# Using Dependency Injection, Portable SQLite, and TinyIoC for your cross-platform Xamarin app

One of the huge advantages of Xamarin for mobile development is the ease with which C# lends itself to dependency injection techniques.

With easy to use dependency injection techniques, coupled with the need to express and yet hide logic on a per-platform basis, the Xamarin / C# stack creates a 1 – 2 punch of slick cross-platform code coupled with easy ways to create per-platform solutions which can all be cross-injected seamlessly.

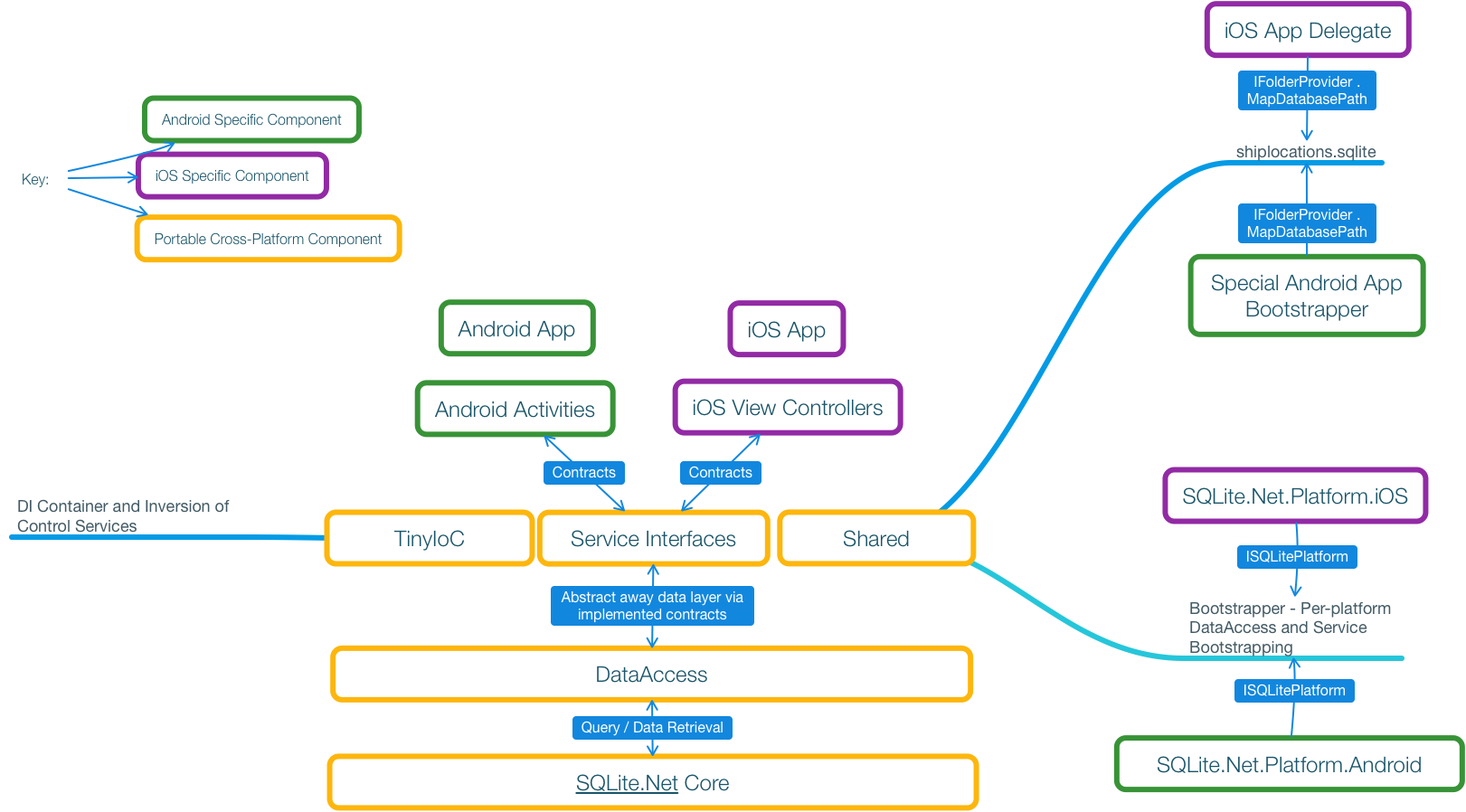
## Start with the end in mind

Throughout this post I will be referencing a sample skeleton app which has a common cross-platform data layer that uses SQLite as the data store.

