UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Tècnica Superior d'Enginyeria
de Telecomunicació de Barcelona

telecos BCN

Creation of a middleware for blockchain interaction based on NestJS

Master Thesis
submitted to the Faculty of the
Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona
Universitat Politècnica de Catalunya
by

Marc Cosgaya Capel

In partial fulfilment
of the requirements for the master in
Cybersecurity

Advisors: Jose Luis Muñoz Tapia
Rafael Genés Durán

Barcelona, July 2023

telecos BCN

Contents

List of Figures

List of Tables

Revision history and approval record

Revision	Date	Purpose
0	21/05/2023	Document creation
1	14/06/2023	Document revision
2	26/06/2023	Document revision
3	28/06/2023	Document revision
4	01/07/2023	Document delivery

DOCUMENT DISTRIBUTION LIST

Name	e-mail
Marc Cosgaya Capel	marc_cosgaya@protonmail.com
Jose Luis Muñoz Tapia	jose.luis.munoz@upc.edu
Rafael Genés Durán	rafael.genes@upc.edu

Written by:		Reviewed and approved by:	
Date	28/06/2023	Date	01/07/2023
Name	Marc Cosgaya Capel	Name	Jose Luis Muñoz Tapia
Position	Project Author	Position	Project Supervisor

Abstract

Accessing and operating with a blockchain node is paramount when interacting with smart contracts. Several API providers already exist, but this approach includes an extra dependency in development. Moreover, these tools require an already provided JSON-RPC node, instead of a locally-managed one. Hence, the scope of this work is to develop a middleware, which provides an API, to interact with a local node using HTTP calls.

In particular, the work focuses on an Ethereum middleware and, in turn, on smart contracts programmed using Solidity. The main technology that allows building the middleware is Nest, a backend framework built on top of Express. And to interact with the blockchain, it uses the Ethers library.

Keywords: Ethereum, Solidity, Nest, Ethers.

1 Introduction

Since Ethereum's inception in 2013, blockchains based on smart contracts have been widely used in decentralized applications, or dApps, in what is called the Web3 paradigm. This paradigm incorporates decentralization into the World Wide Web. Its main advantages are privacy, security and scalability.

Due to the decentralization of blockchain networks, they are comprised of nodes that interact with each other using a consensus algorithm. Outside actors, or regular users, of the blockchain may want to interact with the blockchain, either to send coins or to use a smart contract. The interaction is achieved through a JSON-RPC API provided by the nodes.

Cryptocurrency wallets, or clients, need to use the API to perform the transactions. But other applications may want to connect to the node using a higher level of abstraction. For example, using an API library. In the case of Ethereum, there are a wide array of libraries[1].

However, even though there are several API providers, like Infura[2], that usually require using a remote node instead of a locally-managed one. This approach breaks with the goal of decentralization. Moreover, a developer may not want to rely on an external service since it's another dependency.

For this reason, this work focuses on developing a Nest-based middleware/backend, with a local node, which provides an API library. Nest is a backend framework built on top of Express, but it may support other frameworks. Nest, which was inspired by Angular, has a high degree of modularity, which makes it rather flexible. The JSON-RPC wrapper used to interact with the Ethereum blockchain is Ethers.

Furthermore, this work also includes some use cases implemented in React, a powerful framework for building reactive frontends in single page applications.

1.1 Project development planning

The project is split into four tasks. The first two are related to developing the backend with Nest and Ethers, and the second two to making the React frontend for each use case. Tables 5, 2, 3 and 4.

Study Nest	
Description Since Nest is a rather new framework, some study of its main functionalities is required. The sources are the official documentation and a Udemy course on the topic by Stephen Grider[3].	Start event: 08/02/2023 End event: 14/03/2023

Table 1: Study Nest task.

Develop middleware	
Description The main development of the middleware with Nest and Ethers. This is the main bulk of work of the thesis. Regular meetings with the advisors are required to monitor the development and to consider new functionalities.	Start event: 14/03/2023 End event: 21/04/2023

Table 2: Develop middleware task.

Study React	
Description A reminder of key React concepts with the help of the React video series from Postgraduate course in Full-Stack Web Technologies[4].	Start event: 21/04/2023 End event: 13/05/2023

Table 3: Study React task.

Develop use cases	
Description Development of simple use cases that use the aformentioned API.	Start event: 13/05/2023 End event: 17/06/2023

Table 4: Develop use cases task.

Documentation	
Description Writing this work's Thesis. In addition, since this work has been done in conjunction with an internship at Eseleos SL[5], the task also includes writing extensive documentation of Nest for employee training.	Start event: 14/03/2023 End event: 30/06/2023

Table 5: Documentation task.

The tasks span 5 months, from February to June. The Gantt diagram of the tasks is shown in figure 1.

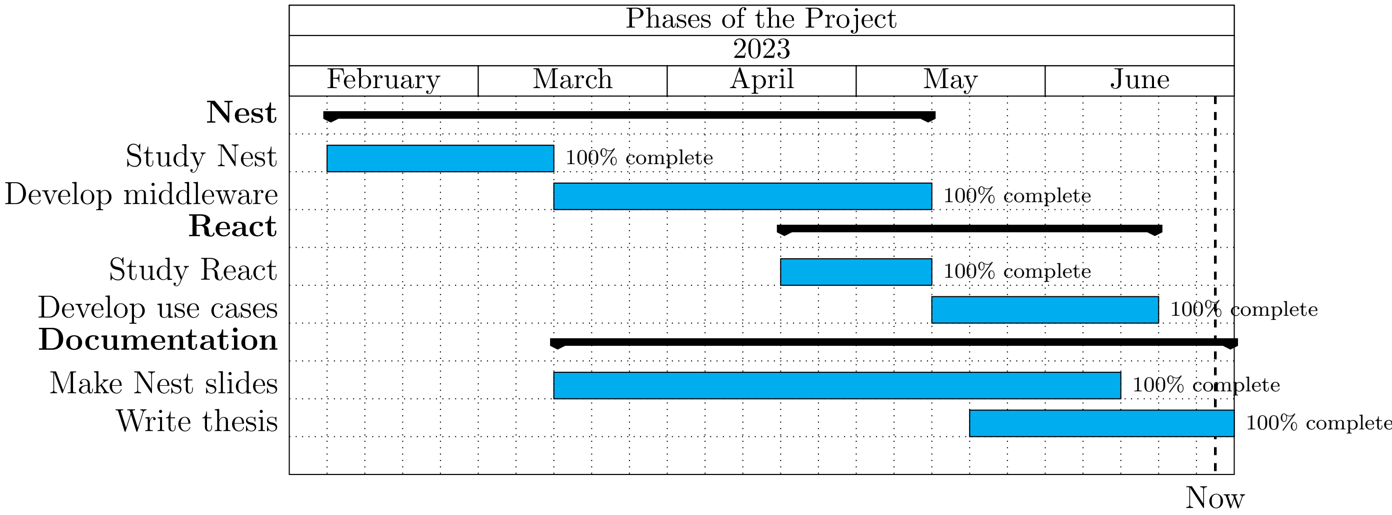


Figure 1: Project's Gantt diagram

2 Background

2.1 Blockchain

A blockchain is a distributed, immutable chain of blocks of data, where each block points to the previous one. Immutability is achieved through the hash values of the previous block, which is referenced in the current one. The main advantage of immutability is data integrity.

