Gardener Documentation

Release 0.1.0

Espeo Blockchain

Aug 14, 2019

Contents

This open source project solves the problem of getting knowledge from outside of the blockchain into smart contracts.

CHAPTER 1

Oracle theory

Oracle is a concept of getting information from outside of the blockchain to the smart contracts. Out of the box smart contracts cannot access anything outside of the blockchain network. That's where the oracle idea fits. The information exchange begins with the smart contract emitting an event describing the necessary information. A trusted off-chain server listening for such events parses it, gets data from a data source and passes it back to the smart contract.

CHAPTER 2

Contents

2.1 Introduction

2.1.1 What is blockchain oracle?

Oracle is a concept of getting information from outside of the blockchain to the smart contracts. Out of the box smart contracts cannot access anything outside of the blockchain network. That's where the oracle idea fits. The information exchange begins with the smart contract emitting an event describing the necessary information. A trusted off-chain server listening for such events parses it, gets data from a data source and passes it back to the smart contract. The name of our blockchain oracle project is Gardener because of reasons explained here.

2.1.2 Why do we need Gardener?

Blockchains function in a closed, trustless environment and can't get any information from outside the blockchain due to security reasons or so-called sandboxing. You can treat everything within the network as a single source of truth, secured by the consensus protocol. Following the consensus, all nodes in the network agree to accept only one version of their managed state of the world. Think of it like blinders on a horse — useful, but not much perspective.

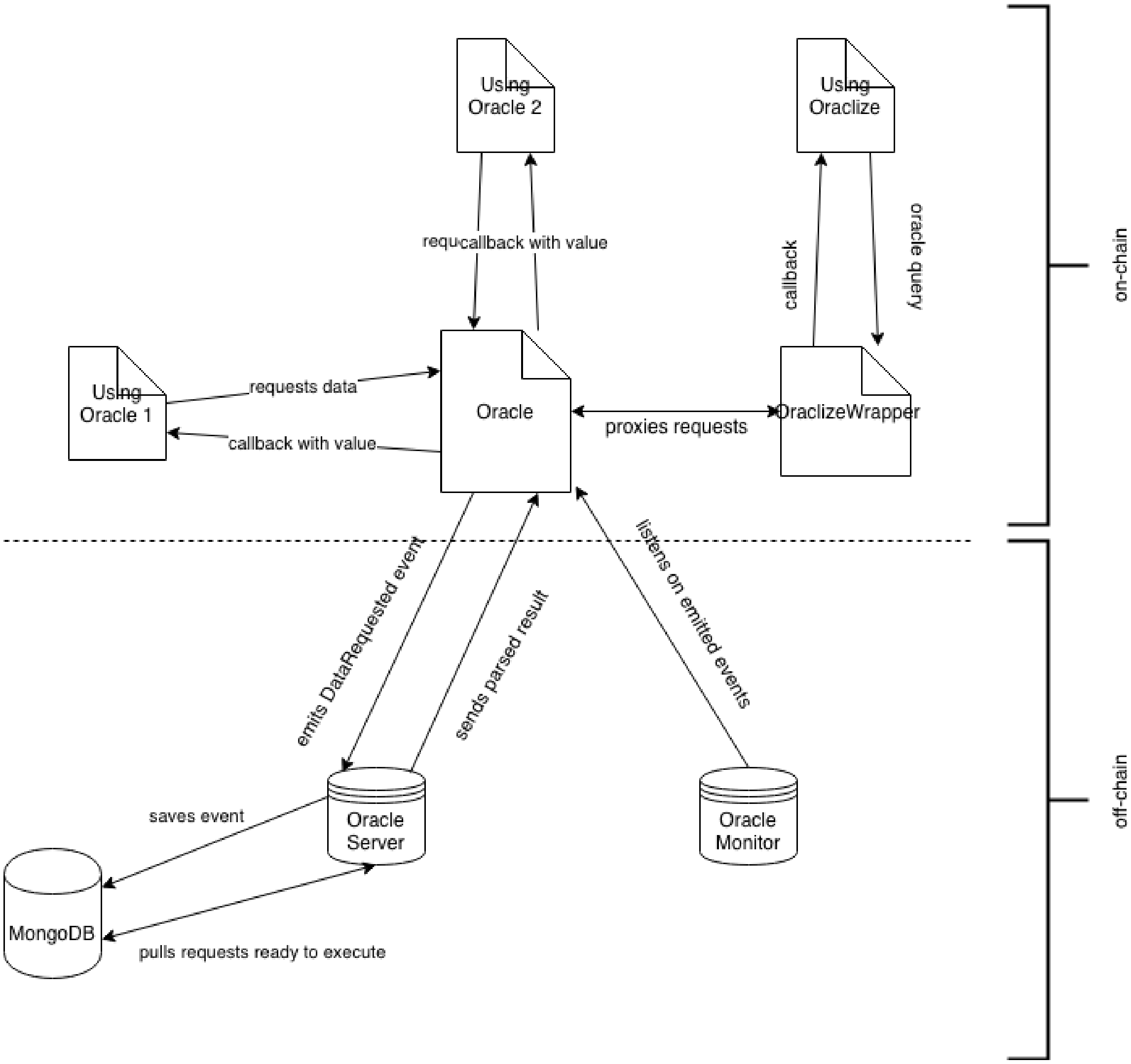
However, sometimes the information available in the network isn't enough. Let's say I need to know what the price of gold is on a blockchain-based derivatives trading app. Using only data from inside the blockchain we have no way of knowing that. Because the smart contract lives in the sandboxed environment it has no option to retrieve that data by itself, the only viable alternative is to request that data and wait for some external party we trust to send it back. That's where the utility of blockchain oracles come in.

