Folke.Elm Documentation Documentation

Release 1.1.0-beta

The Folke Team

Nov 29, 2017

Contents

Welcome to Elm's documentation. Elm is an Object-Relational Mapping (ORM) library written in C#. It's free, open-source and works with .NET 4.5 and .NET Core. Elm supports MySQL, PostgreSQL, Microsoft SQL Server and SQLite. It supports the Fluent syntax. Honestly, it's pretty awesome.

This project uses the standard MIT License.

- Project page
- File a bug

CHAPTER 1

Content:

1.1 Quickstart

This page will explain how to quickly start using Elm. For this we'll build a sample Console Application. You can find the source of this application on GitHub: https://github.com/folkelib/Folke.Elm.Demo

We'll use .NET Core for this Console Application and Visual Studio 2015. Feel free to use any other editor but Visual Studio makes it easy to scaffold the application. Go ahead and create an Empty Console (Package) application.

1.1.1 Package References

Once this is done, the very first thing to do is to add the Folke.Elm packages into the `project.json` file. Elm supports both .NET 4.5 and .NET Core but in that sample we'll use only .NET Core. Add the references to Elm into the appropriate section. In this sample we'll use SQLite but feel free to use a different database driver.

We will also need a few more packages for later in the quickstart:

- *System.ComponentModel.Annotations*
- *System.Reflection*

The JSON file should look like this (for the package versions, just use whatever the latest version is:

```
{
 "version": "1.0.0-*",
 "buildOptions": {
   "emitEntryPoint": true
 },

 "dependencies": {
   "Microsoft.NETCore.App": {
     "type": "platform",
     "version": "1.0.0-rc2-3002702"
   }
 },
```

```
 "frameworks": {
   "netcoreapp1.0": {
     "imports": "dnxcore50",
     "dependencies": {
       "Folke.Elm": "1.2.0-beta-23380600",
       "Folke.Elm.Sqlite": "1.2.0-beta-23380600"
     }
   }
 }
```

}

Obviously the various packages version might change as we publish newer versions. In our example we used the SQLite driver for two reasons:

- as of today Oracle hasn't released a MySQL driver that works under .NET Core
- not everybody wants to run code on Windows/Azure or Microsoft SQL Server

SQLite for its part will work with .NET Core (which means it'll run natively on Windows, Linux and Mac) or with .NET 4.5 (which means it'll work natively on Windows and with Mono on Linux and Mac). In any case, Elm is built in such a way that it's really, really easy to switch from one driver to the other and this documentation will indicate where.

1.1.2 Demo Classes

For the purposes of this sample application we'll use two classes: `Product` and `Order`. As you can imagine, we'll create fake Products and then generate some fake Orders containing these fake Products. Note that for the sake of simplicity these classes will be created in the same project as the Console Application, but you are of course completely free to create them in a separate project (and you probably should).

Here's the first class, `Product.cs`:

```
namespace Folke.Elm.Demo.Data
{
    public class Product: IFolkeTable
    {
        public int Id { get; set; }
        public string PartNumber { get; set; }
        public string Name { get; set; }
        public string Price { get; set; }
        public string Description { get; set; }
        public string Comment { get; set; }
    }
}
```

There are two things your class must have. It must implement the `IFolkeTable` interface, and for that it must have an `Id` property. This property can be an Integer (for typical database primary key usage) or it can also be a String (for instance if you want your Primary Key to be a GUID). For the rest, you are free to create the fields you want. Elm also supports Annotations and will use them as instructions when creating the database schema. For instance, let's say we want to limit the size of the PartNumber to 36 characters. We just need to modify the class to this:

```
using System.ComponentModel.DataAnnotations;

namespace Folke.Elm.Demo.Data
{
```

```
    public class Product: IFolkeTable
    {
        public int Id { get; set; }
        [MaxLength(36)]
        public string PartNumber { get; set; }
        public string Name { get; set; }
        public string Price { get; set; }
        public string Description { get; set; }
        public string Comment { get; set; }
    }
}
```

Note that we had to add the `System.ComponentModel.DataAnnotations` namespace. Make sure to add this package to the .NET Core dependencies (.NET 4.5 doesn't need this to be explicitely added since it's part of that framework). There are many more supported *Annotations*.

The second class is `Order.cs` and looks like this:

```
using System.Collections.Generic;

namespace Folke.Elm.Demo.Data
{
    public class Order: IFolkeTable
    {
        public int Id { get; set; }
        public IList<Product> Products { get; set; }
    }
}
```

The `Products` property is a list of `Product` objects. There can be an arbitrary amount of those in the list. In the database, this will create a relational table. Automatically. No XML or mapping to write, Elm takes care of all that for you!

Now that we have a couple of classes, let's setup the rest of the application.

1.1.3 Creating the database

For this setp we'll put all of the database creation in one big method. There are better ways to do this, by ways of Dependency Injection for instance. Read the dotnetcore page for more details on this.