The first widespread use of this technology is in the Bitcoin protocol. In Bitcoin, the blockchain provides the basis for a distributed ledger of the transactions between actors. Each block contains a list of transactions signed by the owners of the coins.

Adding a new block into the blockchain is achieved with a consensus algorithm between the nodes of the network. Proof of Work and Proof of Stake are the most commonly used. In both cases, a node is chosen to add the new block. The chosen node is rewarded in cryptocurrency.

2.1.1 Proof of Work

In Proof of Work (PoW), each node needs to be the first to solve a problem in order to be chosen. The problem can be loosely defined as follows:

- Let input x be a sequence of bytes representing the new block.
- Let difficulty d be an integer.
- Let n be an arbitrary nonce.
- Let h be a hash function, like SHA-256, which takes a sequence of bytes and outputs a sequence of bytes or digest.
- Let z be a function that returns the number of leading zeros of a sequence of bytes.
- Compute $y = h(x||n)$.
- If $d == z(y)$, success. Otherwise, repeat with a new n.

The hash is defined at the protocol level. The difficulty changes depending on the necessity of the network in order to have the frequency of new blocks be constant. Nodes participating in this algorithm are called miners.

Proof of Work is used in UTXO-based blockchains, in which each coin has its own public address. In other words, the coins are not fungible. Miners are incentivized to behave honestly by the cost of acquiring and running the hardware for mining.

2.1.2 Proof of Stake

Proof of Stake (PoS) requires ”locking” a balance as collateral. There is a minimum amount of coins to be able to lock. Once the coins are locked, they cannot be used. This is achieved using a smart contract. The locked amount equals to a probability of being chosen. Nodes participating in this algorithm are called validators.

PoS is used in balance-based blockchains, in which each account has assigned a value of coins. This is due to the smart contract requirement. In this case, the coins are fungible. Validators are incentivized to behave honestly by the collateral locked in the smart contract.

2.2 Ethereum

The Ethereum blockchain was created in 2013. Since its inception, it has become the go-to blockchain for deploying smart contracts. In 2022, Ethereum switched from a PoW to a PoS consensus algorithm[6]. Contrary to PoW, which requires a lot of computing power, PoS is much more energy efficient.

Ethereum provides the Ethereum Virtual Machine The EVM is a stack-based state-machine, an abstraction of the execution of a program in the nodes of the blockchain network[7]. Smart contracts are programmed in languages like Solidity and then compiled into EVM instructions. Once deployed, the instructions are stored as bytecode in a block. The instructions are executed by a node if the payment of a fee is provided in the native Ethereum cryptocurrency.

Smart contracts have been widely used in decentralized applications, or dApps, in what is called the Web3 paradigm. This paradigm incorporates decentralization into the World Wide Web. Its main advantages are privacy, security and scalability.

2.3 Ethers

Ethers is a TypeScript-based Node.js module that allows an application to interact with Ethereum nodes[8]. This enables a connection to a node with using a higher level of abstraction in what is called an API library.

In Ethereum, clients need to contact nodes using a JSON-RPC API. An API library is a wrapper of the JSON-RPC client functionalities. More abstraction helps in detaching the logic of the application from the blockchain. API libraries outsource the validation and security checks. This reduces the development overhead of the Security by Design methodology. Now, a developer does not need to take into account how the JSON-RPC API is defined, and only focus on the main implementation of the application.

2.4 Nest

NestJS, or simply Nest, is an open-source, extensible, versatile, progressive Node.js framework for building efficient, scalable Node.js server-side applications (backend).

It is built with and fully supports TypeScript (yet still enables developers to code in pure JavaScript) and combines elements of OOP (Object Oriented Programming), FP (Functional Programming), and FRP (Functional Reactive Programming).

Its main features are:

- Easy to use, learn and master.
- Powerful Command Line Interface (CLI) to boost productivity and ease of development.
- Detailed and well-maintained documentation.
- Active codebase development and maintenance.
- Support for dozens of nest-specific modules to integrate with common technologies and concepts like TypeORM, Mongoose, GraphQL, Logging, Validation, Caching, WebSockets and much more.
- Easy unit-testing applications.
- On top of NestJS you can easily build Rest API's, MVC applications, microservices, GraphQL applications, Web Sockets or CLI's and CRON jobs.

2.4.1 Architecture

Roughly, a Nest application will take requests, process it in some way and return a response.

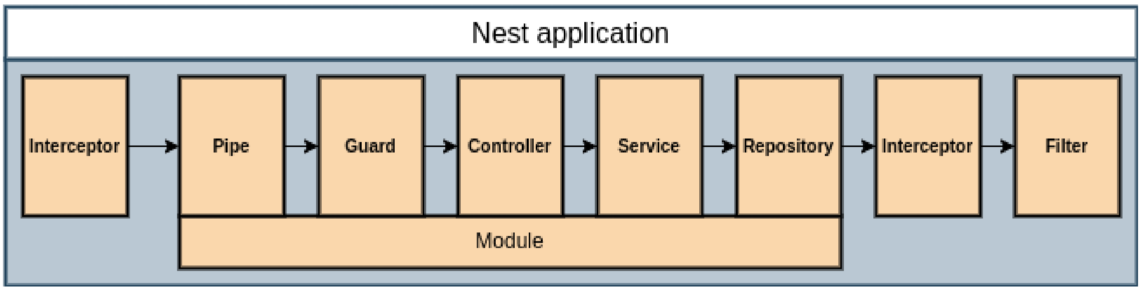


Figure 2: Architecture of a Nest application.

The requests handling logic is divided into modular blocks. Each type of block is intended for a specific purpose. Nest has tools to help writing this blocks in a fast and efficient way. Several building blocks are packed in one module.

Nest, out of the box, provides the following list of building blocks types:

- Pipes: Validates data contained in the requests.
- Guards: Handles authentication strategies.
- Controllers: Handles incoming requests by routing it to a particular function.
- Services: Handles business logic execution or access data through a repository.
- Repositories: Handles data stored in a DB (stores or retrieves data).
- Interceptors: Adds extra logic to incoming requests or outgoing responses.
- Filters: Handles errors that may occur during request handling.
- Modules: Groups together different building blocks

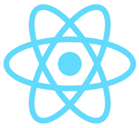
Nest modularity allows developing reusable logical parts that can be used across different types of applications. Nest provides an out-of-the-box application architecture which allows developers and teams to create highly testable, scalable, loosely coupled, and easily maintainable applications.

2.4.2 More information

This work has been done in conjunction with an internship at Eseleos. For this reason, extensive documentation on how Nest works had to be made for employee training. The slides are submitted as a separate file.

2.5 React

React is a framework for developing Single-Page Applications (SPAs). It is open source and being developed by Meta. React uses inversion of control to allow developers to program components while the framework deals with the changes. It is reactive in nature.



Reactive programming is a declarative paradigm. The developer specifies how a page should be, and the framework is tasked with updating the DOM with the changes. To do this, React has a virtual DOM, mirroring the actual HTML DOM tree. Whenever a change occurs, the framework compares the virtual DOM with the real one in an HTML page, and does the necessary changes.

Components return a JSX expression, which is a JavaScript equivalent of an HTML fragment. It is possible to work with TypeScript in TSX[9]. Components can be nested and can have properties, much like HTML elements do. They can be either classes or functions. However, lately the latter approach is more commonly used.

