

COEN 241 Term Project

A Blockchain-based Cloud Service

Submitted By : Team 2

Xiao Zhu,

Yali Zhang

Instructor :

Prof. Ming Hwa Wang

Santa Clara University

Preface

This project identifies the difficulties of developing smart contracts in Ethereum, analyzes security problems with smart contracts, and proposes a decentralized platform for programmers and non-programmers to create smart contracts in an efficient and secure manner.

Acknowledgement

We would like to thank Dr. Ming-Hwa Wang for providing us the opportunity to work on this special mini research project and for enlightening us on all the topics we learned throughout the cloud computing course, especially, those related to this project.

Table of Content

Preface	1
Acknowledgement	2
Table of Content	3
List of tables/figures	8
Abstract	10
2. Introduction	11
Objective	11
What is the problem	12
Why this is a project related to this class	13
Why other approach is no good	13
Why you think your approach is better	14
Statement of the problem	14
Area or scope of investigation	14
3. Theoretical bases and literature review	16
Definition of the problem	16
Theoretical background of the problem	17
Blockchains	17
Consensus Protocol	17
Gas System	18
Smart Contract in Ethereum	18
Related research to solve the problem	20
Security Problems in Smart Contracts	20
Transaction-Ordering Dependence (TOD)	21
An TOD bug example: Puzzle Contract	21

Scenario 1: submit-solution is triggered	22
Scenario 2: Both submit-solution and update-reward are triggered	23
Timestamp Dependence	24
Block Timestamp as Random Seed: theRun Contract	24
Block Timestamp as Global Timestamp: lendGovernmentMoney Contract	24
Mishandled Exceptions	25
Exception Caused by “Not Enough Gas”	26
Exception Caused by “Exceeding Call Stack Limit”	27
Reentrancy Vulnerability	27
An Reentrancy Vulnerability Example: SendBalance Contract	27
Recommendations for Fixing Security Bugs in Smart Contracts	28
Guarded Transactions (for Transaction-Ordering Dependence Bug)	28
Deterministic Timestamp	28
Better Exception Handling	29
Challenges of programming in blockchain	29
Unpatchable	30
Open source	30
How to move services to blockchain	30
Starting with the e-voting system	30
Disadvantages of centralized system	31
Implement a e-voting system in blockchain	31
The idea of Micro-Contract	33
Advantage/disadvantage of those research	34
Your solution to solve this problem	35
Where your solution different from others	36
Why your solution is better	37

4. Hypothesis	39
5. Methodology	40
How to generate/collect input data	40
How to solve the problem	40
Algorithm design	42
Transparency	42
Security	42
Privacy	43
Simplicity	43
Language used	43
Tools used	43
Smart contract development	43
Interact with smart contract	43
Middle layer software development	44
Web interface and api	44
A prototype	44
Implementation draft	45
User identification	46
Data storage scheme	46
How to rebuild the data	48
Verify answers according to question source	49
How to generate output	50
How to verify if we meet our goals	50
6. Implementation	51
Code	51
Contracts	51

Survey Contract	51
Survey factory	56
Unit test for contract	57
Test case on rinkeby network	58
RESTful API service	62
Unit test for api service	66
Web Interface	69
Key Modifications	69
Frontend Modifications	70
Backend Modifications	70
Design document and flowchart	85
Data structure	86
Contract structure	87
RESTful API service	88
Web Interface	89
7. Data analysis and discussion	91
Output generation	91
Step 1: create a new contract	91
Step 2: initialize the contract	94
Step 3: Save data to contract	97
Step 4: View statistics	98
Output analysis	99
Compare output against hypothesis	99
Abnormal case explanation (the most important task)	99
8. Conclusions and recommendations	101
Summary and conclusions	101

Recommendations for future studies	102
9. Bibliography	103
10. Appendices	105
Program flowchart	105
Program source code with documentation	105
Source code of contracts	105
Implementation of API methods	112
Input/output listing	128
create	128
gettx	129
init	130
checkQuestion	132
checkAnswer	133
save	134
getStatistic	134
Gas consume	135

List of tables/figures

Figure 1: The blockchain's design in popular cryptocurrencies like Bitcoin and Ethereum. Each Block consists of several transactions [2].

Figure 2: A contract that rewards users who solve a computational puzzle [2].

Figure 3: Visualization of the content of the Puzzle Smart Contract

Figure 4: Scenario 1: submit-solution is triggered for the Puzzle Smart Contract

Figure 5: Scenario 2: Both submit-solution and update-reward are triggered for the Puzzle Smart Contract

Figure 6: A real contract which depends on block timestamp to send out money [11]. This code is simplified from the original code to save space.

