached within a session instance. Each cached Database is validated in the beginning of a transaction, making sure cached version is up to date. If reads are frequent among multiple threads, it may make sense to use a shared session for the reads, [SessionNoServerShared](#), and maintain an infinitely long open optimistic locking read transaction. Call `ForceDatabaseCacheValidation()` frequently when there is possible other active database clients so that your cache stays up to date. Alternatively trigger validation of only selected databases by setting the Database property `CachedVerified` to `false`.

Enabling recovery check for read transactions

By default when a `BeginRead()` transaction is started, we do not check for the very unlikely event that our previous update transaction was not completely persisted so that we need to revert to prior state. By skipping this check in read transactions we save time. To enable the check start transaction with `BeginRead(true)` instead.

Event subscription and notification

With use of `ServerClientSession` you can subscribe to object add/modification events. The [EventSubscriber](#) sample, part of your `VelocityDb.sln`, in our download shows how it can be used.

A session can subscribe to changes made in other `ServerClientSession` sessions in any process on any host.

An event subscription is started like

```
session.SubscribeToChanges(typeof(Person));
```

subscribes to any updates involving Person objects.

```
session.SubscribeToChanges(typeof(Woman), "OlderThan50");
```

subscribes to any updates involving Woman objects where property OlderThan50 evaluates to true.

Events are received at the start of a transaction by using special begin transaction API

```
List<Oid> changes = session.BeginReadWithEvents();
```

or

```
List<Oid> changes = session.BeginUpdateWithEvents();
```

How to enable persistent objects of some class

There are three major choices for enabling persistence.

1. Make your data model class a subclass of [OptimizedPersistable](#)
2. Implement the interface [IOptimizedPersistable](#). See the sample class [PersistenceByInterfaceSnake](#) as a template for how to implement the required interface API.
3. Implement the interface [ISerializable](#)

These three ways of enabling persistence can be mixed, some classes may implement the interface and others may be subclasses of [OptimizedPersistable](#).

Objects of [ValueType](#) and arrays are embedded within a parent persistent object.

In addition, almost any type of object, except Delegate and Pointer instances, can be made persistent but this way is not very efficient due to requiring use of a fairly inefficient [ConditionalWeakTable](#) internally by VelocityDB due to such objects not maintaining an object identifier as a field.

[OptimizedPersistable](#) implements [IOptimizedPersistable](#).

Implementing ISerializable

This way is NOT recommended as it slows down serialization and deserialization. This is also true for [ISerializable](#) classes that you may use from some library. Be prepared, it will be slow. HashSet is about 60x slower to deserialize vs List/[BTreeSet](#) due to it being ISerializable.

In some cases regular serialization/deserialization is not desired. Good examples of that are the date classes in [NodaTime](#). These object de-serialize to use a shared [CalendarSystem](#) instance. (Very clever!)

If your class implements both ISerializable and IDeserializationCallback then VelocityDB will call your callback function OnDeserialization.

Sample simple use of ISerializable (part of NUnit tests included with our product download & on GitHub)

```
public class TestISerializable : ISerializable
{
    public int m_intOne;
    public string m_stringOne;
    public string m_notSerialized;
}
```

```

public TestISerializable()
{
    m_stringOne = "one";
    m_intOne = 1;
    m_notSerialized = "not";
}

private TestISerializable(SerializationInfo info, StreamingContext context)
{
    m_intOne = info.GetInt32("m_intOne");
    m_stringOne = info.GetString("m_stringOne");
    m_notSerialized = "transient";
}

void ISerializable.GetObjectData(SerializationInfo info, StreamingContext context)
{
    info.AddValue("m_intOne", m_intOne);
    info.AddValue("m_stringOne", m_stringOne);
}
}

```

Collections using OptimizedPersistable.Equals and GetHashCode

Note that `OptimizedPersistable` overrides `Equals` and `GetHashCode`

```

public override bool Equals(Object obj)
{
    OptimizedPersistable otherPersistentObject = obj as OptimizedPersistable;
    if (otherPersistentObject != null)
    {
        if (otherPersistentObject.IsPersistent && IsPersistent)
            return m_id.Equals(otherPersistentObject.m_id);
        return base.Equals(obj);
    }
    else
        return false;
}

public override int GetHashCode()
{
    if (m_id == 0)
        return base.GetHashCode();
    return (int)Oid.DatabaseNumber(Id) << 24 + (int)Id;
}

```

As you can see the behavior is different when object becomes persistent. If you use these functions for objects that you want to use persistently then it is **VERY** important that such objects are persisted **BEFORE** being used with `Equals` and/or `GetHashCode` or else you will end up with a corrupt `HashSet` or whatever way you triggered use of these methods.

DateTime

It is good practice to persist all [DateTime](#) structures using Coordinated Universal Time (UTC) [DateTimeKind](#). If you store `DateTime` using `DateTimeKind.Local`, it is your responsibility to also store/track [TimezoneInfo](#), it is not stored with `DateTime`.

Database Schema

VelocityDB maintains a special database, 1.odt, for all database schema. Objects in this database of type `VelocityDbType`, `TypeVersion` and `DataMember` describes the types and fields your application persists. It is **important** that once you persist an instance of a class that this class remains within your application anytime you access your

databases. Otherwise, database schema will not be able to resolve schema class with a .NET type. If you accidentally do this, it is possible to delete such an entry after you make sure there isn't any instances of it stored in any database. Contact us for assistance if this is required. You can also add an empty (stub) class of the missing type so that it resolves to something at schema load time.

Register all types that you plan on persisting

It is not mandatory, but by doing so you ensure that schema is created one way no matter in what order you persist objects and you avoid potential lock conflicts with the schema database (1.odb). For VelocityGraph, we do this the first time a Graph is persisted as:

```
public override UInt64 Persist(Placement place, SessionBase session, bool persistRefs = true,
                             bool disableFlush = false, Queue<IOptimizedPersistable> toPersist = null)
{
    if (IsPersistent)
        return Id;
    session.RegisterClass(typeof(Graph));
    session.RegisterClass(typeof(BTreeMap<EdgeTypeId, EdgeTypeId>));
    session.RegisterClass(typeof(PropertyType));
    session.RegisterClass(typeof(VertexType));
    session.RegisterClass(typeof(VelocityDbList<VertexType>));
    session.RegisterClass(typeof(EdgeType));
    session.RegisterClass(typeof(UnrestrictedEdge));
    session.RegisterClass(typeof(VelocityDbList<Range<ElementId>>));
    session.RegisterClass(typeof(VelocityDbList<EdgeType>));
    session.RegisterClass(typeof(Range<VertexId>));
    session.RegisterClass(typeof(BTreeSet<Range<VertexId>>));
    session.RegisterClass(typeof(BTreeSet<EdgeType>));
    session.RegisterClass(typeof(BTreeSet<EdgeIdVertexId>));
    session.RegisterClass(typeof(BTreeMap<EdgeId, ulong>));
    session.RegisterClass(typeof(BTreeMap<EdgeId, UnrestrictedEdge>));
    session.RegisterClass(typeof(BTreeMap<string, PropertyType>));
    session.RegisterClass(typeof(BTreeMap<string, EdgeType>));
    session.RegisterClass(typeof(BTreeMap<string, VertexType>));
    session.RegisterClass(typeof(BTreeMap<VertexId, BTreeSet<EdgeIdVertexId>>));
    session.RegisterClass(typeof(BTreeMap<VertexType, BTreeMap<VertexId, BTreeSet<EdgeIdVertexId>>>));
    session.RegisterClass(typeof(BTreeMap<EdgeType, BTreeMap<VertexType, BTreeMap<VertexId,
BTreeSet<EdgeIdVertexId>>>>));
    session.RegisterClass(typeof(BTreeMap<string, BTreeSet<ElementId>>));
    session.RegisterClass(typeof(BTreeMap<int, BTreeSet<ElementId>>));
    session.RegisterClass(typeof(BTreeMap<Int64, BTreeSet<ElementId>>));
    session.RegisterClass(typeof(PropertyTypeT<bool>));
    session.RegisterClass(typeof(PropertyTypeT<int>));
    session.RegisterClass(typeof(PropertyTypeT<long>));
    session.RegisterClass(typeof(PropertyTypeT<double>));
    session.RegisterClass(typeof(PropertyTypeT<DateTime>));
    session.RegisterClass(typeof(PropertyTypeT<string>));
    session.RegisterClass(typeof(PropertyTypeT<IComparable>));
    session.RegisterClass(typeof(AutoPlacement));
    return base.Persist(place, session, persistRefs, disableFlush, toPersist);
}
```

If your application schema is using [indexes](#)

The following is from the test `aaa_IndexRegisterClass` in project `NUnitTests` and class is `IndexingTest`

```
public class InsuranceCompany : OptimizedPersistable
{
    [Index]
    [UniqueConstraint]
    [OnePerDatabase]
    string name;
    string phoneNumber;

    public InsuranceCompany(string name, string phoneNumber)
```

```

{
    this.name = name;
    this.phoneNumber = phoneNumber;
}

[FieldAccessor("name")]
public string Name
{
    get
    {
        return name;
    }
}
}

[UniqueConstraint]
[Index("_registrationState,_registrationPlate")]
public class Car : Vehicle
{
    string _registrationState;
    string _registrationPlate;
    [Index]
    InsuranceCompany _insuranceCompany;
    string _insurancePolicy;

    public Car(string color, int maxPassengers, int fuelCapacity, double litresPer100Kilometers, DateTime modelYear,
        string brandName, string modelName, int maxSpeed, int odometer, string registrationState, string registrationPlate,
        InsuranceCompany insuranceCompany, string insurancePolicy):base(modelYear,color, maxPassengers, fuelCapacity, litresPer100Kilometers,
brandName, modelName, maxSpeed, odometer)
    {
        _registrationState = registrationState;
        _registrationPlate = registrationPlate;
        _insuranceCompany = insuranceCompany;
        _insurancePolicy = insurancePolicy;
    }
    [FieldAccessor("_registrationState")]
    public string RegistrationState => _registrationState;
    [FieldAccessor("_registrationPlate")]
    public string RegistrationPlate => _registrationPlate;
}

```

You should register a few additional index related classes:

```

session.RegisterClass(typeof(IndexDescriptor));
session.RegisterClass(typeof(BTreeSetOidShort<IndexDescriptor>));
session.RegisterClass(typeof(CompareByField<IndexDescriptor>));
session.RegisterClass(typeof(Indexes));
session.RegisterClass(typeof(VelocityDbList<OptimizedPersistable>));

```

For each of your classes that uses indexes (replace with your class name)

```

session.RegisterClass(typeof(CompareByFieldIndex<InsuranceCompany>));
session.RegisterClass(typeof(BTreeSetOidShort<InsuranceCompany>)); // short due to [OnePerDatabase] on this Index
// normally it would be instead
session.RegisterClass(typeof(VelocityDb.Collection.Comparer.CompareByFieldIndex<Car>));
session.RegisterClass(typeof(BTreeSet<Car>));

```

```

// If you are using Reference
session.RegisterClass(typeof(Reference));
session.RegisterClass(typeof(BTreeSet<Reference>));

```

If base classes have indexes (like Vehicle in this example), you need to do it for such classes as well.

Fixed size class instances and limiting string size

Objects of a class that has only fixed size fields can be stored without specifying an object size. This saves four bytes per object and such objects can in some cases be looked up by byte offset. You can make a string field fixed size by using the `StringLength` attribute as in

```

public class TickOptimized : OptimizedPersistable
{
    [StringLength(8)]

```



```
string m_symbol;  
DateTime m_timestamp;  
double m_bid;
```

In this case `m_symbol` will be stored using 8 bytes. We interpret length as number of bytes, not number of characters.

You can calculate how many bytes a certain string uses in persisted state with

```
SessionBase.TextEncoding.GetByteCount(string str);
```

Adding or removing field(s) from a class with existing objects in a database

After making changes to a class, in an update transaction call `session.UpdateClass(typeof(UpdatedClass));` as done in the sample application [UpdateClass](#). This updates the schema to reflect the changes to your class, a new version of the class is created as a new instance of `TypeVersion`. Objects associated with prior versions of this type are migrated to the updated class in memory when read from a database. To make such objects permanently be shaped as the latest version of your class `TypeVersion`, you need to update the object with a call to `UpdateTypeVersion()`. If you fail to call `UpdateClass`, it can lead to exceptions and failures to read/write objects of the Type

Changing a field type without losing already persisted data

We wanted to make a change to our `BTree/BTreeMap` collection to reduce memory usage and improve performance. To do this while still preserving already persisted data of these types. This is how we did it.

First change class definitions by setting prior version usage field as `[NonSerialized]` and add new version of field as in:

```
[NonSerialized]  
internal VelocityDbList<BTreeLeafBase<Key, Value>> nodeList;  
internal WeakReferenceListBase<BTreeLeafBase<Key, Value>> _nodeList;
```

When we read an object of this type as it was before this change, `nodeList` will be set and `_nodeList` will be null so to make the switch to new field type add to this class code like:

```
public override void InitializeAfterRead(SessionBase session)  
{  
    base.InitializeAfterRead(session);  
    if (nodeList != null)  
    {  
        _nodeList = new WeakReferenceList<BTreeLeafBase<Key, Value>>(nodeList.Count, Session);  
        foreach (var e in nodeList)  
            _nodeList.Add(e);  
        if (session.InUpdateTransaction)  
        {  
            session.UpdateClass(GetType());  
            UpdateTypeVersion();  
            nodeList.Unpersist(session);  
        }  
    }  
}
```

After updating all your persisted objects to the new field type, you can remove the `[NonSerialized]` field and the `InitializeAfterRead` override.

Renaming a persisted class or moving it to a different namespace

The `UpdateClass` sample shows how to do this.

Example usage

```
session.ReplacePersistedType(typeof(VelocityDbSchema.Samples.UpdateClass.UpdatedClass),  
typeof(UpdateClass.UpdatedClass));
```

and back again using alternate API

```
session.ReplacePersistedType(typeof(UpdateClass.UpdatedClass).AssemblyQualifiedName,  
typeof(VelocityDbSchema.Samples.UpdateClass.UpdatedClass));
```

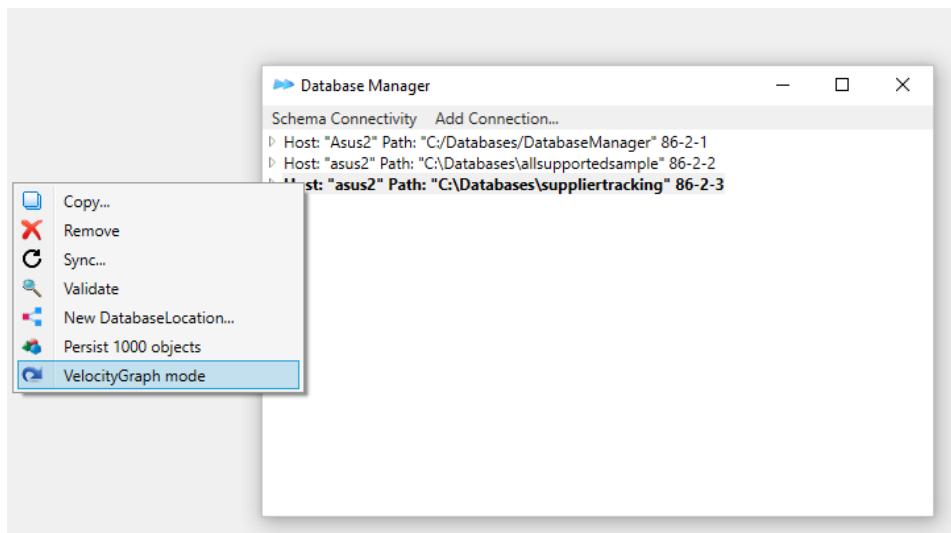
NOTE: The type you are replacing with must not already exist in the database schema. Make sure both old type and replacement type contains the exact same fields.

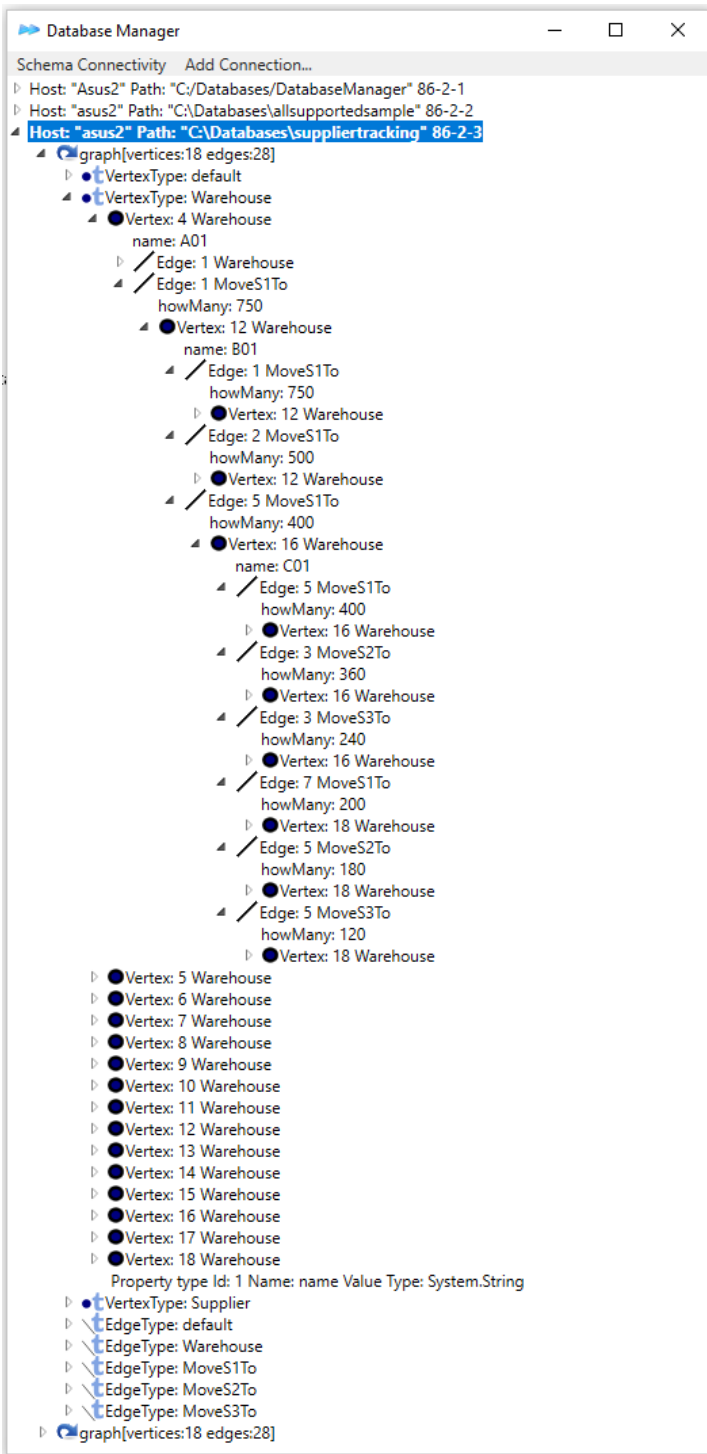
VelocityGraph

Some of the content in this guide does not apply to users that only use VelocityGraph with simple property values such as numbers and strings. As a strict VelocityGraph user you do not need to worry about calling Update() before updating an object and schema is static, only what the base VelocityGraph uses.

Visualizing a VelocityGraph

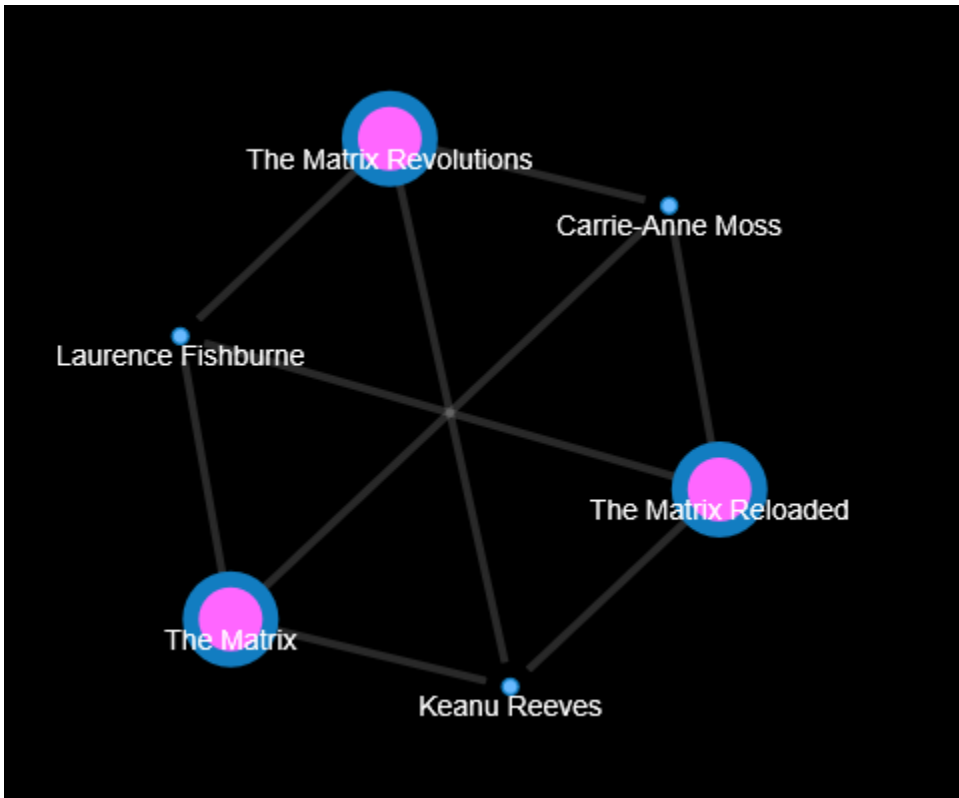
Our [DatabaseManager](#) now includes a **VelocityGraph mode**. Right click to bring up menu. In this mode you see objects as you work with them in VelocityGraph, the other mode (default) shows how objects are stored.



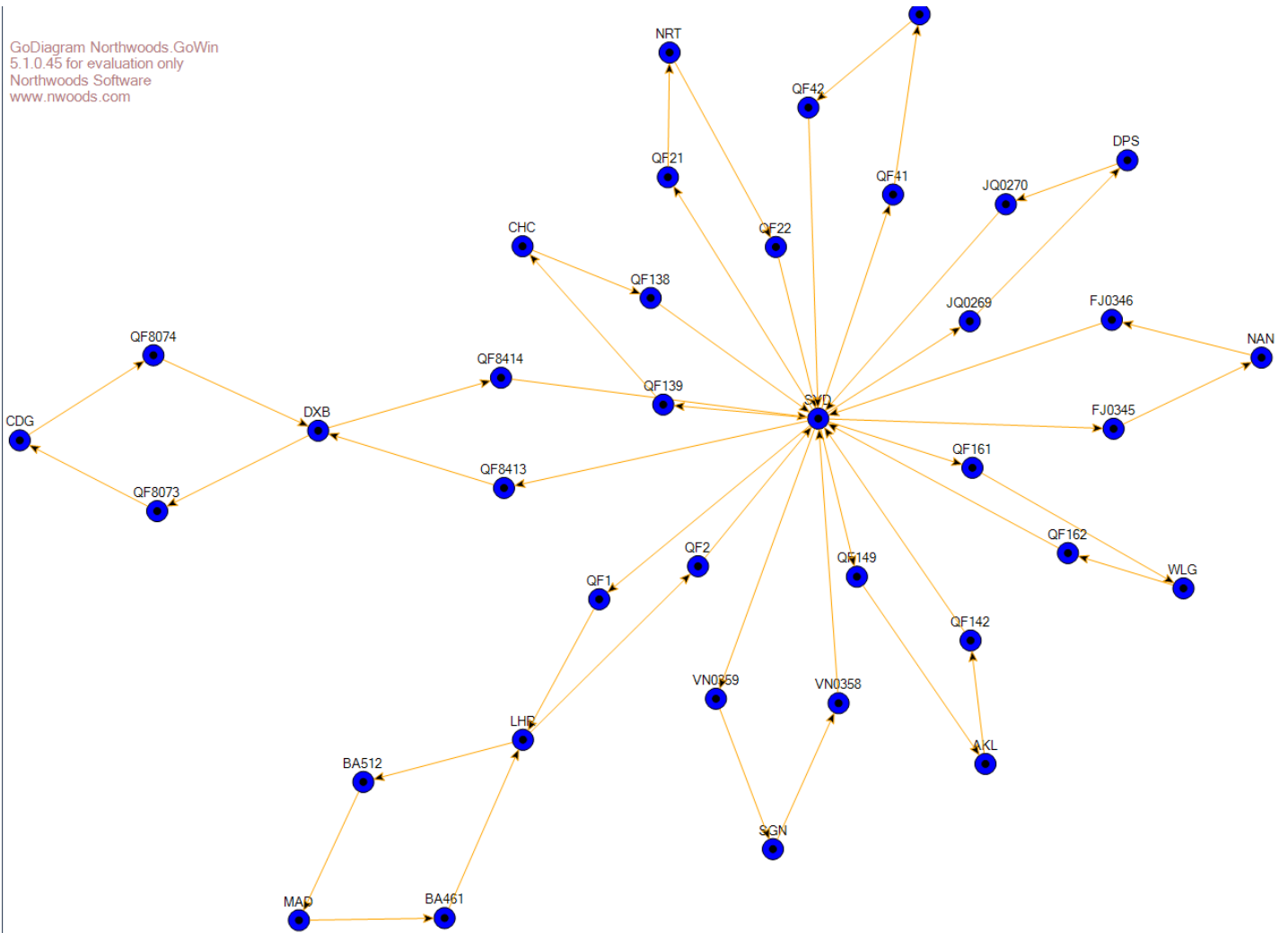


You can export a Graph to [GraphJson](#) and then use [Alchemy.js](#) to visualize the graph.

