General Documentation

Table of Contents

[Overview 3](#_Toc412623144)

[MVVM Design Pattern 4](#_Toc412623145)

[Repository Design Pattern 4](#_Toc412623146)

[Unit of Work Design Pattern 4](#_Toc412623147)

[ORM 4](#_Toc412623148)

[Advantages of ORM 4](#_Toc412623149)

[Entity Framework 5](#_Toc412623150)

[SQLite 5](#_Toc412623151)

[Configuration 5](#_Toc412623152)

[SQLite with Entity Framework 6.1.2 5](#_Toc412623153)

[Add the correct SQLite EF6 NuGet package to your project 8](#_Toc412623154)

[App.Config 9](#_Toc412623155)

[UI (User Inteface) 11](#_Toc412623156)

[UI Undo Framework (Monitored Undo Framework) 15](#_Toc412623157)

[Summary 15](#_Toc412623158)

[Design Goals 15](#_Toc412623159)

[Change Monitoring vs. Command Model 15](#_Toc412623160)

[Classes 16](#_Toc412623161)

[Interfaces 17](#_Toc412623162)

[Class Diagram 19](#_Toc412623163)

[Example 19](#_Toc412623164)

[HomeMenu 20](#_Toc412623165)

[Overview 20](#_Toc412623166)

[Logging 23](#_Toc412623167)

[Overview 23](#_Toc412623168)

[Project dependencies 24](#_Toc412623169)

[DB Access Type Dependencies 24](#_Toc412623170)

[UI Types Dependencies 25](#_Toc412623171)

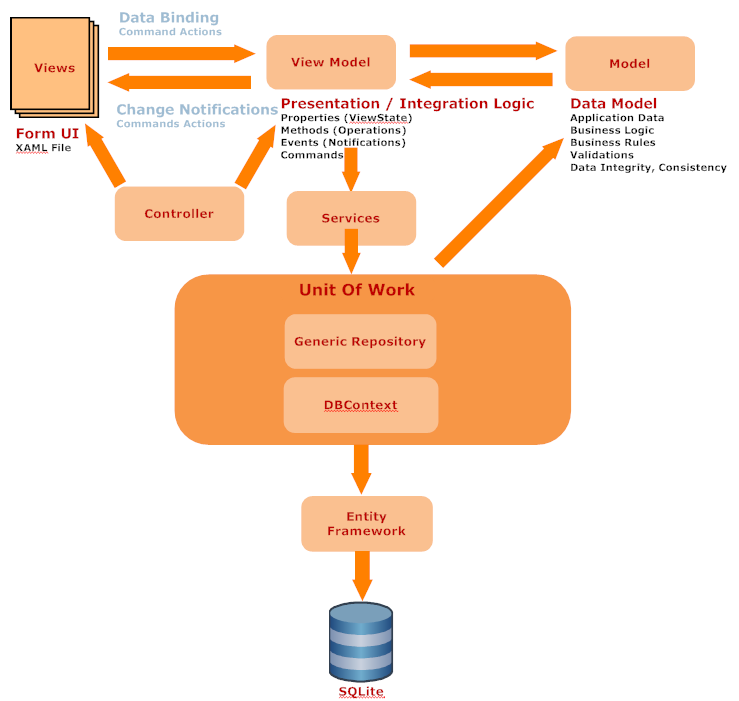
[ViewModels Type Dependencies (Home Menu) 25](#_Toc412623172)

[Entity Designer Diagram 26](#_Toc412623173)

[Guard concept 28](#_Toc412623174)

[References 28](#_Toc412623175)

# Overview



## MVVM Design Pattern

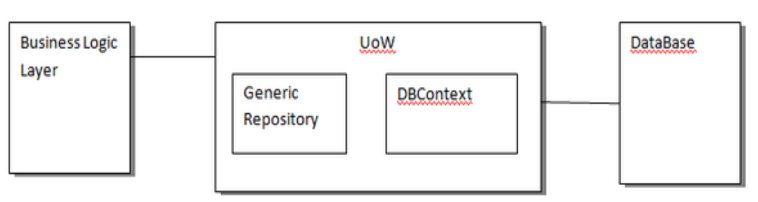
## Repository Design Pattern

Repository Design Pattern is used to create an abstraction layer between the data access layer and the business logic layer. This abstraction layer contains data manipulation methods which communicate with the data access layer to serve data as per the business requirements to the logical layer. The main purpose to create this layer is for isolating data access layer so that changes cannot affect the business logic layer directly.

## Unit of Work Design Pattern

Unit of Work Design Pattern is a pattern to handle transaction during the data manipulation using the repository pattern. According to Martin Fowler, Maintains a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problem.

Implementing these pattern gives advantages to automate unit testing.



## ORM

Object relational mapping is a programming technique for converting data between incompatible type systems in relational databases and object oriented programming languages.

We can handle objects/models as a schema of the database object. We can write code/query on model object collections which load data by maintaining database relations.

### Advantages of ORM

* Data manipulation (CRUD) is handled without writing SQL queries
* Data mapping between result set and entity collection/entity is one of the major features of ORM
* Using UoW/Context, it can handle the transaction during data manipulation
* Writing one Data Model in a single place for managing the model’s storage in DB.
* It fits the natural way of coding with application’s language.
* Programmers can think fully object oriented way to make any operation with business logic layer and data storage.

