# Software Developer (C#)

For this assignment, there is an ideal scenario to achieve, however you can further finetune it with more cases. Last but not least, there is an additional step that you may optionally complete. **You should make a public repository on github for this assignment and provide us with the link.**

For this assignment, the focus is on EIP-721 NFTs on Ethereum blockchain. You are asked to implement an application that will extract metadata for an NFT from blockchain. NFTs can be identified via their contract in which they are defined as well as their index (i.e. contract specific id) set in their contract.

## Input

As input, you will have an object in the following format (NFT and Token will be used interchangeably here)

```
{
  "tokenId":1823502,
  "TokenIndeX":"1234",
  "ContractAddress":"0xbc4ca0eda7647a8ab7c2061c2e118a18a936f13d"
}
```

## Token URI + Smart Contract Interaction

According to EIP-721, in order to access a token's meta, you need to invoke '**tokenURI**' method of a smart contract. In order to invoke a contract, you can use infura.io to create a free account. From here, you need to get the mainnet endpoint in the form of: (API key is supposed to be secret and known to you only)

https://mainnet.infura.io/v3/YOUR_API_KEY_VALUE_IN_HEX

In order to invoke the method, you can use Nethereum. To setup a client using infura provider, check Web3 page.

Aside from client, you need to know the contract's interface so you can implement the interaction layer. Smart Contracts come with Application Binary Interface (ABI) that contains list of events and functions in a contract as well as their parameter details in json format. You can access a smart contract in various ways. One way is to go to the contract page on etherscan using this url:

https://etherscan.io/address/CONTRACT_ADDRESS_GOES_HERE#code

- Example: https://etherscan.io/address/0xbc4ca0eda7647a8ab7c2061c2e118a18a936f13d#code

On contract's page on etherscan.io, you can simply go to contract's tab and then find ABI under the Code tab. Having the ABI, you can now create the necessary objects, based on Smart Contracts Interaction, so tokenURI can be invoked. To do so, you can either write manually or use Nethereum code generation.

To verify you have set everything right, if you get the tokenURI for token 1234 of contract 0xbc4ca0eda7647a8ab7c2061c2e118a18a936f13d, the value (string) should be:

ipfs://QmeSjSinHpPnmXmspMjwiXyN6zS4E9zccariGR3jxcaWtq/1234

which can be parsed using https://ipfs.io/ipfs/{KEY} so:

https://ipfs.io/ipfs/QmeSjSinHpPnmXmspMjwiXyN6zS4E9zccariGR3jxcaWtq/1234

and the content will look like


```
{
  "image": "ipfs://QmZ2ddtVUV1brVGjpq6vgrG6jEgEK3CqH19VURKzdwCSRf",
  "attributes": [
    {
      "trait_type": "Eyes",
      "value": "Sleepy"
    },
    {
      "trait_type": "Background",
      "value": "Army Green"
    },
    {
      "trait_type": "Clothes",
      "value": "Leather Jacket"
    },
    {
      "trait_type": "Fur",
      "value": "Blue"
    },
    {
      "trait_type": "Mouth",
      "value": "Bored Bubblegum"
    },
    {
      "trait_type": "Hat",
      "value": "Fisherman's Hat"
    }
  ]
}
```


Note: tokenURI is expected to return a string with no error, but depending on circumstances it can fail. So, it’s a good idea to have something in place to give us enough detail in when and where it happens.

## Processing Token URI + Output

URIs can be anything ranging from web2 urls (e.g. example.com/example.json) to base64 encoded meta. Your application should have a method to parse the URI and in case it's not supported, it should note it so it will be supported later. The application must get a URI's content first and then try to parse it to generate the unified model for any token.

Even though most contracts try to follow the same way, there could still be differences in the way the content of the URI is formatted. Still, we need to have a code that would be able to handle most of these. For this exercise, you can use either a generic approach or using a json that is built around a contract, however please justify why choose the approach you will follow. Either way, the output of your code should look like the following object (but as a C# model): (Some properties like name may not exist in some projects, so they will be null)

```
{
  "Name": "Happy Ape #234",
  "Description": "An example description for a token",
  "ExternalUrl": "Example.com/tokens/1234",
  "Media": "example.com/image.jpg",
  "Properties": [
    { "Category": "Eyes", "Property": "Sleepy" },
    { "Category": "Background", "Property": "Army Green" },
    { "Category": "Clothes", "Property": "Leather Jacket" },
    { "Category": "Fur", "Property": "Blue" },
    { "Category": "Mouth", "Property": "Bored Bubblegum" },
    { "Category": "Hat", "Property": "Fisherman's Hat"}
  ]
}
```

The following contract addresses (and corresponding token Indexes) should be supported.

| ContractAddress | TokenIndex |
|---|---|
| 0x1a92f7381b9f03921564a437210bb9396471050c | 0 |
| 0xec9c519d49856fd2f8133a0741b4dbe002ce211b | 30 |
| 0xeaa4c58427c184413b04db47889b28b5c98ebb7b | 1 |
| 0x0b22fe0a2995c5389ac093400e52471dca8bb48a | 0 |
| 0xb4d06d46a8285f4ec79fd294f78a881799d8ced9 | 9896 |
| 0xb668beb1fa440f6cf2da0399f8c28cab993bdd65 | 285 |
| 0xbc4ca0eda7647a8ab7c2061c2e118a18a936f13d | 1234 |

**Optionally** you can check and (if needed) adjust your code to support these as well.

| ContractAddress | TokenIndex |
|---|---|
| 0xdbb163b22e839a26d2a5011841cb4430019020f9 | 287 |
| 0x1cb1a5e65610aeff2551a50f76a87a7d3fb649c6 | 30000000 |
| 0x845dd2a7ee2a92a0518ab2135365ed63fdba0c88 | 18 |
| 0x7bd29408f11d2bfc23c34f18275bbf23bb716bc7 | 4563 |
| 0x059edd72cd353df5106d2b9cc5ab83a52287ac3a | 2000000 |
| 0x1b829b926a14634d36625e60165c0770c09d02b2 | 1000001 |
| 0xd4e4078ca3495de5b1d4db434bebc5a986197782 | 1 |
| 0x892848074ddea461a15f337250da3ce55580ca85 | 0 |