Faculty of Sciences and Tecnology

Department of Informatics Engineering

Validation of Smart Contracts Through Automated Tooling

Tomás Morgado de Carvalho Conceição

Internship Report in the context of the Master in Informatics Engineering, Specialization in Software Engineering advised by Professor Raul Barbosa and Eng. João Barbosa and presented to the
Department of Informatics Engineering / Faculty of Sciences and Technology

January 2019

UNIVERSIDADE Ð
COIMBRA

Acknowledgements

I would first like to thank Professor Raul Barbosa, my advisor at the University of Coimbra, for, with his valuable experience and insights, guiding me through this process and always challenging me to do better.

I would like to thank my advisor, João Barbosa, not only for his technical contributions to this dissertation but for always ensuring that I had my priorities straight and was managing my time properly. Thank you for pushing me to always make healthy choices regarding my work-life balance.

I want to thank the members of my jury, Professor Fernando José Barros and Professor César Alexandre Domingues Teixeira, for their input and suggestions for improvement, and their constructive criticism.

I would like to thank Whitesmith and Blocksmith, and in particular Gonçalo Louzada, Maria João Ferreira and Rafael Jegundo, for accepting me on this internship, providing me with all the tools and conditions to achieve my goals and for inviting me to take part on other projects during this year.

I would also like to thank my co-workers, who, since the first day and without exception, made me feel welcome and at home. I have learned something new with every single one of you.

Finally, I would like to thank my family and friends for supporting me and encouraging me relentlessly throughout all of my academic and non-academic life.

Abstract

Smart contracts are computer programs which use blockchain technology to allow users to exchange digital assets without the need for a middle-man. NEO is a recent blockchain platform, which presents programmers with the possibility of developing smart contracts in high-level programming languages. Due to blockchain properties, smart contract code cannot be changed after being deployed. It is then crucial for developers to thoroughly test their smart contracts before using them in a production environment. This dissertation, a product of the intern's internship at Whitesmith, presents the different phases of the development of a testing tool for NEO smart contracts. This tool attempts to fill the gap between the state of the art of NEO and Ethereum developer tools, two promising blockchain technologies which are presented in this report. The study of the state of the art of development tools for these technologies shows that, although NEO has development tools, such as NeoCompiler Eco, which assists in the development of smart contracts, these do not offer any testing features. Taking advantage of the existing NEO tool NeoCompilerEco, a test automation tool was developed, which enables users to create, import, save and run test cases and test suites on a NEO private network, using only a browser. This tool, which assesses tests according to events listened in the NEO blockchain, is the result of the requests made to the intern by developers at Blocksmith, a spin-off startup from Whitesmith, and is now in use by these developers.

Keywords

Automation, Blockchain, NEO, Smart Contract, Testing

Resumo

Os *smart contracts* são programas de computador que usam a tecnologia *blockchain* para permitir que os seus utilizadores troquem bens digitais sem a necessidade de um intermediário. NEO é uma recente plataforma de *blockchain* que oferece aos programadores a possibilidade de desenvolver *smart contracts* em linguagens de programação de alto nível. Devido às propriedades da *blockchain*, o código dos *smart contracts* não pode ser alterado após ser distribuido na *blockchain*. É portanto crucial que os programadores testem cuidadosamente os seus *smart contracts* antes de os usarem em ambiente de produção. Esta dissertação, o produto do estágio do seu autor na Whitesmith, apresenta as diferentes fases do desenvolvimento de uma ferramenta de testes para *smart contracts* de NEO. Esta ferramenta tenta colmatar a distância entre o estado da arte das ferramentas de desenvolvimento de NEO e de Ethereum, duas tecnologias de *blockchain* promissoras, apresentadas neste relatório. O estudo do estado da arte das ferramentas de desenvolvimento para estas tecnologias mostra que, apesar de NEO ter ferramentas de desenvolvimentos tais como o NeoCompiler Eco que assistem no desenvolvimento de *smart contracts*, estas não oferecem quaisquer funcionalidades de teste. Tomando partido de uma ferramenta já existente para NEO, o NeoCompiler Eco, foi desenvolvida uma ferramenta de automação de testes que permite aos seus utilizadores criar, importar, guardar e correr casos de teste e suites de teste numa rede privada de NEO, utilizando apenas um browser. Esta ferramenta, que avalia os casos de teste consoante eventos ouvidos na *blockchain* de NEO, é o resultado dos requisitos feitos ao estagiário pelos programadores da Blocksmith, uma *startup spin-off* da Whitesmith, e é agora usada por estes programadores.

Palavras-Chave

Automação, *Blockchain*, NEO, *Smart Contracts*, Testes

Contents

Acronyms

AJAX Asynchronous JavaScript And XML. 65, 69

API Application Programming Interface. 13, 19, 44, 49, 57, 61, 65, 66, 68, 69, 73, 74, 76, 77

CLI Command-Line Interface. 13, 14, 17, 22, 62, 65

CoZ City of Zion - an independent group of open source developers formed to support the NEO core and ecosystem. xiii, 15–17, 32

CRL Certificate Revocation List. 10

DApp Decentralized Application. 1, 33

dBFt Delegated Byzantine Fault Tolerance. 11

DOM Document Object Model. 65, 69

DoS Denial of Service. 9

GUI Graphical User Interface. xiii, 3, 13–17, 19, 22

IDE Integrated Development Environment. xiii, 14, 15, 19

MVP Minimum Viable Product. 1

ORM Object Relational Mapper. 50, 53, 57, 63

OS Operating System. 19, 21

PKI Public Key Infrastructure. 10

PoW Proof of Work. 6, 7, 10

RPC Remote Procedure Call. xiv, 13, 36, 42, 44, 57, 65–67

SDK Software Development Kit. 24, 33

SPA Single Page Application. 44, 65, 76

UI User Interface. 3

URI Uniform Resource Identifier. 61

VCS Version Control System. 26

VM Virtual Machine. xiii, 5, 8, 12, 17, 18

List of Figures

List of Tables

Chapter 1

Introduction

During their study of the NEO blockchain, both Blocksmith and Whitesmith developers found the state of the art of NEO development tools immature, especially when trying to test their smart contracts and Decentralized Applications (DApps). Running a node in an official NEO test network is also a cumbersome task. Test currency must be requested to the test network owners, a process that can take a few days, and the node must be fully synchronised with the test network, which can take a few hours to accomplish as the client must download and synchronise the full blockchain. While running a local network can solve these problems, as test currency can be generated by the network owner and the small size of a local blockchain allows for a fast synchronization, sharing a local network between a team of developers requires a few tweaks and every developer still has to have a copy of the local network fully synchronized.

Furthermore, after deploying a smart contract, it is not trivial for a developer to know whether the smart contract is performing as expected, and, when noticed that the smart contract is not performing as expected, it is not always easy to understand why. After fixing a problem in a smart contract and when attempting to test it, the developer must deploy the new version of the smart contract to the blockchain, manually invoke the smart contract, call every method on the smart contract, compare the result of the invocation with the expected result and verify whether the problem was fixed or not. This is a very time-consuming task that not only takes valuable time away from development but it also leaves room for error, as developers may eventually forget to test a particular scenario.

Most startups, in order to respond to competitive pressure, rely on Agile development methodologies to be able to validate and deliver Minimum Viable Products (MVPs) as fast as possible[20]. Fast iterations through different product development phases, from idea to deployment, ensure that stakeholders can input their feedback more often, which in turn guarantees that a product succeeds or fails quickly and inexpensively. This ”fail fast, fail often, fail cheap” doctrine is often the difference between succeeding or failing as a startup[21]. To improve product quality, predictability, development, and shipping times, processes, tools, and frameworks were created for most mature technologies and are abundantly used by developers.

For startups, such as Blocksmith, to compete in the fast-paced and ever-changing environment that is blockchain development and consulting, it is essential that their developers have access to tools which enables them to move fast and deliver robust products and solutions in different blockchain technologies.

1.1 Context and Motivation

This report was written as part of the intern's curricular internship, during his Masters in Software Engineering, by the Department of Informatics Engineering (DEI) of the University of Coimbra, Portugal. The internship took place at Whitesmith, LDA.

Headquartered at Instituto Pedro Nunes (IPN) in Coimbra, Portugal, Whitesmith is a product studio and software consultant. Recognising blockchain as a promising technology, Whitesmith launched in 2018 its first spinoff, Blocksmith, a blockchain development agency based in London, United Kingdom. Whitesmith takes pride in iterating from idea to product in record times, a track record that it wants Blocksmith to inherit. To do so, their developers must be able to easily and quickly setup development environments where they can develop, deploy, test and validate their prototypes.

1.1.1 Smart-Contract Development Process

As explained and exemplified by K. Delmolino et al. with a "Rock-Paper-Scissors" smart contract example[17], even simple smart contracts are prone to bugs.

The development of a smart contract generally encompasses the following steps:

1. The developer writes the code for the smart contract, in a language accepted by a compiler for the blockchain to where the smart contract is going to be deployed;

2. The developer compiles the code he has written in the previous step to the machine code accepted by the blockchain;

3. The developer deploys the machine code to a private (local) network in order to assess its behaviour by calling the smart contract different methods;

4. The developer deploys the machine code to a test network, a network that uses play currency and emulates the main blockchain network, and continues assessing the smart contract behaviour;

5. The developer finally deploys the machine code to the main network of the blockchain, a network that uses real currency;

It is important to note that steps 1 to 3 are usually repeated while the developer changes his smart contract code to fix bugs or add new features. The developer then moves to step 4 and deploys the smart contract in the test network in order to understand how it is going to behave in a bigger network, returning to step 1 to make any necessary changes. Finally, after being reasonably confident that the smart contract is working as intended, the developer moves to step 5 and deploys the smart contract on the main network. Due to blockchain properties, after a smart contract is deployed in a main network, it is not possible to patch it. Smart contracts need to be thoroughly tested before deployment as failing to do so can result in hidden bugs and vulnerabilities causing the loss of assets. An example was when a bug in an Ethereum smart contract was exploited by attackers, leading to the loss of more than 50,000,000 dollars worth of Ether on the most significant attack of the kind and forcing a fork on the Ethereum network[4]. Exploits and attacks such as this one are not uncommon, preventing developers from being comfortable when developing smart contracts that deal with users funds.

1.1.2 NEO

NEO is a blockchain platform that supports smart contracts and which competes directly with the Ethereum blockchain[13]. Due to different blockchain platforms being best for different use cases[42], and in order to better serve its clients, Blocksmith started investigating the NEO blockchain, considering it to be a potential platform to be used for future projects.

When searching for tools that could help them increase their productivity and deliver blockchain products faster, Whitesmith and Blocksmith developers realised that such tools were lacking for the NEO blockchain. Realising that contributing with a new tool for this blockchain could improve not only its developers' productivity but also advertise its presence in the blockchain community, Blocksmith decided to study the possibility to develop an open-source tool for testing smart contracts for the NEO blockchain. As such, the intern was tasked with studying the current state of the art on smart contract development tools, with emphasis on the NEO blockchain. The intern was also asked to study and develop a solution that allowed for a developer or a team of developers to easily test and validate NEO smart contracts, if possible without having the burden of having to set up a blockchain. To automate this validation, the developers should be able to set a test suite which would be run against a deployed smart contract.

1.2 Objectives and Scope

In this internship, the intern aims to develop a web application tool to support developers to implement and test NEO smart contracts, providing the necessary infrastructure. This solution should work as a test harness, or test automation framework which, according to *"Introduction to Software Testing"*, should support the ability to evaluate expected test results through assertions[1], test suites to organize and run multiple test cases (as defined in Section 2.6 - Software Testing) and a Graphical User Interface (GUI)[33].

It should also support a NEO blockchain network, where the smart contracts will be deployed and where the tests will be executed. For the purpose of the internship, the developed solution interface should be able to give the users the possibility of creating unit tests to test their smart contracts. The tool should handle the smart contract as a black box, giving the user the possibility to set an input for each test case and to determine the expected output of the smart contract execution. When a test case or test suite ends its execution, the tool should display a test report, which should also reference performance information, such as the amount of NeoGas spent to run the test. The tool should also allow its users to save test cases and test suites and enable them to use the same test cases or test suites in different contracts.

At the end of the internship, a report should be produced which should document the study of state of the art, the requirement's gathering, the architecture used and detail the process of developing and testing the chosen solution. The design of the User Interface (UI) of the tool to be developed is out of the scope of this internship.

[1] Assertion - A boolean expression which should evaluate to true unless there is a defect in the program.

1.3 Proposed Solution

For the presented problem, the proposed solution is a web application for smart contract deployment and testing. This web application features a testing tool, where users can define which operations they want the smart contract to perform and what is the expected outcome of said operations. These operations are invoked in the smart contract, and a custom NEO node is listening to the resulting events. After the execution of the tests end, the user is presented with the test results and with data on the contract execution, such as NeoGas spent. To solve the problem of the developer having to set up a development environment with a private blockchain every time he needs to develop or test a smart contract, it is proposed that the developed tool shall be integrated with an existing open-source tool called NeoCompiler Eco (see 2.5.6). With this solution, a private network is running on a back-end server, with the user just having to access a web client to deploy his smart contract on the platform.

1.4 Report structure

This report follows the following structure:

- Chapter 1 - In this chapter, an introduction to the report is made. This introduction starts by describing the problem. It then introduces the context of the internship, its goals and scope and the proposed solution to the presented problem.
- Chapter 2 - In this chapter, the study of state of the art is presented. An introduction to blockchain and Bitcoin is made, followed by the introduction of smart contracts and Ethereum. The NEO platform is then presented, and a description and comparison of several Ethereum and NEO development tools follow. Finally, some Software Testing terms are presented to the reader.
- Chapter 3 - In this chapter, the development methodologies and tools used in the management of this project are presented, as well as an analysis and identification of the project risks and a mitigation plan for each of these risks. The planning of this project is also described.
- Chapter 4 - In this chapter, the project stakeholders and actors are presented, as well as the requirements gathered from these entities. These requirements are split into functional requirements and non-functional requirements.
- Chapter 5 - In this chapter, the system architecture is discussed. This architecture is presented in different diagrams which provide an overview over different levels of the system.
- Chapter 6 - In this chapter, the implementation of the different components of the tool is described.
- Chapter 7 - In this chapter, the different tests made for the different system components are enumerated.
- Chapter 8 - In this chapter, a conclusion for the report is presented.

Chapter 2

State of the art

In today's society, and with the widespread use of electronic transactions, almost every transaction is mediated by a central authority such as financial institutions or the government. This gives every party involved in the transaction a sense of security. This model works as long as each side trusts the central authority to mediate the trade and enforce penalties in the event that one of the parties breaches his side of the agreement. However, recent events such as the financial crisis of 2007-2008, have shown that even the most significant financial institutions can fail, leaving numerous people with the conviction that the collapse of the world banking system is not an implausible scenario. There are also countries where these central authorities cannot be trusted to mediate agreements or to enforce penalties, either because they are corrupt or because they lack the necessary infrastructures to keep transactions and ownership records safe[38].

With the advent of the blockchain, we are presented with a number of technologies that, through a set of cryptographic and mathematical properties, guarantee that its users can make electronic transactions in a transparent, trust-less and irreversible manner, without having to trust a central authority. In 2009, Bitcoin was the first fully decentralised technology to solve the double-spending problem and fully implement the concept of blockchain[9]. In 2015, a new implementation of the blockchain was launched, called Ethereum[6]. Ethereum features a Turing-complete Virtual Machine (VM) with the capability of executing scripts, the Ethereum Virtual Machine[18]. Suddenly, developers were given the tools to be able to deploy a whole new class of decentralised applications powered by smart contracts running on the Ethereum blockchain. In 2016 another implementation of the Blockchain, NEO (formerly known as Antshares), a Chinese cryptocurrency also featuring a Touring-complete VM (neoVM), was launched[27].

This chapter aims to acquaint the reader with the concept of Blockchain, as well as with the two most notorious and famous blockchain implementations, Bitcoin and Ethereum. The concept of "Smart Contracts" is introduced, followed by the study of the NEO blockchain. An analysis of the current state of Ethereum and NEO smart contract development tools is then presented, and a final comparison between these tools is made.

Finally, some concepts of software testing relevant for this report are reviewed.

2.1 Blockchain and Bitcoin

The first implementation of a blockchain was described in 2008 and implemented in 2009 by a person, or a group of people, under the pseudonym of Satoshi Nakamoto, as the backbone of the Bitcoin cryptocurrency[31]. In a simplistic description, a blockchain can be seen as an immutable and ever-growing list of blocks where each new block is appended to the top of the blockchain. These blocks are connected between them and secured using cryptographic properties.

In most blockchain implementations, these blocks are constituted by a block header and a block body. In the Bitcoin implementation, the body contains the digital transactions to be validated and recorded on the blockchain and its count, whereas the block header includes the version of the block, a nonce, a cryptographic hash of the previous block, a Merkle-tree root, the current difficulty and a timestamp, as illustrated in Figure 2.1.

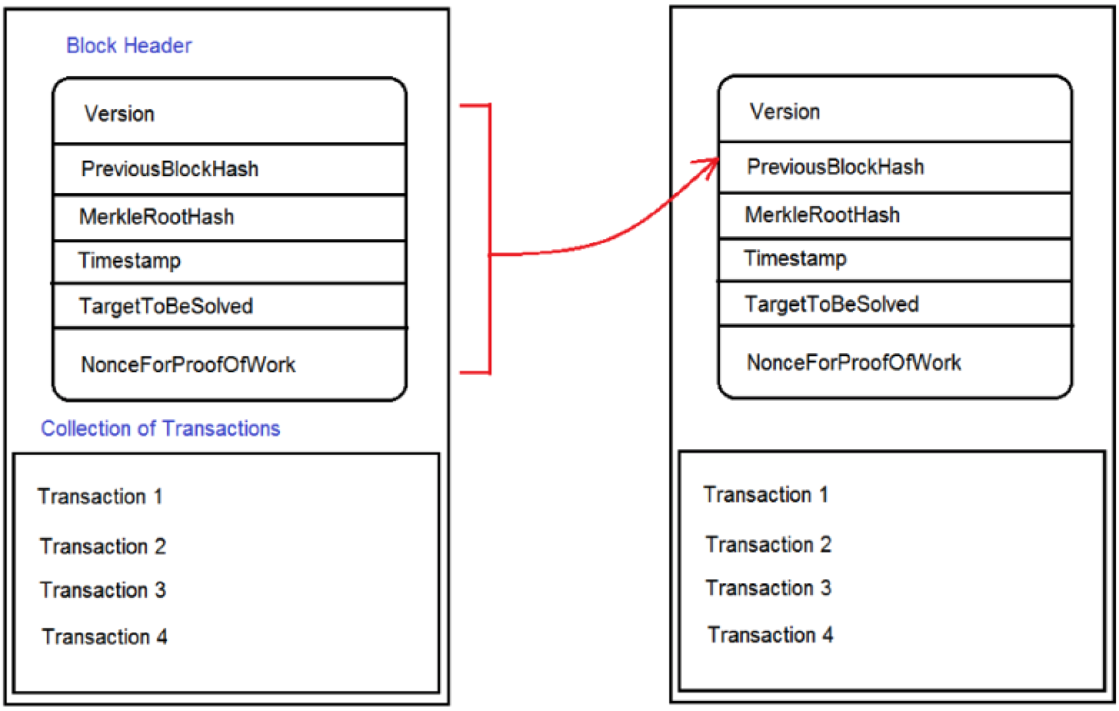


Figure 2.1: Representation of a chain of two Bitcoin blocks. Taken, with owner consent, from `http://blog.brakmic.com/bitcoin-internals-part-1/`.

Blockchains are usually used in peer-to-peer networks, where every node of the network holds its copy of the blockchain. In the case where everyone can access the blockchain (i.e., be a node of the network), the term public blockchain is used, whereas a blockchain where only selected nodes can access its data is called a private blockchain[34]. In private blockchains, reaching a consensus in which transactions to validate is often easy, as a private organisations usually runs the validator nodes without interest in colluding against the blockchain. However, in public blockchains, it is necessary to have a consensus mechanism that prevents nodes from attempting to force a different version of the blockchain to other nodes.

In Bitcoin whitepaper, Nakamoto proposed a consensus mechanism based on a Proof of Work (PoW) protocol[31]. In Nakamoto's PoW implementation, validator nodes, or miners, actively listen to the network, waiting for transactions to be broadcasted by other nodes. These validators can pick the transactions they want to include in the block they will be mining, often opting to prioritise transactions that pay higher transaction fees. A miner can include any amount of transactions they wish in the block they are mining, providing that the block size does not surpass the maximum allowed size of 1 megabyte. To mine a block, the miner increments a nonce attempting to find a value that, hashed with the rest of the content of the block, results in a hash that starts with a pre-determined number of zeros. The number of zeros required in the resulting hash is used to set the difficulty of finding this nonce (the higher the number of zeros required, the hardest it is to generate a hash that complies with this requirement). The PoW difficulty is increased or decreased automatically to compensate for increasing hardware speed and varying interest in running nodes over time and aims to generate a new block every ten minutes. Due to the cryptographic properties of one-way hash functions (in Bitcoin, SHA256 is used), it is hard for a miner to find a nonce that generates a hash that obeys the requirements. However, once the nonce is found it is easy for other miners to verify that it is indeed a correct solution to the problem.

This difficulty in finding the solution is the proof that work has been done on the block and makes it that if a node wants to compromise the integrity of older blocks, it will have to mine every block since the block he wants to change up until the latest mined block. This guarantees that, unless the miner has more than 51% of the CPU power of the miners' network, he will never be able to catch up with the current state of the blockchain, as the nodes will always pick the longest available version of the blockchain as the true blockchain[8].

When a miner finds a solution, he will add a new transaction, called a *generation transaction*, to the beginning of the block issuing a reward of a pre-determined amount to himself. This reward is a special kind of transaction, as, contrary to the other bitcoin transactions, does not contain a *from* address. As such, this transaction will generate new bitcoins into the system. This reward started in 50 bitcoins, with this amount being halved every 210,000 blocks. Eventually, no bitcoins will ever be generated, making it that the network has a hard cap of 21 million bitcoins[25]. In this transaction, the miner also receives every transaction fees existing in the mined block.

The miner then broadcasts the solved block to the entire network and other miners will validate the transactions and check whether the proposed nonce fits the solution. If a miner accepts the block, it will append the block to his blockchain, discard the block he was working on and start generating a new block to mine. If he does not find the block to be valid, he will discard the received block and will keep trying to find a nonce that fits the requirements. Eventually, other blocks will be mined on top of this block, with every block mined on top being called a *confirmation*. If this block is part of the longest version of the blockchain, eventually other branches created by other blocks being mined concurrently will stop receiving work. Usually, users and traders demand more than six confirmations before accepting the transaction as valid[7].

2.2 Smart Contracts

The term "Smart Contract" was coined in 1994 by Nick Szabo, which defined a smart contract as "a computerized transaction protocol that executes the terms of a contract"[39]. Whereas usually, the judicial system or another arbitration method enforces a traditional contract, with smart contracts a program enforces the contract built into the code. Szabo, in "Smart Contracts: Building Blocks for Digital Markets", uses a vending machine as the example of a primitive smart contract ancestor: The vending machine takes currency from its user and, according to a previous agreement on the product price, dispenses product and change fairly[40]. With the advent of the blockchain and Bitcoin, developers were provided with the means of running simple scripts on top of the blockchain, using a Touring-incomplete scripting language. Later, platforms such as Ethereum (see section 2.3) and NEO (see section 2.4), provided developers with Turing-complete virtual machines, where more complex and sophisticated programs could be run. These programs, or smart contracts, are run by every node of the network they belong to every time the node attempts to validate a transaction that inputs a determined value to the smart contract method. As such, smart contracts need to be deterministic, otherwise, each node that executes the contract method to validate the transaction would end with different results and no consensus would be possible[12]. These properties make it possible for programmers to develop smart contracts in a decentralised network, where the oversight of a third-party or centralised authority is not needed and give users the ability to take part on contracts which are auditable, verifiable and deterministic. These contracts can be used to manage assets, as long as these assets that are, themselves, embedded on the blockchain where the smart contract is running.

2.3 Ethereum

In 2013, Vitalik Buterin proposed a new blockchain platform, called Ethereum[11]. Vitalik and his team started publicly developing Ethereum in early 2014[10], and in 2015 Ethereum was officially launched[6]. Ethereum includes a Turing-complete VM, the EthereumVM, which provides a runtime environment for smart contracts in the Ethereum blockchain. In Ethereum, smart contracts can be written in a set of high-level languages which are converted to EthereumVM bytecode and deployed on the Ethereum blockchain for execution. Of these, Solidity, a language similar to C and JavaScript, is the most popular language for Ethereum smart contract development.

Ethereum has a native token, called Ether. Ether is used to pay for transaction fees, working as the fuel of the Ethereum network. Ether can be subdivided down to 10^{-18} Ether, which is also called a "wei"[14]. To hold Ether and interact with the blockchain, a user must create an "account". Each account has a 20 bytes address, which identifies the account, and four fields:

- A nonce, which is a counter used to guarantee that each transaction can only be processed once;
- The account's current Ether balance;
- The account's contract code, if existent;
- The account's key-value storage, which is empty by default;

In general, there are two types of accounts: externally owned accounts[41], belonging to a user of the Ethereum blockchain and which are controlled by private keys, and contract accounts, controlled by their contract code. An externally owned account has no contract code, with the user being able to send messages by creating and signing a transaction. On a contract account, every time it receives a message, its code activates, allowing it to read and write to storage and send other messages, or even create other contracts[41].

To be sent, a message needs to be stored inside a transaction. Transactions, in Ethereum, contain:

- The address of the recipient of the message;
- The sender signature;
- The amount of ether (can be zero) to transfer from the sender to the recipient;
- A *STARTGAS* value, which represents the maximum gas to be spent on the execution of the message;
- A *GASPRICE* value, which represents the value that the sender is willing to pay for each gas consumed, and which is given in Wei;
- An optional data field;

Gas can be seen as the effort required to execute a given transaction. Most of the computational steps taken by the EthereumVM when running a contract cost 1 gas, with some of the most computationally expensive steps costing more. A fee of 5 gas must also be paid for every byte in the transaction data. If a transaction runs out of gas during execution, all the state changes are reverted, but the spent gas is not reimbursed. If a transaction ends its execution with success, the transaction will change the state to a new state, as illustrated in Figure 2.2.

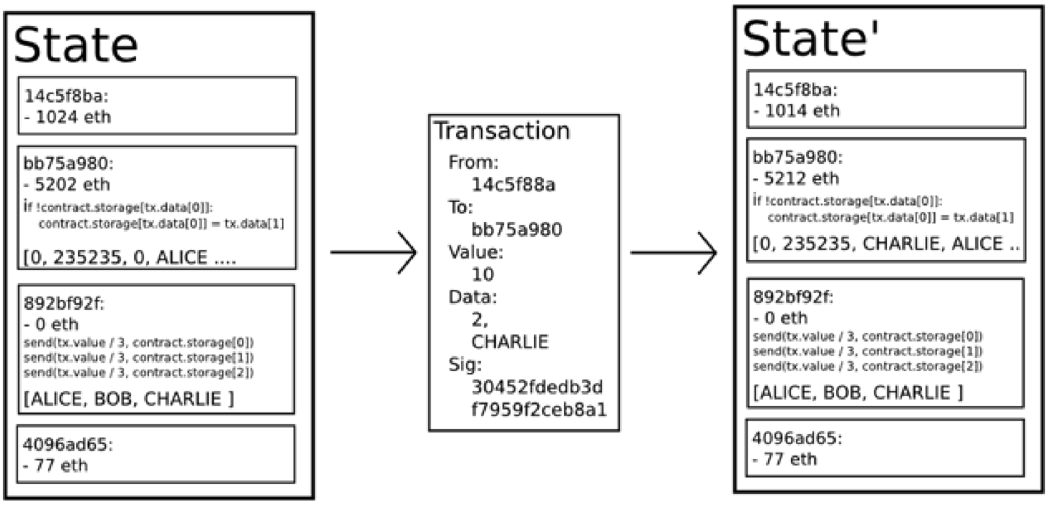

State
14c5f8ba:
- 1024 eth
bb75a980:
- 5202 eth
if !contract.storage[tx.data[0]]:
contract.storage[tx.data[0]] = tx.data[1]
[0, 235235, 0, ALICE ....
892bf92f:
- 0 eth
send(tx.value / 3, contract.storage[0])
send(tx.value / 3, contract.storage[1])
send(tx.value / 3, contract.storage[2])
[ALICE, BOB, CHARLIE]
4096ad65:
- 77 eth
Transaction
From:
14c5f88a
To:
bb75a980
Value:
10
Data:
2,
CHARLIE
Sig:
30452fdedb3d
f7959f2ceb8a1
State'
14c5f8ba:
- 1014 eth
bb75a980:
- 5212 eth
if !contract.storage[tx.data[0]]:
contract.storage[tx.data[0]] = tx.data[1]
[0, 235235, CHARLIE, ALICE ..
892bf92f:
- 0 eth
send(tx.value / 3, contract.storage[0])
send(tx.value / 3, contract.storage[1])
send(tx.value / 3, contract.storage[2])
[ALICE, BOB, CHARLIE]
4096ad65:
- 77 eth

Figure 2.2: Ethereum state transiction function. Taken from *Ethereum Whitepaper: A Next-Generation Smart Contract and Decentralized Application Platform*[11].

If the transaction concludes without using all the available gas, the spare gas is sent back to the sender. This prevents a transaction execution from being locked in a loop, while also preventing Denial of Service (DoS) attacks on the network by making the attackers pay for each computational step taken. The *GASPRICE* also serves for senders to set a priority to transactions, as miners will try to process first the transactions that pay more Ether for each gas used. The transaction fee is calculated by multiplying the *GASPRICE* by the *STARTGAS* and is subtracted from the sender's account balance. If the sender does not have enough balance to pay for the transaction, an error will be raised and the transaction aborted.

Despite some differences, the Ethereum blockchain is in many ways similar to the Bitcoin blockchain. Of this differences, the major one, with regards to the blockchain architecture, is that Ethereum blocks contain a copy of the transaction list and its current state, while Bitcoin only stores a copy of the transaction list, as shown in 2.3. Aside from this difference, both the block number and the difficulty are stored in the block.

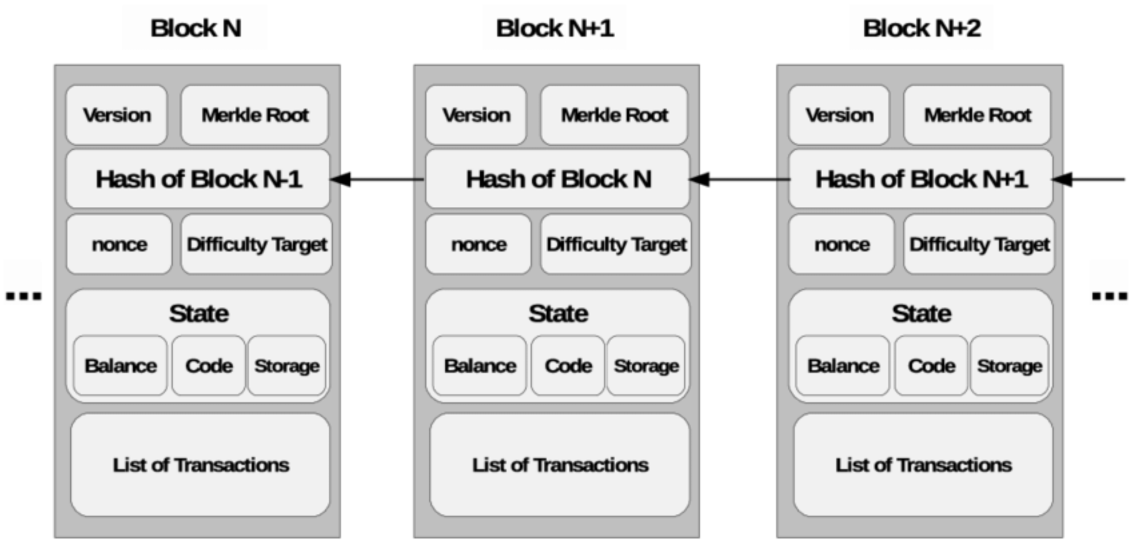


Figure 2.3: A representation of an Ethereum block. Taken from *IoT Security: Review, Blockchain Solutions, and Open Challenges*[36].

The consensus algorithm used on the Ethereum blockchain is the same PoW algorithm used in the Bitcoin blockchain, with the difference that Ethereum attempts to generate a new block every 15 seconds which allows a higher number of transactions being validated per second (roughly a throughput of 15 transactions per second), compared to the 10 minutes that Bitcoin blockchain takes to generate a new block.

2.4 NEO

Formerly known as Antshares, NEO was founded in 2014 by Da Hongfei and had its platform deployed online in 2016. In the NEO whitepaper[24], Da Hongfei states that NEO main objective is to use blockchain technology and digital identity to achieve what Da calls a "Smart Economy". In the whitepaper, "Smart Economy" is depicted as using smart contracts to manage digital assets, which belong to digital entities automatedly. To build this "Smart Economy", NEO promises to establish a sophisticated digital identity system, based in the Public Key Infrastructure (PKI) X.509 Standard. To issue and validate a digital identity, NEO claims that a set of authentication methods will be used, such as "the use of facial features, fingerprint, voice, SMS and other multi-factor authentication methods", which will be managed by an X.509 Certificate Revocation List (CRL) functioning on the blockchain.

2.4.1 NEO Economic Model

NEO has two native tokens, a management token called NEO and a fuel token called NeoGas, which is usually referred to as GAS. The NEO token functions similarly to having a share of a company. Users that hold NEO tokens can vote on managerial decisions, such as choosing bookkeepers (consensus nodes). The minimum unity of NEO is 1, and a token cannot be subdivided. A total supply of 100 million NEO tokens was generated at the beginning of the platform, with half of it belonging to the NEO Council and the rest having been distributed to the platforms investors in a crowdfunding phase. NeoGas is used to pay for smart contract deployment and further operations, preventing the abuse of the network resources, such as DOS attacks on the network. NeoGas can also be used to pay transactions fees, although, at the moment of the writing of this document, transactions are free. Transactions fees will enable higher paying transactions to be prioritised as well as will serve as an economic incentive for bookkeepers to exist and honestly validate blocks. The minimum unit of GAS is 0.00000001. 100 million NeoGas, corresponding to the 100 million NEO, will be generated through a decay algorithm in about 22 years to addresses holding NEO. While in Ethereum accounts hold Ether and GAS, in NEO the same logic is applied, with accounts being called Wallets.

2.4.2 NEO Nodes

In NEO platform two different types of nodes exit. Ordinary nodes, which represent a normal client connected to the blockchain and consensus nodes (or bookkeepers). The nodes connect between them via a peer-to-peer network with all messages within the network being sent by broadcast. Ordinary nodes use the network to transfer and exchange, accepting ledger data. Consensus nodes manage the ledger, validating new blocks.

