Agent Framework Documentation

Release 1.1.0

Agent Framework and contributors

Aug 25, 2019

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

To use the agent framework in your project, add the nuget packages.

If using the package manager:

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

If using the .NET CLI:

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

Available packages:

- `AgentFramework.Core` - core framework package
- `AgentFramework.AspNetCore` - simple middleware and service extensions to easily configure and run an agent
- `AgentFramework.Core.Handlers` - provides a framework for registering custom message handlers and extending the agent functionality

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" />
```

(continues on next page)

(continued from previous page)

```
    </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 for details on installing libindy for different platforms or read the brief instructions below.

Make sure you have .NET Core SDK installed for your platform.

1.2.1 Windows

You can download binaries of libindy and all dependencies from the Sovrin repo. The dependencies 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.

1.2.2 MacOS

Check Setup Indy SDK build environment for MacOS.

Copy `libindy.a` and `libindy.dylib` to the `/usr/local/lib/` directory.

1.2.3 Linux

Build instructions for Ubuntu based distros and RHEL based distros.

CHAPTER 2

Configuration and provisioning

2.1 Services overview

- `IProvisioningService` - used to provision new agents and access the provisioning configuration that contains endpoint data, ownerhip info, service endpoints, etc.
- `IConnectionService` - manage connection records, create and accept invitations
- `ICredentialService` - manage credential records, create offer, issue, revoke and store credentials
- `IProofService` - send proof requests, provide and verify proofs
- `IWalletRecordService` - utility service used to manage custom application records that are stored in the wallet
- `ISchemaService` - create and manage schemas and credential definitions

2.2 Dependency injection

When using ASP.NET Core, you can use the extension methods to configure the agent. This will add all required dependencies to the service provider. Additionaly, the AgentFramework depends on the Logging extensions. These need to be added as well.

If using other tool, you will have to add each required service or message handler manually.

Example if using Autofac

```
// .NET Core dependency collection
var services = new ServiceCollection();
services.AddLogging();

// Autofac builder
var builder = new ContainerBuilder();
```

(continues on next page)

(continued from previous page)

```
// Register all required services
builder.RegisterAssemblyTypes(typeof(IProvisioningService).Assembly)
    .Where(x => x.Namespace.StartsWith("AgentFramework.Core.Runtime",
        StringComparison.InvariantCulture))
    .AsImplementedInterfaces()
    .SingleInstance();

// If using message handler package, you can add all handlers
builder.RegisterAssemblyTypes(typeof(IMessageHandler).Assembly)
    .Where(x => x.IsClass && x is IMessageHandler)
    .AsSelf()
    .SingleInstance();

builder.Populate(services);
```

Check the Xamarin Sample for example registration.

2.3 Provisioning an Agent

The process of provisioning agents will create and configure an agent wallet and initialize the agent configuration. The framework will generate a random Did and Verkey, unless you specify `AgentSeed` which is used if you need determinism. Length of seed must be 32 characters.

```
await _provisioningService.ProvisionAgentAsync(
    new ProvisioningConfiguration
    {
        EndpointUri = "http://localhost:5000",
        OwnerName = "My Agent"
    });
```

Check the ProvisioningConfiguration.cs for full configuration details. You can retrieve the generated details like agent Did and Verkey using

```
var provisioning = await _provisioningService.GetProvisioningAsync(wallet);
```

2.3.1 Trust Anchor requirement

If an agent is intended to act as an issuer, i.e. be able to issue credentials, their DID must be registered on the ledger with the *TRUST_ANCHOR* role. Additionally, when provisioning the agent, set the `ProvisioningConfiguration.CreateIssuer` propety to true. If you already have a seed for creating the issuer DID set the `ProvisioningConfiguration.IssuerSeed` to that value. Otherwise, a random DID will be generated. This DID must be added to the ledger as *TRUST_ANCHOR*.

Tip: If you are using the development indy node docker image, use `000000000000000000000000Steward1` as issuer seed. This will create a DID that has all required permissions.

CHAPTER 3

Agent Workflows

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

3.1 Models and states

The framework abstracts the main workflows of Indy into a state machine model. The following models and states are defined:

3.1.1 Connections

Represented with a `ConnectionRecord`, this entity describes the pairwise relationship with another party. The states for this record are:

- `Invited` - initially, when creating invitations to connect, the record will be set to this state.
- `Negotating` - set after accepting an invitation and sending a request to connect
- `Connected` - set when both parties have acknowledged the connection and have a pairwise record of each others DID's

3.1.2 Credentials

Represented wih a `CredentialRecord`, this entity holds a reference to issued credential. While only the party to whom this credential was issued will have the actual credential in their wallet, both the issuer and the holder will have a CredentialRecord with the associated status for their reference. Credential states:

- `Offered` - initial state, when an offer is sent to the holder
- `Requested` - the holder has sent a credential request to the issuer
- `Issued` - the issuer accepted the credential request and issued a credential

- `Rejected` - the issuer rejected the credential request
- `Revoked` - the issuer revoked a previously issued credential

3.1.3 Proofs

Represented with a `ProofRecord`, this entity references a proof flow between the holder and verifier. The `ProofRecord` contains information about the proof request as well as the disclosed proof by the holder. Proof states:

- `Requested` - initial state when the verifier sends a proof request
- `Accepted` - the holder has provided a proof
- `Rejected` - the holder rejected providing proof for the request