2.1.3 How does Gardener work?

This section describes request workflow starting with asking contract with specific request and finishing up to returning result into it. During reading it could be helpful to take a look at the architecture diagram in the *Gardener architecture* section.

According to the diagram, let's start from UsingOracle 1 contract. It interacts with the Oracle contract by passing request parameter to fetch the data it needs. The Oracle contract emits then a DataRequested (or DelayedDataRequested) event. Gardener server is listening to both of these event types and stores every incoming event into its persistence layer (in memory or MongoDB currently). When requests are ready to execute, they are pulled by the server, the data is fetched from external data sources, parsed accordingly and result data in a form of value and error code (0 when the request is fulfilled successfully) is passed to the Oracle contract using configured account. In the current model all costs related to transaction fees are covered by this account (in the future we plan to introduce other pricing models). Oracle contract then proxies this data to the Using Oracle 1 contract fulfilling the whole workflow.

2.1.4 Gardener architecture



2.2 Prerequisites

Before you start using Gardener, you may wish to check if that you have all requirements installed on your platform.

2.2.1 Docker

Docker toolbox can be downloaded here.

You can verify if you have successfully installed docker by running the following command.

> docker —version

2.2.2 Node

[Optional] If you want to run Gardener server without docker container then Node.js is needed. Node is available and you can download it here.

Make sure you have successfully installed it by typing in command line

> node —version

Note: Keep in mind that your Node.js version has to be at least 7.6 as gardener uses async/await pattern.

Note: Please make sure to have Python 2.7.16 (not newer!) installed. Reason for that is node-ffi library that we use is not compatible with newest Python.

2.3 Getting started

2.3.1 Repositories

Gardener consists of three repositories, two main ones: gardener-smart-contracts which holds smart contracts, and gardener-server, which is responsible for fetching data from third party data sources. The third one, gardener-monitor is optional and it helps visualizing requests.

