|  |  |
| --- | --- |
|  | **Szymon Babczyński**  Contact: szybabczynski@gmail.com  Github: https://github.com/rvnlord/ |

**Developing Crypto Bot Lesson 1: Communicating with Private and Public DEX API**

**Prerequisites:**

* General knowledge about how object-oriented programming languages work (examples will be written in C#)
* General knowledge about Visual Studio IDE, although you can write in any IDE of your choice
* General knowledge about cryptocurrency Exchanges
* Basic knowledge about cryptography, we will be using Bouncy Castle library

I would like to welcome everybody interested in learning new and exciting programming skills. The knowledge you will be presented with in this series will allow you to create your own tools for communicating with cryptocurrency exchanges. You will be no longer dependent on third party software that is often costly and difficult to understand. At first glance these articles might feel way harder than the ones you often see on our website but the only requirements to overcome this difficulty are your will and patience.

In general *Web APIs* offered by the exchanges can be divided into two categories: *HTTP APIs* and *WebSocket APIs*. First type is used mostly for sending commands and querying data sources on demand. The other one serves as a way of retrieving real-time data continuously. Today we will be working with *HTTP API* only.

*HTTP API* exposes endpoints that are associated with two different kinds of queries: public – used for retrieving data that is not tied to any specific user account and private – that allows to manage orders and data that is not exposed publicly.

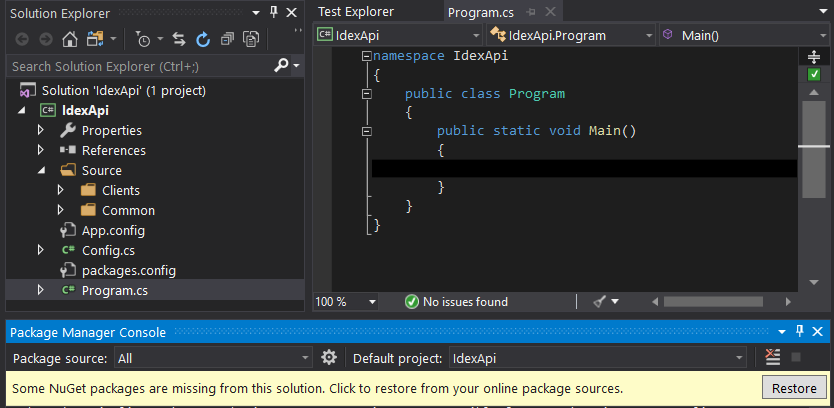
Today we will learn how to create both types of request objects. As an example we will use *API* that is exposed by the *IDEX Decentralized Exchange*. Documentation that can serve as a reference can be found under the following address:

<https://docs.idex.market/#tag/Contract-Backed-Trade-Endpointsx>

Before we start, to speed things up, I encourage everybody to download the template I made for you. It is not mandatory but it contains all the necessary packages and utility functionality that is beyond the scope of this lesson. You can find the link just below:

<https://www.dropbox.com/s/8n73oj5y4198cdt/IdexApiEmpty.zip?dl=0>

When you open the template project you should be presented with something similar to the picture below:



As soon as you open *Package Manager Console* or *Package Manager*, *IDE* will prompt you about the missing packages. You should restore them because they are vital part of the future logic. You can think of them as libraries that provide additional functionality like cryptography or *JSON* parsing.

Let’s first give ourselves a starting point by pretending that the entire functionality already exists in our app. Let’s write a code that we expect to work in the future and put it in our *Program.cs* file that is our console entry point:

public static readonly string \_ethPrivateKey = "<your private key used on IDEX>";

public static readonly string \_ethAddress = "<your address associated with the private key>";

public static async Task Main()

{

var idex = new IdexClient(\_ethAddress, \_ethPrivateKey);

var ticker = await idex.TickerAsync(new AssetPair("AWC", "ETH"));

var order = await idex.PlaceOrderAsync(OrderType.Buy, 1000, 0.00032010m, new AssetPair("AWC", "ETH"));

Console.WriteLine(ticker.ToString());

Console.WriteLine(order.ToString());

Console.ReadKey();

}

Excellent, we have described how server calls will look like. Notice that in our example we will be getting ticker which is part of the public *API* and then we will place an order that is part of the private *API*.

