M2 Blockchain Project Report: Implementation of a decentralized network with Oracles

CHRISTIAN ADJA

EFREI PARIS | 10/03/22

Ruben NABET, Gabriel PRECIGOUT, Daniil ROSSO

Table of contents

I. What are blockchain oracles?

Nowadays, with the worldwide democratization of blockchain technologies, developers keep building smart contracts that are getting more and more complex as time goes on. These smart contracts rely on data that is accessible to them, in order to perform tasks, transactions or other actions automatically. However, as powerful as they can be, these smart contracts are by definition not able to interact with any data that is not already on the same blockchain.

This is where oracles come in. Oracles act as sort of a "middleware" between real world data, and the smart contracts that need this data to function. Their main function is to aggregate, process, compile and push data into a blockchain, usually through a web2 server. By providing other smart contracts with real-world data, they can sometimes enable the creation of entire blockchain ecosystems, that rely on this data.

Oracles can exist in two types: hardware and software. Hardware oracles rely on various real-world data providers such as sensors (temperature, pressure, etc) or even RFID chips. The data is simply measured and transmitted to the blockchain. However, this is not the type of oracle we decided to work with in this project. The other, more common type is the software oracle, which pulls its data from the Internet. Using mainly APIs (web servers) as their source, they are able to gather a wide variety of data from the web. This is the type of oracle we decided to build, using the Marvel Comics character API.

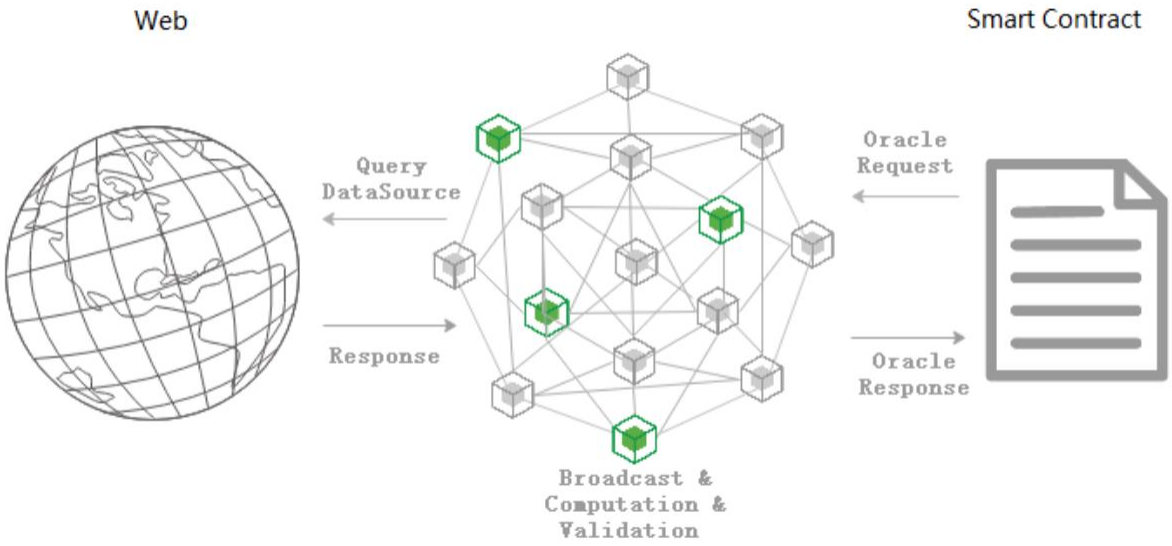


Figure 1 : A representation of the functioning of an oracle

II. Our implementation: a Marvel character data compilator

Our goal with this project is to implement a decentralized data provisioning system using an oracle, which serves a middleman between a regular web2 REST API and the blockchain. We decided to build it in two parts:

- A regular web2 server, which is going to be responsible for fetching, aggregating, processing, and finally pushing data to the blockchain.
- A smart contract, written in Solidity, which is going to enable other smart contracts to interact with the provided data. This also includes a Solidity interface, meant for developers that would want to build a smart contract that implements our oracle.

Our deliverables include the source code as a zipped archive, with a README containing explanations and instructions, and this report. The source code can also be found in this GitHub repository:

https://github.com/cvbenur/blockchain-oracle-project

a. The web2 server

The web2 server we used for this project is a very basic Node.js application written in TypeScript. It uses Web3.js to communicate dynamically with our smart contract. We designed the application to perform one fetch-and-push cycle every 24 hours. Basically, the cycle works like this:

1. The application calls the contract to retrieve the number of characters already present on the blockchain.
2. Then, the application fetches the next 100 characters not already present from the Marvel Comics API.
3. After that, the application formats the retrieved character data in order to standardize it for future use.
4. Once the standardization is complete, the application calls the smart contract to push the processed data into it.