2.4.3 Consensus Mechanism

NEO employes Delegated Byzantine Fault Tolerance (dBFt) as its consensus mechanism. In dBFt consensus nodes voted, or delegated, by NEO holders, listen actively for transaction data being broadcasted by the other nodes. NEO holders have interest in the system being honest and, as such, vote for delegates that they perceive as being trustworthy. Each consensus node store independently, in memory, its version of the data received from the network. After a certain pre-defined time, a speaker is drawn from the group of consensus nodes and forms a block from the transactions he received since the last block was validated. This speaker proposes his version of the block to the other delegates, which in turn verify if it follows a set of rules, compares it with their version of the block and votes the block for approval or disapproval. Two-thirds of the delegates need to approve the block for the block to be validated. In the cases where the block is not approved, a new speaker is drawn, which presents his block for voting. When NEO holders perceive delegates to be dishonest by, for instance, observing their voting patterns, they vote on other nodes to be consensus nodes, demoting the dishonest nodes to ordinary nodes. dBFt has a good finality, which means that once a block confirmation is final it cannot be rolled back or reverted and a fork in the blockchain is not possible.

2.4.4 NEO Smart Contracts

NEO allows a developer to write smart contracts in a set of high-level languages. These high-level languages are subsets of mainstream languages, where not all of the functionalities of the language are available for smart contract development. At the time of the writing of this document, NEO supported subsets of the following languages[24]:

- C#, VB.Net, F#

- Java, Kotlin
- Python

The ability to develop in one of these high-level languages is undoubtedly one of the main attractions of NEO. While Ethereum developers must learn a new programming language, such as Solidity, to develop smart contracts, NEO developers may use a programming language that they already know to start developing. Then, developers just need to adapt to the subset of the chosen programming language.

NEO smart contracts run on top of a virtual machine, the NeoVM, which, as depicted in Figure 2.4, in the dashed board, consists of three main modules.

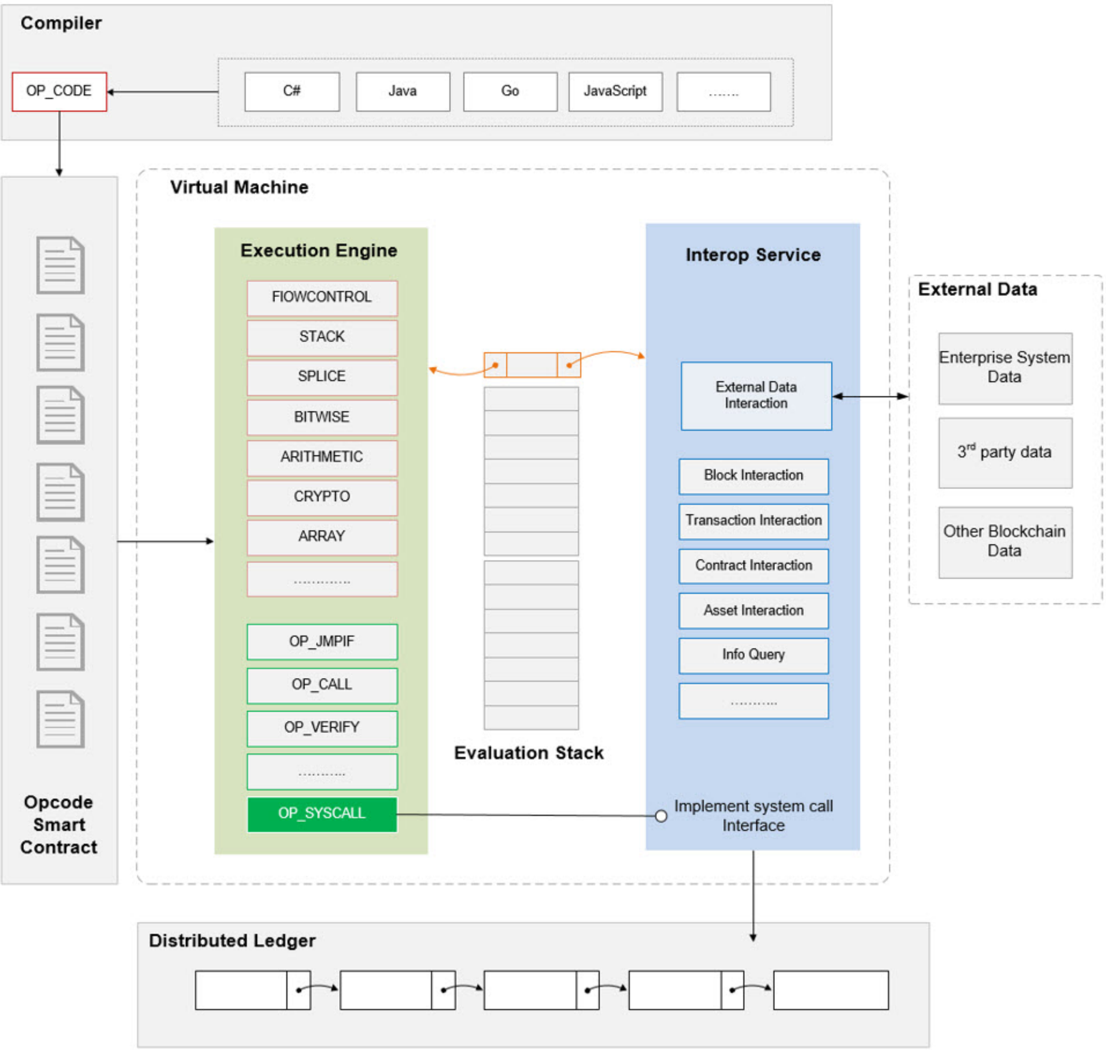


Figure 2.4: NeoVM architecture. Taken from *NEO Smart Contract Introduction*[24].

- In green is the VM execution engine, equivalent to a CPU. It executes common instructions (flow control, stack operations, bit operations, arithmetic operations, logical operations, cryptographic methods, and others) and interacts with the interoperable service layer (in blue).
- In grey, the evaluation stack, which is equivalent to the memory.

- In blue, the interoperable service layer of the virtual machine, which provides some Application Programming Interfaces (APIs) that enable the smart contract to access chain data such as block information, transaction information, smart contract information, asset information, etc. The interoperable service layer also provides persistent storage for each contract.

To deploy a smart contract to the network, the developer must pay a deployment fee[24], which serves as a payment for the resources of the network. At the time of the writing, this fee was of 500 NeoGas (or GAS), which translated to roughly 4,500 euros. Once a smart contract is deployed, its execution requires the payment of additional fees, which depend on the number and type of operations executed by the smart contract, with most of these operations defaulting to a fee of 0.001 NeoGas. However, the first 10 NeoGas of a smart contract execution or deployment are free, which, for most simple contracts, makes it that they can be executed for free. If the execution of the smart contract fails due to lack of NeoGas, the expended NeoGas will not be returned, to prevent malicious attacks that attempt to deplete network resources. These fees are considered to be service fees and are put in a pool for redistribution to all NEO holders. This distribution is proportional to the amount of NEO held by each holder.

2.5 Smart Contract Development Tools

2.5.1 Ganache

Ganache[1], formerly known as TestRPC, is an open-source[2] Node.js based Ethereum client that emulates a local Ethereum blockchain for development and testing purposes. It provides an Remote Procedure Call (RPC) interface, allowing the user to make RPCs to the local blockchain and inspect blocks and transactions during development, to better understand how a smart contract deployed on that local network behaves. This tool also comes with a wallet with ten accounts, each of them having a balance of 100 ether. The tool comes in two flavours:

- Ganache Application[3] - An electron application that provides a fully interactive GUI, depicted in Figure 2.5.
- Ganache-cli - A Command-Line Interface (CLI)[4], which is also included in Truffle Suite (see section 2.5.2).

2.5.2 Truffle Suite

Truffle[5] is a development environment, test framework and asset pipeline for Ethereum. At the time of the writing of this document Truffle claimed to be the most popular Ethereum development framework and implemented the following features[5]:

- Integrated smart contract compilation, linking, deployment and binary management.

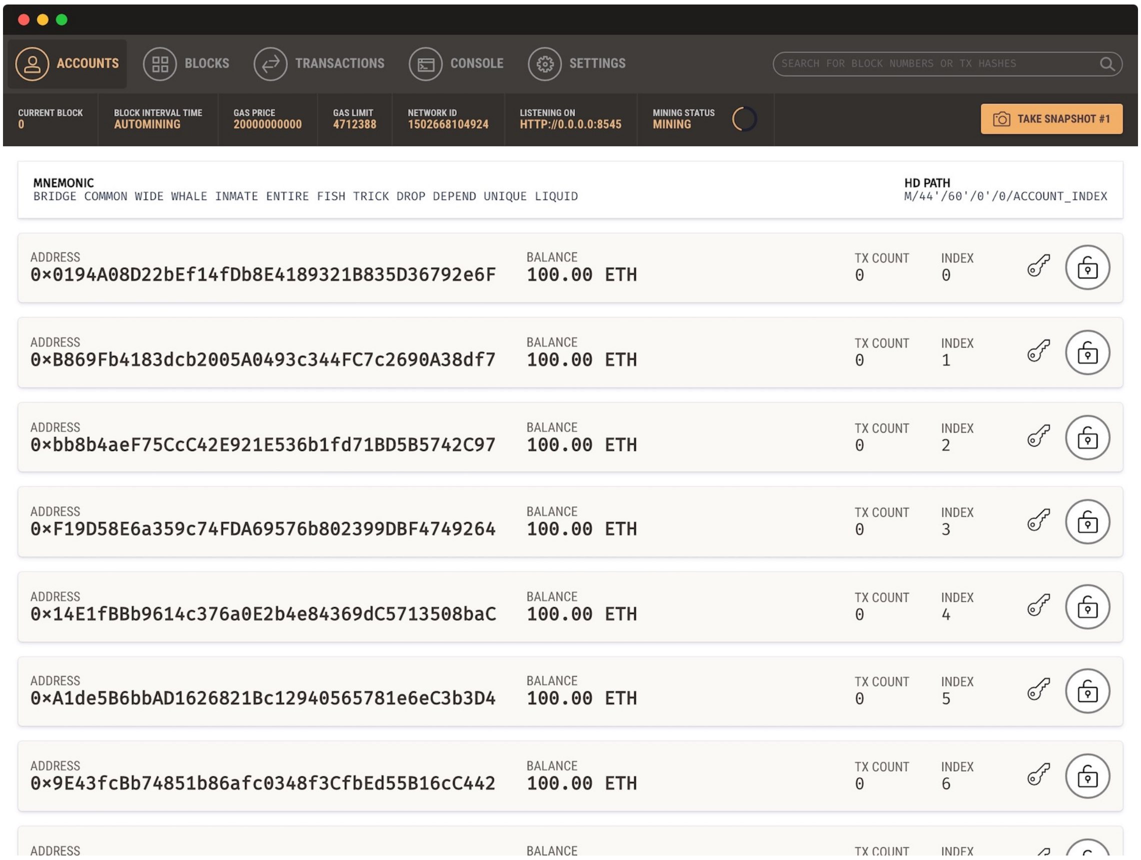
[1] `http://truffleframework.com/ganache/`

[2] `https://github.com/trufflesuite/ganache-core`

[3] `https://github.com/trufflesuite/ganache`

[4] `https://github.com/trufflesuite/ganache-cli`

[5] `http://truffleframework.com`



CURRENT BLOCK 0
BLOCK INTERVAL TIME AUTOMINING
GAS PRICE 20000000000
GAS LIMIT 4712388
NETWORK ID 1502668104924
LISTENING ON HTTP://0.0.0.0:8545
MINING STATUS MINING

MNEMONIC
BRIDGE COMMON WIDE WHALE INMATE ENTIRE FISH TRICK DROP DEPEND UNIQUE LIQUID

HD PATH
M/44'/60'/0'/0/ACCOUNT_INDEX

ADDRESS	BALANCE	TX COUNT	INDEX
0×0194A08D22bEf14fDb8E4189321B835D36792e6F	100.00 ETH	0	0
0×B869Fb4183dcb2005A0493c344FC7c2690A38df7	100.00 ETH	0	1
0×bb8b4aeF75CcC42E921E536b1fd71BD5B5742C97	100.00 ETH	0	2
0×F19D58E6a359c74FDA69576b802399DBF4749264	100.00 ETH	0	3
0×14E1fBBb9614c376a0E2b4e84369dC5713508baC	100.00 ETH	0	4
0×A1de5B6bbAD1626821Bc12940565781e6eC3b3D4	100.00 ETH	0	5
0×9E43fcBb74851b86afc0348f3CfbEd55B16cC442	100.00 ETH	0	6

Figure 2.5: Ganache GUI

- Automated contract testing with Mocha (a JavaScript test framework for Node.js and the browser) and Chai (an assertion library for Node.js and the browser).
- Configurable build pipeline with support for custom build processes.
- Scriptable deployment and migrations framework.
- Network management for deploying to many public and private networks.
- Interactive console for direct contract communication.
- Instant rebuilding of assets during development.
- External script runner that executes scripts within a Truffle environment.

Ganache (see section 2.5.1) is being developed and maintained by Truffle. Truffle is shipped with the CLI version of Ganache.

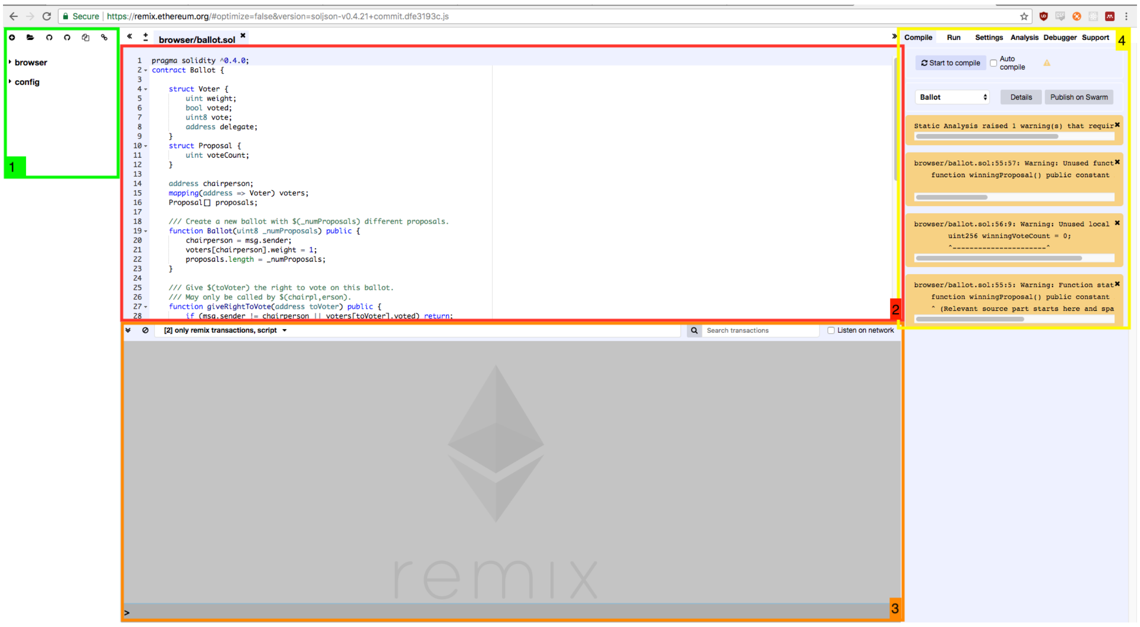
2.5.3 Remix IDE

Remix[6] is a browser-based compiler and Integrated Development Environment (IDE) for the Ethereum smart contract programming language Solidity. Remix features an integrated debugger and testing environment. At the date of the writing of this document, Ethereum developers could use Remix to:

[6] `https://github.com/ethereum/remix-ide`

- Develop smart contracts using its integrated Solidity editor, compiler and debugger.
- Access the state and properties of an already deployed smart contract.
- Debug already committed transactions.
- Reduce code bugs by using Remix to statically analyse Solidity code.
- User Remix to unit test a Solidity smart contract.

As seen in Figure 2.6, Remix IDE GUI can be subdivided in four different parts:



```
pragma solidity ^0.4.0;
contract Ballot {

    struct Voter {
        uint weight;
        bool voted;
        uint8 vote;
        address delegate;
    }
    struct Proposal {
        uint voteCount;
    }

    address chairperson;
    mapping(address => Voter) voters;
    Proposal[] proposals;

    /// Create a new ballot with $(_numProposals) different proposals.
    function Ballot(uint8 _numProposals) public {
        chairperson = msg.sender;
        voters[chairperson].weight = 1;
        proposals.length = _numProposals;
    }

    /// Give $(toVoter) the right to vote on this ballot.
    /// May only be called by $(chairpl,erson).
    function giveRightToVote(address toVoter) public {
        if (msg.sender != chairperson || voters[toVoter].voted) return;
```

```
Static Analysis raised 1 warning(s) that requir

browser/ballot.sol:55:57: Warning: Unused funct
    function winningProposal() public constant

browser/ballot.sol:56:9: Warning: Unused local
        uint256 winningVoteCount = 0;
        ^----------------------^

browser/ballot.sol:55:5: Warning: Function stat
    function winningProposal() public constant
    ^ (Relevant source part starts here and spa
```

Figure 2.6: Remix IDE

1. File Explorer - Allows the user to add, remove or rename Solidity files to be used by the IDE.
2. Solidity Editor - Displays opened Solidity files, automatically saves code editions, provides syntax highlighting of Solidity keywords and re-compiles the code everytime it detects a change in the code.
3. Terminal - It integrates a JavaScript interpreter and the web3 object. It enables the execution of JavaScript scripts which interact with the current context. It displays important actions made while interacting with the Remix IDE, such as sending a transaction. It displays transactions that are mined in the current context[1].
4. Tabs Panel - The tabs panel holds the tools that allows the user to compiler a smart contract, see the result of the static analysis on the contract, unit test a smart contract and change different settings in the editor.

2.5.4 NEO GUI Developer

NEO GUI Developer[7] is a tool being developed by City of Zion - an independent group of open source developers formed to support the NEO core and ecosystem (CoZ) and which is

[7] https://github.com/CityOfZion/neo-gui-developer

built on top of NEO GUI - the official NEO client and wallet. It includes all the features of the official NEO GUI, such as deploying smart contracts and creating NEO wallets, while also adding some useful features for developers. Like the original NEO GUI it also works as a full node of the network, acting both as a client and as a server. At the time of the writing of this document, it included the following features[16]:

- Event log - The tool features a tab on the GUI which displays all the logs created by any smart contract in the network calling `Runtime.Log()` and `Runtime.Notify()`. Along with the log message, it shows the local time when the event is received, the current block height, event type and the script hash.

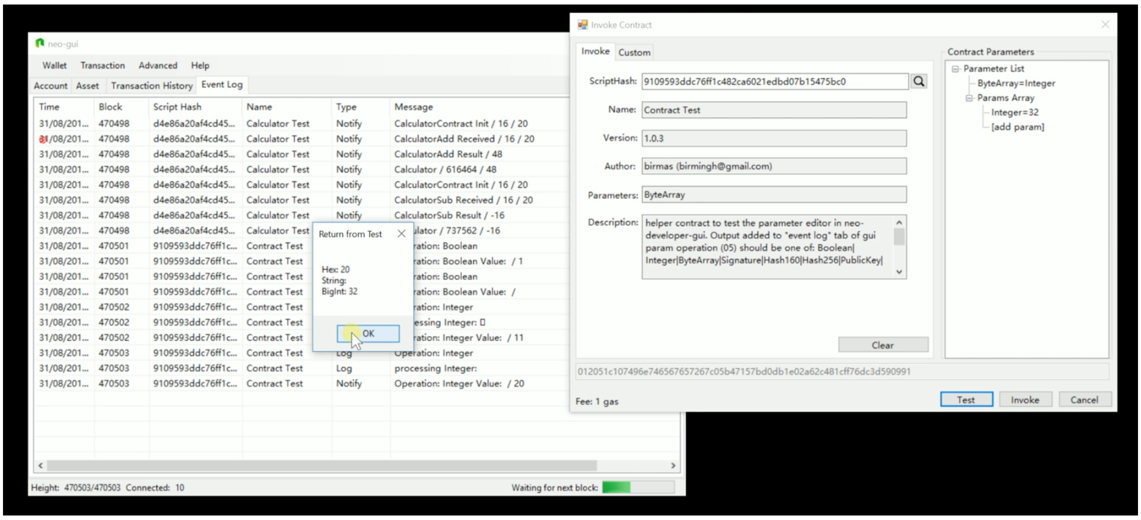
- Smart contract return - This feature allows the user to verify the `CheckWitness()` operation during test invocations by displaying a message pop-up. It also pop-ups a dialogue with the result of a smart contract test invocation.

- Smart contract monitor - After being deployed or looked up, the smart contract script is added to a watch list and its status (successfully deployed or unavailable), script info and script hash are displayed in an information box.

- Add parameters to smart contract invocation - This feature allows the user to build an array of objects in the GUI with the arguments to be passed during a contract invocation. The tool currently supports `Array`, `ByteArray`, `Integer`, `Hash160`, `Hash256`, `PublicKey` and `Signature` types, which can be added by selecting them from a drop-down list.

- Simple field validation - Parameters that were set as required when the contract was deployed are marked as required when the user tries to test invoke a contract. `Hash160`, `Hash256` and `PublicKey` fields are also validated.

Although it provides some useful features, this tool suffers from the same compatibility issues as NEO GUI, being compatible only with Windows 7 (Service Pack 1), Windows 8 and Windows 10. It also has a very minimalist and poor design, as seen in Figure 2.7 and lacks documentation.



Time	Block	Script Hash	Name	Type	Message
31/08/201...	470498	d4e86a20af4cd45...	Calculator Test	Notify	CalculatorContract Init / 16 / 20
31/08/201...	470498	d4e86a20af4cd45...	Calculator Test	Notify	CalculatorAdd Received / 16 / 20
31/08/201...	470498	d4e86a20af4cd45...	Calculator Test	Notify	CalculatorAdd Result / 48
31/08/201...	470498	d4e86a20af4cd45...	Calculator Test	Notify	Calculator / 616464 / 48
31/08/201...	470498	d4e86a20af4cd45...	Calculator Test	Notify	CalculatorContract Init / 16 / 20
31/08/201...	470498	d4e86a20af4cd45...	Calculator Test	Notify	CalculatorSub Received / 16 / 20
31/08/201...	470498	d4e86a20af4cd45...	Calculator Test	Notify	CalculatorSub Result / -16
31/08/201...	470498	d4e86a20af4cd45...	Calculator Test		ulator / 737562 / -16
31/08/201...	470501	9109593ddc76ff1c...	Contract Test		ration: Boolean
31/08/201...	470501	9109593ddc76ff1c...	Contract Test		ration: Boolean Value: / 1
31/08/201...	470501	9109593ddc76ff1c...	Contract Test		ration: Boolean
31/08/201...	470501	9109593ddc76ff1c...	Contract Test		ration: Boolean Value: /
31/08/201...	470502	9109593ddc76ff1c...	Contract Test		ration: Integer
31/08/201...	470502	9109593ddc76ff1c...	Contract Test		essing Integer: □
31/08/201...	470502	9109593ddc76ff1c...	Contract Test		ration: Integer Value: / 11
31/08/201...	470503	9109593ddc76ff1c...	Contract Test	Log	Operation: Integer
31/08/201...	470503	9109593ddc76ff1c...	Contract Test	Log	processing Integer:
31/08/201...	470503	9109593ddc76ff1c...	Contract Test	Notify	Operation: Integer Value: / 20

Height: 470503/470503 Connected: 10

Return from Test

Hex: 20
String:
BigInt: 32

ScriptHash: 9109593ddc76ff1c482ca6021edbd07b15475bc0

Name: Contract Test

Version: 1.0.3

Author: birmas (birmingh@gmail.com)

Parameters: ByteArray

Description: helper contract to test the parameter editor in neo-developer-gui. Output added to "event log" tab of gui param operation (05) should be one of: Boolean| Integer|ByteArray|Signature|Hash160|Hash256|PublicKey|

Contract Parameters
- Parameter List
  - ByteArray=Integer
  - Params Array
    - Integer=32
    - [add param]

012051c107496e746567657267c05b47157bd0db1e02a62c481cff76dc3d590991

Fee: 1 gas

Figure 2.7: Neo GUI Developer - Smart contract test invocation (taken, with authorization, from CoZ Youtube channel)

2.5.5 NEO Debugger Tools

NEO Debugger Tools [8] is a set of tools which aim to help on the development of NEO smart contracts. They were first created by a Portuguese developer, Sérgio Flores, also known as Relfos in the CoZ community, and were later adopted by CoZ. This tool suite consists of a CLI disassembler and a GUI debugger. At the time of the writing of this document, it implemented the following features:

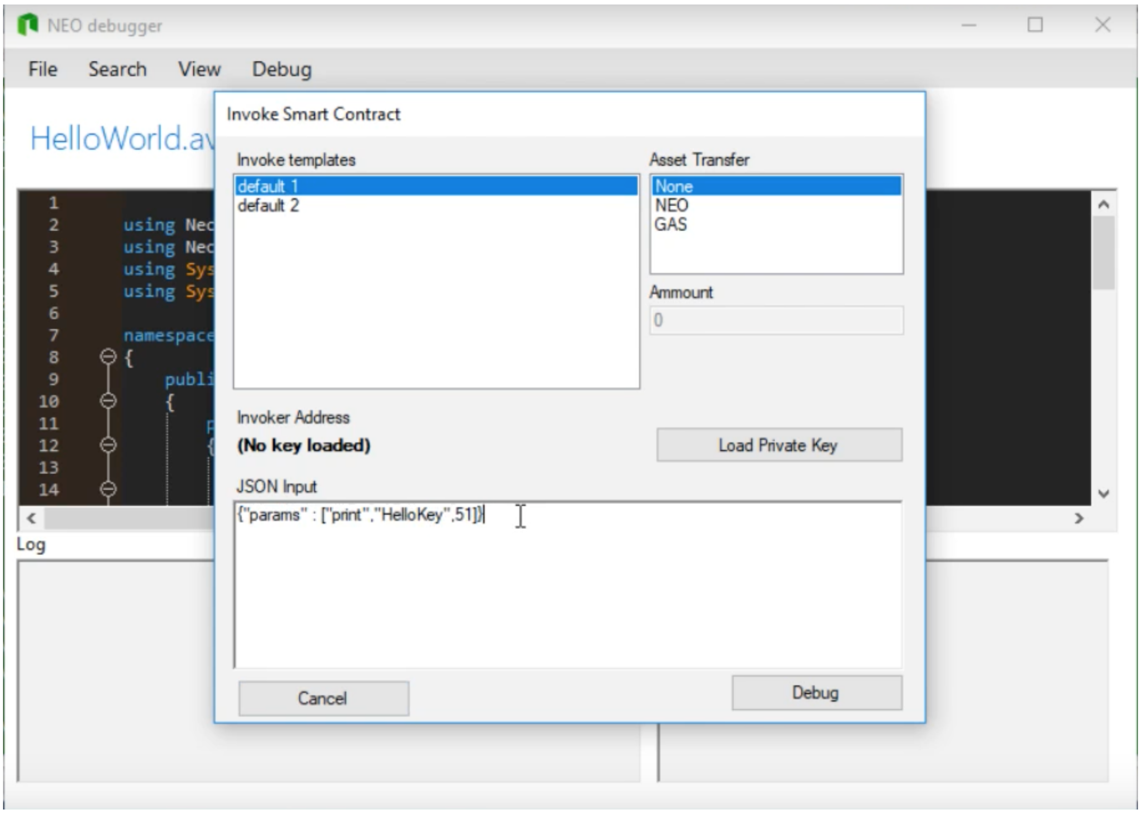
- Support of any NEO .avm (NEO VM bytecode) code, regardless of the language or compiler used.

- Source viewer with syntax highlight, as seen in Figure 2.8.

- Contract debugging with run, step and set breakpoints options, as seen in Figure 2.9.

- Toggling between source code and VM instructions, as seen in Figure 2.10.

- Test invocation result and GAS usage.



Invoke templates: default 1, default 2

Asset Transfer: None, NEO, GAS

Ammount: 0

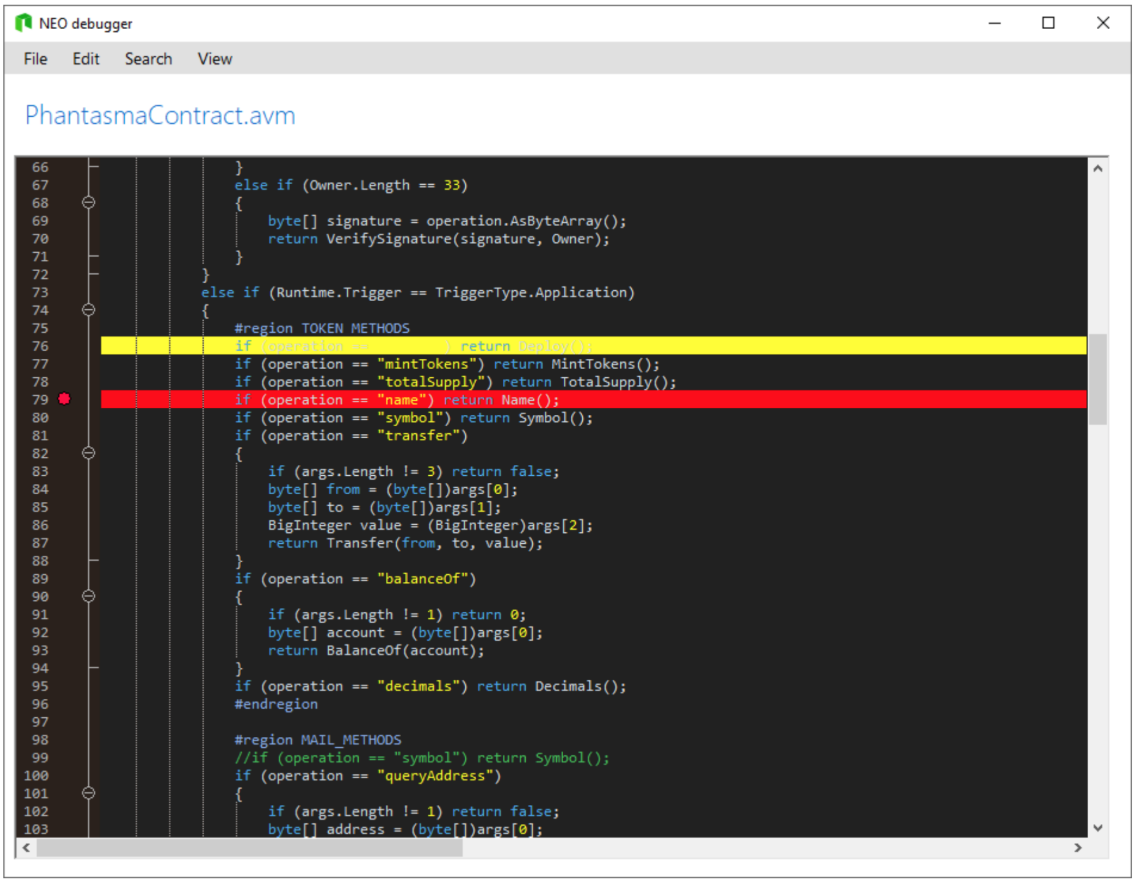
Invoker Address: **(No key loaded)**

JSON Input:

```
{"params" : ["print","HelloKey",51]}
```

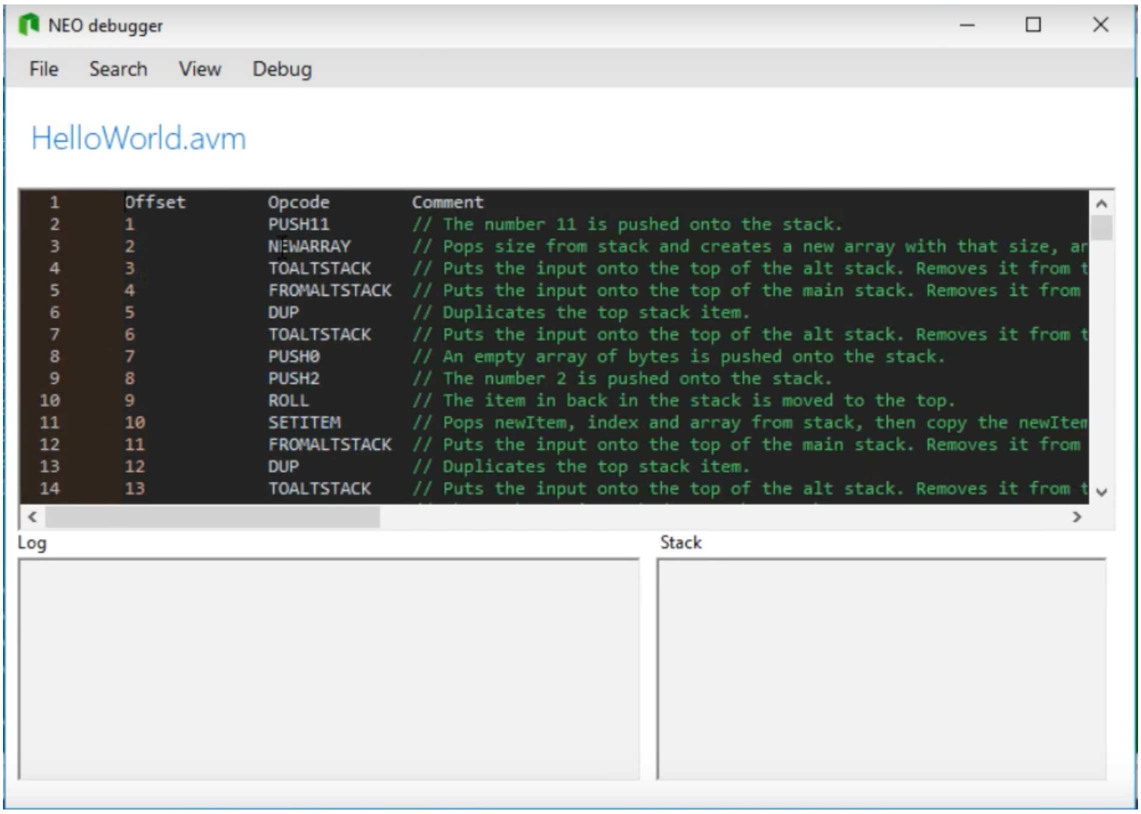
Figure 2.8: Neo Debugger - Debugging a NEO smart contract. Taken, with consent, from Nikolaj-K youtube channel `https://www.youtube.com/watch?v=KnPHIaEsgtA`.