Figure 7: PonziGovernmentMental contract, with over 1000 Ether, allows users to participate/profit from the creation/fall of a government [12].

Figure 8: A code snippet of a real contract which does not check the return value after calling other contracts [13].

Figure 9: An example of the reentrancy bug. The contract implements a simple bank account[2].

Figure 10: Code block to define the structs and variables

Figure 11: Code block of function that initialize voters

Figure 12: Code block of defining the vote casting process

Figure 13: Code block for returning the result of voting

Figure 14: flowchart of operations

Figure 15: flowchart of validation when receiving data from public user

Figure 16: data structure and data flow of the back-end contract management for the survey-focused prototype

Figure 17: The survey results page of our survey web interface showing results for a survey from 2 users with the same answers selected as in Figure 17.

Figure 18: Question and answer generate procedure

Figure 19: Key-value generate procedure and result fetching procedure

Figure 20: The abstract structure of cloud service

Figure 21: The process procedure of a typical web interface.

Figure 22: Architecture diagram of the survey web portal.

Figure 23: web interface for creating an survey.

Figure 24: The contracts lists of web interface

Figure 25: Customize the content of the survey.

Figure 26: check the transaction in a public blockchain explorer

Figure 27: run the survey in the demo

Figure 28: view the statistics of a survey

Abstract

Today, Bitcoin and Ethereum are two cryptocurrency giants that need no introduction for many people. Although both of them exploit the blockchain technology, there is one important difference between them which makes Bitcoin a digital money, while Ethereum a “programmable money”. Ethereum not only handles accounts and money transactions like those in Bitcoin, but also stores and executes programs called “smart contracts” on its blockchain. The powerful smart contracts in Ethereum has opened our world to many previously difficult applications without a trusted third party. Tamper-proof votings, online shopping, escrow agents are some of the many possible applications of permissionless computing networks like Ethereum. In this project, we explore various applications of smart contracts, analyze the challenges that may potentially slow down the further adoption of smart contracts, and propose a solution to tackle these challenges. One area of the challenges we study is the security problems that face smart contracts. The high barriers to entry into the cryptocurrency ecosystem for new participants due to the sophisticated semantics of the blockchain technology is another focus when we study the challenges. To tackle these challenges, we develop an smart contracts development platform that allows anyone, programmers or non-programmer, to create, deploy and manage smart contracts on the Ethereum blockchain with simple configurations. Our platform pre-builds a catalog of common contract templates, from which contract owners could simply choose a template and customize it into a deployment-ready smart contract according to their own business needs. They could deploy contracts to the blockchain right from our platform, and manage their contracts afterwards all via the same interface. Our platform would also provide web services for their existing applications to interact with their smart contracts through API calls. Leverage domain expertise, our platform should ensure that all the pre-built contract templates are always carefully designed, implemented, tested, and validated. When this solution is adopted by the industry in the future, more domain experts could contribute contract templates to the platform which makes this streamlined process even more efficient, cost-effective, and secure.

2. Introduction

Being the very first fully-functional cryptocurrency that is truly decentralized, open-source, and censorship-resistant in the world, Bitcoin has received a lot of attention since its invention by Satoshi Nakamoto in 2008-2009[1]. While the blockchain technology was initially introduced for peer-to-peer payments in Bitcoin, it has wider applications after the invention of smart contracts in Ethereum[4,5,6]. Some examples of smart contracts applications are financial instruments like financial derivatives and sub-currencies, and self-enforcing or autonomous governance applications like outsourced computation and decentralized gambling[2].

All users participating in a cryptocurrency network share and help maintain a global ledger of transactions, called blockchain. Unlike traditional banking systems, which usually involves a trusted third-party acting as a centralized intermediary to facilitate the transaction between two parties, cryptocurrency operates in a decentralized manner. Thus, the global ledger is replicated by all network participants, instead of being administered by a central trusted party. If we consider the cryptocurrency network as a finite state machine, the sequence of transactions stored on the blockchain are the events that trigger the change of the state.

A smart contract is an executable program encoded as the payload of a “creation” transaction. To invoke the smart contract, users send an “invocation” transaction to the blockchain. The invocation transaction contains the instructions and the associated inputs for the contract invocation. If the invocation transaction is accepted by the blockchain and has a contract address as the recipient, then all participants on the mining network execute the contract code with the current state of the blockchain, and the transaction payloads as inputs [2].