I.e. the exported graph of the QuickStart VelocityGraph can look like



Other alternatives include using [Northwoods Software](#). The following graph/diagram was created with very simple C# code from a VelocityGraph.



Persistent placement of objects

The placement (location) of persistent objects affects performance and locking. It is therefore important to make decisions about where to place an object when making it persistent. Once an object has been persisted, it remains in the same location for its persistent life time. You can decide how many objects you want on a single page. For slightly improved storage, require that a page only may contain objects of a specific type. Also fixed size objects (ones with no contained variable size arrays) can further improve object store efficiency. Several ways of controlling the placement when persisting object are provided. First on [IOptimizedPersistable](#) the following helps guide the placement:

```
UInt16 ObjectsPerPage  
{  
    get;  
}
```

Best way to persist an object

The recommended way of persisting objects is using the [SessionBase](#) api:

```
public UInt64 Persist(object obj)
```

When this api is used, each type is stored in its own database. For **best performance** avoid explicitly persisting objects unless: an object is a root object (not referenced by other persisted objects), includes an `[AutoIncrement]` field (unless you don't care what number gets assigned), used in a `VelocityDBWeakReference` or is indexed and you can't wait for the

index update to happen at transaction commit. Objects not persisted explicitly will be made persistent automatically by reachability from a persisted object.

Add the attribute `[NonSerialized]` for each class field you don't want to be persisted.

It is recommended that you make the following override in your `OptimizedPersistable` subclass for better performance:

```
public override bool AllowOtherTypesOnSamePage
{
    get
    {
        return false;
    }
}
```

We may make this default but it could break existing code so it is not a trivial change.

Customizing object placement (most of you can skip this part)

In addition the `IOptimizedPersistable` interface contains API intended for customizing how fields of an object being persisted are to be persisted (including where to place).

```
UInt64 Persist(Placement place, SessionBase session, bool persistRefs = false, bool disableFlush = false);
```

```
UInt64 Persist(SessionBase session, IOptimizedPersistable placeHint, bool persistRefs = false, bool disableFlush = false);
```

```
for (int i = 0; i < numberOfPersons; i++)
```



```
{
    person = new Person();
    person.Persist(session, person);
}
```

```
for (int i = 0; i < numberOfPersons; i++)
```



```
{
    person = new Person();
    if (priorPerson == null)
        priorPerson = person;
    person.Persist(session, priorPerson); // use prior person as object to persist near
    priorPerson = person;
}
```

The second way of controlling the placement while persisting an object is by using persistent or transient instances of the `Placement` class.

```
public Placement(UInt32 db, UInt16 page = 1, UInt16 slot = 1, UInt16 objectsPerPage = 10000, UInt16
pagesPerDatabase = 10000, bool persistRefs = false, bool tryOtherDatabaseIfLockConflict = true, UInt32
maxNumberOfDatabases = UInt32.MaxValue, bool allowOtherTypesOnSamePage = true, bool flushFullPages =
true)
```

```
public Placement(SessionBase session, IOptimizedPersistable placementProviderObject,
IOptimizedPersistable objectToPlace, bool persistRefs = false, UInt32 maxNumberOfDatabases =
UInt32.MaxValue, bool flushFullPages = true)
```

There is also additional API on `Placement` for fine tuning the placement. An instance of `Placement` is used as parameter to the `IOptimizedPersistable` `Persist` API mentioned above.

Sometimes it an advantage to put all related objects in a single database because then 32bit, `OidShort`, object references can be used instead of full 64 bit, `Oid`, object references. A short object reference contains only a page and slot part (16 bit each). Such references use less storage space and if only short references are used within a database, such a database can easily be cloned since it's database number isn't hard coded anywhere within the database. Short references are not automatically used when you place objects this way. The application must explicitly request it in the class definition by using the attribute `[UseOidShort]`. There are also special short references versions of the provided `BTree` collections. The application needs to use those instead of the long reference `BTree` collections when you want all objects within a database to use short references.

How to optimally place/persist objects is application dependent. The sample programs provided try to illustrate some of many use cases for object placement.

Controlling placement of objects persisted by reachability

Be default when you persist some object using the [recommended method](#), all objects reachable from this object are also persisted by the same method. You can override this behavior for persisting reachable objects by overriding the property `IOptimizedPersistable.PlacementDatabaseNumber` to return something different than `Placement.DefaultPlacementDatabaseNumber`.

You can further control the persist of objects by overriding the `Persist` function as in:

```
public override UInt64 Persist(Placement place, SessionBase session, bool persistRefs = true, bool
disableFlush = false, Queue<IOptimizedPersistable> toPersist = null)
{
    base.Persist(place, session, false, disableFlush, toPersist);
    keyArray.Persist(place, session, true, disableFlush, toPersist);
    return Id;
}
```

Looking up objects

The most efficient way is to have one or a few root objects that you look up by the object identifier as in:

```
ImdbRoot imdbRoot = (ImdbRoot)session.Open(session.DatabaseNumberOf(typeof(ImdbRoot)), 2, 1, false);
```

When you open an object this way, all objects referenced by the object is also connected to the object so then to reach related objects all you need to do is navigate to related objects such as in:

```
imdbRoot.ActingByNameSet
BTreeSet<Word> wordSet = indexRoot.lexicon.wordSet;
```

Another way to lookup objects is by using a LINQ query such as:

```
var result = (from ComputerFileData computerFileData in session.AllObjects<ComputerFileData>()
where computerFileData.FileID == 500000
select computerFileData).First();
```

or you can accomplish the same lookup without using LINQ as:

```
var computerFileDataEnum = session.AllObjects<ComputerFileData>();
foreach (ComputerFileData computerFileData in computerFileDataEnum)
{
    if (computerFileData.FileID == 500000)
        break; // found it
}
```

The third way is by looking up from a collection (usually a BTree) as in:

doc.WordHit.TryGetValue(word, out wordHit) or via an [index lookup](#).

DO NOT reference persistent data using static variables

It is not OK to have variables like

```
static VertexType movieType;
static PropertyType movieTitleType;
static PropertyType movieYearType;
```

Updating persistent objects

VelocityDB need to be notified when you want a change to an object to be persisted. The safest way to do this, is to define a property for every field your application data objects have, such as:

```
[FieldAccessor("m_bestFriend ")]
public Person BestFriend
{
    get
    {
        Session?.LoadFields(); // Loads all fields of an object if they are not already loaded.
        return m_bestFriend;
    }
    set
    {
        Update(); // IMPORTANT, call Update() before updating object
        m_bestFriend = value;
    }
}
```

If updating a field that is NOT indexed you can avoid the index update cycle by calling the object update function on `SessionBase` instead of `OptimizedPersistable Update()` as in

```
public string StreetAddress
{
    get
    {
        return m_streetAddress;
    }
    set
    {
        UpdateNonIndexField();
        m_streetAddress = value;
    }
}
```

Note - Do not use any VelocityDB API between `Update()` and the field update or a VelocityDB API as part of part of the field update otherwise the update may not be persisted as this can cause the object page to be flushed.

VelocityDB collection classes like `VelocityDbList<T>`, `BTreeSet<Key>` and `BTreeMap<Key, Value>` calls update automatically internally so you don't need and should not call `Update()` when modifying such collections.

When updating objects that are not implementing `IOptimizedPersistable`, call `session.UpdateObject`. `BindingList<MyItem>` is such a case. Exception are: `List<>`, arrays and `ValueType` objects when embedded in an object that implements `IOptimizedPersistable`. For such lists call `Update()` on the object embedding the list.

```
public class MyContainer : OptimizedPersistable
{
    private BindingList<MyItem> m_items;
    public BindingList<MyItem> Items {
        get { return m_items; }
    }

    public MyContainer()
    {
        m_items = new BindingList<MyItem>();
    }

    public bool UpdateBindingList(SessionBase session)
    {
        return session.UpdateObject(m_items);
    }
}
```

Deleting (unpersisting) persistent objects

Use `OptimizedPersistable.Unpersist` or `Page.UnpersistObject` or `SessionBase.DeleteObject`. You can override the default implementation of `public virtual void Unpersist(SessionBase session, bool disableFlush = true)`, i.e.

```
public override void Unpersist(SessionBase session, bool disableFlush = true)
{
    if (id == 0)
        return;
    if (comparisonByteArrayId != 0)
    {
        comparisonBytesTransient = (BTreeByteArray)session.Open(comparisonByteArrayId);
        comparisonBytesTransient.Unpersist(session, disableFlush);
        comparisonByteArrayId = 0;
    }
    nodeList.Unpersist(session, disableFlush);
    base.Unpersist(session, disableFlush);
}
```

Referential integrity

When removing an object from a database, it is important that references to this object also are removed. Otherwise such references may end up referencing some other object or become a null reference.

It is recommended that you maintain two way relation (bidirectional) as much as possible because that makes it easier to cleanup references and also to diagnose dangling references when they occur.

Interface `IReferenceTracked` and class `ReferenceTracked` was added as an aid to maintain referential integrity. A simple sample project named `Relations` shows how this API can be used.


```

if (info==null) {
    ThrowHelper.ThrowArgumentNullException\(ExceptionArgument.info\);
}
info.AddValue(VersionName, version);

#if FEATURE_RANDOMIZED_STRING_HASHING
info.AddValue(ComparerName, HashHelpers.GetEqualityComparerForSerialization\(comparer\), typeof\(IEqualityComparer<TKey>\));
#else
info.AddValue(ComparerName, comparer, typeof\(IEqualityComparer<TKey>\));
#endif
info.AddValue(HashSizeName, buckets == null ? 0 : buckets.Length); //This is the length of the bucket array.
if( buckets != null) {
    KeyValuePair<TKey, TValue>\[\] array = new KeyValuePair<TKey, TValue>\[Count\];
    CopyTo(array, 0);
    info.AddValue(KeyValuePairsName, array, typeof\(KeyValuePair<TKey, TValue>\[\]\));
}
}

```

This also makes the objects use up more space as persistent. Instead of persisting Dictionary use VelocityDB collection [BTreeMap](#) and instead of HashSet use [BTreeSet](#).

Using the provided BTree collections

Just about all object oriented applications need to use collections. VelocityDB provides [BTree](#) collections which are similar to BTree's of the variety B*. A BTree is a collection where the added objects are sorted. An application can define the sort order by defining a subclass of [VelocityDbComparer<Key>](#) or by using the class [CompareByField<Key>](#), a collection may also have a `null` comparator in which case the objects are ordered by the object identifier or by the [ValueType](#) ordering as defined by the objects `public override int CompareTo(object obj)` implementation. The BTree comes in a few varieties, a key only version and a key value version. They also have a long object Id (db-page-slot) version and a short Id (page-slot) version. A BTree can be used with `comparisonByteArray` data which is used to cache object key data within the BTree nodes so that when a binary search takes place we can avoid opening objects to compare. When you use the predefined class [CompareByField<Key>](#) it is easy to add `comparisonByteArray` data to the BTree nodes, you just specify how many bytes per object it should be and whether the cached node byte contains the entire data being compared when deciding if one object is less, equal or greater compared to another. If you customize building your own comparator, managing the `comparisonByteArray` becomes a little trickier; on the compare class you need to define `SetComparisonArrayFromObject` as in:

```

public override void SetComparisonArrayFromObject(Word key, byte\[\] comparisonArray, bool oidShort)
{
    Int32 hashCode = key.aWord.GetHashCode();
    Buffer.BLockCopy(BitConverter.GetBytes\(IPAddress.HostToNetworkOrder\(hashCode\)\), 0, comparisonArray,
0, comparisonArray.Length);
}

```

In this case we are sorting by the hash code of a string, the corresponding compare function in this case looks like:

```

public override int Compare(Word a, Word b)
{
    UInt32 aHash = (UInt32) a.aWord.GetHashCode();
    UInt32 bHash = (UInt32) b.aWord.GetHashCode();
    int value = aHash.CompareTo(bHash);
    if (value != 0)
        return value;
    return a.aWord.CompareTo(b.aWord);
}

```

A problem here is that a String `GetHashCode()` returns different values on a 32 bit platform then a 64 bit platform. To make your data cross platform compatible don't use the string `GetHashCode`, instead build your own string hash code function. We do so in the VelocityDB build in class [HashCodeComparer<T>](#).

Btree classes provided:

- [BTreeSet<Key>](#)
- [BTreeSetOidShort<Key>](#)

- `BTreeMap<Key, Value>`
- `BTreeMapOidShort<Key, Value>`

Sample usage:

```
public Lexicon(ushort nodeSize, HashCodeComparer<Word> hashComparer, SessionBase session)
{
    wordSet = new BTreeSet<Word>(hashComparer, session, nodeSize);
}
```

[BTreeMap<Key, Value>](#)

Represents a collection of keys that is maintained in sorted order. Each key has an associated value. A persistent BTree references its contained objects by Oid instead of direct object references. This way, we will only open the referenced objects on demand which reduces memory usage and initial BTree load time. Exceptions are ValueType keys and values.

For more see <https://velocitydb.com/Help/html/f12b67ba-577a-7b2e-43a4-d489688f753e.htm>

Indexes

Indexes is a simplified, automated, way of implicitly defining and keeping `BTreeSet<Key>`s up to date when objects are added, deleted and updated. An index is defined by using the class or field `[Index]` attribute. Indexes for a persistent Type is stored in its own system selected database, the range of databases used is between 66000 up to 66000 + the number of Types and versions of a type that your application store persistently. An object gets added to its indexes when an object is persisted. Make sure to set all indexed fields to desired indexed values before persisting object. When an indexed object is updated, its indexes get updated when the page of the objects gets flushed to disk. You can force it to be flushed to disk and have the index updated by calling `Write()` on the object you updated (after you made the changes and object is an `OptimizedPersistable`). If you made a change that does not affect the index, you did not modify an indexed field, you don't need to update the index explicitly since the index is unaffected. An object is removed from its indexes when it is unpersisted and when `Update()` is called. If you want to index objects separately for each Database, tag the class or field with the attribute `[OnePerDatabase]`. Before modifying an indexed field, it is important to call `Update()` on the object having the field before doing the update because the object needs to be removed from its indexes before updates or else the removal code will fail to find the object in its indexes leading to an index corruption. Call `FlushUpdates()` or `FlushUpdates(Database db)` on the session after the changes have been made to add it back to indexes. Use only with subclass of `OptimizedPersistable`.

Using a worker thread to add indexed objects to its indices

Starting in VelocityDB 4.5, we added a feature that relieves the main database thread from the work of adding objects to indices. This feature is available with `SessionNoServerShared`. You can make the indexing happen in the main database thread by setting `session.AddToIndexInSeperateThread = false;` If object indexed contains an `[OnePerDatabase]` index then indexing will happen in main session thread.

Class level index

When you want an index with compound keys, like order by *lastName* and then if two or more *lastnames* are equal by *firstName* and if two or more *firstNames* are equal, order these otherwise equal objects by yet another field name and so on. We currently only allow one class level index (by multiple compound keys) per class.

```
[Index("modelYear,brandName,modelName,color")]
public abstract class Vehicle : OptimizedPersistable
{
    string color;
    int maxPassengers;
    int fuelCapacity; // fuel capacity in liters
    double litresPer100Kilometers; // fuel consumption
}
```

```

DateTime modelYear;
string brandName;
string modelName;
int maxSpeed; // km/h
int odometer; // km

```

You can also use the class level Index attribute without specifying any field names; in that case the contained objects are sorted by the default ordering of the class which is normally by Oid (Id).

Using a class level index

To iterate all Cars in index sorted order

```

foreach (Car c in session.Index<Car>())
    Console.WriteLine(c.ToStringDetails(session));

```

Index by a field

This type of index sorts all persistent instances of a class by a field value. Note that in order to use this type of index in a LINQ query, you need to tell us what property that returns the value of the field. You do that by the `FieldAccessor` attribute as in sample class below. The `[UniqueConstraint]` attribute can be added when you don't want multiple objects with the same field value in the index. An exception is raised if you add a second object with the same field value when `[UniqueConstraint]` is applied to the field. The `[IndexStringByHashCode]` attribute can also be added to string field indexes when you don't care about the sort order. Sorting by hash code is faster than sorting by the normal string ordering.

```

public class InsuranceCompany : OptimizedPersistable
{
    [Index]
    [UniqueConstraint]
    [OnePerDatabase]
    string name;
    string phoneNumber;

    public InsuranceCompany(string name, string phoneNumber)
    {
        this.name = name;
        this.phoneNumber = phoneNumber;
    }

    [FieldAccessor("name")]
    public string Name
    {
        get
        {
            return name;
        }
    }
}

```

Using the index by field in a LINQ query

In every source file that uses an index in a query, it is important to have

```

using static VelocityDBExtensions.Extensions.BTree.BTreeExtensions;

```

This activates the BTree extension methods that overrides the default Enumerable versions. You should see much improved performance when using the extension methods. The following extensions methods are defined:

```

static public IEnumerable<Key> Where<Key>(this BTreeBase<Key, Key> sourceCollection,
Expression<Func<Key, bool>> expr)

```

```
static public int Count<Key>(this BTreeBase<Key, Key> sourceCollection)
```

Let us know if you want other “slow” method overrides of Enumerable with BTree.

The extensions are located in a separate assembly, so you will also need to add a reference to it or use the VelocityDBExtensions NuGet.

If you don’t want to drag in all the additional assembly references, we are currently duplicating the BTree extensions code within the VelocityDB assembly.

So you can use `using static VelocityDb.Collection.BTree.Extensions.BTreeExtensions`; instead to avoid these additional dependencies. We put this code in the open source VelocityDBExtensions so that YOU could help us improve this complicated expression tree code! Any assistance is appreciated and will be rewarded with a VelocityDB license discount.

```
var q = from company in session.Index<InsuranceCompany>("name")
where company.Name == "AAA" select company;
```

```
foreach (InsuranceCompany company in q)
    Console.WriteLine(company.ToStringDetails(session)); // only one will match
```

Enable index usage trace

Not every LINQ query will end up using the fast path with direct index lookups instead of the default Enumerable.Where, this can be because your query contains non indexed fields or because the linq query somehow does not match the Enumerable.Where extension provided with VelocityDB. To find out, enable index tracing by calling `session.TraceIndexUsage = true;` If index is used by a query then you will see output to Console like:

```
20:42:12:982 Index used with BTreeSet<Country> 66206-1-1
```

If index is not used, there will be no output to Trace. If you also want output to Console add the code:

```
Trace.Listeners.Add(new ConsoleTraceListener());
```

Simplify the query as much as possible

The following query will use the fast path

```
BTreeSet<Country> countryIsoIndex = session.Index<Country>("ISO");
string homeCountry = (string)airline_element.Element("Home_Country");
```

```
var res_country_q = from country in countryIsoIndex
where country.I_ISO == homeCountry
select country;
Country res_country = res_country_q.FirstOrDefault();
```

The following equivalent will use the slow non VelocityDB enumeration. The thing that makes it not use the VelocityDB extension is specifying the type of country (`Country country`). Leave it out and it will be much faster! Anyone knows why???

```
var res_country = (from Country country in session.Index<Country>("ISO")
where country.I_ISO == (string)airline_element.Element("Home_Country")
select country).First();
```

Changing indexing for a class after objects of that type already persisted

Changing indexing is handled the same way as any changes to a class definition. For example, if you start out with the following class definition and you commit some

```
[Index("modelYear,brandName,modelName,color")]
public abstract class Vehicle : OptimizedPersistable
```

```

{
[Index]
string color;
int maxPassengers;
int fuelCapacity; // fuel capacity in liters
[Index]
double litresPer100Kilometers; // fuel consumption
[Index]
[UniqueConstraint]
Guid guid = Guid.NewGuid();
DateTime modelYear;
[Index]
[IndexStringByHashCode]
string brandName;
string modelName;
List<VelocityDbSchema.Person> owners;
int maxSpeed; // km/h
int odometer; // km

```

AND later change it to

```

[Index("modelYear,brandName,modelName,color")]
public abstract class Vehicle : OptimizedPersistable
{
string color;
int maxPassengers;
int fuelCapacity; // fuel capacity in liters
[Index]
double litresPer100Kilometers; // fuel consumption
[Index]
[UniqueConstraint]
Guid guid = Guid.NewGuid();
DateTime modelYear;
[Index]
[IndexStringByHashCode]
string brandName;
string modelName;
List<VelocityDbSchema.Person> owners;
int maxSpeed; // km/h
[Index]
int odometer; // km

```

You will need to convert all your existing Vehicle objects to this updated class definition.

```

session.UpdateClass(typeof(Vehicle));
foreach (var v in session.AllObjects< Vehicle >())
{
    v.UpdateTypeVersion();
}

```

This code will remove all Vehicle objects from the “string color” index and will create a new index and add all Vehicle to “int odometer” index.

System.OutOfMemoryException

Make sure that your process is not running as a 32-bit process on a 64-bit Windows, as a 32-bit process you will get the OutOfMemoryException at around 1.5 GB. Use the Task Manager as a way to determine if your process runs as a 64bit process. 32-bit processes has their name appended with the string “(32 bit)”, also do not use the “Visual Studio Hosting Process” – it’s in your projects Debug options - if it is running as a 32 bit process. If your project is using .NET 4.5 make sure that you do not have the option “Prefer 32 bit” set. If this isn’t set but your process still is 32 bit then change to use

.NET 4.0 as a work around. If you absolutely need to run your process as 32-bit then tell VelocityDB to limit its caching by setting: `DataCache.MaximumMemoryUse = 1100000000`; to limit the memory usage.

Limiting graph of objects in memory

When an object is opened by a session object, all object referenced by that object are also brought into memory. In some cases that isn't desired. You can limit the size of such graphs by using `WeakReferenceList` or the `BTree` collections which avoids bringing in all the contained objects. These collections avoids bringing in all referenced objects by not having straight forward C# object references everywhere; instead references are replaced by the object identifier of the referenced object, as in:

```
internal UInt64 comparisonByteArrayId;
internal UInt64[] keysArray;
internal UInt64[] valuesArray;
```

Here each `UInt64` is the Id of some persistent object. The `BTree` fetches such objects on demand:

```
internal override Key GetKey(int index)
{
    if (IsPersistent && UseAlternateKeys == false)
        return Session.Open<Key>(keysArray[index]);
    else
        return keysArrayAlternate[index];
}
```

For single non array references VelocityDB provides `WeakIOptimizedPersistableReference<T>` as in:

```
aMan.spouse = new WeakIOptimizedPersistableReference<VelocityDbSchema.Person>(aWoman);
```

to get the value use `public T GetTarget(bool update, SessionBase session)`.

You can examine how large a loaded object graph might be by examining the Schema Type Connections using the [Database Manager](#).

Implementing your own classes with weak references

Here is one example that we use with the `AllSupported` sample project.

```
public class WeakReferencedConnection<T> : OptimizedPersistable where T : OptimizedPersistable
{
    UInt64 _objId;
    static WeakReferencedConnection()
    {
        var list = new List<Type> { typeof(T) };
        Schema.WeakReferencedTypes[typeof(WeakReferencedConnection<T>)] = list; // register this weak
        reference with schema so that DatabaseManager can recognize this as being a weak referenced object
    }

    public WeakReferencedConnection(T t)
    {
        if (!t.IsPersistent)
            throw new PersistedObjectExpectedException("Persist first");
        t.Session.Persist(this);
        _objId = t.Id;
    }

    public T MyWeakReferencedObject
    {
        get
        {
            return Session.Open<T>(_objId);
        }
    }
}
```



```
}  
}  
}
```

Using only weak references between objects

A benefit of using only weak references is that object caching can be optimized. If your application only uses weak references, such as the case with schema used with VelocityGraph, you can set:

```
SessionBase.ClearAllCachedObjectsWhenDetectingUpdatedDatabase = false;
```

This way you preserve object cache for objects in databases that are up to date in cache. Only objects in a database that is found to have been updated by another transaction is invalidated. This can be a significant performance boost depending on how often updates occur.