[8] `https://github.com/CityOfZion/neo-debugger-tools`



```
            }
            else if (Owner.Length == 33)
            {
                byte[] signature = operation.AsByteArray();
                return VerifySignature(signature, Owner);
            }
        }
        else if (Runtime.Trigger == TriggerType.Application)
        {
            #region TOKEN METHODS
            if (operation ==          ) return Deploy();
            if (operation == "mintTokens") return MintTokens();
            if (operation == "totalSupply") return TotalSupply();
            if (operation == "name") return Name();
            if (operation == "symbol") return Symbol();
            if (operation == "transfer")
            {
                if (args.Length != 3) return false;
                byte[] from = (byte[])args[0];
                byte[] to = (byte[])args[1];
                BigInteger value = (BigInteger)args[2];
                return Transfer(from, to, value);
            }
            if (operation == "balanceOf")
            {
                if (args.Length != 1) return 0;
                byte[] account = (byte[])args[0];
                return BalanceOf(account);
            }
            if (operation == "decimals") return Decimals();
            #endregion

            #region MAIL_METHODS
            //if (operation == "symbol") return Symbol();
            if (operation == "queryAddress")
            {
                if (args.Length != 1) return false;
                byte[] address = (byte[])args[0];
```

Figure 2.9: Neo Debugger - Inserting breakpoints. Taken, with consent, from Nikolaj-K youtube channel `https://www.youtube.com/watch?v=KnPHIaEsgtA`.



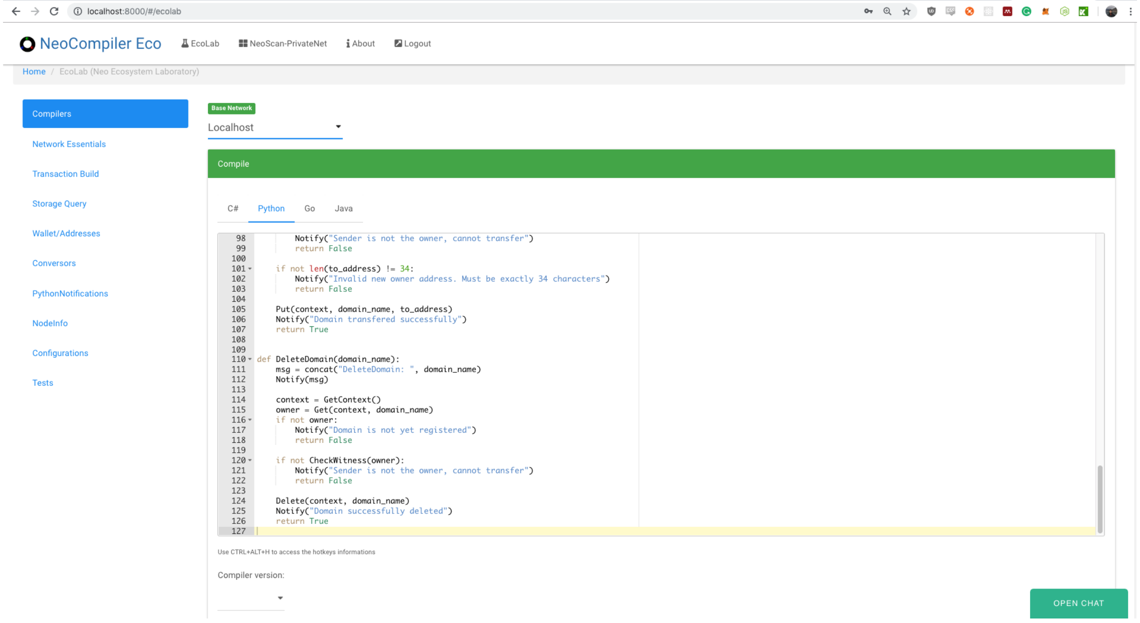
```
Offset          Opcode          Comment
1               PUSH11          // The number 11 is pushed onto the stack.
2               NEWARRAY        // Pops size from stack and creates a new array with that size, ar
3               TOALTSTACK      // Puts the input onto the top of the alt stack. Removes it from t
4               FROMALTSTACK    // Puts the input onto the top of the main stack. Removes it from
5               DUP             // Duplicates the top stack item.
6               TOALTSTACK      // Puts the input onto the top of the alt stack. Removes it from t
7               PUSH0           // An empty array of bytes is pushed onto the stack.
8               PUSH2           // The number 2 is pushed onto the stack.
9               ROLL            // The item in back in the stack is moved to the top.
10              SETITEM         // Pops newItem, index and array from stack, then copy the newItem
11              FROMALTSTACK    // Puts the input onto the top of the main stack. Removes it from
12              DUP             // Duplicates the top stack item.
13              TOALTSTACK      // Puts the input onto the top of the alt stack. Removes it from t
```

Figure 2.10: Neo Debugger - VM instructions. Taken, with consent, from Nikolaj-K youtube channel `https://www.youtube.com/watch?v=KnPHIaEsgtA`.

However, this suit has some limitations, as recognised by its creator in his documentation (available on the tool Github repository). For now, although supporting any NEO `.avm` code, it only works when a `.neomap` file is generated by the compiler in order to map source code lines with the correspondent `.avm` code. Smart contract source code is also limited to a single file, and it's not yet possible to inspect variable values. Also, some NEO syscalls and API calls are not supported yet. Similarly to NEO GUI Developer, it also only compatible with Windows Operating System (OS).

2.5.6 NeoCompiler Eco

NeoCompiler Eco[9] is an open-source Web and Android NEO smart contract development ecosystem. NeoCompiler aims to provide a didactic and straightforward interface, in order to make the tool usable not only by advanced developers but also by beginners and regular users. Similarly to Ethereum Remix IDE (see section 2.5.3) it also alleviates the user of the burden of setting up a development environment, providing the user with a ready to use compiler and with a NEO private net running on the back-end to where the user can deploy his contracts.



```
        Notify("Sender is not the owner, cannot transfer")
        return False

    if not len(to_address) != 34:
        Notify("Invalid new owner address. Must be exactly 34 characters")
        return False

    Put(context, domain_name, to_address)
    Notify("Domain transfered successfully")
    return True


def DeleteDomain(domain_name):
    msg = concat("DeleteDomain: ", domain_name)
    Notify(msg)

    context = GetContext()
    owner = Get(context, domain_name)
    if not owner:
        Notify("Domain is not yet registered")
        return False

    if not CheckWitness(owner):
        Notify("Sender is not the owner, cannot transfer")
        return False

    Delete(context, domain_name)
    Notify("Domain successfully deleted")
    return True

```

Use CTRL+ALT+H to access the hotkeys informations

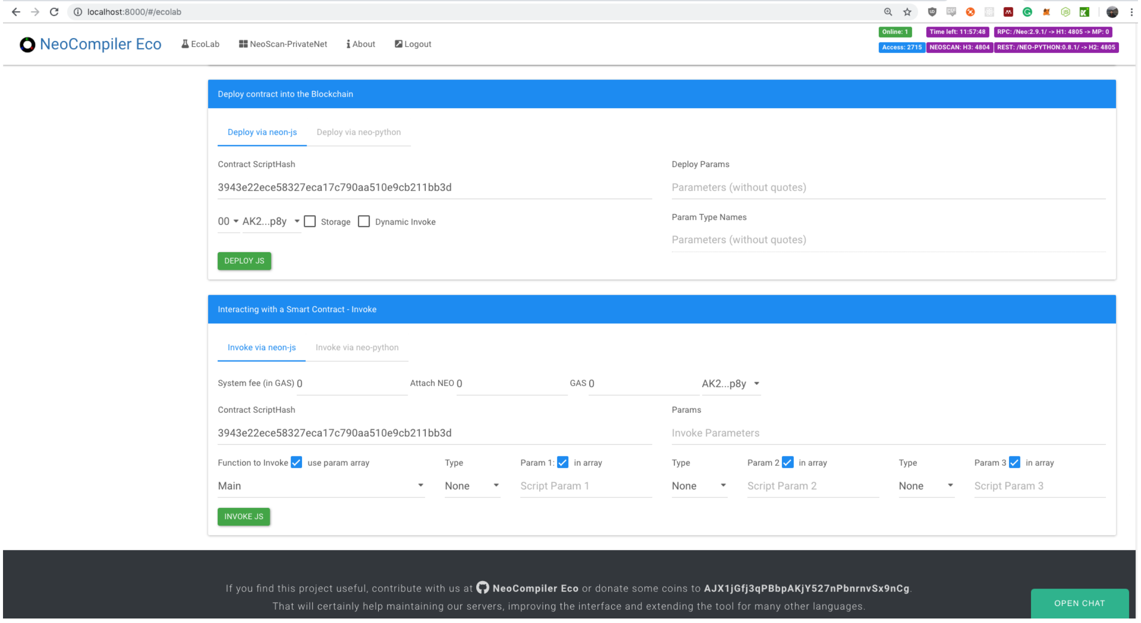
Compiler version:

Figure 2.11: NeoCompiler Eco - Ilustration of the Web tool compiler.

Currently, this tool presents the following features:

- C#, Python, Java and Go code compilation to AVM and ABI code, which is done in a back-end server (as seen in Figure 2.11).
- Smart contract deployment to a private network running on a back-end server and test-invocation of smart contracts deployed on that private network (as seen in Figure 2.12).
- Different wallets provided for testing purposes, containing generous amounts of NEO and GAS to be used in the private network (depicted in Figure 2.13).
- Multiple utilities for interacting with smart contracts with the NEO blockchain.

[9] https://github.com/NeoResearch/neocompiler-eco



Deploy contract into the Blockchain

Contract ScriptHash: 3943e22ece58327eca17c790aa510e9cb211bb3d

Deploy Params: Parameters (without quotes)

00 AK2...p8y □ Storage □ Dynamic Invoke

Param Type Names: Parameters (without quotes)

Interacting with a Smart Contract - Invoke

System fee (in GAS) 0   Attach NEO 0   GAS 0   AK2...p8y

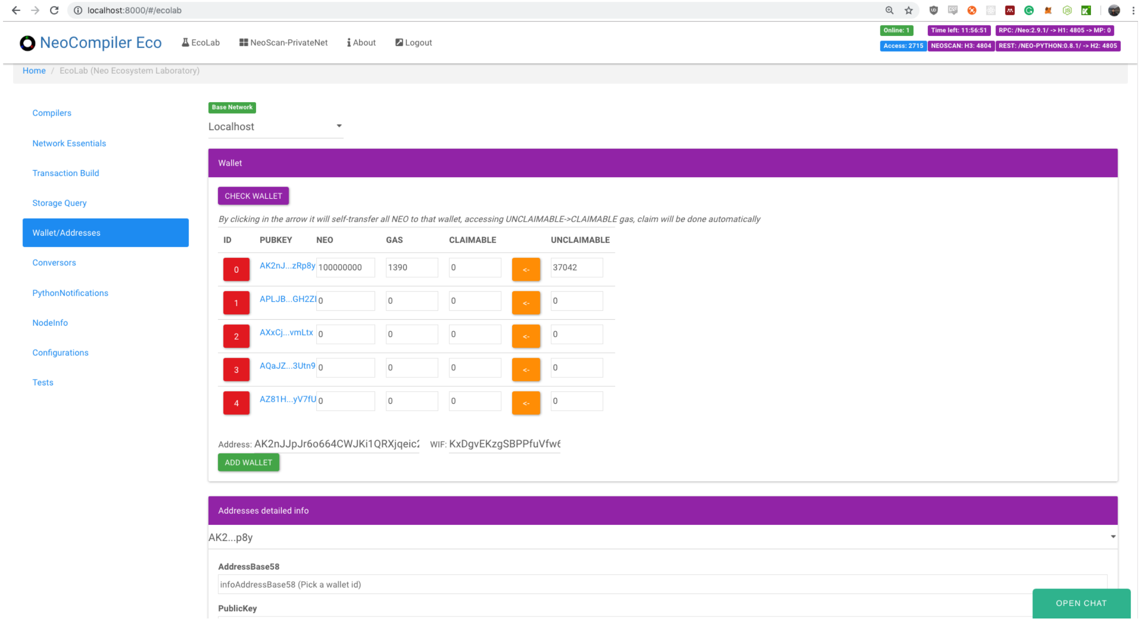
Contract ScriptHash: 3943e22ece58327eca17c790aa510e9cb211bb3d

Params: Invoke Parameters

Function to Invoke ☑ use param array: Main   Type: None   Param 1: ☑ in array: Script Param 1   Type: None   Param 2 ☑ in array: Script Param 2   Type: None   Param 3 ☑ in array: Script Param 3

If you find this project useful, contribute with us at NeoCompiler Eco or donate some coins to **AJX1jGfj3qPBbpAKjY527nPbnrnvSx9nCg**.
That will certainly help maintaining our servers, improving the interface and extending the tool for many other languages.

Figure 2.12: NeoCompiler Eco - Ilustration of the Web tool deployment and test environment.



Base Network: Localhost

Wallet

By clicking in the arrow it will self-transfer all NEO to that wallet, accessing UNCLAIMABLE->CLAIMABLE gas, claim will be done automatically

ID	PUBKEY	NEO	GAS	CLAIMABLE		UNCLAIMABLE
0	AK2nJ...zRp8y	100000000	1390	0	<-	37042
1	APLJB...GH2ZI	0	0	0	<-	0
2	AXxCj...vmLtx	0	0	0	<-	0
3	AQaJZ...3Utn9	0	0	0	<-	0
4	AZ81H...yV7fU	0	0	0	<-	0

Address: AK2nJJpJr6o664CWJKi1QRXjqeic   WIF: KxDgvEKzgSBPPfuVfw

Addresses detailed info

AK2...p8y

AddressBase58

infoAddressBase58 (Pick a wallet id)

PublicKey

Figure 2.13: NeoCompiler Eco - Ilustration of the Web tool wallets page.

2.5.7 NEO WebDebugger

NEO WebDebugger[10] is a planned port of NEO Debugger Tools (see section 2.5.5) for the Web. However, at the time of the writing of this document, this tool was still in development.

[10] `https://github.com/CityOfZion/neo-debugger-tools/tree/master/NEO-WebDebugger`

2.5.8 Conclusion

Comparing the studied tools by observing Table 2.1, a rift between the state of the Ethereum development tools and the NEO development tools can be observed, with Ethereum development tools being more powerful and implementing more features.

	Provides Private Network	Needs Local Node Sync	Contract Deployment	Test Wallets	Invoke	Testing	Code Editor	Static Analysis	Debugger	GUI	Block Explorer	OS Dependent
Ganache	Yes	Yes	No	Yes	No	No	No	-	No	Yes	Yes	No
Truffle	Yes*	Yes	Yes	Yes*	Yes	Yes	No	-	No	No	No	No
Remix	Yes**	Yes**	Yes	Yes	Yes	Yes	Yes	Yes	Yes***	Yes	No	No
NeoGuiDev	No	Yes	Yes	No	Yes	No	No	-	No	Yes	No	Yes
NeoDebugger	No	No	No	No	Yes	No	No	-	Yes	Yes	No	Yes
NeoCompiler	Yes	No	Yes	Yes	Yes	No	Yes	No	No	Yes	Yes****	No

* Provided by Ganache-cli

** The user can either run on client-side simulated blockchain or provide its own private network

*** The debugger can only be used in already existing and commited transactions

**** Implements NEOSCAN, an open-source blockchain explorer

Table 2.1: Comparison between the different development tools - Neo Web Debugger was excluded as it's still in development

It can be seen that while Truflle and Remix implement smart contract testing, there is not any existing tool that does so for the NEO blockchain. Also, while all of the Ethereum tools can run in any common OS, the same does not happen with its counterpart tools, where only the NeoCompiler tool is OS agnostic. Finally, it is also noted that NeoCompiler, the NEO development tool that includes more features, differs from Remix, its Ethereum counterpart, for lacking features such as smart contract testing, debugging and static analysis of code.

This difference between the state of these blockchain tools can be explained by the broader adoption of the Ethereum community, which translates in more developers using Etherium, and as such, on companies and developers in investing in the development and maintenance of these tools.

2.6 Software Testing

"The Art of Software Testing" describes software testing as "a process, or a series of processes, designed to make sure computer code does what it was designed to do and, conversely, that it does not do anything unintended"[30]. Software testing is, as such, essential to guarantee the predictability and consistency of software[30]. If consistency and predictability are valuable properties in simple applications, where software malfunctioning translates into a mild nuisance to the end-user, they are of the utmost importance when developing more serious applications, such as a smart contract that manages user's assets.

2.6.1 Unit Testing

In unit testing, software is assessed in regard to its implementation[33]. Ian Sommerville, in "Software Engineering", defines unit testing as "the process of testing program components, such as methods or object classes"[37]. As such, when unit testing, the developer should attempt to test the software implementation by testing the individual subprograms, classes and procedures of the program, focusing on the building blocks of the program rather than focusing on the program as a whole[30].

2.6.2 Test Case

The "IEEE Standard for System, Software, and Hardware Verification and Validation" defines a test case as:

- (A) "A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement"[2].
- (B) "Documentation specifying inputs, predicted results, and a set of execution conditions for a test item"[2].

To design test cases for unit testing, two types of information are required: the specification of the module to test and its source code[30]. With the module specification, the tester is able to define the module's input and output parameters, as well as its function in the program[30].

A test case should consist, as such, of a description of the input data for the module to be tested and of a precise description of the correct output for that set of input data[30][33].

A collection of test cases is called a test suite, or test set.

2.6.3 Test Automation

Unit testing should be automated whenever possible[37], as it brings clear benefits such as making it easier to run existing (regression) tests on a new version of the program, enabling the tester to run more tests more often and increase test repeatability and consistency[19].

An automated test should be comprised of three parts[37]:

1. A setup part, where the system is initialised with the test case, namely, the expected inputs and outputs;
2. A call part, where the method to be tested is called;
3. An assertion part, where the expected outputs are compared with the result of the call;

2.6.4 Test Automation Framework

"Introduction to Software Testing" defines a test framework as "a set of assumptions, concepts, and tools that support test automation"[33]. A test framework should provide[37]:

- The ability to evaluate expected test results through assertions;
- The ability to share common test data between tests
- The ability to organise tests in suites and run multiple test cases;
- The ability to run tests either from a GUI or a CLI;

Chapter 3

Project Management

3.1 Risk Management

Due to the novelty and constant change of the Blockchain technology, and NEO, in particular, risk management is of the utmost importance in this project.

Risk management is a process that involves anticipating risks that might affect the quality of the software being developed or the project schedule, and then taking action to avoid these risks[22][32]. This is an iterative process that should be addressed during all the stages of the project and, according to Sommerville[37], should be comprised of four phases:

1. Risk identification - where the risks that pose the biggest threats to the project or the product should be identified;

2. Risk analysis - where an estimate of the probability and seriousness of the risk should be assessed;

3. Risk planning - where plans are made to address the risk either by avoiding it or by minimising its effects on the project;

4. Risk monitoring - where risk assessment and mitigation plans should be revised whenever there is more information on the risk;

In the next tables, project risks were identified. An analysis of each risk was conducted, classifying the probability of the risk as insignificant, low, moderate, high, or very high. The effects of each risk were also assessed and classified as catastrophic (threaten the survival of the project), serious (would cause major delays), tolerable (delays are within allowed contingency), or insignificant.

Risk1 - Changes in Underlying Technology	
Description	Changes on underlying technology such as smart contract compilers, SDKs or nodes may cause dependency issues and software crashes
Probability	High
Effects	Serious
Affects	Project Schedule, Performance of the Software
Removable	No
Mitigation Plan	A modular software architecture should be planned for the project, making it easier to update or remove deprecated modules.

Table 3.1: Risk1 - Changes in Underlying Technology

Risk2 - A Competitor Tool is Launched	
Description	A competitor tool, which offers a similar value proposition and features is announced, or launched, before the system is completed
Probability	Moderate
Effects	Tolerable
Affects	Business, Project Schedule
Removable	No
Mitigation Plan	The implementation of new features can be studied in order for the tool to offer a better value proposition than the competitor tool.

Table 3.2: Risk2 - A Competitor Tool is Launched

Risk3 - Unfamiliarity with the required technologies	
Description	The developer unfamiliarity with the required technologies (Angular, Node and Docker) can make development take longer than expected.
Probability	Moderate
Effects	Tolerable
Affects	Project Schedule
Removable	No
Mitigation Plan	An assessment of the state of the development should be done each week to detect possible delays in the project schedule. In the event that these delays are inevitable, the less important functional requirements should be left to the end. To do so, a MoScoW analysis of the requirements should be done in order to prioritise these requirements.

Table 3.3: Risk3 - Unfamiliarity with the required technologies

3.1.1 Risk Monitoring

Risk monitoring was required in order to keep the different risks described in the Risk Management subsection (see 3.1) under control.

To guarantee that the identified risks were under control and the correspondent risk mitigation plans were being followed, as well as to give advice on how to better manage his time and his project, the intern and his advisor from Whitesmith had weekly meetings where project status and concerns were discussed. The intern was also advised to take some time each day to reflect on the latest day of work and to plan his day accordingly.

The intern was also encouraged to contact his company advisor and coworkers whenever needed.

3.2 Development Methodology

NEO is a new platform which is still in development and evolving fast. It was then essential to choose a development methodology that not only fitted with the company's development style, but that could also adapt to the fast pace of changes that happen on the NEO platform. It was also crucial for this methodology to be able to adapt to the mitigation plans chosen to address the risks identified in the previous section (see section 3.1).

The Kanban[3], an agile and lean methodology was then chosen, which is suited for individuals and small teams of developers. Kanban allows for changes in requirements at any time during the project, for short cycle times and fast delivery of features, which results in more frequent feedback. In Kanban, work items are represented visually on a Kanban board, allowing team members and stakeholders to check its state at any given time, with each piece of work represented as a separate card on the board.

3.3 Project Management Tools

In this section, the tools used in the management of the project are presented.

3.3.1 Trello

Trello was used as a Kanban board and project management tool in the project. Trello is a cloud-based tool that enables project managers and developers to manage teams and projects[26].

Trello allows users to organise projects into Trello boards. In a Trello board, users can create tasks, which represent a small body of work. Users can be assigned to tasks and tasks can hold text, lists, to-do items, images, links, pdf files and commentaries.

In this project, each task represented a different User Story and was labelled according to the Epic they belonged to (see section 4.4). Tasks were then given a name, which was preceded by its User Story ID and then split into lists.

Lists are used to create a workflow where cards are moved according to its progress, creating a Kanban board. In this project, the described lists were used, creating the following task life cycle:

- Backlog - This list held tasks that were waiting to be picked by a developer. When a developer picked a task from this list, he would assign himself to the task. Project managers could also assign developers to a task;
- Blocked - This list held tasks that, for some reason, were blocked. Only tasks that already started development could be moved to this list. Tasks could be blocked, for instance, if they needed other tasks to be completed in order to resume development;
- In Development - This list held tasks that were under development. Ideally, only one team member could have owned a task in this list at a time;

- Waiting for Code Review - This list held tasks that had already undergone development but were waiting for its code to be reviewed;

- In Code Review - This list held tasks that were undergoing code review. Only one task should be picked at a time for review by a developer. If a task failed in code review, it would then be moved to the backlog with its original developer assigned to it. Blocksmith developers were responsible for code reviewing the completed tasks;

- Ready to Deploy - This list held tasks that passed code review and were ready to be deployed in a new release;

- Done - This list held tasks that were already released and, as such, concluded the development cycle;

3.3.2 Github

Github offers remote web hosting of Git Version Control System (VCS) repositories, providing all of Git features and adding its own tools on top of them, such as the ability to clone or fork existing open-source repositories, do code reviews and more[29].

Github was used for version control, with the usage of two Git repositories. One for the development of the NEO Node Listener and one containing a fork of the NeoCompiler Eco open-source repository, used for the development of the test automation tools on top of this project. During development different branches were created in each repository for each functionality implemented, with a pull request and corresponding code review being made for each release.

Github issue tracker was also used, which allows users to create and catalogue a new issue if a bug is found on the current version of the project.

3.4 Planning

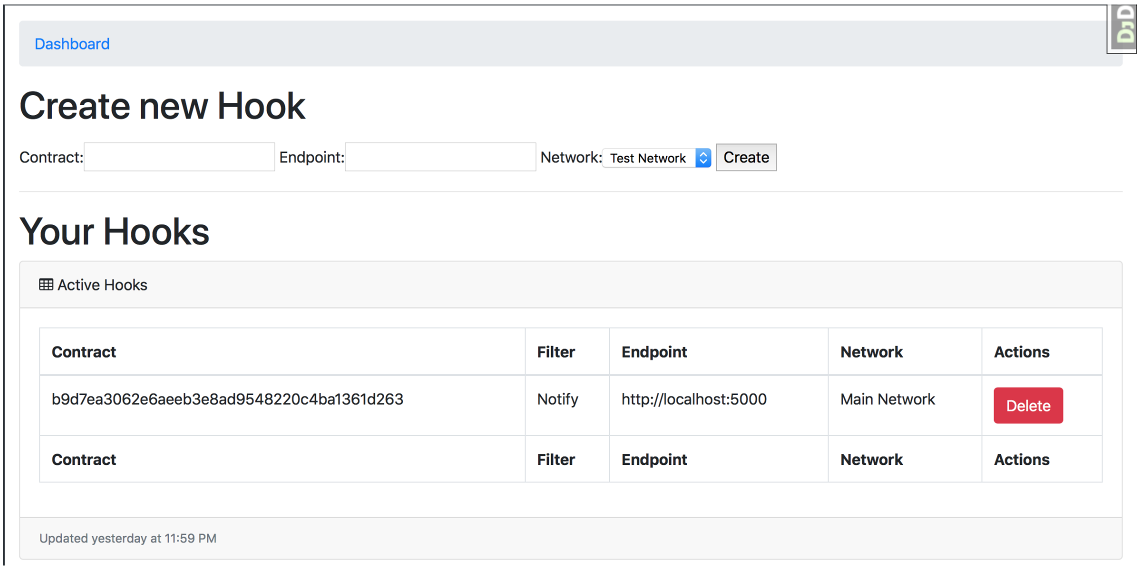
3.4.1 First Semester

To guarantee that the internship goals were possible to accomplish, the first semester was spent studying the state of the art, gathering the relevant requirements, developing a software architecture for the proposed solution and planning the second semester of the internship, dedicated to the development of the proposed solution.

As the NEO platform is still immature in terms of tools and documentation, an experiment was also conducted on the first semester in order to assess whether it would be possible to listen to events thrown by smart contracts in this platform network, as this functionality would be used as the core of the tool to be developed for the internship. To attempt to do so, an event listener based on the NEO Python[1] node client was developed, with which it was possible to access every event thrown by a smart contract running in the network that the client was listening to. The intern, with the help of one of the company's developers, also developed a simple python web application using the Django framework and a PostgreSQL database, which allowed users to register a smart contract to be listened to and an endpoint to be called whenever that smart contract threw an event. To call the

[1] `https://github.com/CityOfZion/neo-python`

endpoint, a queue and worker system, similar to the one depicted in the components diagram of the architecture section (see Figure 5.4), was used, implemented using Celery[2], an asynchronous task queue/job queue based on distributed message passing and Redis[3], an in-memory data structure store, as a message broker. The front-end application is depicted in Figure 3.1, whereas an endpoint, simulated by using python SimpleHTTPServer[4], can be seen being called in Figure 3.2 whenever events from the followed contract were listened (Figure 3.3).



Create new Hook

Contract: Endpoint: Network: Test Network

Your Hooks

Active Hooks

Contract	Filter	Endpoint	Network	Actions
b9d7ea3062e6aeeb3e8ad9548220c4ba1361d263	Notify	http://localhost:5000	Main Network	Delete
Contract	Filter	Endpoint	Network	Actions

Updated yesterday at 11:59 PM

Figure 3.1: The front-end application, where the user can input an endpoint to be called whenever an even from a given small contract is listened.

It was thus concluded that the internship goals were possible to be accomplished and that the development of the tool could be started by the intern in the second semester of the internship.

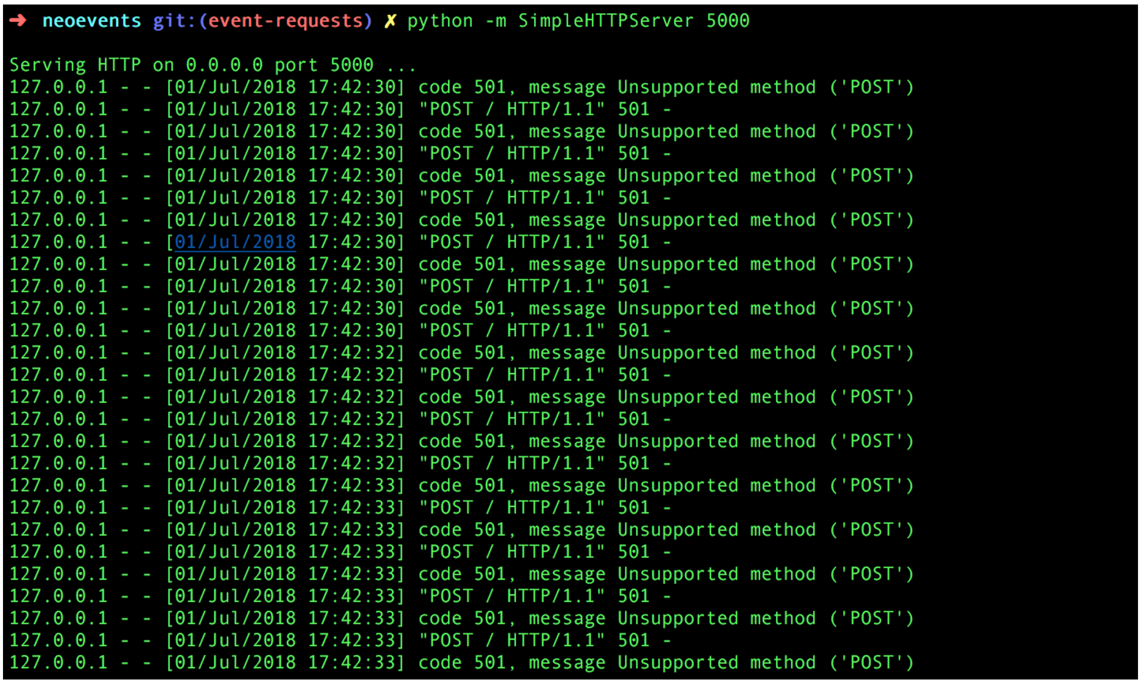
3.4.2 Second Semester

Planning flexibility is one of the features of the Kanban methodology. However, in order to track the progress of the internship and to ensure that the project and the required report are delivered on time, a Gantt diagram was built and was used to track the progress of the project, divided into milestones. This diagram can be consulted in Appendix B. The overall schedule of the second semester is also depicted in Figure 3.4.