Although smart contracts are more and more widely adopted in the industry, many challenges that prevent the further expansion of the application are observed. The programming language for writing smart contracts, Solidity, is a new language to developers who are not working in this domain. The immutability of the blockchain makes smart contract “unpatchable” once deployed. Hence, developers only have one chance to get the program logic correct for any smart contract. Moreover, developers have to consider many existing vulnerability issues with smart contracts. Without a thorough understanding of the semantics of the underlying platform on which smart contracts run, it is very easy for a developer to write and deploy a vulnerable contract to the platform and cause huge financial loss. There are many security problems observed in smart contracts that we will discuss in detail in this report.

Objective

To address the various challenges face the development of smart contracts, we propose a smart contract platform that serves as a middle layer between the blockchain infrastructure and

public users who want to utilize the smart contract technology for their business needs. Through providing high-quality smart contract templates on this platform, public users could create smart contracts by customizing the available contract templates through an easy-to-use interface, no programming required. Besides the contract templates, this platform would provide management tools for users to invoke their contracts and query the state of their contracts after the deployment, through the same easy-to-use interface. In addition, we offer web services from our platform for other applications owned by contract creators to integrate with their contracts through API calls. Through our platform, public users, programmers or non-programmers, should be able to build trustful and secure blockchain-based solutions for their business needs. As a proof-of-concept, in this project, we will demonstrate the idea of our solution by providing a prototype platform with one functioning smart contract template for creating tamper-proof surveys.

What is the problem

In recent years, blockchain technology has attracted widespread attention. More and more individuals, organizations and companies show their interest to blockchain technology and are willing to combine blockchain to their own businesses. As blockchain is a very newly emerging technology with very few technical experts, it is very difficult for the public to figure out how to make use of the blockchain technology to build solutions for their business needs.

Furthermore, it is very likely to make mistakes and cause huge losses for a developer who is new to this domain when he attempts to develop a blockchain-based program, more specifically a smart contract, due to the potential security problems observed in the smart contract platforms. Actually, even seasoned Solidity developers could cause huge financial loss when his program is not carefully designed. For example, On June 17, 2016, a malicious user attacked The DAO by exploiting a combination of vulnerabilities and gained control of 3.6 million Ether, around a third of the total Ether that had been committed to The DAO. The affected Ether had a value of about \$50M at the time of the attack[3]. The immutability of smart contracts makes the programming tasks even harder, because the developers only have one chance to design a secure, quality smart contract. Once the contract is deployed to the blockchain, no patch is possible to fix any bug.

Thus, public users need a service to create a blockchain-based application without worrying about all the technical challenges face the smart contract development, just like we can use Gmail without knowing any technique detail on how to setup a mail server. Our idea is to let the experts handle the technical details, the users just need to enjoy the benefits of using the blockchain technology.

Why this is a project related to this class

The cryptocurrency system operates on a peer-to-peer network, while the blockchain itself, the shared ledger, is stored in a distributed manner. Our proposed solution, including our critical data and verification mechanisms, are all built on top of this decentralized system. In order to complete this project, we need to study all the technical details about this decentralized system, many of which are important topics discussed in the cloud computing class.

Moreover, we are trying to build a middle layer between blockchain infrastructure and public user, which can be considered as a Software-as-a-service, another important topic discussed in this class. Our services have to support concurrent users, so we have to guarantee a high scalability for our services. In particular, for each type of contract template, we provide a web service as part of the platform for other applications (e.g. business applications owned by the contract owner) to interact with the contracts. For example, for the survey contracts, when a survey participant submits his responses to the survey application hosted by the contract owner, the submit action will call our web service (through a predefined API) to write the corresponding response to the blockchain. Suppose there are large amount of users taking surveys that created as smart contracts to store responses, our web service will receive large amount of requests. To support such high demand, we may consider using techniques like load balancing to provide a better solution.

Why other approach is no good

Our goal in this project is to support easier development of quality smart contracts, which should be with correct business logic, secure, and efficient in terms of both running time and cost of gas. To address these problems, existing research mainly focus on two areas of improvement: the programming skills of the smart contract developers, in particular those specific coding practices that help improve the security and correctness of the smart contracts, and the robustness of the smart contract development platform itself, i.e. the Ethereum network and the Ethereum Virtual Machine. Both approaches have their own problems when it comes to the implementation time. The first approach requires those developers who are interested in building smart contracts to have relatively thorough understanding of the semantics of the smart contract platform, i.e. Ethereum, in order to adopt those good coding practices. Without a solid understanding of the platform, developers are unable to integrate those recommended coding practices when writing the smart contracts for different business requirements. Comparing to the first approach, the problem with the second approach is easier to spot. Since any blockchain platform is built on top of a decentralized network, whose operations rely on the participation of all the nodes on this network, i.e. individual machines, all

these nodes need to be upgraded to release any change to the semantics of the platform, which is a relatively difficult task to coordinate and requires huge effort to execute.