In the template, in *ApiClient.cs* file we already have four methods declared. Each one handles different type of call to the exchange. They essentially simplify the process of writing requests by providing access to all available combinations of *GET/POST Public/Private* queries. This way the only thing we need to override for our Exchange Client is *QueryAsync()* method but we can use any of the mentioned combinations.

Now let’s add a new class that will contain all our future *API* calls. To do this let’s first create a file called *IdexClient.cs* inside *Idex* folder, write the constructor’s logic that will be used when creating an instance of the class and override the method that will query the data:

public class IdexClient : ApiClient

{

public IdexClient(string ethAddress, string privateKey, TimeSpan? rateLimit = null)

{

\_baseAddress = "https://api.idex.market/";

\_rateLimit = rateLimit ?? TimeSpan.FromSeconds(0);

ApiPublicKey = ethAddress;

ApiPrivateKey = privateKey;

}

protected override async Task<T> QueryAsync<T>(QueryType queryType, Method method, string action,

OrderedDictionary<string, string> parameters = null, DeserializeCustom<T> deserializer = null)

{

await RateLimitAsync();

var uri = $"{\_baseAddress}{action}";

var request = queryType == QueryType.Private

? new IdexAuthenticatedRequest(method, ApiPrivateKey, ApiPublicKey, parameters)

: new RestRequest(method);

request.AddHeader("Content-Type", "application/json");

request.AddHeader("Accept", "application/json");

request.AddHeader("User-Agent", "cs-idex");

if (parameters?.Any() == true)

request.AddParameter("application/json", parameters.JsonSerialize(), ParameterType.RequestBody);

var rawResponse = new RestClient(uri).Execute(request);

if (rawResponse.ContentLength == 0)

throw new IdexException("Server returned empty content, probably spam prevention");

var response = deserializer == null

? rawResponse.Content.JsonDeserialize().To<T>()

: deserializer(rawResponse.Content);

return response;

}

}

public enum OrderType { Buy, Sell }

In the constructor we define the address, *API* keys for private calls and a way to limit the number of queries within the given time. While the last one is useful, the official *Idex* documentation doesn’t define this value so we leave it as default with an easy way to change it at any time.

*QueryAsync()* method is waiting the given amount of time, then it defines the full address by concatenating *action* (query name) to the base address. Next it creates a request depending on the fact if it should be public or private. Private request contains additional parameters and we will be writing *IdexAuthenticatedRequest* class for that purpose shortly*, d*on’t worry about it now. Lastly, the method adds the appropriate headers and parameters if query provides any. Notice that parameters in the case of *Idex* Exchange are actually added as a single serialized body parameter. If you have experience working with other exchanges this may be a bit unexpected, but this is the way it works on this particular *DEX*. The last thing to do is catching the response and routing it to the proper deserialization mechanism in form of a delegate.

If it is a little bit confusing at first glance, the example of a public call below should clarify it:

public async Task<TickerResponse> TickerAsync(AssetPair pair)

{

var parameters = new OrderedDictionary<string, string>

{

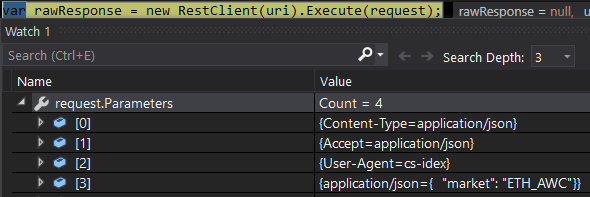
{ "market", $"{pair.QuoteAsset}\_{pair.BaseAsset}" }

};

return await PostPublicAsync("returnTicker", parameters, json => new TickerResponse().Parse(json, pair));

}

We want a specific ticker, so we need to call the *QueryAsync()* method created earlier by calling *PostPublicAsync()* function which calls it internally with parameter being a pair (added as single value to a serialized *JSON* object):



The funny thing on *Idex* is that base and quote assets are inverted so make sure to pay attention when coding.