Lazy load of object references

Another way of limiting what gets loaded when an object is open is the `LazyLoadMembers` property on `OptimizedPersistable`

```
/// <summary>  
/// By default all fields are loaded when opening a persistent object but an option is provided to load  
members on demand (lazy loading).  
/// </summary>  
public virtual bool LazyLoadFields  
{  
    get  
    {  
        return false;  
    }  
}
```

When a class uses lazy loading of fields, each field access must make sure the field is loaded first.

```
public LazyLoadPropertyClass MyRef  
{  
    get  
    {  
        Session?.LoadFields();  
        return myRef;  
    }  
    set  
    {  
        Update();  
        myRef = value;  
    }  
}
```

Specifying depth to load at object open

An alternative to the lazy load property is to specify depth to load at object open.

```
LazyLoadByDepth lazy = (LazyLoadByDepth)session.Open(id, false, false, 0); // load only the root of the object graph
```

Session caching of databases, pages and slots

Each session object maintains a cache of databases, pages and slots. The caching is mostly using weak references. Database pages also have a strong reference cache which is released when available memory is low. By default objects and pages are cached with strong references, unless you override the session constructor parameters for this, but if an object's class overrides the `Cache` property, object caching may not happen for that type of objects. If a cached Database

is found to be out of date, all objects cached are released (even objects cached for other Databases). This is to be sure we don't end up using stale objects indirectly via object references.

Strong reference caching can be disabled by creating the session instance with a parameter that disables caching. Avoid having strong references to persistent object between transactions since a strong referenced object cannot be updated in case the object was updated by another session. Look up persistent objects from scratch in each new transaction so that stale objects can be avoided.

Here is an example of how to create a session without strong referenced page cache and without string object caching:
`using (SessionNoServer session = new SessionNoServer(s_systemDir, 5000, optimisticLocking: false, enablePageCache: false, objectCachingDefaultPolicy: CacheEnum.No)) {}`

Some sections of your code might benefit from object/page caching while other sections do not. You can control the caching as done below.

```
session.ObjectCachingDefaultPolicy = CacheEnum.No; // the following processing works faster without object caching when < 40GB memory not available
session.ClientCache.PageCacheEnabled = true; // strong reference page caching is beneficial in this case
```

It is also possible to enable object/page caching for selected databases. These settings don't persist, it is just until such objects/pages are purged from memory due to memory usage limitations or due to updates from other transactions. Such selected settings are useful when ingesting a billion objects with indexing. Turn on caching of indexing objects and its pages but not for the billion objects!

```
root.Page.Database.PageCacheEnabled = true;
root.GeoHashToNode.Page.Database.PageCacheEnabled = true;
UInt32 dbNum = session.DatabaseNumberOf(typeof(BTreeLeaf<Int64, Node>));
Database db = session.OpenDatabase(dbNum, false, false);
if (db != null)
    db.PageCacheEnabled = true;
root.Page.Database.ObjectCachingDefaultPolicy = CacheEnum.Yes;
root.GeoHashToNode.Page.Database.ObjectCachingDefaultPolicy = CacheEnum.Yes;
```

Databases are cached using weak references by default but you can force use of strong references to existing databases using api on SessionBase.

```
session.CrossTransactionCacheAllDatabases();
```

```
session.CrossTransactionCache(db, true);
```

Diagnostics

When you notice that something isn't the way it should be, maybe something is taking longer than expected, there is useful option you can turn on that logs all activities related to all database files or files of selected databases.

To turn on tracing for a specific database (in this case database 55), use SessionBase api:

```
session.SetTraceDbActivity(55);
```

To turn on tracing of all databases use: `session.SetTraceAllDbActivity();`

Handling exceptions thrown by VelocityDB

A VelocityDB application should handle exceptions thrown by the VelocityDB kernel.

```
try
```

```

{
    using (SessionNoServer session = new SessionNoServer(systemDir))
    {
        session.BeginRead();
        ...
        session.Commit();
    }
}
catch (Exception ex)
{
    Console.WriteLine(ex.ToString());
}

```

Here is a list of the current possible VelocityDB exceptions:

```

AlreadyInCommitException
AlreadyInTransactionException
DatabaseAlreadyExistsException
DatabaseDoesNotExistException
DatabaseReadLockException
DesKeyMissingException
FieldDoesNotExistException
IndexDatabaseNotSpecifiedException
IndexDatabaseOrBTreeMissingException
IndexDatabaseSpecifiedForGlobalIndexException
InternalErrorException
InTransactionException
InUpdateTransactionException
InvalidChangeOfDatabaseLocation
InvalidChangeOfDefaultLocationException
MaxNumberOfDatabasesException
NotInTransactionException
NoValidVelocityDBLicenseFoundException
NullObjectException
ObjectDoesNotExistException
ObjectNotInSameDatabaseAsOidShortCollectionException
OpenDatabaseException
OptimisticLockingFailed
PageDeadLockException
PageDoesNotExistException
PageReadLockException
PageUpdateLockException
PersistedObjectExpectedException
RequestedPlacementDatabaseNumberNotValidException
RequestedPlacementPageNumberNotValidException
SubscriptionsNotAvailableWithNoServerSessionException
SystemDatabaseNotFoundWithReadOnlyTransactionException
TryingToBeginReadOnlyTransactionWhileInUpdateTransactionException
TryingToDeleteDeletedDatabaseException
UnexpectedException
UniqueConstraintException
UpdateLockFailedException
WeakReferenceMustBePersistentException

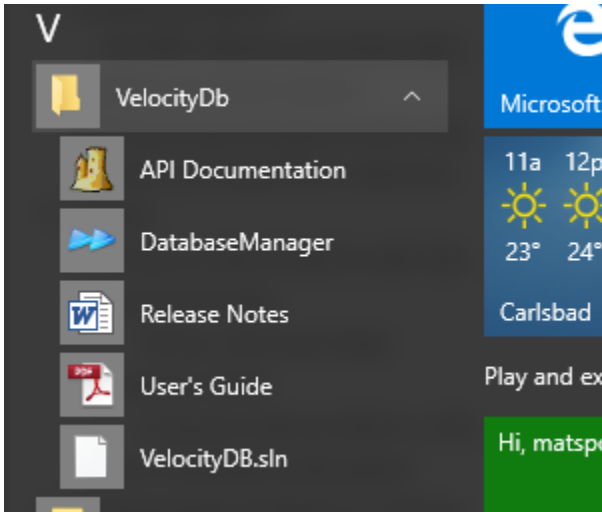
```

Database Manager

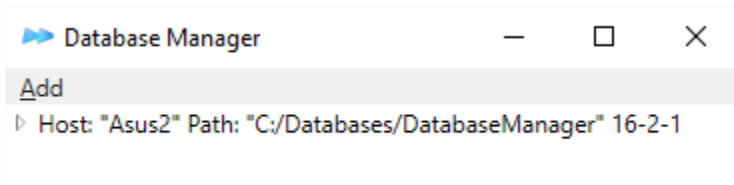
Use DatabaseManager for administrating all your databases. Using Database Manager is a great way to inspect your data, making sure it looks the way you expect it. DatabaseManager is available in the sample VelocityDB.sln provided with the VelocityDB download.

Starting Database Manager

Startup Database Manager (it is in your Start menu). Before you start it you may want to look at DatabaseManager.exe.config in your installation folder and change settings to fit your case. You also want to put your VelocityDB license database, 4.odt, into the DatabaseManager database folder.



An initial admin database is created. This database contain info about all other databases you “Add” to the Database Manager.



Objects are initially lazy loaded

This means you will need to make sure your objects are fully loaded when the object ToString method is called. If you override ToString() and it uses non primitive fields to render string, first call `Session.LoadFields(this);` to make sure all required fields are loaded.

Objects are automatically loaded once you drill down into child objects.

Browsing objects created by Baseball sample application.

Click on the “Add” menu item.

VelocityDB Connection

Database Directory

C:\Databases\baseball [Browse...](#)

Session Details

- Not using VelocityDBServer
- Using VelocityDBServer

Host

- Pessimistic Locking
- Use Windows Authentication
- Create New (if system databases missing)

WaitForMilliseconds

Assemblies

Classes Assemblies

C:\VelocityDB\Debug\VelocityDbSchema.dll

Dependency Assemblies

Restore From

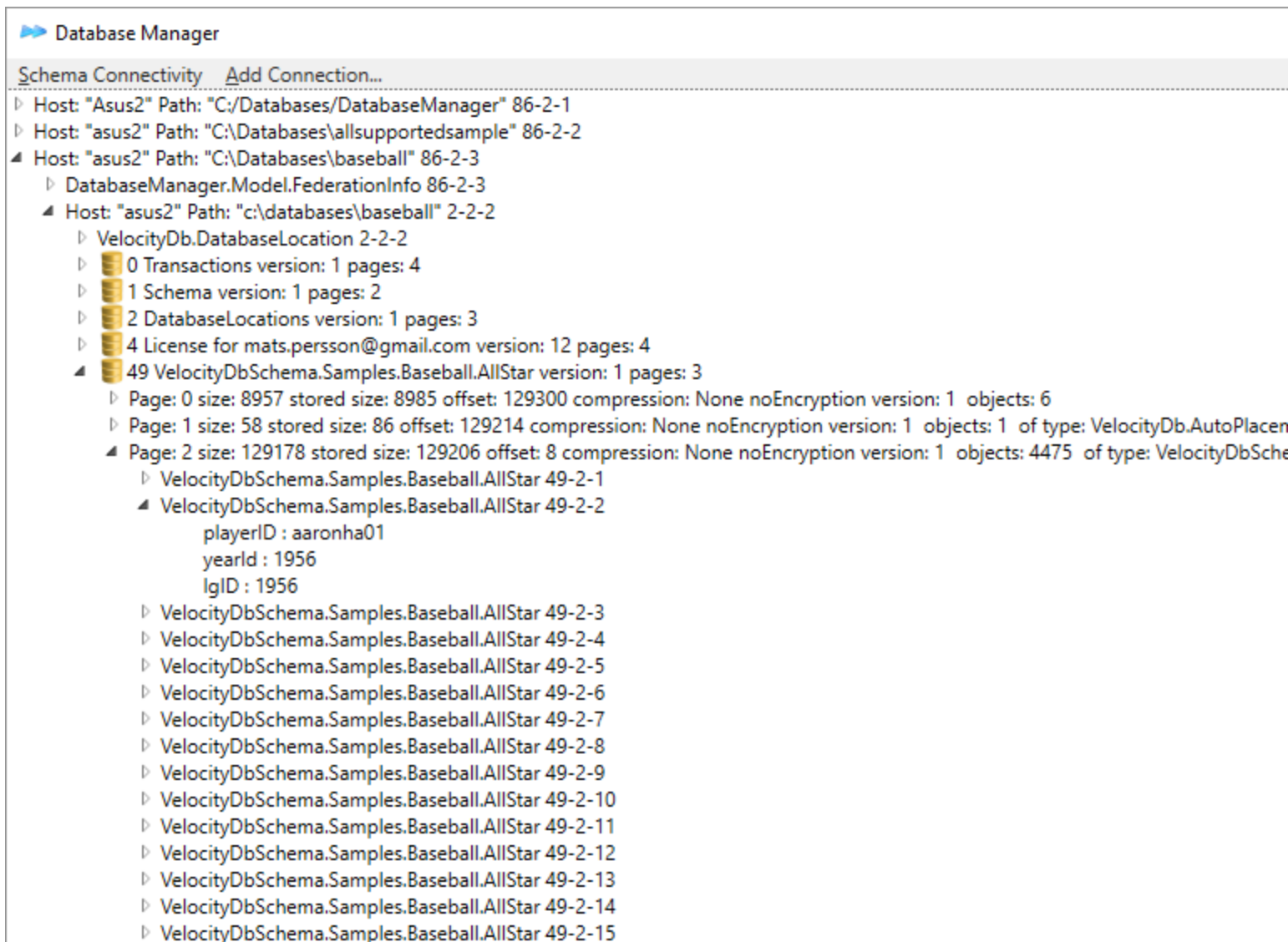
Backup Directory

Host

Database Number

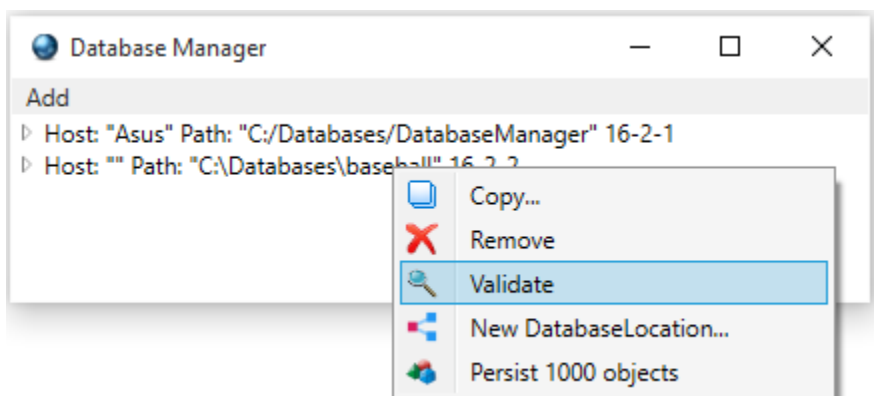
Restore up to

Click Browse... to find the directory of your Baseball databases (build & run this sample first if you have not) then add the VelocityDbSchema.dll to list of classes assemblies and click OK button. Click on arrows to expand.

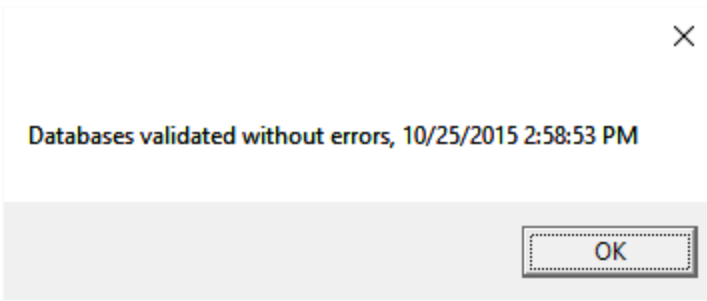


Validating Objects in your databases

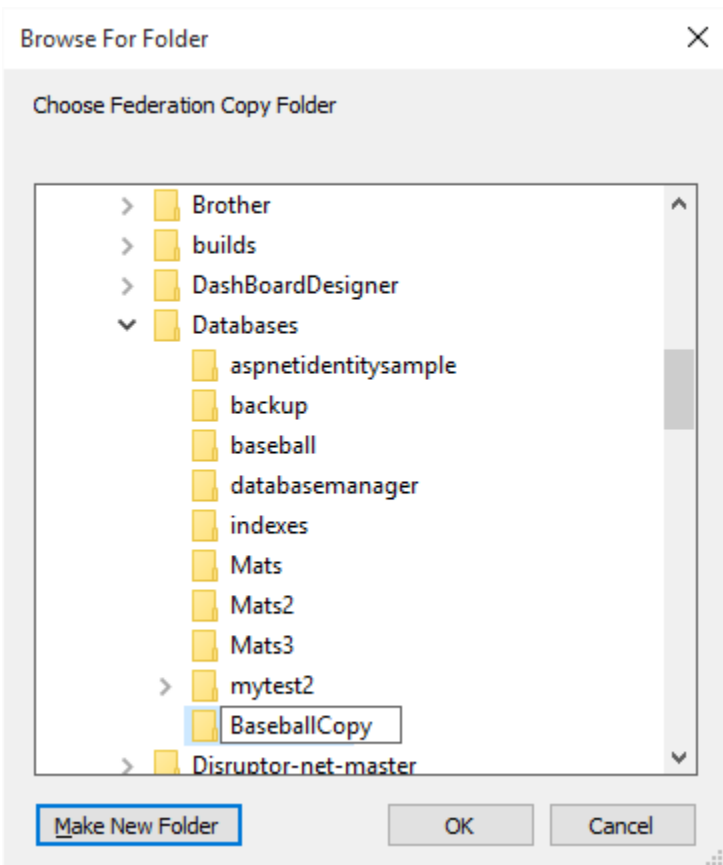
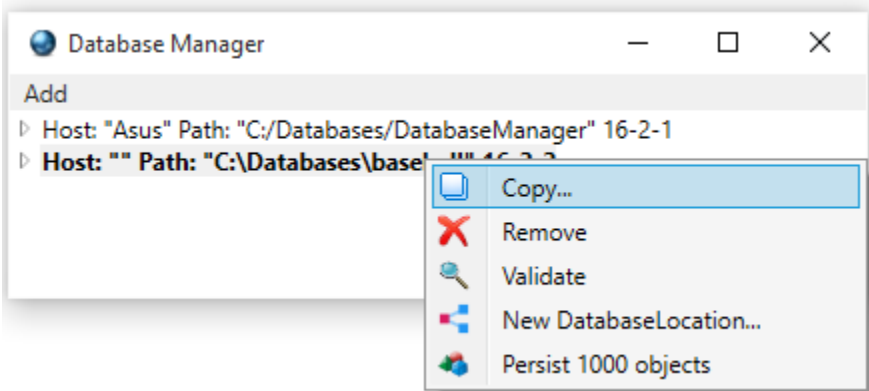
Run `SessionBase.Validate()` on your databases. It checks to make sure that all objects in your databases can be opened without errors.



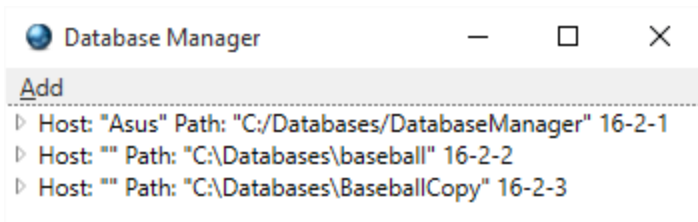
If all is good



Backing up (copy) all your database files

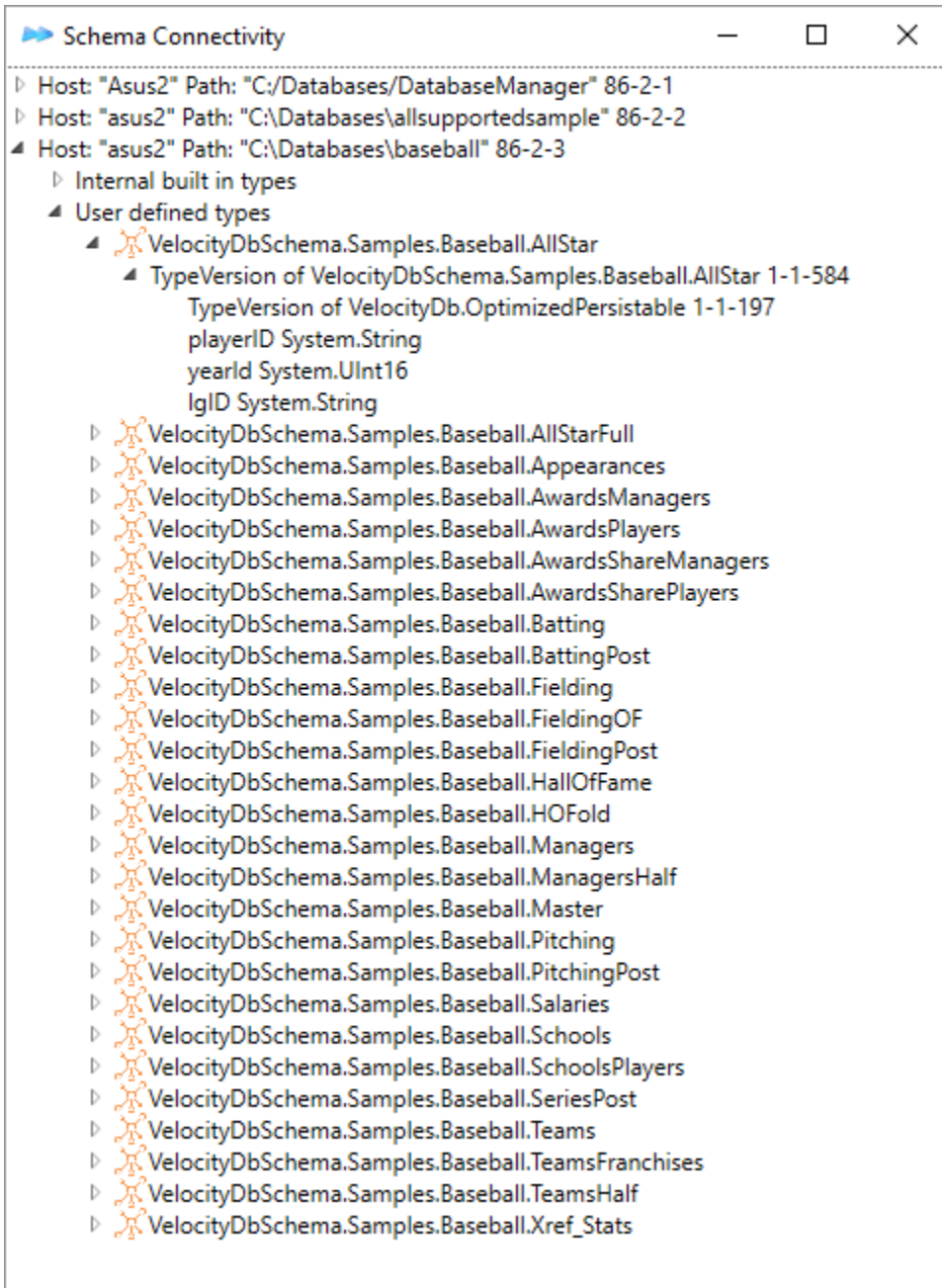


You now have a copy of the Baseball databases in a new folder. You can add this folder to the Database Administrator if you like.

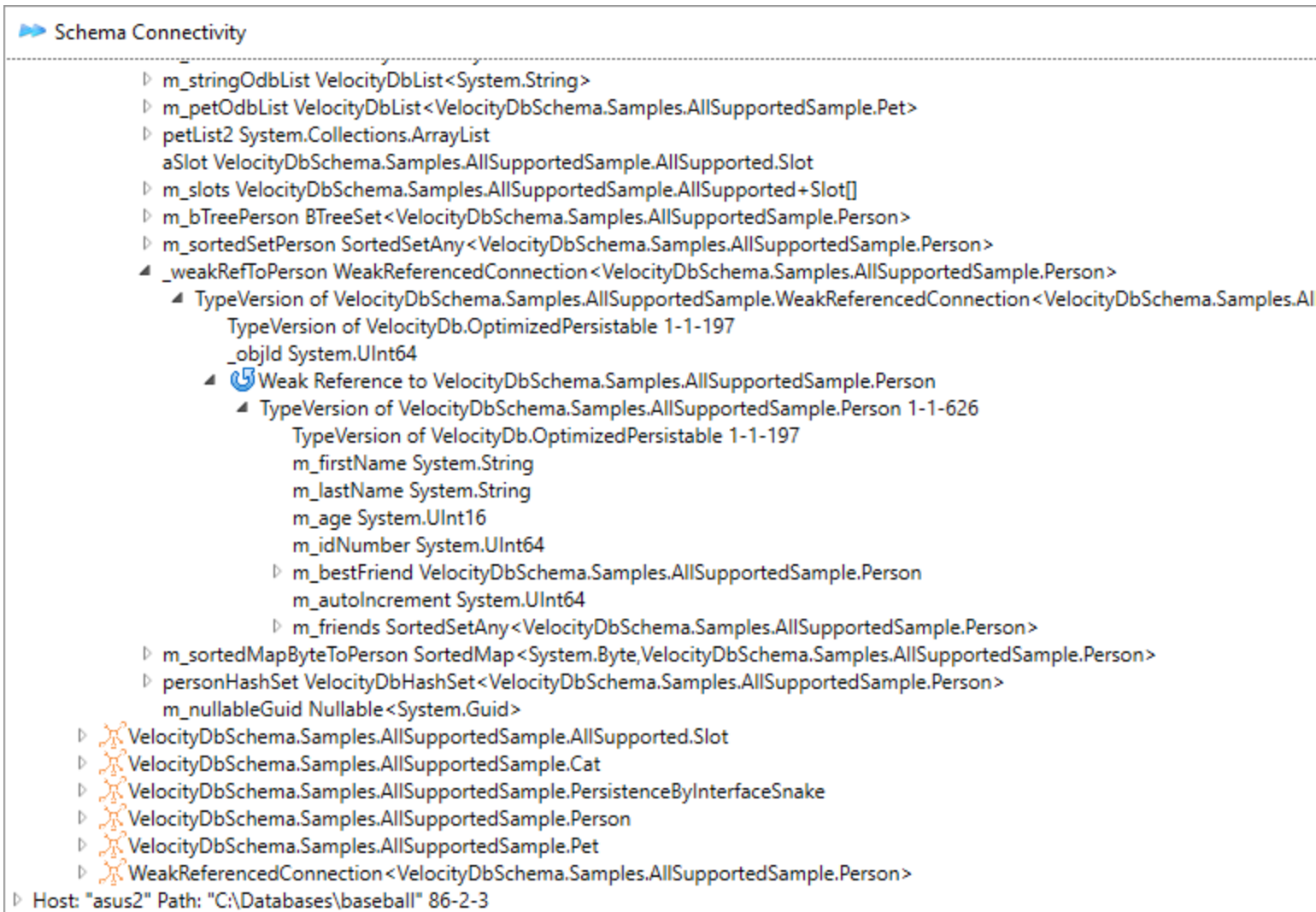


Database Schema Connections

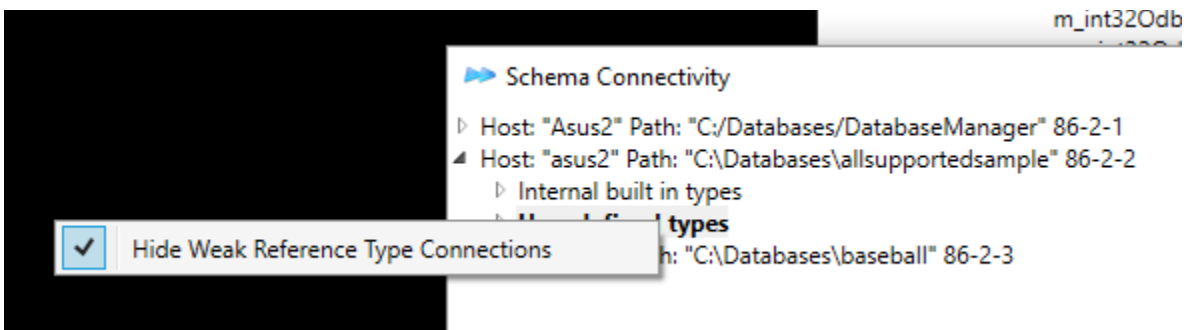
Click on Database Manager menu bar "Schema Connectivity", a second window is opened.