Why you think your approach is better

Unlike these existing approaches, our approach focus on providing a ready-to-use interface for programmers, and non-programmers to create sound smart contracts directly from pre-built contract templates whose security and correctness were guaranteed by adopting the domain expertise from seasoned smart contract developers. Our approach allows the contract creators, i.e. our platform users, to create smart contracts without worrying about the technical details of the contract platform such as Ethereum. They can create, deploy and manage smart contracts on the Ethereum blockchain with just simple configurations on our platform. At the same time, they can still enjoy the benefits of any improvements recommended by existing researches via the up-to-date contract templates that are pre-built by domain experts in the blockchain world, and via the up-to-date infrastructure of the network that are integrated with and managed by our platform.

Providing efficiency is another advantage of our approach. By providing a catalogue of high-quality contract templates, from which the contract creators can easily build new contracts of the same category, repeated work is avoided. Contract creators no longer build a new contract from scratch for each new business need. Besides the reduced development effort through this streamlined build-from-template process, our platform enables shorter development time, also benefit from the reuse of existing code.

Statement of the problem

With blockchain technology, people can easily deploy secure, transparent products. But there are two main challenges for new participant in the area. One is security, the other one is the high barriers to entry into the cryptocurrency ecosystem. To offer a secure, transparent and easy-to-use service to public, we must solve the follow problems:

1. Development smart contract template with security concerned
2. Build a middle layer software to interact with blockchain and serve requests from users.

Area or scope of investigation

Our scope of investigation in this project includes the following:

- Analyzing challenges face the smart contract development, e.g. security problems, high barrier to enter this area
- Studying other researchers' recommendations on how to tackle the challenges

- Proposing a smart contract development platform that serves as a middle layer between the blockchain infrastructure and public users who want to utilize the smart contract technology for their business needs
- Implementing a prototype for the proposed platform, with a survey type of smart contract template to demonstrate a typical use case of the platform

3. Theoretical bases and literature review

In this section, we discuss all the research papers we reviewed for our own research and discuss how we formulate our solution by utilizing these papers as the theoretical foundation.

Definition of the problem

Since the hit of BitCoin, blockchain has earned wide attention. But, just like the initial phase of the adoption of the Internet, blockchain has not yet reached the stage of being adopted by widespread applications. The main reason is that it is somewhat complicated to get started for users who are new to the ecosystem. Many people in the traditional industries are eager to get involved in this hitting technology but they do not know how to integrate this technology with their existing businesses and/or software systems. There is apparently a need for a easy-to-use service which would allow these people to develop blockchain-based solutions for their businesses without digging into the involved technical details of blockchain.

Smart contracts has made more and more applications possible through storing executable programs in the blockchain. To exploit smart contracts, developers first define the contract in a high-level language called Solidity, which is a Javascript-like language. Once defined, they introduce contracts to the blockchain by “encoding” the contract in a “creation” transaction. In Ethereum, the Solidity code is encoded as Ethereum virtual machine (EVM) code, a low-level, stack-based bytecode language, before being stored in the blockchain. Once deployed to the blockchain, the bytecode-formed contract is immutable and unpatchable. Due to the gas system, each transaction costs a fee to be deployed to the cryptocurrency network. Therefore, the correctness of a smart contract is much more critical than many traditional software programs.

In addition, smart contracts platform like Ethereum is vulnerable to many security problems commonly seen in traditional distributed systems. The cryptocurrency network is a peer-to-peer open (permissionless) network into which arbitrary adversaries may join, especially when the financial incentive for manipulating transactions in a cryptocurrency network is high enough to attract those arbitrary adversaries. Therefore, it is also critical to write secure code when developing for smart contracts.

Given the hard requirement on both correctness and security for developing smart contracts, such task is fairly difficult for users who are new to this ecosystem. Thus, in this project, we are trying to study how to make the provision and deployment of quality and secure smart contracts in Ethereum easier for end users and organizations.

Theoretical background of the problem

Blockchains

Blockchains are used in decentralized cryptocurrency networks such as Bitcoin and Ethereum. While the blockchain is commonly understood as the shared ledger, we can consider it as a chain of blocks with each block containing a set of facts shown in [Figure 1](#) below. If we view the cryptocurrency network as a finite state machine, the blockchain is the data structure that stores a sequence of tamper-proof transactions which may trigger a change to the state of the state machine.