3.2 Schemas and definitions

Before an issuer can create credentials, they need to register a credential definition for them on the ledger. Credential definition requires a schema, which can also be registered by the same issuer or it can already be present on the ledger.

```
// creates new schema and registers the schema on the ledger
var schemaId = await _schemaService.CreateSchemaAsync(
    _pool, _wallet, "My-Schema", "1.0", new[] { "FirstName", "LastName", "Email" });

// to lookup an existing schema on the ledger
var schemaJson = await _schemaService.LookupSchemaAsync(_pool, schemaId);
```

Once a `schemaId` has been established, an issuer can send their credential definition on the ledger.

```
var definitionId = await _schemaService.CreateCredentialDefinitionAsync(_pool, _
↪wallet,
    schemaId, supportsRevocation: true, maxCredentialCount: 100);
```

The above code will create `SchemaRecord` and `DefinitionRecord` in the issuer wallet that can be looked up using the `ISchemaService`.

> **Warning:** Creating schemas and definition requires an issuer. See the *Trust Anchor requirement* above.

To retrieve all schemas or definitions registered with this agent, use:

```
var schemas = await _schemaService.ListSchemasAsync(_wallet);
var definitions = await _schemaService.ListCredentialDefinitionsAsync(_wallet);

// To get a single record
var definition = await _schemaService.GetCredentialDefinitionAsync(wallet,␣
↪definitionId);
```

3.3 Establishing secure connection

Before two parties can exchange agent messages, a secure connection must be established between them. The agent connection workflow defines this handshake process by exchanging a connection request/response message.

3.3.1 Sending invitations

Connection invitations are exchanged over a previously established trusted protocol such as email, QR code, deep link, etc. When Alice wants to establish a connection to Bob, she can create an invitation:

```
// Alice creates an invitation
var invitation = await connectionService.CreateInvitationAsync(aliceWallet);
```

She sends this invitation to Bob using the above described methods.

3.3.2 Negotating connection

Once Bob received the invitation from Alice, they can accept that invitation and initiate the negotiation process

```
// Bob accepts invitation and sends a message request
await connectionService.AcceptInvitationAsync(bobWallet, invitation);
```

If you are using the default message handlers, no other step in needed - connection between Alice and Bob has been established. Use `IConnectionService.ListAsync` to fetch the connection records. Established connections will have the `State` property set to `Connected`.

Tip: If you decide to use custom handlers and want more control over the negotiation process, the connection service provides methods to work with the connections message flows, such as processing and accepting requests/responses. A full step by step code is available in the unit tests project in `EstablishConnectionAsync`.

3.4 Credential issuance

An issuer may use the `ICredentialService` to issue new credentials. A credential issuance starts with a credential offer.

```
var offerConfig = new OfferConfiguration()
{
    // the id of the connection record to which this offer will be sent
    ConnectionId = connectionId,
    CredentialDefinitionId = definitionId
};

// Send an offer to the holder using the established connection channel
var credentialRecordId = await credentialService.SendOfferAsync(issuerWallet,␣
↪offerConfig);
```

When credential offer is sent, new `CredentialRecord` will be created and it's state set to `Offered`. You can list all credential records using

```
var credentials = await credentialService.ListAsync();
```

3.4.1 Issuing credential

```
var values = new Dictionary<string, string>
{
    {"FirstName", "Jane"},
    {"LastName", "Doe"},
    {"Email", "no@spam"}
};

// Issuer accepts the credential requests and issues a credential
await credentialService.IssueCredentialAsync(pool, issuerWallet, credentialRecordId,␣
↪values);
```

An issuer can issue a credential only if the credential record state is `Requested`. This means that the holder has accepted the offer and sent back a credential request message.

3.4.2 Storing issued credential

If using the default handlers, once a credential has been issued and received by the holder's agent, it will be automatically stored and available in the wallet.

3.4.3 Revocation

If the credential definition supports revocation (can only be set when creating the definition), an issuer may decide to revoke a credential.

```
// Revokes a credential, updates the tails file and sends the delta to the ledger
await credentialService.RevokeCredentialAsync(pool, wallet, credentialRecordId)
```

3.5 Proof verification

3.5.1 Proof requests

3.5.2 Preparing proof

3.5.3 Verification

CHAPTER 4

Payments

Payments are a core feature of the framework and are implemented as optional module. Working with payments requires an implementation of the `IPaymentService` for that specific payment method. Currently, there's support for Sovrin Token payments.

4.1 Installation

Packages supporting Sovrin Token payments can be found on nuget.org

Package Manager CLI:

```
Install-Package AgentFramework.Payments.SovrinToken
```

.NET CLI:

```
dotnet add package AgentFramework.Payments.SovrinToken
```

4.1.1 Add libsovtoken static library

4.2 Configuration

4.3 Records and services

4.4 Working with payments

4.4.1 Create and set default payment address

4.4.2 Check balance at address

4.4.3 Attaching payments to agent messages

4.4.4 Making payments

4.4.5 Attaching payment receipt to agent messages

4.5 Using libnullpay for development

CHAPTER 5

Mobile Agents with Xamarin

5.1 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.

5.2 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.

5.2.1 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).

5.2.2 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.

5.2.3 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");
```

5.2.4 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);
}
```

5.3 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.

5.3.1 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/).

