Agent Framework Documentation

Release 1.1.0

Agent Framework and contributors

Dec 26, 2018

Initial Setup

AgentFramework is a .NET Core library for building Sovrin interoperable agent services. It is an abstraction on top of Indy SDK that provides a set of API's for building Indy Agents. The framework runs on any .NET Standard target, including ASP.NET Core and Xamarin.

CHAPTER 1

Installation and configuration

1.1 Using NuGet

```
Install-Package AgentFramework.Core -Source https://www.myget.org/F/agent-framework/
↪api/v3/index.json
```

The framework will be moved to nuget.org soon. For the time being, stable and pre-release packages are available at `https://www.myget.org/F/agent-framework/api/v3/index.json`. You can add nuget.config anywhere in your project path with the myget.org repo.

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
    <packageSources>
        <add key="myget.org" value="https://www.myget.org/F/agent-framework/api/v3/
↪index.json" />
        <add key="nuget.org" value="https://api.nuget.org/v3/index.json" />
    </packageSources>
</configuration>
```

1.2 Setting up development environment

Agent Framework uses Indy SDK wrapper for .NET which requires platform specific native libraries of libindy to be available in the running environment. Check the [Indy SDK project page](https://github.com/hyperledger/indy-sdk) for details on installing libindy for different platforms or read the brief instructions below.

Make sure you have [.NET Core SDK](https://dotnet.microsoft.com/download) installed for your platform.

1.2.1 Windows

You can download binaries of libindy and all dependencies from the [Sovrin repo](https://repo.sovrin.org/windows/libindy/). The dependcies are under *deps* folder and *libindy* under one of streams (rc, master, stable). There are two

options to link the DLLs

- Unzip all files in a directory and add that to your PATH variable (recommended for development)
- Or copy all DLL files in the publish directory (recommended for published deployments)

More details at the [Indy documentation for setting up Windows environment](https://github.com/hyperledger/indy-sdk/blob/master/doc/windows-build.md)

1.2.2 MacOS

Check Setup Indy SDK build environment for MacOS

1.2.3 Linux

Build instructions for [Ubuntu based distros](https://github.com/hyperledger/indy-sdk/blob/master/doc/ubuntu-build.md) and [RHEL based distros](https://github.com/hyperledger/indy-sdk/blob/master/doc/rhel-build.md).

CHAPTER 2

Agent Workflows

Before you begin reading any of the topics below, please familiarize youself with the core idea behind Hyperledger Indy. We suggest that you go over the Indy SDK Getting Started Guide.

2.1 Roles and players

2.2 Establishing secure connection

2.2.1 Sending invitations

2.2.2 Negotating connection

2.3 Credential issuence

2.3.1 Issuing credential

2.3.2 Storing issued credential

2.3.3 Revocation

2.4 Proof verification

2.4.1 Proof requests

2.4.2 Preparing proof

2.4.3 Verification

CHAPTER 3

Mobile Agents with Xamarin

3.1 Using Indy with Xamarin

When working with Xamarin, we can fully leverage the offical Indy wrapper for dotnet, since the package is fully compatible with Xamarin runtime. The wrapper uses `DllImport` to invoke the native Indy library which exposes all functionality as C callable functions. In order to make the library work in Xamarin, we need to make *libindy* available for Android and iOS, which requires bundling static libraries of libindy and it's dependencies built for each platform.

3.2 Instructions for Android

To setup Indy on Android you need to add the native libindy references and dependencies. The process is described in detail at the official Xamarin documentation Using Native Libraries with Xamarin.Android.

Below are a few additional things that are not covered by the documentation that are Indy specific.

3.2.1 Download static libraries

- Our repo (includes *libgnustl_shared.so*) - samples/xamarin/libs-android
- Sovrin repo - https://repo.sovrin.org/android/libindy/

For Android the entire library and its dependencies are compiled into a single shared object (libindy.so). In order for `libindy.so` to be executable we must also include `libgnustl_shared.so`.

Note: You can find `libgnustl_shared.so` in your `android-ndk` installation directory under `\sources\cxx-stl\gnu-libstdc++\4.9\libs`.

Depending on the target abi(s) for the resulting app, not all of the artifacts need to be included, for ease of use below we document including all abi(s).

3.2.2 Setup native references

In Visual Studio (for Windows or Mac) create new Xamarin Android project. If you want to use Xamarin Forms, the instructions are the same. Apply the changes to your Android project in Xamarin Forms.

The required files can be added via your IDE by clicking Add-Item and setting the build action to `AndroidNativeLibrary`. However when dealing with multiple ABI targets it is easier to manually add the references via the android projects .csproj. Note - if the path contains the abi i.e *..x86library.so* then the build process automatically infers the target ABI.

If you are adding all the target ABI's to you android project add the following snippet to your .csproj.