Alright, so we got a public request with parameters that is being send to the server. As you remember, the received response is routed to the delegate. In this case the delegate points to *Parse()* function of our response object, so now we need to write this function and the class itself:

public class TickerResponse : IdexResponse

{

public AssetAmount LastPrice { get; set; }

public AssetAmount High { get; set; }

public AssetAmount Low { get; set; }

public AssetAmount BaseCurrencyVolume { get; set; }

public AssetAmount QuoteCurrencyVolume { get; set; }

public AssetAmount HighestBuyOrder { get; set; }

public AssetAmount LowestSellOrder { get; set; }

public AssetAmount Average { get; set; }

public decimal PercentChange { get; set; }

public TickerResponse Parse(string json, AssetPair pair)

{

RawParse(json);

var rawTickers = json.JsonDeserialize();

Initialize(CreateTicker(pair, rawTickers));

return this;

}

public static TickerResponse CreateTicker(AssetPair pair, JToken rawTicker)

{

var baseCurrency = pair.BaseAsset;

var quoteCurrency = pair.QuoteAsset;

var high = (rawTicker["high"].ToDecimalN() ?? 0).ToAssetAmount(quoteCurrency);

var low = (rawTicker["low"].ToDecimalN() ?? 0).ToAssetAmount(quoteCurrency);

return new TickerResponse

{

LastPrice = new AssetAmount(rawTicker["last"].ToDecimalN() ?? 0, quoteCurrency),

High = high,

Low = low,

LowestSellOrder = (rawTicker["lowestAsk"].ToDecimalN() ?? 0).ToAssetAmount(quoteCurrency),

HighestBuyOrder = (rawTicker["highestBid"].ToDecimalN() ?? 0).ToAssetAmount(quoteCurrency),

PercentChange = rawTicker["percentChange"].ToDecimalN() ?? 0,

BaseCurrencyVolume = (rawTicker["quoteVolume"].ToDecimalN() ?? 0).ToAssetAmount(baseCurrency),

QuoteCurrencyVolume = (rawTicker["baseVolume"].ToDecimalN() ?? 0).ToAssetAmount(quoteCurrency),

Average = ((low.Amount + high.Amount) / 2).ToAssetAmount(quoteCurrency)

};

}

private void Initialize(TickerResponse ticker)

{

LastPrice = ticker.LastPrice;

High = ticker.High;

Low = ticker.Low;

BaseCurrencyVolume = ticker.BaseCurrencyVolume;

QuoteCurrencyVolume = ticker.QuoteCurrencyVolume;

HighestBuyOrder = ticker.HighestBuyOrder;

LowestSellOrder = ticker.LowestSellOrder;

Average = ticker.Average;

PercentChange = ticker.PercentChange;

}

public override string ToString()

{

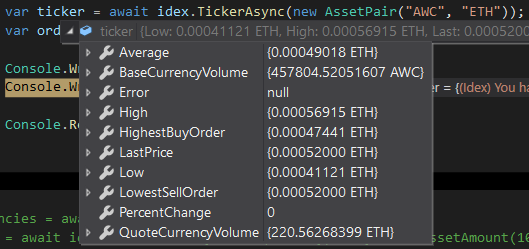
return $"Low: {Low}, High: {High}, Last: {LastPrice}, Base Volume: {BaseCurrencyVolume}, " +

$"Quote Volume: {QuoteCurrencyVolume}";

}

}

*Parse()* method receives *JSON* object returned by *Idex* and requested ticker pair as arguments. The pair is passed here because there is no way of retrieving it out of the response itself, so it serves as a convenience. At the beginning *RawParse()* function is deserializing the response as a general object just to see if there are any unusual problems or errors so they can be thrown. Next, the response is deserialized properly to the appropriate properties and mapped to the current object. *Initialize()* method serves here as sort of a helper because in *C#* you can’t assign a new object to *this* keyword. Also notice that *AssetAmount* class is a utility that allows to store currency value with its symbol. The complete response should look like this:



Private call would be tiny bit more complicated than that because we would need to sign our request, that’s when authenticated request becomes handy, but first things first. Let’s take a look at the query itself:

public async Task<PlaceOrderResponse> PlaceOrderAsync(OrderType orderType, decimal amount, decimal price,

AssetPair pair)

{

var assets = (await CurrenciesAsync()).Currencies;

var baseAsset = assets.Single(a => a.Currency == pair.BaseAsset);

var quoteAsset = assets.Single(a => a.Currency == pair.QuoteAsset);

var amountBuy = orderType == OrderType.Buy ? amount : amount \* price;

var amountSell = orderType == OrderType.Buy ? amount \* price : amount;

var parameters = new OrderedDictionary<string, string>

{

{ "contractAddress", (await ContractAddressAsync()).ContractAddress },

{ "tokenBuy", baseAsset.TokenAddress },

{ "amountBuy", amountBuy.ToWei(baseAsset.DecimalPlaces.ToInt()).ToString() },

{ "tokenSell", quoteAsset.TokenAddress },

{ "amountSell", amountSell.ToWei(quoteAsset.DecimalPlaces.ToInt()).ToString() },

{ "expires", 0.ToString() }

};

return await PostPrivateAsync("order", parameters, json => new PlaceOrderResponse().Parse(json, pair));

}

We can see above that we don’t have a response class and a parser yet. But there is more, to call private *API* we will also need *DEX* contract address as well as token addresses and precision of currencies themselves. At this point public *API* calls should be straightforward enough, so I won’t be diving into the code of them. They are provided within the template. You just need to add the appropriate calls into *IdexClient* class:

public async Task<CurrenciesResponse> CurrenciesAsync()

{

return await PostPublicAsync("returnCurrencies", null, new CurrenciesResponse().Parse);

}

public async Task<ContractAddressResponse> ContractAddressAsync()

{

return await PostPublicAsync("returnContractAddress", null, new ContractAddressResponse().Parse);

}

At this point we still lack the *IdexAuthenticatedRequest* class that will contain parameters that let the Exchange verify the validity of our query:

public class IdexAuthenticatedRequest : AuthenticatedRequest

{

public IdexAuthenticatedRequest(Method method, string privKey, string address,

OrderedDictionary<string, string> parameters) : base(method)

{

var nonce = UnixTimestamp.UtcNow.ToLong();

if (nonce <= \_lastNonce) nonce = ++\_lastNonce;

var strNonce = nonce.ToString();

parameters.Add("nonce", strNonce);

parameters.Add("address", address);

var signature = CreateSignature(privKey, parameters);

parameters.Add("v", signature.V.ToBigIntU().ToString());

parameters.Add("r", signature.R.ToHexString().PadLeft(64, '0').EnforceHexPrefix());

parameters.Add("s", signature.S.ToHexString().PadLeft(64, '0').EnforceHexPrefix());

parameters.Remove("contractAddress");

\_lastNonce = nonce;

}

private static EthECDSASignature CreateSignature(string strPrivKey,

OrderedDictionary<string, string> parameters)

{

var bParamVals = new List<byte>();

foreach (var p in parameters)

{

var pVal = p.Value;

bParamVals.AddRange(pVal.StartsWith("0x")

? pVal.RemoveHexPrefix().ToHexByteArray()

: pVal.ToBigInt().ToByteArray().PadStart(32));

}

var rawHash = Sha3(bParamVals);

var hashOfSaltedHash = Sha3(ConcatMany(

"0x19".ToHexByteArray(),

"Ethereum Signed Message:\n".ToUTF8ByteArray(),

rawHash.Length.ToString().ToUTF8ByteArray(),

rawHash));

var privateKey = strPrivKey.ECPrivateKeyStringToECPrivateKey();

var signature = SignEthECDSA(privateKey, hashOfSaltedHash);

return signature;

}

}

The parameters we need for the private part of the *API* are *nonce* – time based, always incrementing value; *address* – the Ethereum public address and *signature* created using Ethereum private key with other request specific parameters in predefined order. The signature takes form of *v, r, s* values where the first two are a product of applying *ECDSA* algorithm and the last one is derived from the algorithm that recovers Ethereum public key from the signature.