[2] `https://celeryproject.org`

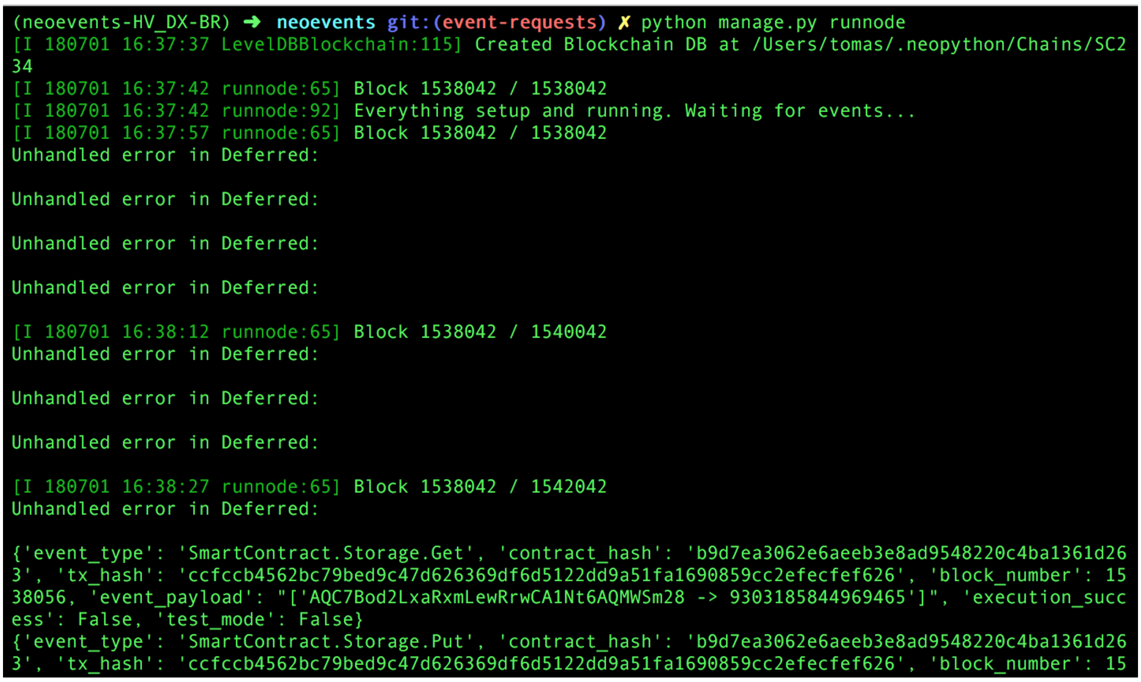
[3] `https://redis.io`

[4] `https://docs.python.org/2/library/simplehttpserver.html`



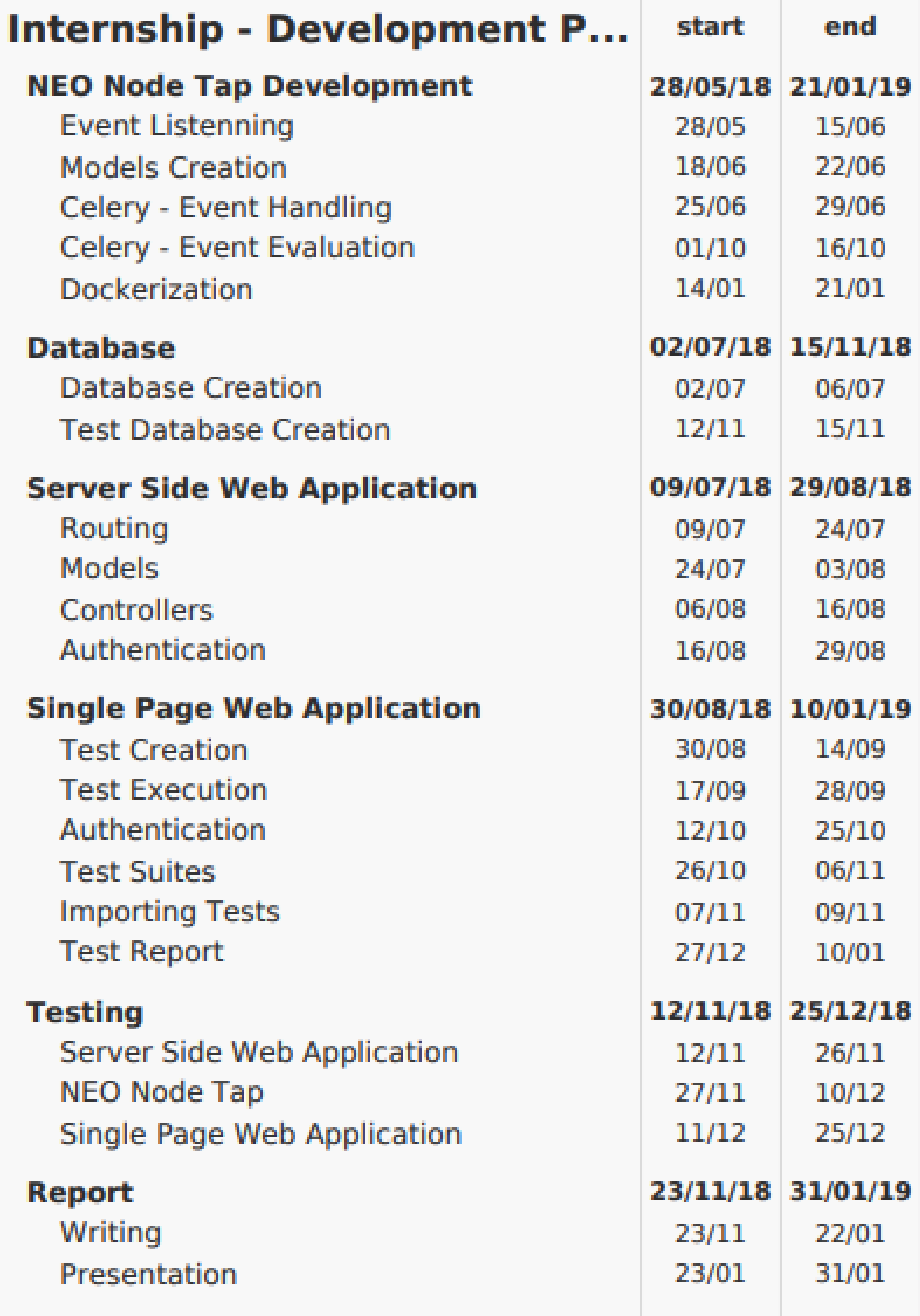
```
➜ neoevents git:(event-requests) ✗ python -m SimpleHTTPServer 5000

Serving HTTP on 0.0.0.0 port 5000 ...
127.0.0.1 - - [01/Jul/2018 17:42:30] code 501, message Unsupported method ('POST')
127.0.0.1 - - [01/Jul/2018 17:42:30] "POST / HTTP/1.1" 501 -
127.0.0.1 - - [01/Jul/2018 17:42:30] code 501, message Unsupported method ('POST')
127.0.0.1 - - [01/Jul/2018 17:42:30] "POST / HTTP/1.1" 501 -
127.0.0.1 - - [01/Jul/2018 17:42:30] code 501, message Unsupported method ('POST')
127.0.0.1 - - [01/Jul/2018 17:42:30] "POST / HTTP/1.1" 501 -
127.0.0.1 - - [01/Jul/2018 17:42:30] code 501, message Unsupported method ('POST')
127.0.0.1 - - [01/Jul/2018 17:42:30] "POST / HTTP/1.1" 501 -
127.0.0.1 - - [01/Jul/2018 17:42:30] code 501, message Unsupported method ('POST')
127.0.0.1 - - [01/Jul/2018 17:42:30] "POST / HTTP/1.1" 501 -
127.0.0.1 - - [01/Jul/2018 17:42:30] code 501, message Unsupported method ('POST')
127.0.0.1 - - [01/Jul/2018 17:42:30] "POST / HTTP/1.1" 501 -
127.0.0.1 - - [01/Jul/2018 17:42:32] code 501, message Unsupported method ('POST')
127.0.0.1 - - [01/Jul/2018 17:42:32] "POST / HTTP/1.1" 501 -
127.0.0.1 - - [01/Jul/2018 17:42:32] code 501, message Unsupported method ('POST')
127.0.0.1 - - [01/Jul/2018 17:42:32] "POST / HTTP/1.1" 501 -
127.0.0.1 - - [01/Jul/2018 17:42:32] code 501, message Unsupported method ('POST')
127.0.0.1 - - [01/Jul/2018 17:42:32] "POST / HTTP/1.1" 501 -
127.0.0.1 - - [01/Jul/2018 17:42:33] code 501, message Unsupported method ('POST')
127.0.0.1 - - [01/Jul/2018 17:42:33] "POST / HTTP/1.1" 501 -
127.0.0.1 - - [01/Jul/2018 17:42:33] code 501, message Unsupported method ('POST')
127.0.0.1 - - [01/Jul/2018 17:42:33] "POST / HTTP/1.1" 501 -
127.0.0.1 - - [01/Jul/2018 17:42:33] code 501, message Unsupported method ('POST')
127.0.0.1 - - [01/Jul/2018 17:42:33] "POST / HTTP/1.1" 501 -
127.0.0.1 - - [01/Jul/2018 17:42:33] code 501, message Unsupported method ('POST')
127.0.0.1 - - [01/Jul/2018 17:42:33] "POST / HTTP/1.1" 501 -
127.0.0.1 - - [01/Jul/2018 17:42:33] code 501, message Unsupported method ('POST')
```

Figure 3.2: The endpoint being called whenever the selected contract broadcasted an event. The requests are being made to a SimpleHTTPServer which was not prepared to handle them.



```
(neoevents-HV_DX-BR) ➜ neoevents git:(event-requests) ✗ python manage.py runnode
[I 180701 16:37:37 LevelDBBlockchain:115] Created Blockchain DB at /Users/tomas/.neopython/Chains/SC2
34
[I 180701 16:37:42 runnode:65] Block 1538042 / 1538042
[I 180701 16:37:42 runnode:92] Everything setup and running. Waiting for events...
[I 180701 16:37:57 runnode:65] Block 1538042 / 1538042
Unhandled error in Deferred:

Unhandled error in Deferred:

Unhandled error in Deferred:

Unhandled error in Deferred:

[I 180701 16:38:12 runnode:65] Block 1538042 / 1540042
Unhandled error in Deferred:

Unhandled error in Deferred:

Unhandled error in Deferred:

[I 180701 16:38:27 runnode:65] Block 1538042 / 1542042
Unhandled error in Deferred:

{'event_type': 'SmartContract.Storage.Get', 'contract_hash': 'b9d7ea3062e6aeeb3e8ad9548220c4ba1361d26
3', 'tx_hash': 'ccfccb4562bc79bed9c47d626369df6d5122dd9a51fa1690859cc2efecfef626', 'block_number': 15
38056, 'event_payload': "['AQC7Bod2LxaRxmLewRrwCA1Nt6AQMWSm28 -> 9303185844969465']", 'execution_succ
ess': False, 'test_mode': False}
{'event_type': 'SmartContract.Storage.Put', 'contract_hash': 'b9d7ea3062e6aeeb3e8ad9548220c4ba1361d26
3', 'tx_hash': 'ccfccb4562bc79bed9c47d626369df6d5122dd9a51fa1690859cc2efecfef626', 'block_number': 15
```

Figure 3.3: The custom event listener connecting to the blockchain and starting listening to events.



Internship - Development P...	start	end
NEO Node Tap Development	**28/05/18**	**21/01/19**
Event Listenning	28/05	15/06
Models Creation	18/06	22/06
Celery - Event Handling	25/06	29/06
Celery - Event Evaluation	01/10	16/10
Dockerization	14/01	21/01
Database	**02/07/18**	**15/11/18**
Database Creation	02/07	06/07
Test Database Creation	12/11	15/11
Server Side Web Application	**09/07/18**	**29/08/18**
Routing	09/07	24/07
Models	24/07	03/08
Controllers	06/08	16/08
Authentication	16/08	29/08
Single Page Web Application	**30/08/18**	**10/01/19**
Test Creation	30/08	14/09
Test Execution	17/09	28/09
Authentication	12/10	25/10
Test Suites	26/10	06/11
Importing Tests	07/11	09/11
Test Report	27/12	10/01
Testing	**12/11/18**	**25/12/18**
Server Side Web Application	12/11	26/11
NEO Node Tap	27/11	10/12
Single Page Web Application	11/12	25/12
Report	**23/11/18**	**31/01/19**
Writing	23/11	22/01
Presentation	23/01	31/01

Figure 3.4: The overall planning of the second semester.

Chapter 4

Requirement Specification

A system's requirements document describes the services provided by the system, as well as the constraints on its operation[37]. In this project, these requirements reflect the needs of NEO smart contract developers for the proposed solution. This chapter details the requirements gathered from the stakeholders, as well as the processes followed to obtain them.

4.1 Stakeholders

Name	Description	Responsibility
Whitesmith	Software consultant and product studio. Launched the Blocksmith spinoff and are mentoring the project. Hired the intern.	Monitoring the project progress. Funding. Committing the necessary resources to ensure the project succeeds. Providing assistance to the intern.
Blocksmith	Blockchain development studio. Requested the project.	Ensuring that the project fits business objectives. Making their detailed requirements known. Ensuring project quality.
System Actors	NEO blockchain developers who will be the end-users of the final product.	None

Table 4.1: Stakeholders summary

4.2 System Actors

In order to clearly define the system's requirements, it is important to understand who are the actors in this system. The actors are system stakeholders, as they will be the ones using the system after its release. Reviewing the intern's code in code reviews.

4.2.1 Non-Authenticated User

Non-Authenticated User (NAU)	
Description	A human entity which will have access to a subset of the system functionalities. This entity has not yet authenticated itself in the system, but may, or may not, already have a registered account.
Assumptions	It is assumed that this entity is a NEO smart contract developer, and that, as such, has at least a basic knowledge of how to interact with a web system and perform actions such as creating an account and logging in. It is also assumed that this entity has at least a basic understanding of the NEO blockchain and of smart contract development.
Expectations	It is expected that this entity will use the application to attempt to deploy and test a smart contract. It is expected that this entity will register an account and turn into an Authenticated User, in order to have access to extended functionalities.

Table 4.2: System actor - Non-Authenticated User (NAU)

4.2.2 Authenticated User

Authenticated User (AU)	
Description	A human entity with a registered account in the system and which has logged-in in the system, authenticating itself. This entity will have access to extended functionalities, such as saving a test suit in the system.
Assumptions	As with NAUs, it is assumed that this entity is a NEO smart contract developer, and that, as such, has at least a basic knowledge of how to interact with a web system and a basic understanding of the NEO blockchain and of smart contract development.
Expectations	It is expected that this entity will use the application to deploy and test smart contracts. It is also expected that this entity will be able to save its progress by saving current test cases and test suites in the system.

Table 4.3: System actor - Authenticated User (AU)

4.3 Requirements Gathering

In order to understand the needs and difficulties of smart contract developers, and also to validate the need for this tool, a survey was made and distributed to Blocksmith blockchain developers, as well as circulated in a NEO CoZ discussion channel. The survey, which can be seen in Appendix A, only aimed to collect answers from smart contract developers, regardless of the platform where these contracts were run, and was answered by 19 developers.

This survey revealed that, although most developers were successful in developing a smart contract, they faced difficulties which seem to be a direct consequence of the lack of maturity of the technology and its development tools. The main difficulties identified by these developers were:

- Lack of documentation, with most tutorials on contract development scattered along random blog posts;

- Deprecated documentation as blockchain platforms and SDKs keep evolving at a fast pace;
- Lack of debugging tools;
- Lack of tools to deploy and test contracts;
- Difficulty in installing a development environment;

These developers were also asked for their feedback on what made them succeed in developing their contracts, with many of them stating that the support of the development community was one of the key reasons why they were able to succeed, as well as tools and tutorials that were written by members of this community.

To further understand the needs of a smart contract developer, the intern was involved in a project with other members of Blocksmith, which aimed to create a DApp to enter the NEO DApps competition[28]. This DApp consisted of a decentralised curriculum network, where a user could see its curriculum validated by an entity, and was called NEO Vitae. This DApp ended up being one of the honourable mentions, winning an Award of Merit prize which amounted to 500 NeoGas (worth roughly 13,000 euros at the time of the competition)[15].

Finally, a study of the current state of the art of NEO and Ethereum smart contract development tools was done, which aimed not only to understand the shortcomings of these tools but also to identify any open-source tools with interesting functionalities. This study of the state of the art was previously presented in this report (see section 2.5).

4.4 Functional Requirements

In this section, the system's functional requirements are described through user stories. User stories are often used in Agile software development to capture a description of a software feature from the end-user perspective, with each user story describing the actors for which they apply as well as what they want from the system and why. User stories can be added or removed during the development phase to better adapt to the proposed solution or to increase or decrease the solution complexity. To easily understand where each story fits in the system, stories are assigned to larger bodies of work, called "Epics"[35]. The epics to be implemented in the solution are:

- Landing page - This epic is concerned with the first contact the user has with the application;
- Authentication - Stories that deal with user authentication should be included in this epic;
- Private Network - All the stories that are concerned with functionalities that include the interaction of the user with the private network should be included in this epic;
- Testing - All of the stories related to smart contract testing should belong to this epic;

To prioritise user stories inside each epic, the MoSCoW technique was used, which gives each user story one of four possible classifications[23]:

- **Must Have:** These features are indispensable for the functionality of the system and for the required objectives to be achieved in the given timebox. Without these features, the system will not be an adequate solution.

- **Should Have:** These features can be as important as the "Must Have" features, but are not critical for this timebox. These are features that will eventually be on the solution but are not guaranteed to be delivered in the first iteration.

- **Could Have:** These are desired features that will be included in the solution if time and resources will permit it. They usually represent features that usually will increase user experience for little development cost.

- **Won't Have:** These features were agreed by the stakeholders not to be critical. These are features that will not be included in a first delivery, but that can be included in a later version of the product.

ID	Actor(s)	Description	Depends on	MoSCoW
US1	AU AUG NAU	As a user, I want to be able to visit a landing page so that I can learn more about the features provided by the application		Must Have
US2	AU AUG NAU	As a user, I want to be able to easily access an authentication form from the landing page, so I can create a new account or login with an existing account	US1	Must Have

Table 4.4: User Stories - Landing Page

ID	Actor(s)	Description	Depends on	MoSCoW
US3	NAU	As a user, I want to be able to create an account using my email and a password as credentials, so that I can access extra system features	US2	Must Have
US4	NAU	As a non-authenticated user, I want to be able to login into my account so that I can authenticate myself and save data under my account	US2, US3	Must Have
US5	AU AUG	As an authenticated user, I want to be able to delete my account, so that I remove all of my information from the system	US3	Should Have
US6	AU	As an authenticated user, I want to be able to confirm my email so that I can prove that I own my email and receive notifications from the system	US3	Could Have
US7	AU	As an authenticated user, I want to be able to reset my password so that I can change it for security reasons	US3	Could Have
US8	NAU	As a non-authenticated user, I want to be able to authenticate myself using my Github credentials so that I can log in using my Github credentials and use the Github integration		Won't have
US9	AU	As an authenticated user, I want to be able to link my Github account to my already existing account so that I can log in with them and use the Github integration	US3, US8	Won't Have

Table 4.5: User Stories - Authentication

ID	Actor(s)	Description	Depends on	MoSCoW
US10	AU AUG NAU	As a user, I want to be able to deploy my NEO smart contract code to a private network provided by the website, so that I don't have to set up my own private network		Must Have
US11	AU AUG NAU	As a user, I want to be able to make JSON RPCs to the private network so that I can use blockchain functionalities such as smart contract invocation	US10	Should Have
US12	AU AUG NAU	As a user, I want to have an online editor so that I can create or update my smart contract code		Should Have
US13	AU AUG NAU	As a user, I want to be able to compile my smart contract to NEO AVM code	US12	Should Have
US14	AU AUG NAU	As a user, I want to be able to be able to use all of the tool functionalities on the official test network, so I can test my smart contracts on a bigger network	US10	Won't Have

Table 4.6: User Stories - Private Network

ID	Actor(s)	Description	Depends on	MoSCoW
US15	AU AUG NAU	As a user, I want to be able to create a test case, so that I can use it to test my smart contract		Must Have
US16	AU AUG NAU	As a user I want to be able to set the test input on my test case, so I can test specific methods of my smart contract	US15	Must Have
US17	AU AUG NAU	As a user, I want to be able to define the expected output of my test case, so that the tool can evaluate if my smart contract passed the test case.	US15, US16	Must Have

Table 4.7: User Stories - Testing (1)

ID	Actor(s)	Description	Depends on	MoSCoW
US18	AU AUG NAU	As a user, I want to be able to run a test case so that I can unit test my smart contract	US10, US15	Must Have
US19	AU AUG NAU	As a user, I want to be able to see a detailed result of the test case so that I can understand why is my smart contract passing or failing a test	US15	Must Have
US20	AU AUG	As an authenticated user, I want to be able to save a test case that I created, so that I can use it later or on another smart contract		Must Have
US21	AU AUG	As an authenticated user, I want to be able to set a name and a description for my saved test cases so that they are easier to manage	US20	Must Have
US22	AU AUG	As an authenticated user, I want to be able to edit a test case that I saved, so that I can update it when needed	US20, US21	Must Have
US23	AU AUG	As an authenticated user, I want to be able to delete a test case that I saved, so that I can remove it from the system	US20	Must Have
US24	AU AUG NAU	As a user, I want to be able to create a test suite for my smart contract so that I can test more than one test case at a time	US15	Must Have
US25	AU AUG NAU	As a user, I want to be able to run a test suite for my smart contract so that I can run more than one unit test at a time	US10, US18, US24	Must Have
US26	AU AUG NAU	As a user, I want to be able to see the detailed result of running a test suite, so I can see in which cases my smart contract is passing or failing	US19, US25	Must Have

Table 4.8: User Stories - Testing (2)

ID	Actor(s)	Description	Depends on	MoSCoW
US27	AU AUG	As an authenticated user, I want to be able to save a test suite that I created so that I can use it later or run it on another smart contract	US24	Must Have
US28	AU AUG	As an authenticated user, I want to be able set a name and a description for my saved test suites so that they are easier to manage.	US27	Must Have
US29	AU AUG	As an authenticated user, I want to be able to delete a test suite that I saved, so that I can remove it from the system	US27	Must Have
US30	AU AUG	As an authenticated user, I want to be able add a saved test case to a saved test suite, so I can create new test suites from existing test cases	US20, US27	Must Have
US31	AU AUG	As an authenticated user, I want to be able to remove a test case from a test suite, so that I can remove non necessary test cases	US30	Should Have
US32	AU AUG NAU	As a user, I want to be able to have access to different wallets, so that I can test my smart contract with more than one wallet		Could Have
US33	AUG	As an authenticated user with a Github account, I want to be able to access code from a Github repository that I have access to, so that I can retrieve smart contract code	U8, US9, US12	Won't Have
US34	AU AUG NAU	As a user, I want to be able to set breakpoints in my smart contract	US12, US13	Won't Have

Table 4.9: User Stories - Testing (3)

4.5 Non-Functional Requirements

4.5.1 Technical Constraints

The intern was required to attempt to integrate his solution with NeoCompiler Eco development tool:

- **NFR1 -** The platform must be integrated with the NeoCompiler Eco development tool.

To make the integration of this solution with the NeoCompiler Eco development tool easier, the intern must use the same stack of technologies used in that tool. As such, the frontend client must be developed in Angular.js, whereas the backend server must be developed in Node.js, using the Express.js framework.

- **NFR2 -** The frontend client must be developed using Angular.js.
- **NFR3 -** The back-end server must be developed in Node.js, using the Express.js framework.

4.5.2 Quality Attributes

Whereas functional requirements define what a system should do, quality attributes define how the system shall behave at all times. As they define the system behaviour, the system must be designed around them, and they must be taken into consideration in every step, from the solution proposal until the delivery of the system.

Security

As every data sent to the blockchain is, by definition, public and accessible to all of the existent nodes, there are no concerns to be had regarding the possibility of an attacker gaining access to smart contract code or test cases. However, user personal information, such as his password, must be kept safe from attackers.

- **NFR4 -** The application must not keep user passwords in plaintext. Only a hashed and salted representation of the password can be stored by the application.

Modularity

The lack of maturity but fast pace of the blockchain technology, and NEO in particular, makes it that new features are added daily to this blockchain and its smart contracts.

- **NFR5 -** To ensure future compatibility with any changes made to smart contract properties, this solution must follow a modular design, in order to quickly remove and update deprecated components.

Compatibility

Being a web application, it is important that the front-end client is compatible with popular web browsers.

- **NFR6 -** The front-end desktop client must be compatible with the following desktop web browsers: Google Chrome (v67.0 or newer), Mozilla Firefox (v61.0 or newer) and Apple Safari (v11.1.1 or newer).

Chapter 5

Architecture

In order to guarantee that both the project's functional and non-functional requirements are obeyed, it was essential to define an architecture that addresses them, which is described in this chapter. The project's first technical constraint, NFR1, states that the tool to be developed had to be integrated with the NeoCompiler Eco, a development tool for NEO smart contracts that provides the user with a remote private blockchain and the possibility of deploying and invoking smart contracts.

As NeoCompiler Eco is developed following a micro-services architecture deployed in Docker containers (as shown in Figure 5.2), the intern planned to use Docker to encapsulate the NEO Node Tap component and make it easier to integrate it with the tool.

5.1 Context Diagram

The application System Context Diagram, shown in Figure 5.1, depicts a high-level view of how the system interacts with the environment.

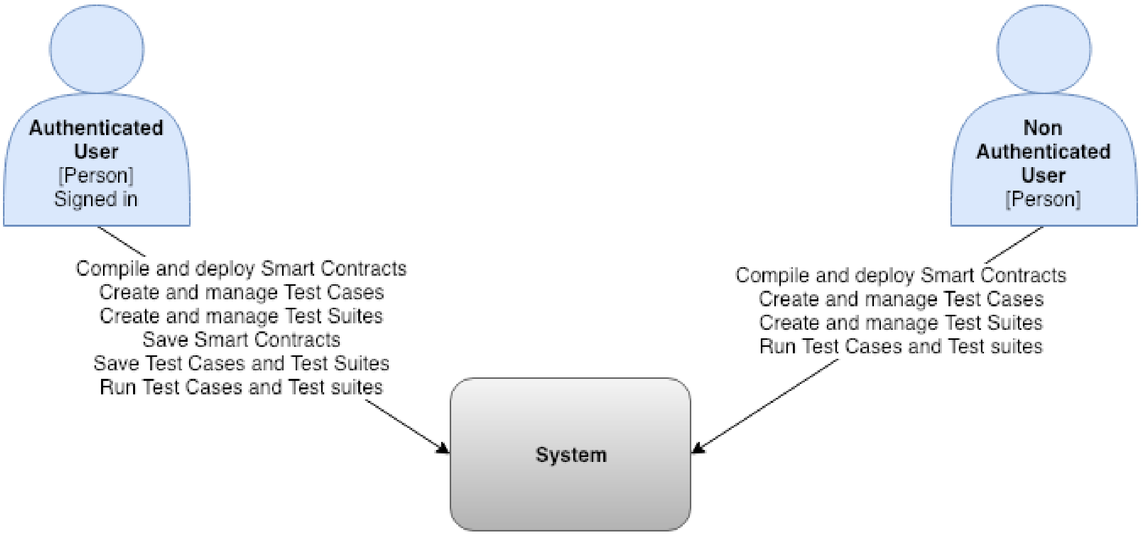

Authenticated User
[Person]
Signed in
Non Authenticated User
[Person]
Compile and deploy Smart Contracts
Create and manage Test Cases
Create and manage Test Suites
Save Smart Contracts
Save Test Cases and Test Suites
Run Test Cases and Test suites
Compile and deploy Smart Contracts
Create and manage Test Cases
Create and manage Test Suites
Run Test Cases and Test suites
System

Figure 5.1: The System Context Diagram, showing the system interactions with the environment.

As seen in the diagram, two actors interact with the system. These actors, introduced in Chapter 4 - Requirements, are the non-authenticated user and the authenticated user.

5.2 Containers Diagram

With the Containers Diagram, it is possible to go down a level of abstraction and see how different containers inside the system interact between them, how different system responsibilities are distributed and how the new system is integrated with the existing NeoCompiler Eco.

In the diagram depicted in Figure 5.2 the following containers are represented:

- Single Page Web Application - Provides all tool features to the users via their web browsers. Makes requests to the server side application. Uses the Neon.js library (described in Section 6.4 - Single Page Web Application) to interact with the NEO blockchain through the NEO Python RPC Node.
- Server-Side Web Application - Delivers and updates all the content present in the front-end application. Uses the different compilers to compile smart contract code to machine code. Reads and writes data to the Database container.
- Database - Stores application data such as test cases, test suites and user data.
- NEO Python Compiler - Compiles Smart Contracts written in Python.
- NEO C# Compiler - Compiles Smart Contracts written in C#.
- NEO GO Compiler - Compiles Smart Contracts written in Golang.
- NEO Java Compiler - Compiles Smart Contracts written in Java.
- NEO Python RPC Node - Python based P2P NEO client node that offers a RPC interface that allows the deployment of NEO smart contracts in the NEO blockchain, among other operations.
- Database - Stores user authentication data, references to smart contracts, tests, and test suites.
- NEO Node Tap - NEO node that listens to events in the network and relays them to the Server-side Web application.
- NEO Blockchain - A container encapsulating a network of four NEO consensus nodes that validate NEO blockchain data.

The green containers presented in this (Figure 5.2) and in the following diagrams (Figure 5.3 and Figure 5.4), are containers that are already a part of the NeoCompiler Eco system and which will not need to be altered. Blue containers already exist in the NeoCompiler Eco system but needed to be altered in order to perform new functionalities. White containers are containers that do not exist yet and needed to be developed.

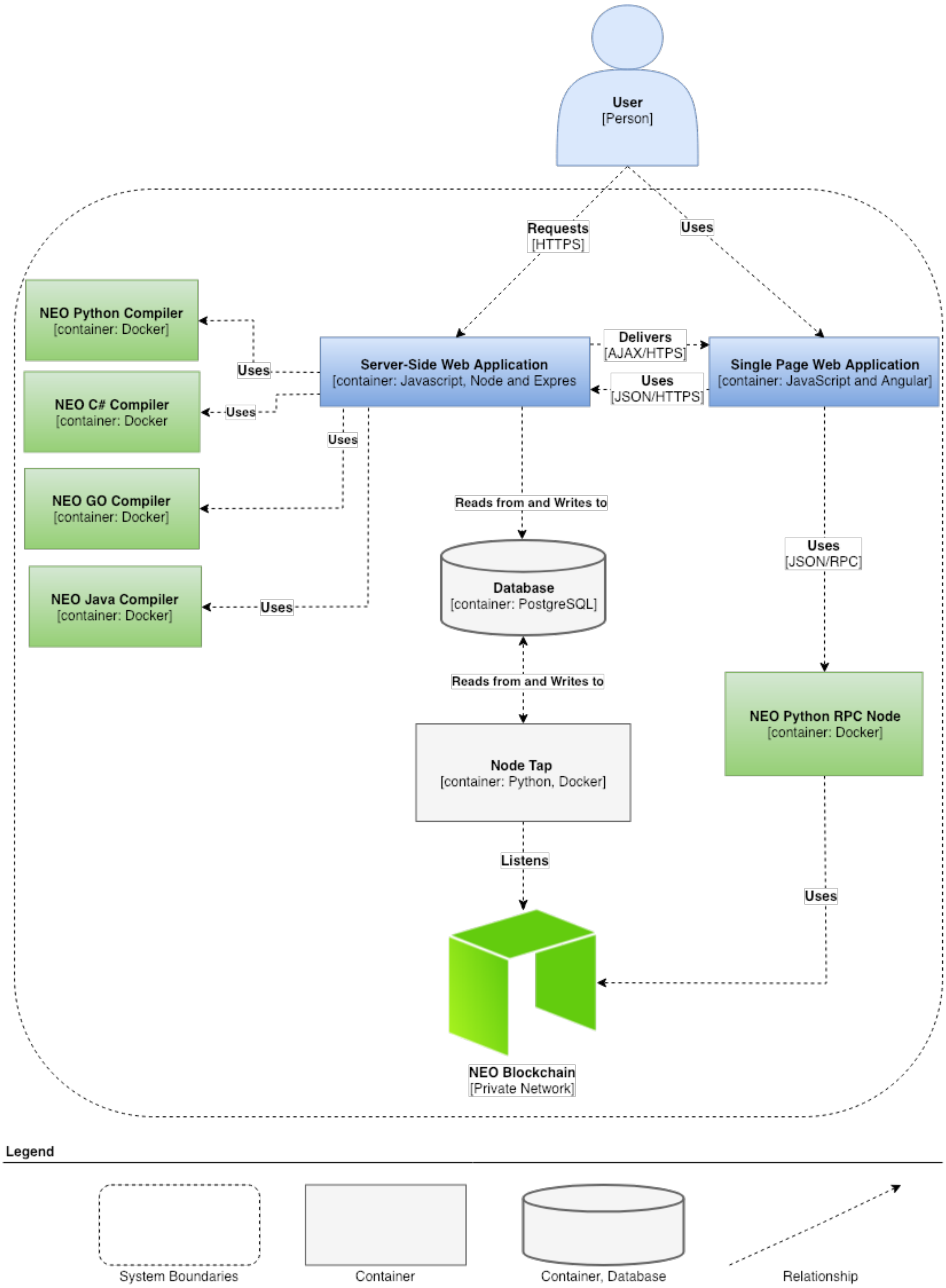


Figure 5.2: The Containers Diagram of the proposed solution.

5.3 Components Diagram

5.3.1 Server Side Web Application

As seen in Figure 5.3, the Server Side Web Application container is composed by three applications:

- appHttp.js - This application serves the static single page application to the client. An API that handles all operations concerned with authentication, test cases and test suites was added to this application, with its implementation being described in Section 6.3 - Server Side Web Application.
- appCompiler.js - This application receives requests from the Single Page Application (SPA) with smart contract code to be compiled. It then uses the correspondent NEO compiler and returns the compiled code to the client.
- appEcoServices.js - This application uses web sockets to provide real-time information on the status of the different services, such as the RPC node status and the number of users active on the platform.

The `appHttp.js` application is the only one that was changed, as it was the one serving the single page application to the client. The other two applications are simple applications, shipped with the NeoCompiler Eco.

5.3.2 NEO Node Tap

The NEO Node Tap components diagram, as shown in Figure 5.4, zooms in on the NEO Node Tap container and offers a view of the existing components inside it. In this diagram, it is possible to see how these components relate between them and how they interact with other containers. In this diagram, the external services and the containers that compile smart contracts are not presented in order to reduce space.

The following components are present in this diagram:

- NEO Node - A node based on the NEO Python node implementation, which listens to events broadcasted by smart contracts in the blockchain and puts them in the Queue A component.
- Queue A - A queue that temporarily stores events received from the NEO Node component.
- Worker A - A worker that removes an event from the top of the queue A, queries the database to verify if it belongs to a contract currently being tested and in the case of a positive answer puts the event in queue B.
- Queue B - A queue that receives events that belong to smart contracts being currently tested and are waiting to be processed.
- Worker B - A worker that processes and evaluates events from queue B and updates the correspondent test in the database.

The implementation of the NEO Node Tap is discussed on Section 6.2 - NEO Node Tap.

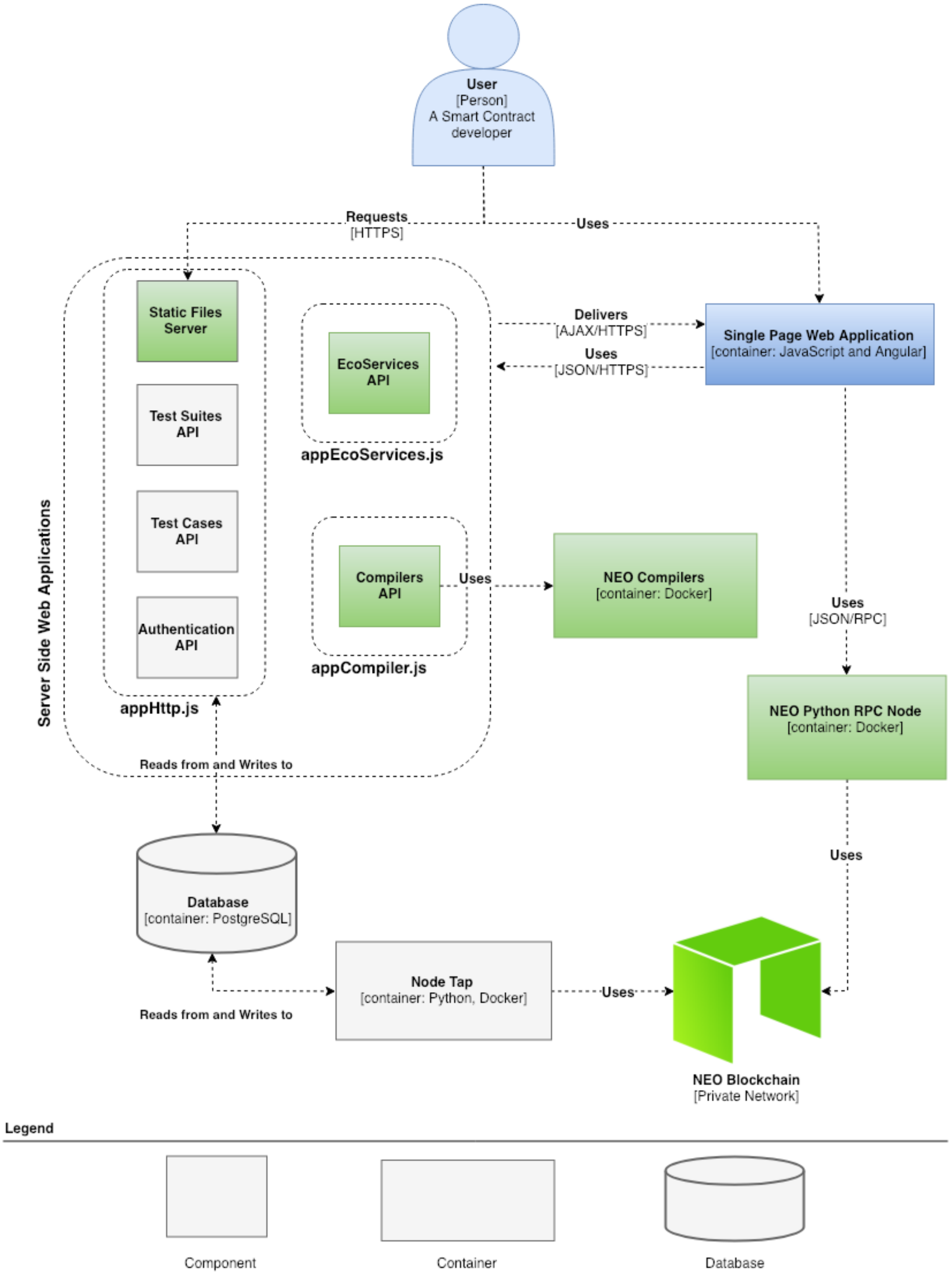


Figure 5.3: A lower-level view of the Server Side Web Application, showing their components and how they interact with the rest of the system.