5.3.2 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>
```

5.3.3 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.

5.3.4 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 6

Agent services with ASP.NET Core

6.1 Installation

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

6.2 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();
}
```

6.2.1 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" }
    });
});
```

6.2.2 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
    }
}
```

6.3 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.

6.4 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 7

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.

7.1 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.

7.2 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 8

Getting Started Guide

8.1 Creating a New Project

This getting started guide will show you how to create a custom AspNetCore web application and use the agent framework to create connections and send basic messages. We are going to start from scratch. All you need to have installed in Visual Studio and the .NET Core SDK.

8.1.1 Prerequisites

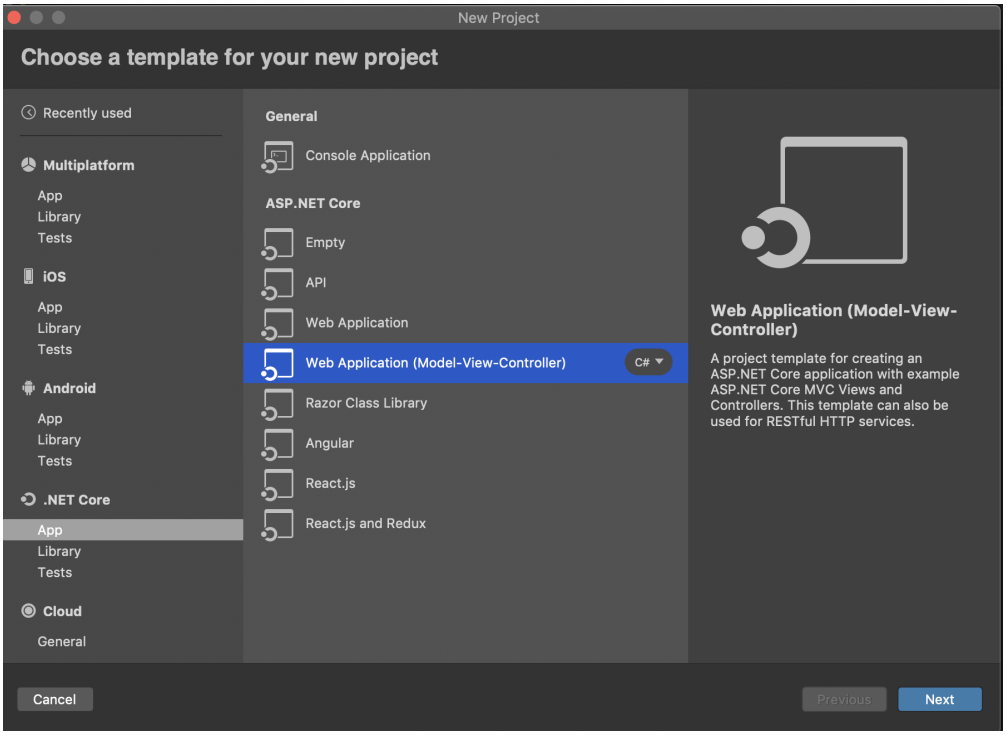
If you haven't already done so, install Visual Studio community and the .NET Core 2.2.300 SDK

- Install Visual Studio
- Install .NET Core SDK 2.2

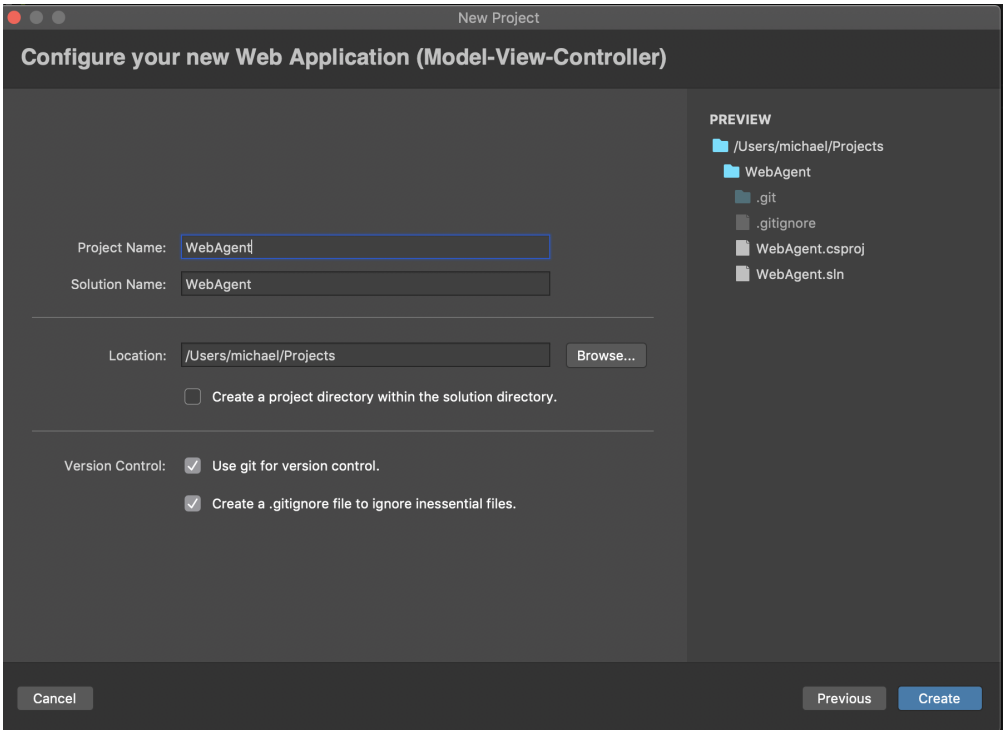
You should be running an instance of Windows or MacOS for this particular demo.