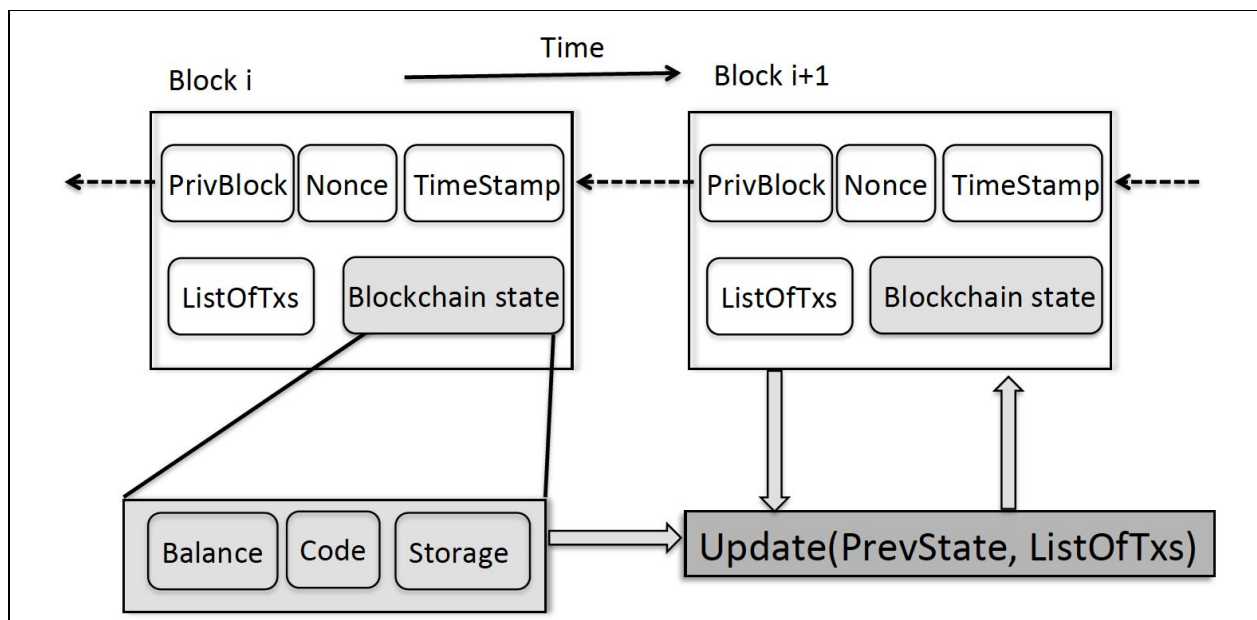


Figure 1: The blockchain's design in popular cryptocurrencies like Bitcoin and Ethereum. Each Block consists of several transactions [2].

Besides a list of transactions, each block on the blockchain contains a link to its previous block, a nonce, and a timestamp to denote its creation timestamp, and the state of the blockchain after it is added. Once a block is added to the blockchain, it can be never modified or removed, i.e. it is immutable. The blockchain state stored in a new block is populated based on the blockchain state in the previous block and the list of transactions stored in this new block.

Consensus Protocol

Since the goal of using cryptocurrencies is to remove the centralized and trusted third-party who acts as the intermediary to perform a transaction in traditional banking systems, the shared ledger of the cryptocurrency network is stored in a decentralized manner. Thus, all participants of the cryptocurrency network replicate a copy of the shared ledger, and update

the ledger accordingly whenever there is a change to the state of the ledger. All network participants of a cryptocurrency network, also called miners, run a consensus protocol to maintain and secure the shared ledger of data, i.e. the blockchain. In every epoch, each miner can propose a block that stores a sequence of new transactions and a timestamp selected by this miner to update the blockchain. However, only one miner, the elected leader, will be selected to update the blockchain in every epoch. During the leader election phase, each miner in this cryptocurrency network competes with each other for the reward of being the first one to solve a proof-of-work puzzle. And the first one who successfully solves the puzzle becomes the leader in this round of election. The leader then broadcasts its proposed block to every network participants, who then updates their own copy of the blockchain accordingly.

Gas System

The leader who wins the election when running the consensus protocol gets some reward for solving the proof-of-work puzzle which usually requires expensive computational power. The rewarding mechanism is part of the cryptocurrency network's gas system. To ensure fair compensation to the network participants for their computation effort contributed to the cryptocurrency network, the cryptocurrency network pays miners fees proportional to their contributed computation. For example, when a user sends a transaction to invoke a smart contract in the Ethereum network, she has to specify both the "gas limit" and the "gas price". The gas limit defines the maximum amount of gas the user agrees to spend on the execution, while the gas price defines the price for each gas unit at which the user agrees to pay for the execution. A miner who includes the transaction in a newly proposed block will be paid the transaction fee corresponding to the amount of gas the execution actually burns multiplied by the gas price. The amount of gas the execution actually burns depends on the content of the transaction, because each type of instruction in the Ethereum bytecode to which smart contracts are encoded consumed a pre-specified amount of gas.