React also introduced the concept of hooks[10], which lets components access different React features. The following are the most relevant:

- **useState** lets components have a state. States in React are internal attributes that, unlike regular props, can change dynamically. This enables a page to update its appearance in relation to events like clicking on a button or inputting text.
- **useEffect** lets components tap into side effects tied to the updating of the value of an attribute. This is useful if a component relies on an external input of the React page, like an API call to a server or to the browser.
- **useRef** lets components hold the reference to a DOM object. This can be used to access a specific part of the HTML manually. However, this should be avoided as it can result in unexpected behaviour.

3 Methodology

This section describes the methodology of the development of the middleware/backend using Nest and the frontend using React.

3.1 Development

Development is done using VSCodium[11], an open-source and telemetry-free alternative to Microsoft's VSCode, which is a modular IDE with a wide range of extensions. The ones that are used are:

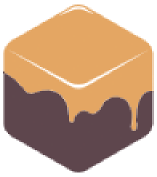
- **REST Client** allows for defining HTTP calls to the endpoints of the middleware[12].
- **SQLite** provides tools for visualizing and interacting with an SQLite database[13].

The smart contracts have been coded using the Remix IDE[14]. This environment, developed by the Ethereum team, allows for compiling Solidity code and deploying it through a JSON-RPC node.

To test the backend, Ganache is used to simulate an Ethereum blockchain[15]. Ganache generates a mnemonic seed, easier to remember than a private key, and a set of 10 private keys for testing. It also provides a local node with a JSON-RPC API. Moreover, Ganache can either mine blocks instantaneously or have a fixed delay.

(a) VSCodium

(b) Remix



(c) Ganache

The backend is developed using Nest v9 (July 2022)[16], while Prisma v4 (June 2022) was chosen as the ORM for this project[17][18]. Prisma provides a virtual relation field for working with foreign keys. It also provides a type-safe model in a specific schema format. For development purposes, the used DBMS is SQLite v3 (September 2004)[19].

A little clarification on previous concepts:

- **ORM** means Object-Relational Mapping. It is a tool for mapping an entry/model in a database with an object in the application. It makes it easier to interact with the DB.
- **DBMS** means DataBase Management System. It is a piece of software that manages the data of the databases. Examples of DBMS are SQLite, MySQL, Oracle and MongoDB.

The frontend is developed using React v18 (March 2022) together with Axios v1 (October 2022)[20][21]. The latter is a lightweight package for making HTTP calls that can be used both in a browser and in Node.js.

Git is employed as the version management system, in conjunction with GitHub. The resulting repositories are:

- **Ethereum-NestJS-Middleware** for the backend[22].
- **Ethereum-NestJS-Frontend** for the frontend[23].

3.2 Documentation

Notion was used during the study of Nest and its functionalities[24]. The website, which is a great tool for taking notes, is designed for making documentation and wikis.

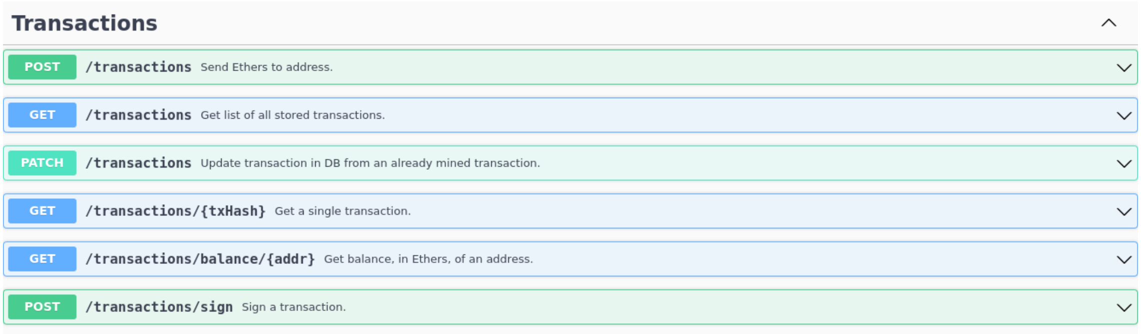
This thesis has been written using LaTeX. The Nest slides are also made in this way. Diagram 2 is made with draw.io/diagrams.net[25]. Figure 6 is made with a Prism ERD tool[26].

In the next section, the resulting application will be analysed.

4 Evaluation

The middleware has been split into two modules: *transactions* and *contracts*.

- *transactions* focuses on the transaction logic and balances. See diagram 4 for a full list of endpoints.
- *contracts* focuses on deploying and verifying smart contracts. More importantly, it manages the calls to contract methods. See diagram 5 for a full list of endpoints.



Transactions

POST /transactions Send Ethers to address.

GET /transactions Get list of all stored transactions.

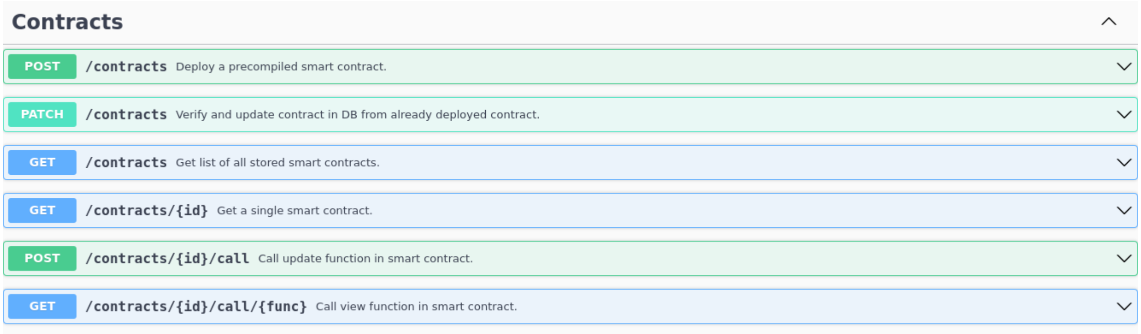
PATCH /transactions Update transaction in DB from an already mined transaction.

GET /transactions/{txHash} Get a single transaction.

GET /transactions/balance/{addr} Get balance, in Ethers, of an address.

POST /transactions/sign Sign a transaction.

Figure 4: Swagger fragment of transaction API.



Contracts

POST /contracts Deploy a precompiled smart contract.

PATCH /contracts Verify and update contract in DB from already deployed contract.

GET /contracts Get list of all stored smart contracts.

GET /contracts/{id} Get a single smart contract.

POST /contracts/{id}/call Call update function in smart contract.

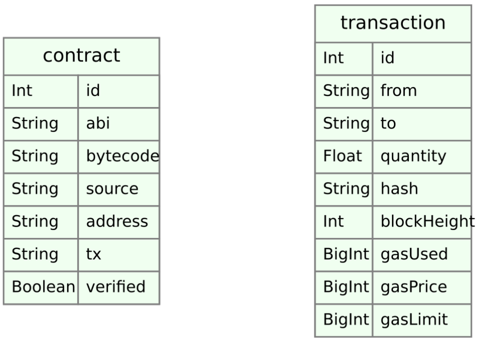
GET /contracts/{id}/call/{func} Call view function in smart contract.

Figure 5: Swagger fragment of smart contract API.

For a detailed definition of the API following the OpenAPI specification, in YAML format, see annex 7.2[27].

The database is used as a ”caching” mechanism. It allows for accessing blockchain information without having to connect to an actual node. This approach is a good solution to a slow or unreliable network. There are also endpoints for manually adding transactions and contracts that have already been deployed on the blockchain. The object-relational diagram is shown in figure 6. Notice that the tables are decoupled. Annex 7.1 shows the Prisma schema of the database.

