User's Manual

.NetScaffolder

for Users

Prepared by: Laredo Tirnanic

October, 2018

Revision Sheet

Release No.	Date	Revision Description
Rev. 0	10/08/18	User's Manual Created

USER'S MANUAL

TABLE OF CONTENTS

1.0   GENERAL INFORMATION

1.0   GENERAL INFORMATION

1.1   System Overview

.NetScaffolder is a simple, highly flexible scaffolding framework based on T4 Scripts.

The Scaffolder Application is used to import and manage models. Model data can be imported from various sources. These sources can be databases or modelling tool files. Imported data is saved in a metadata file which is then used to generate the different layers.

The layers generated depend on the package selected. Packages consist of several templates. Templates use T4 files and metadata to generate a layer. Currently we are focusing on a domain driven implementation.

2.0   SYSTEM SUMMARY

2.0   SYSTEM SUMMARY

2.1   Import Sources

- Edmx Files (No index data is available)
- MySql Metadata (Version 5.7 - Known issues with version 8)
- Oracle Metadata
- Sql Server Metadata
- PostgreSQL (Planned)
- Extendable Custom sources

2.2   Data Layers

- Context (Entity Framework 6.0 / Entity Framework Core / NHibernate)
- Entities
- Repository Layer
- Application Service
- Dto

2.3   Supported Database Drivers

- Entity Famework 6.0 (MySql, Sql Server, Oracle, PostgreSQL)
- Entity Framework Core 2.1 (MySql - Pomelo Driver, Sql Server)
- NHibernate (Mysql, Sql Server, Oracle)

2.4   Planned Project Types

- MVC
- Angular
- Xamarin

3.0   GETTING STARTED

3.0   GETTING STARTED

3.1   Software Requirements

- Visual Studio
- T4Tollbox(https://marketplace.visualstudio.com/items?itemName=OlegVSych.T4Toolboxfor VisualStudio2017)

3.2   Downloading the source

Clone the source from Gitthub:
Git clone https://github.com/laredoza/.NetScaffolder.git.

Open /src/DotNetScaffolder.sln in visual studio and set DotNetScaffolder.Presentation.Forms as the starting project. Run the project.
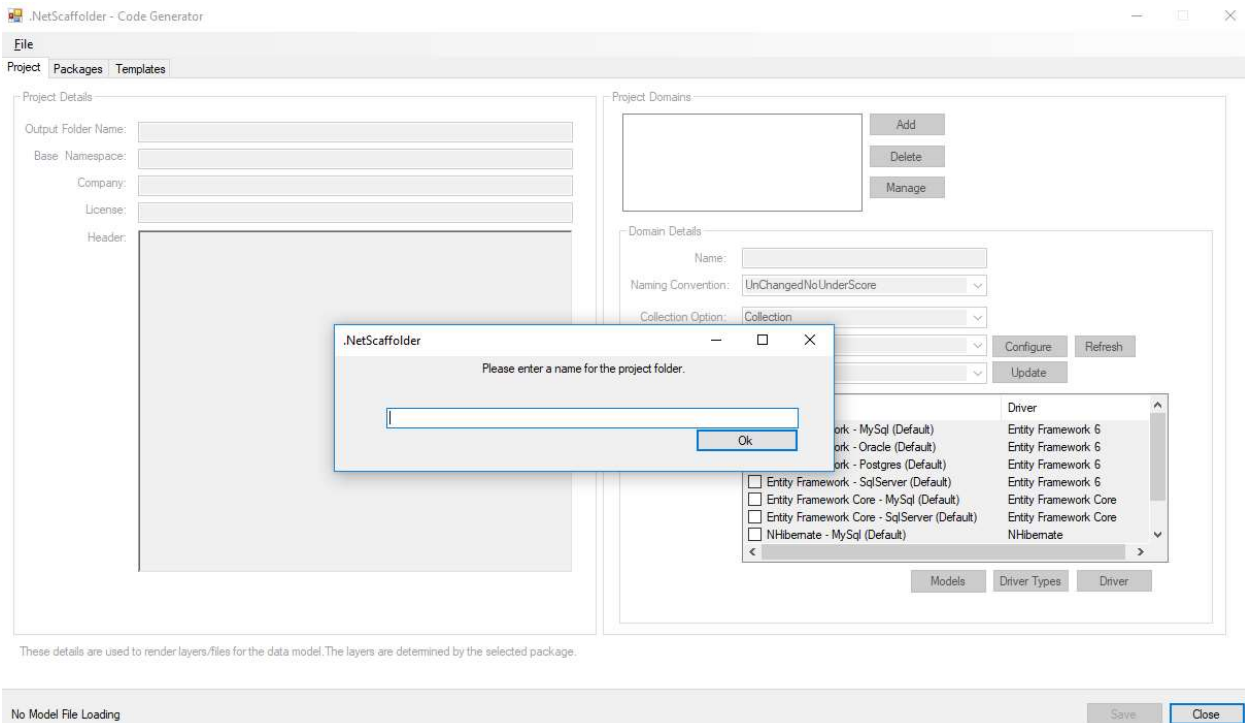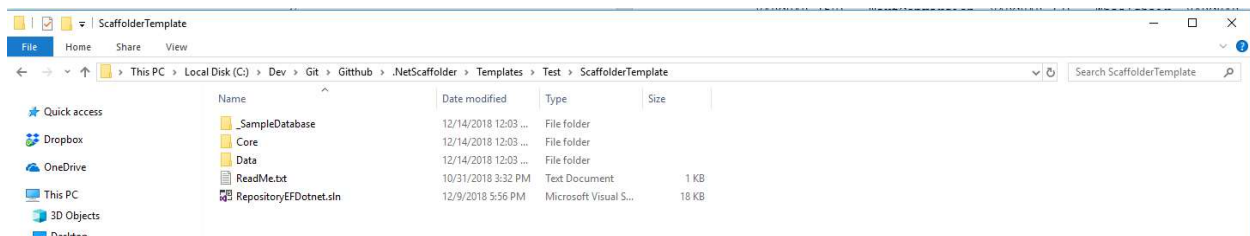
You should now see the following:
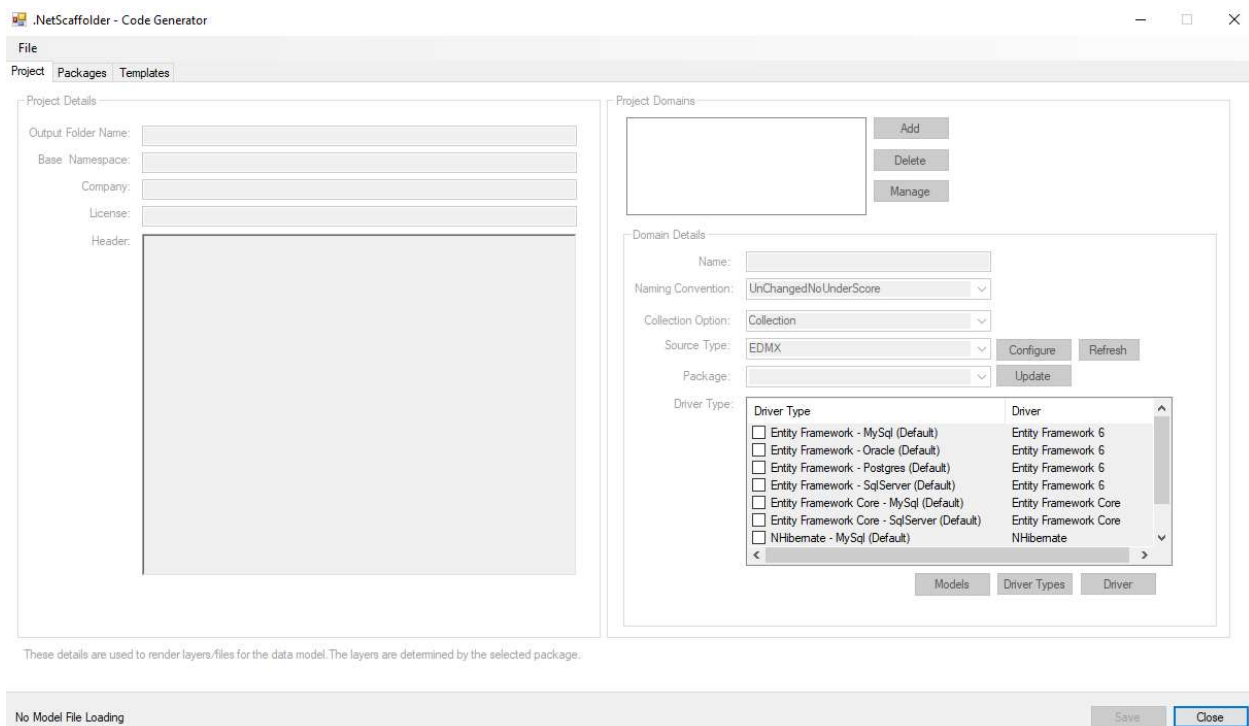
This is the Scaffolder application. It will be used to configure code generation.