Smart Contract in Ethereum

A smart contract is a computer program that runs on the blockchain. A user defines a smart contract in Ethereum's native programming language Solidity, which will be compiled to Ethereum Virtual Machine (EVM) bytecode for deployment. During the deployment phase, the smart contract is encoded as EVM bytecode in a "creation" transaction, which is included in a miner's proposed block. Once the proposed block is added to the blockchain by the miner, the smart contract is successfully introduced to Ethereum. Once created in Ethereum, a smart contract is uniquely identified by an 160-bit address, has its own private storage, and optionally holds some amount of virtual coins in its account. To invoke a smart contract, a user sends a "invocation" transaction to the contract address in Ethereum with the input data for the invocation and optionally specify the amount of money involved in the execution.

A Smart Contract Example

```
1 contract Puzzle{
2   address public owner;
3   bool public locked;
4   uint public reward;
5   bytes32 public diff;
6   bytes public solution;
7
8   function Puzzle() //constructor{
9     owner = msg.sender;
10    reward = msg.value;
11    locked = false;
12    diff = bytes32(11111); //pre-defined difficulty
13  }
14
15  function(){ //main code, runs at every invocation
16    if (msg.sender == owner){ //update reward
17      if (locked)
18        throw;
19      owner.send(reward);
20      reward = msg.value;
21    }
22    else
23      if (msg.data.length > 0){ //submit a solution
24        if (locked) throw;
25        if (sha256(msg.data) < diff){
26          msg.sender.send(reward); //send reward
27          solution = msg.data;
28          locked = true;
29        }
26      }
27    }
28  }
29 }
```

Figure 2: A contract that rewards users who solve a computational puzzle [2].

[Figure 2](#) shows an example of a smart contract that rewards users who solve a computational puzzle. The constructor function defined in line 8 - 13 runs at the creation of the contract, while the default anonymous function (line 15 - 29 in [Figure 2](#)) runs at every invocation of the contract. During the actual creation of the contract, the information of the owner and the amount of money the owner is willing to reward the puzzle solver are retrieved from a default input variable called msg. The reward money is then transferred from the owner's account to the contract's account. Later, a solution submitter invokes the contract by sending a solution to the puzzle as the input data to the address of the contract. During the actual invocation of the contract, the information of the sender (i.e. the solution submitter) and the input data for the

invocation (i.e. the submitter's solution to the puzzle) is stored in a default input variable called `msg`. When a solution submitter invokes the contract, since the `msg.sender` is the submitter, instead of the owner, line 23 - 29 executes in this case. The submitter's solution will first be verified. The money reward held in the contract's account will then be sent to the submitter's account once the solution is validated. Finally, the contract will be locked. In contrast, when the puzzle owner invokes the Puzzle contract by updating the reward amount, `msg.sender` is the owner, and the additional input `msg.value` (i.e. the amount of money (Ether for Ethereum) to be sent to the contract) needs to be specified. During the actual invocation, the existing amount of money stored in the reward variable is sent from the contract's account back to the owner's account before the new amount of money specified by `msg.value` is sent from the owner's account to the contract's account. And once the transfer of money is done, the reward variable will be updated to reflect the new amount.

Related research to solve the problem

In this section, we present previous research works focusing on problems with the development of quality smart contracts and their recommendations to address the discussed problems.

Security Problems in Smart Contracts

As an alternative to the traditional financial system, a smart contract platform such as Ethereum supports peer-to-peer electronic money transfer in a censorship-resistant manner [7]. On one hand, as smart contracts handle transactions involving virtual coins worth hundreds of dollars per piece, the high financial incentive for attacking a smart contract platform apparently attracts adversaries. On the other hand, since smart contract platforms operate in a decentralized and open (i.e. permissionless) network, the invocation of smart contracts are vulnerable to attempted manipulation by arbitrary adversaries - a type of threat that is difficult to carry out in traditional permissioned networks such as centralized cloud services [8, 9], as anyone can join the network as a participant. Although the users of smart contract platforms like Ethereum follow a set of predefined rules (i.e. a protocol) when participating on the network, research has shown considerable room for manipulation of smart contracts' execution by these participants. For example, Ethereum allows miners to choose which transactions to accept for their proposed new block to be added to the blockchain, to arrange the chosen transactions in any order, and to set the block timestamp at will as long as it is within a 900 seconds window [2]. Thus, Contracts with logic that depends on any of these factors are subject to manipulation by malicious users or miners to gain profit. Miners could exploit these vulnerabilities to steal significant amount of money from the execution of smart contracts. In this section, we first discuss the details of several security bugs seen in the most popular smart

contracts platform, Ethereum. Then we will discuss some of the recommendations proposed by researchers to fix these security bugs. One important note to take is that, to deploy the proposed fixes, all clients in the Ethereum network must upgrade.