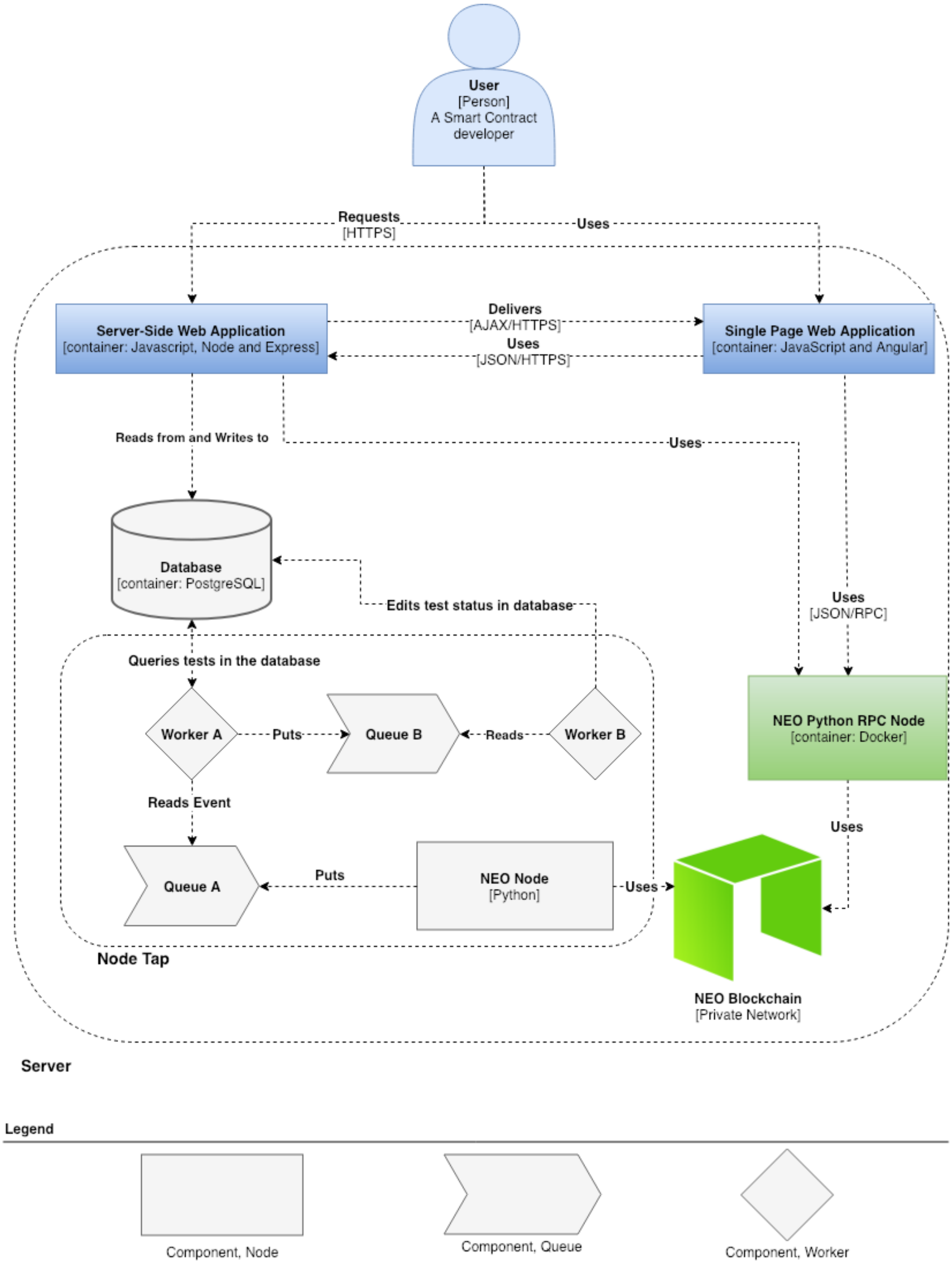


Figure 5.4: A lower-level view of Node Tap, showing its components and how they interact with the rest of the system.

5.4 Entity Relationship Diagram

To be able to visualise which objects are stored in the database and how they relate between them, an entity-relationship diagram, depicted in Figure 5.5, was drawn. The database acts as a bridge between the NEO Node Tap and the Web Application Server, as the server updates the database with the currently existing test cases as the NEO Node Tap queries the database to see which events it should update. The following data-model objects are represented in the diagram:

- User - This object stores data on the users authenticated in the application.
- Test Case - This object holds data of test cases created in the application. Also holds data on the required inputs necessary to call the smart contract with the required test input data. If the user chooses to save any test case, the test case is bound to the user via a foreign key correspondent to the owner ID, forming a one-to-many relationship, where a user can have many test cases. If the user assigns the test case to a test suite, the test suite foreign key is saved in the test case, forming a one-to-many relationship, where a test suite can have multiple test cases.
- Test Suite - It stores test suites data, such as name and description. If a user opts to save any test suite, the test suite is bound to the user via a foreign key correspondent to the user ID, forming a one-to-many relationship, where a user can have multiple test suites.

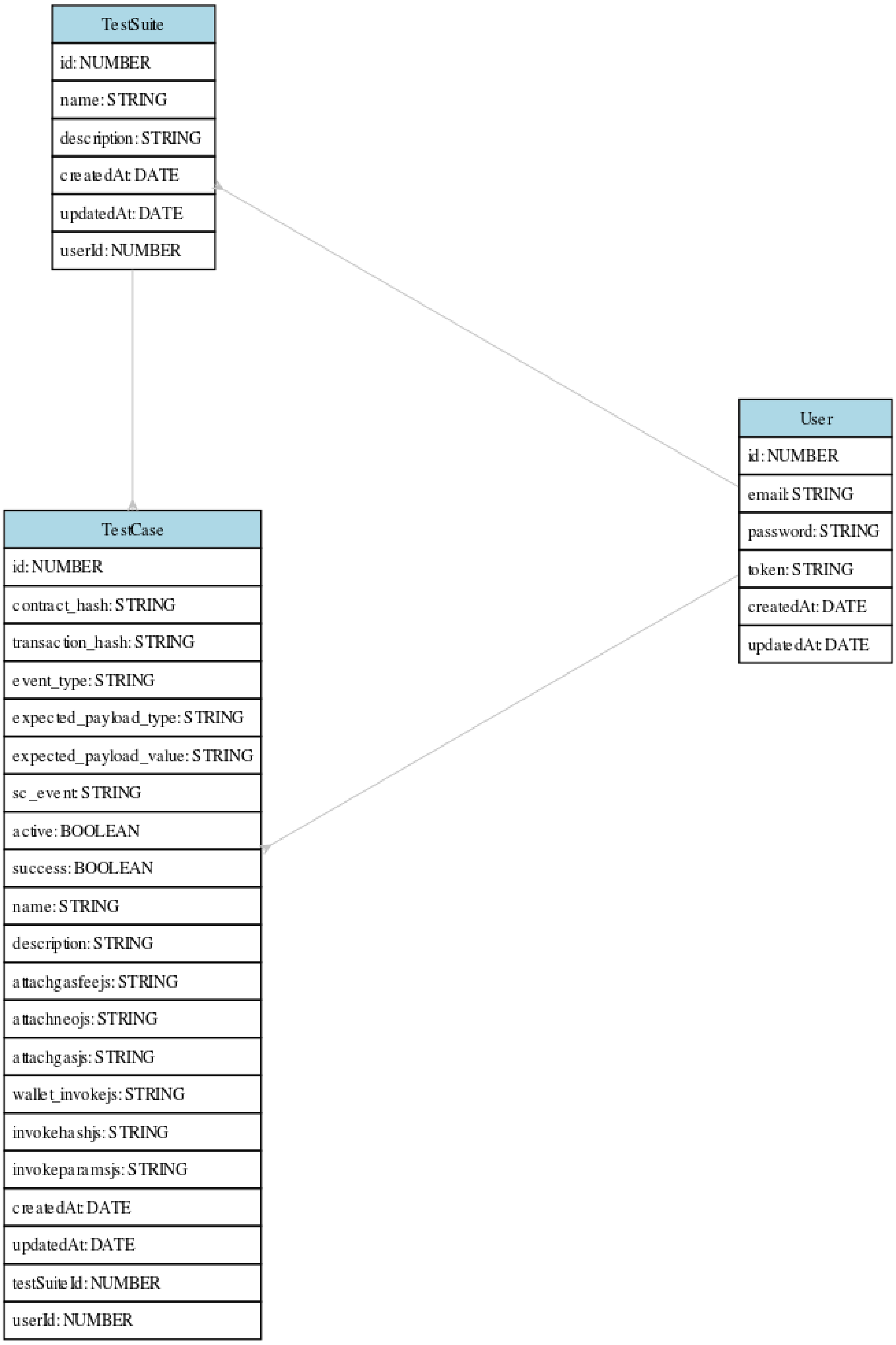


Figure 5.5: The entity-relationship diagram of the solution.

Chapter 6

Implementation

This chapter describes the implementation process of the different components of the proposed solution and their integration. A shortened project tree, depicting the project structure and the organisation of the project files and folders, can be consulted in Appendix C - Project Tree, which can be useful for the reader in the following sections.

As discussed in Section 3.4 - Planning, the first step in the development of the solution was to create a NEO Node that could listen and relay events emitted by smart contracts running on the NEO blockchain it was listening to. As such, this module was named NEO Node Tap, as it acts as a wiretap, and is described in detail in section 6.2 - NEO Node Tap.

After confirming that listening to Blockchain events in real time was possible, the next step was to start developing the test harness that enables users to deploy a smart contract on a private network and unit test it. As shown in the study of the state of the art of smart contract development tools (section 2.5 - Smart Contract Development Tools), there is already an open-source tool, NeoCompiler Eco, that offers some of these features for NEO smart contracts. This tool implements an online smart contract editor, a compiler for the different NEO smart contract programming languages and the ability for developers to deploy the compiled smart contract to a private network. So, the intern decided to fork this existing tool and implement the testing and authentication features on top of it. This implementation is described in section 6.3 - Server Side Web Application and in section 6.4 - Single Page Web Application.

6.1 Database

In order to store and relate user and test data, PostgreSQL[1], an open-source relational database, was used. PostgreSQL was chosen, not only for being a relational database but also for being an ACID-compliant and transactional database, as access to the database happens concurrently between the NEO Node Tap and the backend API, with a reasonable possibility of attempts to update the same entry on the database being made at the same time.

An entity-relationship (ER) diagram of the database can be seen in Figure 5.5 of section 5.4 - Entity Relationship Diagram, visually representing how the different models relate between themselves.

[1] `https://www.postgresql.org/`

6.2 NEO Node Tap

NEO Node Tap is a simplified version of a full NEO-Python node that connects to a NEO blockchain network and to the application database. This module was developed in Python 3.6, and uses the following dependencies:

- Pip[2], a Python package manager;
- Virtualenv[3], that enables the creation of isolated Python environments;
- Neo-Python[4], the original NEO-Python SDK and full-node;
- Celery[5], a distributed task queue that asynchronously processes tasks, and which in this application uses Redis as a message broker;
- Redis[6], a key-value in-memory database;
- SQLAlchemy[7], a Python Object Relational Mapper (ORM).

A diagram of the NEO Node Tap flow can be seen in Figure 6.1.

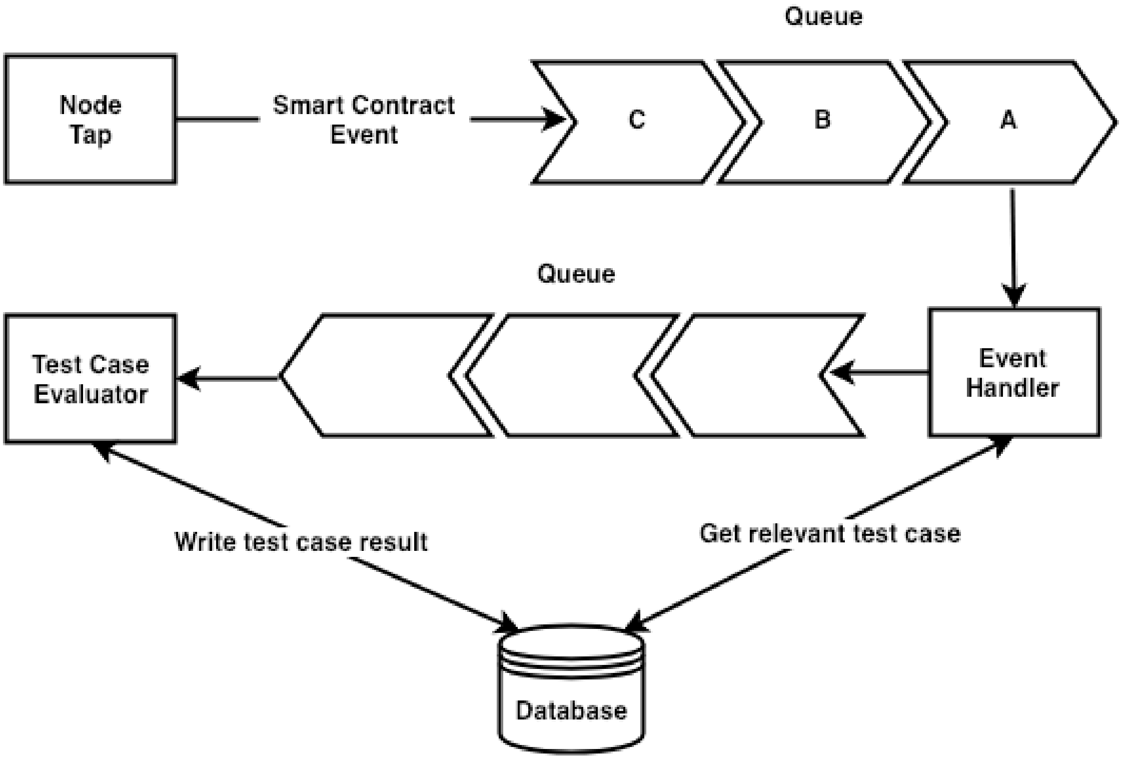


Figure 6.1: Flow diagram of the NEO Node Tap.

When a new block is added to the blockchain, the node runs the smart contract transactions contained on the block and listens for events emitted during this execution. If an event is captured, it is then passed to a celery task that handles it. The task queries the

[2] `https://github.com/pypa/pip`
[3] `https://github.com/pypa/virtualenv/`
[4] `https://github.com/CityOfZion/neo-python`
[5] `https://github.com/celery/celery/`
[6] `https://github.com/antirez/redis`
[7] `https://github.com/sqlalchemy/sqlalchemy`

database for any test case that might be related to that event transaction. If no relevant test case is found, the task discards the event. However, if a test case is found, the event is sent to other task, along with the relevant test case ID, which then evaluates the test case by comparing the listened event with the expected test case output, saving on the database the outcome of the test case as well as data related with the listened event.

The smart contract events mentioned in this report are instances of the `SmartContractEvent` object and occur when the node is processing blockchain blocks[8].

6.2.1 Event Listening

Event listening is started by running the `runnode.py` Python script on the `neo_node_tap` folder (see Appendix C - Project Tree). As seen in Listing 6.1, the script starts by loading the private network settings, setting up the blockchain database (the `LevelDB` database) and then starts a new node (see the code in Listing 6.1 from line 1 to line 9).

```
 1 def main(**options):
 2     settings.setup_privnet()
 3 
 4     # Setup the blockchain
 5     blockchain = LevelDBBlockchain(settings.chain_leveldb_path)
 6     Blockchain.RegisterBlockchain(blockchain)
 7     dbloop = task.LoopingCall(Blockchain.Default().PersistBlocks)
 8     dbloop.start(.1)
 9     NodeLeader.Instance().Start()
10 
11     # Setup Notifications Database
12     NotificationDB.instance().start()
13 
14     # Disable smart contract events for external smart contracts
15     settings.set_log_smart_contract_events(False)
16 
17     # Start a thread with custom code
18     d = threading.Thread(target=custom_background_code)
19     d.setDaemon(True)  # daemonizing the thread will kill it when the main
        thread is quit
20     d.start()
21 
22     # Run all the things (blocking call)
23     logger.info("Everything setup and running. Waiting for events...")
24     reactor.run()
25     logger.info("Shutting down.")
26     logger.info("Closing databases...")
27     NotificationDB.close()
28     Blockchain.Default().Dispose()
29     NodeLeader.Instance().Shutdown()
30 
31 
32 if __name__ == "__main__":
33     main()
```

Listing 6.1: runnode.py - Setting the blockchain and running a blockchain node

After starting the blockchain node, a thread is launched (see line 18 in Listing 6.1) that monitors and logs the state of the node. This thread is also responsible for restarting the node if the node seems to be stuck on a block. This was a recurring problem in older versions of NEO-Python. The code run by this thread can be seen in Listing 6.2.

[8] `https://neo-python.readthedocs.io/en/latest/neo/SmartContract/smartcontracts.html#event-types`

```python
def custom_background_code():
    """ Custom code run in a background thread. Prints the current block height.
    Raises an exception every 5 minutes the block count is blocked and restarts the node
    """
   if settings.is_mainnet:
        network = 'Mainnet'
    elif settings.is_testnet:
        network = 'Testnet'
    else:
        network = 'Privnet'
    counter = 0
    previous_block_count = 0

    while True:
        try:
            logger.info(f"Block {str(Blockchain.Default().Height)} / {str(Blockchain.Default().HeaderHeight)}")
            logger.info(f"Connected to {len(NodeLeader.Instance().Peers)} peers.")
            for peer in NodeLeader.Instance().Peers:
                logger.info(peer.Address)
            logger.info("\n")
            if previous_block_count == Blockchain.Default().Height:
                counter += 1
                if counter % 5 == 0:
                    error = '{} Node is blocking'.format(network)
                    message = 'Node is blocking at block: {}. Connected peers: {}. Attempting restart.'
                    message = message.format(Blockchain.Default().Height, len(NodeLeader.Instance().Peers))
                    raise NodeBlockingException(error, message)
            else:
                previous_block_count = Blockchain.Default().Height
                counter = 0
        except Exception as e:
            logger.warning(e)
            NodeLeader.Instance().Shutdown()
            NodeLeader.Instance().Start()
            counter = 0
        sleep(60)
```

Listing 6.2: runnode.py - Code that monitors and handles the node state

The code in Listing 6.3 shows how new events occurring on a block are delivered to the Celery workers. The `call_on_event()` function receives the new event and creates a new `dictionary` data type to hold the smart contract event, as data passed to Celery workers needs to be serializable, and the original `SmartContractEvent` data type is not serializable.

```python
from logzero import logger
from neo.Core.Blockchain import Blockchain
from neo.EventHub import events, SmartContractEvent
from neo.Implementations.Blockchains.LevelDB.LevelDBBlockchain import LevelDBBlockchain
from neo.Implementations.Notifications.LevelDB.NotificationDB import NotificationDB
from neo.Network.NodeLeader import NodeLeader
from neo.Settings import settings
from twisted.internet import reactor, task

from exceptions import NodeBlockingException
from tasks import handle_event
from tasks import app as celery_app

__all__ = ('celery_app',)

@events.on(SmartContractEvent.RUNTIME_NOTIFY)
@events.on(SmartContractEvent.RUNTIME_LOG)
@events.on(SmartContractEvent.EXECUTION_SUCCESS)
@events.on(SmartContractEvent.EXECUTION_FAIL)
@events.on(SmartContractEvent.EXECUTION_INVOKE)
@events.on(SmartContractEvent.VERIFICATION_FAIL)
@events.on(SmartContractEvent.VERIFICATION_SUCCESS)
@events.on(SmartContractEvent.STORAGE_DELETE)
@events.on(SmartContractEvent.STORAGE_GET)
@events.on(SmartContractEvent.STORAGE_PUT)
@events.on(SmartContractEvent.CONTRACT_CREATED)
@events.on(SmartContractEvent.CONTRACT_DESTROY)
@events.on(SmartContractEvent.CONTRACT_MIGRATED)
def call_on_event(sc_event):
    """ Is called whenever an event occurs and puts event into queue.

    Event types can be found at:
    https://neo-python.readthedocs.io/en/latest/neo/SmartContract/
        smartcontracts.html#event-types
    """
    if settings.is_mainnet:
        network = 'mainnet'
    elif settings.is_testnet:
        network = 'testnet'
    else:
        network = 'privnet'

    try:
        event_data = {
            'event_type': sc_event.event_type,
            'contract_hash': str(sc_event.contract_hash),
            'tx_hash': str(sc_event.tx_hash),
            'block_number': sc_event.block_number,
            'event_payload': sc_event.event_payload.ToJson(),
            'execution_success': sc_event.execution_success,
            'test_mode': sc_event.test_mode,
            'extra': {'network': network}
        }
        logger.info(event_data)
        handle_event.delay(event_data)
    except Exception as e:
        logger.warning(e)
```

Listing 6.3: runnode.py - Sending the smart contract event to Celery

6.2.2 Models

To be able to fetch and edit test cases from the database, a database connection, and a test case model, both handled by the SQLAlchemy ORM, were defined, as seen in Listing 6.4.

```python
from sqlalchemy import create_engine, Column, String, Integer, Sequence,
    Boolean, DateTime
from sqlalchemy_utils.types.choice import ChoiceType
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

from settings import database_url
from datetime import datetime

Base = declarative_base()


class TestCase(Base):
    DATA_TYPES = [
        ('Signature', 'Signature'),
        ('Boolean', 'Boolean'),
        ('Integer', 'Integer'),
        ('Hash160', 'Hash160'),
        ('Hash256', 'Hash256'),
        ('ByteArray', 'ByteArray'),
        ('PublicKey', 'PublicKey'),
        ('String', 'String'),
        ('Array', 'Array'),
        ('InteropInterface', 'InteropInterface'),
        ('Void', 'Void')
    ]

    EVENT_TYPES = [
        ('SmartContract.Runtime.Notify', 'SmartContract.Runtime.Notify'),
        ('SmartContract.Runtime.Log', 'SmartContract.Runtime.Log'),
        ('SmartContract.Execution.*', 'SmartContract.Execution.*'),
        ('SmartContract.Execution.Invoke', 'SmartContract.Execution.Invoke'),
        ('SmartContract.Execution.Success', 'SmartContract.Execution.Success'),
        ('SmartContract.Execution.Fail', 'SmartContract.Execution.Fail'),
        ('SmartContract.Verification.*', 'SmartContract.Verification.*'),
        ('SmartContract.Verification.Success', 'SmartContract.Verification.Success'),
        ('SmartContract.Verification.Fail', 'SmartContract.Verification.Fail'),
        ('SmartContract.Storage.*', 'SmartContract.Storage.*'),
        ('SmartContract.Storage.Get', 'SmartContract.Storage.Get'),
        ('SmartContract.Storage.Put', 'SmartContract.Storage.Put'),
        ('SmartContract.Storage.Delete', 'SmartContract.Storage.Delete'),
        ('SmartContract.Contract.*', 'SmartContract.Contract.*'),
        ('SmartContract.Contract.Create', 'SmartContract.Contract.Create'),
        ('SmartContract.Contract.Migrate', 'SmartContract.Contract.Migrate'),
        ('SmartContract.Contract.Destroy', 'SmartContract.Contract.Destroy')
    ]

    __tablename__ = 'test_cases'
    id = Column(Integer, Sequence('test_case_id_seq'), primary_key=True)
    contract_hash = Column(String(128))
    transaction_hash = Column(String(128))
    event_type = Column(ChoiceType(EVENT_TYPES))
    expected_payload_type = Column(ChoiceType(DATA_TYPES))
    expected_payload_value = Column(String(512))
    sc_event = Column(String(1024))
    active = Column(Boolean, default=True)
    success = Column(Boolean, default=None)
    createdAt = Column(DateTime, default=datetime.now)
    updatedAt = Column(DateTime, default=datetime.now)

    def __repr__(self):
        return "<TestCase(test_id='%s' contract_hash='%s' contract_hash='%s' event='%s' expected_payload_type='%s " \
            "expected_payload_value='%s' active='%s')>" % (
                self.id, self.contract_hash, self.transaction_hash, self.event_type, self.expected_payload_type,
                self.expected_payload_value, self.active)

engine = create_engine(database_url)
Session = sessionmaker(bind=engine)
engine.dispose()
```

Listing 6.4: models.py - Setting the test case model to be used by SQLAlchemy

6.2.3 Event Handling

Before starting the Celery workers, the Redis database, which serves as a message broker, needs to be running. After starting the Redis database server (with `redis-server`), to start the Celery application and its workers, the `tasks.py` file is run (see Appendix C - Project Tree) with the `celery -A tasks worker` command.

In this file, which can be seen in Listing 6.5, we start by defining a `base task` that handles and cache the database connection and is the base task for the two tasks.

The `handle_event` task is the one that receives and handles the smart contract event listened by the node. It starts by attempting to fetch from the database a test case that might relate to the received event. If it fails, it will retry 5 more times with an exponential backoff. This is mechanism was implemented as the database retrieval could have failed because the transaction hash was not yet saved in the test case table, as it is only known after the smart contract is invoked, as this transaction hash is saved on the test case table at approximately the same time as the node is receiving the new block. The exponential backoff makes the function retry five times over a 31 seconds time span, giving enough time for the transaction hash to be present in the database.

If a test case is found that relates to the received event, both the event and the test case ID are passed to another task, the `evaluate` task. When the `evaluate` task starts handling this event, the test case is again fetched from the database. This is done as if we were to pass an instance of the test case to the task, the instance could no longer be up to date with the current state of the database by the moment that it is handled by the `evaluate` task. This task then compares the test case expected outputs with the received event and evaluates the test case, after which it saves the execution result as an attribute of the test case.

```
import json
from celery import Task, Celery, shared_task
from exceptions import NoTransactionFound
from models import Session, TestCase
from settings import redis_url

app = Celery('tasks', broker=redis_url)

class DatabaseTask(Task):
    _session = None

    @property
    def session(self):
        if self._session is None:
            self._session = Session()
        return self._session
```

```python


@shared_task(base=DatabaseTask, autoretry_for=(NoTransactionFound,), default_retry_delay=1, max_retries=5, retry_backoff=True)
def handle_event(sc_event):
    """
    This shared_task handles smart contract events by querying the database searching for tests that might concern them
    :param sc_event: a smart contract event
    :return: True if task is successful, False otherwise
    """
    try:
        test_case = handle_event.session.query(TestCase). \
            filter_by(contract_hash=sc_event['contract_hash'], transaction_hash=sc_event['tx_hash'],
                      event_type=sc_event['event_type'], active=True).first()
        if not test_case:
            print("Retrying...")
            raise NoTransactionFound("No test found with tx_hash: " + sc_event['tx_hash'])
    except NoTransactionFound as ntf:
        print(ntf)
    else:
        evaluate.delay(sc_event, test_case.id)
    return True


@shared_task(base=DatabaseTask)
def evaluate(sc_event, test_case_id):
    """
    This shared_task evaluates if a pre-determined test fails or passes.
    Stores the result on the database.
    :param sc_event: a smart contract event
    :param test_case_id: a test case ID
    :return: True if task is successful, False otherwise
    """
    test_case = evaluate.session.query(TestCase).get(test_case_id)
    if not test_case.success:
        sc_event_payload = sc_event['event_payload']

        if sc_event['event_type'] == test_case.event_type.value and sc_event_payload['type'] == test_case.\
                expected_payload_type.value:
            if str(sc_event_payload['value']) == test_case.expected_payload_value:
                test_case.sc_event = json.dumps(sc_event)
                test_case.active = False
                test_case.success = True
        else:
            test_case.sc_event = json.dumps(sc_event)
            test_case.active = False
            test_case.success = False

        evaluate.session.add(test_case)
        try:
            evaluate.session.commit()
        except Exception:
            evaluate.session.rollback()
            raise
        finally:
            evaluate.session.close()
        return True
```

Listing 6.5: tasks.py - Celery application that handles event and evaluates test cases

6.3 Server Side Web Application

As shown in Figure 5.3 of subsection 5.3.1 - Server Side Web Application, the backend of the web application is composed by a database and three applications:

- appHttp.js - Serves the static single page application to the client. An API was added to this application, that handles all operations concerned with authentication, test cases and test suites.
- appCompiler.js - Receives requests from the single page application with smart contract code to be compiled. It then uses the correspondent NEO compiler and returns the compiled code to the client.
- appEcoServices.js - Uses web sockets to provide real-time information on the status of the different services, such as the RPC node status and the number of users active on the platform.

Of these applications, only the appHttp.js application was significantly changed. Similarly to the other two applications, it was developed using Node.js[9], a JavaScript runtime environment that enables developers to use JavaScript as a server language, and Express[10], a minimal Node.js web framework that offers a number of HTTP methods and middleware that makes developing APIs easier and more straightforward. NPM[11], a JavaScript package manager, was used to manage the project dependencies. Sequelize[12], an ORM for Node.js was also used in the project, providing its own PostgreSQL connector and offering useful features such as database models creation and association, and database migrations.

6.3.1 Express Application

The `appHttp.js` file (see Appendix C - Project Tree) creates an Express application object named `app`, imports the required dependencies and libraries and sets up the application settings and middleware. In this file, the application routes are also defined, which have access to the application controllers. These routes are presented in subsection 6.3.3 - Routes and Controllers. The code inside the `appHttps.js` file that creates and sets up the application can be seen in Listing 6.6. Node libraries are installed using the NPM package manager and imported using `require()`.

```
require('dotenv').config();
const express = require('express');
const http = require('http');
const logger = require('morgan'); // log requests to the console (express4)
const app = express();
const bodyParser = require('body-parser'); // pull information from HTML
    POST (express4)
```

[9] https://github.com/nodejs/node

[10] https://github.com/expressjs/express/

[11] https://www.npmjs.com/

[12] https://github.com/sequelize/sequelize

```
const session = require('express-session');
const passport = require('passport'),
    LocalStrategy = require('passport-local').Strategy,
    BearerStrategy = require('passport-http-bearer');

const server = http.createServer(app);
const testController = require('./controllers').test_cases;
const testSuiteController = require('./controllers').test_suite;
const userController = require('./controllers').user;

const bcrypt = require('bcrypt');
const saltRounds = 10;
const jwt = require('jwt-simple');

app.use(express.static(__dirname + '/')); // set the static files location /public/img will be /img for users
app.use(session({
    resave: false,
    saveUninitialized: true,
    secret: process.env.SESSION_SECRET || 'set secret in production', // try to load secret from .env
    cookie: {}
}));

app.use(passport.initialize());
app.use(passport.session());

app.use(logger('dev')); // log every request to the console
app.use(bodyParser.urlencoded({ // parse application/x-www-form-urlencoded
    parameterLimit: 100000, // bigger parameter sizes
    limit: '5mb', // bigger parameter sizes
    extended: false
}));
app.use(bodyParser.json()); // parse application/json
app.use(bodyParser.json({
    type: 'application/vnd.api+json'
})); // parse application/vnd.api+json as json

app.set('jwtTokenSecret', process.env.JWT_TOKEN_SECRET || 'SETSECRETINPRODUCTION');

module.exports = app;
```

Listing 6.6: appHttp.js - Creating the Express application

6.3.2 Authentication

While some resources, such as creating temporary test cases, are available to non-authenticated users, it was a functional requirement that users should be able to create accounts to manage their test cases and suites. Four main packages are used to implement authentication in the application:

- express-session[13], a session middleware to manage user session, saves the session data on server side and sends a cookie with the session ID to the frontend;
- Passport[14], an authentication middleware that offers plugins (called `strategies`) to authenticate requests;

[13] https://www.npmjs.com/package/express-session

[14] https://www.npmjs.com/package/passport

- bcrypt[15], a cryptographic library used to hash passwords;
- jwt-simple[16], a JSON Web Token encoding and decoding library;

The configuration of these packages can be seen in the code presented in Listing 6.7.

```
const app = express();
const session = require('express-session');
const passport = require('passport'),
    LocalStrategy = require('passport-local').Strategy,
    BearerStrategy = require('passport-http-bearer');

app.use(session({
    resave: false,
    saveUninitialized: true,
    secret: process.env.SESSION_SECRET || 'set secret in production', // try to load secret from .env
    cookie: {}
}));

app.set('jwtTokenSecret', process.env.JWT_TOKEN_SECRET || 'SETSECRETINPRODUCTION');

const testController = require('./controllers').test_cases;
const userController = require('./controllers').user;
const testSuiteController = require('./controllers').test_suite;

const bcrypt = require('bcrypt');
const saltRounds = 10;

const jwt = require('jwt-simple');

// This is used for auth. http://www.passportjs.org/docs/
passport.use(new LocalStrategy(
    function (email, password, done) {
        userController.getUserByEmail(email).then(function (user) {
            if (!user) {
                return done(null, false, {
                    message: 'Invalid email or password.'
                });
            }
            bcrypt.compare(password, user.password, function (err, res) {
                if (res) {
                    let expires = new Date();
                    expires.setDate((new Date()).getDate() + 5);
                    let token = jwt.encode({
                        id: user.id,
                        expires: expires
                    }, app.get('jwtTokenSecret')); // get this from env

                    user.token = token;
                    user.save().then(() => {
                        return done(null, user);
                    })
                } else {
                    return done(null, false, {
                        message: 'Invalid email or password.'
                    })
                }
            });
        });
```

[15] https://www.npmjs.com/package/bcrypt

[16] https://www.npmjs.com/package/jwt-simple

```
    }
));

passport.use(new BearerStrategy(
    function (token, done) {
        userController.getUserByToken(token).then(function (user) {
            let decodedToken = jwt.decode(token, app.get('jwtTokenSecret'));
            if (!user) {
                return done(null, false);
            }
            if (new Date(decodedToken.expires) < new Date() || user.id != decodedToken.id) {
                user.token = null;
                return done(null, false);
            }
            return done(null, user, {
                scope: 'all'
            });
        });
    }
));

passport.serializeUser(function (user, done) {
    done(null, user);
});

passport.deserializeUser(function (user, done) {
    done(null, user);
});
```

Listing 6.7: appHttp.js - Authentication middleware used on the application

One of the quality attributes detailed (see subsection 4.5.2 - Quality Attributes) in the non-functional requirements, states that only hashed and salted representations of the password can be stored by the application. In Listing 6.8, the bcrypt library can be seen being used to hash and salt a password sent by the user before calling the `create` method in the `userController` controller.

```
app.post('/api/user',
    function (req, res, next) {
        if (req.body.password != req.body.confirmPassword) {
            res.status(400).send({
                status: "Passwords don't match"
            });
            return;
        }

        bcrypt.hash(req.body.password, saltRounds, function (err, hash) {
            req.body.password = hash;
            next()
        });
    },
    userController.create
);
```

Listing 6.8: appHttp.js - Hashing a user password before calling userController.create

6.3.3 Routes and Controllers

Routes are sections of Express code that associate an HTTP method such as GET, POST, PUT or DELETE, an Uniform Resource Identifier (URI) and a function that handles that request. So, when an HTTP request from the frontend application is made to a given endpoint of the REST API, it is routed to the correspondent controller. The controller then retrieves or modifies data from the models and creates an HTTP response, which is returned to the frontend. This flow can be observed in Figure 6.2.