3.3 Creating a new project

Select → File → New menu. You should see the following:

Supply a folder name. In our example we will be using "Test". Click OK.
This will create a solution in the templates folder (C:\Dev\Git\Gitthub\.NetScaffolder\Templates\Test)

The scaffolder application will have opened the model file:

You will now be able configure the layers to be generated.

3.4   Generating the Data Layer

Move the test folder to a folder within your own repository. Open the RepositoryEFDotnet.sln solution:

The next step is to open the RunGenerator.tt and save it by pressing Ctrl + S inside the file. This will generate the data layers. This will take a while. You will probably get a dialogue box which says "Microsoft Visual Studio 2017 has stopped working". Just click the "Close Program" button.
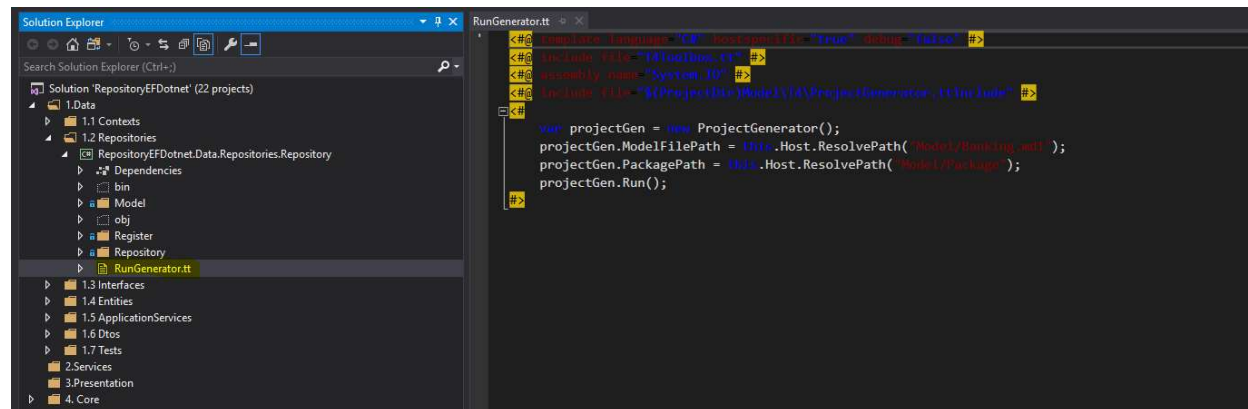
The model folder contains all the T4 files and configuration files required to generate the data layer.
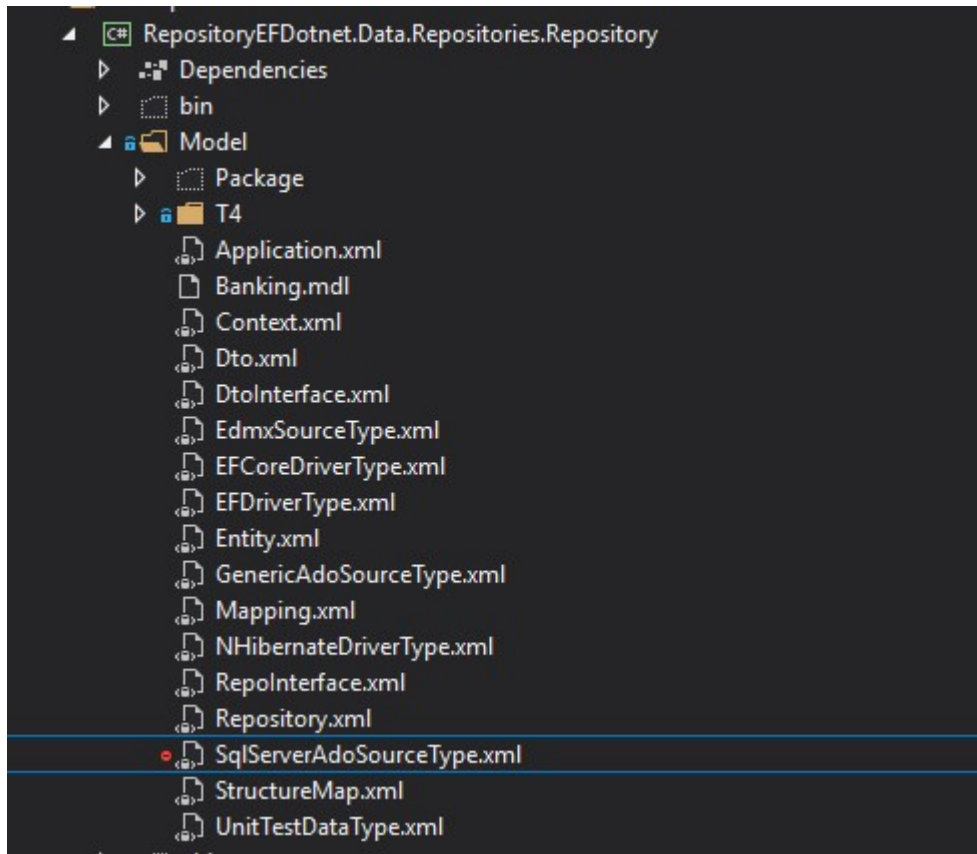
The package folder contains the dll's required to generate the data layers, the T4 folder contains all the T4's required and the rest of the files contain the configuration files. These can be managed with the Scaffolder applications. All of these will be discussed in more details later.

4.0   Managing Data Generation

4.0   MANAGING DATA GENERATION

4.1   Project

This is where code generation is configured. This includes the project details, project domains and
domain details. Models are also imported and updated here. Domains are a way to categories
functionality. Example domains would be Security, web, mobile, etc. In the example application there is
only one domain.

4.1.1 Project Details

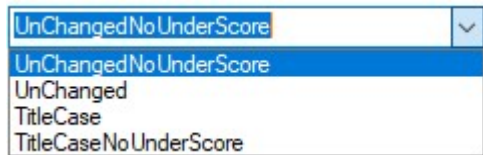This is where project details such as the Base Namespace, Company Name, License and header details are
configured.

4.1.2 Domain Details

This is used to specify which layers to generate (Packages) and how to generate them. Additionally models can be imported or updated here. Context layers (Drivers) are also specified here.

4.1.2.1 Naming Convention

There are four default naming conventions which are used to generate Classes and methods. If the UnchangedNoUnderScore convention was used on a model / table called "My_Config" an entity called MyConfig would be generated. These are Mef components which implement INamingConvention. It is easily extended.

```csharp
/// <summary>
///     The title case no under score naming convention.
/// </summary>
[Export(typeof(INamingConvention))]
[ExportMetadata("NameMetaData", "TitleCaseNoUnderScore")]
[ExportMetadata("ValueMetaData", "1BC1B0C4-1E41-9146-82CF-599181CE4411")]
public class TitleCaseNoUnderScoreNamingConvention : INamingConvention
{
    #region Public methods and operators

    /// <summary>
    /// Apply the naming convention.
    /// </summary>
    /// <param name="value">
    /// The value.
    /// </param>
    /// <returns>
    /// The <see cref="string"/>.
    /// </returns>
    /// <exception cref="NotImplementedException">
    /// </exception>
    public string ApplyNamingConvention(string value)
    {
        return Thread.CurrentThread.CurrentCulture.TextInfo.ToTitleCase(value).Replace("_", string.Empty);
    }

    #endregion
}
```

4.1.2.2 Collection Option

This specifies what type will be used to indicate a collection. An example of this usage would be in Entity generation. These are mef implementations of ICollectionOption. This is easily extendible

```csharp
/// <summary>
///     The collection and hash set collection option.
/// </summary>
[Export(typeof(ICollectionOption))]
[ExportMetadata("NameMetaData", "Collection")]
[ExportMetadata("ValueMetaData", "0BC1B0C4-1E41-9146-82CF-599181CE4410")]
public class CollectionAndHashSetCollectionOption : ICollectionOption
{
    #region Public Properties

    /// <summary>
    ///     The class name.
    /// </summary>
    public string ClassName => "HashSet";

    /// <summary>
    /// The class name interface.
    /// </summary>
    public string ClassNameInterface => "ICollection";

    #endregion
}
```