The backend supports two modes of using private keys: *internal* and *provided*. In the *internal* mode, it uses a key specified in the PKEY environment variable. In the *provided* mode, on the other hand, the private key is given as part of the body in a *mnemonic* property. The property needs to follow the BIP32 standard[28]. This property contains:



contract	
Int	id
String	abi
String	bytecode
String	source
String	address
String	tx
Boolean	verified

transaction	
Int	id
String	from
String	to
Float	quantity
String	hash
Int	blockHeight
BigInt	gasUsed
BigInt	gasPrice
BigInt	gasLimit

Figure 6: ER diagram of the database.

- mnemonic: A 12-word string of the mnemonic.
- password: An optional string for the password used to protect the mnemonic.
- path: An optional string of the hierarchical deterministic wallet. The structure follows diagram 7.

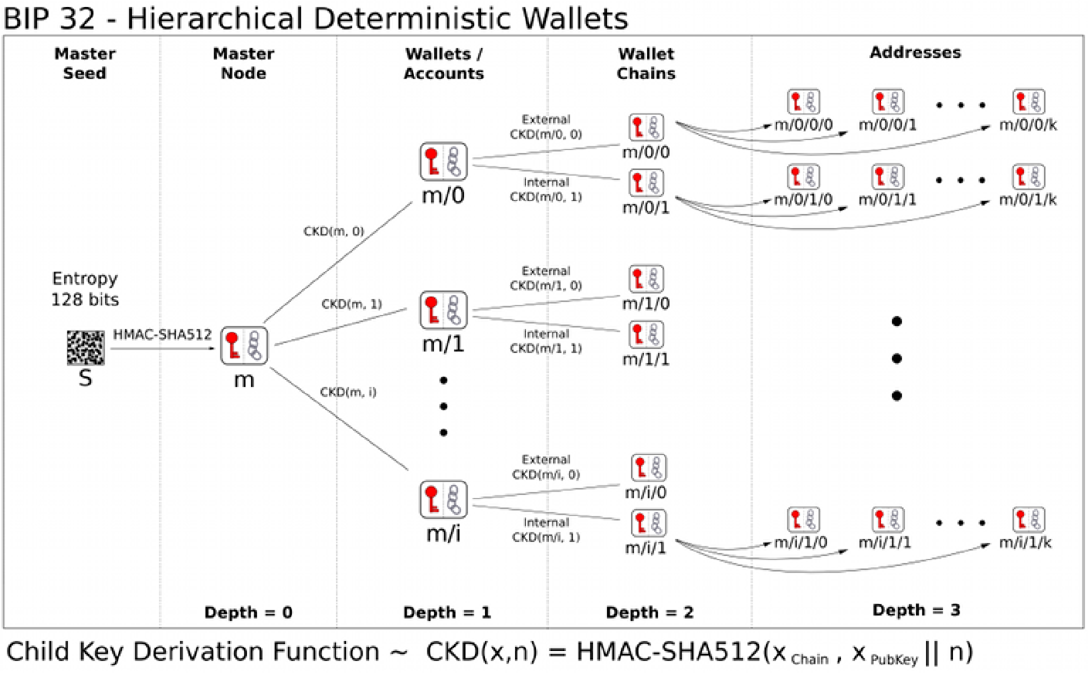


Figure 7: Derivation of BIP32 HD wallets. Source: BIPS[29].

4.1 Endpoints

The following sections delve deeper into relevant aspects of the endpoints.

4.1.1 Transactions

First, in the *transactions* module, there is the possibility of signing a transaction without sending it, through a POST call to */api/transactions/sign*. This can be used for generating the transactions locally and sending them once the network is reliable.

The same POST call to */api/transactions* can be used for sending an already signed transaction. The request body of that endpoint can have either a *new* or *raw* property containing the transaction. The latter is used for this purpose.

Secondly, Ethers makes a distinction between the type of data returned by some functions.

- **TransactionResponse** refers to any type of transaction. It can be either mined or not. *JsonRpcProvider.getTransaction()*, *JsonRpcProvider.broadcastTransaction()* and BaseWallet.sendTransaction() are functions that return it. Making calls to a smart contract, as well.
- **TransactionReceipt** refers to mined transactions. In other words, transactions that have been added into a block. *JsonRpcProvider.getTransactionReceipt()* returns this.

There is a PATCH call to */api/transactions* to update the information of a mined transaction. In order to do this, the client needs to send the hash of the transaction. This call gives the possibility of saving/caching remote transactions in the database.

4.1.2 Contracts

For the smart contracts module, it was decided that an option to verify the source code could be an added plus to the application. At first, third-party options like Etherscan were considered[30]. This blockchain explorer has a built-in code verifier accessible through an API. However, there is a maximum of 5 calls per second on the free subscription. To prevent further expenses with paid subscriptions and to avoid another dependency, it was decided that verification would be done locally.

To do this, the middleware employs the solc package, which is actually developed by the Ethereum team[31]. The package enables the compilation of Solidity code in Node.js. It also allows for runtime downloading of compiler versions, which makes it rather versatile.

Contracts POSTed to */api/contracts* can be directly verified or not. If it is the latter case, verification is done through a PATCH call to */api/contracts*. Internally, the middleware takes the source code, compiler version and file name to compile the Solidity code into bytecode. It then checks if the bytecodes match to verify the code.

It is important to note that contracts do not require being verified for making calls to them. But the *verified* attribute allows for the auditing of the smart contract.

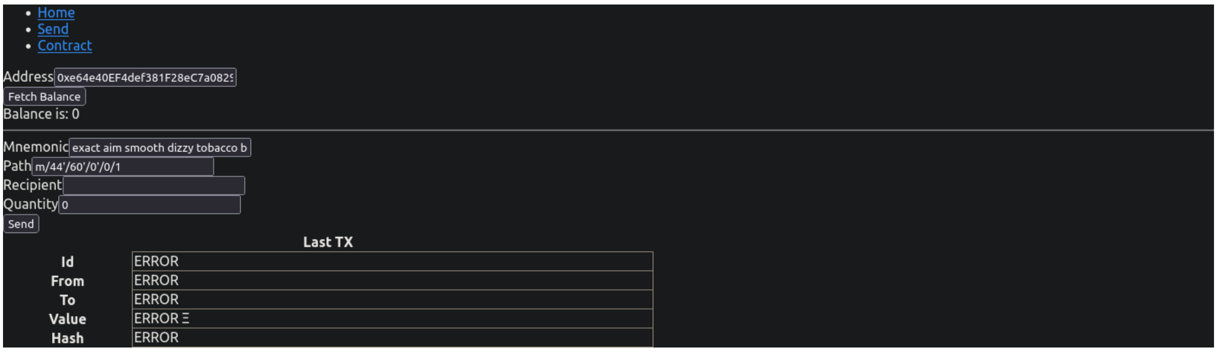
Similarly to the *transactions* module, the PATCH call allows for loading the data related to a remotely-deployed contract.

4.2 Frontend

Two use cases have been developed for the frontend. They match each of the two modules in the middleware.

4.2.1 Send

The first uses the *transactions* module to send Ethereum to a recipient. It works in the *provided* mode.