The app also abstracts away all data access via a set of shared interfaces and data structures.

TinyIoC is used as the bridge between each platform’s main app and the concrete implementations of the data access layer.

It’s crazy diagram time!



Don’t Panic! All will eventually become clear between the diagram, the code, and this wonderful companion explanation.

Let’s start with the complicated stuff first: Portable SQLite packages and per-platform architecture.

## Portable SQLite Package Basics

As of April 2016 there are 2 top-level Portable SQLite products available via NuGet.

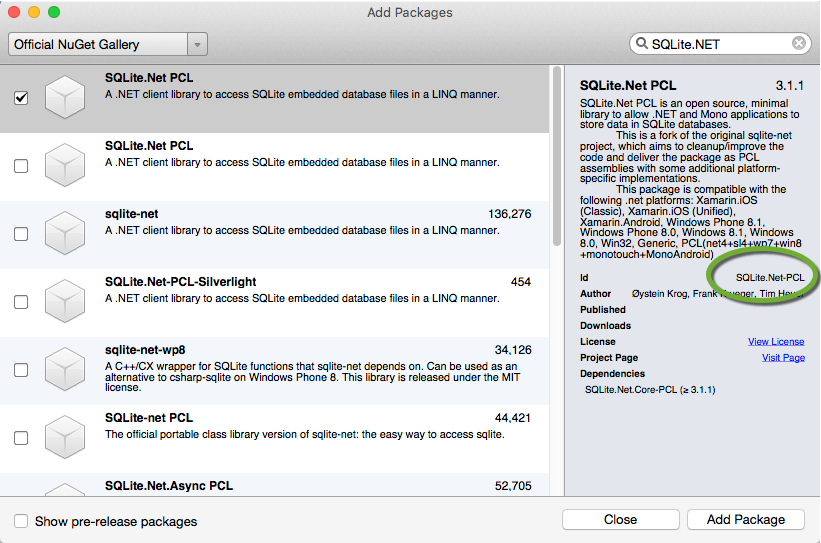
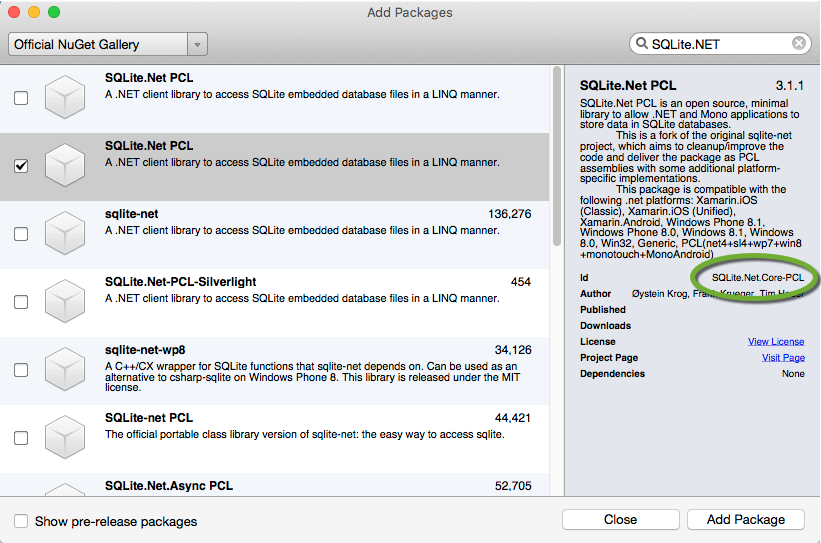
* SQLite.NET-PCL
* SQLite.NET.Async-PCL

At the end of the day, whichever package you pick depends on whether or not you want async / await behavior with your SQLite access calls. Just to make things simple, this sample uses Portable SQLite (not Async), however the rest of this guide still applies just put in “Async” everywhere I reference Portable SQLite.