4.1.2.3 Source Type

This specifies what source you will be using for model generation. This can be used to initially import models or update them. As can be seen from the above example data sources can be metadata from databases or edmx's. These are mef implementations of `ISourcetype`. These are extendible.
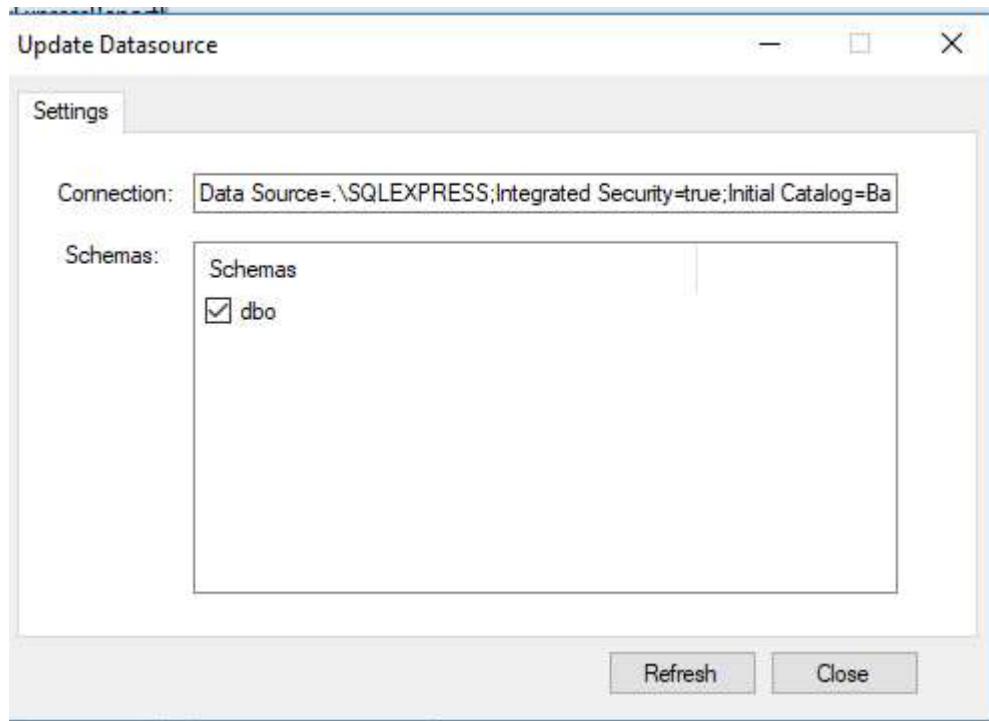

Restore the banking database supplied:
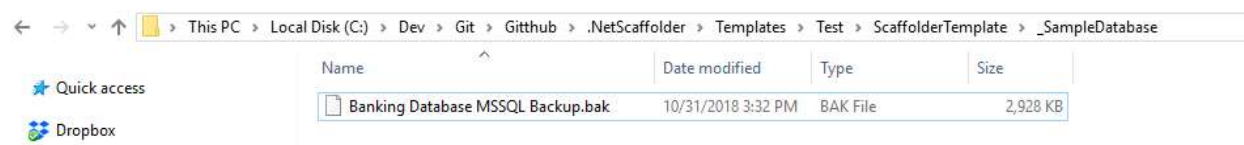
You will now have a design database called Banking.

Click the configure button. Change security settings and select schema/s. Close the page:

Open the refresh page:
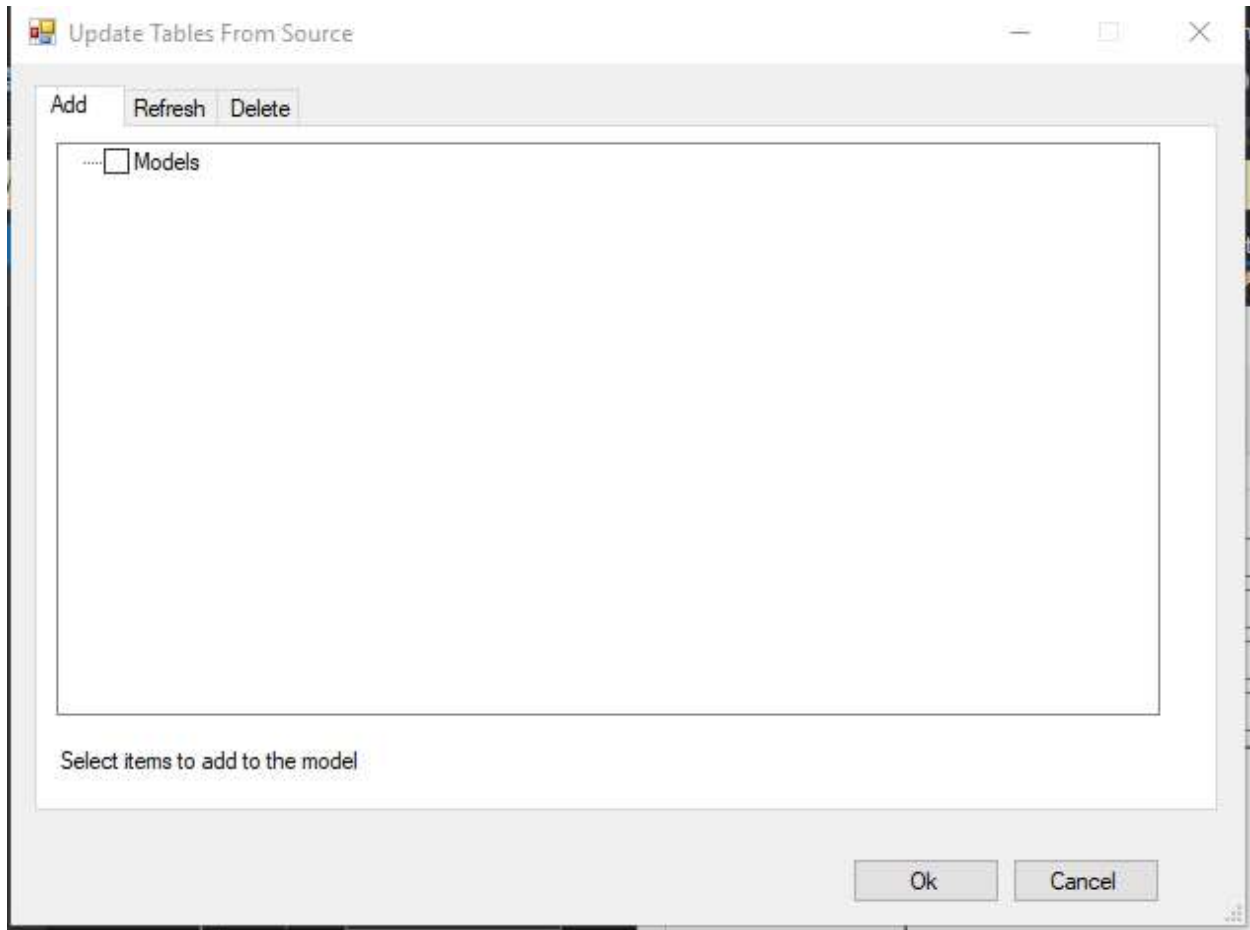
New tables are added to the add tab, updated tables will be found in the refresh tab and removed tables will be found in the deleted tab. Selecting any changes will cause the model to be updated. At the moment there are no changes, so just click the cancel button.