8.1.2 Create an AspNetCore Project

Open Visual Studio and select new project, then choose Web Application (Model-View-Controller):



Select the .NET Core 2.2, then name your project WebAgent.



8.1.3 Installing the Required Packages

Use one of the three methods below to load the AgentFramework.Core packages into your project using nuget.

Package Manager CLI:

```
Install-Package AgentFramework.Core -Source https://www.myget.org/F/agent-framework/
↪api/v3/index.json -v 4.0.0-preview.662
```

.NET CLI:

```
dotnet add package AgentFramework.Core -s https://www.myget.org/F/agent-framework/api/
↪v3/index.json -v 4.0.0-preview.662
```

NOTE: For these first two CLI options, make sure to install `AgentFramework.AspNetCore` and `AgentFramework.Core.Handlers` by replacing the package name above with these two names as well.

Visual Studio Nuget Package Manager:

- click on your new project WebAgent in the side bar
- go to the project tab in your menu, select `add nuget packages...`
- click on the dropdown that says nuget.org, and select `configure sources...`
- click add, and then name the new source `Agent Framework Beta` and paste the url `https://www.myget.org/F/agent-framework/api/v3/index.json` into the location field.
- check the `show pre-release packages` box on the bottom left
- choose the `Agent Framework Beta` source from the dropdown
- select the AgentFramework.AspNetCore package from the list. Make sure it is on a 4.0.0 version
- click add.

We will also use one other package from Nuget called `Jdenticon`. Add that from the nuget.org repository list.

8.1.4 Installing the libindy SDK on your computer

Windows

You can download binaries of libindy and all dependencies from the Sovrin repo. The dependencies 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.

MacOS

Check Setup Indy SDK build environment for MacOS.

Copy `libindy.a` and `libindy.dylib` to the `/usr/local/lib/` directory.

8.1.5 Configuring your own Agent

In this section, we'll walk through some sections of the WebAgent sample to understand how the AgentFramework can be used in a running application.

The WebAgent sample was created using the same steps as listed above. It may help to follow along to the steps below in your own new project. However, you may want to also open and try to run the fully working WebAgent sample in Visual Studio first to see if all the dependancies are working as they should. When the project was created, `Startup.cs` and `Program.cs` files were built using a template. These control how your webserver starts. We will need to edit them to use the agent framework.

Startup.cs

Our first goal is to edit the Startup file. Copy and paste the below code into your `Startup.cs` file:

Startup.cs (click to show/hide)

```
using System;
using AgentFramework.AspNetCore;
using AgentFramework.Core.Models.Wallets;
using Jdenticon.AspNetCore;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using WebAgent.Messages;
using WebAgent.Protocols.BasicMessage;
using WebAgent.Utils;

namespace WebAgent
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        // This method gets called by the runtime. Use this method to add services to the container.
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddMvc();

            services.AddLogging();

            // Register agent framework dependency services and handlers
            services.AddAgentFramework(builder =>
            {
                builder.AddBasicAgent<SimpleWebAgent>(c =>
                {
                    c.OwnerName = Environment.GetEnvironmentVariable("AGENT_NAME") ?? NameGenerator.GetRandomName();
```

(continues on next page)

(continued from previous page)

```
37                      c.EndpointUri = new Uri(Environment.GetEnvironmentVariable(
↪"ENDPOINT_HOST") ?? Environment.GetEnvironmentVariable("ASPNETCORE_URLS"));
38                      c.WalletConfiguration = new WalletConfiguration { Id =
↪"WebAgentWallet" };
39                      c.WalletCredentials = new WalletCredentials { Key = "MyWalletKey"␣
↪};
40                  });
41              });
42
43              // Register custom handlers with DI pipeline
44              services.AddSingleton<BasicMessageHandler>();
45              services.AddSingleton<TrustPingMessageHandler>();
46          }
47
48          // This method gets called by the runtime. Use this method to configure the␣
↪HTTP request pipeline.
49          public void Configure(IApplicationBuilder app, IHostingEnvironment env)
50          {
51              if (env.IsDevelopment())
52              {
53                  app.UseBrowserLink();
54                  app.UseDeveloperExceptionPage();
55              }
56              else
57              {
58                  app.UseExceptionHandler("/Home/Error");
59              }
60
61              app.UseStaticFiles();
62
63              // Register agent middleware
64              app.UseAgentFramework();
65
66              // fun identicons
67              app.UseJdenticon();
68              app.UseMvc(routes =>
69              {
70                  routes.MapRoute(
71                      name: "default",
72                      template: "{controller=Home}/{action=Index}/{id?}");
73              });
74          }
75      }
76  }
```

In this file, we congigure and add the Agent Framework to the project. If you are building the project from scratch, make sure to comment out lines 44-45 until you have created these services. Line 72 specifies how the API to trigger actions should be called. ————-

Program.cs

Next, we will edit the `Program.cs` file. Copy and paste this code too:

Program.cs

```
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;

namespace WebAgent
{
    public class Program
    {
        public static void Main(string[] args)
        {
            BuildWebHost(args).Run();
        }

        public static IWebHost BuildWebHost(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseKestrel()
                .UseStartup<Startup>()
                .Build();
    }
}
```

Once you have finished with this code, take a moment to look over the changes that we've made.