1. Gardener server

```
git clone https://github.com/EspeoBlockchain/gardener-server.git
```

2. Gardener smart contracts

```
git clone https://github.com/EspeoBlockchain/gardener-smart-contracts.git
```

3. Gardener monitor (optional)

```
git clone https://github.com/EspeoBlockchain/gardener-monitor.git
```

2.3.2 Running the test blockchain - Ganache

Before we will get information from external sources to our blockchain, we have to run it first. First of all, copy server variables from template in *gardener-server* directory:

```
make copy-env
```

After that, we can run our blockchain. Let’s use Ganache for it:

```
make ganache
```

If you see this information:

```
Creating test-blockchain ... done
```

That means you have created test blockchain successfully. You can verify its status using:

```
docker ps
```

2.3.3 Running the Monitor (optional)

```
cp .env.tpl .env
npm install
npm start
```

2.3.4 Deploying the Smart contracts

After starting blockchain, we need to copy our smart contracts variables from template. Make sure that you are in gardener-smart-contracts directory, then:

```
make copy-env
```

Now, we are going to install dependencies that Gardener smart contract relies on.

```
npm install
```

After installing dependencies, we are going to migrate our contracts to test blockchain network

```
npx truffle migrate --network ganache --reset
```

2.3.5 Making example oracle request

After we have successfully configured environment, we can make example oracle request. Go to *gardener-server* directory, then:

```
make local
```

Change your directory to *gardener-smart-contract*, then:

```
npx truffle console --network ganache
```

At this moment we are in Truffle Framework console, which can be used for communicating with blockchain network. Let's make a sample request. You can find more information about request specification *Making requests* section.

Example

```
truffle(ganache)> UsingOracle.deployed().then(instance => instance.request(
↪"json(https://api.coindesk.com/v1/bpi/currentprice.json).chartName"))
```

If you did everything correctly you should see something similar to

```
 { tx: '0x57a34e45e1f187ddeb4cbd1be3597561855563e5735a483a5b1edeb73a511278',
   receipt:
    { transactionHash:
↪'0x57a34e45e1f187ddeb4cbd1be3597561855563e5735a483a5b1edeb73a511278',
      transactionIndex: 0,
      blockHash: '0x212e264c92bef193e4531cc151d5b3b36818bc4bf82e154e84af6a7c6a153b43',
      blockNumber: 18,
      from: '0x90f8bf6a479f320ead074411a4b0e7944ea8c9c1',
      to: '0x9561c133dd8580860b6b7e504bc5aa500f0f06a7',
      gasUsed: 97604,
      cumulativeGasUsed: 97604,
      contractAddress: null,
      logs: [ [Object], [Object] ],
      status: '0x1',
      logsBloom:
↪'0x0004000000000000000010000000000000000000000000000000000000000000000000000000000020000000000000100
↪',
      v: '0x1b',
      r: '0x21052fa282f9723d221ef288cec6e947cb2ba0ef3f1d470f5dc8845806f66977',
      s: '0x1723cb7b4288f6ae32f3495d666522859150fe3d0c8e4debd3a80d452f940f50'
      },
   logs:
    [ { logIndex: 1,
        transactionIndex: 0,
        transactionHash:
↪'0x57a34e45e1f187ddeb4cbd1be3597561855563e5735a483a5b1edeb73a511278',
        blockHash: '0x212e264c92bef193e4531cc151d5b3b36818bc4bf82e154e84af6a7c6a153b43
↪',
        blockNumber: 18,
        address: '0x9561c133dd8580860b6b7e504bc5aa500f0f06a7',
        type: 'mined',
        event: 'DataRequestedFromOracle',
        args: [Object]
    } ]
}
```

2.3.6 Server logs

Look up the server container logs to check if response was sent. Moreover, you can check the request and response in the monitor if you've installed and ran it.

2.3.7 Read more

https://truffleframework.com/ganache - Information about Ganache

2.4 Making requests

2.4.1 Request types

You have two types of requests at your disposal. You can use either an instant request, which should fulfill as fast as possible or schedule a delayed request, performed at a specific point of time in the future.

Instant requests

To perform instant request you need to call `request(string _url)` method in `Oracle` contract passing as its only parameter an url wrapped in a format type you want to get back. More about format types in *Response formats* section.