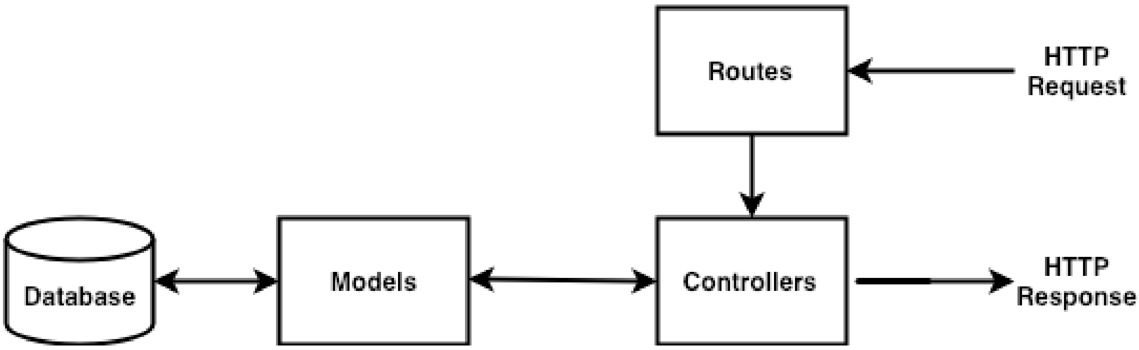


Figure 6.2: Flow diagram of the backend server.

If the HTTP request is requesting an existing resource, a JSON representation of the resource is then returned to the frontend, together with a `200-OK` status code. If the HTTP request is trying to change an existing resource or create a new resource and succeeds, a representation of the new resource is returned to the user, together with a `20X` status code. If the HTTP request tries to access a non-existing resource or sends a bad request, a `40X` status error and message is returned.

As mentioned in subsection 6.3.2 - Authentication, some routes require the user to be authenticated to give access to the requested resources. In Listing 6.9, an example is presented, where, for the user to be able to successfully delete a test suite, the request to the API must pass the authentication.

```
app.delete('/api/test_suite/:testSuiteID',
    function (req, res, next) {
        if (!req.isAuthenticated()) {
            res.status(401).send({
                status: "Unauthorized"
            });
        }
        next();
    },
    passport.authenticate('bearer'),
    testSuiteController.destroy
);
```

Listing 6.9: appHttp.js - Routing to the delete test suite controller.

If the request passes the authentication, it is then routed to the `destroy` function of the `testSuiteController` (see in Listing 6.10), which, if the test suite belongs to the user that made the request, fetches and destroys the chosen test suite.

```
const TestSuite = require('../models').TestSuite;
const TestCase = require('../models').TestCase;

module.exports = {
    create(req, res) {
        ...
    },
    list(req, res) {
        ...
    },
    retrieve(req, res) {
        ...
    },
    destroy(req, res) {
        return TestSuite
        .findByPk(req.params.testSuiteID)
        .then(testSuite => {
            if (!testSuite) {
                return res.status(400).send({
                    message: 'TestSuite Not Found',
                });
            } else if (testSuite.userId != null && testSuite.userId != req.user.id) {
                return res.status(403).send({
                    message: 'Forbidden!',
                });
            }
            return testSuite
            .destroy()
            .then(() => res.status(200).send({ message: 'Test Suite deleted successfully.' }))
            .catch(error => res.status(400).send(error));
        })
        .catch(error => res.status(400).send(error));
    },
    update(req, res) {
        ...
    }
};
```

Listing 6.10: controllers/test_suite.js - The destroy function inside the test suite controller.

6.3.4 Models

Three different models were defined in the application:

- TestCase - This model can belong to both a user and a test suite, in a one-to-many relationship.
- TestSuite - This model can belong to a user, in a one-to-many relationship.
- User - This model can own one or more test cases and test suites.

Mappings between the database tables and models are defined by Sequelize. The Sequelize CLI automatically creates the `models/index.js` file, shown in Listing 6.11, when Sequelize is added to the project and sets a new connection to the database (line 13 or 15 of the file, depending on which configuration file is being read).

```
1 'use strict';
2
3 const fs = require('fs');
4 const path = require('path');
5 const Sequelize = require('sequelize');
6 const basename = path.basename(__filename);
```

```
 7 const env = process.env.NODE_ENV || 'development';
 8 const config = require(__dirname + '/../config/config.json')[env];
 9 const db = {};
10
11 let sequelize;
12 if (config.use_env_variable) {
13     sequelize = new Sequelize(process.env[config.use_env_variable], config);
14 } else {
15     sequelize = new Sequelize(config.database, config.username, config.password, config);
16 }
17
18 fs
19     .readdirSync(__dirname)
20     .filter(file => {
21         return (file.indexOf('.') !== 0) && (file !== basename) && (file.slice(-3) === '.js');
22     })
23     .forEach(file => {
24         const model = sequelize['import'](path.join(__dirname, file));
25         db[model.name] = model;
26     });
27
28 Object.keys(db).forEach(modelName => {
29     if (db[modelName].associate) {
30         db[modelName].associate(db);
31     }
32 });
33
34 db.sequelize = sequelize;
35 db.Sequelize = Sequelize;
36
37 module.exports = db;
```

Listing 6.11: The models/index.js file

All of the application models are located as separate files inside the `models` folder (which can be seen in Appendix C - Project Tree). Sequelize, through the code present in the `models/index.js` file (see Listing 6.11, from line 18 to line 32), automatically tracks every model present in that folder and associates them, if needed.

The `test_suite.js` file, depicted on Listing 6.12, is a good and short example of how models are defined in the application. The `define` method defines the mapping between the model and a database table. The model attributes and its data types are then defined. After defining the model, associations with other models can be defined, as shown from line 15 to line 24 in Listing 6.12. One of the advantages of using an ORM such as Sequelize is that the syntax used often has excellent readability. As such, it is easy to understand which associations are being formed in the code above: the `hasMany` association creates a 1 to N relationship between the `TestSuite` model and the `TestCase` model, with one test suite having zero or more test cases. This relationship is creating by adding a `foreign key` `testSuiteId` that refers to a test suite id, its `primary key`. The `as` keyword makes it possible to refer to every test case belonging to a test suite by using the alias `testCases`. It is also evident that the `belongsTo` association is part of an N to 1 association between the `TestSuite` and `User` models. In this case, one user can have 0 or more test suites, and a `foreign key` `userId` referring to the user `primary key` is saved as a `TestSuite` model column.

```
1 module.exports = (sequelize, DataTypes) => {
2     const TestSuite = sequelize.define('TestSuite', {
        name: {
            type: DataTypes.STRING,
            allowNull: false
        },
        description: {
            type: DataTypes.STRING,
            allowNull: true
        },
    }, {
        tableName: 'test_suites'
    });

    TestSuite.associate = (models) => {
        TestSuite.hasMany(models.TestCase, {
            foreignKey: 'testSuiteId',
            as: 'testCases',
        });
        TestSuite.belongsTo(models.User, {
            foreignKey: 'userId',
            onDelete: 'CASCADE',
        });
    };

    return TestSuite;
};
```

Listing 6.12: The models/test_suite.js file

Migrations are used in the project to keep track of changes to the database and change the database state. Those state transitions are defined in JavaScript and stored in the `migrations` folder (which can be seen in Appendix C - Project Tree). The code inside a migration file, as shown in Listing 6.13, describes how to change the database to the intended state and how to revert the changes in order to get back to the original state. When a model definition is changed, a migration is created to reflect those changes.

```
module.exports = {
    up: (queryInterface, Sequelize) =>
        queryInterface.createTable('test_suites', {
            id: {
                allowNull: false,
                autoIncrement: true,
                primaryKey: true,
                type: Sequelize.INTEGER,
            },
            name: {
                type: Sequelize.STRING,
                allowNull: false,
            },
            description: {
                type: Sequelize.STRING,
                allowNull: false,
            },
            createdAt: {
                allowNull: false,
                type: Sequelize.DATE,
            },
            updatedAt: {
                allowNull: false,
                type: Sequelize.DATE,
            },
            userId: {
                type: Sequelize.INTEGER,
                onDelete: 'CASCADE',
                references: {
                    model: 'users',
                    key: 'id',
                    as: 'userId',
                },
            },
        }),
    down: (queryInterface) =>
        queryInterface.dropTable('test_suites'),
};
```

Listing 6.13: A migration to create the test suites table

Sequelize keeps track of all of the applied migrations in a table inside the database. To apply all of the new migrations to the database, the developer needs to run the `sequelize db:migrate` command on the Sequelize CLI.

6.4 Single Page Web Application

This SPA is the frontend application that enables the user to interact with the system.

The frontend application was developed using AngularJS[17], JQuery[18], the Neon-js library[19], vanilla JavaScript, Bootstrap [20] and CSS.

AngularJS is a JavaScript framework which extends HTML syntax to create dynamic web applications and was mainly used in this application to create routes inside the SPA and group HTML into reusable components. Neon-js is a JavaScript library that enables a developer to interact, through JavaScript, with the NEO blockchain. It's used, in this project, for most of the operations exposed on the frontend that require interaction with the NEO blockchain, such as smart contract deployment, smart contract invocation, and RPCs to the NEO blockchain. JQuery is a JavaScript library that simplifies Asynchronous JavaScript And XML (AJAX) requests, Document Object Model (DOM) manipulation and event handling, among other useful features. JQuery was mainly used to make requests to the backend API and present the retrieved data through DOM manipulation. Bootstrap is an HTML and CSS framework, which provides design templates for typography, forms, buttons, navigation, and other interface components. All of the presented libraries and frameworks are open-source, and their source code can be consulted on the links provided on the footnotes of this report.

6.4.1 Landing Page and Authentication

As mentioned on the User Stories 2, 3 and 4 (see US2 on table 4.4 - User Stories - Landing Page and US3 and US4 on table 4.5 - User Stories - Authentication), the user must be able to easily access a an authentication form from the landing page and must be able to create an account and login with his account. This simple form, depicted in Figure 6.3, is implemented on top of a Bootstrap modal component.

To signup, the user needs to insert an email address on the form, a password, and the password confirmation. After submitting the form, a POST request is made to the au-

[17] `https://github.com/angular/angular.js`

[18] `https://github.com/jquery/jquery`

[19] `https://github.com/CityOfZion/neon-js`

[20] `https://github.com/twbs/bootstrap`

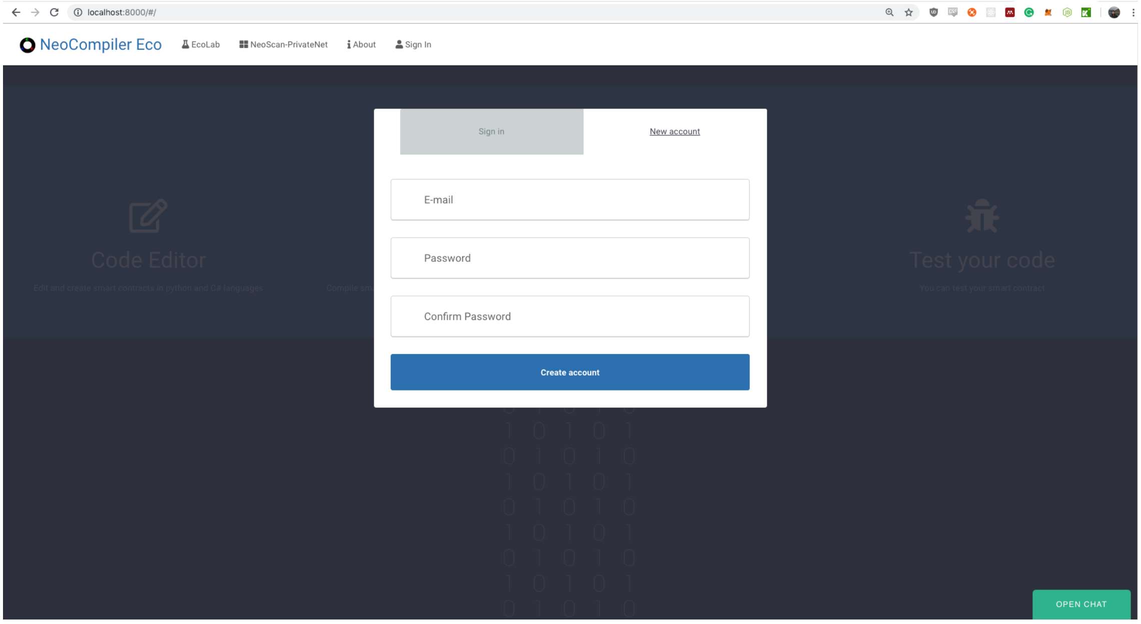


Figure 6.3: Landing Page - Sign In and Sign Up form.

thentication endpoint in the backend API. If the user is logging in with a valid account and password, the server returns a session ID which is stored in the frontend application session storage and a session cookie, which are used in subsequent requests while the session is alive to authenticate the user.

6.4.2 Editing, Compiling and Deploying Smart Contracts

By building the testing tool on top of the NeoCompiler Eco tool, it is possible to ensure that the functional requirements presented on User Stories 10, 11, 12 and 13 (see US10, US11, US12 and US13 on table 4.6 - User Stories - Private Network) are satisfied, as this tool already offers an online editor to edit smart contract code, which can then be compiled and deployed to a private network, as mentioned on subsection 2.5.6 - NeoCompiler Eco, and depicted in Figures 2.11 and 2.12 of that subsection. As shown in Figure 6.4, the original tool also offers an interface which the user can use to make JSON RPCs to the private network.

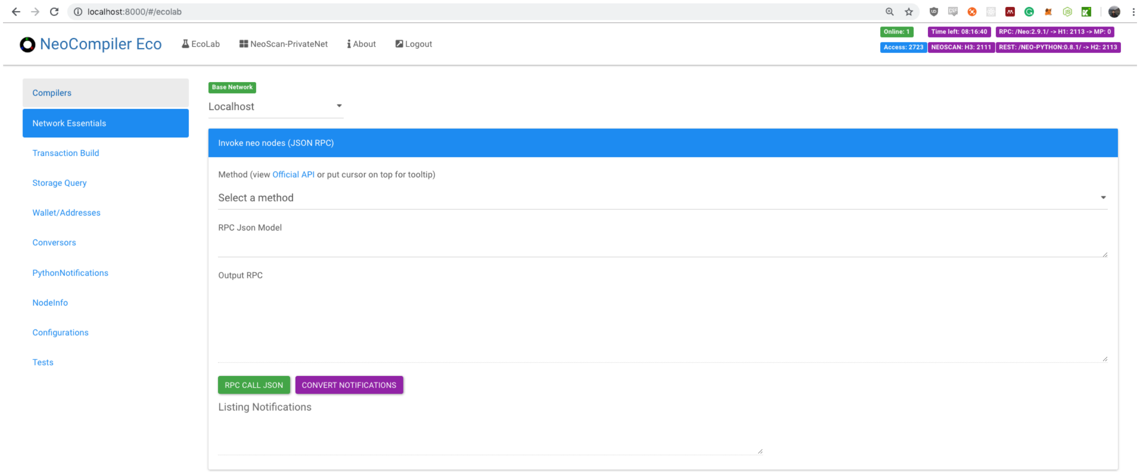
6.4.3 Test Cases and Test Suites

The biggest challenge on the frontend development was to create a testing interface that would allow the tool to satisfy the functional requirements represented by the User Stories 15 to 32 see US15-32 on tables 4.7, 4.8 and 4.9 - User Stories - Testing).

To satisfy the requirement that the user must be able to create a test case, the information required for the test to be created needed to be defined. As described in subsection 2.6.2 - Test Case, a set of test inputs, execution conditions and expected outputs are required in order to define the test case.

To define the test inputs, the user needs to provide:

- The contract hash of the contract to be invoked;



Base Network: Localhost

Method (view Official API or put cursor on top for tooltip)
Select a method

RPC Json Model

Output RPC

Listing Notifications

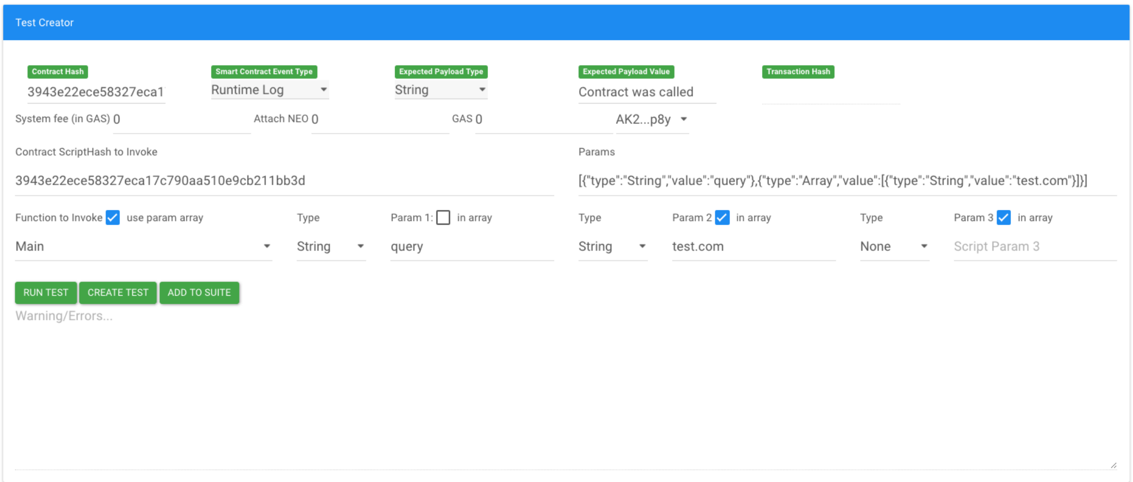
Figure 6.4: Network Essential - RPCs to the private network.

- The contract hash of the contract that will emit the output;
- The function to invoke on the smart contract and its parameters;
- The amount of NEO and GAS to be sent in the transaction;
- The GAS fee to be paid for the transaction;
- The wallet that will make the transaction;

To define the expected outcome of the test, the user needs to insert:

- The event type that is expected to result from running the test;
- The payload of that event;

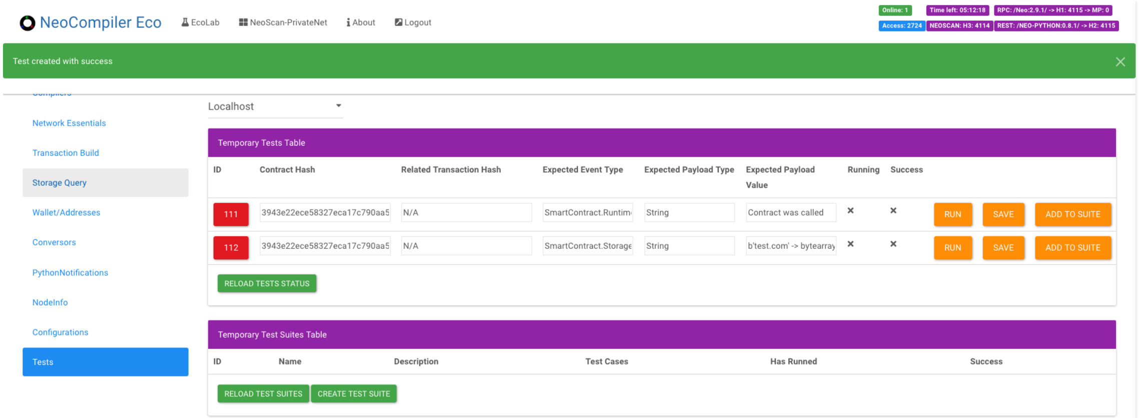
For this tool, the execution condition is the state of the blockchain at the start of the test, which can not be changed by the user. A drawback of the blockchain technology is that every operation is permanent on the blockchain, and as the blockchain is running on the server it is not feasible to bring the blockchain to a clean state after each test execution. As such, execution order is important when running tests, as non-idempotent operations permanently change the state of the blockchain, and a test that evaluated to true the first time it ran may evaluate to false in a future execution.



Contract Hash: 3943e22ece58327eca1
Smart Contract Event Type: Runtime Log
Expected Payload Type: String
Expected Payload Value: Contract was called
Transaction Hash:
System fee (in GAS) 0
Attach NEO 0
GAS 0
AK2...p8y
Contract ScriptHash to Invoke: 3943e22ece58327eca17c790aa510e9cb211bb3d
Params: [{"type":"String","value":"query"},{"type":"Array","value":[{"type":"String","value":"test.com"}]}]
Function to Invoke ☑ use param array: Main
Type: String
Param 1: □ in array: query
Type: String
Param 2 ☑ in array: test.com
Type: None
Param 3 ☑ in array: Script Param 3
Warning/Errors...

Figure 6.5: Testing - Creating a test case using the test creator.

When developing the testing interface, to simplify test creation and execution for new users and developers, it was decided that a "Test Creator" would be implemented, where the user could easily create tests by inputting the required information in a form. This interface can be seen in Figure 6.5. After providing the required information in the Test Creator, the user can either choose to immediately run the test, temporarily save the test or add it to an existing test suite. If the user chooses to run the test or to create it without running it, the test is presented on the temporary tests table, depicted in Figure 6.6. This table temporarily keeps user tests while the browser is opened in the tool. If the user closes or refreshes the page, the temporary tests are lost.



Test created with success

Temporary Tests Table

ID	Contract Hash	Related Transaction Hash	Expected Event Type	Expected Payload Type	Expected Payload Value	Running	Success			
111	3943e22ece58327eca17c790aa5	N/A	SmartContract.Runtim	String	Contract was called	×	×	RUN	SAVE	ADD TO SUITE
112	3943e22ece58327eca17c790aa5	N/A	SmartContract.Storage	String	b'test.com' -> bytearray	×	×	RUN	SAVE	ADD TO SUITE

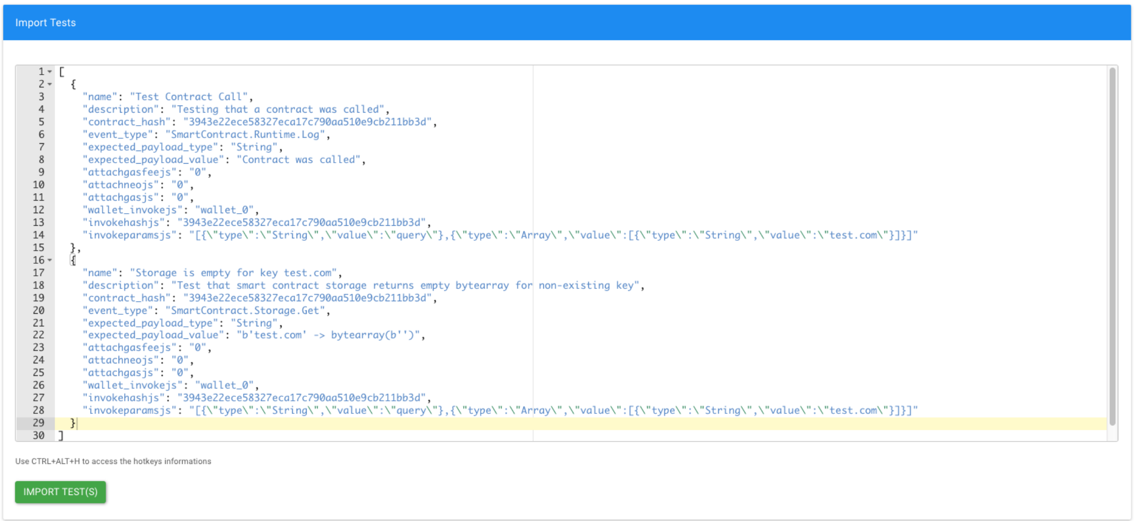
Temporary Test Suites Table

ID	Name	Description	Test Cases	Has Runned	Success

Figure 6.6: Testing - Temporary tests table.

Using the Test Creator is the easiest way for a new user to create a test. However, after understanding how the tool works, it would be faster if the user could create tests in bulk. To enable the user to do so, an editor was added that lets the user write tests in JSON and import them to the tool. This way, the user can create an unlimited amount of tests, in the online editor or in his favourite IDE, and quickly import them to the tool, as shown in Figure 6.7. Tests added through the editor are also added to the temporary test table depicted in Figure 6.6. The data required in the JSON for Test Import is specified in Appendix D - JSON Structure, and is the same as the data required in the Test Creator, with optional test name and description fields.

When a new test is created, it is sent to the backend API through a POST request and



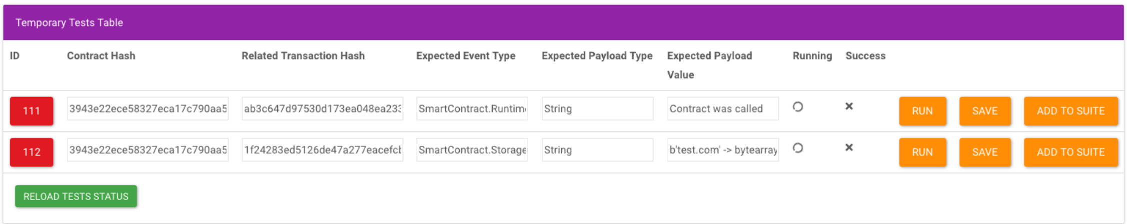
```
[
  {
    "name": "Test Contract Call",
    "description": "Testing that a contract was called",
    "contract_hash": "3943e22ece58327eca17c790aa510e9cb211bb3d",
    "event_type": "SmartContract.Runtime.Log",
    "expected_payload_type": "String",
    "expected_payload_value": "Contract was called",
    "attachgasfeejs": "0",
    "attachneojs": "0",
    "attachgasjs": "0",
    "wallet_invokejs": "wallet_0",
    "invokehashjs": "3943e22ece58327eca17c790aa510e9cb211bb3d",
    "invokeparamsjs": "[{\"type\":\"String\",\"value\":\"query\"},{\"type\":\"Array\",\"value\":[{\"type\":\"String\",\"value\":\"test.com\"}]}]"
  },
  {
    "name": "Storage is empty for key test.com",
    "description": "Test that smart contract storage returns empty bytearray for non-existing key",
    "contract_hash": "3943e22ece58327eca17c790aa510e9cb211bb3d",
    "event_type": "SmartContract.Storage.Get",
    "expected_payload_type": "String",
    "expected_payload_value": "b'test.com' -> bytearray(b'')",
    "attachgasfeejs": "0",
    "attachneojs": "0",
    "attachgasjs": "0",
    "wallet_invokejs": "wallet_0",
    "invokehashjs": "3943e22ece58327eca17c790aa510e9cb211bb3d",
    "invokeparamsjs": "[{\"type\":\"String\",\"value\":\"query\"},{\"type\":\"Array\",\"value\":[{\"type\":\"String\",\"value\":\"test.com\"}]}]"
  }
]
```

Use CTRL+ALT+H to access the hotkeys informations

Figure 6.7: Testing - Creating two test cases by importing them with JSON.

saved in the database. After the user imports or creates a new test and it being available in the temporary test table, the user can now execute the test by pressing the "Run" button on the correspondent test in the temporary test table or the "Run Test" button on the Test Creator. Running a test results in the browser making an invocation to the smart contract to be tested, with the parameters defined in the test case. This invocation is done through the Neon-js library presented previously. This invocation returns a transaction hash, which uniquely identifies the resulting transaction inside the NEO blockchain. This transaction hash is then saved in the backend database together with the rest of the test case data, waiting for the NEO Node Tap to do its job. This operation can take a few seconds due to the nature of the blockchain technology. The browser periodically tries to retrieve new data corresponding to the tests presented in the tables (shown in Figure 6.8) from the backend API and update the DOM with AJAX GET requests. The user is able to force these requests by pressing the "Reload Tests Status" button.

A test that is running and waiting for its result has a spinning-wheel on the "Running" column, as represented in Figure 6.8, while a test that has not yet run has a cross and a test that has already run has a check mark.

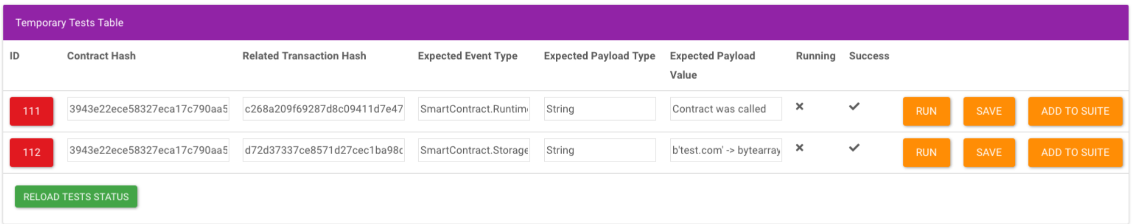


ID	Contract Hash	Related Transaction Hash	Expected Event Type	Expected Payload Type	Expected Payload Value	Running	Success			
111	3943e22ece58327eca17c790aa5	ab3c647d97530d173ea048ea23	SmartContract.Runtim	String	Contract was called	○	×	RUN	SAVE	ADD TO SUITE
112	3943e22ece58327eca17c790aa5	1f24283ed5126de47a277eacefcl	SmartContract.Storage	String	b'test.com' -> bytearray	○	×	RUN	SAVE	ADD TO SUITE

Figure 6.8: Testing - Running test cases are displayed on the temporary tests table.

If a test case passes, a check mark appears in the "Success" column, while a cross appears in the cases where the test has failed or has not run yet. An example of successful test cases can be seen in Figure 6.9.

Test cases in the temporary tests table can be removed from the table, run (and re-run) by pressing the "Run" button, or they can be added to a test suite. If the user is authenticated in the system, they can also be saved by pressing the "Save" button, only visible to authenticated users. These options can be seen, for instance, in Figure 6.9.

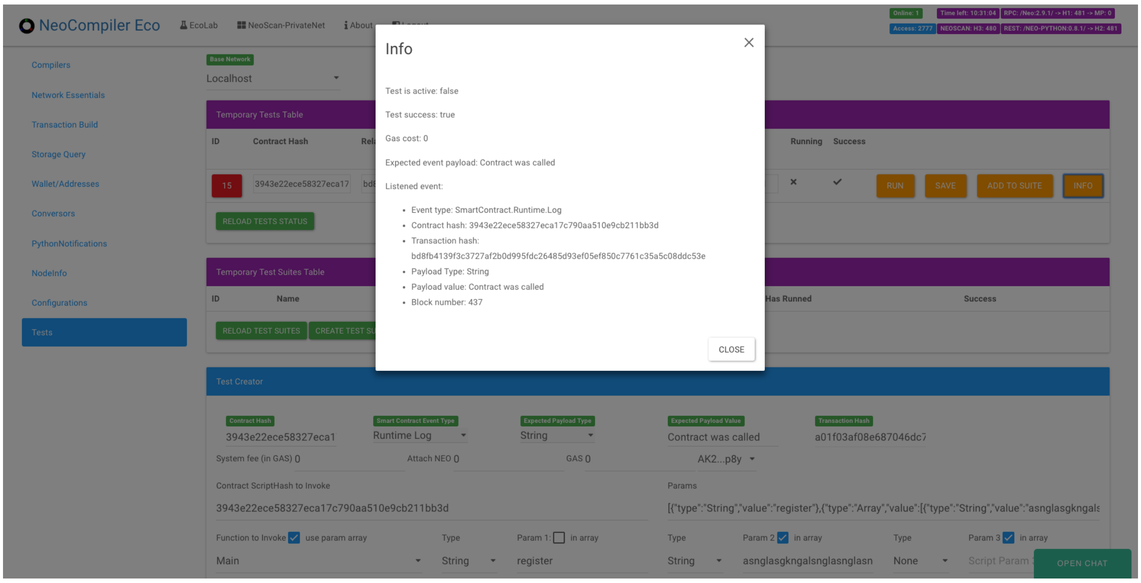


Temporary Tests Table

ID	Contract Hash	Related Transaction Hash	Expected Event Type	Expected Payload Type	Expected Payload Value	Running	Success			
111	3943e22ece58327eca17c790aa5	c268a209f69287d8c09411d7e47	SmartContract.Runtim	String	Contract was called	✕	✓	RUN	SAVE	ADD TO SUITE
112	3943e22ece58327eca17c790aa5	d72d37337ce8571d27cec1ba98c	SmartContract.Storage	String	b'test.com' -> bytearray	✕	✓	RUN	SAVE	ADD TO SUITE

Figure 6.9: Testing - Successful test cases are displayed on the temporary tests table.

After a test is executed, the user can have access to a small report on the test execution. This report, depicted in Figure 6.10, displays to the user the NEOGas cost of having run the test as well as details on the listened event that led to the evaluation of the test. By looking at this report, the user can understand how much NEOGas it would cost to run that transaction and understand why the test case is passing or failing.



Info

Test is active: false

Test success: true

Gas cost: 0

Expected event payload: Contract was called

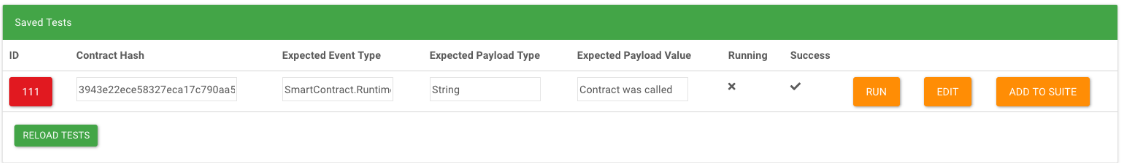
Listened event:

- Event type: SmartContract.Runtime.Log
- Contract hash: 3943e22ece58327eca17c790aa510e9cb211bb3d
- Transaction hash: bd8fb4139f3c3727af2b0d995fdc26485d93ef05ef850c7761c35a5c08ddc53e
- Payload Type: String
- Payload value: Contract was called
- Block number: 437

Contract Hash: 3943e22ece58327eca1
Smart Contract Event Type: Runtime Log
Expected Payload Type: String
Expected Payload Value: Contract was called
Transaction Hash: a01f03af08e687046dc7
System fee (in GAS) 0
Attach NEO 0
GAS 0
AK2...p8y
Contract ScriptHash to Invoke: 3943e22ece58327eca17c790aa510e9cb211bb3d
Params: [{"type":"String","value":"register"},{"type":"Array","value":[{"type":"String","value":"asnglasgkngals
Function to Invoke ☑ use param array: Main
Type: String
Param 1: □ in array: register
Type: String
Param 2 ☑ in array: asnglasgkngalsnglasnglasn
Type: None
Param 3 ☑ in array: Script Param

Figure 6.10: Testing - Modal displaying a small report of a test case execution.

When saving a test case, the user is prompted to insert a test case name and description. After being saved, the test is displayed on the saved tests table, as depicted in Figure 6.11 and is available to the user until it is deleted. As such, the user can access saved tests in any browser compatible with the tool, anywhere, and re-use them easily.