```
<ItemGroup>
  <AndroidNativeLibrary Include="..\libs-android\armeabi\libindy.so" />
  <AndroidNativeLibrary Include="..\libs-android\arm64-v8a\libindy.so" />
  <AndroidNativeLibrary Include="..\libs-android\armeabi-v7a\libindy.so" />
  <AndroidNativeLibrary Include="..\libs-android\x86\libindy.so" />
  <AndroidNativeLibrary Include="..\libs-android\x86_64\libindy.so" />
  <AndroidNativeLibrary Include="..\libs-android\armeabi\libgnustl_shared.so" />
  <AndroidNativeLibrary Include="..\libs-android\arm64-v8a\libgnustl_shared.so" />
  <AndroidNativeLibrary Include="..\libs-android\armeabi-v7a\libgnustl_shared.so" />
  <AndroidNativeLibrary Include="..\libs-android\x86\libgnustl_shared.so" />
  <AndroidNativeLibrary Include="..\libs-android\x86_64\libgnustl_shared.so" />
</ItemGroup>
```

Note: Paths listed above will vary project to project.

3.2.3 Load runtime dependencies

Load these dependencies at runtime. To do this add the following to your MainActivity.cs

```
JavaSystem.LoadLibrary("gnustl_shared");
JavaSystem.LoadLibrary("indy");
```

3.2.4 Setup Android permissions

In order to use most of libindy's functionality, the following permissions must be granted to your app, you can do this by adjusting your AndroidManifest.xml, located under properties in your project.

```
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.INTERNET" />
```

If you are running your android app at API level 23 and above, these permissions also must be requested at runtime, in order to do this add the following to your MainActivity.cs

```
if (Build.VERSION.SdkInt >= BuildVersionCodes.M)
{
    RequestPermissions(new[] { Manifest.Permission.ReadExternalStorage }, 10);
    RequestPermissions(new[] { Manifest.Permission.WriteExternalStorage }, 10);
    RequestPermissions(new[] { Manifest.Permission.Internet }, 10);
}
```

3.3 Instructions for iOS

To setup Indy on iOS you need to add the native libindy references and dependencies. The process is described in detail at the official Xamarin documentation Native References in iOS, Mac, and Bindings Projects.

Below are a few additional things that are not covered by the documentation that are Indy specific.

3.3.1 Download static libraries

In order to enable the Indy SDK package to recognize the *DllImport* calls to the native static libraries, we need to include them in our solution.

These includes the following static libraries:

- libindy.a
- libssl.a
- libsodium.a
- libcrypto.a
- libzmq.a

Pre-built libraries

Can be found in the iOS sample project.

Build your own libs

The Indy team doesn't provide static libraries for all of the dependencies for iOS. Here are some helpful instructions on building the dependencies for iOS should you decide to build your own.

- Open SSL for iOS
- Build ZeroMQ library
- libsodium script of iOS

The above links should help you build the 4 static libraries that libindy depends on. To build libindy for iOS, check out the offical Indy SDK repo or [download the library from the Sovrin repo](https://repo.sovrin.org/ios/libindy/).

3.3.2 Setup native references

In Visual Studio (for Windows or Mac) create new Xamarin iOS project. If you want to use Xamarin Forms, the instructions are the same. Apply the changes to your iOS project in Xamarin Forms.

Add each library as native reference, either by right clicking the project and Add Native Reference, or add them directly in the project file.

Note: Make sure libraries are set to `Static` in the properties window and `Is C++` is selected for `libzmq.a` only.

The final project file should look like this (paths will vary per project):