If tables have been selected in the Add treeview, they still have to be added to the context. This is done by opening the mange domains form. Selecting contexts and adding the models to the context there.

Update the Customer table in the Banking database by adding TestField : nvarchar(50):

Open the refresh page and you should see the following:

Select the Customer table and click ok. This change is now added to the Model.
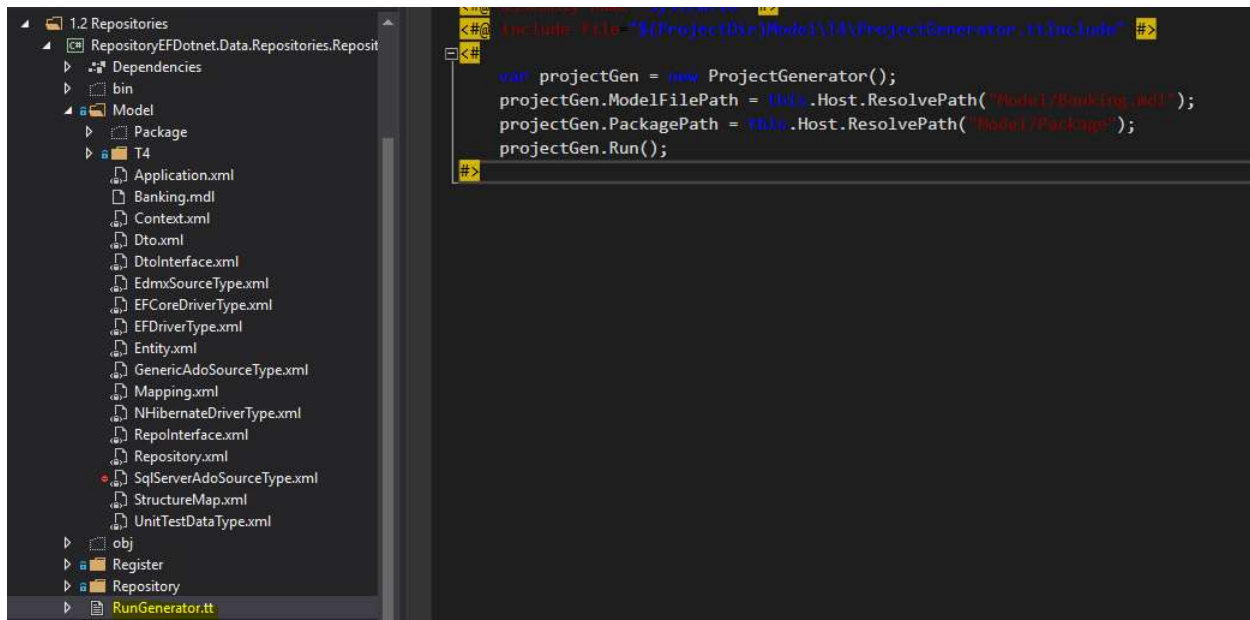
Open the Models page:

Close this page and click Save. This will update the Banking.mdl file.