- Home
- Send
- Contract

Address 0xe64e40EF4def381F28eC7a082!
Fetch Balance
Balance is: 0

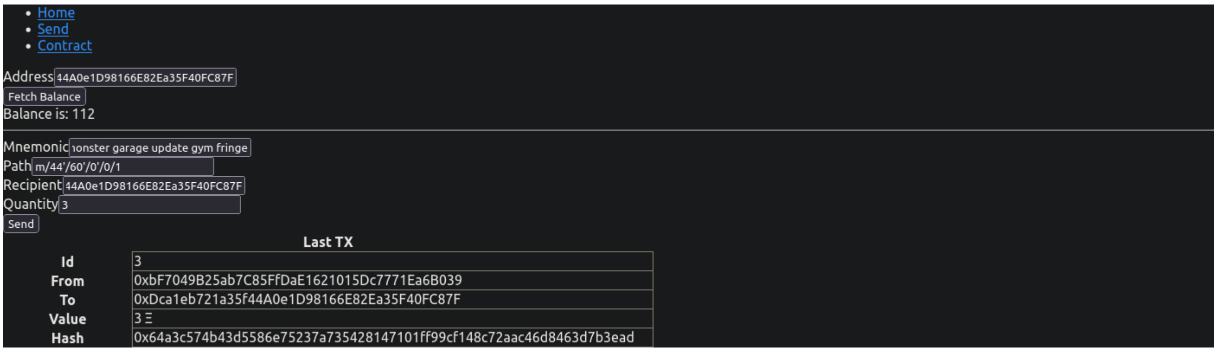
Mnemonic exact aim smooth dizzy tobacco b
Path m/44'/60'/0'/0/1
Recipient
Quantity 0
Send

Last TX

Id	ERROR
From	ERROR
To	ERROR
Value	ERROR Ξ
Hash	ERROR

Figure 8: Send page.

It is divided into two sections, one for checking an account's balance and the other for sending the coins. The latter is more relevant. The mnemonic and HD path need to be chosen as required by the *provided* mode. And finally there are the inputs for the recipient and amount, in ETH. Details about the generated transaction are shown in the table below.



- Home
- Send
- Contract

Address 44A0e1D98166E82Ea35F40FC87F
Fetch Balance
Balance is: 112

Mnemonic ıonster garage update gym fringe
Path m/44'/60'/0'/0/1
Recipient 44A0e1D98166E82Ea35F40FC87F
Quantity 3
Send

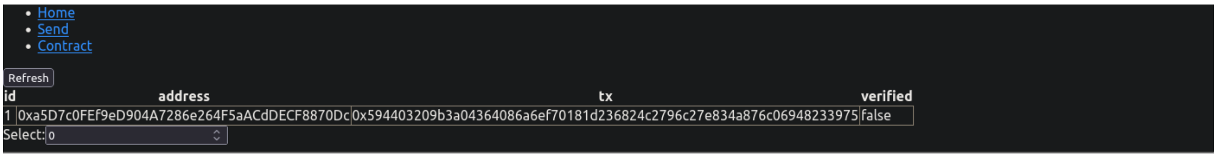
Last TX

Id	3
From	0xbF7049B25ab7C85FfDaE1621015Dc7771Ea6B039
To	0xDca1eb721a35f44A0e1D98166E82Ea35F40FC87F
Value	3 Ξ
Hash	0x64a3c574b43d5586e75237a735428147101ff99cf148c72aac46d8463d7b3ead

Figure 9: Send page after having sent 3 ETH.

4.2.2 Contract

The second uses the *contracts* module to make calls to a smart contract. It works in the *internal* mode. It contains three sections.



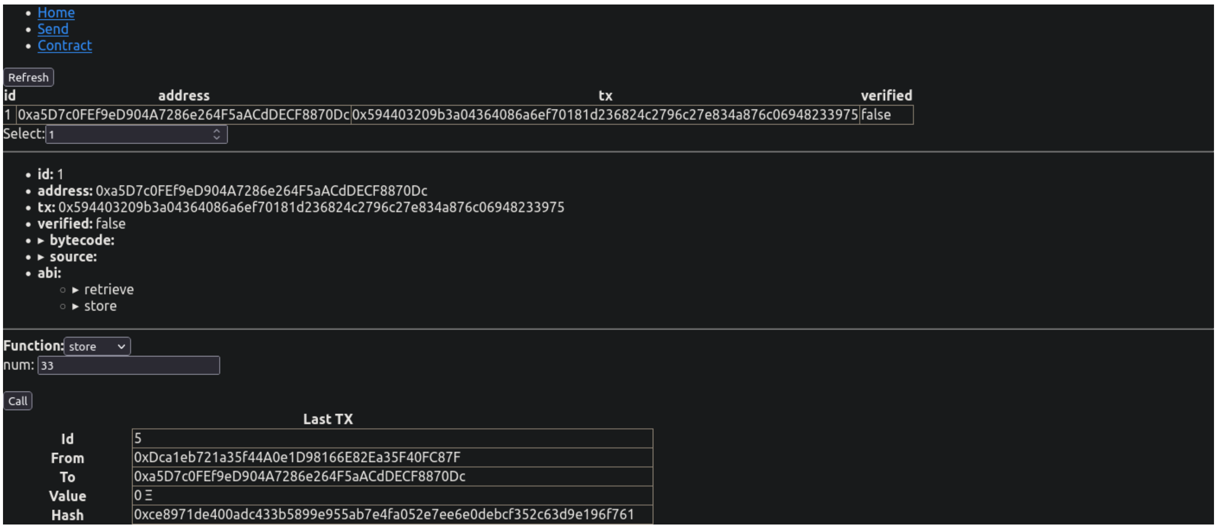
id	address	tx	verified
1	0xa5D7c0FEf9eD904A7286e264F5aACdDECF8870Dc	0x594403209b3a04364086a6ef70181d236824c2796c27e834a876c06948233975	false

Select: 0

Figure 10: Contract page with a simple storage available.

First, a list of available smart contracts show up. The table shows the address of the smart contract and the transaction that deployed it. It also tells if the source code has been verified.

Second, once a contract is selected, more relevant information will appear. Information like the ABI, bytecode and source code. The ABI section contains a drop-down menu with each call and their properties. For example, the inputs, outputs or whether it's payable or not.



id	address	tx	verified
1	0xa5D7c0FEf9eD904A7286e264F5aACdDECF8870Dc	0x594403209b3a04364086a6ef70181d236824c2796c27e834a876c06948233975	false

Select: 1

- **id:** 1
- **address:** 0xa5D7c0FEf9eD904A7286e264F5aACdDECF8870Dc
- **tx:** 0x594403209b3a04364086a6ef70181d236824c2796c27e834a876c06948233975
- **verified:** false
- ▸ **bytecode:**
- ▸ **source:**
- **abi:**
  - ▸ retrieve
  - ▸ store

Function: store

num: 33

Last TX	
Id	5
From	0xDca1eb721a35f44A0e1D98166E82Ea35F40FC87F
To	0xa5D7c0FEf9eD904A7286e264F5aACdDECF8870Dc
Value	0 Ξ
Hash	0xce8971de400adc433b5899e955ab7e4fa052e7ee6e0debcf352c63d9e196f761

Figure 11: A store call has been made with value '33'.

Third, the last section shows a selector for the smart contract call. If the call accepts one or more inputs, one or more text inputs appear in order to be filled. Once a transaction-generating call is made, the same transaction information table from the other use case shows up. If the call does not generate a transaction, only the returned value is shown.

5 Budget

This section presents the estimated budget needed in order to develop and maintain the middleware.

5.1 Development

The development budget takes into account the price of the laptop in which the application was developed. In this case, it has been an Acer Predator 300PT315-53. It also takes into account an ECTS of 25h/credit. So the total amount of hours is $25 * 12 = 300$.

