**<https://medium.com/@taublast/entity-framework-with-code-first-migrations-in-net-maui-3efbdb765592>**

**Entity Framework With Code-First Migrations in .Net Maui**

**[[Nick Kovalsky](https://medium.com/@taublast?source=post_page-----3efbdb765592--------------------------------)](https://medium.com/@taublast?source=post_page-----3efbdb765592--------------------------------)**

[**Nick Kovalsky**](https://medium.com/@taublast?source=post_page-----3efbdb765592--------------------------------)

By Nick Kovalsky



**Onboarding**

If you are a fan like me of EF and would like to use it in your mobile app, my guess is with[**.Net Maui**](https://dotnet.microsoft.com/en-us/apps/maui) on the market it’s a great time to start.

Just a small reminder, for better app startup time it might be better to store boostrap data to the mobile device local storage in a json form. And when it comes to managing big local data with filters, ordering and such, EF is definitely a way to go.

This article’s goal is to help one avoid all the hassle of looking for different solutions to small problems when implementing a production-ready mobile local database, and creating code-first migrations for it on both **Windows** and **Mac** machines. I invite you to browse and to reuse this article sample source code, you will find the link at the end. As you will see it is a Maui App template with EF database logic added.

The proven standart for a mobile client database is SQLite. We’ll instantly look for *Microsoft.EntityFrameworkCore.Sqlite* nuget package to install, along with *SQLitePCLRaw.bundle\_e\_sqlite3* for native sqlite implementations. To create EF migrations we’d also need to install *Microsoft.EntityFrameworkCore.Tools*.

Things are that we cannot install these nugets into our single maui project because if we then try to create migrations we’d get a nice

*Startup project ‘MauiEF.Client’ targets platform ‘Android’. The Entity Framework Core Package Manager Console Tools don’t support this platform. See*[*https://aka.ms/efcore-docs-pmc-tfms*](https://aka.ms/efcore-docs-pmc-tfms)*for more information.*

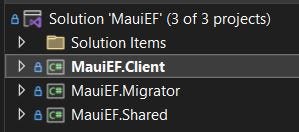
..so we’d need to find a way to work around it.

For those who are not sure what migrations are for, it’s basically the code that instructs EF of database structure to create and use. The important benefit here is that when you change your models you would just have to auto-generate additional migrations for EF to modify the existing database accordingly, adding-removing tables, columns etc.

**Setup Projects**

Now that we know that EF design tools have unsupported frameworks, we would have to create a special “Migrator” project directly runnable by EF and targeting pure *net-7.0*, and we will be able to create migrations on a windows or an apple machine.

Both *Migrator* and our *Client* would need to access the database, so we’ll move all our context-related code into a separate *Shared* project. Our solution structure would then look like this:



The *Shared* project would reference

<!--Local database-->  
<PackageReference Include="Microsoft.EntityFrameworkCore.Sqlite" Version="7.0.1" ></PackageReference>  
<PackageReference Include="SQLitePCLRaw.bundle\_e\_sqlite3" Version="2.1.3" ></PackageReference>

while *Migrator* would need additional tools to create migrations:

<PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="7.0.1">  
 <PrivateAssets>all</PrivateAssets>  
 <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>  
</PackageReference>

**Create Database**

We will define our example context inside the *Shared* project as follows:

*LocalDatabase.cs*

**///** **<summary>**  
 **///** Constructor for creating migrations  
 **///** **</summary>**  
 public LocalDatabase()  
 {  
 File = Path.Combine("../", "Data1.db3");  
 Initialize();  
 }  
 **///** **<summary>**  
 **///** Constructor for mobile app  
 **///** **</summary>**  
 **///** **<param name="filenameWithPath"></param>**  
 public LocalDatabase(string filenameWithPath)  
 {  
 File = filenameWithPath;  
 Initialize();  
 }  
 void Initialize()  
 {  
 if (!Initialized)  
 {  
 Initialized = true;  
 SQLitePCL.Batteries\_V2.Init();  
 Database.Migrate();  
 }  
 }  
 protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)  
 {  
 optionsBuilder  
 .UseSqlite($"Filename={File}");  
 }

Notice two constructors here, one for the EF migrator and one for our app consumption. *Database.Migrate();* creates the database files if it doesn’t exist and applies provided migrations.

You might maybe want to implement the *Database.EnsureDeleted();* method for debug purposes, to wipe out the data at start.

In case you have a breaking app change you might also want change the database filename to recreate the db from scratch for existing users.