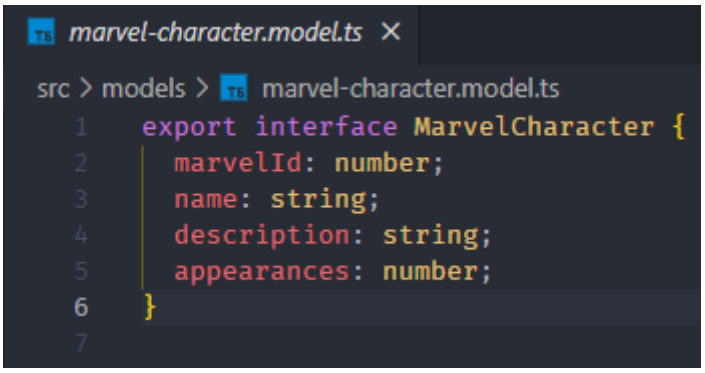


Figure 2: Our standardized data model for the web2 server

We decided to work with an easy to manipulate data source: the Marvel Comics public API (https://developer.marvel.com/). It offers a clear and easy-to-use documentation, with good readability. We chose this particular data source because we decided that it would be perfect to use as a proof of concept for an oracle that would push this information into a blockchain. We crafted all of our API requests using the following parameters (see documentation for a full list):

Parameter	Value	Description	Parameter Type	Data Type
name		Return only characters matching the specified full character name (e.g. Spider-Man).	query	string
nameStartsWith		Return characters with names that begin with the specified string (e.g. Sp).	query	string
modifiedSince		Return only characters which have been modified since the specified date.	query	Date
comics		Return only characters which appear in the specified comics (accepts a comma-separated list of ids).	query	int

Most of the data handling and smart contract interactions in this part of the project are done asynchronously thanks to Node.js' built-in Promises feature.

b. The smart contracts & interfaces

For this part, we decided to work with the Ethereum blockchain and Solidity, because of the freedom it provides and of the easy-to-use developer tools. We

used the Truffle suite for the development, and we also used Ganache and its built-in local Ethereum test network for the deployment and tests, which worked very well.

Using Solidity, we were able to write the main smart contract behind our oracle (MarvelCharactersOracle) as well as all the moving parts necessary to handle our data on the blockchain. The contract contains methods designed to provide access to the characters data stored inside of it, as well as several onlyOwner methods meant either for testing or to add data to the oracle. Each individual character is defined inside the contract by a data structure which contains all its relevant information.

We also included a Solidity interface (MarvelCharactersOracleInterface), which covers all the necessary methods of our oracle. This interface is designed for developers. It allows them to easily write smart contracts that implement our oracle's data retrieval methods.

```
struct Character {
    uint256 id;
    uint256 marvelId;
    string name;
    string description;
    uint256 appearances;
}
```

Figure 3: The character data structure inside our oracle

c. Testing the smart contract

Nowadays, creating a project without testing it before using it is unthinkable and may result in serious problems, especially if a feature or method added does not work as intended and is pushed to production. It would be a disaster. Knowing that testing our smart contract is a crucial part of development, we had to implement it to make sure that every method called is doing what it's supposed to do. This allows us to prevent most bugs, and to have a better understanding of our code, as we should.

From a technical point of view, we wrote the tests in JavaScript using Truffle's built-in test feature, and Ganache's local Ethereum network. Our tests use the async/await design pattern, to make sure that each contract call or transaction is fully executed before moving on to the next one.

First, we setup our test lifecycle by deploying the Oracle before each test, allowing us to connect to the blockchain and access our smart contract and its data.

Then, we wrote tests for each of our contract methods. In order to test the **testConnection() method**, we wrote a test to make sure that the connection to our smart contract was working properly.

In order to test the **getAddress() method**, we wrote a test to check whether we're connected to the right instance of the smart contract. We can know that by comparing the smart contract's address and the value returned by the method.

In order to test the **characterExists(uint256 _marvelId) method**, we wrote a test to check whether a character exists in the smart contract, from its Marvel ID (that we retrieved from the Marvel API). Here, we can do that by using the method with some test data.

In order to test the **getAllCharacters() method**, we wrote a test to check that the contract call returns an array containing all the Marvel characters stored in our smart contract. We can do that by using the method and comparing the output to our test values.