Concept	Hours	€/h	Cost (€)
Developer	300	25	7500
Acer Laptop			910
Total (€)			**8410**

Table 6: Estimated budget for the development.

5.2 Maintenance

The maintenance budget takes into the account the wages of a fullstack developer working half-time. The amount of hours is, approximately, 86.67 (21.67 days/month in average). For running the middleware and frontend, the budget also takes into account a VPS with 8GB of RAM and 4 CPU cores, for example[32][33][34]. This is about 45€/month.

Concept	Hours	€/h	Cost (€)
Developer	86.67	25	2166.67
VPS			45
Total (€/month)			**2211.67**

Table 7: Estimated budget for the maintenance.

6 Conclusions and future development

To start off, working in TypeScript taught me that it helps in avoiding bugs coming from wrong variable types. Coupling this with VSCodium's linting make it very unlikely to have type-related errors.

In addition, I found out that Ethers has extensive documentation and is straightforward to use. It is important to note that the package's functionalities used in this work are a small portion of all that it can do.

Moreover, I learned that proper testing needs to be done to avoid errors. So using tools like Ganache or the REST client extension helped in this regard.

Finally, I discovered that Solidity is a simple and easy to understand smart contract language. And that Remix errors help a lot in understanding how the language works. Because of this, it might seem that it is more restrictive. In this regard, Solidity is like C or C++.

All in all, the development of this work taught me that smart contracts provide a flexible dApp framework. And also that blockchain is a promising technology. Working in a controlled environment, with extensive documentation at my disposal, was key in the development of a secure and bug-free application.

6.1 Future work

Despite this, there are several improvements that have not been made due to time constraints. They will be explained in this section.

6.1.1 Security

The first and most important improvement would be to use certificates to ensure a secure channel. Using TLS would ensure that no information regarding the private key is leaked. In principle, a replay attack should not be possible due to the nature of Ethereum transactions using a nonce. A man-in-the-middle attack should also not be possible due to transactions being digitally signed.

The second improvement would be to do a lot of testing of the overall application in what is called E2E (end-to-end) testing. This is a must in order to prevent unexpected behaviour from the client's point of view. Luckily, Nest uses the Jest library, which provides a simple testing framework[35].

Another security improvement would be to enable Cross-Origin Resource Sharing, or CORS. This would make the middleware be available only from authorized domains. In other words, forcing the use of the application through specific web pages, which may add additional input validation. Because of this, it would be harder to exploit the API from an HTTP client. However, this would prevent the middleware from providing an open API.

Moreover, even though private keys can be manually provided by the *mnemonic* property, it would make more sense to store the keys in the database. This solution would require some considerations. The first, that proper access control policies would need to be implemented. The second, that keys should be securely stored in the database, preferably salted and peppered. Third, that it would add a considerable storage overhead. And fourth, that it would require storing the keys in a relatively centralized way, deviating from the goal of dApps.

6.1.2 Deployment

For the testing environment, using a testnet would provide a more realistic scenario for testing. Testnets like Sepolia or Goerli are recommended by Ethereum[36]. Even though this is mitigated by using a fixed delay between blocks, it may not be enough to satisfy all possible scenarios. For example, a scenario in which the network is split, with more than one consensus, has not been tested yet.

For a production environment, it would make sense to dockerise the middleware. This would greatly speed up the setup and replication of the software, as well as make it more scalable. We could even set up a load balancer for handling large volumes of requests. Or, alternatively, deploy it on cloud providers like Microsoft Azure or Amazon Web Services[37][38].

Even though SQLite was used for development, it is not intended to be the final DBMS used. Prisma offers a list of database connectors compatible with its client[39]. The most relevant to the requirements of this work is CockroachDB, which is a SQL-based distributed DBMS centred around scalability and survivability[40]. Coupling this with the previously mentioned cloud deployment would ensure the resiliency of the middleware.

Furthermore, detaching the database would be another improvement. Doing so would require following a microservice architecture, which is easier to manage. This allows for swapping and scaling specific components, such as the DBMS, of the overall application with relative ease.

6.1.3 Compilation

The next improvements involve the smart contract compiler.

First, a new module could be considered for only compiling smart contracts, without interacting with the blockchain. This would cut off another dependency in development. This would be especially effective if the frontend provided a compiling interface.

Second, having solc download major compiler versions on startup would greatly reduce the response time of some requests. Even though the library caches the compilers, clients still need to wait for the compiler to download if it is a non-cached version.

And third, adding support for other smart contract languages could make the application more flexible for developers. Languages like Vyper, which resembles Python, or Yul, which is more low-level, should be considered[41].

Another improvement would be to switch from Ganache to Hardhat[42]. This Node.js framework speeds up smart contract development by providing compilation and testing tools. A new *contract_dev* module could be made to use Hardhat via HTTP requests to avoid programming errors and help in debugging Solidity code.

6.1.4 Budget

Transactions may sometimes require a high gas fee. The solution to this would be to allow metatransactions[43]. They allow a client to sign a transaction without having to pay the fee. This is achieved with a smart contract that relays the transaction. The actual fee is then paid by the middleware.

In this work's case, a fee policy should be implemented depending on the use case. An example could be a fee paid at the end of the month in euros for a limited amount of calls to transaction-generating endpoints. The due amount should take into account the cost of maintaining the middleware.

Note that there is already a pseudo-metatransaction mechanism. This is implemented with a private key, provided by the middleware in the *internal* mode.

References

[1] Ethereum. Backend API libraries. https://ethereum.org/en/developers/docs/apis/backend. (accessed: 21.05.2023).

[2] Infura. BUILD SCALE DISRUPT. https://www.infura.io/. (accessed: 22.06.2023).

[3] Stephen Grider. NestJS: The Complete Developer's Guide. https://www.udemy.com/course/nestjs-the-complete-developers-guide. (accessed: 22.05.2023).

[4] Pablo Fernández Duran & Jose L. Muñoz Tapia. Postgraduate course in Full-Stack Web Technologies. https://www.talent.upc.edu/ing/estudis/formacio/curs/313400/posgrado-full-stack-web-technologies. (accessed: 24.05.2023).

[5] ESELEOS. https://www.eseleos.com/. (accessed: 14.06.2023).

[6] Bhaskar Kashyap & other contributors. PROOF-OF-STAKE (POS). https://ethereum.org/en/developers/docs/consensus-mechanisms/pos. (accessed: 01.06.2023).

[7] Joshua & other contributors. ETHEREUM VIRTUAL MACHINE (EVM). https://ethereum.org/en/developers/docs/evm. (accessed: 01.06.2023).

[8] Richard Moore. The Ethers Project. https://www.npmjs.com/package/ethers. (accessed: 01.06.2023).

[9] Create React App. Adding TypeScript. https://create-react-app.dev/docs/adding-typescript. (accessed: 10.06.2023).

[10] React. Built-in React Hooks. https://react.dev/reference/react. (accessed: 10.06.2023).

[11] Peter Squicciarini & other contributors Baptiste Augrain. VSCodium - Open Source Binaries of VSCode. https://vscodium.com. (accessed: 05.06.2023).

[12] Huachao Mao. REST Client. https://open-vsx.org/extension/humao/rest-client. (accessed: 10.06.2023).

[13] Alex Covizzi. SQLite. https://open-vsx.org/extension/alexcvzz/vscode-sqlite. (accessed: 10.06.2023).