OK, now we can inject the context inside our *MauiProgram.cs*:

builder.Services.AddTransient<LocalDatabase>((services) =>  
 {  
 return new LocalDatabase(Path.Combine(FileSystem.AppDataDirectory, "SQLite001.db3"));  
 });

**Create Migrations**

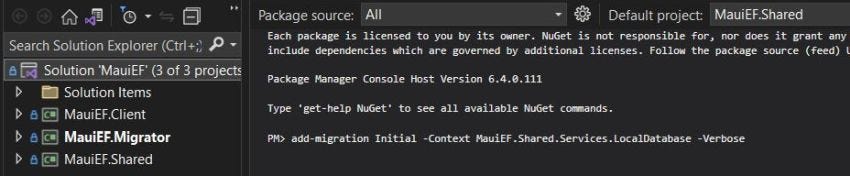
The provided source code doesn’t include migrations. If you just compile and run the solution it will throw an exception, as EF would not know how you what it to *Migrate();*.

But please don’t worry migrations are very easy to create.

If you are using **Visual Studio for Windows** :

1. Change your startup project from *Client* to *Migrator*.
2. Open Package Manager Console: go to View->Other Windows->Package Manager Console.
3. Set the default project to *Shared*, EF will look for the context model inside and add migrations there.
4. Enter the following command to create the initial migration:

add-migration Initial -Context MauiEF.Shared.Services.LocalDatabase -Verbose



The next method via command like will be illustrated using **Visual Studio for Mac**.

1 Open console: Right-click your solution name, then select Open In Terminal. You should land inside the solution folder.

2 Enter the following command to create the initial migration via command line:

dotnet ef migrations add Initial -s Migrator -p Shared -c MauiEF.Shared.Services.LocalDatabase

You will notice that we specified our startup project subfolder *-s Migrator* and default project subfolder *-p Shared*.

If you encounter an error due to ef command not present, install it and add it to global path:

dotnet tool install --global dotnet-ef  
export PATH="$PATH:/Users/YOUR\_USERNAME/.dotnet/tools"

Every time you change your context models you’d have to create additional migrations. This way your app won’t break, if you release a new version with modified migrations EF will (most probably) keep the user’s existing database and modify it according to the new scheme upon initialization of your context model. Why most probably? Because you might change your models in a way a foreign key relation or property will be lost, in this case EF will warn you when creating a migration with something like “Warning data will be lost”, so it’s all under your control, to manage the existing users old apps versions data.

To create a new migration just create a unique name for it (**Visual Studio for Windows** example):

add-migration Change1 -Context MauiEF.Shared.Services.LocalDatabase -Verbose

**Sample App**

We can now compile our sample and run, to have user-created data to be persistent between app launches. Context operations are executed asynchronously, so we don’t block the UI thread.

*MainPage.cs*

public MainPage()  
 {  
 \_context = App.Services.GetService<LocalDatabase>();  
 InitializeComponent();  
 var mainAuthor = \_context.Authors  
 .Include(i => i.Books)  
 .FirstOrDefault(x => x.FirstName == "John" && x.LastName == "Doe");  
 if (mainAuthor == null)  
 {  
 Task.Run(async () =>  
 {  
 mainAuthor = new Author()  
 {  
 FirstName = "John",  
 LastName = "Doe"  
 };  
 \_context.Authors.Add(mainAuthor);  
 await \_context.SaveChangesAsync();  
 \_author = mainAuthor;  
 Update();  
 }).ConfigureAwait(false);  
 }  
 else  
 {  
 \_author = mainAuthor;  
 Update();  
 }  
 }  
 private void OnCounterClicked(object sender, EventArgs e)  
 {  
 count++;  
 var title = $"My Story Part {count}";  
 var book = \_author.Books.FirstOrDefault(x => x.Title == title);  
 if (book == null)  
 {  
 CounterBtn.Text = $"Wrote \"{title}\"";  
 Task.Run(async () =>  
 {  
 \_author.Books.Add(new Book  
 {  
 Title = title  
 });  
 \_context.Authors.Update(\_author);  
 await \_context.SaveChangesAsync();  
 Update();  
 }).ConfigureAwait(false);  
 }  
 else  
 {  
 CounterBtn.Text = $"Reading \"{title}\"";  
 }  
 SemanticScreenReader.Announce(CounterBtn.Text);  
 }

**Final Words**

When you compile your first EF Maui app for iOS Release it could crash at runtime on real device, due to the fact that iOS AOT compilation doesn’t support some EF techniques. I wouldn’t be more precise here, you can [read more about it](http://github.com/xamarin/xamarin-macios/issues/16228), but the remedy is to add some flavor into your .csprj file for that specific case:

<!--IOS RELEASE-->  
<PropertyGroup Condition="'$(Configuration)|$(TargetFramework)|$(Platform)'=='Release|net7.0-ios|AnyCPU'">  
 <!--to be able to use some EF core methods-->  
 <MtouchExtraArgs>--interpreter</MtouchExtraArgs>  
 <UseInterpreter>True</UseInterpreter>  
 <!--your codesign parameters will go below-->  
</PropertyGroup>

Apps compiled with such settings already have been approved to AppStore and no performance impact has been reported.

I hope you would find this to be useful, please feel free to ask questions if any: [@nickkovalsky](https://twitter.com/nickkovalsky)

Source code: <https://github.com/taublast/MauiEF>