This window shows how the types of your persisted objects are connected via direct (strong) object references and via indirect (weak) references.



It is possible to hide all weak references by right click on "Internal built in types or User defined types.

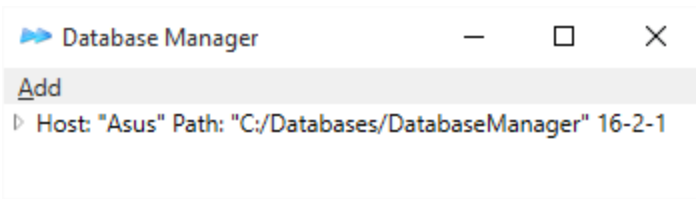


Backup & Restore using Database Manager

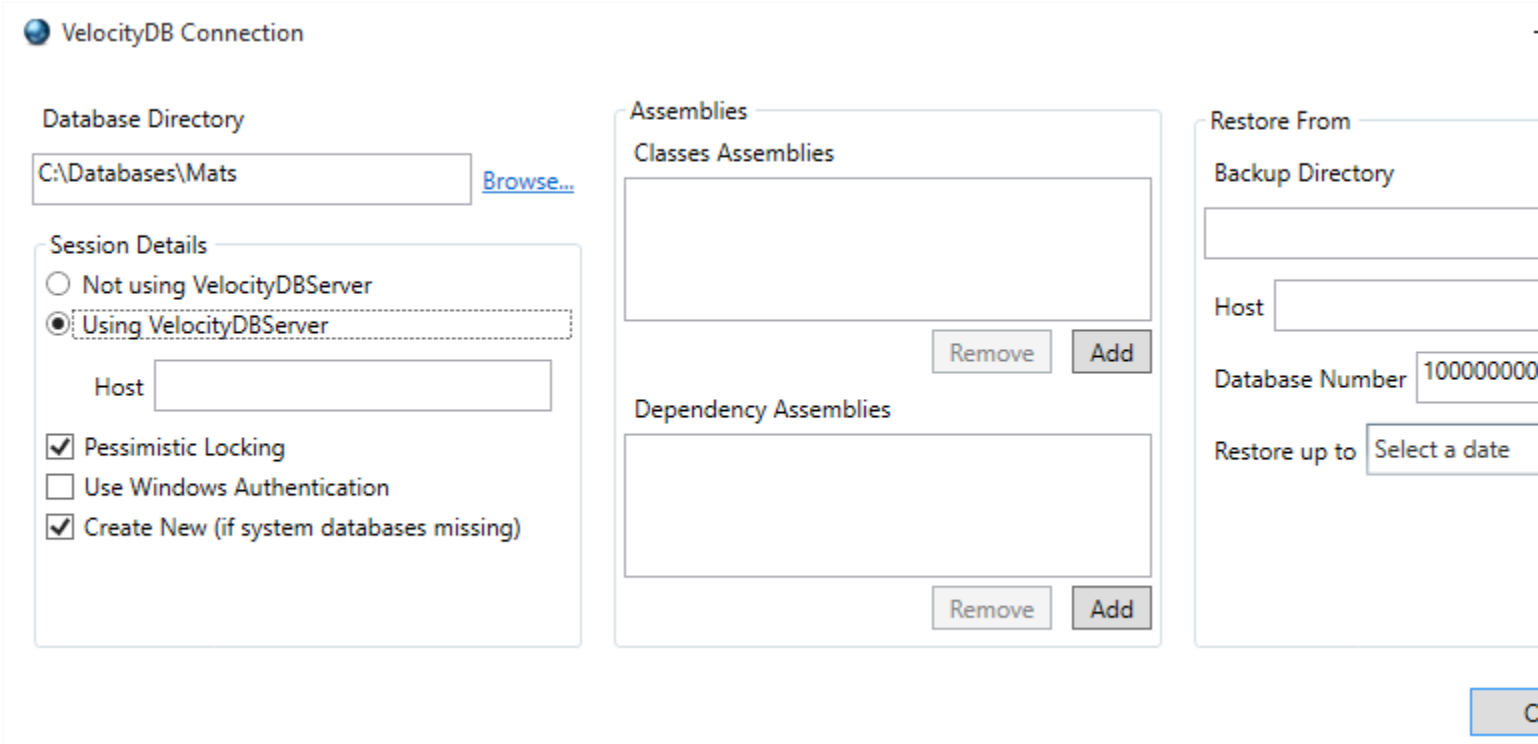
We will go through a simple scenario for this. Using a backup DatabaseLocation is not a one-time backup of your databases. When you create a backup DatabaseLocation a contiguous backup of all changes to the backed up DatabaseLocation starts and continues forever. The backing up is managed by the VelocityDBServer whenever you commit a change. All history of your changes is by default kept in the backup DatabaseLocation.

Create Database

Startup Database Manager



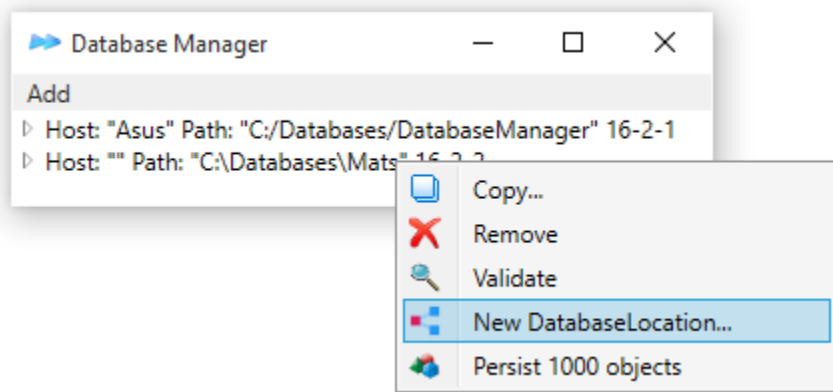
An initial admin database is created. This database contains info about all other databases you "Add" to the Database Manager. Now **click on the "Add" menu item**.



Fill in the requested data and **click on OK**

Create a backup Database Location

Right click on the newly created database and select "New DatabaseLocation..."



New DatabaseLocation [Minimize] [Maximize] [Close]

Directory
 [Browse...](#)

Host

Backup Location
 Is Backup Location
 Backup of

Compression

Encryption
 Type
 Key

Start Database Number
 End Database Number

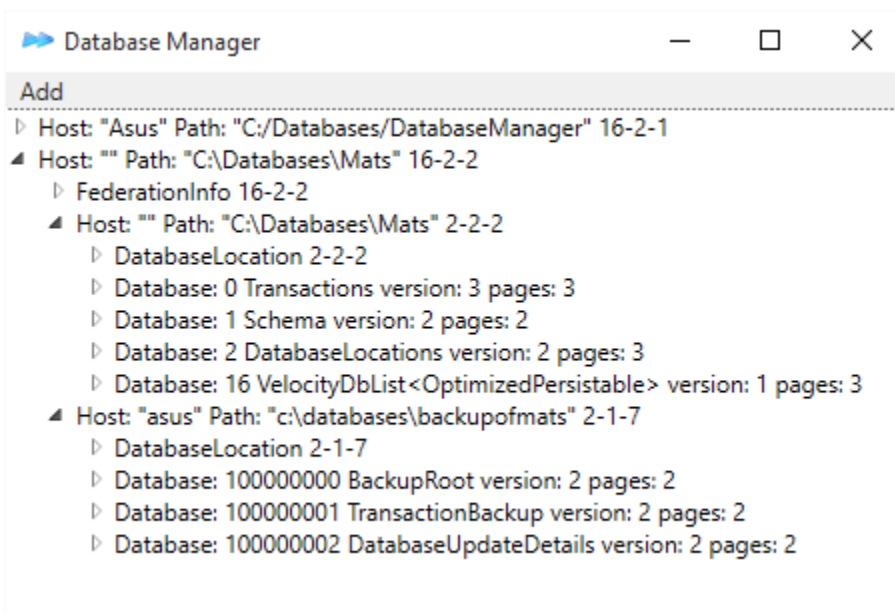
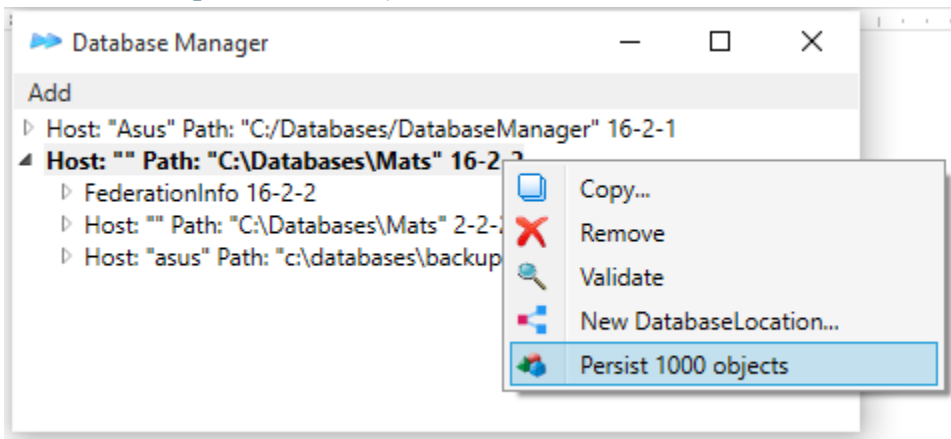
Fill in requested DatabaseLocation data like above and click on OK. Expand to see the new DatabaseLocation.

Database Manager [Minimize] [Maximize] [Close]

A**dd**

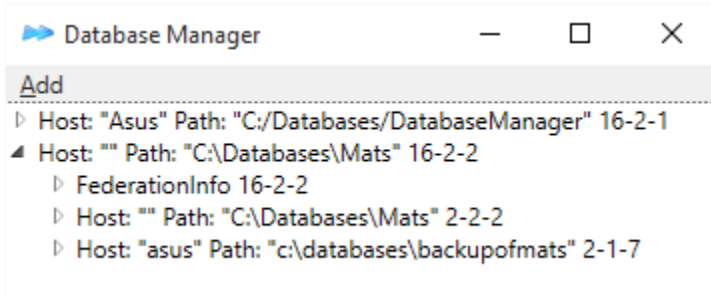
- Host: "Asus" Path: "C:/Databases/DatabaseManager" 16-2-1
- ▲ Host: "" Path: "C:\Databases\Mats" 16-2-2
 - FederationInfo 16-2-2
 - Host: "" Path: "C:\Databases\Mats" 2-2-2
 - Host: "asus" Path: "c:\databases\mats2" 2-1-7

Create some persistent objects

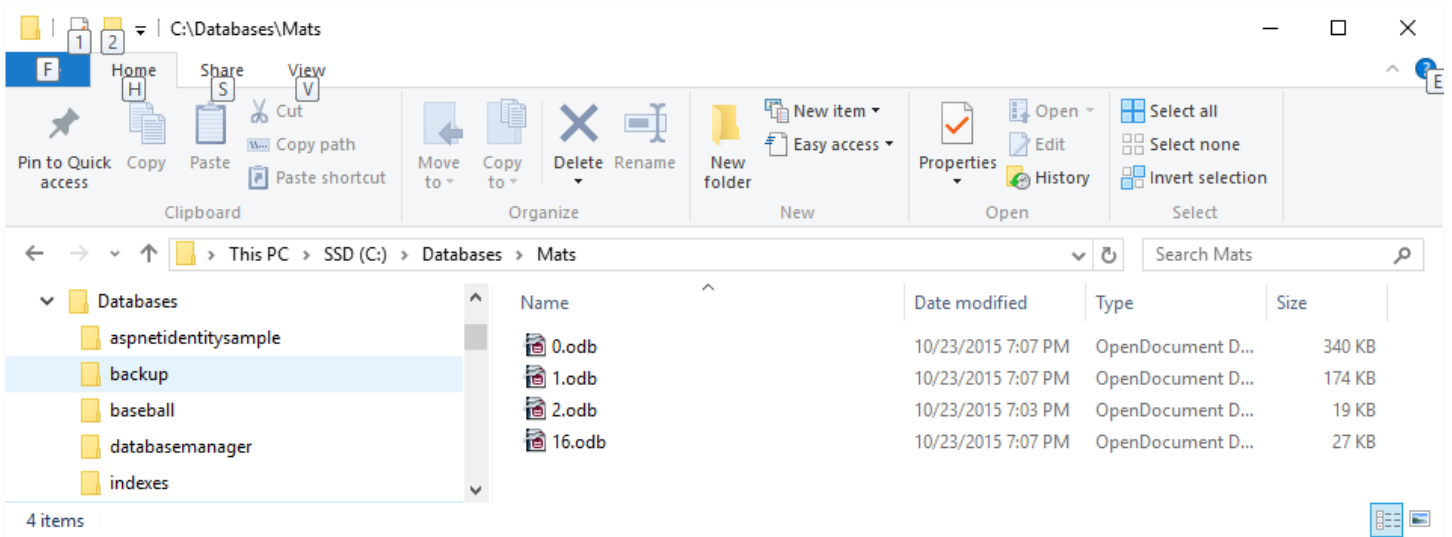


We now have some persistent objects and a backup of all data in original DatabaseLocation.

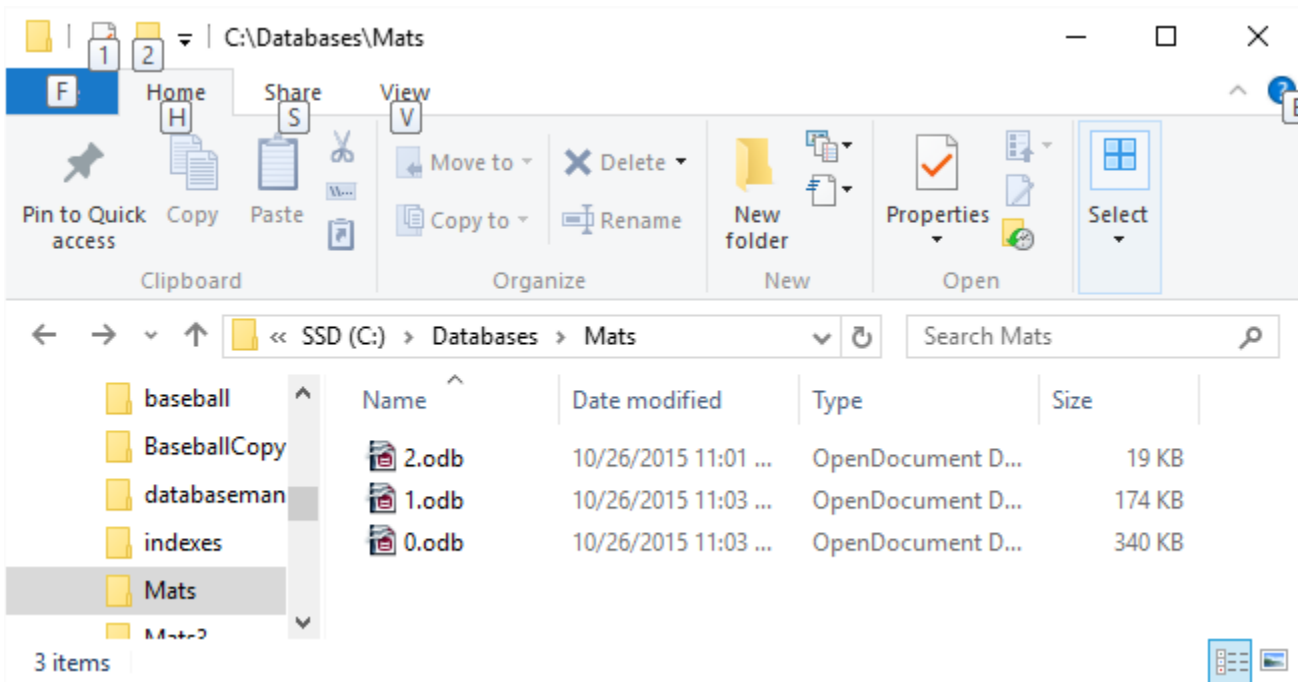
Simulate loosing files in original DatabaseLocation



Manually delete 16.odb in original DatabaseLocation

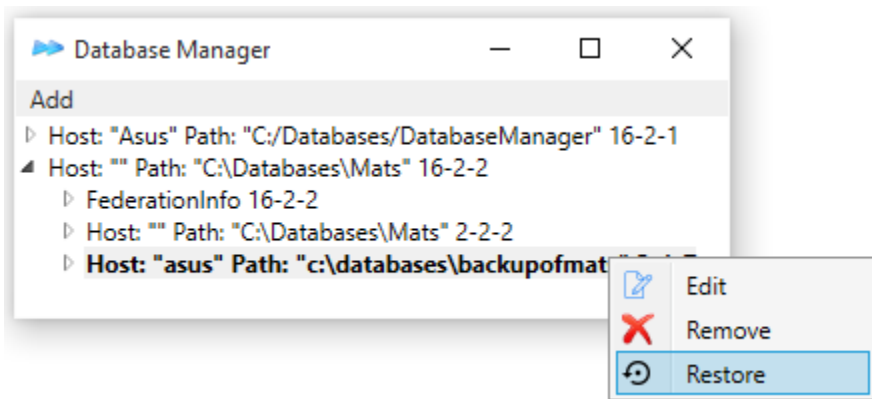


TO (by deleting using file Explorer)



Restore these databases from backup DatabaseLocation

Be sure to expand before deleting the files!



Database Manager

Add

- Host: "Asus" Path: "C:/Databases/DatabaseManager" 16-2-1
- ▲ Host: "" Path: "C:\Databases\Mats" 16-2-2
 - FederationInfo 16-2-2
 - ▲ Host: "" Path: "C:\Databases\Mats" 2-2-2
 - DatabaseLocation 2-2-2
 - Database: 0 Transactions version: 5 pages: 3
 - Database: 1 Schema version: 4 pages: 2
 - Database: 2 DatabaseLocations version: 4 pages: 3
 - ▲ Database: 16 version: 1 pages: 3
 - Page: 0 size: 8916 stored size: 8944 compression: None noEncryption version: 1 objects: 6
 - Page: 1 size: 58 stored size: 86 compression: None noEncryption version: 1 objects: 1 of type: VelocityDb.AutoPlacement
 - Page: 2 size: 18000 stored size: 18028 compression: None noEncryption version: 1 objects: 1000 of type: VelocityDbList<VelocityDb.
 - ▲ Host: "asus" Path: "c:\databases\backupofmats" 2-1-7
 - DatabaseLocation 2-1-7
 - Database: 100000000 BackupRoot version: 4 pages: 2
 - Database: 100000001 TransactionBackup version: 2 pages: 2
 - Database: 100000002 DatabaseUpdateDetails version: 2 pages: 2

Your database file is now restored in your original DatabaseLocation.

Name	Date modified	Type	Size
0.odb	10/26/2015 12:18 ...	OpenDocument D...	349 KB
1.odb	10/26/2015 12:18 ...	OpenDocument D...	270 KB
2.odb	10/26/2015 12:18 ...	OpenDocument D...	27 KB
16.odb	10/26/2015 12:18 ...	OpenDocument D...	27 KB

Restore a backup DatabaseLocation to a brand new directory

A backup DatabaseLocation can be used to create a new set of databases on a new host and directory. Given the backup made in prior section, we will show how to use it to create a new DatabaseLocation in a new directory.

Startup DatabaseManager and click on "Add"

VelocityDB Connection

Database Directory
 [Browse...](#)

Session Details

Not using VelocityDBServer
 Using VelocityDBServer

Host

Pessimistic Locking
 Use Windows Authentication
 Create New (if system databases missing)

Assemblies

Classes Assemblies

Dependency Assemblies

Restore From

Backup Directory

Host

Database Number

Restore up to

Fill in data like above. The above "Database Number" correspond to the first database in the backup DatabaseLocation, by default we set it to 100000000. Click OK.

You now have a brand new DatabaseLocation with all the data backed up in the backup DatabaseLocation.