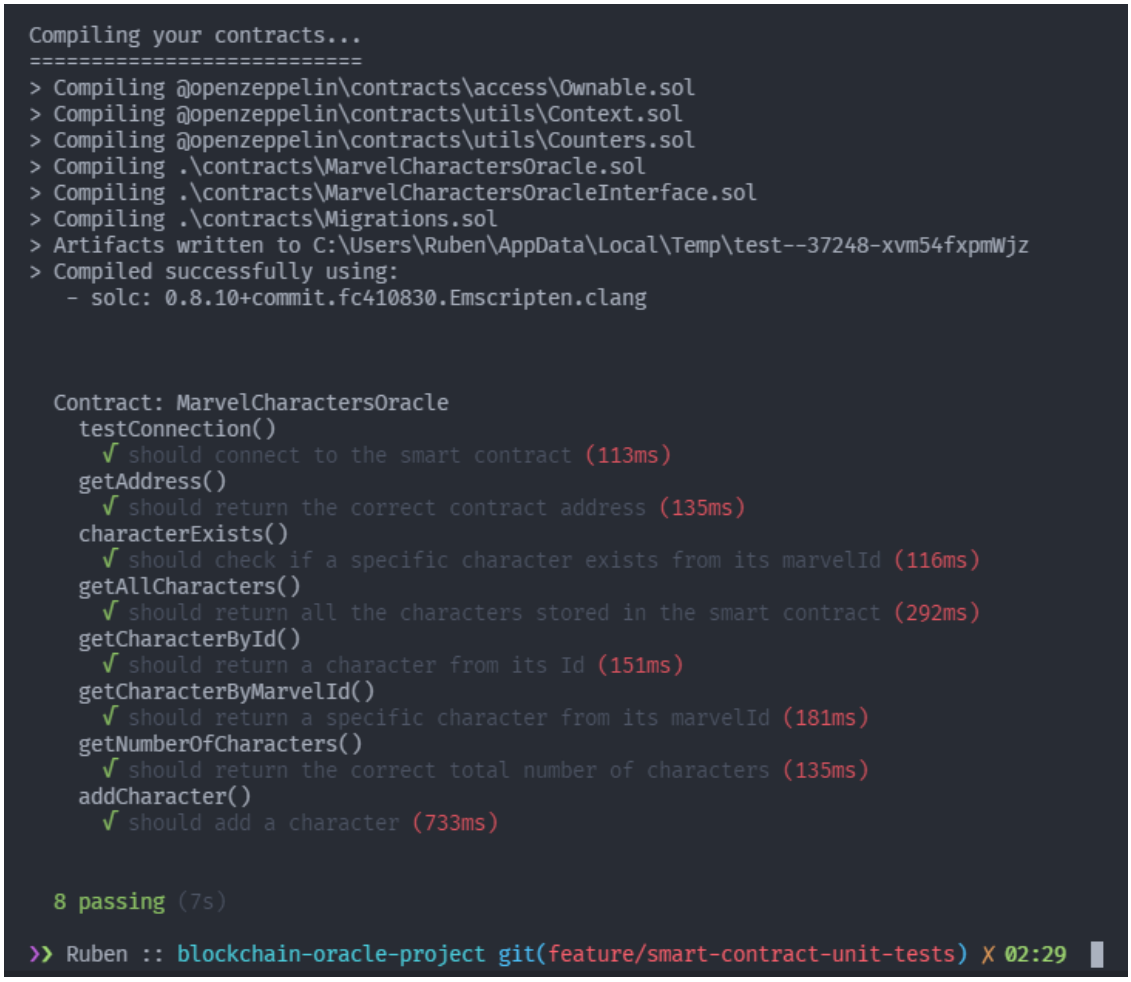
In order to test the **getCharacterById(uint256 _charId) method**, we wrote a test to check that the contract retrieves us the correct character data from its ID. We can do that by using the method and comparing the results with our expected values.

In order to test the **getCharacterByMarvelId(uint256 _marvelId) method**, we setup a similar test as to the previous method, but this time using the character's Marvel ID.

In order to test the **getNumberOfCharacters() method**, we wrote a test to check whether the method returns us the correct test character data.

Finally, in order to test the **addCharacter(uint256 _marvelId, string memory _name, string memory _description, uint256 _appearances) method**, we wrote a test to check whether the methods correctly adds a new character to the smart contract's storage.

Properly testing our application was essential to ensure that our Oracle was setup properly and interacting correctly with our smart contract. These tests allowed us to prove that out smart contract exhibits the expected behaviour when its methods are called and works as intended. We have provided a screenshot of all of our tests passing:



III. Final thoughts

In conclusion, a lot of what we learned about blockchain technologies was applied during this project. And even if we encountered a fair share of difficulties with the tools that were used (like Ganache randomly deciding to not work for instance), a lot more was understood about how smart contracts work and their relationship with real-world data particularly via oracles.

One question arises though: if oracles rely on a “trusted authority” principle to supply data to the blockchain, does it not go against the core principles of web3 (where decentralization is key)? There are numerous much more complex solutions to explore than our own simple implementation of the concept, and this problem should remain a focus point for further development in the technology.