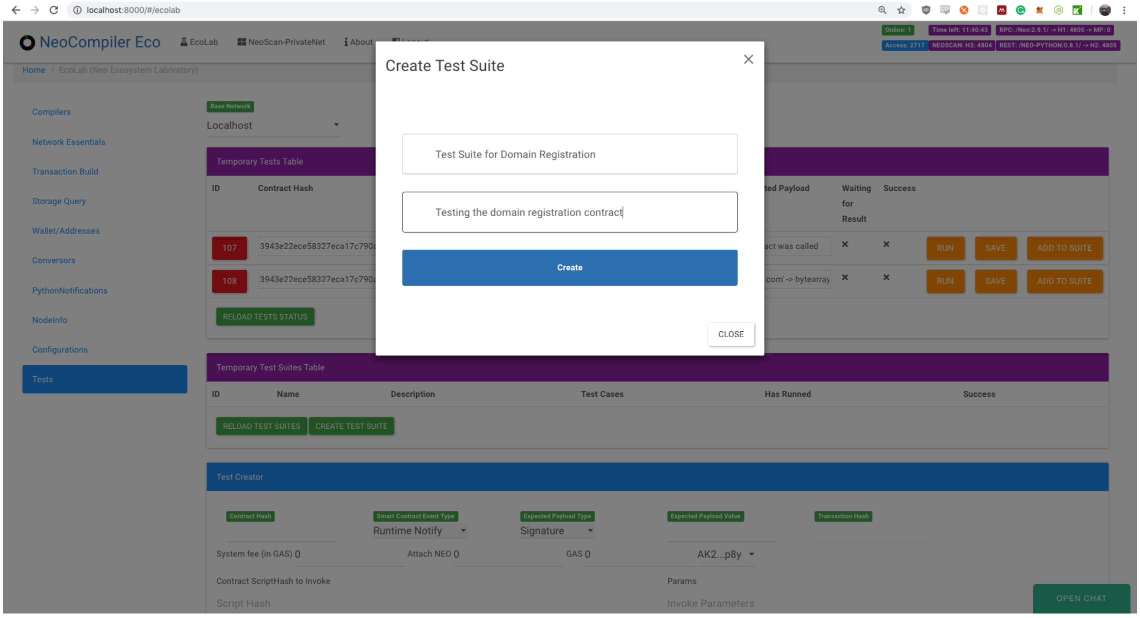


Saved Tests

ID	Contract Hash	Expected Event Type	Expected Payload Type	Expected Payload Value	Running	Success			
111	3943e22ece58327eca17c790aa5	SmartContract.Runtim	String	Contract was called	✕	✓	RUN	EDIT	ADD TO SUITE

Figure 6.11: Testing - Saved tests are displayed in the saved tests table.

To add a test case to a test suite, the user needs to have a test suite created, or create a new test suite first. The user can create a test suite by selecting the "Create Test Suite" button on the temporary test suites table, which can be seen in Figure 6.6. The user then needs to fill the "Create Test Suite" prompt with the test suite name and description, as seen in Figure 6.12.

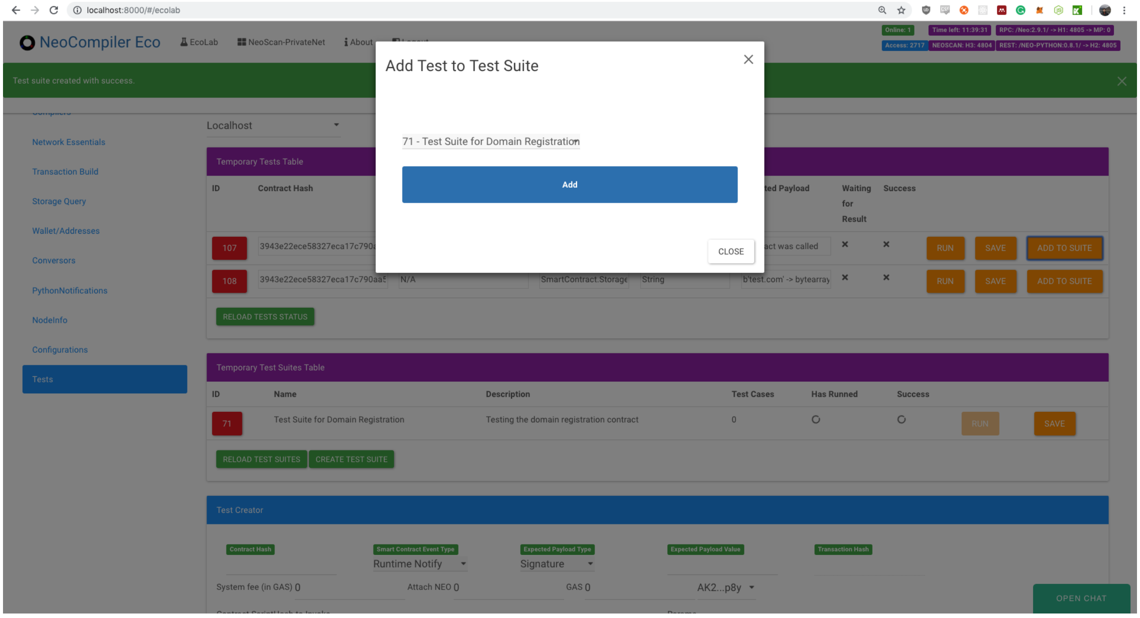


Test Suite for Domain Registration

Testing the domain registration contract

Figure 6.12: Testing - Creating a new test suite.

After creating a test suite, the user has to select a test case from the temporary tests table or from the saved tests table, select the "Add To Suite" button and choose the test suite to which to add the test case, as seen in Figure 6.13.



Test suite created with success.

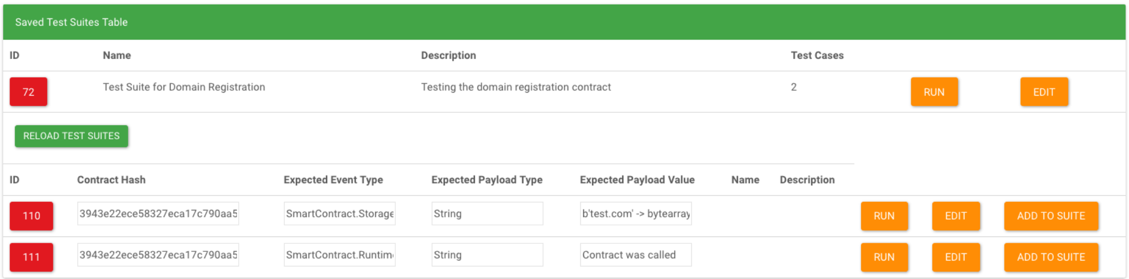
71 - Test Suite for Domain Registration

ID	Name	Description	Test Cases	Has Runned	Success		
71	Test Suite for Domain Registration	Testing the domain registration contract	0	○	○	RUN	SAVE

Figure 6.13: Testing - Adding a test case to a test suite.

Similarly to a test case, a test suite can be saved in order to be permanently available to the user, moving to the saved test suites table, as seen in Figure 6.14. If the test suite is not saved, it is only kept in the frontend while the user does not refresh or closes the page.

A test suite that has one or more test cases can be run by pressing the button "Run". This follows the same process as when running a single test case, but it automatically applies it to every test case in the suite. If all the test cases pass, the test suite displays a check mark on the success column, as seen in Figure 6.15. Otherwise, it displays a cross.



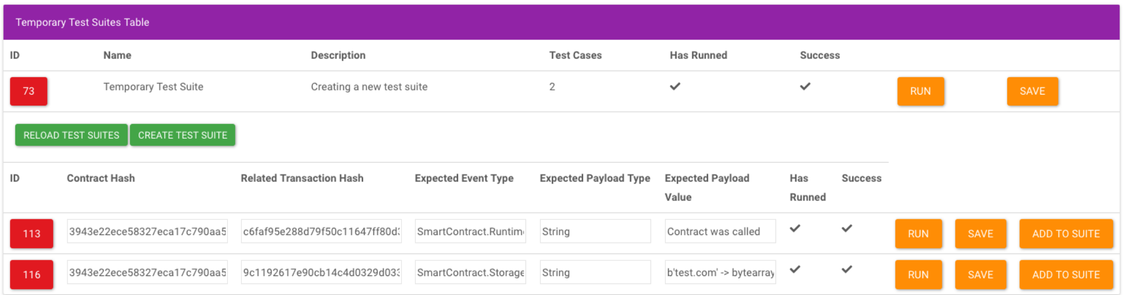
Saved Test Suites Table

ID	Name	Description	Test Cases		
72	Test Suite for Domain Registration	Testing the domain registration contract	2	RUN	EDIT

RELOAD TEST SUITES

ID	Contract Hash	Expected Event Type	Expected Payload Type	Expected Payload Value	Name	Description			
110	3943e22ece58327eca17c790aa5	SmartContract.Storage	String	b'test.com' -> bytearray			RUN	EDIT	ADD TO SUITE
111	3943e22ece58327eca17c790aa5	SmartContract.Runtim	String	Contract was called			RUN	EDIT	ADD TO SUITE

Figure 6.14: Testing - Saved test suites table.



Temporary Test Suites Table

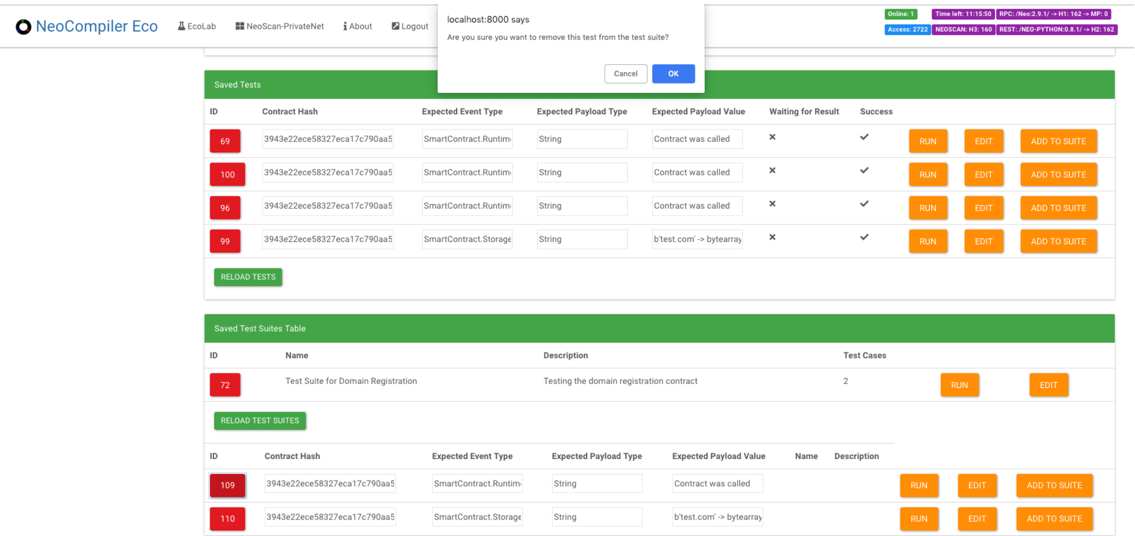
ID	Name	Description	Test Cases	Has Runned	Success		
73	Temporary Test Suite	Creating a new test suite	2	✓	✓	RUN	SAVE

RELOAD TEST SUITES CREATE TEST SUITE

ID	Contract Hash	Related Transaction Hash	Expected Event Type	Expected Payload Type	Expected Payload Value	Has Runned	Success			
113	3943e22ece58327eca17c790aa5	c6faf95e288d79f50c11647ff80d:	SmartContract.Runtim	String	Contract was called	✓	✓	RUN	SAVE	ADD TO SUITE
116	3943e22ece58327eca17c790aa5	9c1192617e90cb14c4d0329d03:	SmartContract.Storage	String	b'test.com' -> bytearray	✓	✓	RUN	SAVE	ADD TO SUITE

Figure 6.15: Testing - Running a test suite.

The user can also delete saved test cases and saved test suites. Deleting a saved test suite also deletes every test case that belongs to it. As shown in Figure 6.16, the user can also delete a test case from inside a test suite.



localhost:8000 says

Are you sure you want to remove this test from the test suite?

Cancel OK

Saved Tests

ID	Contract Hash	Expected Event Type	Expected Payload Type	Expected Payload Value	Waiting for Result	Success			
69	3943e22ece58327eca17c790aa5	SmartContract.Runtim	String	Contract was called	✗	✓	RUN	EDIT	ADD TO SUITE
100	3943e22ece58327eca17c790aa5	SmartContract.Runtim	String	Contract was called	✗	✓	RUN	EDIT	ADD TO SUITE
96	3943e22ece58327eca17c790aa5	SmartContract.Runtim	String	Contract was called	✗	✓	RUN	EDIT	ADD TO SUITE
99	3943e22ece58327eca17c790aa5	SmartContract.Storage	String	b'test.com' -> bytearray	✗	✓	RUN	EDIT	ADD TO SUITE

RELOAD TESTS

Saved Test Suites Table

ID	Name	Description	Test Cases		
72	Test Suite for Domain Registration	Testing the domain registration contract	2	RUN	EDIT

RELOAD TEST SUITES

ID	Contract Hash	Expected Event Type	Expected Payload Type	Expected Payload Value	Name	Description			
109	3943e22ece58327eca17c790aa5	SmartContract.Runtim	String	Contract was called			RUN	EDIT	ADD TO SUITE
110	3943e22ece58327eca17c790aa5	SmartContract.Storage	String	b'test.com' -> bytearray			RUN	EDIT	ADD TO SUITE

Figure 6.16: Testing - Removing a test case from a test suite.

Chapter 7

Testing

As discussed and presented on Section 2.6 - Software Testing, testing attempts to ensure that an application performs as its developers expected. As this project concerns the development of a testing framework, it is important to understand if the application assesses test results correctly, and if all of its features work as required. Bugs in the application can be critical, not only because users will be discouraged from using the application, but also because wrongly assessed tests may cause users to deploy buggy smart contracts into production. Creating tests for the application also ensures that, through regression testing, any future alteration to the application code will not cause previously implemented features to malfunction.

In an attempt to determine whether the application is functioning as intended, the NEO Node Tap, the Web Server Application and the Web Client were tested, as described in the next sections. A copy of the original database was created, to be used during testing.

7.1 Server Side Web Application

Mocha[1] and Chai[2] were the libraries used to unit test the backend API. Mocha is a test framework for Node.js, which can be used along with many different assertion libraries, providing the testing environment for testing the application. Chai was the assertion library used chosen to be used alongside Mocha, which, together with the Chai-HTTP plugin, enables to make and test HTTP calls to the API.

To test that the API was working as intended, three files were created, correspondent to the three different models used by the system (test cases, test suites and users) and were placed under the `tests` folder of the web application project (see Appendix C - Project Tree). Each file contains a set of test suites, with each suite dedicated to testing a determined endpoint in the application, and `after` and `before` methods that bring the environment to the desired state before and after the tests are run, as seen in Listing 7.1.

```
//During the test the env variable is set to test
process.env.NODE_ENV = 'test';

const User = require('../models/').User;

//Require the test libraries
const chai = require('chai');
```

[1] https://github.com/mochajs/mocha

[2] https://github.com/chaijs/chai

```
const chaiHttp = require('chai-http');
chai.use(chaiHttp);
const should = chai.should();

//Require the app
const app = require('../appHttp');

let agent = chai.request.agent(app); // request agent used to mantain cookies between requests
...

describe('Users', () => {
    // Clean existing user data
    before(function (done) {
        User.sync({
            force: true
        }) // drops table and re-creates it
        .then(function () {
            done(null);
        }, function (err) {
            done(err);
        });
    },
    after(function (done) {
        agent.close();
        User.sync({
            force: true
        }) // drops table and re-creates it
        .then(function () {
            done(null);
        }, function (err) {
            done(err);
        });
    }));

    ...
});
```

Listing 7.1: user.js - Requiring the test libraries and setting up the test environment. The before and after methods guarantee that the User table is clean before and after the tests in this file have run.

After setting the environment, tests to each route follow. In Listing 7.2, it can be seen how each test is implemented. Each test is preceded by a small description. Chai-HTTP is then used to make a request to the API. Finally, the response is received, and its status and content are compared against the expected output. If this assessment fails, the test raises an error and is marked as failed, otherwise the test passes. As also seen in Listing 7.2, endpoints are not only tested with valid data but also with invalid requests that should be rejected by the application.

```
    /*
     * Test /api/user and create user method
     */
    describe('/POST user', () => {
        it('it should create a valid user', (done) => {
            const user = {
                email: "user123@mail.com",
                password: "uns4f3P4ss0rd",
                confirmPassword: "uns4f3P4ss0rd"
            }
            chai.request(app)
                .post('/api/user')
                .send(user)
                .end((err, res) => {
                    res.should.have.status(201);
                    res.body.should.be.a('object');
                    res.body.should.have.property('username');
                    res.body.should.have.property('username').eql(user.email);
                    done();
                });
        });

        it('it should not allow different passwords', (done) => {
            const user = {
                email: "otheruser@mail.com",
                password: "uns4f3P4ss0rd",
                confirmPassword: "unsafepassword"
            }
            chai.request(app)
                .post('/api/user')
                .send(user)
                .end((err, res) => {
                    res.should.have.status(422);
                    res.body.should.be.a('object');
                    res.body.should.have.property('errors');
                    res.body.errors.should.be.a('array');
                    res.body.errors[0].should.be.a('object');
                    res.body.errors[0].should.have.property('msg');
                    res.body.errors[0].should.have.property('msg').eql('Passwords don\'t match');
                    done();
                });
        });

        it('it should not allow dupplicate users', (done) => {
            const user = {
                email: "user123@mail.com",
                password: "anotherpassword",
                confirmPassword: "anotherpassword"
            }
            chai.request(app)
                .post('/api/user')
                .send(user)
                .end((err, res) => {
                    res.should.have.status(400);
                    res.body.should.be.a('object');
                    res.body.should.have.property('errors')
                    res.body.errors.should.be.a('array');
                    res.body.errors[0].should.be.a('object');
                    res.body.errors[0].should.have.property('message');
                    res.body.errors[0].should.have.property('message').eql('email must be unique');
                    done();
                });
        });

        it('it should not allow users with empty password', (done) => {
            const user = {
                email: "newuser@mail.com",
                password: "",
                confirmPassword: ""
            }
            chai.request(app)
                .post('/api/user')
            .send(user)
            .end((err, res) => {
                res.should.have.status(422);
                res.body.should.be.a('object');
                res.body.should.have.property('errors');
                res.body.errors.should.be.a('array');
                res.body.errors[0].should.be.a('object');
                res.body.errors[0].should.have.property('msg');
                res.body.errors[0].should.have.property('msg').eql('Invalid value');
                done();
            });
    });
});
```

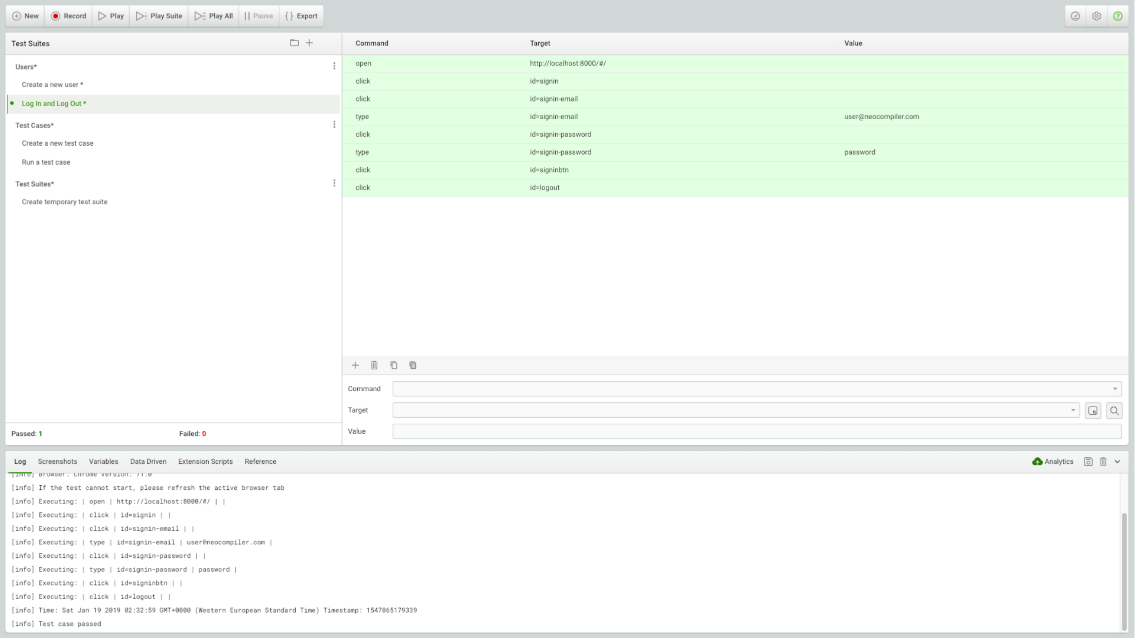
Listing 7.2: user.js - Set of tests that test the API endpoint that creates new users.

After implementing the tests with Chai, Mocha is used to run them, presenting a report at the end of its execution.

7.2 Single Page Web Application

The SPA was tested using the Katalon Recorder, a browser add-on built on top of the Selenium test automation framework. With this IDE it is possible to create and record frontend tests and organise them into test suites.

Katalon Recorder was used to realistically simulate all of the operations performed by the user on the frontend web application, which are specified in the User Stories (see Section 4.4 - Functional Requirements). In the Katalon IDE, each test case is composed by a series of actions, which target a given component in the SPA. If the simulator fails to perform any of the actions, the test case fails. If it succeeds in performing every action, the test case passes, as depicted in Figure 7.1.



Users*
Create a new user *
Log In and Log Out *
Test Cases*
Create a new test case
Run a test case
Test Suites*
Create temporary test suite

Command	Target	Value
open	http://localhost:8000/#/	
click	id=signin	
click	id=signin-email	
type	id=signin-email	user@neocompiler.com
click	id=signin-password	
type	id=signin-password	password
click	id=signinbtn	
click	id=logout	

Passed: 1

Failed: 0

```
[info] Browser: Chrome version: 71.0
[info] If the test cannot start, please refresh the active browser tab
[info] Executing: | open | http://localhost:8000/#/ | |
[info] Executing: | click | id=signin | |
[info] Executing: | click | id=signin-email | |
[info] Executing: | type | id=signin-email | user@neocompiler.com |
[info] Executing: | click | id=signin-password | |
[info] Executing: | type | id=signin-password | password |
[info] Executing: | click | id=signinbtn | |
[info] Executing: | click | id=logout | |
[info] Time: Sat Jan 19 2019 02:32:59 GMT+0000 (Western European Standard Time) Timestamp: 1547865179339
[info] Test case passed
```

Figure 7.1: Katalon - Running a test case on Katalon that attempts to login and logout an existing user.

7.3 NEO Node Tap

To unit test the NEO Node Tap, the `unittest`[3] test framework, a Python standard library, was used.

Similarly to the API tests, a `setUp` and `tearDown` methods were defined, which create a new database session before running each test, and rolls-back every transaction made after each test has run, as seen in Listing 7.3.

```python
import unittest
import json

from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from models import Base, TestCase
from settings import database_url_tests

class TestCaseAssertion(unittest.TestCase):
    # uses test database
    engine = create_engine(database_url_tests)
    Session = sessionmaker(bind=engine)
    session = Session()

    def setUp(self):
        """
        setUp is called when tests start to setup the database
        """
        Base.metadata.create_all(self.engine)
        # connect to the database
        self.connection = self.engine.connect()

        # begin a non-ORM transaction
        self.trans = self.connection.begin()

        # bind an individual Session to the connection
        self.test_session = self.Session(bind=self.connection)

    ...

    def tearDown(self):
        """
        tearDown is called when tests end to clean reset database
        """
        self.test_session.close()
        # rollback - everything that happened with the
        # Session above (including calls to commit())
        # is rolled back.
        self.trans.rollback()
        # return connection to the Engine
        self.connection.close()
        Base.metadata.drop_all(self.engine)
```

Listing 7.3: tests.py - Setting up the test environment.

The tests unit test the different components of the application, specified in Section 6.2 - NEO Node Tap. As seen in Listing 7.4, test cases are creating by subclassing the `unittest.TestCase` class, and should be defined by methods whose names start with `test`, so the test runner knows they should be run as test cases. At the end of the test, the `assert` method is called, which compares a given result with the expected result.

[3] `https://docs.python.org/3/library/unittest.html`

```
class TestCaseAssertion(unittest.TestCase):
    ...

    def test_test_case_creation(self):
        # directly create test into the database
        test_case = TestCase(contract_hash='
            f3da12622e9bb2b3f367a650a81fd8c70b2eb495',
                transaction_hash='c33fd08be47a978778f1c7098804d8339ce',
                event_type='SmartContract.Runtime.Log',
                expected_payload_type='String',
                expected_payload_value='Contract was called',
                active= True
            )
        self.test_session.add(test_case)
        self.test_session.commit()
        t_case = self.test_session.query(TestCase).filter_by(
                contract_hash='f3da12622e9bb2b3f367a650a81fd8c70b2eb495',
                transaction_hash='c33fd08be47a978778f1c7098804d8339ce',
                event_type='SmartContract.Runtime.Log').first()
        self.assertEqual(test_case, t_case)
```

Listing 7.4: tests.py - Testing the creation of a new test case in the database.

When the test runner is called, it runs all the tests present in the `tests.py` file and compile a report.

Chapter 8

Conclusion

This internship aimed to study and implement a tool capable of automating the validation and testing of smart contracts during their development.

During his research of the blockchain state of the art, the intern concluded that two of the most promising blockchain technologies, Ethereum and NEO, were in different states concerning the maturity of the existing tools for development of smart contracts. In Ethereum, tools such as Truffle Suite and the Remix IDE could be used to develop and test smart contracts, while others, such as Ganache, could be used to provide the user with a local private network for smart contract deployment. NEO, on the other hand, despite featuring useful tools such as NeoCompiler Eco, a web application which provides a server-side private blockchain for developing, deploying and interacting with NEO smart contracts, lacked tools that enabled the developer to create automated tests for these contracts.

By taking advantage of the features already present in the NeoCompiler Eco application, the intern managed to successfully implement a test automation framework on top of it. This framework allows users to create, import, save and execute smart contract tests, without having to run anything else than a browser. The intern also developed a NEO blockchain event listener, able to catch events emitted on a NEO blockchain. By comparing the resulting event from the test case execution with the expected test case output, the tool is then able to assess whether the test has passed or failed, and present the user with a small test report.

This tool fulfils the internship and the company's goals of having a tool capable of automating smart contracts testing, while also providing, in the backend, the required infrastructure to run those tests. This backend infrastructure improves the quality and easiness of developing smart contracts for the NEO blockchain, while the implementation of the test tool provides greater assurance that these smart contracts are functioning as intended. Furthermore, a pull request with this tool was made to the NeoCompiler Eco tool repository so that it can be used by NEO developers who use NeoCompiler Eco.

8.1 Future Work

The intern planned to use Docker to encapsulate the NEO Node Tap module of the system, an objective that, despite not having a significant impact in the project state, was not achieved due to time constraints. This remains, as such, as a feature to be implemented on further iterations.

During the discussion of the project, both the intern and Blocksmith also considered that, despite not being a part of the scope of the project, adding static analysis to the smart contract code editor would be a useful and interesting feature to implement on a next version of the tool.

References

[1] Remix, Ethereum IDE - Terminal. `https://remix.readthedocs.io/en/latest/terminal.html`. Accessed: 2018-09-30.

[2] Ieee standard for system, software, and hardware verification and validation. *IEEE Std 1012-2016 (Revision of IEEE Std 1012-2012/ Incorporates IEEE Std 1012-2016/Cor1-2017)*, page 25, Sept 2017.

[3] David J Anderson. *Kanban: successful evolutionary change for your technology business*. Blue Hole Press, 2010.

[4] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In Matteo Maffei and Mark Ryan, editors, *Principles of Security and Trust*, pages 164–186, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.

[5] J.J. Bambara, P.R. Allen, K. Iyer, R. Madsen, S. Lederer, and M. Wuehler. *Blockchain: A Practical Guide to Developing Business, Law, and Technology Solutions*. McGraw-Hill Education, 2018.

[6] bitfly.at. Account 0x5abfec25f74cd88437631a7731906932776356f9 - etherchain.org - the ethereum blockchain explorer.

[7] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A Kroll, and Edward W Felten. Sok: Research perspectives and challenges for bitcoin and cryptocurrencies. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 104–121. IEEE, 2015.

[8] Danny Bradbury. The problem with bitcoin. *Computer Fraud & Security*, 2013(11):5–8, 2013.

[9] Jerry Brito and Andrea Castillo. *Bitcoin: A primer for policymakers*. Mercatus Center at George Mason University, 2013.

[10] Vitalik Buterin. Ethereum: Now going public, Jan 2014.

[11] Vitalik Buterin. A next-generation smart contract and decentralized application platform - ethereum whitepaper, 2014.

[12] Konstantinos Christidis and Michael Devetsikiotis. Blockchains and smart contracts for the internet of things. *Ieee Access*, 4:2292–2303, 2016.

[13] Jamil Civitarese. Technical development, asset prices and market efficiency in alternative cryptocurrencies. 2018.

[14] Ethereum Community. Ether: What is ether - denominations.

[15] NEO Council. Neo dapps competition, 2018.

[16] Tomás Morgado de Carvalho Conceição. Debugging tools for neo blockchain development, 2018. `https://medium.com/blocksmithtech/debugging-tools-for-neo-blockchain-development-4a2f319e0464`.

[17] Kevin Delmolino, Mitchell Arnett, Ahmed Kosba, Andrew Miller, and Elaine Shi. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In Jeremy Clark, Sarah Meiklejohn, Peter Y.A. Ryan, Dan Wallach, Michael Brenner, and Kurt Rohloff, editors, *Financial Cryptography and Data Security*, pages 79–94, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[18] J Dienelt. Understanding ethereum. *New York, NY: CoinDesk*, 2016.

[19] Mark Fewster and Dorothy Graham. *Software test automation: effective use of test execution tools*. ACM Press/Addison-Wesley Publishing Co., 1999.

[20] C. Giardino, M. Unterkalmsteiner, N. Paternoster, T. Gorschek, and P. Abrahamsson. What do we know about software development in startups? *IEEE Software*, 31(5):28–32, Sept 2014.

[21] Doug Hall. Fail fast, fail cheap. *Business Week*, 32:19–24, 2007.

[22] Elaine M Hall. *Managing risk: Methods for software systems development*. Pearson Education, 1998.

[23] Jim Highsmith and Alistair Cockburn. Agile software development: The business of innovation. *Computer*, 34(9):120–127, 2001.

[24] Da Hongfei and Erik Zhanh. Neo white paper.

[25] George F Hurlburt and Irena Bojanova. Bitcoin: Benefit or curse? *IT Professional*, 16(3):10–15, 2014.

[26] Heather A Johnson. Trello. *Journal of the Medical Library Association: JMLA*, 105(2):209, 2017.

[27] Sudhir Khatwani. Neo cryptocurrency: Everything you need to know about china ethereum, Dec 2017.

[28] Malcolm Lerider. Neo to announce the first dev competition intended for blockchain apps eco-development, Nov 2017.

[29] Jon Loeliger and Matthew McCullough. *Version Control with Git: Powerful tools and techniques for collaborative software development*. " O'Reilly Media, Inc.", 2012.

[30] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[31] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[32] Martyn A Ould. *Managing software quality and business risk*. John Wiley & Sons, Inc., 1999.

[33] Jeff Offutt Paul Ammann. *Introduction to Software Testing*. Cambridge University Press, 2 edition, 2017.

[34] Jayachandran Praveen. The difference between public and private blockchain - blockchain unleashed: Ibm blockchain blog, May 2018.

[35] Max Rehkopf. Agile epics: Definition, examples, & templates.

[36] Khaled Salah and Minhaj Ahmad Khan. Iot security: Review, blockchain solutions, and open challenges. 11 2017.

[37] Ian Sommerville. *Software Engineering*. Pearson, 10th edition, 2015.

[38] Economist Staff. Blockchains: The great chain of being sure about things. *The Economist*, 18, 2016.

[39] Nick Szabo. Smart contracts. *Unpublished manuscript*, 1994.

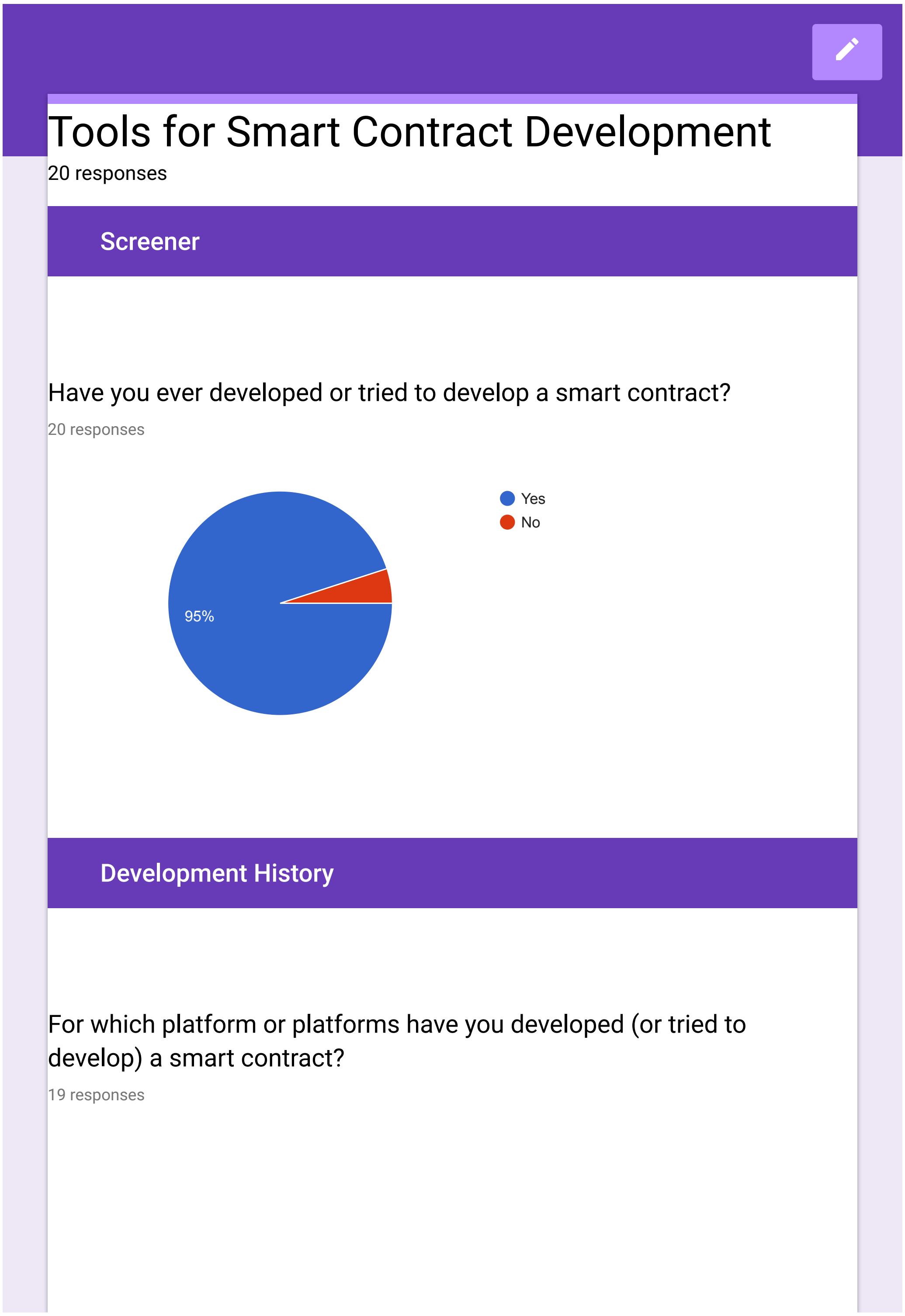
[40] Nick Szabo. Smart contracts: building blocks for digital markets. *EXTROPY: The Journal of Transhumanist Thought,(16)*, 1996.

