An introduction to agent-oriented programming with AgentDotNet

Contents

Preface

The programming of computers has a story as long as the life of computers. In the beginning, programmers were programming computers by tools such as punch cards, by use of zeros and ones. After that, assembly codes were introduced, codes which made programming much faster and easier. Then, other languages like Pascal and C at the top of this stack appeared which introduces modular programming. Finally, programmers went through object-oriented programming, the paradigm now is used as much as possible.

The new paradigm we are going to talk about alongside presentation of our framework AgentDotNet is agent-oriented programming (AOP). AOP is based on the concept of Agents instead of objects. Actually, agents are objects which have specific behaviours which make them different than simple objects.

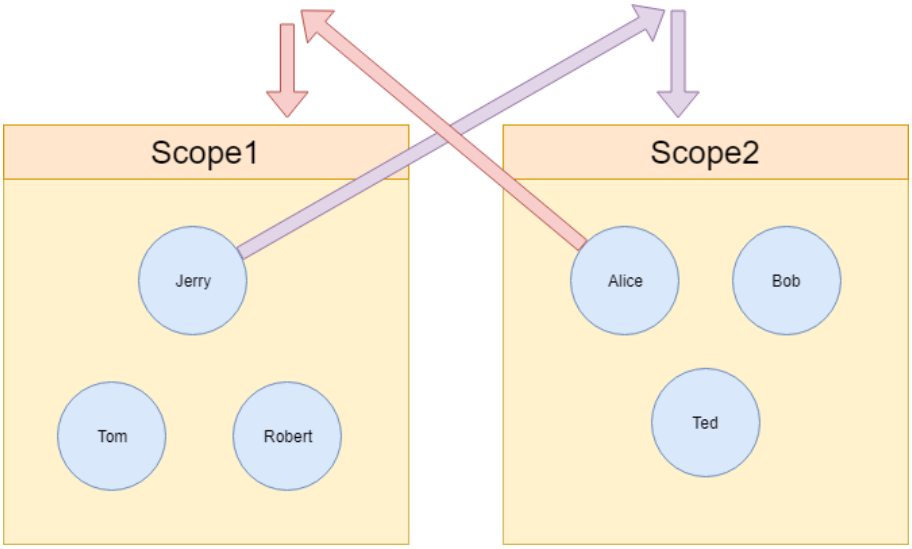
Our framework AgentDotNet offers a set of classes and tools for creating agents, define their behaviours and facilitate communication between them. In this document, we are going to describe technical details of AgentDotNet framework and how to use it by giving a famous example named BookTrading which is used to give tutorials about JADE framework.

AgentDotNet overview

In the AgentDotNet framework, there are some important definitions. The first one is **Agent.** An **Agent** is a class which extends from the original Agent class which is predefined in the framework library and overrides its behaviours. In fact, we code the problem supposed to solve by defining agents and using them to communicate with each other. A sample agent definition comes below.

```
public class HelloWorldAgent : AgentDotNet.Agent
{
    public HelloWorldAgent (string Name) : base(Name)
    {
    }

    protected override void Begin()
    {
        Console.WriteLine("{0} is starting now", Uri.ToString());
    }
    protected override vodi Finish()
    {
        Console.WriteLine("{0} is shutting down now", Uri.ToString());
    }
}
```

The second definition is **Scope**. A **Scope** is a class in which agents are bound to it. In fact, a **Scope** is an environment agents live in and it's a higher entity which wraps agents and makes a bridge to outer environments.



In the above picture, we can see a diagram describing relations between agents in Scope1 and agents Scope2. The main idea is that if an agent wants to send a message to another one in its scope or other scopes, the agent must ask it from scope recipient agent living in. For example, if Jerry in Scope1 wants to send a message to Alice in Scope2, it must send the message to Scope2 and Alice receive it from its scope. Also, when Alice in Scope2 wants to send the reply to Jerry in Scope1, it must send the message to Jerrys scope, i.e Scope1. Also, if one agent in Scope1 wants to send a message to another agent in Scope1, yet it's necessary to ask it from agents scope, i.e Scope1.