Delayed requests

To perform request, which will be executed in the future you need to call `delayedRequest(string url, uint delay)` method in `Oracle` contract passing as it's first parameter wrapped url and amount of time you need to wait before execution.

The delay parameter can be given in two ways: as a unix timestamp or as a relative number of seconds. Using both options you can delay a request for a maximum of 2 years from now.

2.4.2 Request sources

Currently we support following request sources:

- open REST api
- random data

In the future we plan to implement:

- IPFS
- closed APIs

2.4.3 Response formats

For REST api responses, we support JSON, XML and HTML formats. In the future we plan to support also raw binary data.

JSON format

To parse and select response in a JSON format use the json() wrapper. You can also query response following JsonPath.

Note: `json(...)` wrapper is treated as `$` in JsonPath. Omit it when constructing request. In order to fetch `sth` parameter from response make following request `json(...).sth` (instead of `json(...)$.sth`).

Example request

```
json(https://api.coindesk.com/v1/bpi/currentprice.json).chartName
```

Example response

```
value: "Bitcoin"
error: 0
```

Example request with error

```
json(https://api.coindesk.com/v1/bpi/currentprice.json).sth
```

Example response with error

```
value: ""
error: 4004
```

XML format

To parse and select response in a XML format use the xml() wrapper. You can also query response following XPath.

Note: To make selected response a valid well-formed XML if the result is an array of nodes they are wrapped in a `<resultlist>` tag. Moreover if any of these results is a raw value it's also wrapped in a `<result>` tag.

Example request

```
xml(http://samples.openweathermap.org/data/2.5/weather?q=London&mode=xml&
↪appid=b6907d289e10d714a6e88b30761fae22)string(/current/temperature/@value)
```

Example response

```
value: "280.15"
error: 0
```

HTML format

To parse and select response from HTML site use the html() wrapper. You can also query response following XPath.

Example request

```
html(https://www.w3schools.com)/html/head/title/text()
```

Example response

```
value: "W3Schools Online Web Tutorials"
error: 0
```

ENCRYPTED url fragments

For all URL datasources (XML, HTML, JSON) it is also possible to encrypt entirety, part or parts of your query using encrypted() tag. You might want to do that if you wouldn't like some parts of your URL to be visible to everyone on blockchain. An obvious example would be using some API key as a parameter to your REST query. In order to pass part of your query secretly, simply encrypt it using Gardener public key and wrap it in encrypted() tag. Gardener will decrypt it using its private key and then process it as usual. Any assymetric encryption implementation may be used as long as it produces a stringified version of a following object: {iv, ephemPublicKey, ciphertext, mac} . We recommend using https://www.npmjs.com/package/eth-crypto as shown below.

Example request encryption

import EthCrypto from 'eth-crypto';

const gardenerPublicKey = '9c691b945b14656b98edbf4d3657290c65cad377bca44da4d54e88cd2bbdefb2e063267b06183029fea501756
// if you want to create it programmatically, derive it from Gardener address or Gardener private key if you are owner of the instance const cipher = await EthCrypto.encryptWithPublicKey(gardenerPublicKey, 'SECRET_DATA'); constEncryptedSecret = EthCrypto.cipher.stringify(cipher);

Example request

```
json(https://api.coindesk.com/v1/bpi/historical/close.json?
↪currency=encrypted(c317e44653b8cc3e3ca7f6d9686711c60269a5fd41490868ad8b9cc054836af9d074670241860036
↪disclaimer
json(https://api.coindesk.com/v1/bpi/historical/close.json?
↪currency=encrypted(c317e44653b8cc3e3ca7f6d9686711c60269a5fd41490868ad8b9cc054836af9d074670241860036
↪someOtherParam=encrypted(someOtherEncryptedValue)).disclaimer
```

Example response

```
value: "This data was produced from the CoinDesk Bitcoin Price Index. BPI value data␣
↪returned as EUR."
error: 0
```