## Entity Framework

Entity Framework is a popular object relational mapper that enables .NET developers to work with relational data using domain specific objects. You can use POCO (plain old CLR objects), such as existing domain objects with your data model.

## SQLite

# Configuration

## SQLite with Entity Framework 6.1.2

For Visual Studio 2013, the download we need is named: [sqlite-netFx451-setup-bundle-x86-2013-1.0.94.0.exe](http://system.data.sqlite.org/downloads/1.0.94.0/sqlite-netFx451-setup-bundle-x86-2013-1.0.94.0.exe)  for the current version of the ADO.NET provider

The actual version of the SQLite is 1.0.94.

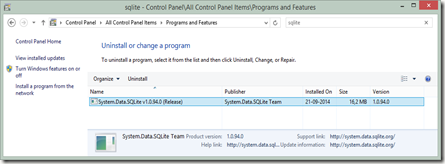


For SQLite you must download the provider from <http://system.data.sqlite.org/index.html/doc/trunk/www/downloads.wiki> - this page is an unreadable mess, sadly.

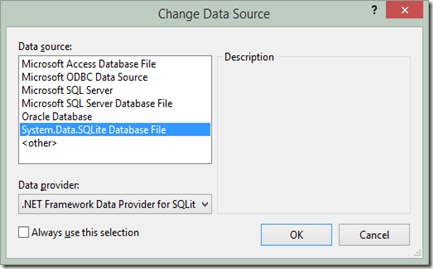
**It is important that the DDEXprovider version matches the version of the current SQLite EF6 NuGet package that you use in your project!**

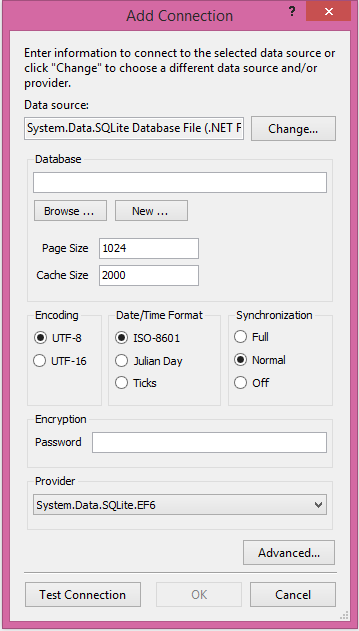
Make sure to enable Visual Studio integration and GAC registration during installation.

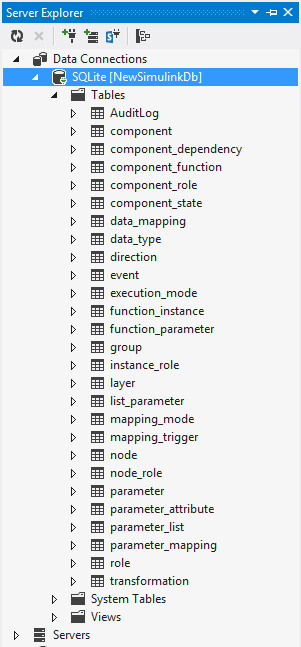
You can check which version of the DDEX provider you have installed in “Add or Remove Programs”:

[](http://lh6.ggpht.com/-76rQX48LiWk/VHNuoUKJRcI/AAAAAAAABuQ/CHaPLLn7EM4/s1600-h/image7.png)

Once installed, you can add a connection to your SQLite database from Server Explorer:

[](http://lh5.ggpht.com/-gvLI4RaxpaQ/VHNupnobv-I/AAAAAAAABug/XB8ZGLXxyBA/s1600-h/image17.png)

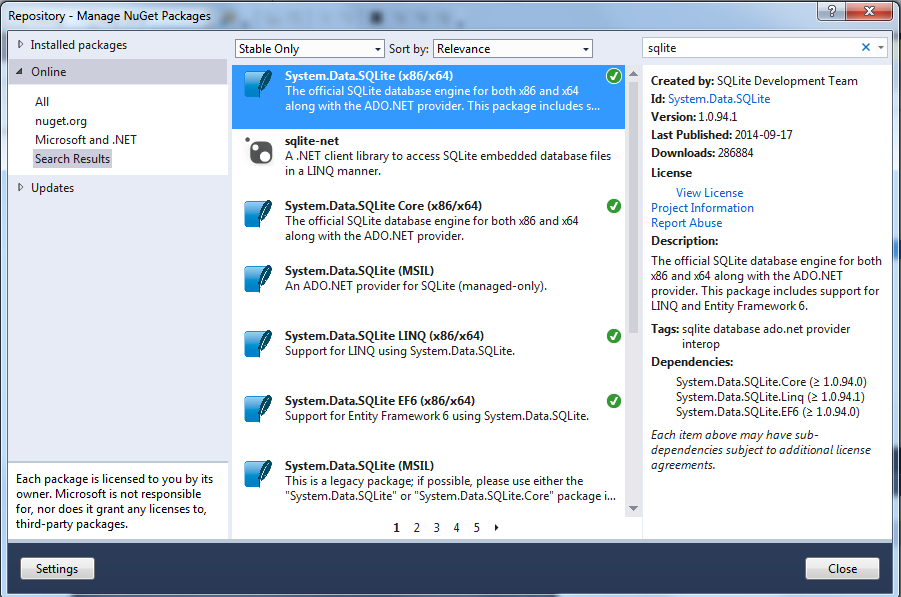




### Add the correct SQLite EF6 NuGet package to your project

To demonstrate using the SQLite EF6 provider, let’s go through creating a console app in Visual Studio 2013, that uses the Chinook sample database that we connected to above.