Transaction-Ordering Dependence (TOD)

When miners propose a new block to be added to the blockchain, they can choose which transactions to include in the new block and how to order these transactions. And each invocation of a smart contract function is executed via a invocation transaction. Hence, it is possible that multiple invocations of the same contract are initiated by multiple users at about the same time, and are subsequently included in a single block. In this case, a network participant could easily manipulate the execution of this smart contract by carefully arranging the transactions in a particular order, which is known as a transaction-ordering dependence (TOD) bug in smart contracts.

An TOD bug example: Puzzle Contract

We use the Puzzle smart contract discussed in previous section (see [Figure 2](#) for source code of this contract) as an example to illustrate the TOD bug.

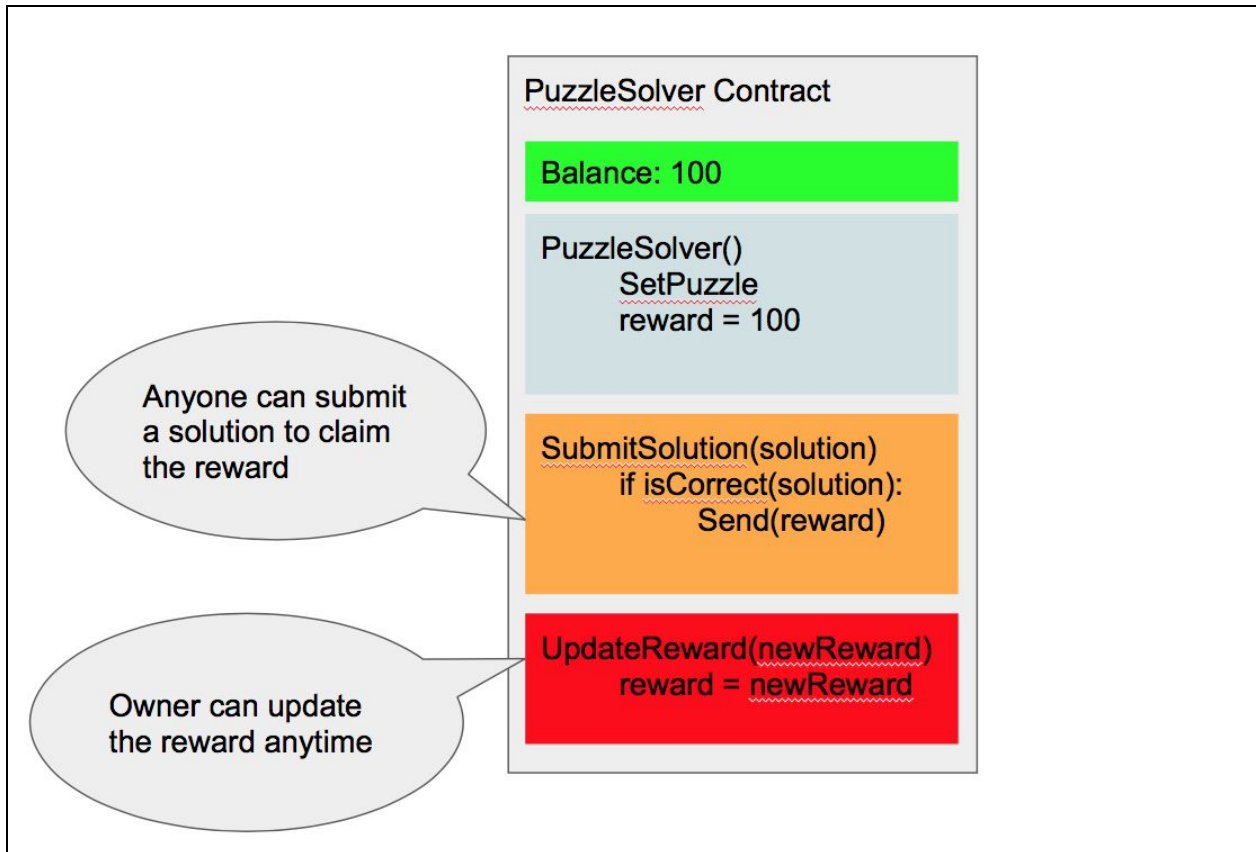


Figure 3: Visualization of the content of the Puzzle Smart Contract

Suppose the owner sets the initial amount of reward for solving this puzzle to 100, then initially the balance of the account of this contract is 100 (transferred from the owner during the initialization) (see [Figure 3](#)). This contract defines two types of invocations: 1) submit-solution to be invoked by anyone on the network; 2) update-reward to be invoked by the puzzle owner only. Now let us compare the expected scenario of an execution of this contract with the unexpected scenario of that.