At the first look, the proposed method for communication between agents may look over complex and the reader may think that it could be done simpler by using direct communication between agents. But if we consider that the communication is done by the network protocol TCP/IP, the importance of the proposed method can be realized. The advantage of the proposed method is that we can give the IP and port to the scope and access to agents by meaningful names. As an example, we can access an agent by its identifier Jerry@127.0.0.1:8080, this phrase tries to point to an agent named Jerry living at scope addressed by 127.0.0.1:8080. An agent can be instanced and bound to a scope the way following shows.

```
class Program
{
    static void Main(string[] args)
    {
        //It is possible to bind agents to scope when the scope is constructing
        var mainScope = new Scope(Port: scopePort, Agents: new List<Agent>()
        {
            new HelloWorldAgent("MyHelloWorldAgent"),
        });

        //It is possible to bind agents to scope after creating scope at runtime
dynamicly
        var secondHelloWorldAgent = new HelloWorldAgent("Second One");
        secondHelloWorldAgent.Bind(mainScope);

        mainScope.Start();

        Console.WriteLine("Press any key to stop things...");
        Console.ReadKey();

        mainScope.Stop();
    }
}
```

Now as the third and four definitions we introduce classes are used to identify and address scopes and agents. We have two classes named ScoperUri and AgentUri which their data structure can be seen below.

```
class ScopeUri
{
    IPAddress ScopeAddress;
    int ScopePort;
}

class AgentUri
{
    ScopeUri ScopeUri;
    string AgentName;
}
```

As these data structures represent, ScopeUri class is made up of two fields, ScopeAddress is an IPAddress which refers to IPAddress of the machine the scope is running on it, and ScopePort which indicates the port the scope listens to.

The AgentUri class is made up of two fields too, ScopeUri refers to the identifier of the scope in which the agent is running on it, and the AgentName is an obvious field which refers to the name of the agent.

The fifth definition is Message. A Message is the data structure which is used to transfer messages between agents. In fact, it is a wrapper for the raw message the agent will send or receive. The data structure of a Message can be seen below.

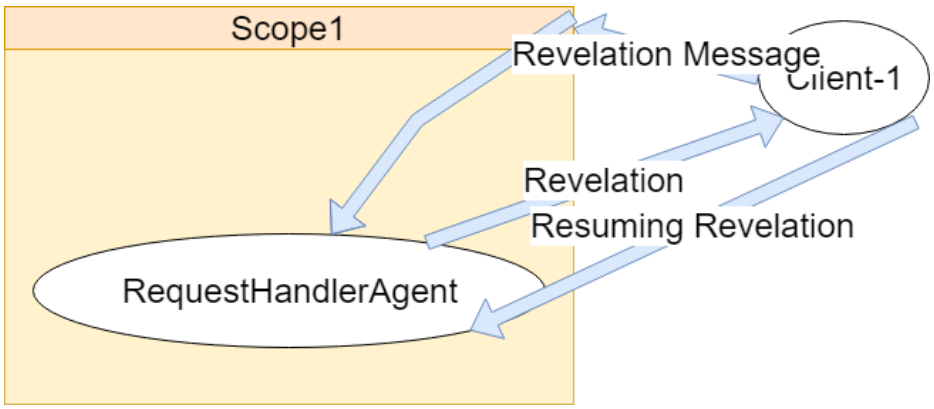
```
namespace AgentDotNet
{
    public enum MessageType { Unicast, Broadcast, Revelation }
    public class Message
    {
        public string Nonce;

        public string Session;

        public string Body;

        public MessageType Type;

        public List<ScopeUri> RecieverScopes;

        public List<AgentUri> RecieverAgents;

        public AgentUri Sender;

        public Socket Socket;
    }
}
```

As the data structure represents, any Message could be in one of three types of Unicast, Broadcast and Revelation. A message of type Unicast is a message which has specific receiver agents as a list, the message could be

received by specified agents. A message of type Broadcast has no specific receiver agents, instead it has a specific list of receiver scopes, the message will be received by all of the agents live in specified scopes.

So far, two types of messages, Unicast and Broadcast use default described method in previous sections, but the message of type Revelation differs a bit. Just like the Unicast messages, the Revelation ones will be sent to receiver agents scope, but the response of the message, instead of receiving by the scope and then passing to owner agent, it will be received directly. The reason of existing such type of message is that sometimes agents are used as temporary clients that do not belong to any scopes, because there is no need to start a scope listening to a specific port at all or it's important to us to keep the connection open. Consider following client-server scenario for better understanding.

We want to build an agent-based server application which returns the files requested by clients. Also, we want to write client application in an agent-oriented manner. Thus we have a scope for server app which listens to requests and passes them two RequestHandlerAgent, then RequestHandlerAgent parses the request and returns the file. In the client app, we have no scope listening to anything. Instead, a client app is just a ClientAgent which request for a file from the server app by asking its request from RequestHandlerAgent@serverAddress:scopePort. Now the problem is that whom should RequestHandlerAgent return file to? Since there is no scope ready for receiving a response, the RequestHandlerAgent should return the answer by Revelation.