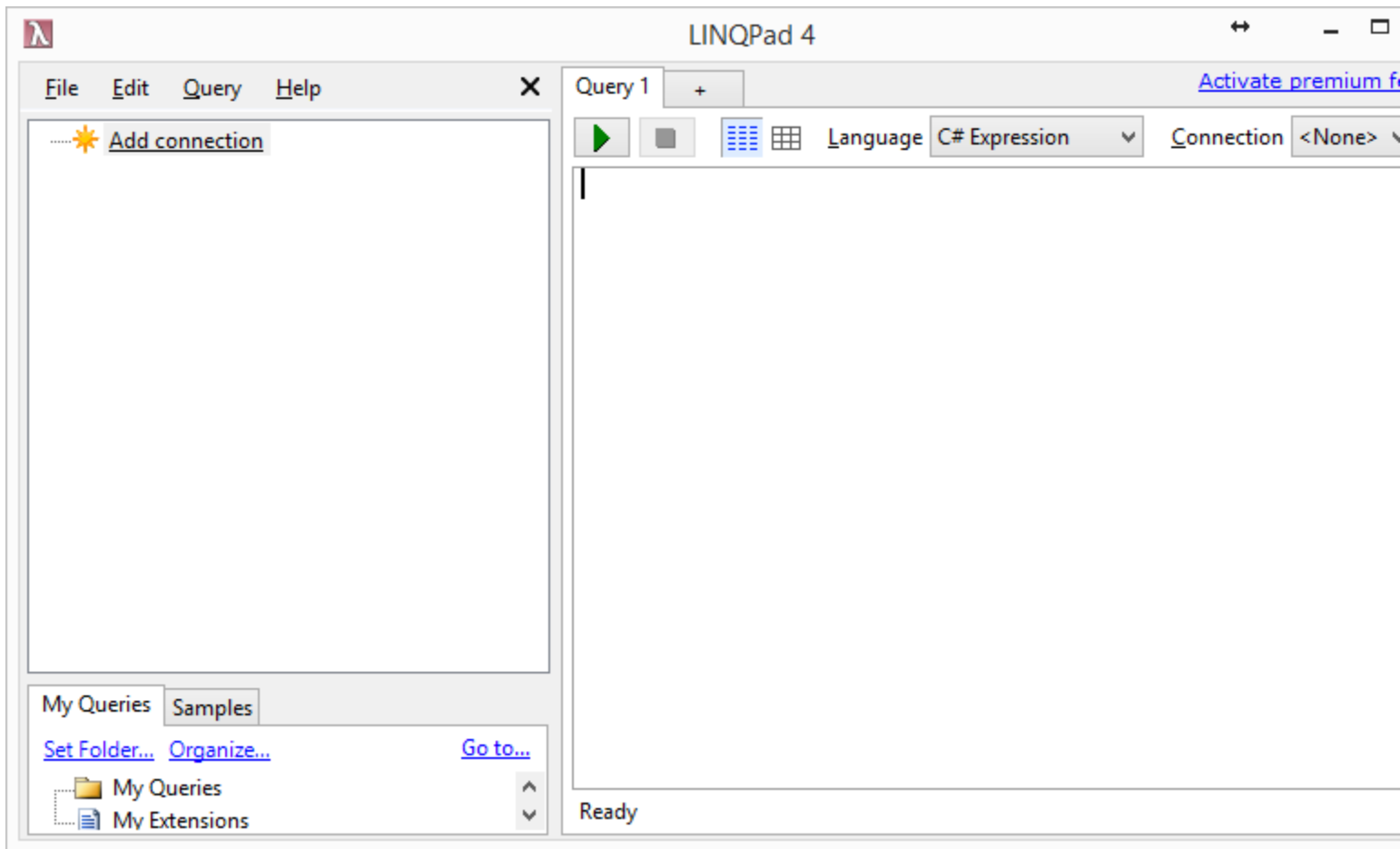
Database Manager

Add

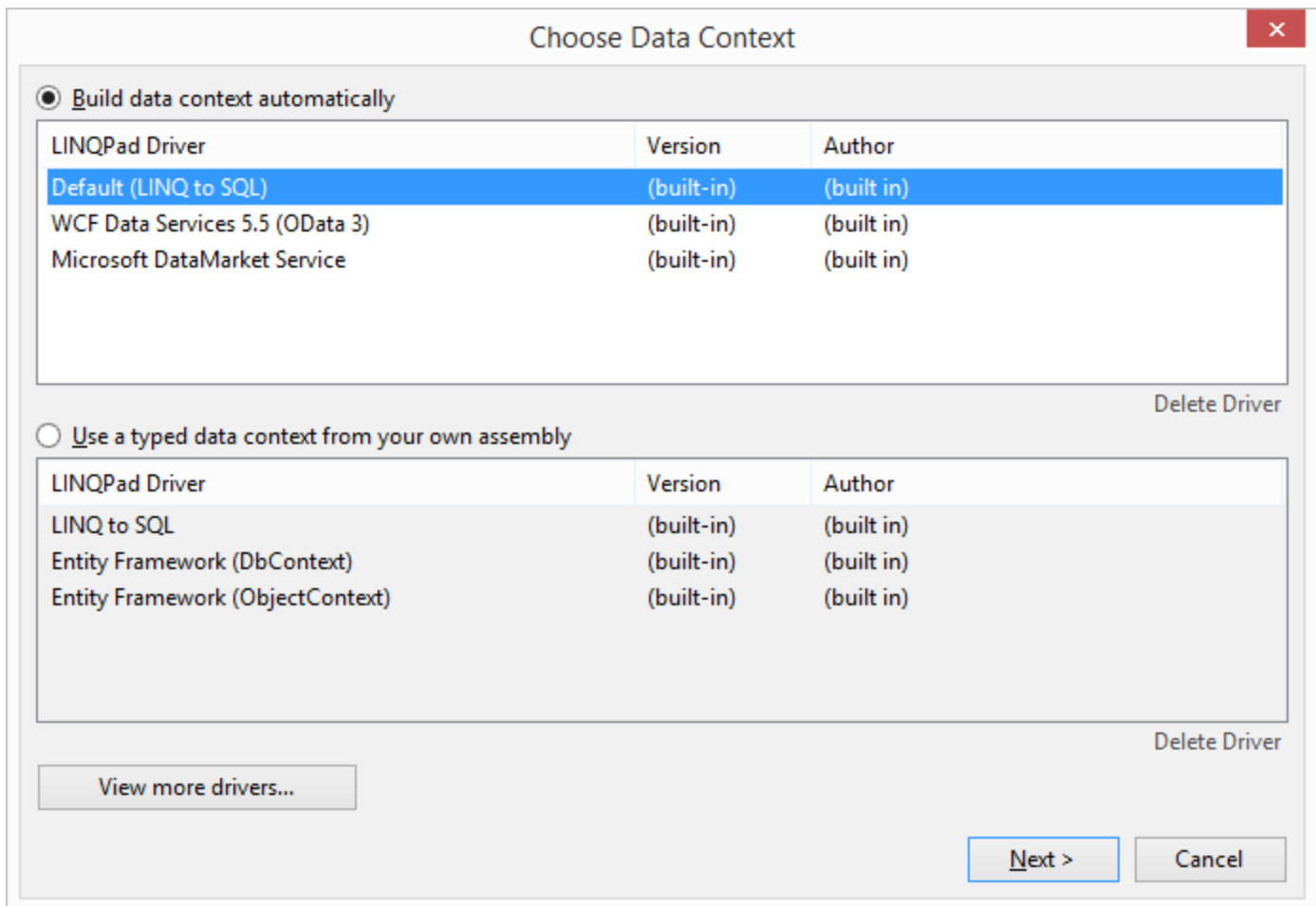
- ▶ Host: "Asus" Path: "C:/Databases/DatabaseManager" 16-2-1
- ▶ Host: "" Path: "C:\Databases\Mats" 16-2-2
- ▲ Host: "" Path: "C:\Databases\Mats3" 16-2-3
 - ▲ FederationInfo 16-2-3
 - m_name : null
 - m_hostName :
 - m_systemDbsPath : C:\Databases\Mats3
 - m_portNumber : 7031
 - m_windowsAuthentication : False
 - m_usesServerClient : False
 - m_usePessimisticLocking : False
 - m_validated List<DateTime> size: 0
 - ▶ m_typesAssemblies System.String[] size: 0
 - ▶ m_typesDependencyAssemblies System.String[] size: 0
 - m_federationCopies List<FederationCopyInfo> size: 0
 - ▲ Host: "asus" Path: "c:\databases\mats3" 2-2-2
 - ▶ DatabaseLocation 2-2-2
 - ▶ Database: 0 Transactions version: 1 pages: 3
 - ▶ Database: 1 Schema version: 1 pages: 2
 - ▶ Database: 2 DatabaseLocations version: 1 pages: 3
 - ▲ Database: 16 VelocityDbList<OptimizedPersistable> version: 1 pages: 3
 - ▶ Page: 0 size: 8952 stored size: 8980 compression: None noEncryption version: 1 objects: 6
 - ▶ Page: 1 size: 58 stored size: 86 compression: None noEncryption version: 1 objects: 1 of type: VelocityDb.AutoPlacer
 - ▶ Page: 2 size: 18000 stored size: 18028 compression: None noEncryption version: 1 objects: 1000 of type: VelocityDb

Using LINQPad to make VelocityDB LINQ queries/browsing

Here is how to set it up. Start by downloading and installing LINQPad from <http://www.linqpad.net/>. Start it. It should look like this:



Click on "Add connection", takes you to this:



Click on “View more drivers...”, takes you to this:

Choose a Driver

Choose from the featured drivers:

LINQPad Supplementary Data Context Drivers

Blocked by a proxy or firewall? [Click here](#) to download these drivers from a web browser.

IQ Driver - for MySQL, SQLite, Oracle

by Joe Albahari, Matt Warren, WICKY Hu **Version 2.1.1**

▼ [Download & Enable Driver](#) (download again to update the driver)

This LINQPad driver uses [Matt Warren's IQueryable toolkit](#) and supports MySQL, SQLite and Oracle. Everything you need is included and you can be querying within seconds: no extra drivers or providers are required (you don't even have to install the Oracle client!) Querying functionality is (almost) on par with LINQ to SQL, and updates are supported (although not through associations). Plain SQL queries are supported, too. This driver does not alter your machine configuration in any way (nor does it install anything into the GAC).

Matt Warren's IQueryable Toolkit © Microsoft Corporation (used under [Ms-PL license](#)). Oracle IQ Provider by [WICKY Hu](#) ([BSD License](#)). ADO.NET providers for MySQL and Oracle by [DevArt](#).

Microsoft StreamInsight Driver

By downloading the Microsoft StreamInsight Driver for Linqpad you agree to the terms of the [Microsoft Software License Terms](#).

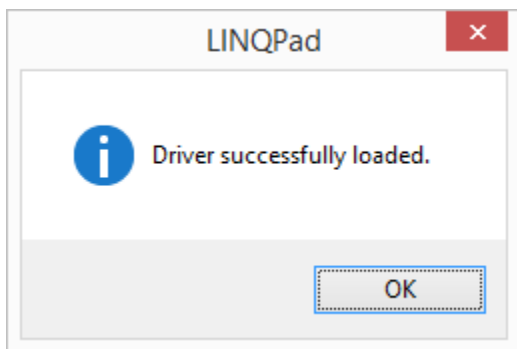
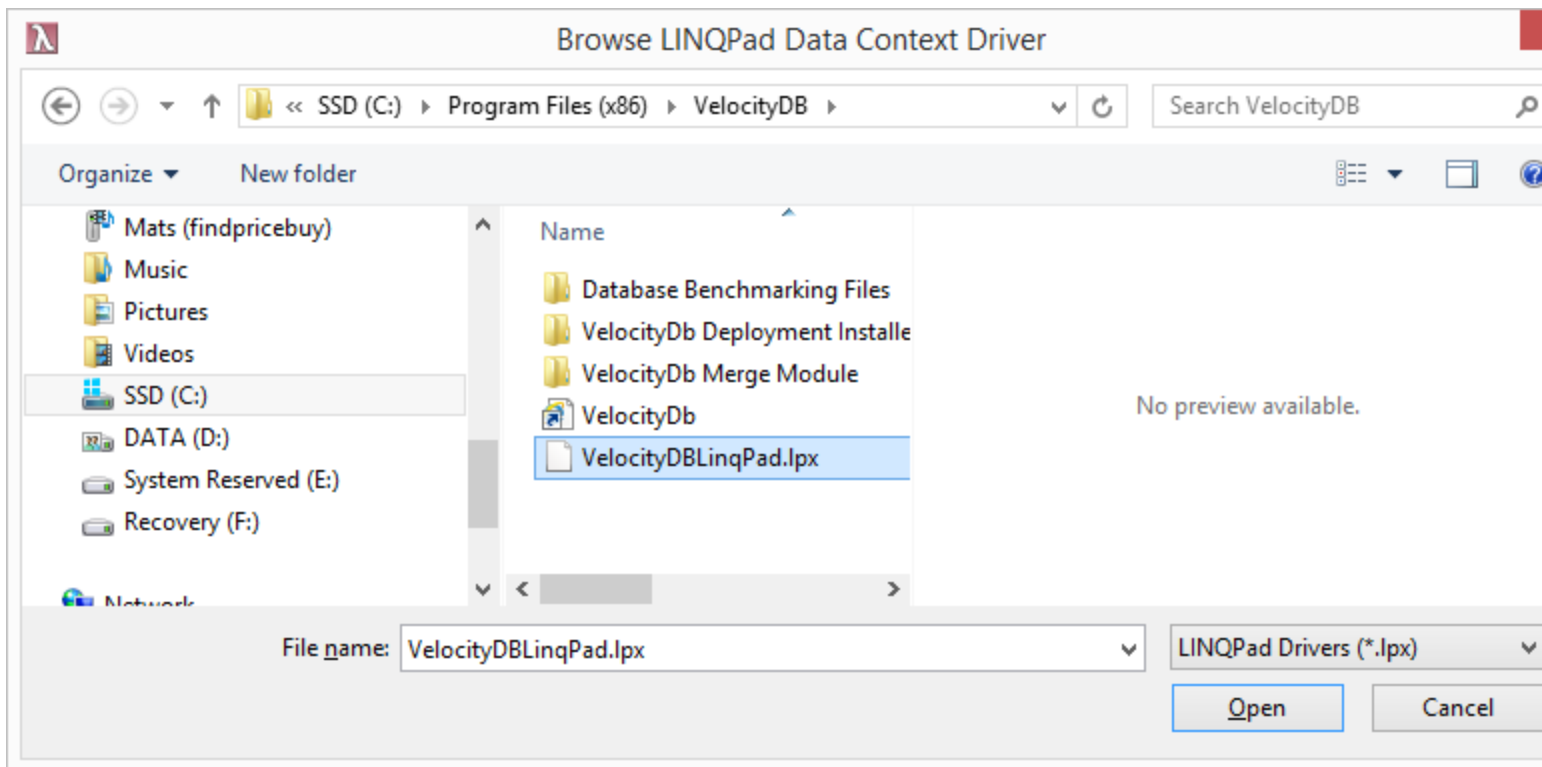
▼ [Download Driver for StreamInsight 2.1](#) Version 0.9.3 (download again to update)

▼ [Download Driver for StreamInsight 2.0](#) Version 0.9.2 (download again to update)

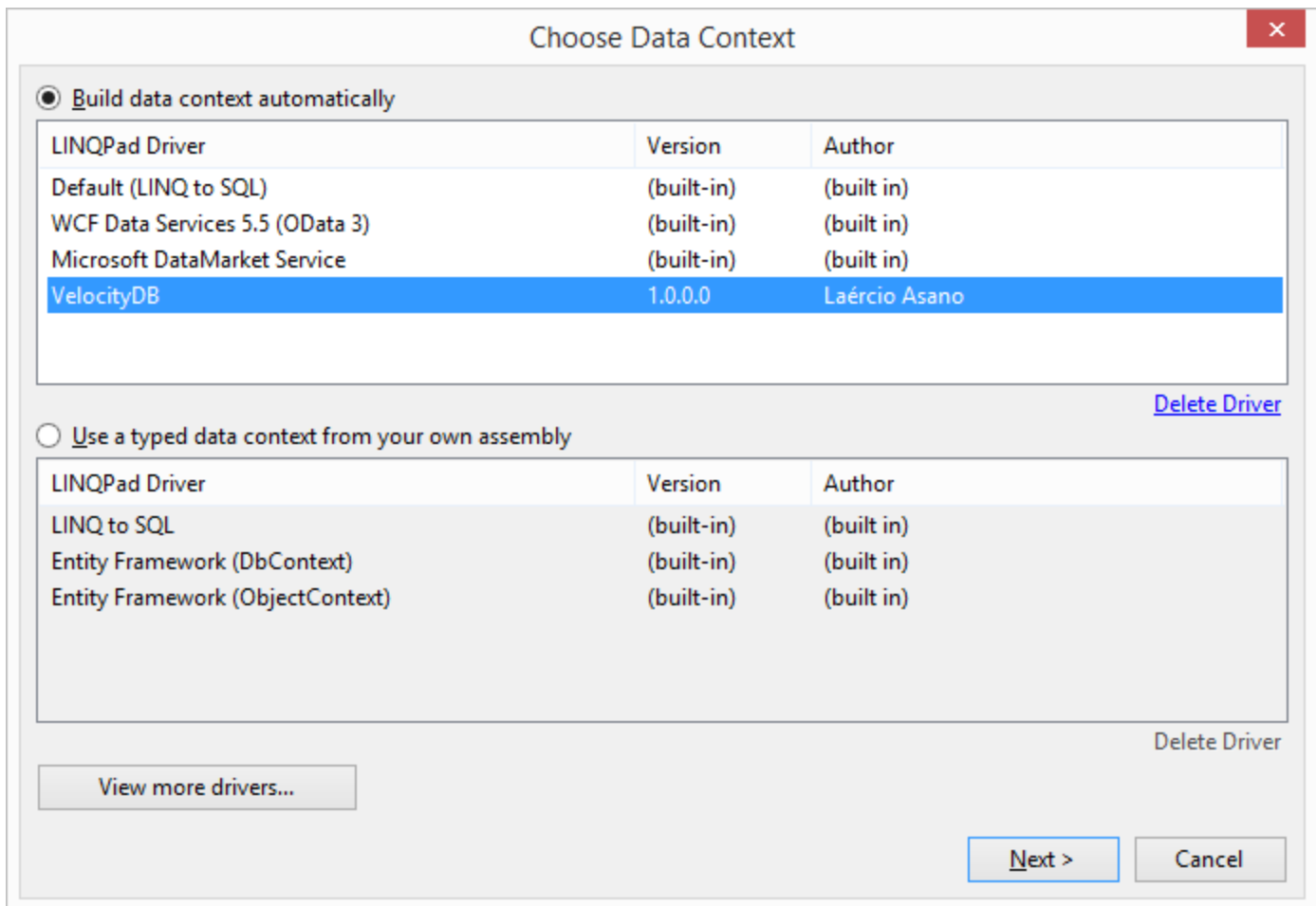
▼ [Download Driver for StreamInsight 1.1 / 1.2](#) (download again to update)

Or, browse to a .LPX file:

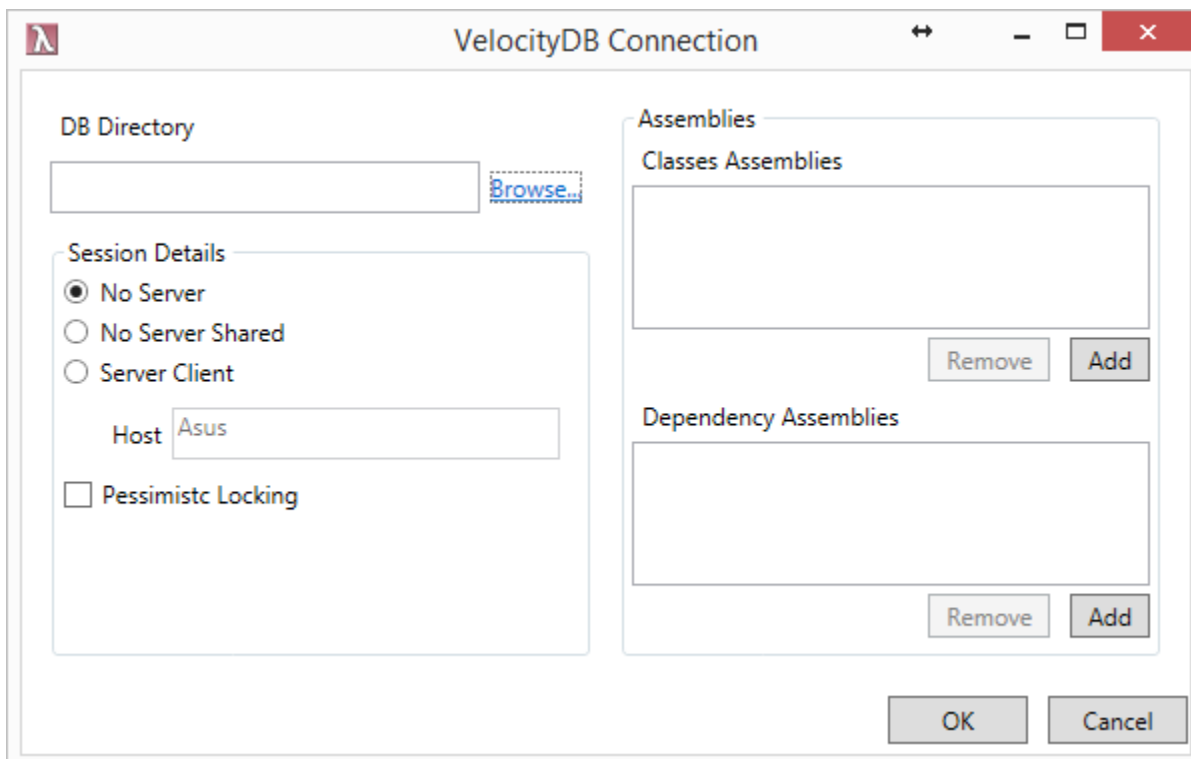
Click on "Browse...", select the file VelocityDBLinqPad.lpx from your VelocityDB installation directory



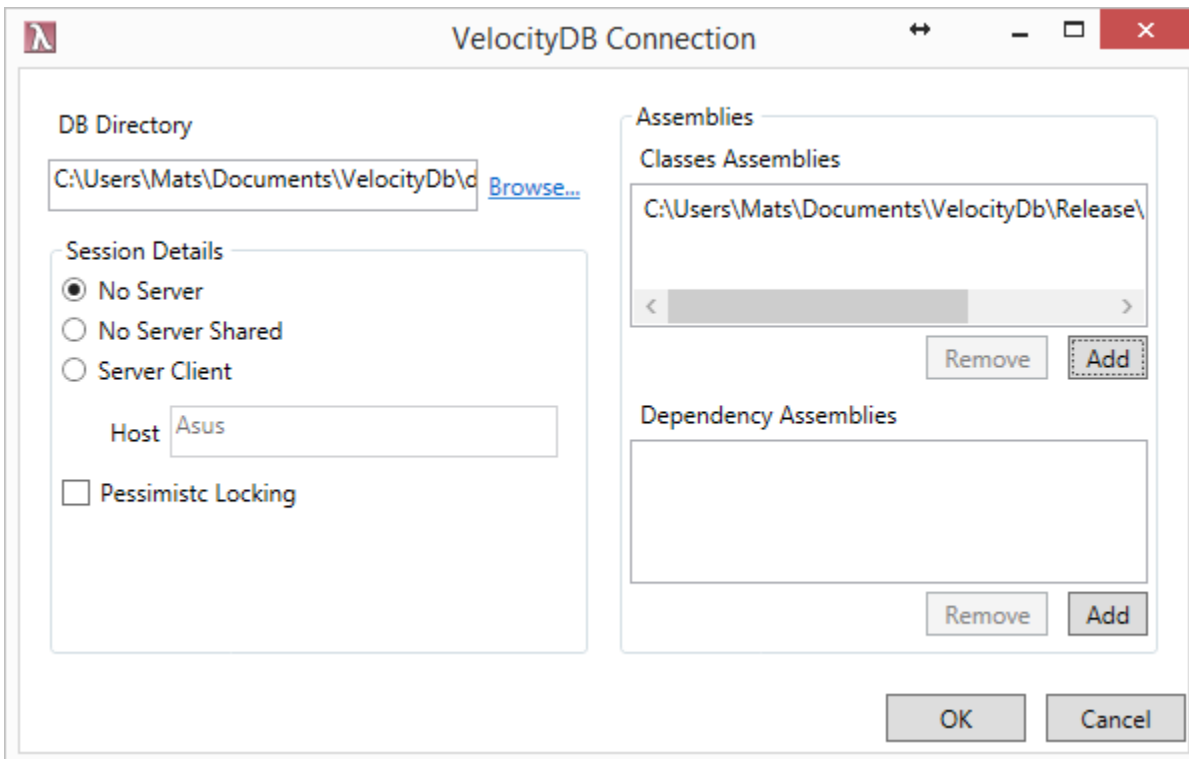
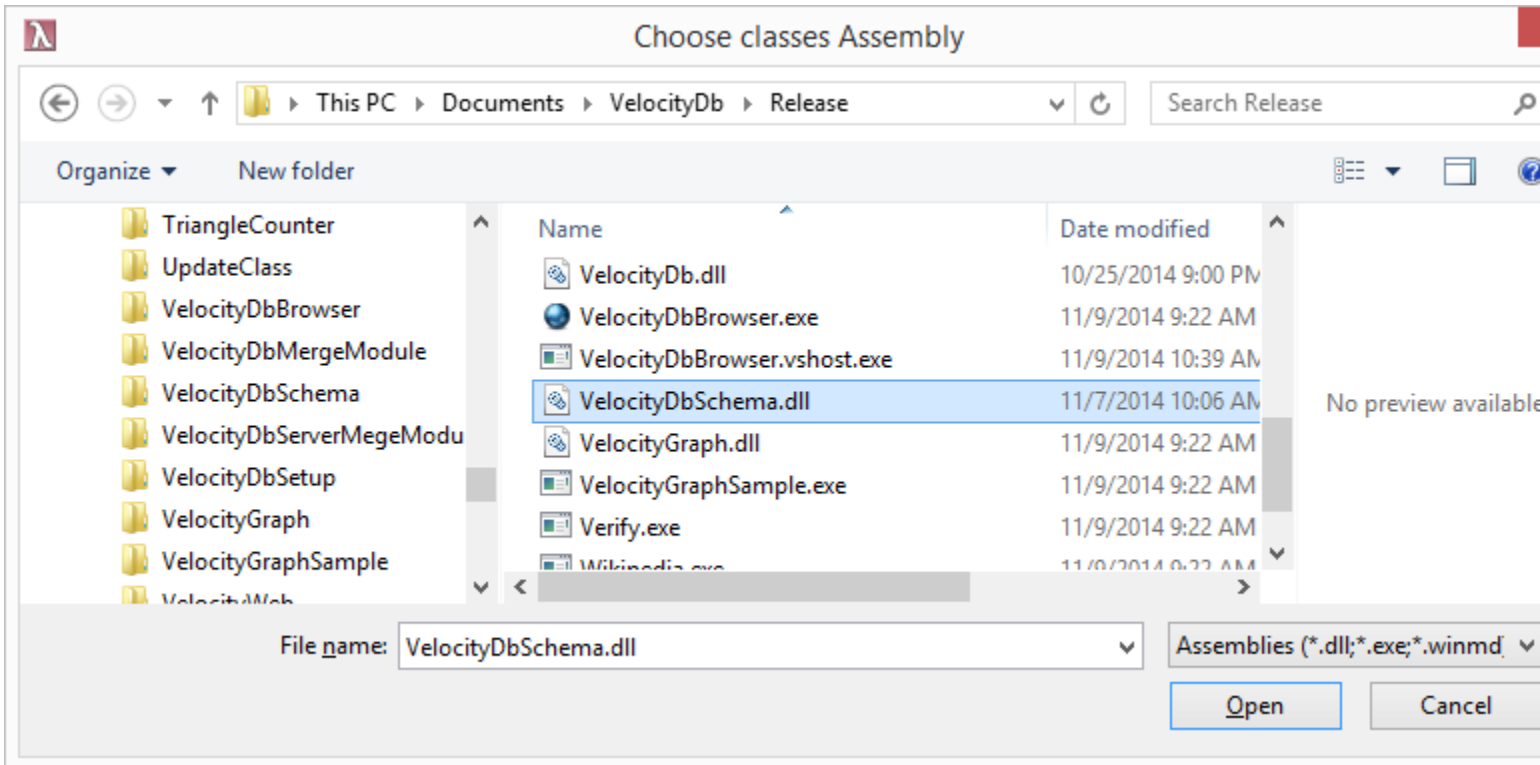
Then select the VelocityDB data context and click on “Next >”