RANDOM datasource

Random numbers can be generated using either random.org service or a dedicated SGX application. This is configurable by setting SGX_ENABLED in your .env file to either true or false. There are many benefits of generating random numbers using SGX. We haven't fully leveraged this exciting technology, but after we do, every number will be securely and verifiably generated with the ONLY Third Trusted Party being Intel. That's right, you don't even have to trust whoever hosts a Gardener instance! This is further explained in our article: https://medium.com/gardeneroracle/random-number-generation-in-gardener-1660e5c25e00 . You are also read up about Intel SGX technical details, this is a good starting point: https://software.intel.com/en-us/sgx Using SGX requires a specific hardware and OS (is your environment compatible? check it here https://github.com/ayeks/SGX-hardware) as well as SGX PSW installed. If you don't feel like doing that, you are welcome to use a random.org source which can be considered a less secure but easy to use fallback option.

Note: In order to switch between SGX and random.org way of generating random numbers, use SGX_ENABLED in your .env file.

To generate a random value use the random() wrapper with inclusive upper and lower bounds specified. Both of these bounds should be integers. For random.org acceptable bounds are [-1000000000,1000000000] while for SGX they are defined by 8-byte long datatype: [-9223372036854775808, 9223372036854775807]

Example requests

```
random(10,20)
random(-2,33)
random(-124354325432,-34325253)
```

Example response

```
value: 13
error: 0

value: -2
error: 0

value: -9532532335
error: 0
```

2.4.4 Response error codes

When your requests can be fulfilled succesfully you would get value with error code equals to 0. Any non zero error code means that the request failed to process. Any three-digit code is standard HTTP status code, proxied from the HTTP client. Four-digit errors come from the Gardener server and are listed in the table below.

Error name	Error code	Description
INVALID_URL	1000	Text between type(. . .) wrapper isn't valid url
INVALID_CONTENT_TYPE	1001	This response format wrapper isn't supported
INVALID_ENCRYPTION	1002	Invalid encrypted data. This probably means your data was not encrypted using Gardener public key.
INVALID_SELECTOR	4000	The selector isn't valid JsonPath or XmlPath
NO_MATCHING_ELEMENTS_FOUND	4004	No elements found for given selector
INTERNAL_SERVER_ERROR	5000	Unhandled error happens inside Gardener Server

2.5 Configuration

Each repository of Gardener project (smart contracts, server, monitor) contains an .env variables file. This section is going to explain parameters from them. Default parameters are set in a way everything should work correctly when using local test blockchain (ganache) and following the *Getting started* guide.

2.5.1 Server

- *ADDRESS* - address of the server's account, from which it sends the results of request to the smart contracts
- *PRIVATE_KEY* - private key of the server's account
- *ORACLE_ADDRESS* - address of the *Oracle* smart contract
- *DATABASE_URL* - URL with port for MongoDB connection
- *DATABASE_NAME* - MongoDB database name
- *NODE_URL* - URL for the provider to blockchain network
- *API_PORT* - port for exposing the server's REST API, currently the only endpoint is heartbeat used by Gardener's monitor
- *SAFE_DELAY_BLOCKS* - set a delay in a number of blocks to resist chain reorganization problem, server loads target block when it's number is at least *SAFE_DELAY_BLOCKS* lower than the youngest one
- *START_BLOCK* - set starting block number from which server listen for oracle requests
- *PERSISTENCE* - selected persistence type, currently supported: *INMEMORY* or *MONGODB*

2.5.2 Smart contracts

- *PRIVATE_KEY* - private key used for contract deployment, if both *PRIVATE_KEY* and *MNEMONIC* are passed, *PRIVATE_KEY* is used
- *MNEMONIC* - 12 secret random words for accessing HD wallet (deployment)

- *PROVIDER_URL* - provider for deploying contracts into test network
- *ORACLE_SERVER_ADDRESS* - address of oracle server account

2.5.3 Monitor

- *REACT_APP_PROVIDER_URL* - URL for the provider to blockchain network
- *REACT_APP_STATUS_URL* - hearbeat endpoint, responsible for checking if server is working
- *REACT_APP_ORACLE_ADDRESS* - address of the *Oracle* smart contract