Tangle Net Documentation

Release stable

Jul 03, 2019

Contents

This is an inoffical .NET Standard 2.0 and .NET Framework 4.6.1 port of the IOTA Client library.

It implements all standard API calls, as described in the API documentation (https://docs.iota.org/docs/iri/0.1/references/api-reference) and the extended methods for signing, sending and receiving bundles.

CHAPTER 1

Getting Started

1.1 Installation

Tangle.Net is compatible with .NET Standard 2.0 and .NET Framework 4.6.1.

You can install the packages via nuget

```
https://www.nuget.org/packages/Tangle.Net/
https://www.nuget.org/packages/Tangle.Net.Standard/
```

The most simple way to start using the library is by instantiating the repository via factory. Note that you could also create instances of the repository via any dependency injection framework.

```
var repository = new RestIotaRepository(new RestClient("https://localhost:14265"));
var nodeInfo = repository.GetNodeInfo();
var neighbours = repository.GetNeighbors();
```

CHAPTER 2

Working with the Core

2.1 Dependencies

The library currently uses RestSharp for all it's API calls and the official C# Bouncy Castle Port for encryption.

2.2 Entities

Since the library emphasizes an object orientated approach combined with Clean Architecture there are a few objects you have to be familiar with in order to work effectively.

2.3 TryteString

A TryteString is the ASCII representation of a sequence of trytes. Please note that only the letters A-Z and the number 9 are allowed. The following regular expression verifies if a given string is a TryteString or not.

```
^[9A-Z]*$
```

As the TryteString is only the basic class you will propably stumble accross many references for its subclasses:

- Address

  If you want to sign an input, you should generate the address with the AddressGenerator. That way the private key will be generated aswell, which is needed to sign the transaction. For any other cases you can simply instantiate the address with its trytes.

- Checksum

  Checksum for a given address

- Digest

  Digest to a generated private key for an address. Will be generated along with an address and its private key

- Fragment

  Payload part of a transaction. Can either contain parts of the signature or carry data

- Hash

  81 trytes long hash to a bundle, transaction or similar

- Seed

  81 trytes long "master key". Used to derive addresses and private keys

- Tag

  27 trytes long part of a transaction. Can be set to any value

- TransactionTrytes

  2673 trytes long. Represents a transaction as trytes

2.4 Bundle

A bundle represents a set of transactions that have been or should be attached to the tangle. The entity itself is able to finalize and sign bundles in order to send them to the tangle.

Note that you technically can create all transactions on a bundle manually, but that is not neccessary and more error prone.

The simpliest way to send a bundle to the tangle is to use a bundle without value

```
var bundle = new Bundle();
bundle.AddTransfer(new Transfer
                    {
                        Address = new Address(Hash.Empty.Value),
                        Tag = new Tag("TANGLENET"),
                        Timestamp = Timestamp.UnixSecondsTimestamp,
                        Message = TryteString.FromUtf8String("Hello #1!")
                    });

bundle.AddTransfer(new Transfer
                    {
                        Address = new Address(Hash.Empty.Value),
                        Tag = new Tag("TANGLENET"),
                        Timestamp = Timestamp.UnixSecondsTimestamp,
                        Message = TryteString.FromUtf8String("Hello #2!")
                    });

bundle.Finalize();
bundle.Sign();

// see Getting Started to see how a repository is created
repository.SendTrytes(bundle.Transactions);
```

2.5 Transaction

As transactions are generated within a bundle or are received via API call you won't need to create them manually (but you can if you want to).

2.6 Addresses

Addresses in IOTA are derived deterministically from your seed. That means that you can access your funds everywhere as long as you know your seed.

Please note that anyone with access to your seed, also has access to your funds. More on security here: https://blog.iota.org/the-secret-to-security-is-secrecy-d32b5b7f25ef

Never ever use an online seed generator

2.7 Generating an address

```
var seed = new Seed("SOMESEEDHERE")
var addressGenerator = new AddressGenerator(seed, SecurityLevel.Medium);
var address = addressGenerator.GetAddress(0);
var addresses = addressGenerator.GetAddresses(0, 10);
```

When you generate an address you will need to provide an index. Since addresses are generated deterministically the first address index will always result in the same address. For generating more than one address use the GetAddresses method, provided with a count.