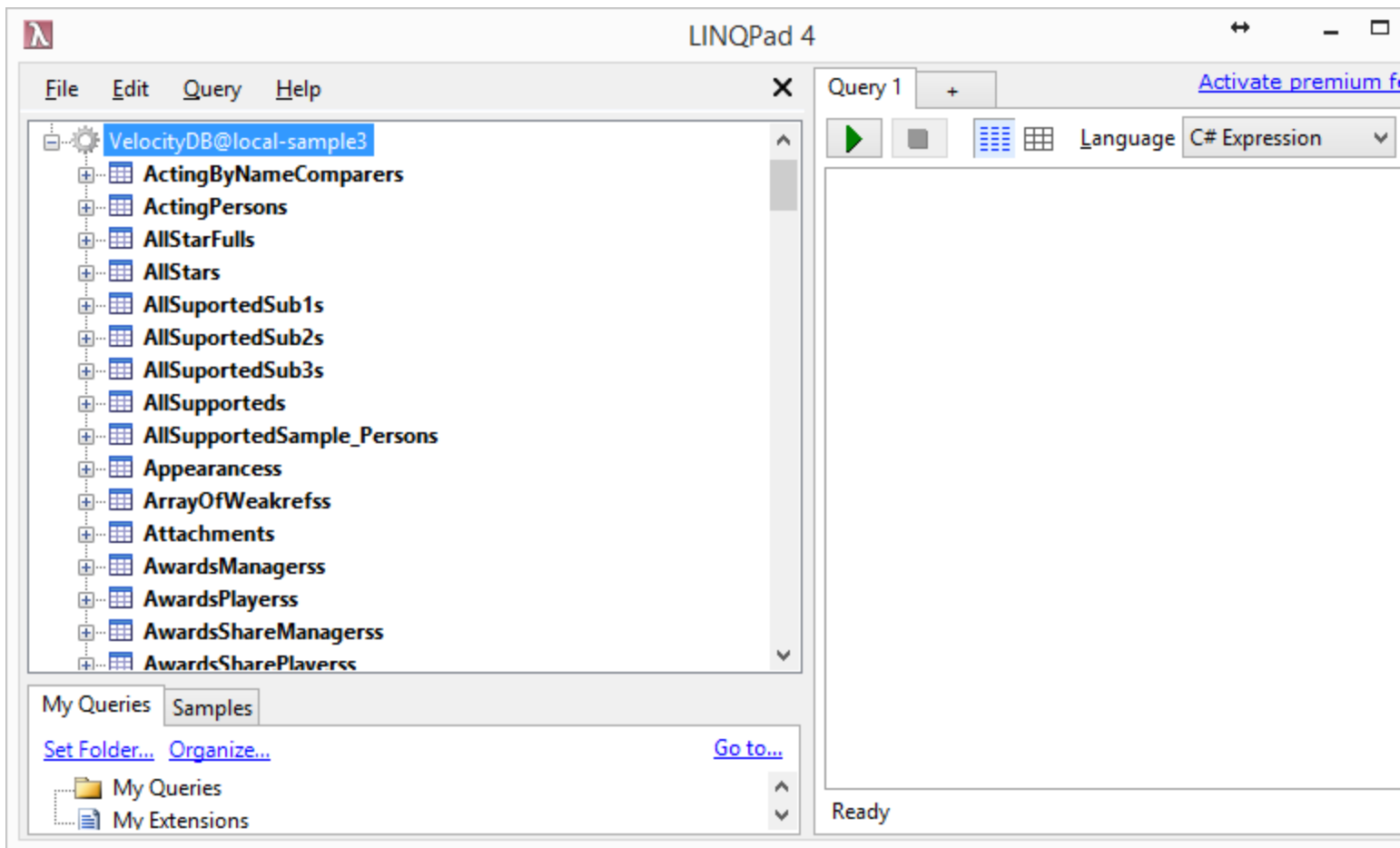
You should now see:



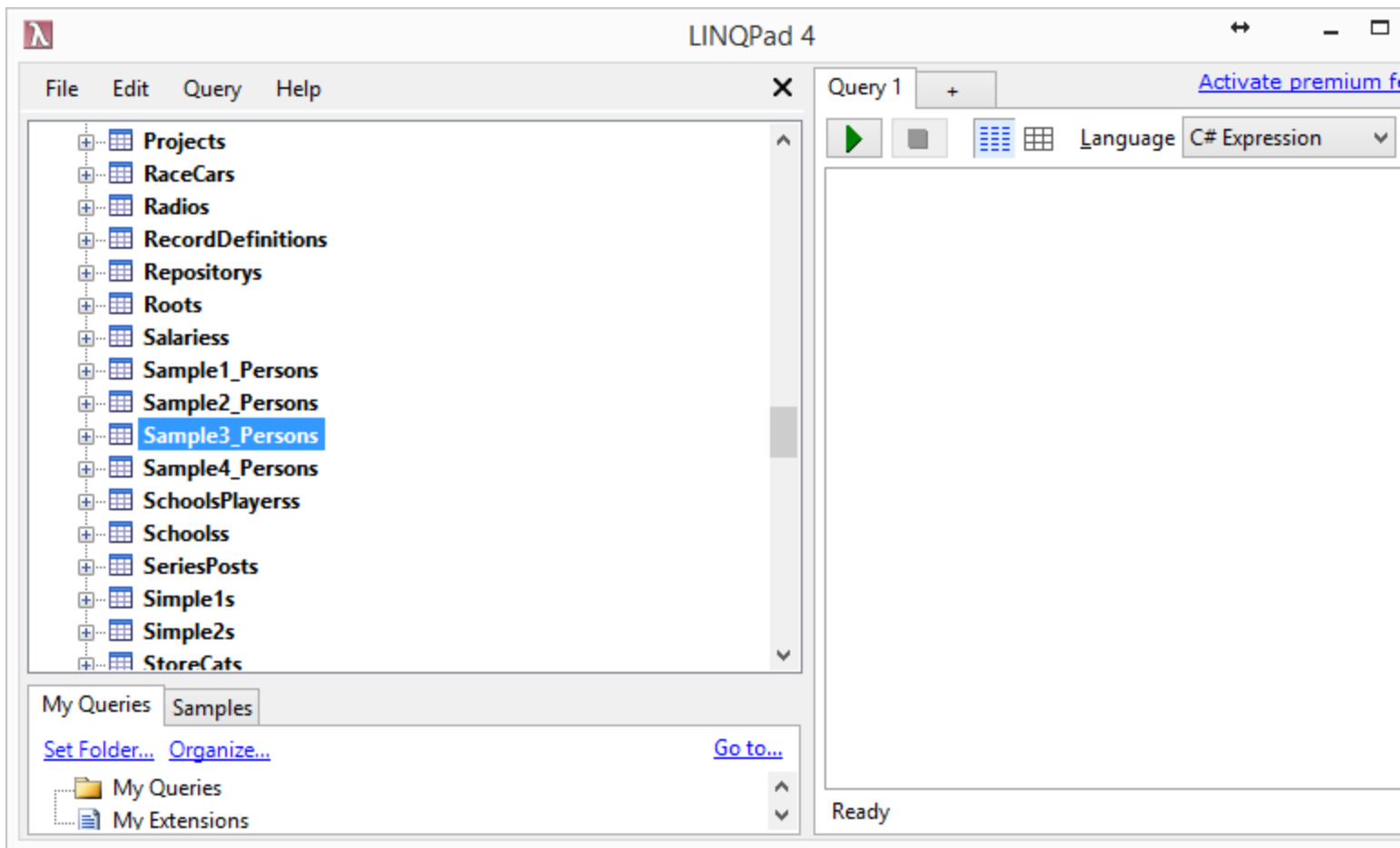
Choose DB Directory and Assemblies. Choose the assembly where your persisted classes are defined. If these are defined in an .exe file you may have to move them to a library project instead and reference it from your .exe.



Then press "OK", should take you to this (in this case using Sample3 database directory from VelocityDB.sln samples)



Scroll down to "Sample3_Persons", select and right click with mouse, choose "Sample3_Persons.Take(100)"



Result should now show as:

File Edit Query Help

- Projects
- RaceCars
- Radios
- RecordDefinitions
- Repositories
- Roots
- Salariess
- Sample1_Persons
- Sample2_Persons
- Sample3_Persons
- Sample4_Persons
- SchoolsPlayerss
- Schoolss
- SeriesPosts
- Simple1s
- Simple2s
- StoreCats
- StoreLists
- StoreStructs
- StringRecord2s
- StringRecords
- StringTests
- SubTasks
- Tables
- TeamsFranchisess
- TeamsHalfs
- Teamss
- TestCases
- TestClasss
- TestRecords
- TestRecs
- TickOptimizeds
- Ticks
- Trucks

My Queries Samples

[Set Folder...](#) [Organize...](#)[Go to...](#)

- My Queries
- My Extensions

Query 1 Query 2 +

Language C# Expression

Sample3_Persons.Take (100)

Results λ SQL IL

IEnumerable<Person> (3 items)

BestFriend

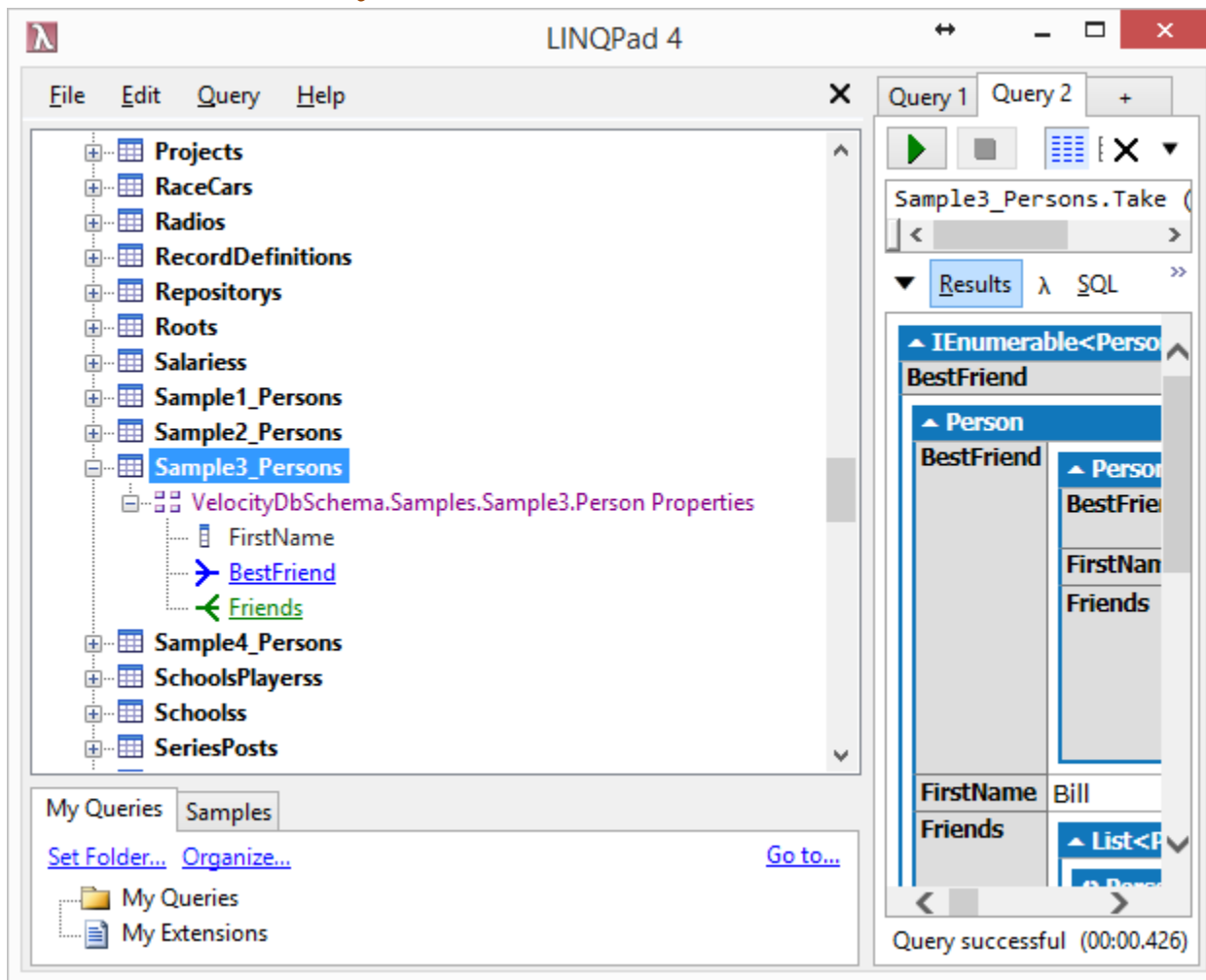
Person	
BestFriend	Person
FirstName	Robin
Friends	List<Person> (1 item)
Person	Person
FirstName	Bill
Friends	List<Person> (1 item)
Person	Person

Person	
BestFriend	Person
FirstName	Robin
Friends	List<Person> (1 item)
BestFriend	Person
FirstName	Steve
Friends	List<Person>

Person	
BestFriend	Person
FirstName	Bill
Friends	List<Person> (1 item)
Person	Person

Query successful (00:00.426)

Issues with current LINQPad driver



Proper class names are not displayed, above "Sample3_Persons" should really be 'VelocityDbSchema.Samples.Sample3.Person' as shown when you expand to see properties. Also objects and classes of template classes are not included. We'll try to resolve these issues as soon as possible but it's tricky due to using properties to expose each class and property names cannot have the characters "<.>" in them.

Controlling the in memory page and object caching

By default VelocityDB tries to cache database pages whenever there is enough available RAM memory. You can control how much enough RAM memory is by API on the `DataCache` object that is accessed from a session object by the property `ClientCache`. You can also completely turn off page caching by specifying this as one of the optional parameters when creating a session. Object caching is also supported, see how to [here](#).

Verifying all objects and references

The `Verify.exe` application provided in the sample solution can be used to verify your data. Run `Verify.exe` and specify as command line parameter the directory where your databases are located. `Verify.exe` walks through all objects and opens all their references and it iterates through all enumeration types such as `BTreeSet` and other collections. An exception will be thrown if a failure is found. You can also verify all objects by API using `SessionBase.Verify()`.

Scalability

A single session can manage uncompressed data at a maximum size of a half trillion terabytes (half a yottabyte). To reach that maximum size you need 4 billion databases (.odb files) with 65 thousand pages in each and each page size near 2 GB. An application can simultaneously use multiple sessions so total data size is unlimited.

- 2 GB is maximum size for a page. Limit is due to .Net [2GB limitation](#) of byte[].

Given this 2GB size limitation, it is not possible to persist objects such as [Dictionary<TKey, TValue>](#) that are larger than 2 GB. However, our BTree and BTreeMap collections can be used because they are composite objects where each object is smaller than 2GB no matter how large the total size of the collection (or map).

Database backup and restore

Database backup is an option on each [DatabaseLocation](#), you can request that all databases of a specified [DatabaseLocation](#) are backed up to a backup [DatabaseLocation](#). This API is currently only supported with [ServerClientSession](#).

Backup

The following code create a backup [DatabaseLocation](#) for the default [DatabaseLocation](#) (the one containing the system database 0, 1, 2, and 4)

```
using (ServerClientSession session = new ServerClientSession(systemDir, Dns.GetHostName()))
{
    const bool isBackupLocation = true;
    session.BeginUpdate();
    DatabaseLocation backupLocation = new DatabaseLocation(Dns.GetHostName(),
        "c:/NUnitTestDbsBackup",
        (uint)Math.Pow(2, 24),
        UInt32.MaxValue,
        session,
        false,
        PageInfo.encryptionKind.noEncryption,
        isBackupLocation,
        session.DatabaseLocations.Default());
    session.NewLocation(backupLocation);
    session.Commit();
}
```

From now on, every time a default [DatabaseLocation](#) database is created/updated, it will be backed up to the backup [DatabaseLocation](#).

Restore

The following code restores the default [DatabaseLocation](#) from its backup.

```
using (SessionNoServer session = new SessionNoServer(systemDir))
{
    session.BeginUpdate();
    DatabaseLocation backupLocation = new DatabaseLocation(Dns.GetHostName(), "c:/NUnitTestDbsBackup",
        (uint)Math.Pow(2, 24), UInt32.MaxValue, session,
        false, PageInfo.encryptionKind.noEncryption, true, session.DatabaseLocations.Default());
    session.RestoreFrom(backupLocation, DateTime.Now);
    session.Commit(false, true);
}
```

CopyAllDatabasesTo

A fast and easy way to backup your databases is to use [SessionBase.CopyAllDatabasesTo](#), as in

```
using (ServerClientSession session = new ServerClientSession(systemDir))
{
    session.CopyAllDatabasesTo(copyDbsDir);
}
```

```
using (SessionNoServer session = new SessionNoServer(copyDbsDir))
{
    session.BeginRead();
    session.Verify();
    session.Commit();
}
```

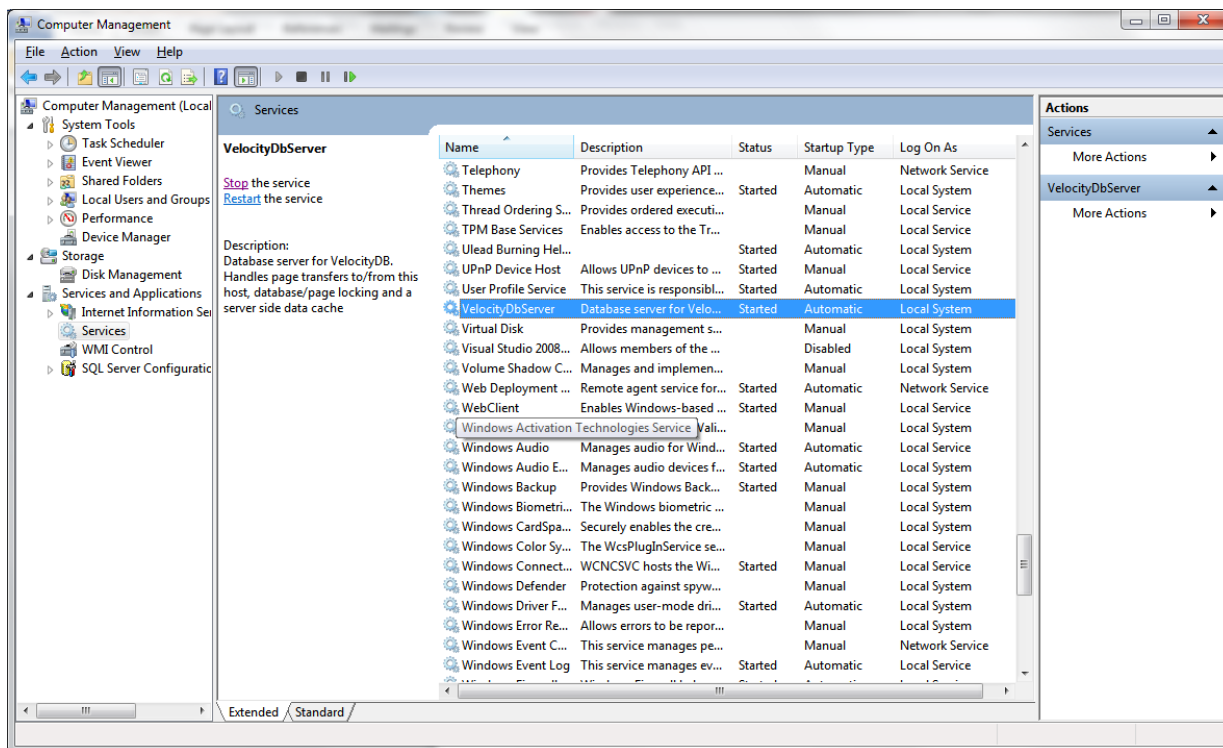
ExportToCSV and ImportFromCsv

SessionBase provides a to and from CSV file option. The CSV export files contains one csv file for each Type stored in the databases.

VelocityDbServer.exe

This is a server process that manages data transfer between client and server hosts. It also handles the page/database locking and manages a shared cache. The use of this server process is optional but is required in order to distribute databases and the server is also required when page level locking is requested.

VelocityDbServer.exe is installed as a service unless you did the install choosing VelocityDbNoServer.exe. You can configure it using the Windows Computer Management



If you don't want it running as a service, you can remove it after stopping by the command: `sc delete VelocityDbServer`. Or simply change the "Automatic" start to "Manual" start.

The server can be started from command line: `VelocityDbServer true 10`

Substitute "10" with how many worker threads you want it to use for each system database directory (one containing 0.odb, 1.odb, 2.odb 4.odb) this server is serving. The process runs as background process. A non-service VelocityDbServer is stopped by using the Task manager.

In order to distribute databases to multiple hosts, you need to install VelocityDb on each host where you want to place databases.

The VelocityDbServer is communicating on tcp/ip port number: **7031**. This server can only handle .NET clients, not .net core clients. .net core clients use port 7032 instead and the server is [VelocityDbCoreServer](#). Both these servers are installed as services by the installer.

Make sure that your Firewall lets VelocityDbServer listen/talk to other hosts with VelocityDbServer running on them.

If you are experiencing issues with the VelocityDbServer, it may help to look at the VelocityDBServerLog in the Event log, as in

Changing the default SessionBase.BaseDatabasePath in a VelocityDbServer

Edit VelocityDbServer.exe.config (in Program Folder (x86)\VelocityDB)

```
<?xml version="1.0"?>
<configuration>
<startup><supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.0"/></startup>
  <appSettings>
    <add key="BaseDatabasePath" value="c:\Databases"/>
    <add key="DoWindowsAuthentication" value="false"/>
    <add key="NumberOfWorkerThreads" value="10"/>
  </appSettings>
</configuration>
```

Option to log all activity in VelocityDBServer

You can turn on a log of all activity in a VelocityDBServer by setting the file path of ServerActivityLogFile. Set to empty string if you don't want it.

```
<?xml version="1.0"?>
<configuration>
<startup><supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.0"/></startup>
  <appSettings>
    <add key="BaseDatabasePath" value="c:/Databases"/>
    <add key="DoWindowsAuthentication" value="false"/>
    <add key="NumberOfWorkerThreads" value="10"/>
    <add key="ServerActivityLogFile" value="d:/serverLog.txt"/>
    <add key="TcpIpPortNumber" value="7031"/>
    <add key="MaximumMemoryUse" value="10000000000"/>
  </appSettings>
</configuration><?xml version="1.0"?>
```

Changing the tcp/ip port number used when communication with a VelocityDBServer

By default, VelocityDbServer is communicating on tcp/ip port number: **7031**.

If you need to use a different port number, set `SessionBase.s_serverTcpIpPortNumber` and update VelocityDBServer.exe.config (in VelocityDB installation directory) of each VelocityDB installation where you want this change.

Enabling Windows Authentication

By default Windows Authentication is now disabled when connecting to a VelocityDBServer. It is disabled by default due to a slight performance cost when connecting to a server and also due to issues with making it work with Windows 8.1 clients.