[14] Ethereum. Remix - Ethereum IDE. https://remix.ethereum.org/. (accessed: 21.06.2023).

[15] Truffle. Ganache - Truffle Suite. https://trufflesuite.com/ganache. (accessed: 05.06.2023).

[16] Kamil Mysliwiec. NestJS v9 is now available! https://trilon.io/blog/nestjs-9-is-now-available. (accessed: 18.06.2023).

[17] Prisma Team. Next-generation Node.js and TypeScript ORM. https://www.prisma.io. (accessed: 05.06.2023).

[18] Alex Ruheni. What's new in Prisma? (Q2/22). https://www.prisma.io/blog/wnip-q2-2022-pmn7rulcj8x#releases--new-features. (accessed: 18.06.2023).

[19] SQLite. SQLite Older News. https://www.sqlite.org/oldnews.html. (accessed: 18.06.2023).

[20] The React Team. React v18.0. https://react.dev/blog/2022/03/29/react-v18. (accessed: 18.06.2023).

[21] Axios contributors. Release v1.0.0. https://github.com/axios/axios/releases/tag/v1.0.0. (accessed: 18.06.2023).

[22] Marc Cosgaya Capel. Ethereum-NestJS-Middleware. https://github.com/MarcCosgaya/Ethereum-NestJS-Middleware. (accessed: 18.06.2023).

[23] Marc Cosgaya Capel. Ethereum-NestJS-Frontend. https://github.com/MarcCosgaya/Ethereum-NestJS-Frontend. (accessed: 18.06.2023).

[24] Notion Labs. Notion. https://www.notion.so/. (accessed: 22.06.2023).

[25] draw.io. https://www.drawio.com/. (accessed: 22.06.2023).

[26] John Fay Simon Knott. Prisma ERD. https://prisma-erd.simonknott.de. (accessed: 10.06.2023).

[27] OpenAPI Initiative. https://www.openapis.org/. (accessed: 22.06.2023).

[28] Ethers. HD Wallet. https://docs.ethers.org/v5/api/utils/hdnode. (accessed: 06.06.2023).

[29] bitcoin. bips. https://github.com/bitcoin/bips. (accessed: 06.06.2023).

[30] Etherscan. https://etherscan.io/. (accessed: 20.06.2023).

[31] Ethereum. solc. https://www.npmjs.com/package/solc. (accessed: 20.06.2023).

[32] Vultr. Cloud Compute. https://www.vultr.com/pricing/#cloud-compute/. (accessed: 28.06.2023).

[33] Kamatera. Cloud Servers. https://www.kamatera.com/Products/201/Cloud_Servers#page_250. (accessed: 28.06.2023).

[34] RamNode. Cloud VPS Hosting Pricing. https://ramnode.com/pricing/. (accessed: 28.06.2023).

[35] Jest. Delightful JavaScript Testing. https://jestjs.io/. (accessed: 27.06.2023).

[36] Ethereum. Ethereum Testnets. https://ethereum.org/en/developers/docs/networks/#ethereum-testnets. (accessed: 22.06.2023).

[37] Microsoft. Learn, connect, and explore. https://azure.microsoft.com/en-us/. (accessed: 18.06.2023).

[38] Amazon. Start Building on AWS Today. https://aws.amazon.com/. (accessed: 18.06.2023).

[39] Prisma. Database connectors. https://www.prisma.io/docs/concepts/database-connectors. (accessed: 18.06.2023).

[40] CockroachDB. Build what you dream. https://www.cockroachlabs.com/product/. (accessed: 18.06.2023).

[41] Ethereum. SMART CONTRACT LANGUAGES. https://ethereum.org/en/developers/docs/smart-contracts/languages/. (accessed: 19.06.2023).

[42] Hardhat. Ethereum development environment for professionals. https://hardhat.org/. (accessed: 18.06.2023).

[43] CoinMarketCap Alexandria. Metatransaction. https://coinmarketcap.com/alexandria/glossary/metatransaction. (accessed: 19.06.2023).

7 Appendices

7.1 Prisma schema file of the database

```
// This is your Prisma schema file,
// learn more about it in the docs: https://pris.ly/d/prisma-schema

generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "sqlite"
  url      = env("DATABASE_URL")
}

model contract {
  id Int @id @default(autoincrement())
  abi String
  bytecode String
  source String?
  address String @unique
  tx String @unique
  verified Boolean
}

model transaction {
  id Int @id @default(autoincrement())
  from String
  to String
  quantity Float
  hash String @unique
  blockHeight Int?
  gasUsed BigInt?
  gasPrice BigInt?
  gasLimit BigInt
}
```