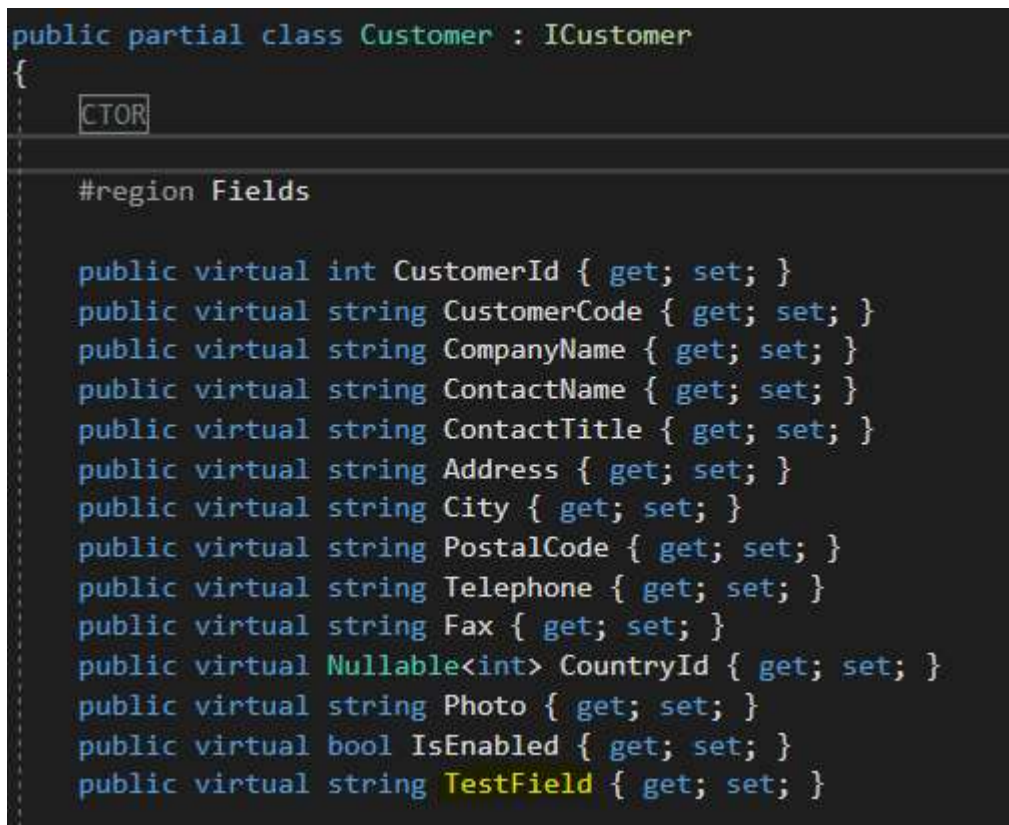Open the Customer.g.cs file in the repositoryEFDotnet solution:
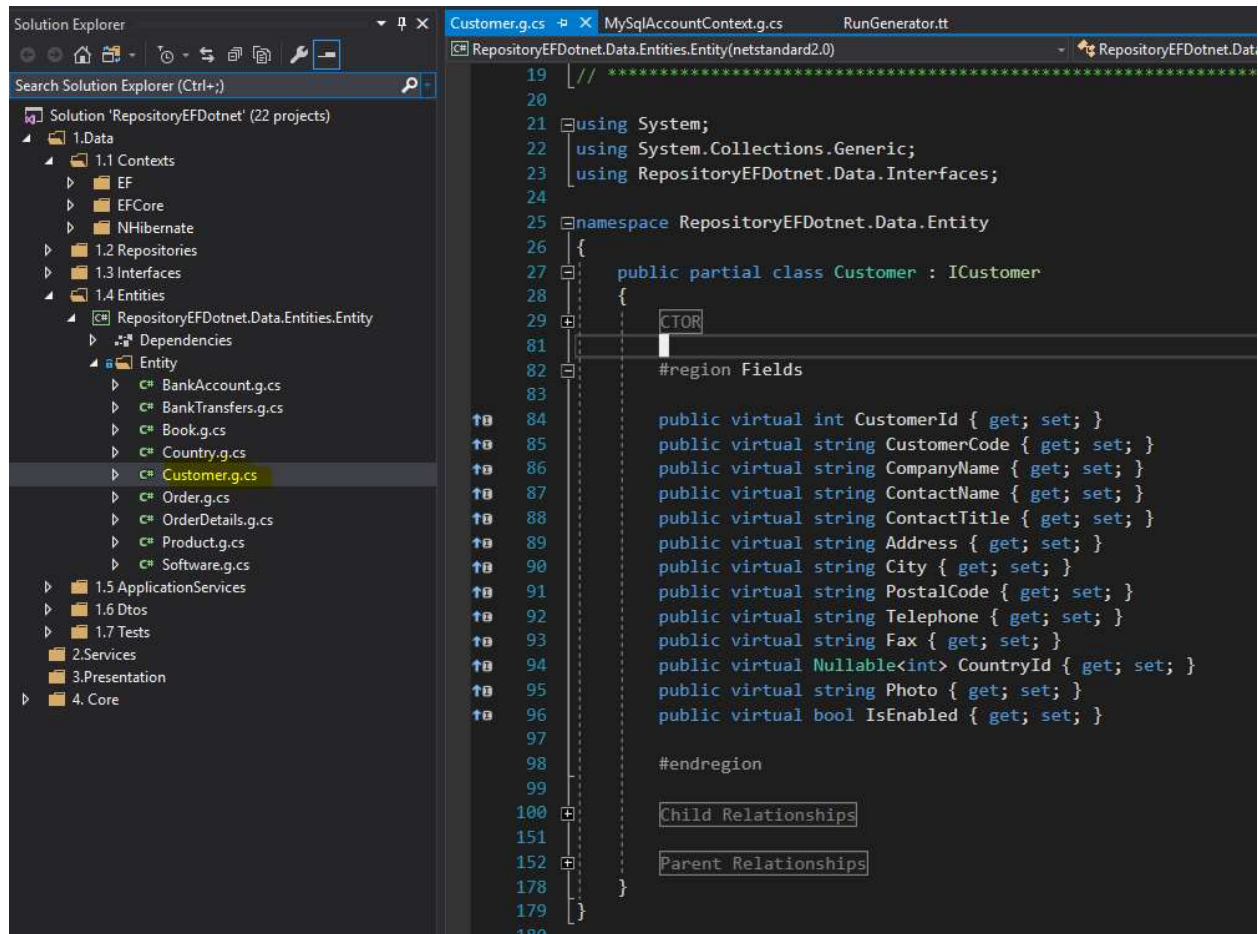
TestField is currently not defined in the Entity