The object type for establishing connections (or you could call them *sessions* as well) is `FolkeConnection`. This class has a `Create()` method that you can use as a factory.

```
FolkeConnection.Create(
        IDatabaseDriver databaseDriver,
        IMapper mapper,
        string connectionString);
```

Since we use SQLite in this example the session will be created like this:

```
IFolkeConnection session = FolkeConnection.Create(
                                        new Sqlite.SqliteDriver(),
                                        new Mapping.Mapper(),
                                        "Data Source=test.db");
```

Then we need to tell that session to update the schema (this method will create it instead if it doesn't exist).

```
session.UpdateSchema(typeof(Product).GetTypeInfo().Assembly);
```

Here we just specify the Assembly from one of our Classes. Elm will automatically read that Assembly and create the schema appropriately.

1.1.4 Saving to the database

Now we need an object to save into the database. Let's start with a Product.

```
Product product = new Product
        {
            Comment = "Awesome product",
            Description = "This product is awesome, trust us, we just want to sell it␣
↪to you.",
            Name = "Awesome-O",
            PartNumber = "AWE-SOME-O1",
            Price = "10000"
        };
```

And now to save it, we only need to create a transaction.

```
using (var t = session.BeginTransaction())
        {
            session.Save(product);
            t.Commit();
        }
```

And that's it, the product is saved!

Now we can create an `Order` object that references that product.

```
Order order = new Order();
order.Products.Add(product);

using(var t = session.BeginTransaction())
{
    session.Save(order);
    t.Commit();
}
```

1.1.5 Reading from the database

Now that the `product` object has been saved into the database, its relevant properties have been automatically updated. In our case, the `Id` has been set. We can access it directly in the follow up code.

```
Console.WriteLine("New product Id is {0}", product.Id);
```

But most of the time the object will be saved in a different scope than the one you want to read it from. In that case, there are two methods you can use: `IFolkeConnection.Get<ObjectType>(Id)` or `IFolkeConnection.Load<ObjectType>(Id)`. The only difference between these two methods is that `Load()` will throw an error if the object cannot be found in database whereas `Get()` will return `null`.

```
// Returns the correct product
Product loadedProduct = session.Load<Product>(product.Id);
```

```
 // Returns the correct product
Product getProduct = session.Load<Product>(product.Id);

// Returns an error
Product loadedProduct = session.Load<Product>(2323);

// Returns a null object
Product getProduct = session.Load<Product>(2323);
```

1.1.6 Selecting from the database

Elm supports the Fluent syntax for selecting from the database. You can apply any number of filters. Make sure to add the `using Folke.Elm.Fluent;` statement. For instance, to select all the Products you would do this:

```
var Products = session.SelectAllFrom<Product>().ToList();
```

We can order them, for instance by `Id` to get them by chronological order.

```
var Products = session.SelectAllFrom<Product>()
                        .OrderBy(x => x.Id).Desc
                        .ToList();
```

Maybe we want to select only the `Id` and `PartNumber` fields.

```
var Products = session.SelectAllFrom<Product>(x => x.Id, x => x.PartNumber)
                        .OrderBy(x => x.Id).Desc
                        .ToList();
```

1.2 Dependency Injection

1.2.1 Registering the Service

Elm supports Dependency Injection out of the box and it's fairly simple to set up. In this page we'll use an ASP.NET Core WebApi/MVC6 project as an example but this would work with a Console project as well.

In your `Startup.cs` file you should have a method to configure services. The default Microsoft scaffolding creates it as `public void ConfigureServices(IServiceCollection services).`

To add Elm simply add this line:

```
services.AddElm<MySqlDriver>(options => options.ConnectionString = Configuration[
↪"Data:ConnectionString"]);
```

This assumes you want to use MySQL/MariaDb. Replace `MySqlDriver` by `SQLiteDriver` or `SqlServerDriver` depending on your requirements. This also assumes that your ConnectionString is stored in a JSON file that was provided to the `ConfigurationBuilder` in the `Startup` method, like this:

```
public IConfigurationRoot Configuration { get; set; }
    public Startup(IHostingEnvironment env)
    {
        var configurationBuilder = new ConfigurationBuilder();
        configurationBuilder
            .AddJsonFile("config.json")
```

```
            .AddEnvironmentVariables();
        configurationBuilder.AddCommandLine(new string[] { });
        Configuration = configurationBuilder.Build();
    }
```

You will of course need to use the relevant namespaces, in our cases they would be:

```
using Folke.Elm;
using Folke.Elm.Mysql;
```

Now that the service is registered, let's see how to setup a Controller to use it.

1.2.2 Use the Service

In your Controller, simply specify in the constructor that you want an `IFolkeConnection` object. Create a Property for it and associate it.

```
[Route("api/[controller]")]
public class MyController : Controller
{
    protected readonly IFolkeConnection session;
    public MyController(IFolkeConnection session)
    {
        this.session = session;
    }
    [HttpGet("{id}")]
    public async Task<IActionResult> Get(int id)
    {
        Product myProduct = session.Load<Product>(id);
        return Ok(myProduct);
    }
}
```

1.3 Annotations

Work in progress. Sorry.

1.4 License

1.5 Contact

If you have any question about this project, please contact the maintainers on the project's Github repository.