2.8 Security Levels

The higher the security level the longer the private key for your address (used to sign spending of funds) will be. Even though address generation is deterministically a different security level will result in a different address even if the index is the same.

There currently are three security levels (range 1-3). You can either use the numbers directly of access them via the SecurityLevel class.

2.9 Clients

For most use cases it should be fine to instantiate the repository as displayed in Getting Started.

Anyway it sometimes may be useful to have some kind of fallback mechanism in place to handle unresponsive or out of sync nodes. To handle this you can use the fallback client, that will handle node errors internally.

```
var repository = new RestIotaRepository(
new FallbackIotaClient(
    new List<string>
    {
        "https://invalid.node.com:443",
        "https://peanut.iotasalad.org:14265",
        "http://node04.iotatoken.nl:14265",
        "http://node05.iotatoken.nl:16265",
    },
    5000),
new PoWSrvService());
```

Besides the timeout for calls against the node you can specify an error threshold along with a reset timeout. The client behaves similar to a [circuit breaker](https://martinfowler.com/bliki/CircuitBreaker.html).

If you are using the fallback client, make sure that the calls you run against the nodes are correctly formed, since there is no distinction between exceptions.

CHAPTER 3

MAM

Masked Authenticated Messaging (MAM) is a second layer data communication protocol which adds functionality to emit and access an encrypted data stream. You can read more about it here: https://blog.iota.org/introducing-masked-authenticated-messaging-e55c1822d50e

3.1 Compatibility

The current C# MAM implementation is compatible to https://www.npmjs.com/package/mam.ts. Compatibility with the iota.mam.js has not been tested and may therefore not be compatible.

3.2 Channels

In the context of MAM, channels represent the sender. A channel manages the its seed, tracks its state, creates and signs messages. More about the statefullness of channels can be read below.

Creating a message through a channel and publishing it, can be done with a few lines of code. Channels should not be instantiated directly, rather they are the product of a channel factory.

```
var factory = new MamChannelFactory(CurlMamFactory.Default, CurlMerkleTreeFactory.
↪Default, iotaRepository);
var channel = factory.Create(Mode.Restricted, seed, SecurityLevel.Medium,
↪"yourchannelkey");

var message = channel.CreateMessage(TryteString.FromAsciiString("This is my first␣
↪message with MAM from CSharp!"));
await channel.PublishAsync(message);
```

3.3 Subscriptions

Subscriptions are used to listen to certain channels and retrieve messages from it. Listening do a channel can be done from any point (root) on, but not backwards. For a subscription it is not needed to know the channels seed.

```
var factory = new MamChannelSubscriptionFactory(iotaRepository, CurlMamParser.Default,
↪ CurlMask.Default);

var channelSubscription = factory.Create(new Hash("CHANNELROOT"), Mode.Restricted,
↪"yourchannelkey");
var publishedMessages = await channelSubscription.FetchAsync();
```

3.4 Serialization and State

Given the statefullness of channels and subscriptions, any application should persist the state of them. This is especially true for channels, where each message has its own index. No second message should be published to that index (similar to the address reuse issie).

The state of a channel/subscription can be retrieved by simply calling the .ToJson method. This generates a JSON representation of the channel/subscription. When recreating the channel/subscription, simply use the factories CreateFromJson method.

```
var subscrptionJson = subscription.ToJson();
var factory = new MamChannelSubscriptionFactory(iotaRepository, CurlMamParser.Default,
↪ CurlMask.Default);

var channelSubscription = factory.CreateFromJson(subscrptionJson)
```

```
var channelJson = channel.ToJson();
var factory = new MamChannelFactory(CurlMamFactory.Default, CurlMerkleTreeFactory.
↪Default, iotaRepository);

var channel = factory.CreateFromJson(channelJson)
```

3.5 Code examples

To deepen your unterstanding on how channels, subscriptions and compatibility with the TS version (see above) work, take a look at https://github.com/Felandil/tangle-.net/tree/master/Tangle.Net/Tangle.Net.Examples/Examples/Mam