even though it's in Banking.mdl
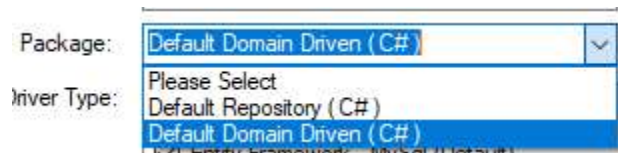
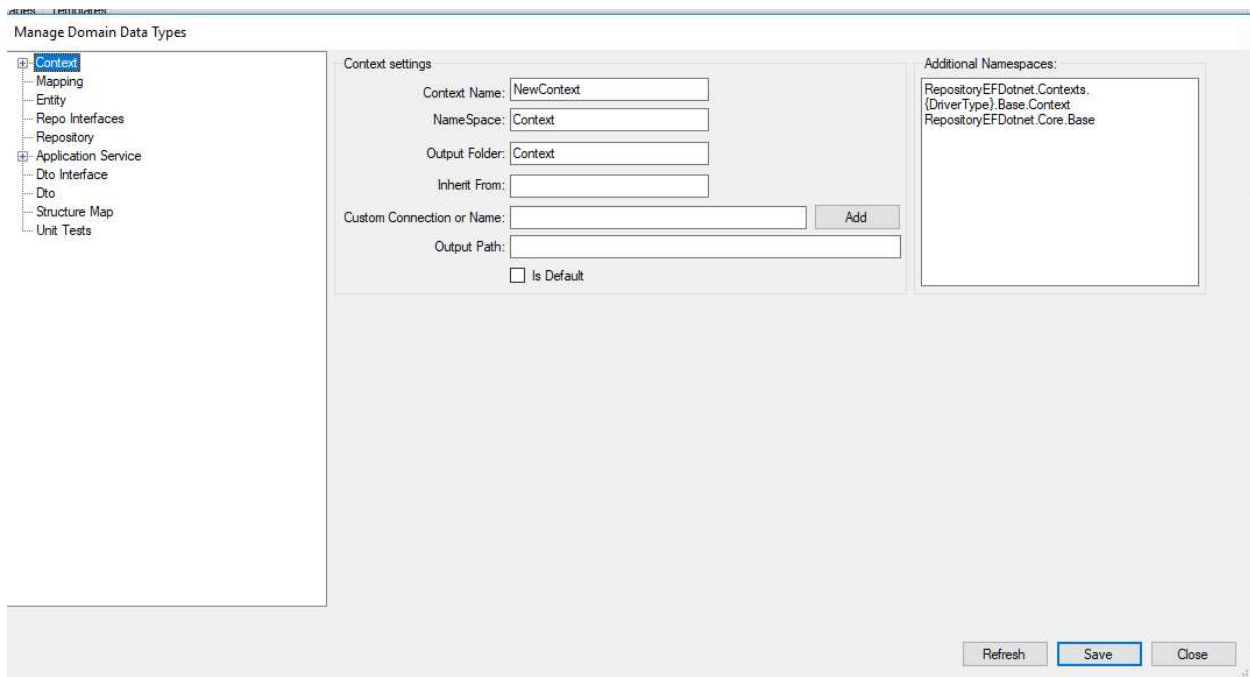Find the RunGenerator.tt and save it. This will update the generated files:

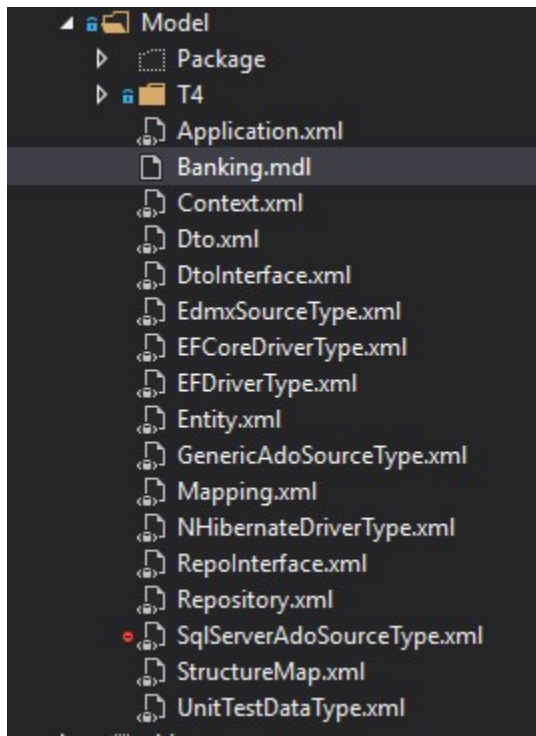Customer.g.cs should now look as follows:
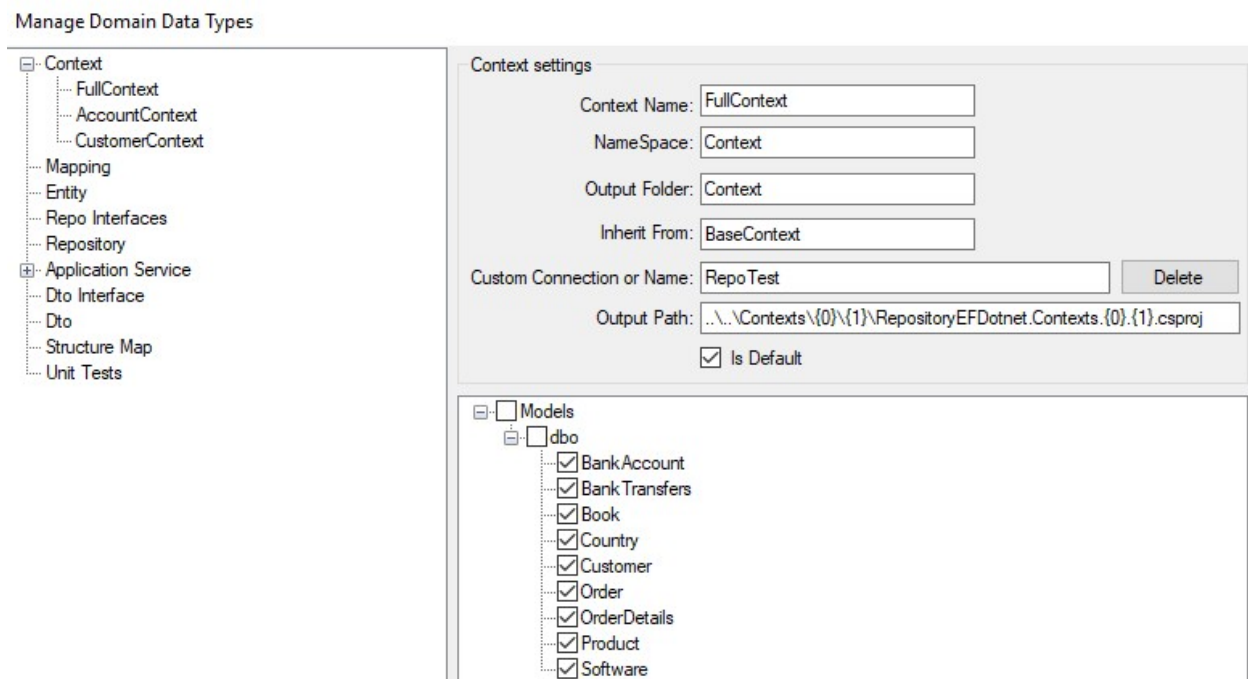
4.1.2.4 Selecting Packages

Packages consist of several templates. These templates use data in the model file to generate the required layers. Open the manage domain page to configure these templates.