NOTE! – IMPORTANT! – Due to history of Portable SQLite, random Internet posts, promoted Google results, and even Xamarin misinformation it took me way too long to figure this out! DO NOT make the same mistakes I did!

All of these NuGet packages are sourced from the [SQLite.NET-PCL Github](https://github.com/oysteinkrog/SQLite.Net-PCL) repo.

The Portable SQLite distribution has 2 main NuGet packages available.

* SQLite.NET PCL:
  + 
* … and yet another one named SQLite.NET PCL:
  + 

Do you notice any differences between the package named **SQLite.NET PCL** and **SQLite.NET PCL**?

Yes, I have circled the differences in green for you. No, I really, really don’t like the fact that the top level NuGet packages in the main listing are named the same thing. The rampant confusion across the Internet for SQLite.NET PCL inclusion is enough to make you pull your hair out.

Look at the right hand side, at the text in the green circle.

**SQLite.NET-PCL** is a super helper cross-platform package. When included in a Xamarin.iOS, or Mono.Android project it will automatically include all specific platform assemblies needed for SQLite to work correctly for that platform. You can look at the NuGet definition for SQLite.NET-PCL within Github [here](https://github.com/oysteinkrog/SQLite.Net-PCL/blob/master/nuget/SQLite.Net.nuspec). – Go ahead, take a look! See all those cool per-platform line items? So great!

The second package **SQLite.NET.*Core*-PCL** (notice the word ***Core***) just includes the abstract interfaces and core SQLite implementation (no per-platform implementation helpers) to allow you to create connections, do queries, and do other operations from within a Portable Library. If you look at the [NuGet definition](https://github.com/oysteinkrog/SQLite.Net-PCL/blob/master/nuget/SQLite.Net.Core.nuspec) for **SQLite.NET.*Core*-PCL** you will see that it ONLY includes core assemblies. The Core assemblies are useless without a per-platform implementation injection.

Let’s hopefully bring all this together, and bring in that per-platform implementation.

You see, SQLite has a problem. There is so much stuff within it that needs to be implemented per-platform that they had to hide it all behind an interface (or 4?). In their case: [ISQLitePlatform](https://github.com/oysteinkrog/SQLite.Net-PCL/blob/377cc65b57f63e1337058bd382a207cbbc1cfc68/src/SQLite.Net/Interop/ISQLitePlatform.cs) (Go ahead, click the link, Github code awaits!).

In Github you can see the per-platform concrete implementations for ISQLitePlatform:

* [iOS](https://github.com/oysteinkrog/SQLite.Net-PCL/blob/377cc65b57f63e1337058bd382a207cbbc1cfc68/src/SQLite.Net.Platform.XamarinIOS.Unified/SQLitePlatformIOS.cs)
* [Android](https://github.com/oysteinkrog/SQLite.Net-PCL/blob/377cc65b57f63e1337058bd382a207cbbc1cfc68/src/SQLite.Net.Platform.XamarinAndroid/SQLitePlatformAndroid.cs)

… and yes, many more platforms now that you know where to look…

The bummer is that in order to create a SQLiteConnection you need a per-platform concrete instance of ISQLitePlatform in order to rev up your connection.

Now it is all up to you to find a way to get a platform specific concrete instance of ISQLitePlatform from your per-platform app all the way to the portable code where you want to create a SQLiteConnection.

In our case, we want to want to create SQLiteConnection objects from within our portable cross-platform DataAccess assembly.

In my case, I really like TinyIoC as a DI bridge between assemblies (both native and portable) because it is an awesome container, can do tons of stuff, and is cross-platform and portable in the truest sense of the words.