Edit VelocityDbServer.exe.config (in Program Folder (x86)\VelocityDB)

```
<?xml version="1.0"?>
<configuration>
<startup><supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.0"/></startup>
  <appSettings>
    <add key="BaseDatabasePath" value="c:\Databases"/>
    <add key="DoWindowsAuthentication" value="false"/>
    <add key="NumberOfWorkerThreads" value="10"/>
  </appSettings>
</configuration>
```

In each of your clients set

```
SessionBase.DoWindowsAuthentication = true;
```

VelocityDBCoreServer with http REST Api

Preview release of VelocityDB http API via Asp.Net Core 3.1 server combined with regular VelocityDBServer functions all in one. Connect as: `localhost:7033/active`, this server uses `port 7032` for VelocityDBServer functions. .NET Core have a mismatch with regular .NET in type names. Specifically noticed so far is `mscorlib` -> `System.Private.CoreLib`. To test using this server instead of regular .NET one, build a .netcore application using `ClientServerSession`.

API will eventually almost everything imaginable that can be done: retrieving object by id, creating new objects, deleting objects, updating objects, authentication...

What we probably will not support:

- Creating new classes or other Type instances. DLLs containing the application classes will have to be provided to server, so it can work with such object instances.

Let us know what you would like to see in this https REST API? We really appreciate all input. The source code of the VelocityDBCoreServer will soon be included in our sample solution and on GitHub.

Active connections to VelocityDBCoreServer

localhost:7033/active

```
[  
  "c:\\databases\\sample1"  
]
```

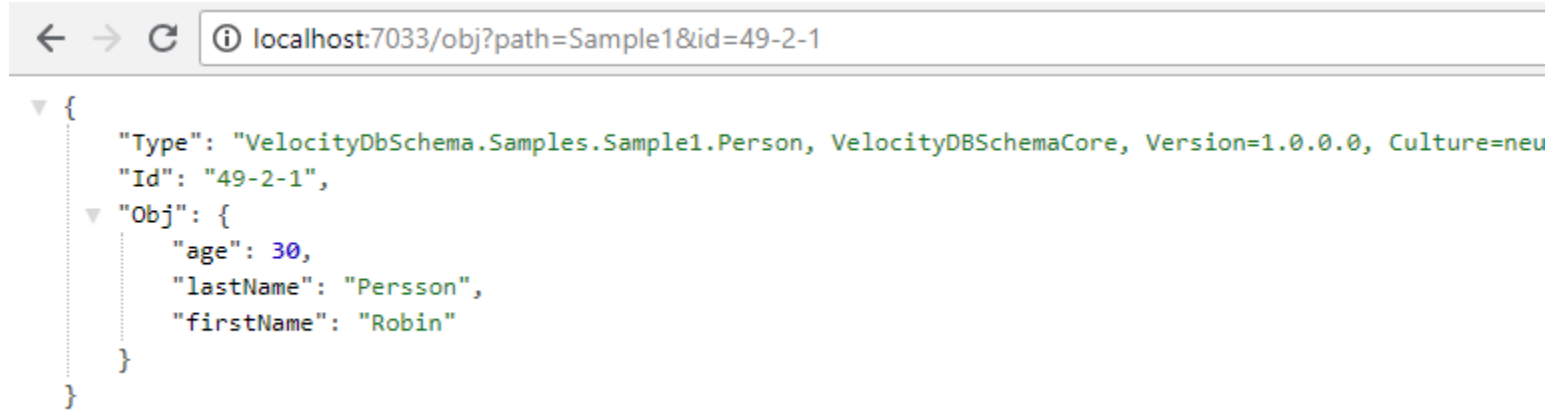
Database Manager

Schema Connectivity Add Connection...

- Host: "Asus2" Path: "C:/Databases/DatabaseManager" 86-2-1
- Host: "asus2" Path: "C:\Databases\Velocity" 86-2-2
- Host: "asus2" Path: "C:\Databases\sample1" 86-2-3
 - DatabaseManager.Model.FederationInfo 86-2-3
 - Host: "asus2" Path: "c:\databases\sample1" 2-2-2
 - VelocityDb.DatabaseLocation 2-2-2
 - 0 Transactions version: 7 pages: 4
 - 1 Schema version: 2 pages: 2
 - 2 DatabaseLocations version: 2 pages: 3
 - 5 version: 7 pages: 4
 - 6 VelocityDb.Sync.Change version: 7 pages: 3
 - 49 VelocityDbSchema.Samples.Sample1.Person version: 1 pages: 3
 - Page: 0 size: 8955 stored size: 8983 offset: 622 compression: None noEncryption version: 1 objects: 6
 - Page: 1 size: 58 stored size: 86 offset: 8 compression: None noEncryption version: 6 objects: 1 of type: VelocityDb.AutoPlac
 - Page: 2 size: 500 stored size: 528 offset: 94 compression: None noEncryption version: 6 objects: 20 of type: VelocityDbSche
 - VelocityDbSchema.Samples.Sample1.Person 49-2-1
 - firstName : Robin
 - lastName : Hood
 - age : 30
 - VelocityDbSchema.Samples.Sample1.Person 49-2-2
 - VelocityDbSchema.Samples.Sample1.Person 49-2-3
 - VelocityDbSchema.Samples.Sample1.Person 49-2-4
 - VelocityDbSchema.Samples.Sample1.Person 49-2-5
 - VelocityDbSchema.Samples.Sample1.Person 49-2-6
 - VelocityDbSchema.Samples.Sample1.Person 49-2-7
 - VelocityDbSchema.Samples.Sample1.Person 49-2-8
 - VelocityDbSchema.Samples.Sample1.Person 49-2-9
 - VelocityDbSchema.Samples.Sample1.Person 49-2-10
 - VelocityDbSchema.Samples.Sample1.Person 49-2-11
 - VelocityDbSchema.Samples.Sample1.Person 49-2-12
 - VelocityDbSchema.Samples.Sample1.Person 49-2-13
 - VelocityDbSchema.Samples.Sample1.Person 49-2-14
 - VelocityDbSchema.Samples.Sample1.Person 49-2-15
 - VelocityDbSchema.Samples.Sample1.Person 49-2-16
 - VelocityDbSchema.Samples.Sample1.Person 49-2-17
 - VelocityDbSchema.Samples.Sample1.Person 49-2-18
 - VelocityDbSchema.Samples.Sample1.Person 49-2-19
 - VelocityDbSchema.Samples.Sample1.Person 49-2-20
 - 51 VelocityDbSchema.NUnit.LargeObject version: 1 pages: 3

In screen capture above we see an active database session coming from DatabaseManager which includes a core server connection to databases in Sample.

Viewing object

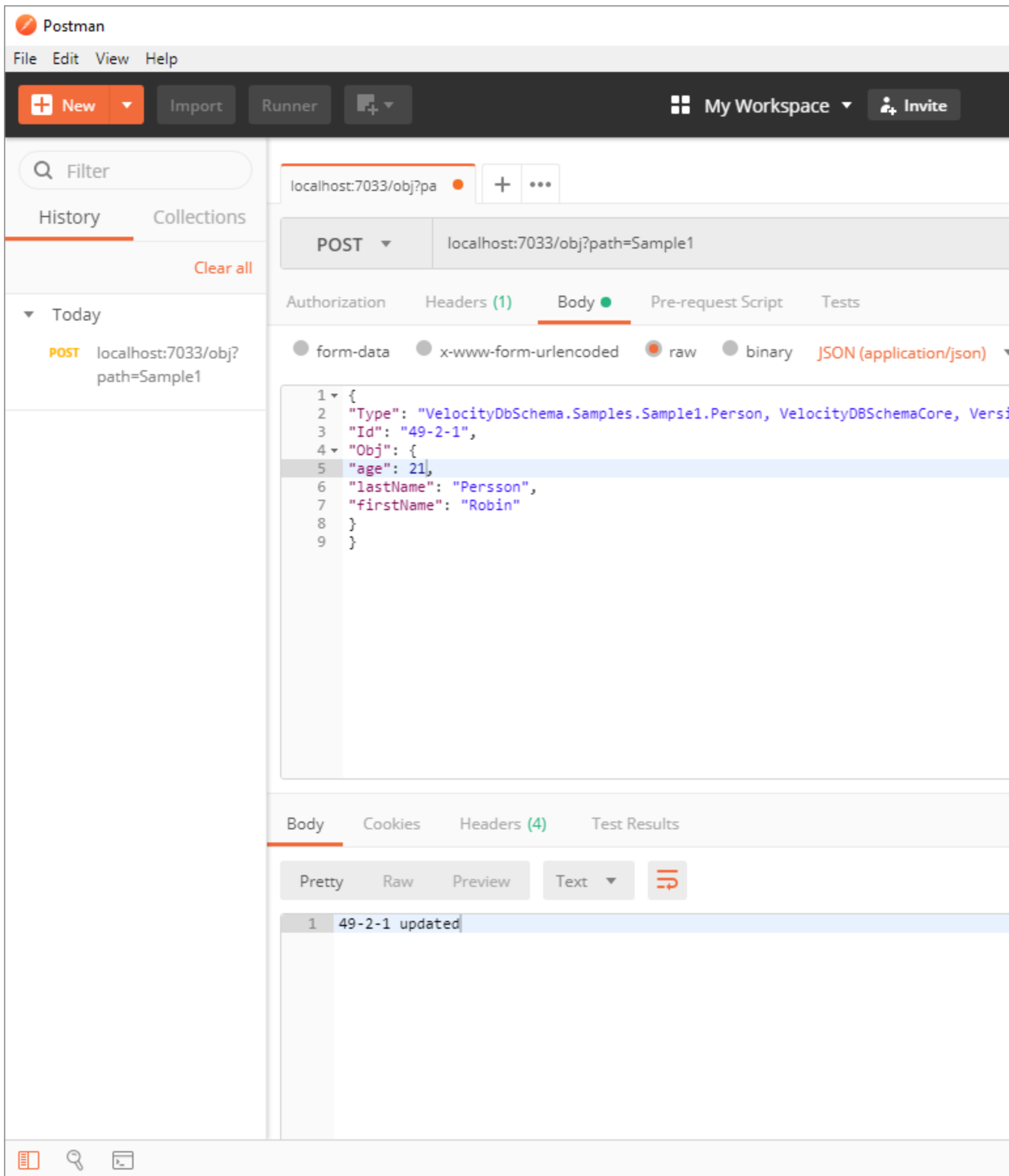


The screenshot shows a web browser window with the address bar containing the URL `localhost:7033/obj?path=Sample1&id=49-2-1`. Below the address bar, a JSON object is displayed in a collapsed state, with a small downward arrow next to the opening curly brace. The JSON structure is as follows:

```
{
  "Type": "VelocityDbSchema.Samples.Sample1.Person, VelocityDBSchemaCore, Version=1.0.0.0, Culture=new",
  "Id": "49-2-1",
  "Obj": {
    "age": 30,
    "lastName": "Persson",
    "firstName": "Robin"
  }
}
```

Updating object

Here we use the excellent tool called [Postman](#)



Now back in Chrome browser we can see that object was updated.

localhost:7033/obj?path= x

localhost:7033/obj?path=Sample1&id=49-2-1

```
▼ {
  "Type": "VelocityDbSchema.Samples.Sample1.Person, VelocityDBSchemaCore, Version=1.0.0.0, Culture=neu
  "Id": "49-2-1",
  ▼ "Obj": {
    "age": 21,
    "lastName": "Persson",
    "firstName": "Robin"
  }
}
```

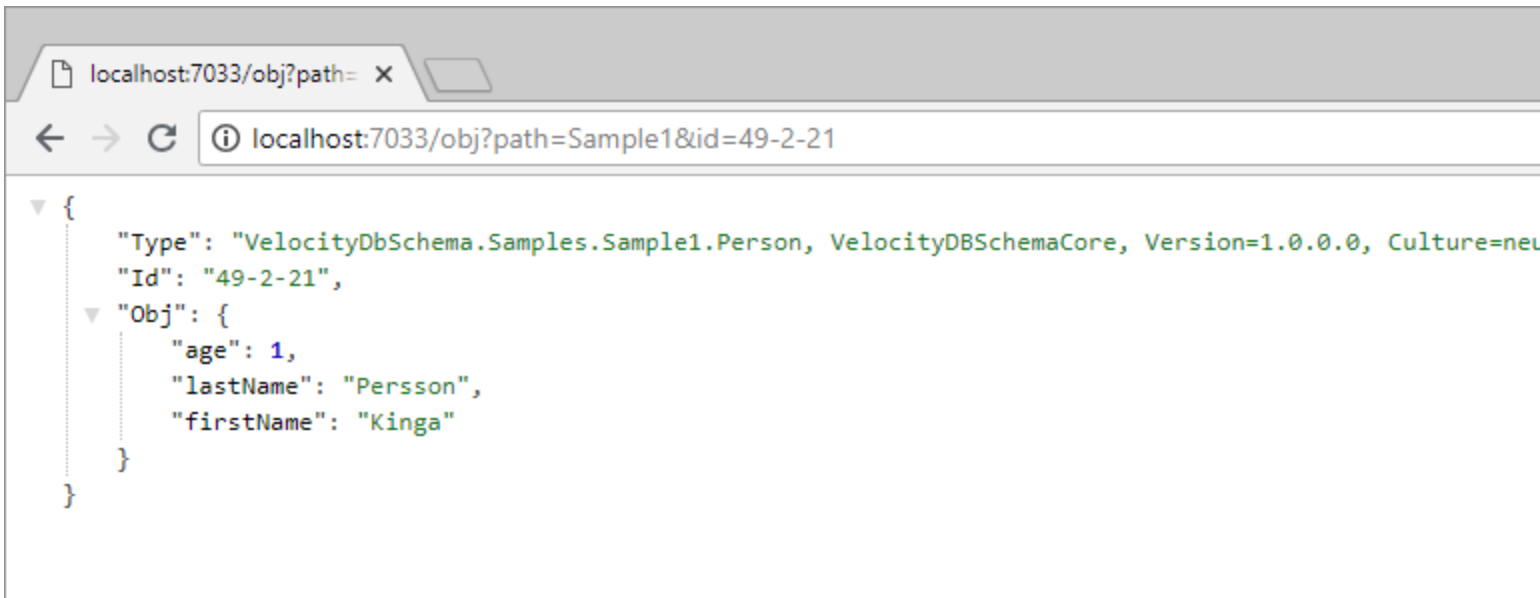
Adding Object

The screenshot shows the Postman interface for a POST request. The URL is `localhost:7033/obj?path=Sample1`. The request body is a JSON object:

```
1 {
2   "Type": "VelocityDbSchema.Samples.Sample1.Person, VelocityDBSchemaCore, Versio
3   "Obj": {
4     "age": 1,
5     "lastName": "Persson",
6     "firstName": "Kinga"
7   }
8 }
```

The response is displayed in the bottom panel as `49-2-21`.

Not that we **didn't** specify Id in the Json body of the message.



```
{
  "Type": "VelocityDbSchema.Samples.Sample1.Person, VelocityDBSchemaCore, Version=1.0.0.0, Culture=neu",
  "Id": "49-2-21",
  "Obj": {
    "age": 1,
    "lastName": "Persson",
    "firstName": "Kinga"
  }
}
```

Settings for the VelocityDBCoreServer

To change settings you have to first stop the servers: VelocityDBServer and VelocityDBCoreServer in the Services Window. You may have to stop them multiple times as shutdown isn't graceful right now and it restarts. Windows makes it hard to edit the file. You may have to save edited version somewhere else and then move the file via an Administrator command line shell. Sorry about that.

Settings file is in: C:\Program Files (x86)\VelocityDB\core (or similar path from your Windows drive)

core

Share View

> This PC > Local Disk (C:) > Program Files (x86) > VelocityDB > core

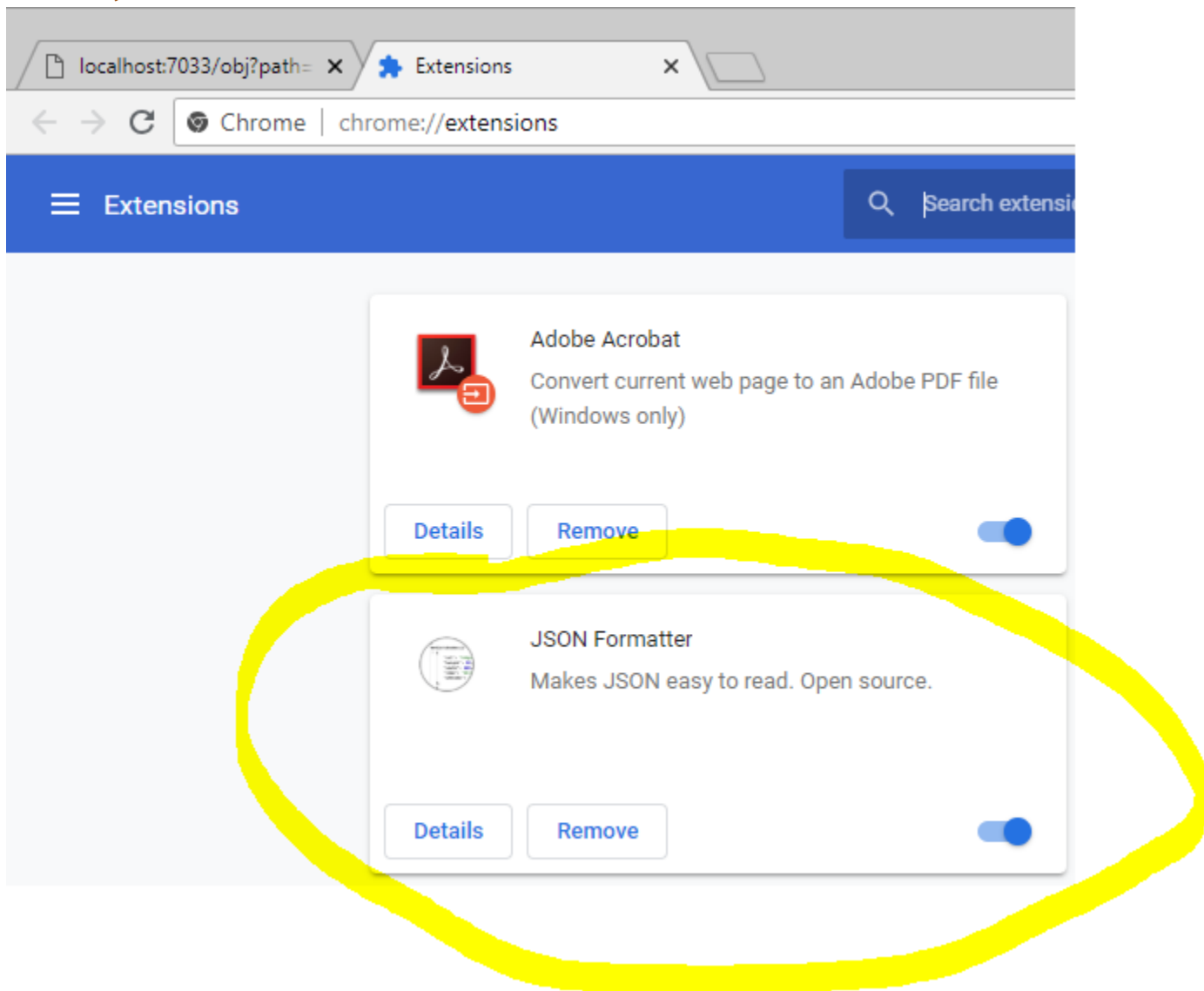
Name	Date modified	Type
api-ms-win-core-console-l1-1-0.dll	7/25/2018 9:24 AM	Application extens
api-ms-win-core-datetime-l1-1-0.dll	7/25/2018 9:24 AM	Application extens
api-ms-win-core-debug-l1-1-0.dll	7/25/2018 9:24 AM	Application extens
api-ms-win-core-errorhandling-l1-1-0.dll	7/25/2018 9:24 AM	Application extens
api-ms-win-core-file-l1-1-0.dll	7/25/2018 9:24 AM	Application extens
api-ms-win-core-file-l1-2-0.dll	7/25/2018 9:24 AM	Application extens
api-ms-win-core-file-l2-1-0.dll	7/25/2018 9:24 AM	Application extens
api-ms-win-core-handle-l1-1-0.dll	7/25/2018 9:24 AM	Application extens
api-ms-win-core-heap-l1-1-0.dll	7/25/2018 9:24 AM	Application extens
api-ms-win-core-interlocked-l1-1-0.dll	7/25/2018 9:24 AM	Application extens
api-ms-win-core-libraryloader-l1-1-0.dll	7/25/2018 9:24 AM	Application extens
api-ms-win-core-localization-l1-2-0.dll	7/25/2018 9:24 AM	Application extens
api-ms-win-core-memory-l1-1-0.dll	7/25/2018 9:24 AM	Application extens
api-ms-win-core-namedpipe-l1-1-0.dll	7/25/2018 9:24 AM	Application extens
api-ms-win-core-processenvironment-l1-1-0.dll	7/25/2018 9:24 AM	Application extens
api-ms-win-core-processthreads-l1-1-0.dll	7/25/2018 9:24 AM	Application extens
api-ms-win-core-processthreads-l1-1-1.dll	7/25/2018 9:24 AM	Application extens
api-ms-win-core-profile-l1-1-0.dll	7/25/2018 9:24 AM	Application extens
api-ms-win-core-rtlsupport-l1-1-0.dll	7/25/2018 9:24 AM	Application extens
api-ms-win-core-string-l1-1-0.dll	7/25/2018 9:24 AM	Application extens
api-ms-win-core-synch-l1-1-0.dll	7/25/2018 9:24 AM	Application extens
api-ms-win-core-synch-l1-2-0.dll	7/25/2018 9:24 AM	Application extens
api-ms-win-core-sysinfo-l1-1-0.dll	7/25/2018 9:24 AM	Application extens
api-ms-win-core-timezone-l1-1-0.dll	7/25/2018 9:24 AM	Application extens
api-ms-win-core-util-l1-1-0.dll	7/25/2018 9:24 AM	Application extens
api-ms-win-crt-conio-l1-1-0.dll	7/25/2018 9:24 AM	Application extens
api-ms-win-crt-convert-l1-1-0.dll	7/25/2018 9:24 AM	Application extens
api-ms-win-crt-environment-l1-1-0.dll	7/25/2018 9:24 AM	Application extens
api-ms-win-crt-filestream-l1-1-0.dll	7/25/2018 9:24 AM	Application extens
api-ms-win-crt-heap-l1-1-0.dll	7/25/2018 9:24 AM	Application extens
api-ms-win-crt-locale-l1-1-0.dll	7/25/2018 9:24 AM	Application extens
api-ms-win-crt-math-l1-1-0.dll	7/25/2018 9:24 AM	Application extens
api-ms-win-crt-multibyte-l1-1-0.dll	7/25/2018 9:24 AM	Application extens
api-ms-win-crt-private-l1-1-0.dll	7/25/2018 9:24 AM	Application extens
api-ms-win-crt-process-l1-1-0.dll	7/25/2018 9:24 AM	Application extens
api-ms-win-crt-runtime-l1-1-0.dll	7/25/2018 9:24 AM	Application extens
api-ms-win-crt-stdio-l1-1-0.dll	7/25/2018 9:24 AM	Application extens
api-ms-win-crt-string-l1-1-0.dll	7/25/2018 9:24 AM	Application extens
api-ms-win-crt-time-l1-1-0.dll	7/25/2018 9:24 AM	Application extens
api-ms-win-crt-utility-l1-1-0.dll	7/25/2018 9:24 AM	Application extens
appsettings.Development.json	8/29/2018 1:01 PM	JSON File
appsettings.json	9/8/2018 12:55 PM	JSON File

Schema: <http://json.schemastore.org/appsettings>

```
1  {
2  |  "Logging": {
3  |  |  "LogLevel": {
4  |  |  |  "Default": "Warning"
5  |  |  |  }
6  |  |  },
7  |  |  "AllowedHosts": "*",
8  |  |  },
9  |  |  "VelocityDBServer": {
10 |  |  |  "BaseDatabasePath": "c:/Databases",
11 |  |  |  "DowindowsAuthentication": false,
12 |  |  |  "NumberOfWorkerThreads": 10,
13 |  |  |  "ServerActivityLogFile": null,
14 |  |  |  "TcpIpPortNumber": 7032,
15 |  |  |  "MaximumMemoryUse": 1000000000,
16 |  |  |  "Schema": [ "C:/VelocityDB/Sample1Core/bin/Release/netcoreapp2.1/VelocityDbSchemaCore.dll" ]
17 |  |  |  }
18 |  |  }
19 }
```

The server needs access to your **application schema** built as a **.Net Core 3.1 library**. A regular .Net library doesn't work as .Net Core Types differ from regular .Net (a Microsoft issue, not our). Specify path in "Schema" section of the appsettings, if multiple separate with a ,.