SimpleWebAgent.cs

Now create a file name `SimpleWebAgent.cs` in the main directory

This file will inherit from the AgentBase class in the AgentFramework, and it extends the IAgent Interface. This interface includes only one function named `Task<MessageResponse>ProcessAsync(IAgentContext context, MessageContext messageContext)` This will process any message that is sent to the agent's endpoint.

Copy and paste the below code into the file:

SimpleWebAgent.cs (Click to show)

```
using System;
using AgentFramework.Core.Handlers;
using WebAgent.Messages;
using WebAgent.Protocols.BasicMessage;

namespace WebAgent
{
    public class SimpleWebAgent : AgentBase
    {
        public SimpleWebAgent(IServiceProvider serviceProvider)
            : base(serviceProvider)
        {
        }

        protected override void ConfigureHandlers()
        {
            AddConnectionHandler();
            AddForwardHandler();
            AddHandler<BasicMessageHandler>();
            AddHandler<TrustPingMessageHandler>();
```

(continues on next page)

(continued from previous page)

```
        }
    }
}
```

bundleconfig.json

Create a bundleconfig.json file in your project root directory, and paste this json array into to it:

bundleconfig.json

```
// Configure bundling and minification for the project.
// More info at https://go.microsoft.com/fwlink/?LinkId=808241
[
  {
    "outputFileName": "wwwroot/css/site.min.css",
    // An array of relative input file paths. Globbing patterns supported
    "inputFiles": [
      "wwwroot/css/site.css"
    ]
  },
  {
    "outputFileName": "wwwroot/js/site.min.js",
    "inputFiles": [
      "wwwroot/js/site.js"
    ],
    // Optionally specify minification options
    "minify": {
      "enabled": true,
      "renameLocals": true
    },
    // Optionally generate .map file
    "sourceMap": false
  }
]
```

launchSettings.json

Edit the `Property/launchSettings.json`

launchSettings.json

```
{
  "profiles": {
    "WebAgent": {
      "commandName": "Project",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      },
      "applicationUrl": "http://localhost:5000/"
    }
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Finally, to get this program to run we will need to add a couple of utility files.

First, add a Utils folder. Add these two files to the Utils folder (Open these links in new tabs):

NameGenerator.cs Extensions.cs

Click run, you should see your template home page will appear in your web browser at http://localhost:5000. Congratulations! You've successully included the Agent framework into your project. Continue on to see how you might use it in your project.

8.2 WebAgent Walkthrough

We will learn how to create a web agent in this section. If you have created your own agent, you should be able to use that here. You may want to download the View files and import them into your project to save time on copy/paste: We're assuming that you are already familiar with the high level concepts of Indy and Sovrin.

The first thing is to understand about Wallets. Read about wallets here: **Insert link/paragraphs about wallet infra based on hipe:** Wallet HIPE

8.2.1 Opening a wallet in Agent Framework

We open a wallet in Agent Framework by provisioning an agent. This process of provisioning agents will create and configure an agent wallet and initialize the agent configuration. The framework will generate a random Did and Verkey, unless you specify `AgentSeed` which is used if you need determinism. Length of seed must be 32 characters.

```
await _provisioningService.ProvisionAgentAsync(
    new ProvisioningConfiguration
    {
        EndpointUri = "http://localhost:5000",
        OwnerName = "My Agent"
    });
```

We provision this agent in the `Startup.cs` file by building it directly with the AgentBuilder

HomeController.cs (Show)

```
using System;
using AgentFramework.AspNetCore;
using AgentFramework.Core.Models.Wallets;
using Jdenticon.AspNetCore;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using WebAgent.Messages;
using WebAgent.Protocols.BasicMessage;
using WebAgent.Utils;

```

(continues on next page)

(continued from previous page)

```
namespace WebAgent
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        // This method gets called by the runtime. Use this method to add services to the container.
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddMvc();

            services.AddLogging();

            // Register agent framework dependency services and handlers
            services.AddAgentFramework(builder =>
            {
                builder.AddBasicAgent<SimpleWebAgent>(c =>
                {
                    c.OwnerName = Environment.GetEnvironmentVariable("AGENT_NAME") ?? NameGenerator.GetRandomName();
                    c.EndpointUri = new Uri(Environment.GetEnvironmentVariable("ENDPOINT_HOST") ?? Environment.GetEnvironmentVariable("ASPNETCORE_URLS"));
                    c.WalletConfiguration = new WalletConfiguration { Id = "WebAgentWallet" };
                    c.WalletCredentials = new WalletCredentials { Key = "MyWalletKey" };
                });
            });

            // Register custom handlers with DI pipeline
            services.AddSingleton<BasicMessageHandler>();
            services.AddSingleton<TrustPingMessageHandler>();
        }

        // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
        public void Configure(IApplicationBuilder app, IHostingEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseBrowserLink();
                app.UseDeveloperExceptionPage();
            }
            else
            {
                app.UseExceptionHandler("/Home/Error");
            }

            app.UseStaticFiles();

            // Register agent middleware
```

(continues on next page)

(continued from previous page)

```
            app.UseAgentFramework();