So, my SQLiteConnection rev up code in DataAccess looks like:

var connection = new SQLite.Net.SQLiteConnection (  
                TinyIoC.TinyIoCContainer.Current.

Resolve<ISQLitePlatform>(),  
 TinyIoC.TinyIoCContainer.Current.

Resolve<IFolderProvider>().MapDatabasePath);

So, I just go to my TinyIoC container and pull out an ISQLitePlatform via **Resolve**.

But, if I am pulling out the ISQLItePlatform, that means I have to put one in via **Register** too!

We will talk more about this with the Bootstrapper discussion, but at the end of the day the code for each platform’s ISQLitePlatform implementation, and placement within the TinyIoC container via Register, and sent off to the Bootstrapped rev up looks like:

On iOS:

new SQLite.Net.Platform.XamarinIOS.SQLitePlatformIOS ()

On Android:

new SQLite.Net.Platform.XamarinAndroid.SQLitePlatformAndroid ()

To bring things full circle, those platform specific implementations were brought in via the **SQLite.NET-PCL** NuGet package included at the ‘app head’ specific level for Android and iOS, new instances of those objects were created and then **Registered** within TinyIoC as a ISQLitePlatform type via the Bootstrapper.

More on all of this later when we talk about the per-app head calls to the Bootstrapper.

You will also notice I have yet another interface called *IFolderProvider* as part of this whole SQLiteConnection construction! What is this madness!?!?!

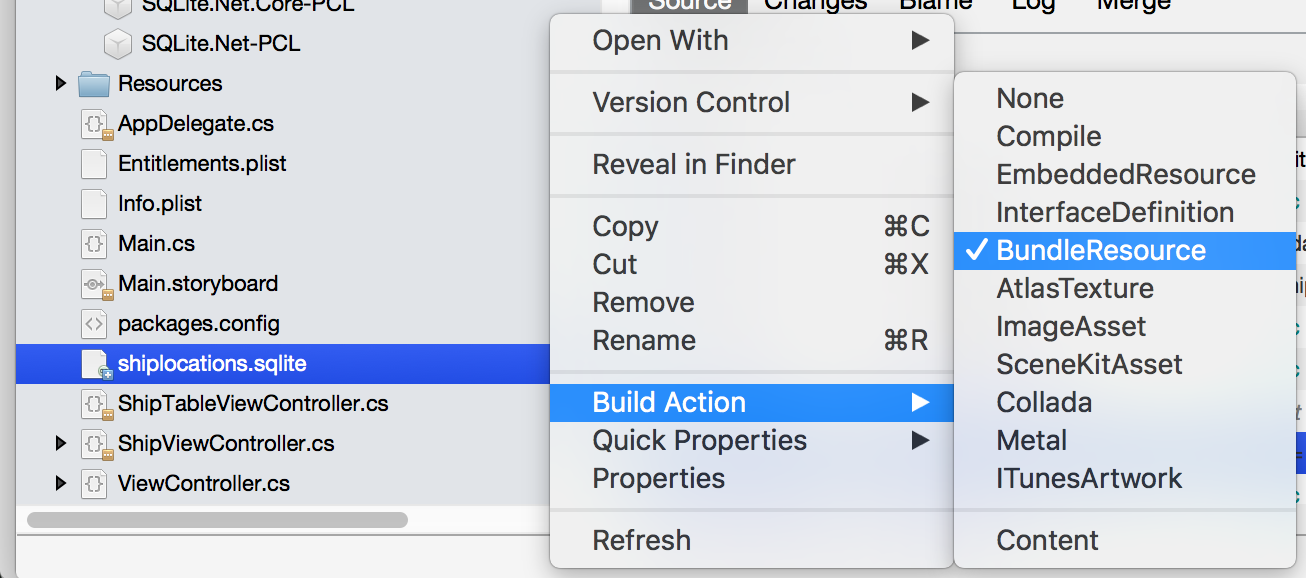
It turns out that each platform (Android, iOS, …) has totally differing ways to handle the file system.