In fact, a message of type Revelation helps to find targeting agent and after that communicate with it directly by filling the Socket field of the message.

The field Body is the exact message we want to send or receive and the Sender field indicates the agent which has sent this message. Two reminder fields Nonce and Session are random strings generated while creating the message. The field Nonce is a unique id for the message. The field Session is used in sending and receiving messages with type Unicast or Broadcast and is useful for recognizing replies to the sending message.

How do agents communicate?

AgentDotNet framework provides easy approaches for communication. To an agent communicate with another one, it can use built-in methods AgentDoNet provides to transfer Message object described previously.

To send messages, we can categorize them into two major groups. First, those which only carry a message and do not anticipate any response. For this purpose, AgentDotNet provides two methods **Say** and **Lecture**. The method **Say** sends a messages to specified list of receiver agents. It is considered as unicast communication as it only sends message from one specified source to obviously specified receivers.

The method **Lecture** is just like the method **Say** with a small difference. Instead of having a list of agents as receivers, there is list of scopes as receivers. In this approach we don't have any idea that what will be our receiver agent, Instead, we send the message to scopes containing agents and the receiver scope will broadcast the message to all of agents live in it.

The second group of messages are those which anticipate a response from the receiver. Messages like "What is your name?". In fact in this type of messages we are asking a thing from one or a group. Thus AgentDotNet provides two methods called **Ask** and **Query**. The method **Ask** is just like the method **Say,** in fact we **Ask** somebody, we are saying things to him/her and then wait for response. Also the method **Query** asks a message from a list of scopes, instead of list agents. The prototype of the methods can be seen below.

```
public Message Reavel(string sentence, AgentUri reciever)
public async Task<Message> ReavelAsync(string sentence, AgentUri reciever)
public void Say(string sentence, AgentUri reciever)
public void Lecture(string sentence, ScopeUri recievers)
public void Lecture(string sentence, List<AgentUri> recievers)
public void Lecture(string sentence, List<ScopeUri> recievers)
public Message Ask(string question, AgentUri reciever, uint timeOut=0)
public Message Query(string question, ScopeUri reciever, uint timeOut = 0)
public Message Query(string question, List<AgentUri> recievers, uint timeOut = 0)
public Message Query(string question, List<ScopeUri> recievers, uint timeOut = 0)
public async Task<Message> AskAsync(string question, AgentUri reciever, uint timeOut = 0)
```

```
public async Task<Message> QueryAsync(string question, ScopeUri reciever, uint timeOut = 0)
public async Task<Message> QueryAsync(string question, List<AgentUri> recievers, uint timeOut =
0)
public async Task<Message> QueryAsync(string question, List<ScopeUri> recievers, uint timeOut =
0)
```

In the above prototypes, there is one another method called **Revelation**. As we described before, the method Revelation is used when we need a direct connection among agents.

The book trading example

In this example we are going to design and write an application containing agents cooperating to buy and sell books. The project has three type of agents. The first one is **BookSellerAgent**, agents of this type have a list of books with their prices, waiting for requests for a book to sell. The second type is **BookBuyerAgent**, this is the agent which asks seller agents to buy desired books. The third type is **ServiceProviderAgent**, this agent acts like yellow pages in JADE framework. Every seller agent registers itself on **ServerProviderAgent**, so **BookBuyerAgent** can find seller agents and buy from found service providers. Also there are two simple data structures, **Book** and **ServiceRecord**. The **Book** data structure has two fields Name and Price of book. And The **ServiceRecord** has two fields ServiceName and Uri, the Uri field refers to identifier of agent which provides service mentioned by ServiceName. Now, we are going to explain the code of agents.

ServiceProviderAgent

This is the agent services provided by other agents are registered on it. To construct such an agent, first of all we need a list of ServiceRecords, that is explained before. Thus, the constructor of this agent will look like to below code.

```
class ServiceProviderAgent : Agent
{
    public List<ServiceRecord> ServiceRecords { get; set; }
    public ServiceProviderAgent(string Name) : base(Name)
    {
        ServiceRecords = new List<ServiceRecord>();
    }
}
```