            // fun identicons
            app.UseJdenticon();
            app.UseMvc(routes =>
            {
                routes.MapRoute(
                    name: "default",
                    template: "{controller=Home}/{action=Index}/{id?}");
            });
        }
    }
}
```

You now have a functioning wallet in your agent, ready to store all your secrets.

The Agent framework abstracts the main workflows of Indy into a state machine. For our agent, we will show how to create and receive invitations with other people. Although simple, it builds the foundation for all other potential agent communications like credentials and proofs.

TODO: Basic message and routing info

8.2.2 Connections

Every connection is a unique relationship with another agent. The Agent Framework represents this relationship with a `ConnectionRecord`, this entity describes the pairwise relationship with another party. The states for this record are:

- `Invited` - initially, when creating invitations to connect, the record will be set to this state.
- `Negotating` - set after accepting an invitation and sending a request to connect
- `Connected` - set when both parties have acknowledged the connection and have a pairwise record of each others DID's

For us to send basic messages back and forth, we will first need to establish that protocol. We will walk through how a message is processed in the WebAgent.

First, we need to define the basic message structure in the Protocols folder. These are all child classes of the AgentMessage class that is in the AgentFramework.

AgentMessage.cs (Show)

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.Linq;
using AgentFramework.Core.Exceptions;
using AgentFramework.Core.Extensions;
using Newtonsoft.Json;
using Newtonsoft.Json.Linq;

namespace AgentFramework.Core.Messages
{
    /// <summary>
    /// Represents an abstract base class of a content message.
```

(continues on next page)

(continued from previous page)

```
    /// </summary>
    public abstract class AgentMessage
    {
        /// <summary>
        /// Internal JObject representation of an agent message.
        /// </summary>
        [JsonIgnore]
        private IList<JProperty> _decorators = new List<JProperty>();

        /// <summary>
        /// Gets or sets the message id.
        /// </summary>
        /// <value>
        /// The message id.
        /// </value>
        [JsonProperty("@id")]
        public string Id { get; set; }

        /// <summary>
        /// Gets or sets the type.
        /// </summary>
        /// <value>
        /// The type.
        /// </value>
        [JsonProperty("@type")]
        public string Type { get; set; }

        /// <summary>
        /// Gets the decorators on the message.
        /// </summary>
        /// <returns>The decorators as a JObject.</returns>
        public IReadOnlyList<JProperty> GetDecorators() => new ReadOnlyCollection
↪<JProperty>(_decorators);

        /// <summary>
        /// Internal set method for setting the collection of decorators attached to␣
↪the message.
        /// </summary>
        /// <param name="decorators">JObject of decorators attached to the message.</
↪param>
        internal void SetDecorators(IList<JProperty> decorators) => _decorators =␣
↪decorators;

        /// <summary>
        /// Gets the decorator.
        /// </summary>
        /// <typeparam name="T"></typeparam>
        /// <param name="name">The name.</param>
        /// <returns></returns>
        /// <exception cref="AgentFrameworkException">ErrorCode.InvalidMessage Cannot␣
↪deserialize packed message or unable to find decorator on message.</exception>
        public T GetDecorator<T>(string name) where T : class
        {
            try
            {
                var decorator = _decorators.First(_ => _.Name == $"~{name}");
                return decorator.Value.ToObject<T>();
```

(continues on next page)

(continued from previous page)

```
            }
            catch (Exception e)
            {
                throw new AgentFrameworkException(ErrorCode.InvalidMessage, "Failed to extract decorator from message", e);
            }
        }

        /// <summary>
        /// Finds the decorator with the specified name or returns <code>null</code>.
        /// </summary>
        /// <typeparam name="T"></typeparam>
        /// <param name="name">The name.</param>
        /// <returns>The requested decorator or null</returns>
        public T FindDecorator<T>(string name) where T : class
        {
            var decorator = _decorators.FirstOrDefault(_ => _.Name == $"~{name}");
            return decorator?.Value.ToObject<T>();
        }

        /// <summary>
        /// Adds the decorator.
        /// </summary>
        /// <param name="decorator">The decorator.</param>
        /// <param name="name">The decorator name.</param>
        public void AddDecorator<T>(T decorator, string name) where T : class =>
            _decorators.Add(new JProperty($"~{name}", JsonConvert.DeserializeObject<JToken>(decorator.ToJson())));

        /// <summary>
        /// Sets the decorator overriding any existing values
        /// </summary>
        /// <typeparam name="T"></typeparam>
        /// <param name="decorator">The decorator.</param>
        /// <param name="name">The name.</param>
        public void SetDecorator<T>(T decorator, string name) where T : class
        {
            _decorators.Remove(_decorators.FirstOrDefault(x => x.Name == $"~{name}"));
            AddDecorator(decorator, name);
        }
    }
}
```

The BasicMessage uses the AgentMessage attributes as shown in the class above but adds "content" and "sent_time" attributes as well. This will allow something like text messages to be sent between agents.

BasicMessage.cs (Show)

```
using System;
using AgentFramework.Core.Messages;
using WebAgent.Messages;
using Newtonsoft.Json;

namespace WebAgent.Protocols.BasicMessage
{
    public class BasicMessage : AgentMessage
    {
```

(continues on next page)

(continued from previous page)

```
        public BasicMessage()
        {
            Id = Guid.NewGuid().ToString();
            Type = CustomMessageTypes.BasicMessageType;
        }

        [JsonProperty("content")]
        public string Content { get; set; }

        [JsonProperty("sent_time")]
        public string SentTime { get; set; }
    }
}
```

When your agent receives a message, it gets put directly into the wallet by the basic message handler.