In order to produce the correct signature we need to hash the parameters with *Sha3* (*Keccak*) function, prefix the message: *\x19Ethereum Signed Message* followed by the hash length and then hash the result again with *Sha3* function. The resulting double hash should be signed by using deterministic *ECDSA* algorithm. I also provided *VerifyECDSA()* function in the template if you’d like to verify the signature manually before sending it. The good thing is that *Idex* tend to be very verbose about any errors so if you’ve made a mistake during implementation of the signer it will prompt you about which parameter is incorrect and why.

The correct request should look like this:

{

"tokenBuy": "0xad22f63404f7305e4713ccbd4f296f34770513f4",

"amountBuy": "100000000000",

"tokenSell": "0x0000000000000000000000000000000000000000",

"amountSell": "320100000000000000",

"expires": "0",

"nonce": "1558549716",

"address": "0x856fb4423F9554700Fd04a6e0d6052e72B0364a1",

"v": "28",

"r": "0x35c36ad09b27a7f6069855d774c7c76f5d07b7a45a833829789c3a261d4d35c7",

"s": "0xfc92f16aa4566a82843b45343919b6ae59d1a5e695690126d6628813d3c20c72"

}

The only thing that remains now is the response itself:

public class PlaceOrderResponse : IdexResponse

{

public long OrderNumber { get; set; }

public string OrderHash { get; set; }

public AssetAmount Price { get; set; }

public AssetAmount Amount { get; set; }

public AssetAmount Cost { get; set; }

public OrderType OrderType { get; set; }

public long BlocksToExpire { get; set; }

public string User { get; set; }

public long Nonce { get; set; }

public PlaceOrderResponse Parse(string json, AssetPair pair)

{

RawParse(json);

var rawAddOrderResponse = json.JsonDeserialize();

var rawAddOrderParamsResponse = rawAddOrderResponse["params"];

OrderNumber = rawAddOrderResponse["orderNumber"].ToLong();

OrderHash = rawAddOrderResponse["orderHash"].ToString();

Price = rawAddOrderResponse["price"].ToAssetAmount(pair.QuoteAsset);

Amount = rawAddOrderResponse["amount"].ToAssetAmount(pair.BaseAsset);

Cost = rawAddOrderResponse["total"].ToAssetAmount(pair.QuoteAsset);

OrderType = rawAddOrderResponse["type"].ToEnum<OrderType>();

BlocksToExpire = rawAddOrderParamsResponse["expires"].ToLong();

User = rawAddOrderParamsResponse["user"].ToString();

Nonce = rawAddOrderParamsResponse["nonce"].ToLong();

return this;

}

public override string ToString()

{

return Error?.Message ?? $"{OrderType.EnumToString()} {Amount} @ {Price} for " +

$"{(Amount \* Price).ToAssetAmount(Price.Asset)}";

}

}

The response will be either the placed order or a meaningful error:

{ orderNumber: 2101,

orderHash: '0x3fe808be7b5df3747e5534056e9ff45ead5b1fcace430d7b4092e5fcd7161e21',

price: '0.000129032258064516',

amount: '3100',

total: '0.4',

type: 'buy',

params:

{ tokenBuy: '0x7c5a0ce9267ed19b22f8cae653f198e3e8daf098',

buyPrecision: 18,

amountBuy: '3100000000000000000000',

tokenSell: '0x0000000000000000000000000000000000000000',

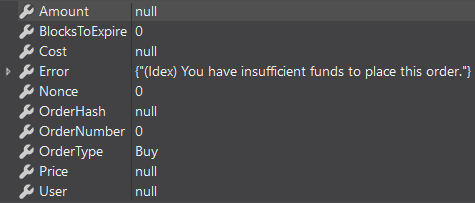
sellPrecision: 18,

amountSell: '400000000000000000',

expires: 100000,

nonce: 1,

user: '0x57b080554ebafc8b17f4a6fd090c18fc8c9188a0' } }



That’s it! Now you know how to correctly and effortlessly connect to the exchange *API* and how to make any type of available request.