To add the SQLite EF6 provider to your project, select the System.Data.SQLite.EF6 package:



## App.Config

The most important settings in the application configuration file are the SQLite version number and the provider.so all of this interfaces Entity Framework and SQLite involved work together seamlessly.

<?xml version="1.0" encoding="utf-8"?>

<configuration>

<configSections>

<section name="entityFramework" type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection, EntityFramework, Version=6.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" requirePermission="false"/>

</configSections>

<startup>

<supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5.1"/>

</startup>

<connectionStrings>

<add name="SimulinkContext" connectionString="Data Source=C:\testcode\abb\AC800PEC\Data\simulinkSampleDB.db" providerName="System.Data.SQLite"/>

<!--<add name="SimulinkContext" connectionString="Data Source=.\Data\simulinkSampleDB.db" providerName="System.Data.SQLite"/>-->

</connectionStrings>

<system.data>

<DbProviderFactories>

<remove invariant="System.Data.SQLite"/>

<add name="SQLite Data Provider" invariant="System.Data.SQLite" description="Data Provider for SQLite" type="System.Data.SQLite.SQLiteFactory, System.Data.SQLite,Version=**1.0.94.0**, Culture=neutral, PublicKeyToken=db937bc2d44ff139"/>

</DbProviderFactories>

</system.data>

<entityFramework>

<providers>

**<provider invariantName="System.Data.SQLite" type="System.Data.SQLite.EF6.SQLiteProviderServices, System.Data.SQLite.EF6"/>**

</providers>

<defaultConnectionFactory type="System.Data.Entity.Infrastructure.LocalDbConnectionFactory, EntityFramework">

<parameters>

<parameter value="v11.0"/>

</parameters>

</defaultConnectionFactory>

</entityFramework>

</configuration>

# UI (User Inteface)

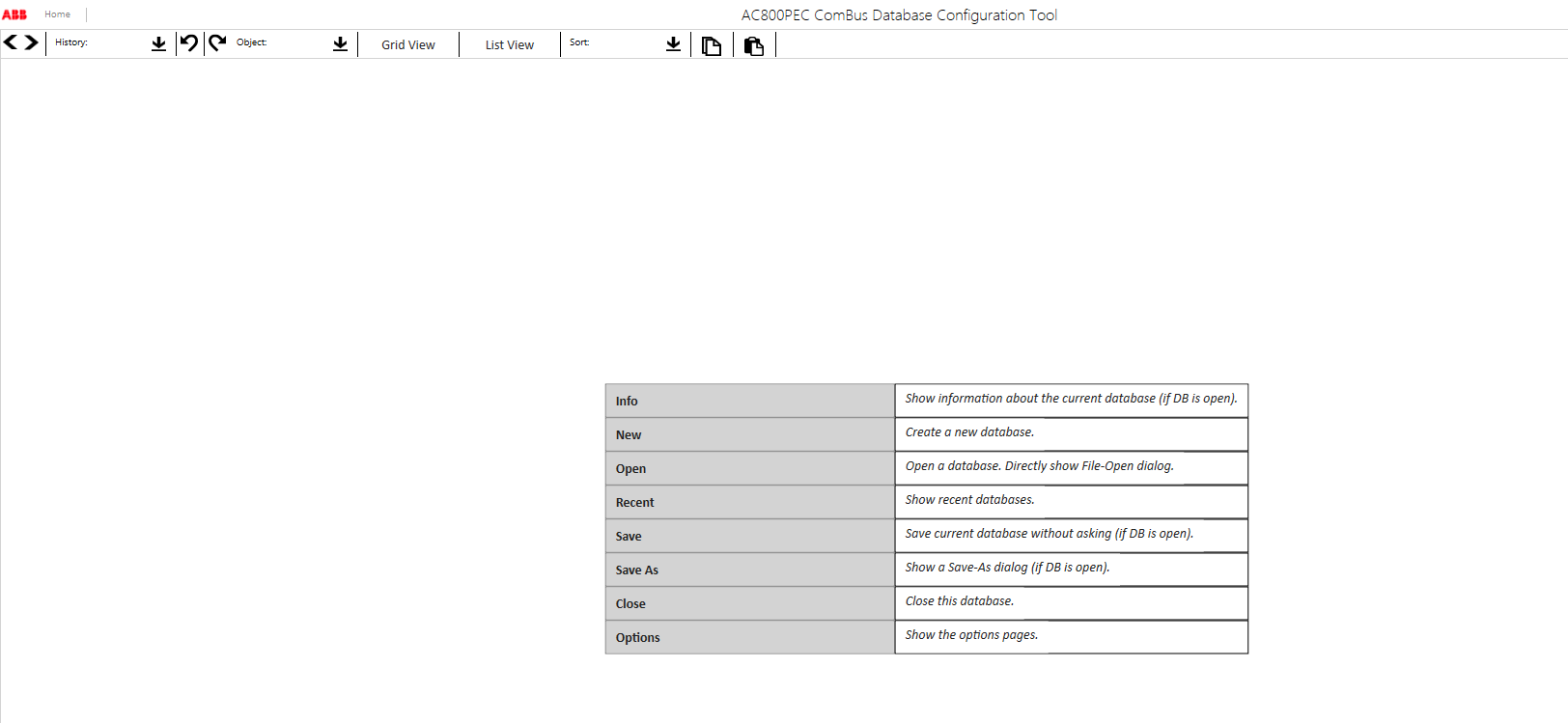
* Functional Requirements
* Undo/Redo
* Navigation
  + Forward/Backward
  + History
* Non-functional Requirements
  + Look and feel should be modern, like Office 2013
  + Development needs to be fast due to time/budget constraints