BasicMessageHandler.cs (Show)

```
using System;
using System.Threading.Tasks;
using AgentFramework.Core.Contracts;
using AgentFramework.Core.Handlers;
using AgentFramework.Core.Messages;

namespace WebAgent.Protocols.BasicMessage
{
    public class BasicMessageHandler : MessageHandlerBase<BasicMessage>
    {
        private readonly IWalletRecordService _recordService;

        public BasicMessageHandler(IWalletRecordService recordService)
        {
            _recordService = recordService;
        }

        protected override async Task<AgentMessage> ProcessAsync(BasicMessage message,
↪ IAgentContext agentContext, MessageContext messageContext)
        {
            Console.WriteLine($"Processing message by {messageContext.Connection.Id}
↪");

            await _recordService.AddAsync(agentContext.Wallet, new BasicMessageRecord
            {
                Id = Guid.NewGuid().ToString(),
                ConnectionId = messageContext.Connection.Id,
                Text = message.Content,
                SentTime = DateTime.TryParse(message.SentTime, out var dateTime) ?␣
↪dateTime : DateTime.UtcNow,
                Direction = MessageDirection.Incoming
            });

            return null;
        }
    }
}
```

Then when the view is reloaded, the Controller class takes over.

This is following the MVC pattern of ASPNetCore.

Here is the ConnectionsController class that will show a BasicMessage if one is sent or received:

ConnectionsController.cs (Show)

```
using System;
using System.Globalization;
using System.Reactive.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using System.Web;
using AgentFramework.Core.Contracts;
using AgentFramework.Core.Extensions;
using AgentFramework.Core.Handlers;
using AgentFramework.Core.Handlers.Agents;
using AgentFramework.Core.Messages.Connections;
using AgentFramework.Core.Models;
using AgentFramework.Core.Models.Connections;
using AgentFramework.Core.Models.Events;
using AgentFramework.Core.Models.Records.Search;
using AgentFramework.Core.Utils;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Options;
using Newtonsoft.Json;
using WebAgent.Messages;
using WebAgent.Models;
using WebAgent.Protocols.BasicMessage;

namespace WebAgent.Controllers
{
    public class ConnectionsController : Controller
    {
        private readonly IEventAggregator _eventAggregator;
        private readonly IConnectionService _connectionService;
        private readonly IWalletService _walletService;
        private readonly IWalletRecordService _recordService;
        private readonly IProvisioningService _provisioningService;
        private readonly IAgentProvider _agentContextProvider;
        private readonly IMessageService _messageService;
        private readonly WalletOptions _walletOptions;

        public ConnectionsController(
            IEventAggregator eventAggregator,
            IConnectionService connectionService,
            IWalletService walletService,
            IWalletRecordService recordService,
            IProvisioningService provisioningService,
            IAgentProvider agentContextProvider,
            IMessageService messageService,
            IOptions<WalletOptions> walletOptions)
        {
            _eventAggregator = eventAggregator;
            _connectionService = connectionService;
            _walletService = walletService;
            _recordService = recordService;
            _provisioningService = provisioningService;
            _agentContextProvider = agentContextProvider;
```

(continues on next page)

(continued from previous page)

```
            _messageService = messageService;
            _walletOptions = walletOptions.Value;
        }

        [HttpGet]
        public async Task<IActionResult> Index()
        {
            var context = await _agentContextProvider.GetContextAsync();

            return View(new ConnectionsViewModel
            {
                Connections = await _connectionService.ListAsync(context)
            });
        }

        [HttpGet]
        public async Task<IActionResult> CreateInvitation()
        {
            var context = await _agentContextProvider.GetContextAsync();

            var (invitation, _) = await _connectionService.
↪CreateInvitationAsync(context, new InviteConfiguration { AutoAcceptConnection =␣
↪true });
            ViewData["Invitation"] = $"{(await _provisioningService.
↪GetProvisioningAsync(context.Wallet)).Endpoint.Uri}?c_i=
↪{EncodeInvitation(invitation)}";
            return View();
        }

        [HttpPost]
        public async Task<IActionResult> AcceptInvitation(AcceptConnectionViewModel␣
↪model)
        {
            var context = await _agentContextProvider.GetContextAsync();

            var invite = MessageUtils.DecodeMessageFromUrlFormat
↪<ConnectionInvitationMessage>(model.InvitationDetails);
            var (request, record) = await _connectionService.
↪CreateRequestAsync(context, invite);
            await _messageService.SendAsync(context.Wallet, request, record, invite.
↪RecipientKeys[0]);

            return RedirectToAction("Index");
        }

        [HttpPost]
        public IActionResult ViewInvitation(AcceptConnectionViewModel model)
        {
            if (!ModelState.IsValid)
            {
                return Redirect(Request.Headers["Referer"].ToString());
            }

            ViewData["InvitationDetails"] = model.InvitationDetails;

            var invite = MessageUtils.DecodeMessageFromUrlFormat
↪<ConnectionInvitationMessage>(model.InvitationDetails);
```

(continues on next page)