Chrome Json Formatter



With the formatter the JSON code looks much better!

Why installation ends up in Program Files (x86) instead of Program Files?

An issue is that Install Shield LE 2013 does not support 64bit installers so installation ends up in Program Files (86) instead of Program Files.

We use Install Shield LE 2013 which comes with Visual Studio. For VelocityDbServer service install we create a merge module using [WiX Toolset](#).

The latest version with Visual Studio 2013 is supposed to support 64 bit installers but we have not figured out how to do it yet. Be patient, we will solve it eventually or let us know how it's accomplished!

.NET CORE

This version of the VelocityDB library lets you build [portable apps](#) that can run on multiple platforms including: Windows, Linux and Mac. Reference VelocityDBCore.dll in your app or install the [VelocityDB NuGet](#).

This platform requires a default constructor as with [Universal Windows](#). The API provided by .Net Core libraries is not yet complete. Notably missing and causing performance/functionality issues for VelocityDB are:

1. [ResolveEventHandler](#)
2. [Assembly.LoadWithPartialName](#)
3. [Environment](#)
4. [public static Type GetType\(string typeName, Func<AssemblyName, Assembly> assemblyResolver, Func<Assembly, string, bool, Type> typeResolver, bool throwOnError\)](#)
5. [FormatterServices.GetUninitializedObject](#)
6. [AppDomain](#)
7. [Trace](#)

Consequences of missing API include: each persisted class must have a constructor with no parameters, a Type cannot be loaded if the assembly version is changed so we'll have to NOT update the assembly version of VelocityDBCore.dll.

.NET 5 and .NET Standard 2.0

Most or all the above-mentioned missing API is now available with .NET 5 (was .NET Core), .Net Standard 2.0 is no longer missing any of this API.

Universal Windows

This version of the VelocityDB library lets you build [native Windows apps](#), compiles to machine code as with unmanaged C++ applications. Reference VelocityDBUniversalWindows.dll in your app or install the [VelocityDB NuGet](#).

This platform requires a default constructor as with [.Net CORE](#). The API provided by Microsoft for Universal Windows libraries is not yet complete. Notably missing and causing performance/functionality issues for VelocityDB are:

8. [System.Security.Cryptography](#)
9. [Thread](#)
10. [TcpClient](#)
11. [Environment](#)
12. [System.Reflection.Assembly](#)
13. [Assembly.LoadWithPartialName](#)
14. [Dns](#)
15. [public static Type GetType\(string typeName, Func<AssemblyName, Assembly> assemblyResolver, Func<Assembly, string, bool, Type> typeResolver, bool throwOnError\)](#)
16. [Type.GetTypeCode](#)
17. [DynamicMethod](#)
18. [FormatterServices.GetUninitializedObject](#)
19. [Console](#)
20. [AppDomain](#)
21. [Trace](#)

Consequences of missing API include: each persisted class must have a constructor with no parameters, a Type cannot be loaded if the assembly version is changed so we'll have to NOT update the assembly version of VelocityDBUniversalWindows.dll.

Where to store databases with Universal Windows?

We tested using this path: `Windows.Storage.ApplicationData.Current.LocalFolder.Path;`

We tried to set the `SessionBase.BaseDatabasePath` to this but then when ran into errors while doing the obfuscation of the library. We will try again! No obfuscation required with apps since they are compiled to binary code as with C++!

iOS

The installation directory contains `iOS\VelocityDB.dll` and `iOS\VelocityDB.xml`, add a reference to this DLL if you are targeting iOS for your application. Some of the VelocityDB code is not as efficient on iOS due to `System.Reflection.Emit` not being supported, see reasons [here](#).

Android

The installation directory contains `Android\VelocityDB.dll` and `Android\VelocityDB.xml`, add a reference to this DLL if you are targeting Android for your application. You can develop Android applications using Visual Studio 2015 with Xamarin. We currently don't have any sample applications but follow the [Android Xamarin guides](#) and ask us if you get stuck with how to use it with VelocityDB.

Asp.Net Identity

A driver for storing user credentials in VelocityDB using [Asp.Net Identity](#) is part of the VelocityDB.sln and a sample Web site, `AspNetIdentitySample`, is also provided that uses asp.Net Identity with VelocityDB. These projects require .Net 4.5.2 or higher.

Application Deployment and VelocityDB license check

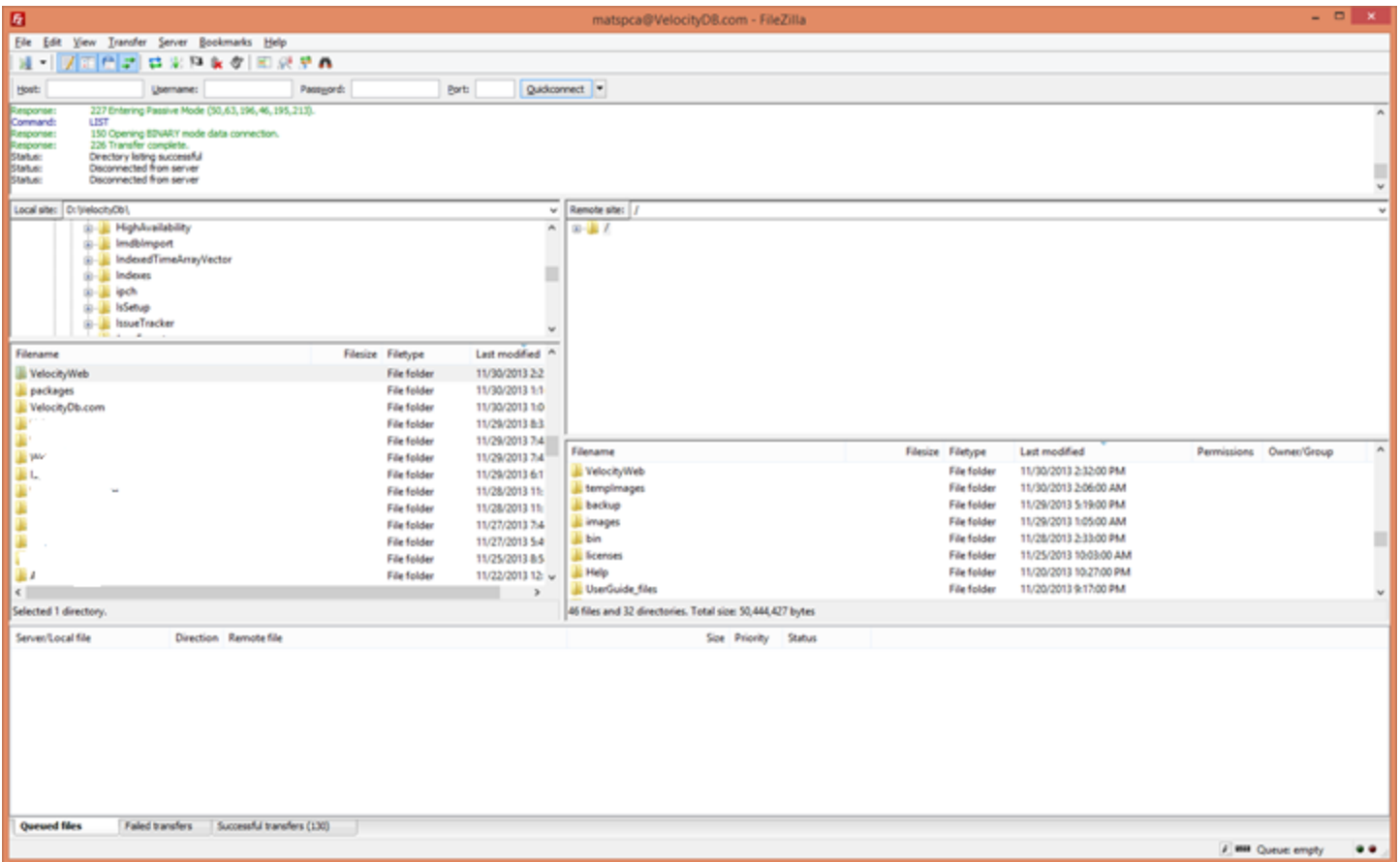
Normally you need to deploy the license database, `4.odt`, but if you are publishing your application as open source or your database files in a publicly accessible directory then do not include `4.odt` since that would enable unlicensed usage of VelocityDB. Instead [register](#) all your persistent classes prior to deployment and deploy database `1.odt` which then contains your entire database schema. VelocityDB may do a license check whenever database schema is added to or is updated.

Setting Up the sample Web Site (VelocityWeb) on a hosting web site (in this case GoDaddy)

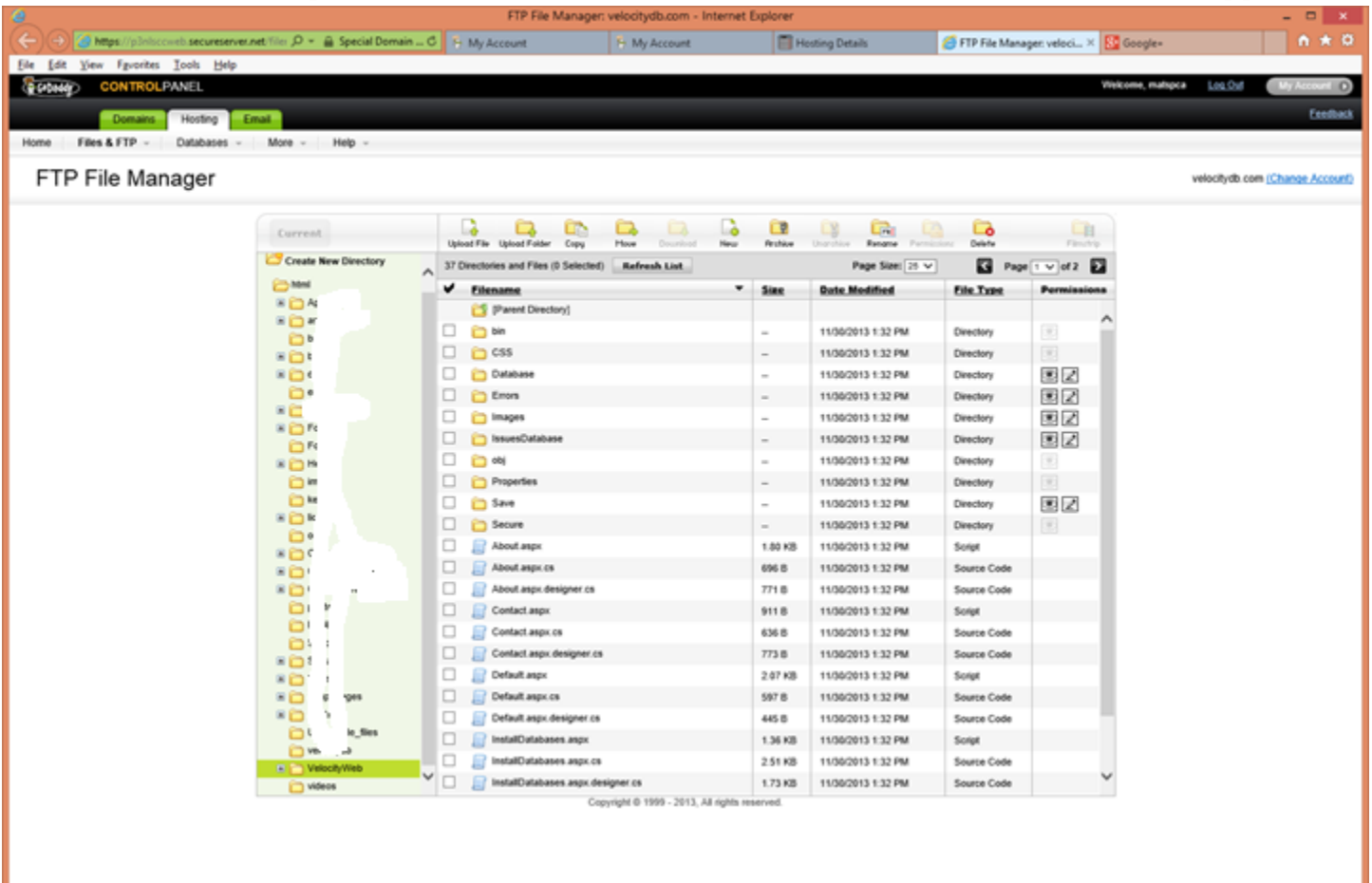
The VelocityDB sample solution contains a sample web application using VelocityDB, here we show you how to deploy this application.

Transfer all the files to your hosting account

Copy the entire directory named `VelocityWeb` to the root of your hosting directory. We use FileZilla (free software).



Login to your hosting provider to enable write access to a few of the directories in the application



FTP File Manager

velocitydb.com [\(Change Account\)](#)

Current

Upload File Upload Folder Copy Move Download New Archive Unarchive Rename Permissions Delete Filetree

Create New Directory

Set Permissions

Advanced Permissions

Multiple items selected may not have matching permissions. Any changes made will apply to all items.

Set permissions for selected folders.

Inherit (Inherit permissions from parent directory)

Read (Directory contents are visible to users)

Write (Applications can write to this directory)

Reset all children to inherit (All subdirectories will be reset to inherit from current directory)

OK Cancel

37 Directories and Files (5 Selected) Refresh List Page Size: 25 Page 1 of 2

Filename	Size	Date Modified	File Type	Permissions
[Parent Directory]				
<input type="checkbox"/> bin	-	11/00/2013 1:32 PM	Directory	
<input type="checkbox"/> CSS	-	11/00/2013 1:32 PM	Directory	
<input checked="" type="checkbox"/> Database	-	11/00/2013 1:32 PM	Directory	
<input checked="" type="checkbox"/> Errors	-	11/00/2013 1:32 PM	Directory	
<input checked="" type="checkbox"/> Images	-	11/00/2013 1:32 PM	Directory	
<input checked="" type="checkbox"/> IssuesDatabase	-	11/00/2013 1:32 PM	Directory	
<input type="checkbox"/> obj	-	11/00/2013 1:32 PM	Directory	
<input type="checkbox"/> Properties	-	11/00/2013 1:32 PM	Directory	
<input checked="" type="checkbox"/> Save	-	11/00/2013 1:32 PM	Directory	
<input type="checkbox"/> Secure	-	11/00/2013 1:32 PM	Directory	

Create an application root virtual directory for the new web application

Creating directories

To create a new IIS virtual directory, select the parent directory and click Create. In the Create panel, enter a directory name and specify folder permissions. Multiple directories can be created by separating each directory name with a comma. When finished click OK.

Create

Directory Name: VelocityWeb

Path: /

Select Settings (Non-IIS Permissions can be set in the [File Manager](#))

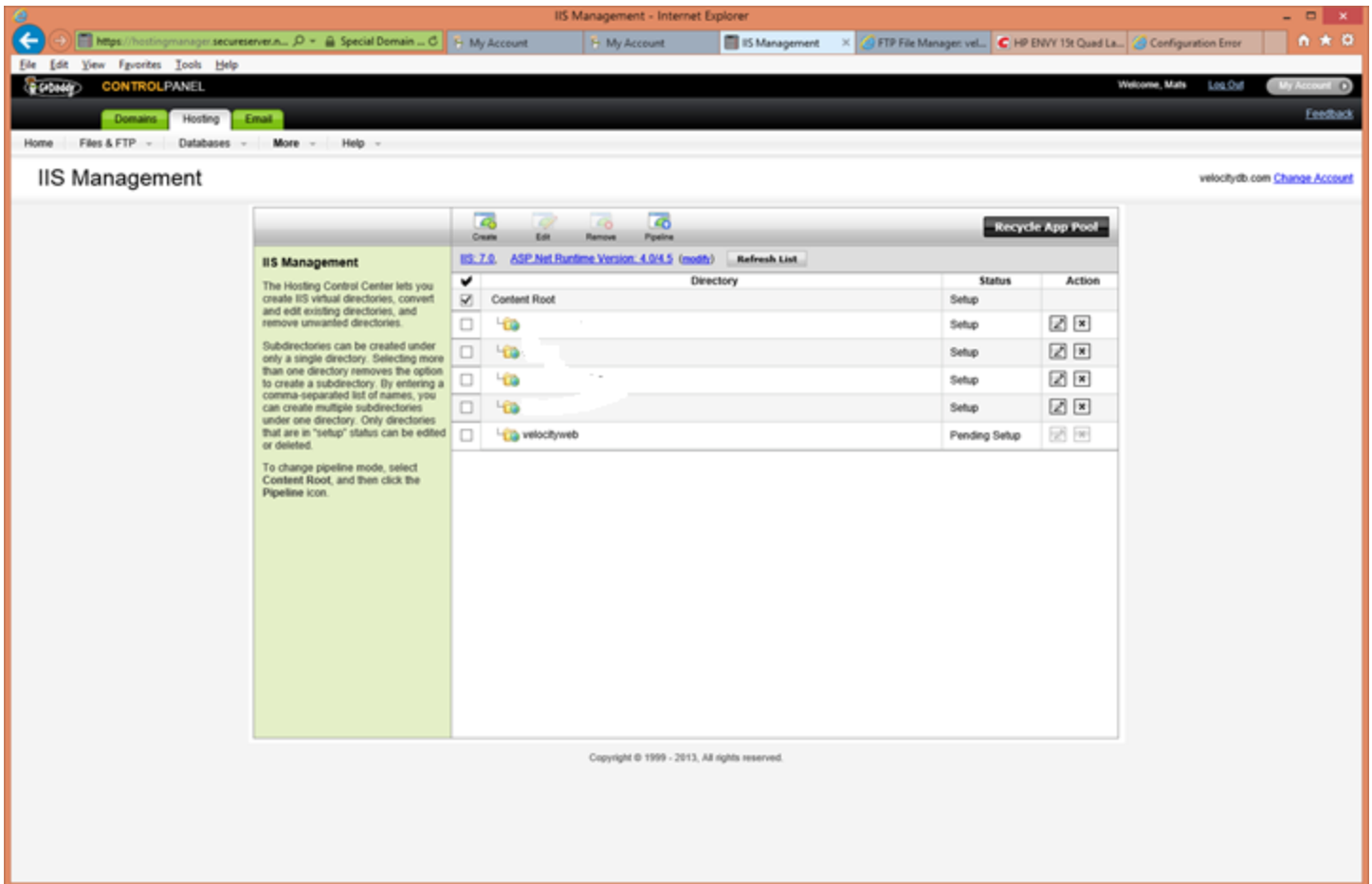
Anonymous Access Directory Browsing Set Application Root

OK Cancel

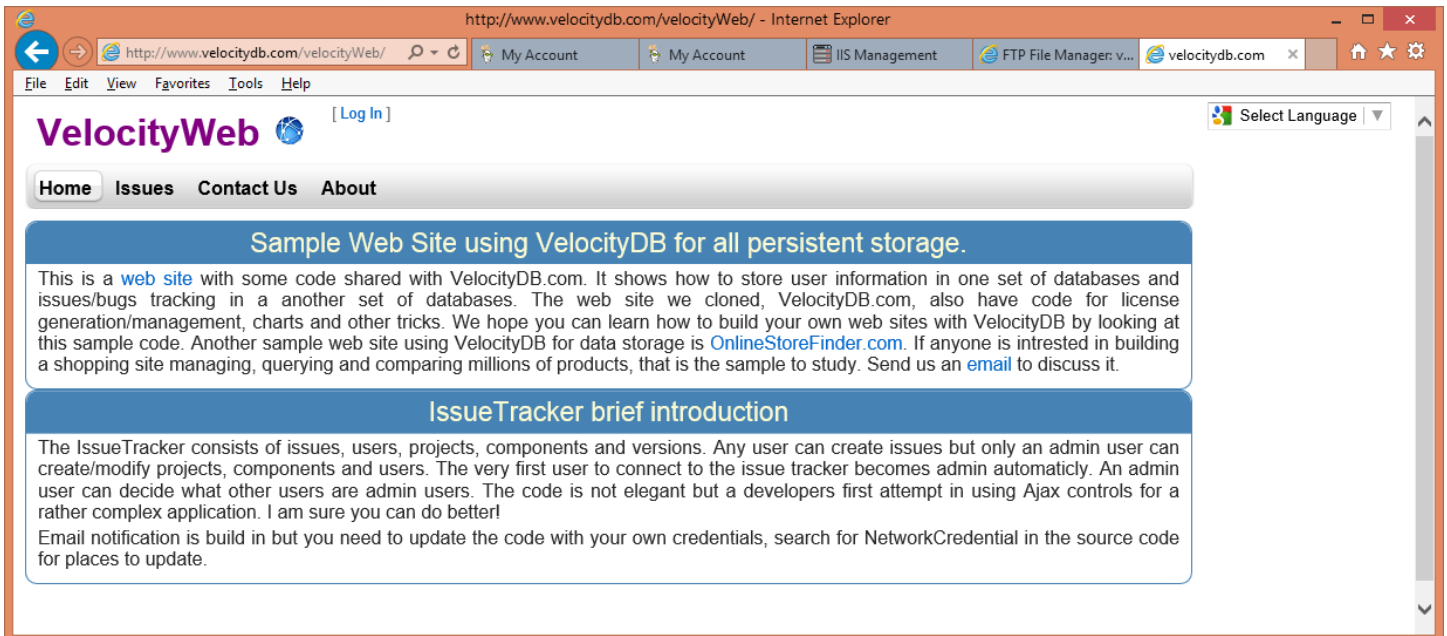
IS 7.0 ASP.NET Runtime Version: 4.0.5045 (month) Refresh List

Directory	Status	Action
Content Root	Setup	
...	Setup	
...	Setup	
velocitydb	Setup	

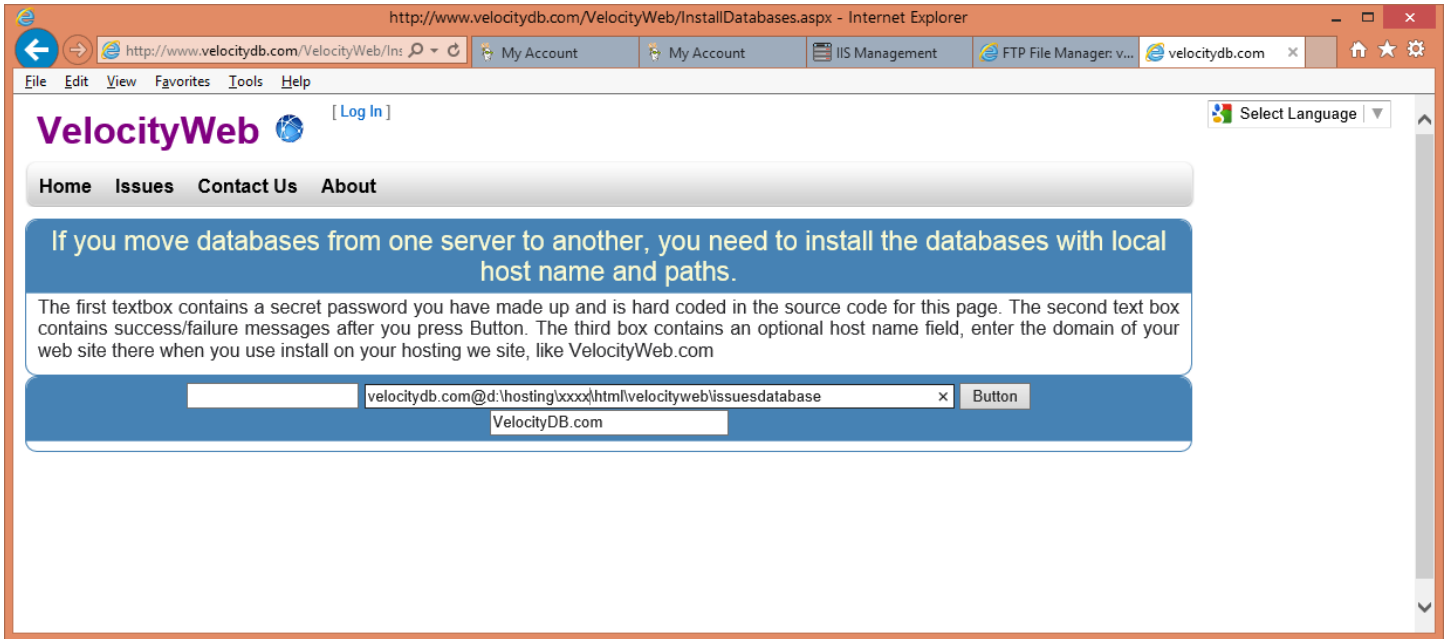
Copyright © 1999 - 2013. All rights reserved.



Wait a few minutes then point your browser at your web application



If you transferred your application directory with databases then install your databases in their new location.



If all is well, you are done, access the application and the databases!