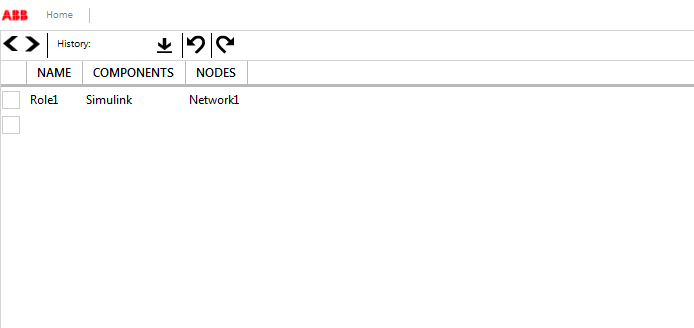
Frameworks

* Navigation: Magellan
  + Magellan is a reasonably mature (since 2010) WPF framework designed to facilitate navigation in WPF applications based on a web-oriented approach like ASP.NET MVC. It gives us history and forward/back functions more or less out of the box. It does require the adoption of some conventions, and some code in the application startup to register each view with a controller, but that is a small price to pay.
  + Concept is: the Magellan controller handles routing of requests for object overview and detail pages. These are separated by physical placement and convention, and parameters are passed into them by the controller to allow the viewmodel to initialize itself correctly. Then object-specific layout and logic concerns are handled by the individual view and viewmodel classes.
* Look and Feel: Mahapps.Metro
  + Mahapps.Metro is an extremely successful design framework which gives a windows 8 or “metro” – like look and feel to WPF and Silverlight applications. It is widely used with good support and extensive documentation. We have used it with relative ease so far to generate a reasonably modern look and feel for the AC800PEC application and we are confident that we can make it even better with time.
* MonitoredUndoRedo:
  + We are planning to use this to enable Undo and redo in the UI layer. So far we have not implemented anything but it looks promising

Screenshots of app so far:

Landing Page:



Roles list view (so far readonly)

## UI Undo Framework (Monitored Undo Framework)

### Summary

Undo (and Redo) is a feature expected by most users, especially with complex or well established applications. The monitored undo framework provides a foundation for Undo / Redo, making it simple to track and apply in various portions of the application.

### **Design Goals**

The implementation has the following design attributes:

* Generic, re-usable design that can be used with a variety of applications.
* Simple usage patterns.
* A “change monitoring” approach, rather than a “command model” approach.

### Classes

#### **UndoService**

UndoService is the top level of the undo / redo system. It contains one or more UndoRoots, accessible via the indexer on the UndoService.

UndoService.Current property will return the singleton instance of the UndoService. Use this when interacting with the undo service.

UndoService.Current[modelRoot] will return an instance of UndoRoot for the specified modelRoot.

#### **UndoRoot**

UndoRoot collects changes related to a specific document or instance of a model. This allows an application to track multiple, distinct undo stacks. This class has most of the public API methods that you’ll use to undo, redo, and add changes.

Contains FIFO stacks of ChangeSets for undo and redo actions. Includes the logic to manage the undo and redo stacks. For example, the redo stack is cleared whenever a new undo ChangeSet is added.

UndoRoot.Undo() will undo the last operation. (Overloads available)

UndoRoot.Redo() will redo the last operation. (Overloads available)

UndoRoot.AddChange() will add a new Change to the system. ChangeSets are automatically created as needed.

UndoRoot.Clear() will clear the undo and redo stacks.

#### **ChangeSet**

ChangeSet has a collection of Change instances. It represents a unit of undo or redo work.

#### Change

Change is an individual action to perform when undoing or redoing a ChangeSet. The Change class contains Action() delegates (or lambdas) that perform the undo and redo operations.

#### **UndoBatch**

Is a "scope based" helper that can group changes into a single ChangeSet. It detects new Changes that occur and automatically groups them into a single ChangeSet.

UndoBatch is designed similarly to the TransactionScope class in the .NET framework. It is most useful via a using block that contains a set of changes. UndoBatch will start a new batch at the start of the using block and then close that batch at the end of the using statement, when the Dispose method is called.

UndoBatch supports nested usage, but will only ever start a single ChangeSet. This means that the top-most UndoBatch controls the batching boundary.

#### DefaultChangeFactory

DefaultChangeFactory is a static utility class that helps populate the undo system with ChangeSet and Change instances. The default implementation uses reflection to access the properties of a class.

### Example

        [TestMethod]  
        public void UndoRoot\_Supports\_Starting\_a\_Batch\_Of\_Changes()  
        {  
            var orig = Document1.A.Name;  
            var firstChange = "First Change";  
            var secondChange = "Second Change";  
            var root = UndoService.Current[Document1];  
   
            using (new UndoBatch(Document1, "Change Name", false))  
            {  
                Document1.A.Name = firstChange;  
                Document1.A.Name = secondChange;  
            }  
   
            Assert.AreEqual(1, root.UndoStack.Count());  
            Assert.AreEqual(0, root.RedoStack.Count());  
   
            root.Undo();  
            Assert.AreEqual(orig, Document1.A.Name);  
            Assert.AreEqual(0, root.UndoStack.Count());  
            Assert.AreEqual(1, root.RedoStack.Count());  
   
            root.Redo();  
            Assert.AreEqual(secondChange, Document1.A.Name);  
            Assert.AreEqual(1, root.UndoStack.Count());  
            Assert.AreEqual(0, root.RedoStack.Count());  
        }

## HomeMenu

### Overview

The following Diagram illustrates the HomeMenu features, when the user executes the following commands.