(continued from previous page)

```
102
103             return View(invite);
104         }
105
106         [HttpPost]
107         public async Task<IActionResult> SendTrustPing(string connectionId)
108         {
109             var context = await _agentContextProvider.GetContextAsync();
110             var connection = await _connectionService.GetAsync(context, connectionId);
111             var message = new TrustPingMessage
112             {
113                 ResponseRequested = true,
114                 Comment = "Hello"
115             };
116
117             var slim = new SemaphoreSlim(0, 1);
118             var success = false;
119
120             using (var subscription = _eventAggregator.GetEventByType
    ↪<ServiceMessageProcessingEvent>()
121                 .Where(_ => _.MessageType == CustomMessageTypes.
    ↪TrustPingResponseMessageType)
122                 .Subscribe(_ => { success = true; slim.Release(); }))
123             {
124                 await _messageService.SendAsync(context.Wallet, message, connection);
125
126                 await slim.WaitAsync(TimeSpan.FromSeconds(5));
127
128                 return RedirectToAction("Details", new { id = connectionId,␣
    ↪trustPingSuccess = success });
129             }
130         }
131
132         [HttpGet]
133         public async Task<IActionResult> Details(string id, bool? trustPingSuccess =␣
    ↪null)
134         {
135             var context = new AgentContext
136             {
137                 Wallet = await _walletService.GetWalletAsync(_walletOptions.
    ↪WalletConfiguration,
138                     _walletOptions.WalletCredentials)
139             };
140
141             var model = new ConnectionDetailsViewModel
142             {
143                 Connection = await _connectionService.GetAsync(context, id),
144                 Messages = await _recordService.SearchAsync<BasicMessageRecord>
    ↪(context.Wallet,
145                     SearchQuery.Equal(nameof(BasicMessageRecord.ConnectionId), id),␣
    ↪null, 10),
146                 TrustPingSuccess = trustPingSuccess
147             };
148
149             return View(model);
150         }
151
```

(continues on next page)

(continued from previous page)

```
152         [HttpPost]
153         public async Task<IActionResult> SendMessage(string connectionId, string text)
154         {
155             if (string.IsNullOrEmpty(text)) return RedirectToAction("Details", new {␣
↪id = connectionId });
156 
157             var context = new AgentContext
158             {
159                 Wallet = await _walletService.GetWalletAsync(_walletOptions.
↪WalletConfiguration,
160                     _walletOptions.WalletCredentials)
161             };
162 
163             var sentTime = DateTime.UtcNow;
164             var messageRecord = new BasicMessageRecord
165             {
166                 Id = Guid.NewGuid().ToString(),
167                 Direction = MessageDirection.Outgoing,
168                 Text = text,
169                 SentTime = sentTime,
170                 ConnectionId = connectionId
171             };
172             var message = new BasicMessage
173             {
174                 Content = text,
175                 SentTime = sentTime.ToString("s", CultureInfo.InvariantCulture)
176             };
177             var connection = await _connectionService.GetAsync(context, connectionId);
178 
179             // Save the outgoing message to the local wallet for chat history purposes
180             await _recordService.AddAsync(context.Wallet, messageRecord);
181 
182             // Send an agent message using the secure connection
183             await _messageService.SendAsync(context.Wallet, message, connection);
184 
185             return RedirectToAction("Details", new {id = connectionId});
186         }
187 
188         [HttpPost]
189         public IActionResult LaunchApp(LaunchAppViewModel model)
190         {
191             return Redirect($"{model.UriSchema}{Uri.EscapeDataString(model.
↪InvitationDetails)}");
192         }
193 
194         /// <summary>
195         /// Encodes the invitation to a base64 string which can be presented to the␣
↪user as QR code or a deep link Url
196         /// </summary>
197         /// <returns>The invitation.</returns>
198         /// <param name="invitation">Invitation.</param>
199         public string EncodeInvitation(ConnectionInvitationMessage invitation)
200         {
201             return invitation.ToJson().ToBase64();
202         }
203 
204         /// <summary>
```

(continues on next page)

(continued from previous page)

```
205         /// Decodes the invitation from base64 to strongly typed object
206         /// </summary>
207         /// <returns>The invitation.</returns>
208         /// <param name="invitation">Invitation.</param>
209         public ConnectionInvitationMessage DecodeInvitation(string invitation)
210         {
211             return JsonConvert.DeserializeObject<ConnectionInvitationMessage>
    ↪(Encoding.UTF8.GetString(Convert.FromBase64String(invitation)));
212         }
213     }
214 }
```

Read through the rest of the ConnectionsController class to understand how the Agent Framework can be used to implement the other website features.

CHAPTER 9

Samples

9.1 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.

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

9.1.1 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.

9.1.2 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
```

9.2 Mobile Agent with Xamarin Forms

9.3 Docker container example

9.3.1 Running the example

At the root of the repo run:

```
docker-compose up
```

This will create an agent network with a pool and two identical agents able to communicate with each other in the network. Navigate to http://localhost:7000/ and http://localhost:8000/ to create and accept connection invitations between the different agents.

9.3.2 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*

CHAPTER 10

Common Errors and Problems

10.1 System.DllNotFoundException

Problem Runtime exception thrown

```
System.DllNotFoundException : Unable to load shared library 'libindy' or one of its␣
↪dependencies. In order to help diagnose loading problems, consider setting the DYLD_
↪PRINT_LIBRARIES environment variable: dlopen(libsovtoken, 1): image not found
```

Solution Missing static library. Check the installation section for guidance on how to add static libraries to your environment.