7.2 Definition, in YAML format, of the API following the OpenAPI specification.

```
openapi: 3.0.0
paths:
  /contracts/{id}/call/{func}:
    get:
      operationId: ContractsController_viewFunction
      summary: Call view function in smart contract.
      parameters:
        - name: id
          required: true
          in: path
          description: Contract id.
          example: 3
          schema:
            type: number
        - name: func
          required: true
          in: path
          description: Function name in smart contract.
          example: get
          schema:
            type: string
        - name: args
          required: true
          in: query
          description: List of arguments of the function.
          example:
            - a
            - b
            - c
          schema:
            type: array
            items:
              type: string
      responses:
        '200':
          description: ''
      tags: &ref_0
        - Contracts
  /contracts/{id}/call:
    post:
      operationId: ContractsController_updateFunction
      summary: Call update function in smart contract.
      parameters:
        - name: id
          required: true
          in: path
          description: Contract id.
          example: 3
          schema:
            type: number
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/UpdateFunctionBodyDto'
      responses:
        '201':
          description: ''
      tags: *ref_0
  /contracts:
    post:
      operationId: ContractsController_deploy
      summary: Deploy a precompiled smart contract.
      parameters: []
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/DeployDto'
      responses:
        '201':
          description: ''
      tags: *ref_0
    patch:
      operationId: ContractsController_updateContract
      summary: Verify and update contract in DB from already deployed contract.
      parameters: []
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/UpdateContractDto'
      responses:
        '200':
          description: ''
      tags: *ref_0
    get:
      operationId: ContractsController_getAll
      summary: Get list of all stored smart contracts.
      parameters:
        - name: pageSize
          required: false
          in: query
          description: Number of elements per page.
          example: 30
          schema:
            default: 10
            type: number
        - name: pageIndex
          required: false
          in: query
          description: Index of the page.
          example: 3
          schema:
            default: 0
            type: number
      responses:
        '200':
          description: ''
      tags: *ref_0
  /contracts/{id}:
    get:
      operationId: ContractsController_getOne
      summary: Get a single smart contract.
      parameters:
        - name: id
```

```
          required: true
          in: path
          description: Contract id.
          example: 3
          schema:
            type: number
      responses:
        '200':
          description: ''
      tags: *ref_0
  /transactions:
    post:
      operationId: TransactionsController_send
      summary: Send Ethers to address.
      parameters: []
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/SendDto'
      responses:
        '201':
          description: ''
      tags: &ref_1
        - Transactions
    get:
      operationId: TransactionsController_getAll
      summary: Get list of all stored transactions.
      parameters:
        - name: pageSize
          required: false
          in: query
          description: Number of elements per page.
          example: 30
          schema:
            default: 10
            type: number
        - name: pageIndex
          required: false
          in: query
          description: Index of the page.
          example: 3
          schema:
            default: 0
            type: number
      responses:
        '200':
          description: ''
      tags: *ref_1
    patch:
      operationId: TransactionsController_updateTransaction
      summary: Update transaction in DB from an already mined transaction.
      parameters: []
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: ’#/components/schemas/UpdateTransactionDto’
      responses:
        ’200’:
          description: ’’
      tags: *ref_1
  /transactions/{txHash}:
    get:
      operationId: TransactionsController_getOne
      summary: Get a single transaction.
      parameters:
        - name: txHash
          required: true
          in: path
          description: Hash of the transaction.
          example: ’0x9df7ba8ae253f458defb309e55c6f374c31c504f1e19f073a913ec8a87fa717d’
          schema:
            type: string
      responses:
        ’200’:
          description: ’’
      tags: *ref_1
  /transactions/balance/{addr}:
    get:
      operationId: TransactionsController_getBalance
      summary: Get balance, in Ethers, of an address.
      parameters:
        - name: addr
          required: true
          in: path
          description: Wallet address.
          example: ’0xA46B8f9D99446AF2E0d536B4A89C17Cb62A6ad8A’
          schema:
            type: string
      responses:
        ’200’:
          description: ’’
      tags: *ref_1
  /transactions/sign:
    post:
      operationId: TransactionsController_sign
      summary: Sign a transaction.
      parameters: []
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: ’#/components/schemas/SendNewDto’
      responses:
        ’201’:
          description: ’’
      tags: *ref_1
info:
  title: Ethereum-NestJS-Middleware
  description: NestJS API for interacting with the Ethereum blockchain.
  version: 0.0.3
  contact: {}
tags: []
servers: []
components:
  schemas:
    MnemonicDto:
      type: object
      properties:
        mnemonic:
          type: string
          description: Seed phrase.
          example: >-
            twin alley estate barrel bicycle crawl ocean better blanket exotic
            tone bid
        password:
          type: string
          description: Password used to protect the HD Wallet.
          example: p4ssw0rd
        path:
          type: string
          description: HD path for the account.
          example: m/44’/60’/0’/0/2
          default: m/44’/60’/0’/0/0
      required:
        - mnemonic
    GasSettingsDto:
      type: object
      properties:
        gasLimit:
          format: int64
          type: integer
          example: 21000
        gasPrice:
          format: int64
          type: integer
          example: 1122646121
        maxFeePerGas:
          format: int64
          type: integer
        maxPriorityFeePerGas:
          format: int64
          type: integer
      required:
        - gasLimit
        - gasPrice
        - maxFeePerGas
        - maxPriorityFeePerGas
    UpdateFunctionBodyDto:
      type: object
      properties:
        mnemonic:
          description: If provided, replaces internal private key.
          allOf:
            - $ref: '#/components/schemas/MnemonicDto'
        func:
          type: string
          description: Function name in smart contract.
          example: get
        args:
          description: List of arguments of the function.
          example:
            - a
            - b
            - c
          type: array
          items:
            type: string
        gasSettings:
          description: Gas settings for the transaction.
          allOf:
            - $ref: '#/components/schemas/GasSettingsDto'
        quant:
          type: number
          description: Send Ethers if payable.
          example: 4.2
      required:
        - func
        - args
    DeployDto:
      type: object
      properties:
        mnemonic:
          description: If provided, replaces internal private key.
          allOf:
            - $ref: '#/components/schemas/MnemonicDto'
        abi:
          type: string
          description: JSON-formatted ABI of compiled smart contract.
          example: >-
            [{"constant":false,"inputs":[],"name":"pay","outputs":[],"payable":true,"stateMutability":"payable","type":"function"},{"constant":false,"inputs":[],"name":"set","outputs":[],"payable":false,"stateMutability":"nonpayable","type":"function"},{"constant":true,"inputs":[],"name":"get","outputs":[{"internalType":"uint256","name":"","type":"uint256"}],"payable":false,"stateMutability":"view","type":"function"}]
        bytecode:
          type: string
          description: Hex-formatted bytecode of compiled smart contract.
          example: >-
            6080604052348015610010576000080
   fd5b5060c48061001f6000396000f3fe6080604052600436106030576000356 0e01c80631b9265b8
        source:
          type: string
          description: Minified source code of the smart contract.
          example: >-
            pragma solidity ^0.5.0; contract SimpleStorage { uint x;
   function
            set() public { x = 333; } function get() public view returns
    (uint)
            { return x; } function pay() public payable {} }
        fileName:
          type: string
          description: File name used to compile the contract.
          example: contract-2c390734c4.sol
        compilerVersion:
          type: string
          description: Compiler version used to compile the contract.
          example: 0.5.14
          default: latest
        gasSettings:
          description: Gas settings for the transaction.
          allOf:
            - $ref: '#/components/schemas/GasSettingsDto'
      required:
        - abi
        - bytecode
    UpdateContractDto:
      type: object
      properties:
        tx:
          type: string
          description: Hash of the transaction that deployed the
   contract.
          example: '0
   x0ce48a5a0779e86dcdfd546098de79e2ba4e46bca478461f6c6f9a9565c55d93'
        abi:
          type: string
          description: JSON-formatted ABI of compiled smart contract.
          example: >-
            [{"constant":false,"inputs":[],"name":"pay","outputs":[],"
   payable":true,"stateMutability":"payable","type":"function"},{"
   constant":false,"inputs":[],"name":"set","outputs":[],"payable":false
   ,"stateMutability":"nonpayable","type":"function"},{"constant":true,"
   inputs":[],"name":"get","outputs":[{"internalType":"uint256","name":"
   ","type":"uint256"}],"payable":false,"stateMutability":"view","type":
   "function"}]
        source:
          type: string
          description: Minified source code of the smart contract.
          example: >-
            pragma solidity ^0.5.0; contract SimpleStorage { uint x;
   function
```

```
            set() public { x = 333; } function get() public view returns (uint)
            { return x; } function pay() public payable {} }
        fileName:
          type: string
          description: File name used to compile the contract.
          example: contract-2c390734c4.sol
        compilerVersion:
          type: string
          description: Compiler version used to compile the contract.
          example: 0.5.14
          default: latest
      required:
        - tx
        - abi
        - source
        - fileName
    SendNewDto:
      type: object
      properties:
        mnemonic:
          description: If provided, replaces internal private key.
          allOf:
            - $ref: '#/components/schemas/MnemonicDto'
        to:
          type: string
          description: Address to send to.
          example: '0x1b973BC2cb3e4413a6B3E302357Fe9d1D586028e'
        quant:
          type: number
          description: Quantity (in Ethers).
          example: 4.3
        gasSettings:
          description: Gas settings for the transaction.
          allOf:
            - $ref: '#/components/schemas/GasSettingsDto'
      required:
        - to
        - quant
    SendRawDto:
      type: object
      properties:
        tx:
          type: string
          description: Raw transaction in hex format.
          example: >-
            0x02f87482053914843b9aca0084443bdd24825208941b973bc2cb3e4413a6b3e302357fe9d1d5860
      required:
        - tx
    SendDto:
      type: object
      properties:
        new:
          description: Settings for a new transaction.
          allOf:
            - $ref: '#/components/schemas/SendNewDto'
        raw:
          description: Settings for a raw transaction.
          allOf:
            - $ref: '#/components/schemas/SendRawDto'
    UpdateTransactionDto:
      type: object
      properties:
        txHash:
          type: string
          description: Hash of the transaction.
          example: '0xa83dc996c182595ee819868a83e0f5b39c3088f04051494dba9fa784f4430a01'
      required:
        - txHash
```