New: Create a new database

Open: Open a database.

Save: Save current database without asking.

Save As: Show a save as dialog.

Close: Close the database.

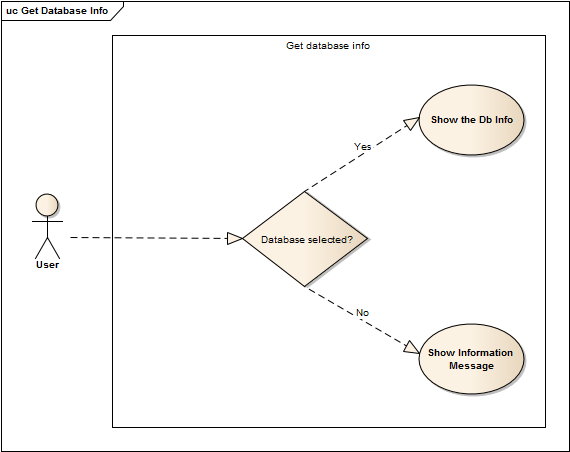
The HomeMenu offers more features likes:

Info: Show information about the current database.

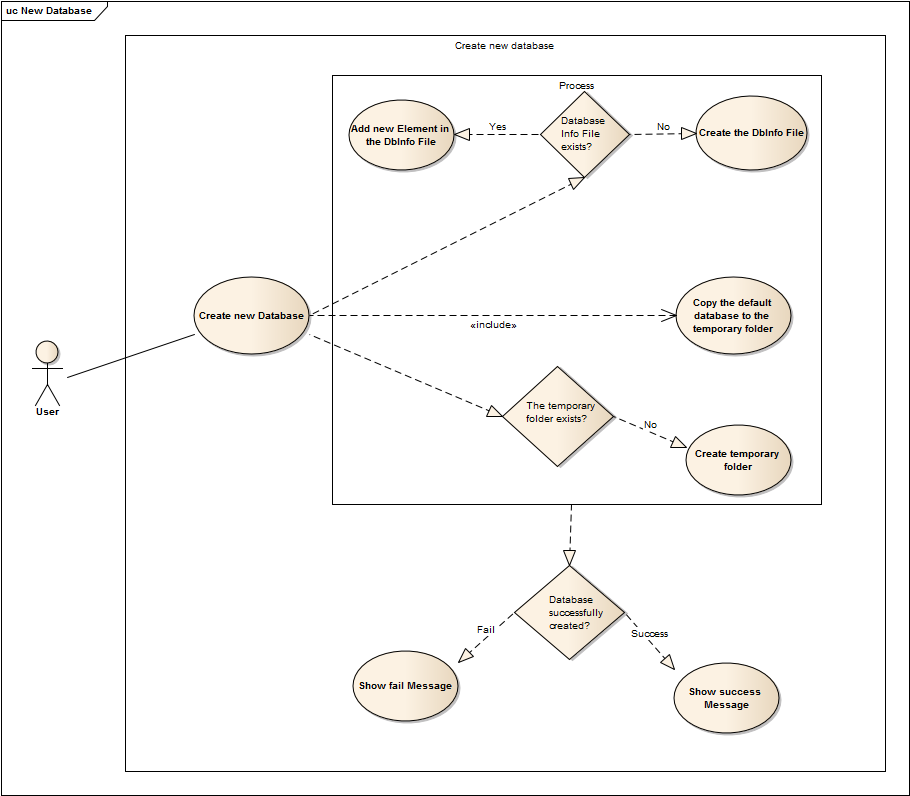
Recent: Show recent database without asking.

Options: Show the options pages.