```
<ItemGroup>
  <NativeReference Include="..\libs-ios\libcrypto.a">
    <Kind>Static</Kind>
  </NativeReference>
  <NativeReference Include="..\libs-ios\libsodium.a">
    <Kind>Static</Kind>
  </NativeReference>
  <NativeReference Include="..\libs-ios\libssl.a">
    <Kind>Static</Kind>
  </NativeReference>
  <NativeReference Include="..\libs-ios\libzmq.a">
    <Kind>Static</Kind>
    <IsCxx>True</IsCxx>
  </NativeReference>
  <NativeReference Include="..\libs-ios\libindy.a">
    <Kind>Static</Kind>
  </NativeReference>
</ItemGroup>
```

3.3.3 Update MTouch arguments

In your project options under *iOS Build* add the following to *Additional mtouch arguments*

```
-gcc_flags -dead_strip -v
```

If you prefer to add them directly in the project file, add the following line:

```
<MtouchExtraArgs>-gcc_flags -dead_strip -v</MtouchExtraArgs>
```

> **Warning:** This step is mandatory, otherwise you won't be able to build the project. It prevents linking unused symbols in the static libraries. Make sure you add these arguments for all configurations. See example project file.

3.3.4 Install NuGet packages

Install the Nuget packages for Indy SDK and/or Agent Framework and build your solution. Everything should work and run just fine.

```
dotnet add package AgentFramework.Core --source https://www.myget.org/F/agent-
↪framework/api/v3/index.json
```

If you run into any errors or need help setting up, please open an issue in this repo.

Finally, check the Xamarin Sample we have included for a fully configured project.

CHAPTER 4

Agent services with ASP.NET Core

4.1 Installation

A package with extensions and default implementations for use with ASP.NET Core is available.

4.2 Configure required services

Inside your `Startup.cs` fine in `ConfigureServices(IServiceCollection services)` use the extension methods to add all dependent services and optionally pass configuration data.

```
public void ConfigureServices(IServiceCollection services)
{
    // other configuration
    services.AddAgent();
}
```

4.2.1 Configure options manually

You can customize the wallet and pool configuration options using

```
services.AddAgent(config =>
{
    config.SetPoolOptions(new PoolOptions { GenesisFilename = Path.GetFullPath("pool_
↪genesis.txn") });
    config.SetWalletOptions(new WalletOptions
    {
        WalletConfiguration = new WalletConfiguration { Id = "MyAgentWallet" },
        WalletCredentials = new WalletCredentials { Key =
↪"SecretWalletEncryptionKeyPhrase" }
    });
});
```

4.2.2 Use options pattern

Alternatively, options be configured using APS.NET Core `IOptions<T>` pattern.

```
services.Configure<PoolOptions>(Configuration.GetSection("PoolOptions"));
services.Configure<WalletOptions>(Configuration.GetSection("WalletOptions"));
```

Set any fields you'd like to configure in your `appsettings.json`.