That second parameter to SQLiteConnection is a file path to my existing SQLite database that I package with the app. Since file paths are handled different per platform, I used a custom interface (I called it IFolderProvider) to allow me to hand out different file paths for each platform off of the ‘MapDatabasePath’ property of my per-platform implementation of IFolderProvider.

Here is a sneak peek of the iOS implementation of IFolderProvider.MapDatabasePath as implemented off of my main iOS App Delegate class:

public string MapDatabasePath {  
 get  
  {   
    return NSBundle.MainBundle.PathForResource ("shiplocations", "sqlite");  
  }  
 }

Notice that in the case of iOS I have embedded the shiplocations.sqlite database file directly within the iOS app bundle using the *BundleResource* **Build Action** for the file in Xamarin Studio:



Pro tip: You can include shortcuts to files within your Xamarin Studio / Visual Studio 2015 projects when you add a file. Even though it is a shortcut you can still embed the whole file as if it is really part of the project. In this case, I just put the shiplocations.sqlite file into a shared folder, and added it as a shortcut to my Xamarin.iOS project.

There is a very good reason that I didn’t choose to include the Android version of IFolderProvider implementation of MapDatabasePath property as a ‘quick sample’. That implementation of this simple property requires [way more work than the iOS version](http://blog.reigndesign.com/blog/using-your-own-sqlite-database-in-android-applications/) (Probably best to just check out the source code for more info on the Android implementation).

Ok. To review the above:

* We discovered how to properly include the portable ***Core*** of SQLite.NET PCL vs. **the per-platform master** SQLite.NET PCL NuGet package.
* We took a few side digressions regarding TinyIoC and its use as a master object container across all of our assemblies.
* We gave some hints as to iOS vs. Android file handling, and talked a little about the custom IFolderProvider interface as a way to abstract it per-platform.

Now, let’s talk about the Bootstrapper. The Bootstrapper should help close the gaps across all this hookup, get us to nice solid ground, and provide our top level apps our cross-platform data access layer.

## Bringing it all together – Bootstrapper and TinyIoC

I am going to admit it. TinyIoC is nothing other than a smart *global* singleton dictionary that stores and hands out objects based on specific .NET types.

Ok. It’s true! Yes, *global* variables as a hack! Sorry. ☹ But at least it is type safe? …. Why are you still shaking your head? … Oh well, you will get over it.

The key to brining everything together is to get concrete implementations of the ISQLPlatform, IDataAccess, and IFolderProvider implementations into the TinyIoC container so everything can properly interoperate.

You may be saying: Hey, I recognize ISQLPlatform and IFolderProvider from the previous section, where did this IDataAccess thing come from!

### DataAccess / IDataAccess – Side Note

More on IDataAccess later, but I figure you are going to see hints of it as we talk about the Bootstrapper, so it is better to have a mention of it now.

IDataAccess is the only way for each of our apps to get and set data from our cross-platform, portable, data layer. I purposely choose to make sure that all of my data access is hidden behind a completely neutral contract interface. This allows me to abstract and re-direct things later on, perform unit tests, and easily mock the entire data layer if needed.

### Ok. Back to the Bootstrapper show….

I chose to create a helper assembly that is shared between the app heads (iOS and Android) which can contain any common helpers and also serve as an abstraction for revving up all the needed dependencies across the different heads in a common way. The helper assembly is just called “Shared”.

Within the “Shared” assembly exists a class called Bootstrapper. Within Bootstrapper there is one function: Startup.

public static void Startup(ISQLitePlatform platform,  
            IFolderProvider folderProvider)  
        {  
            TinyIoC.TinyIoCContainer.Current.Register<IFolderProvider> (folderProvider);  
            TinyIoC.TinyIoCContainer.Current.Register<ISQLitePlatform> (platform);  
            TinyIoC.TinyIoCContainer.Current.Register<IDataAccess> (new DataAccess.DataAccess ());  
        }

Startup takes in as parameters implementations of each of the interfaces that need to be implemented per-platform (in our case ISQLitePlatform and IFolderProvider).