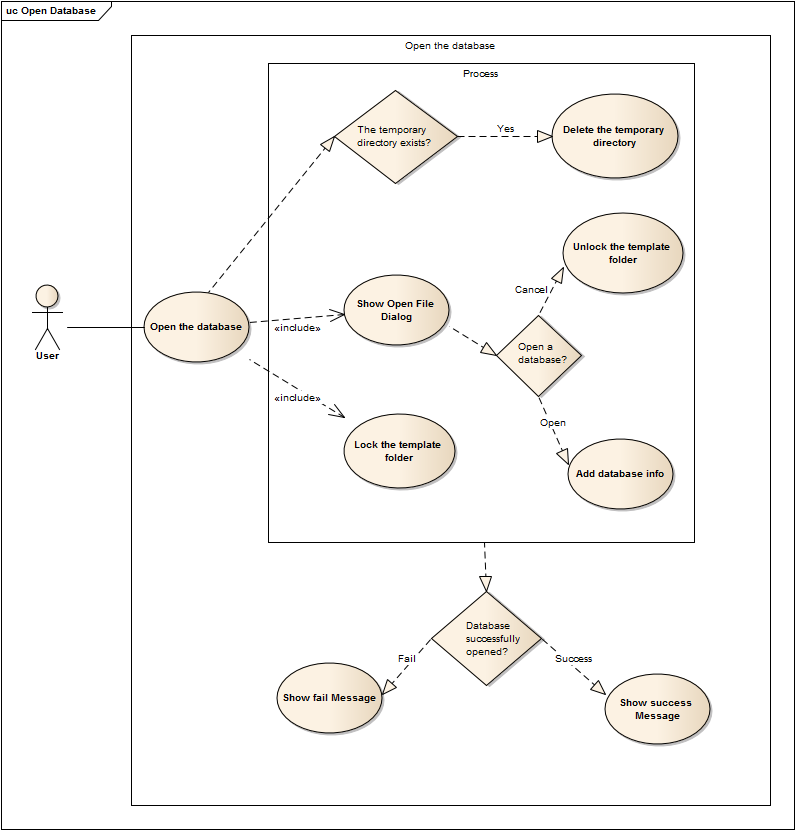
#### Show the database info



#### Create new database



#### Open the database



#### Save the database

#### Save As the database

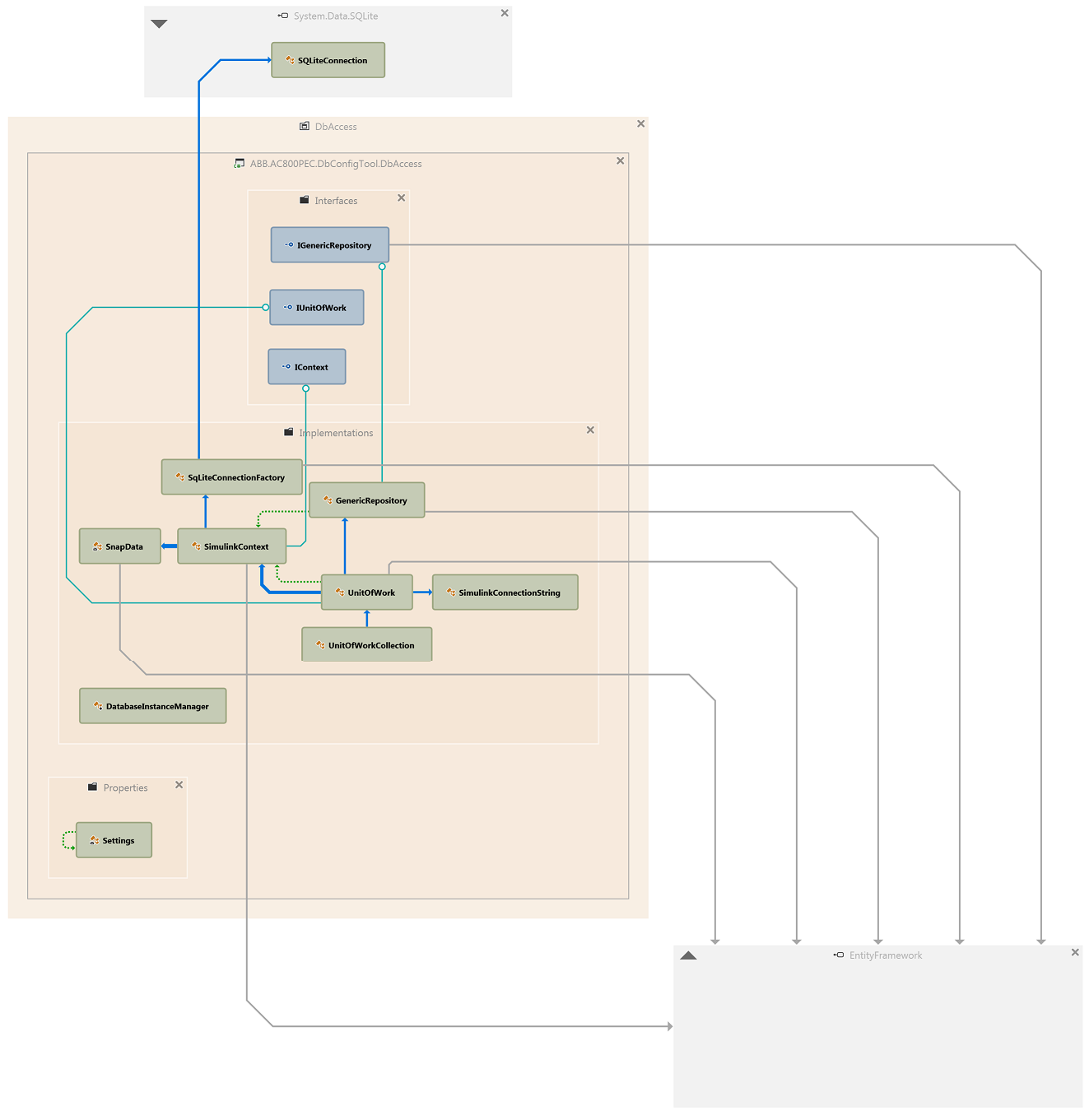
#### Close the database

# Logging

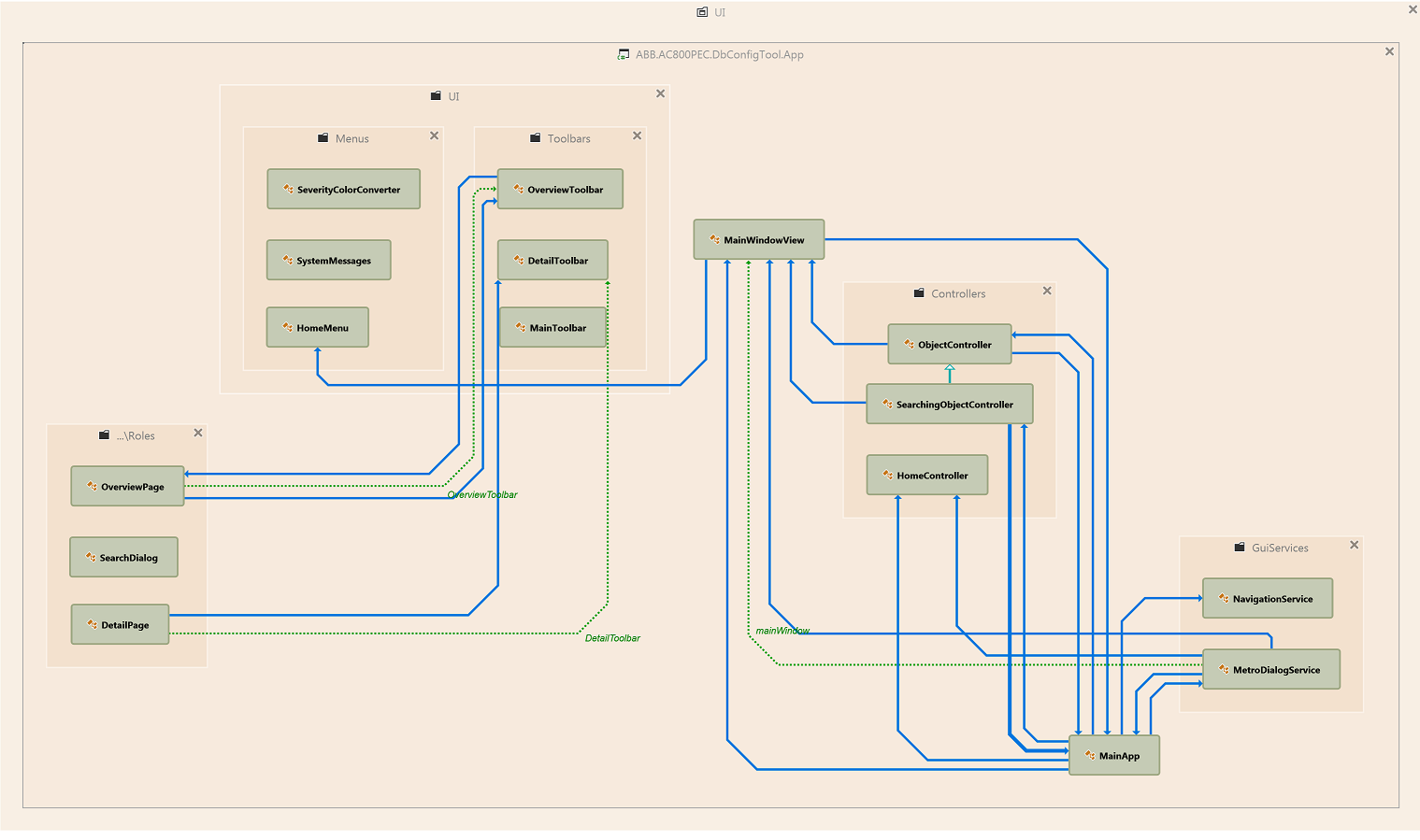
## Overview

# Project dependencies

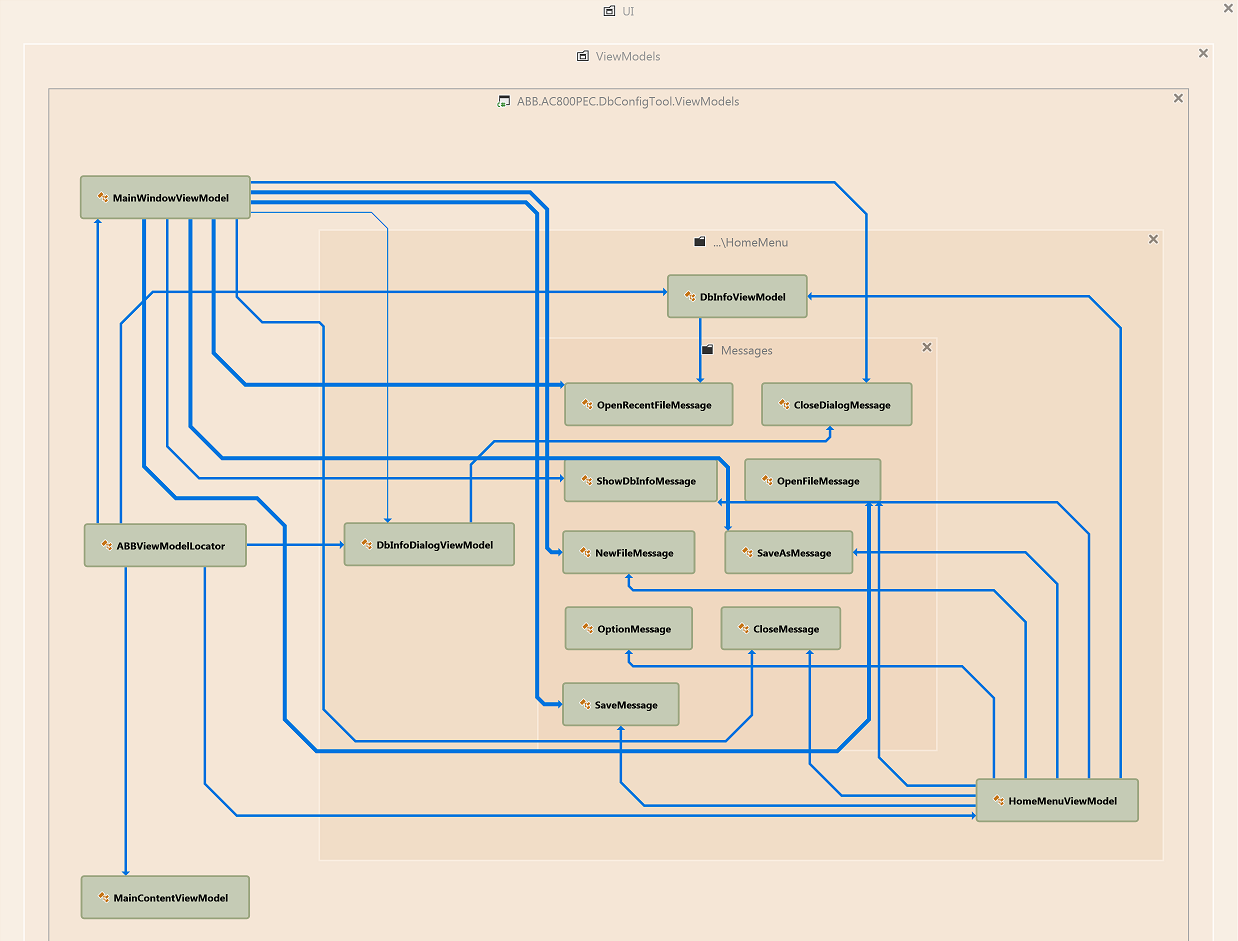