After constructing the agent, we need to override two important methods. This is how we write Begin method.

```
protected override void Begin()
{
    Console.WriteLine($"Starting agent {Name}...");
    while(true)
    {
        var message = Recieve();
        dynamic json = JObject.Parse(message.Body);
        if(json.Command == "register")
        {
            var serviceName = (string)json.ServiceName;
            var uri = (string)json.Uri;

            Console.WriteLine($"{uri} registered as {serviceName}");

            if (ServiceRecords
                 .Any(q => q.ServiceName == serviceName && q.Uri.ToString() == Uri.ToString())
                 ) continue;
            else
            {
                ServiceRecords.Add(new ServiceRecord
                {
                    ServiceName = serviceName,
                    Uri = uri
                });
            }
        }
        else if(json.Command == "remove")
        {
            var serviceName = (string)json.ServiceName;
            var Uri = (string)json.Uri;
            ServiceRecords
                 .RemoveAll(
                 q => q.ServiceName == serviceName && q.Uri.ToString() == Uri.ToString()
                 );
        }
        else if(json.Command == "query")
        {
            var serviceName = (string)json.ServiceName;
            var Uris = ServiceRecords
                        .Where(q => q.ServiceName == serviceName)
                        .Select(q => q.Uri.ToString()).Distinct();
            message.CreateReply(Uri, JsonConvert.SerializeObject(Uris)).Send();
        }
    }
}
```

The code in above represents that when the agent starts, after printing a simple message, in a loop, it always listens to receive requests. It's important to know that all of its requests are formatted in JSON. After parsing the receiving JSON request, there is a simple IF statement with three branches.

The command **register** is used to register agents with services they provide. The command **remove** is used to remove registered services for each agent and finally the command **query** is used to ask **ServiceProviderAgent** who provides services an agent wants. From the code it is understood that if we need to reply a message, we can use its **CreateReply** method.

BookBuyerAgent

This is the agent which buys the books. The constructor takes the name of book we want to agent buy and the **Begin** method does the job. Let's take a look at the code.

```
class BookBuyerAgent : Agent
{
  public string BookToBuy { get; set; }

  public BookBuyerAgent(string Name, string BookToBuy) : base(Name)
  {
    this.BookToBuy = BookToBuy;
  }

  protected override void Begin()
  {
    Thread.Sleep(500);
    Console.WriteLine($"Starting agent {Name}...");
    var response =
                    Ask
                    (
                        JsonConvert.SerializeObject(new
                        {
                            Command = "query",
                            ServiceName = "BookSeller"
                        }),
                        new AgentUri("ServiceProviderAgent"),
                        timeOut: 5000
                    );
```

```
if(response == null)
{
   Console.WriteLine($"Agent {Uri} failed to do anything!");
   return;
}
var uris = JsonConvert
                        .DeserializeObject<IEnumerable<string>>(response.Body);
string targetUri = "";
decimal targetPrice = decimal.MaxValue;
foreach (var uri in uris)
{
   Console.WriteLine($"Asking for {BookToBuy} from {uri}");
   var r =
                        Ask(
                                JsonConvert.SerializeObject(
                                new
                                {
                                 Command = "query",
                                BookName = BookToBuy
                                }),
                                new AgentUri(uri), 5000);
   if (r == null) Console.WriteLine($"{uri} has no such book");
   else
   {
      dynamic bookInfo = JObject.Parse(r.Body);

      if (bookInfo.Command == "not found")
      {
         Console.WriteLine($"{uri} has no such book");
         continue;
      }

      string buyerUri = (string)r.Sender.ToString();
      decimal bookPrice = (decimal)bookInfo.BookPrice;

      if (bookPrice < targetPrice)
      {
         targetUri = buyerUri;
         targetPrice = bookPrice;
      }

      Console
                                .WriteLine($"{uri} has {bookInfo.BookName} with price
                        {bookInfo.BookPrice}");
   }
```

```
        }

        if (!string.IsNullOrEmpty(targetUri))
        {
            Console.ForegroundColor = ConsoleColor.Green;
            Console
                              .WriteLine($"BestPrice={targetPrice}, BestSeller={targetUri.ToString()}");
            var r = Ask(JsonConvert.SerializeObject(new {
                                          Command = "accept",
                                         BookName = BookToBuy,
                                          BookPrice = targetPrice }),
                                         new AgentUri(targetUri), 5000);

            dynamic resultInfo = JObject.Parse(r.Body);
            string command = (string)resultInfo.Command;

            if (command == "your welcome")
            {
                Console.ForegroundColor = ConsoleColor.Yellow;
                Console.WriteLine("Done!");
            }
            else
            {
                Console.ForegroundColor = ConsoleColor.Magenta;
                Console.WriteLine("OOPS!");
            }
        }
        else
        {
            Console.ForegroundColor = ConsoleColor.Red;
            Console.WriteLine("Oops! Couldn't find anything!");
        }
    }

    protected override void Finish()
    {
        Console.WriteLine($"Shutting down agent {Name}...");
    }

}
```

The code first asks the ServiceProviderAgent agents are BookSeller and the ServiceProviderAgent replies the list. After receiving the list of

BookSeller agents, the BookBuyerAgent asks each of them price of the book it want to buy and stores the cheapest one. By finishing the query, the BookBuyerAgent asks the seller which offers the best price and buys the book.