We then take those interfaces and shove their instances into the TinyIoC container via **Register** so everybody in the app can use them via the TinyIoC **Resolve** function.

Notice, there is that DataAccess stuff again!

In this case, by having a Shared assembly that can directly reference our concrete DataAccess assembly, we don’t have to have each app head reference our concrete DataAccess assembly. This ensures that our app heads ONLY use the contracts as defined within our Services assembly for any and all data access.

This allows us to transplant in mocks to the app in one place via the Bootstrapper. A central Bootstrapper even allows us to parameterize the Bootstrapper to include mocks as needed. The options are endless and quite awesome.

It is a good thing to have this level of separation, especially for a cross-platform mobile app as you can see when/where the per-platform stuff is injected into the overall application all in one location: The Bootstrapper.

In the case of the Bootstrapper, the order of calls to TinyIoC Resolve may matter. It turns out that the DataAccess class needs the ISQLitePlatform and IFolderProvider interfaces to get the needed stuff to construct a SQLiteConnection:

var connection = new SQLite.Net.SQLiteConnection (  
                TinyIoC.TinyIoCContainer.Current.Resolve<ISQLitePlatform>(),  
                TinyIoC.TinyIoCContainer.Current.Resolve<IFolderProvider>().MapDatabasePath);

Since everything was Bootstrapped, all is good with fully shared cross-platform SQLite access within our DataAccess layer!

Ok, at this point you should now see how the whole app knits together.

You can see how we use TinyIoC as a global cross-assembly object container for our interface only **resolve** and **registration**.

You can see how we have injected a per-platform implementation of ISQLitePlatform and IFolderProvider which will eventually be used by DataAccess to access our SQLite database.

Then we have also done the reverse of per-platform injection by registering the cross-platform shared IDataAccess for future resolves within our per-platform code (i.e. Activities on Android, and ViewControllers on iOS).

## IDataAccess within iOS View Controllers and Android Activities

Ok. The app is revved up. We have bootstrapped in our per-platform interfaces, and registered our root IDataAccess interface.

Now we need to actually do some work!

So, lets go into our iOS head and take a look at our ShipTableViewController.

We want to get a list of all the ships in our sample and show it in a UITableView.

private async void LoadShips()  
        {  
            var dataAccess = TinyIoC.TinyIoCContainer.Current.Resolve<IDataAccess> ();  
  
            this.Ships = await dataAccess.GetAllShips ();  
  
            *//It may have taken a while for the ships to load. Refresh the list.*  
            this.TableView.ReloadData();  
        }  
  
        IList<ShipToken> Ships {get; set;}

We go into the cloud that is our TinyIoC container. We pull out our IDataAccess interface initialized during app startup via the Bootstrapper. Then we call GetAllShips on it, and await the response.

GetAllShips returns an IList<ShipToken> so we store it in a local variable.

We then tell our associated table view to reload all of its data to start showing the ships we just retrieved. The table view follows all the standard conventions via the iOS table view data source, and table view delegates to render the table cells.

That’s iOS, but what about Android?

On Android we rev up a ListActivity that contains a ListView.

Within our custom ListActivity we rev up a custom adapter.

The custom adapter then loads the ship list via the exact same call off of the IDataAccess interface retrieved via TinyIoC.

private async void LoadData()  
        {  
            *//Start loading the ShipList...*  
            var dataAccess = TinyIoC.TinyIoCContainer.Current.Resolve<IDataAccess>();  
            Ships = await dataAccess.GetAllShips ();  
            this.NotifyDataSetChanged ();  
        }

Woo hoo! Same data access layer code! Multiple platforms! All with the correctly wired per-platform wrapping for Android (ListActivity / BaseAdapter) and iOS (UITableView / UITableViewController) to show lists of data!