Each template in a package has it's own configuration page. Each of these configurations are saved in separate files.
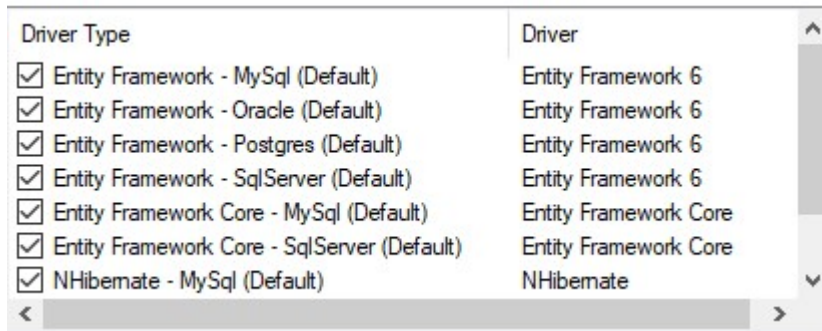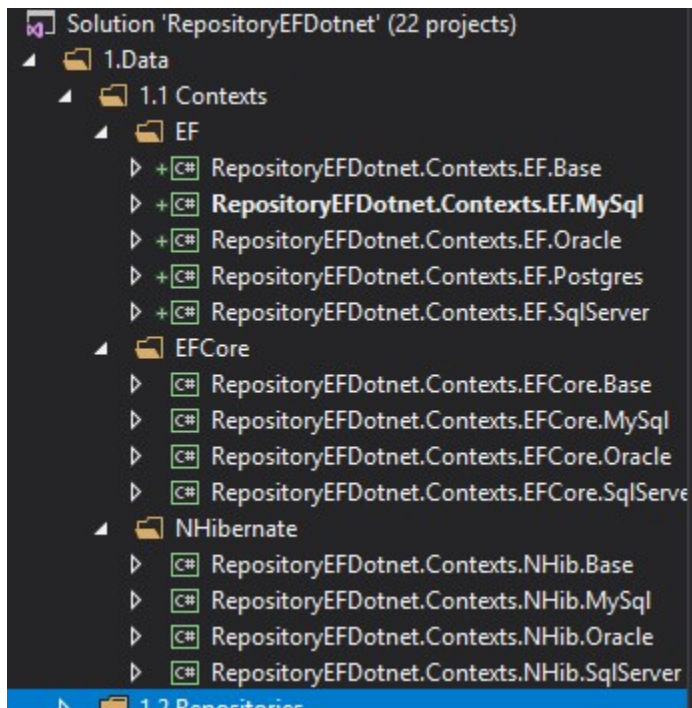

Example Context configuration:

How to extend these templates / DataTypes is discussed in the templates section.

4.1.2.5 Selecting Drivers

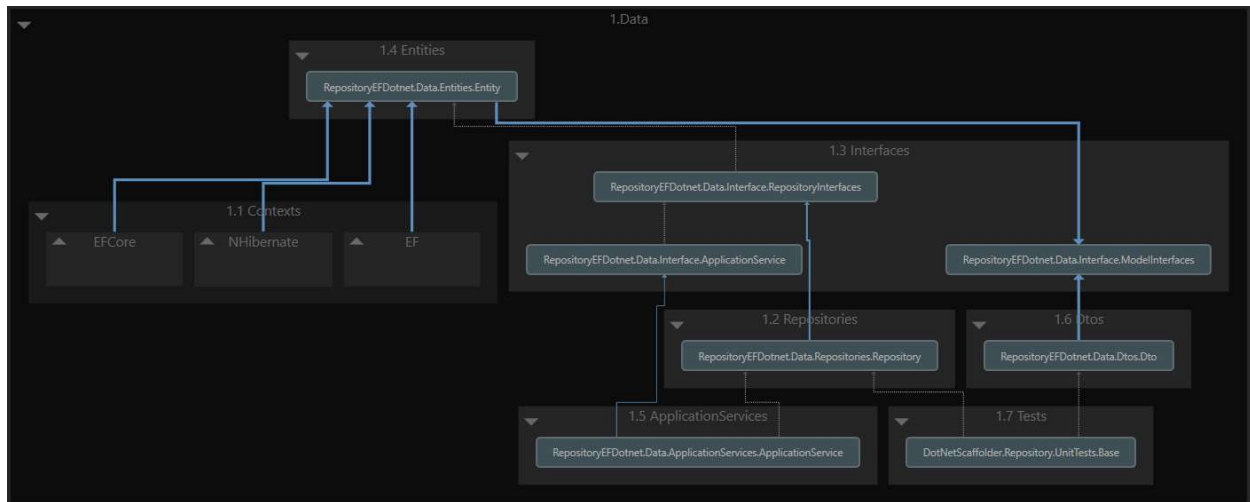Drivers are used to generate contexts. Each driver links to a context project in RepositoryEFDotnet:

Select required drivers and remove the rest.

5.0 Domain Driven

5.0   DOMAIN DRIVEN

5.1   Overview

Applications will use the data layer by invoking an application service. An application service will implement domain specific rules. Applications pass dto's to application service methods, and vica versa. These are converted to entities which are then passed to repositories. Repositories use the specified context / unit of work to do the actual database specific function. By swapping out the context / unit of work you could change your target database / orm without changing any of your other code.