BookSellerAgent

This is the agent has a list of books and sells them. The constructor takes a list of books. The code can be seen below.

```
public class BookSellerAgent : Agent
  {
    public List<Book> Books { get; set; }
    public BookSellerAgent(string Name):base(Name) { }

    protected override void Begin()
    {
      Thread.Sleep(100);
      Console.WriteLine($"Starting agent {Name}...");

      Say(JsonConvert.SerializeObject(new
                {
                Command = "register",
                ServiceName = "BookSeller",
                Uri = Uri.ToString()
                }),
                new AgentUri("ServiceProviderAgent"));

      while (true)
      {
        var message = Recieve();

        dynamic json = JObject.Parse(message.Body);
        if (json.Command == "query")
        {
          var bookName = (string)json.BookName;
          var book = Books.FirstOrDefault(q => q.Name == bookName);
          if (book == null) message.CreateReply(Uri,
                                JsonConvert.SerializeObject(new
                                { Command = "not found" }))
                                .Send();
                                else message.CreateReply(Uri, JsonConvert.SerializeObject(new {
                                Command = "price", BookName = book.Name, BookPrice = book.Price
                                })).Send();
        }
        else if (json.Command == "accept")
```

```
            {
                var bookName = (string)json.BookName;
                var bookPrice = (decimal)json.BookPrice;
                var book = Books
                                        .FirstOrDefault(
                                        q => q.Name == bookName && q.Price == bookPrice);
                Books.Remove(book);
                message.CreateReply(Uri, JsonConvert.SerializeObject(new
                                        {
                                                Command = "your welcome",
                                                BookName = bookName,
                                                BookPrice = bookPrice }))
                                                .Send();
            }
            else message.CreateReply(Uri, "undefined request").Send();
        }
    }

    protected override void Finish()
    {
        Console.WriteLine($"Shutting down agent {Name}...");
    }
}
```

The BookSellerAgent, by starting, first of all registers itself into ServiceProviderAgent as a book seller. Then, in a loop waits for messages from BookBuyerAgents and depending on what it receives behaves. It supports two type of messages “Query” and “Accept”. Message of type “Query” is used to search if this agent has desired book and if it has how much does it cost. Message of type “Accept” is sent to seller agent when the proposals has been accepted from buyer agent and it really wants to buy the book from targeted agent.

Main program

After creating agents, we need to gather and bind them all to a scope. This is done in the below code.

```
class Program
  {
    static void Main(string[] args)
    {
      var scope = new Scope(Agents: new List<Agent>
```

```
        {
            new Agents.ServiceProviderAgent("ServiceProviderAgent"),

            new Agents.BookSellerAgent("SellerAgent1")
            {
                Books = new List<Book>
                {
                    new Book{ Name = "Harry potter", Price = 25000 },
                    new Book{ Name = "Lord of the rings", Price = 50000 },
                    new Book{ Name = "The hobbit", Price = 45000 }
                }
            },

            new Agents.BookSellerAgent("SellerAgent2")
            {
                Books = new List<Book>
                {
                    new Book{ Name = "Harry potter", Price = 25500 },
                    new Book{ Name = "Lord of the rings", Price = 49000 },
                }
            }
        });

        scope.Start();

        var bookBuyerAgent = new Agents
                                        .BookBuyerAgent("BookBuyerAgent", "Harry potter");
        bookBuyerAgent.Bind(scope);
        bookBuyerAgent.Start();

        Thread.Sleep(2000);
        Console.ResetColor();
        var bookBuyerAgent2 = new Agents.BookBuyerAgent("BookBuyerAgent2", "Harry potter");
        bookBuyerAgent2.Bind(scope);
        bookBuyerAgent2.Start();

        Console.ReadKey();
    }
}
```

In this code, we create a scope by use of its constructor, we provide **BookSeller** agents by instancing them inline. After creating the scope, we

can easily start it to let environment run. Also we can register agent BookBuyerAgent after creating the scope.

Conclusion

In this article, we introduced a framework named AgentDotNet and described how it works. We talked about architecture of AgentDotNot, Scopes, Agents and some other entities. Also we talked about how agents communicate. After these we explored an example called book trading example which shows us how to use agent programming to buy books with an agent from other agents which sell books. As future works, we consider to develop this framework by use of python language. Also we have the desire to add the ability to develop mobile agents to the proposed framework.