```
{
    // config options
    "WalletOptions": {
        "WalletConfiguration": {
            "Id": "MyAgentWallet",
            "StorageConfiguration": { "Path": "[path to wallet storage]" }
        },
        "WalletCredentials": { "Key": "SecretWalletEncryptionKeyPhrase" }
    },
    "PoolOptions": {
        "GenesisFilename": "[path to genesis file]",
        "PoolName": "DefaultPool",
        "ProtocolVersion": 2
    }
}
```

4.3 Initialize agent middleware

In `Configure(IApplicationBuilder app, IHostingEnvironment env)` start the default agent middleware

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    // Endpoint can be any address you'd like to bind to this middleware
    app.UseAgent("http://localhost:5000/agent");

    // .. other services like app.UseMvc()
}
```

The default agent middleware is a simple implementation. You can create your middleware and use that instead if you'd like to customize the message handling.

```
app.UseAgent<CustomAgentMiddlware>("http://localhost:5000/agent");
```

See AgentMiddleware.cs for example implementation.

Tip: In ASP.NET Core, the order of middleware registration is important, so you might want to add the agent middleware before any other middlewares, like MVC.

4.4 Calling services from controllers

Use dependency injection to get a reference to each service in your controllers.

```
public class HomeController : Controller
{
    private readonly IConnectionService _connectionService;
    private readonly IWalletService _walletService;
    private readonly WalletOptions _walletOptions;

    public HomeController(
        IConnectionService connectionService,
        IWalletService walletService,
        IOptions<WalletOptions> walletOptions)
    {
        _connectionService = connectionService;
        _walletService = walletService;
        _walletOptions = walletOptions.Value;
    }

    // ...
}
```

CHAPTER 5

Hosting agents in docker containers

Hosting agents in docker container is the easiest way to ensure your running environment has all dependencies required by the framework. We provide images with libindy and dotnet-sdk preinstalled.

5.1 Usage

```
FROM streetcred/dotnet-indy:latest
```

The images are based on *ubuntu:16.04*. You can check the docker repo if you want to build your own image or require specific version of .NET Core or libindy.

5.2 Example build

Check the web agent docker file for an example of building and running ASP.NET Core project inside docker container with libindy support.

CHAPTER 6

Samples

6.1 ASP.NET Core Agents

A sample agent running in ASP.NET Core that runs the default agent middleware can be found in samples/aspnetcore. This agent is also used in the Docker sample.

```
ddotnet run --project samples/aspnetcore/WebAgent.csproj
```

6.1.1 Running multiple instances

To run multiple agent instances that can communicate, you can specify the binding address and port by setting the `ASPNETCORE_URLS` environment variable

```
# Unix/Mac:
ASPNETCORE_URLS="http://localhost:5001" dotnet run --no-launch-profile --project␣
↪samples/aspnetcore/WebAgent.csproj

# Windows PowerShell:
$env:ASPNETCORE_URLS="http://localhost:5001" ; dotnet run --no-launch-profile --
↪project samples/aspnetcore/WebAgent.csproj

# Windows CMD (note: no quotes):
SET ASPNETCORE_URLS=http://localhost:5001 && dotnet run --no-launch-profile --project␣
↪samples/aspnetcore/WebAgent.csproj
```

Note: The sample web agent doesn't use any functionality that requires a local indy node, but if you'd like to extend the sample and test interaction with the ledger, you can run a local node using the instructions below.

6.1.2 Run a local Indy node with Docker

The repo contains a docker image that can be used to run a local pool with 4 nodes.

```
docker build -f docker/indy-pool.dockerfile -t indy_pool .
docker run -itd -p 9701-9709:9701-9709 indy_pool
```

6.2 Mobile Agent with Xamarin Forms

6.3 Docker container example

6.3.1 Running the example

At the root of the repo run:

```
docker-compose up
```

This will create an agent network with a pool and three identical agents able to communicate with each other in the network. Navigate to http://localhost:7001/, http://localhost:7002/ and http://localhost:7003/ to create and accept connection invitations between the different agents.

6.3.2 Running the unit tests

```
docker-compose -f docker-compose.test.yaml up --build --remove-orphans --abort-on-
↪container-exit --exit-code-from test-agent
```

Note: You may need to cleanup previous docker network created using *docker network prune*