## DB Access Type Dependencies



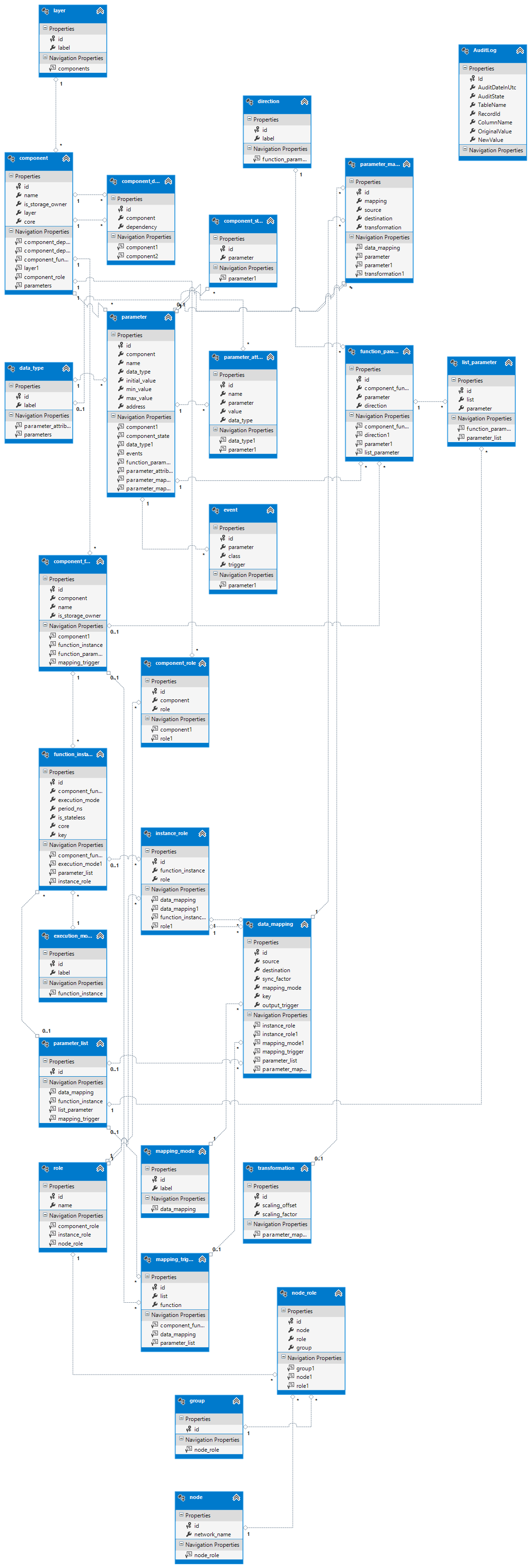
## UI Types Dependencies



## ViewModels Type Dependencies (Home Menu)



# Entity Designer Diagram



# Guard concept

We use the Guard concept in the application to specify and check the code’s behavior at Runtime. Guarding against passing invalid arguments into class methods.

For example:

var info = DatabaseInfoManager.GetCurrentDatabaseInfo();

Guard.ArgumentNotNull(info, "SimulinkConnectionString.info");

Guard.ArgumentNotNullOrEmpty(info.FileName, "SimulinkConnectionString.info.FileName");

Guard.ArgumentNotNullOrEmpty(info.Location, "SimulinkConnectionString.info.Location");

this.ConnectionString = string.Format(@"{0}{1}\{2}", "Data Source=", info.Location, info.FileName);

# References