Even better: We have full non-UI blocking code in the form of async / await via our IDataAccess interface across 2 different platforms!

## Last stop for this post: async / await basics via an interface

If you have experience with async / await, jump to the conclusion. All is cool.

One thing I encountered early on was the need to have async / await behaviors via an interface.

async / await is [syntactic sugar](https://en.wikipedia.org/wiki/Syntactic_sugar) that builds on the Task<T> construct by providing a seamless way for you to call long running tasks from within your functions, not block your running thread, yet still resume execution when the long running task is complete.

Non-blocking behavior is especially important on mobile platforms so animations, spinners, and core UI does not freeze up during long running tasks.

IMHO: No other cross-platform language makes non-blocking mobile interfaces easier than C# via Xamarin and async / await.

Very basic example:

        public **async Task<System.Collections.Generic.IList<ShipToken>>** GetAllShips() {  
            var client = new HttpClient ();  
            var stream = **await**client.GetStreamAsync ("http://s3-us-west-2.amazonaws.com/danfs/manifest.json");  
  
            var serializer = new JsonSerializer();  
  
            using (var sr = new StreamReader(stream))  
            using (var jsonTextReader = new JsonTextReader(sr))  
            {  
                var allShipsResponse = serializer.Deserialize<List<ShipToken>>(jsonTextReader);  
  
                return allShipsResponse.Cast<ShipToken> ().ToList();  
            }  
        }

The above function is responsible for going out via HTTP, retrieving the JSON at the HTTP endpoint, deserializing all that JSON into a list of ShipToken POCOs.

We have already seen code that calls this function from the iOS and Android sides:

var dataAccess = TinyIoC.TinyIoCContainer.Current.Resolve<IDataAccess>();  
            Ships = await dataAccess.GetAllShips ();

Let’s review standard async / await stuff with the 2 examples above in mind.

If a function is marked with the **async** keyword it means that the function will create a new task and execute it. This is equivalent to the function needing to **await** on another **async** function.

If a function is marked with the **async** keyword and returns a **Task<T>** then the function can be

awaited and the return type from the **await** will be the **T** data type wrapped by the **Task<T**>.

No example, but worth noting: If a function is marked with the **async** keyword and **void** is the return type, the function is a ‘fire and forget’ method and nobody can ever wait for the internal tasks it creates and awaits on to complete. The usual return from an **async void** function is an event fire or an Action or Func<T, …> callback.

async / await functions can be chained together and other async functions can call other async functions. As the old lady once said “[It’s turtles all the way down](https://en.wikipedia.org/wiki/Turtles_all_the_way_down)” as far as the use of async/await in this scenario.

When you want an async method on an interface, all the function has to do is to return a Task<T>, then the user of the interface can await on the function.

From our IDataAccess interface definition:

Task<IList<ShipToken>> GetAllShips();

Because the GetAllShips() method returns a **Task<IList<ShipToken>>** we can await on the function and get our list of ship tokens:

Ships = await dataAccess.GetAllShips ();

Keep in mind, when we call the GetAllShips() method, the wrapping function also needs to be declared **async** otherwise the main UI will block during the entire HTTP call.

Another way to think of the **await** keyword would be:

Hey, I want this function to execute on a different thread, but don’t block my thread! When you return, then please resume execution within the thread that I called you from, and on the line of code right after the **await,** so I can do more stuff, especially with your return value.

## Conclusion

So in this document we went over a cross-platform scheme to share SQLite data access, and wrapped business logic, across multiple native platforms using Portable Assemblies, SQLite.NET PCL, TinyIoC, custom interfaces for file access, and a custom Bootstrapper go-between.

I have feeling that you can take this design and really run with it. I bet you can add a Windows 10 / Universal Windows Platform head to it and get the same SLN to load in Visual Studio 2015 on Windows and via Xamarin Studio on Mac OS X.

The possibilities are endless, and awesome because as you add more and more features to your shared data layers, you are then able to access and project those features onto any platform that portable assemblies and/or the Xamarin frameworks are able to work with.