[41] Fabian Vogelsteller, Vitalik Buterin, et al. Ethereum whitepaper, 2014.

[42] Xiwei Xu, Ingo Weber, Mark Staples, Liming Zhu, Jan Bosch, Len Bass, Cesare Pautasso, and Paul Rimba. A taxonomy of blockchain-based systems for architecture design. In *Software Architecture (ICSA), 2017 IEEE International Conference on*, pages 243–252. IEEE, 2017.

Appendices

Appendix A - A Survey on Tools Used for Smart Contract Development



Tools for Smart Contract Development

20 responses

Screener

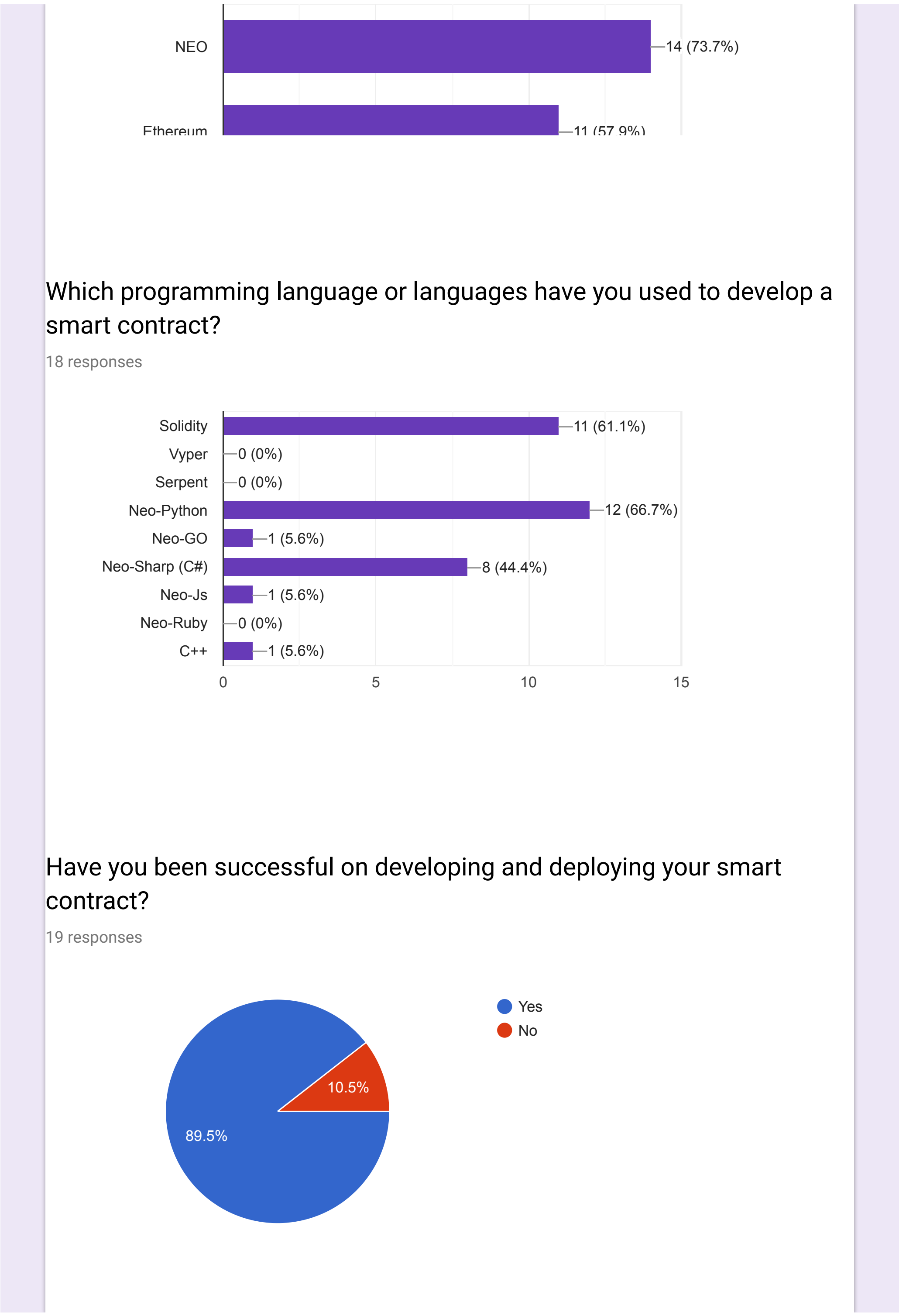
Have you ever developed or tried to develop a smart contract?

20 responses

Development History

For which platform or platforms have you developed (or tried to develop) a smart contract?

19 responses

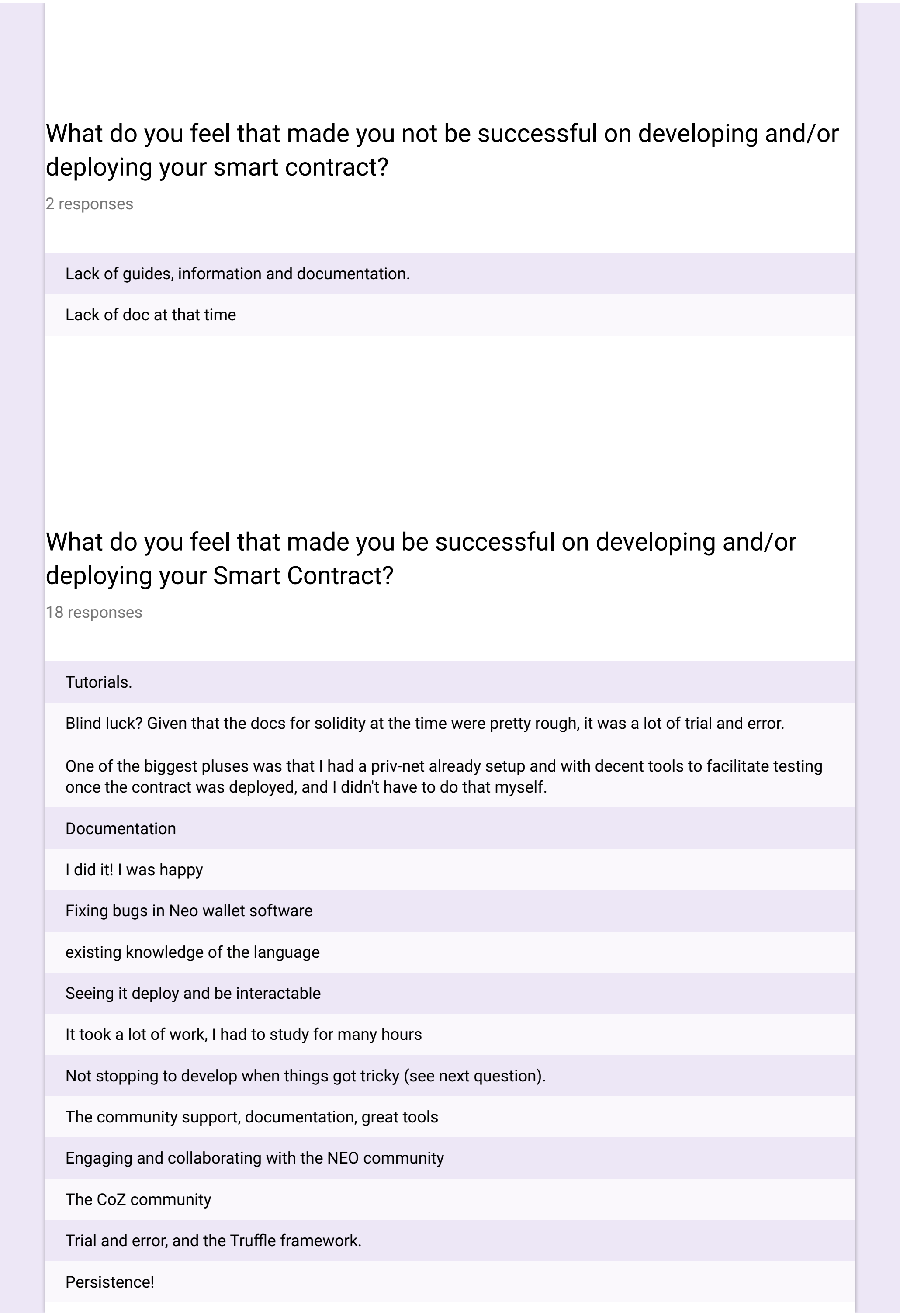


Which programming language or languages have you used to develop a smart contract?

18 responses

Have you been successful on developing and deploying your smart contract?

19 responses



What do you feel that made you not be successful on developing and/or deploying your smart contract?

2 responses

Lack of guides, information and documentation.

Lack of doc at that time

What do you feel that made you be successful on developing and/or deploying your Smart Contract?

18 responses

Tutorials.

Blind luck? Given that the docs for solidity at the time were pretty rough, it was a lot of trial and error.

One of the biggest pluses was that I had a priv-net already setup and with decent tools to facilitate testing once the contract was deployed, and I didn't have to do that myself.

Documentation

I did it! I was happy

Fixing bugs in Neo wallet software

existing knowledge of the language

Seeing it deploy and be interactable

It took a lot of work, I had to study for many hours

Not stopping to develop when things got tricky (see next question).

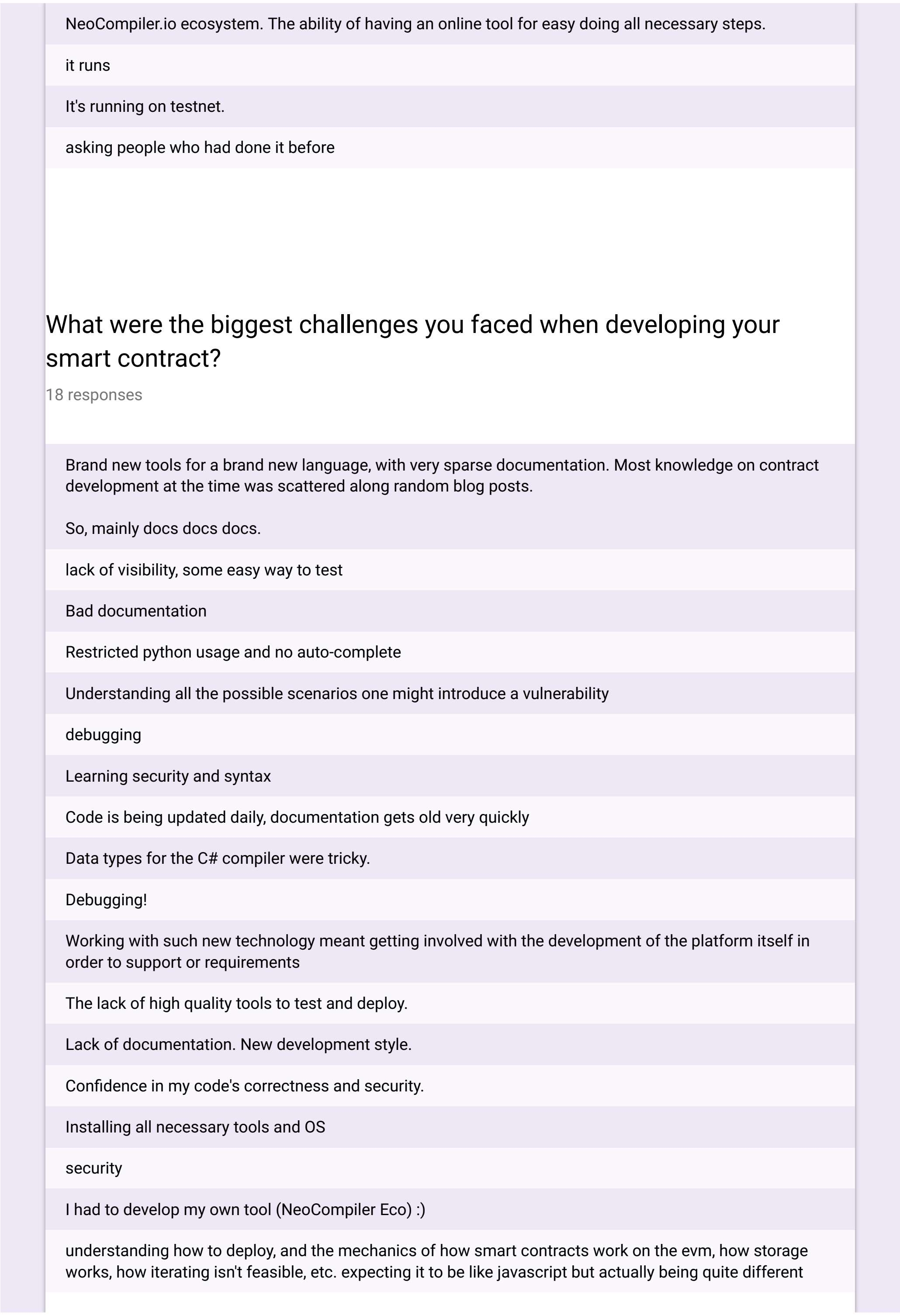
The community support, documentation, great tools

Engaging and collaborating with the NEO community

The CoZ community

Trial and error, and the Truffle framework.

Persistence!



NeoCompiler.io ecosystem. The ability of having an online tool for easy doing all necessary steps.

it runs

It's running on testnet.

asking people who had done it before

What were the biggest challenges you faced when developing your smart contract?

18 responses

Brand new tools for a brand new language, with very sparse documentation. Most knowledge on contract development at the time was scattered along random blog posts.

So, mainly docs docs docs.

lack of visibility, some easy way to test

Bad documentation

Restricted python usage and no auto-complete

Understanding all the possible scenarios one might introduce a vulnerability

debugging

Learning security and syntax

Code is being updated daily, documentation gets old very quickly

Data types for the C# compiler were tricky.

Debugging!

Working with such new technology meant getting involved with the development of the platform itself in order to support or requirements

The lack of high quality tools to test and deploy.

Lack of documentation. New development style.

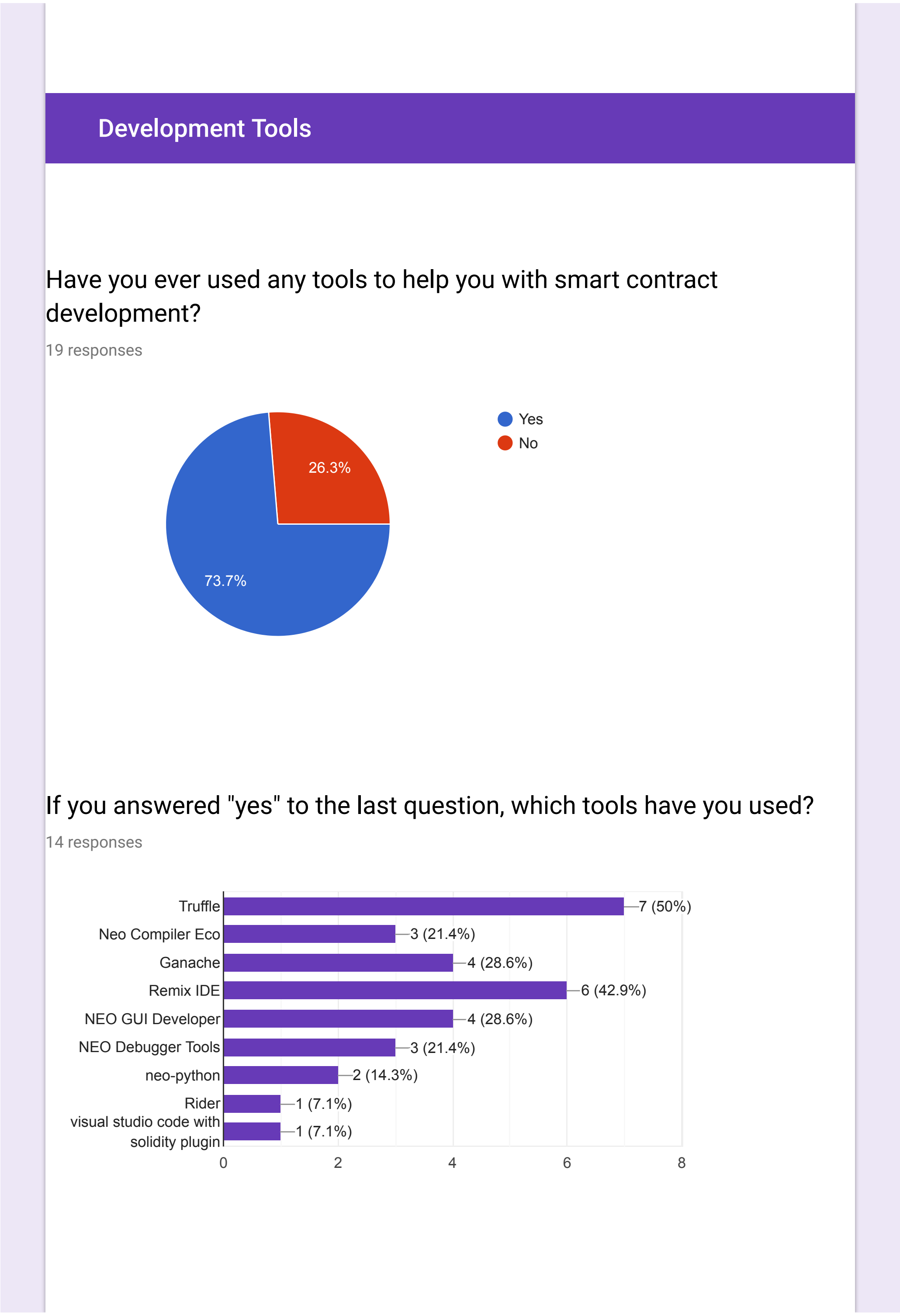
Confidence in my code's correctness and security.

Installing all necessary tools and OS

security

I had to develop my own tool (NeoCompiler Eco) :)

understanding how to deploy, and the mechanics of how smart contracts work on the evm, how storage works, how iterating isn't feasible, etc. expecting it to be like javascript but actually being quite different



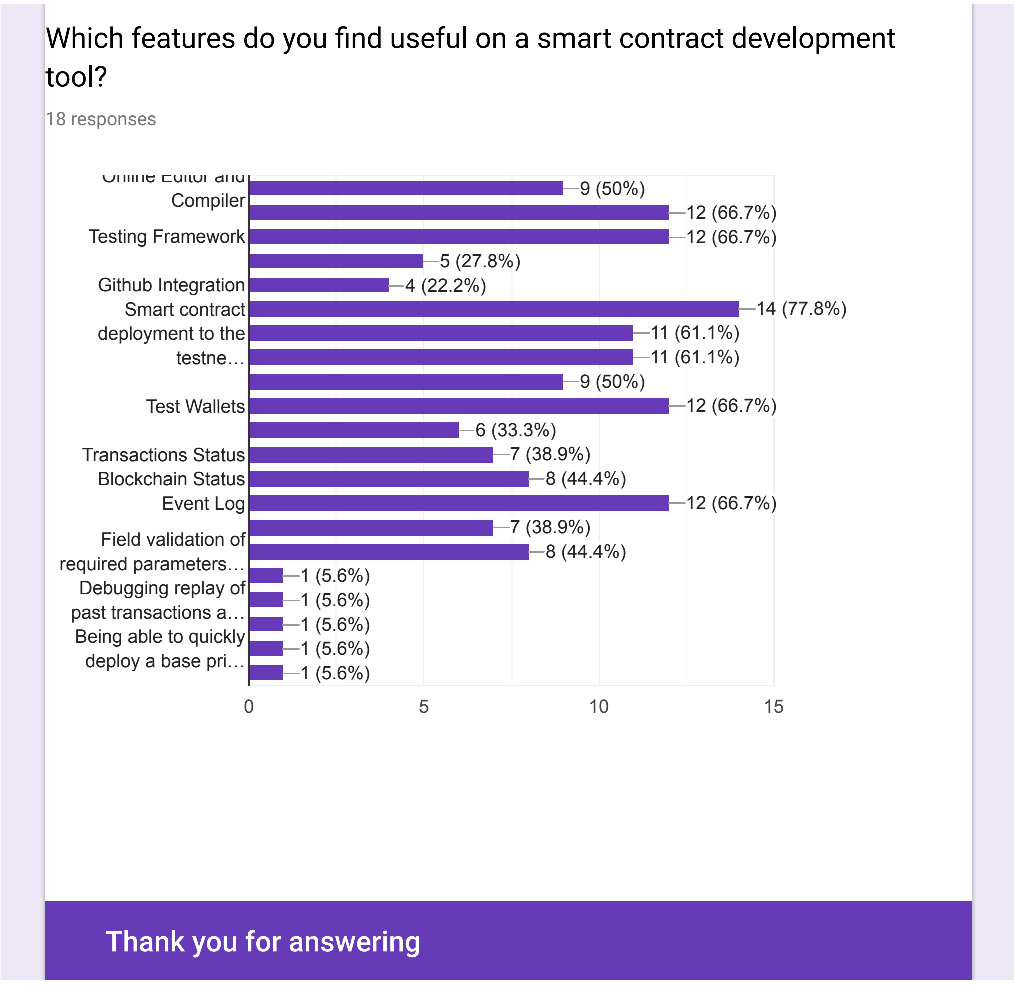
Development Tools

Have you ever used any tools to help you with smart contract development?

19 responses

If you answered "yes" to the last question, which tools have you used?

14 responses

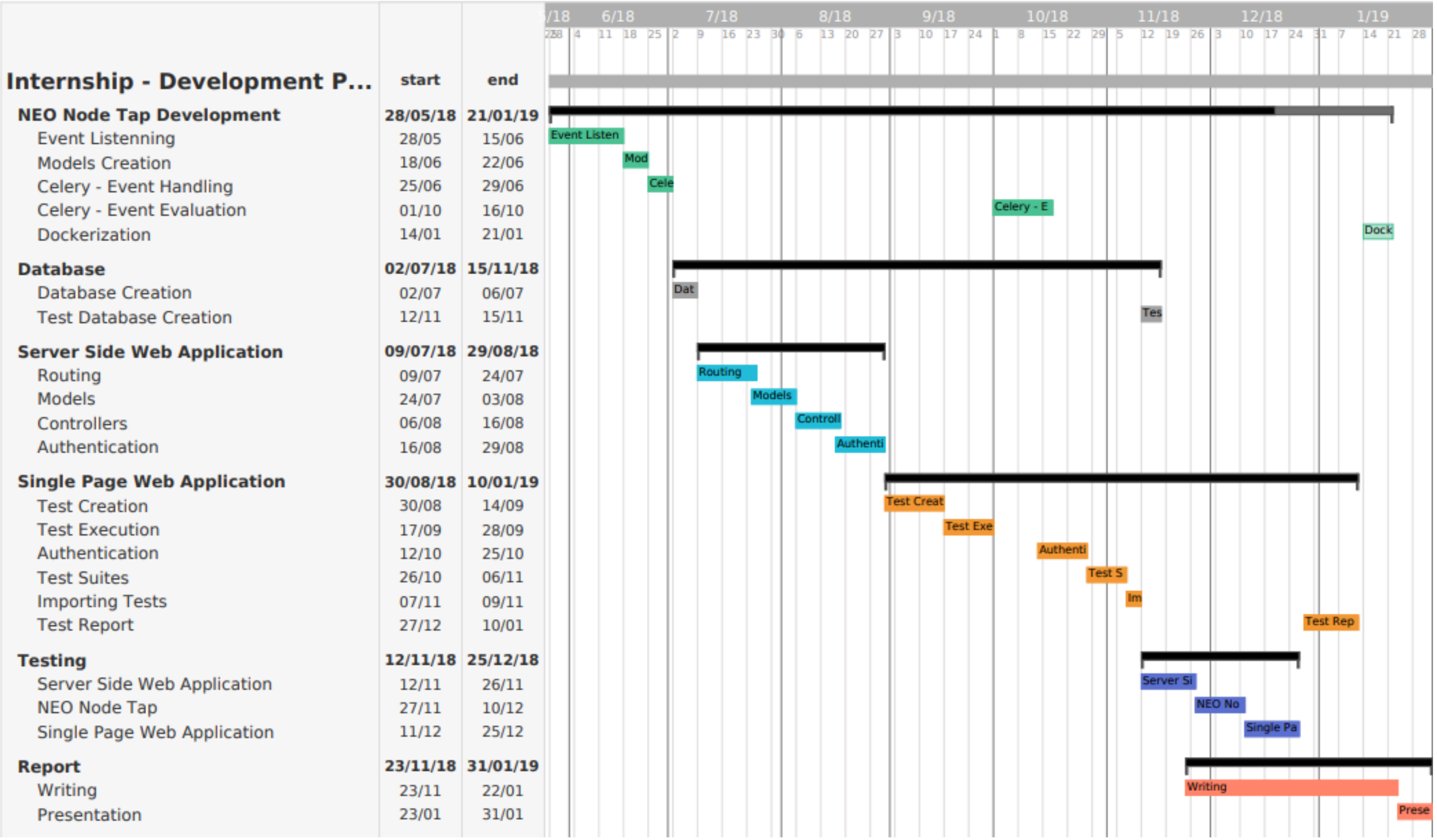


Which features do you find useful on a smart contract development tool?

18 responses

Thank you for answering

Appendix B - Planning of the Second Semester - Gantt Chart



Internship - Development P...	start	end
NEO Node Tap Development	**28/05/18**	**21/01/19**
Event Listenning	28/05	15/06
Models Creation	18/06	22/06
Celery - Event Handling	25/06	29/06
Celery - Event Evaluation	01/10	16/10
Dockerization	14/01	21/01
Database	**02/07/18**	**15/11/18**
Database Creation	02/07	06/07
Test Database Creation	12/11	15/11
Server Side Web Application	**09/07/18**	**29/08/18**
Routing	09/07	24/07
Models	24/07	03/08
Controllers	06/08	16/08
Authentication	16/08	29/08
Single Page Web Application	**30/08/18**	**10/01/19**
Test Creation	30/08	14/09
Test Execution	17/09	28/09
Authentication	12/10	25/10
Test Suites	26/10	06/11
Importing Tests	07/11	09/11
Test Report	27/12	10/01
Testing	**12/11/18**	**25/12/18**
Server Side Web Application	12/11	26/11
NEO Node Tap	27/11	10/12
Single Page Web Application	11/12	25/12
Report	**23/11/18**	**31/01/19**
Writing	23/11	22/01
Presentation	23/01	31/01

Appendix C - Project Tree

NeoCompiler-Eco

```
neocompiler-eco
├── CHANGELOG.md
├── CNAME
├── LICENSE
├── README.md
├── VERSION
├── appHttp.js
├── buildCompilers.sh
├── build_everything.sh
├── compilers
│   ├── docker-compiler-csharp
│   │   └── ...
│   ├── docker-compiler-go
│   │   └── ...
│   ├── docker-compiler-java
│   │   └── ...
│   ├── docker-compiler-js
│   │   └── ...
│   ├── docker-compiler-python
│   │   └── ...
│   └── docker-compiler-ts
│       └── ...
├── config
│   ├── config.example.json
│   └── config.json
├── connections.json
├── controllers
│   ├── index.js
│   ├── test_cases.js
│   ├── test_suite.js
│   └── user.js
├── docker-compose-eco-network
│   ├── docker-compose.yml
│   ├── logs-neocli-node1
│   │   ├── ...
│   ├── logs-neocli-node2
│   │   ├── ...
│   ├── logs-neocli-node3
│   │   ├── ...
│   ├── logs-neocli-node4
│   │   ├── ...
│   ├── logs-neopython-rest-rpc
│   │   ├── ...
│   ├── neo-cli
│   │   ├── configs
│   │   └── wallets
│   └── neo-python
│
```

```
│       └── custom-config.json
├── docker-neo-csharp-node
│   └── ...
├── docker-neo-python
│   └── ...
├── docker-neo-scan
│   └── ...
├── docker-ubuntu-dotnet
│   └── ...
├── dockers-neo-scan-neon
│   └── ...
├── examples
│   └── ...
├── exportingGitCommitVersion.sh
├── express-servers
│   ├── appCompiler.js
│   ├── appEcoServices.js
│   ├── connections.json
│   ├── outputs
│   │   └── ...
│   ├── run-CompilerExpress-RPC.sh
│   ├── run-EcoServicesExpress-RPC-SocketIo.sh
│   ├── socket-js
│   │   └── connections.js
│   └── startAllExpressNohup.sh
├── index.html
├── migrations
│   ├── 2018-11-21-1500-create_user.js
│   ├── 2018-11-21-1515-create_test_suite.js
│   ├── 2018-11-21-1530-create_test_case.js
│   ├── 2018-12-04-1600-add_atachgasfeejs_to_test.js
│   ├── 2018-12-04-1615-add_attachneojs_to_test.js
│   ├── 2018-12-04-1615-add_wallet_invokejs_to_test.js
│   ├── 2018-12-04-1616-add_attachgasjs_to_test.js
│   ├── 2018-12-04-1618-add_invokehashjs_to_test.js
│   ├── 2018-12-04-1619-remove_params_from_test.js
│   └── 2018-12-04-1620-add_invokeparamsjs_to_test.js
├── models
│   ├── index.js
│   ├── test_cases.js
│   ├── test_suite.js
│   └── user.js
├── node_modules
│   ├── ...
├── npm-debug.log
├── npm_prune_install.sh
├── package-lock.json
├── package.json
├── public
│   ├── assets
│   │   ├── coz-tesnet.json
│   │   ├── eco-shared-privanet.json
│   │   ├── localhost.json
│   │   ├── mainnet.json
│   │   └── testnet.json
│   ├── css
│   │   ├── bootstrap-3.2.0.min.css
│   │   ├── bootstrap-theme-3.2.0.min.css
│   │   ├── bootwatch-theme.min.css
│   │   ├── codemirror.css
│   │   ├── fontawesome-all.min.css
│   │   ├── login-modal.css
│   │   └── main.css
│   ├── fonts
│   │   ├── ...
│   ├── images
│   │   ├── ...
│   ├── js
│   │   ├── ace-v133-min-noconflict
│   │   ├── angular-1.2.18.min.js
│   │   ├── angular-1.6.9.min.js
│   │   ├── angular-route-1.2.18.min.js
│   │   ├── angular-route.min.js.map
│   │   ├── angular.min.js.map
│   │   ├── bootstrap-3.2.0.min.js
│   │   ├── bootstrap.min.js
│   │   ├── codemirror.js
│   │   ├── common
│   │   │   ├── blockchain.js
│   │   │   ├── cookies.js
│   │   │   ├── crypto.js
│   │   │   ├── helper.js
│   │   │   ├── neoEditorScripts.js
│   │   │   ├── neon-opt
│   │   │   ├── opcodes.js
│   │   │   ├── sessionManagementScripts.js
│   │   │   ├── testSuiteScripts.js
│   │   │   ├── testsScripts.js
│   │   │   └── wallet_AutomaticClaimAndSend.js
│   │   ├── jquery-3.3.1.min.js
│   │   ├── login-modal.js
│   │   ├── main.js
│   │   ├── neon-js-browser-3.10.0.js
│   │   ├── rawgit
│   │   └── socket.io-2.1.1.js
│   ├── partials
│   │   ├── 404.html
│   │   ├── about.html
│   │   ├── ecolab.html
│   │   └── home.html
│   ├── phonegap
│   │   ├── README.md
│   │   ├── config.xml
│   │   └── createNeoCompiler-Phonegap-Zip.sh
│   ├── templates
│   │   ├── footer.html
│   │   ├── header.html
│   │   ├── login-modal.html
│   │   ├── modal.html
│   │   ├── test-case-modal.html
│   │   ├── test-case-to-suite-modal.html
│   │   └── test-suite-modal.html
│   └── webfonts
│       ├── ...
├── pull.sh
├── runEco_network.sh
├── runHttpExpress.sh
├── seeders
├── stopEco_network.sh
└── stopExpressServers.sh
└── tests
    ├── test_case.js
    ├── test_suite.js
    └── user.js
```

NEO Node Tap

```
neo_node_tap
├── README.md
├── dump.rdb
├── events
│   ├── __init__.py
│   ├── __pycache__
│   ├── exceptions.py
│   ├── models.py
│   ├── runnode.py
│   ├── settings.py
│   ├── tasks.py
│   └── tests.py
├── requirements.txt
└── venv
```

Appendix D - JSON Structure for Test Import

```
[
  {
    "name": "",
    "description": "",
    "contract_hash": "",
    "event_type": "",
    "expected_payload_type": "",
    "expected_payload_value": "",
    "attachgasfeejs": "",
    "attachneojs": "",
    "attachgasjs": "",
    "wallet_invokejs": "",
    "invokehashjs": "",
    "invokeparamsjs": ""
  }
]
```

- ”name” - optional test case name field;
- ”description” - optional test case description field;
- ”contract_hash” - the contract hash of the smart contract which will emit the event;
- ”event_type” - the expected type of the event resulting from the smart contract invocation[1] (i.e.: SmartContract.Runtime.Log);
- ”expected_payload_type” - the expected data type of the event resulting from running the test[2] (i.e.: String, ByteArray);
- ”expected_payload_value” - the expected value of the event resulting from running the test;
- ”attachgasfeejs” - the amount of NEOGas to be paid as fee in the test case execution;
- ”attachneojs” - the amount of NEO to be attached in the test case invocation;
- ”attachgasjs” - the amount of NEOGas to be attached in the test case invocation;
- ”wallet_invokejs” - the NeoCompiler Eco wallet to be used on the test case invocation (i.e.: wallet_0, wallet_1);
- ”invokehashjs” - the hash of the contract to be invoked by the test;
- ”invokeparamsjs” - a stringified JSON with the parameters of the test case invocation;

[1] https://neo-python.readthedocs.io/en/latest/neo/SmartContract/smartcontracts.html#event-types

[2] https://neo-python.readthedocs.io/en/latest/data-types.html