Scenario 1: submit-solution is triggered

One expected invocation of this contract is for a solution submitter to submit a correct solution to the puzzle via a Ethereum invocation transaction (see [Figure 4](#)).

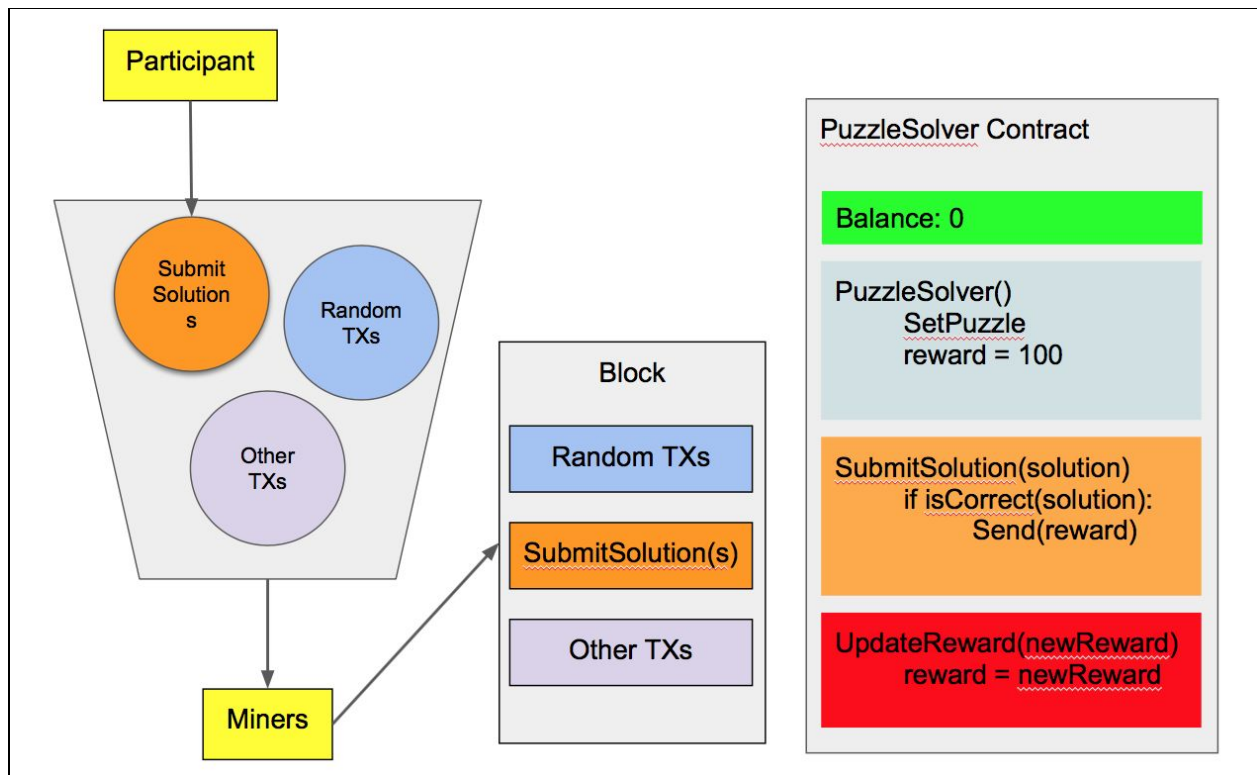


Figure 4: Scenario 1: submit-solution is triggered for the Puzzle Smart Contract

In this happy scenario, the submit-solution transaction together with other irrelevant transactions are included in a block proposed by a miner. During the invocation, the reward of amount 100 is transferred from the contract's account to the submitter's account. After the execution of this transaction, the state of this contract is updated to 0 to reflect the new balance. Then the contract is marked as locked to prevent further submissions of solutions.

Scenario 2: Both submit-solution and update-reward are triggered

Now consider the scenario when a solution submitter submits a correct solution to the puzzle while the puzzle owner is trying to update the amount of the reward to 0 (see [Figure 5](#)).

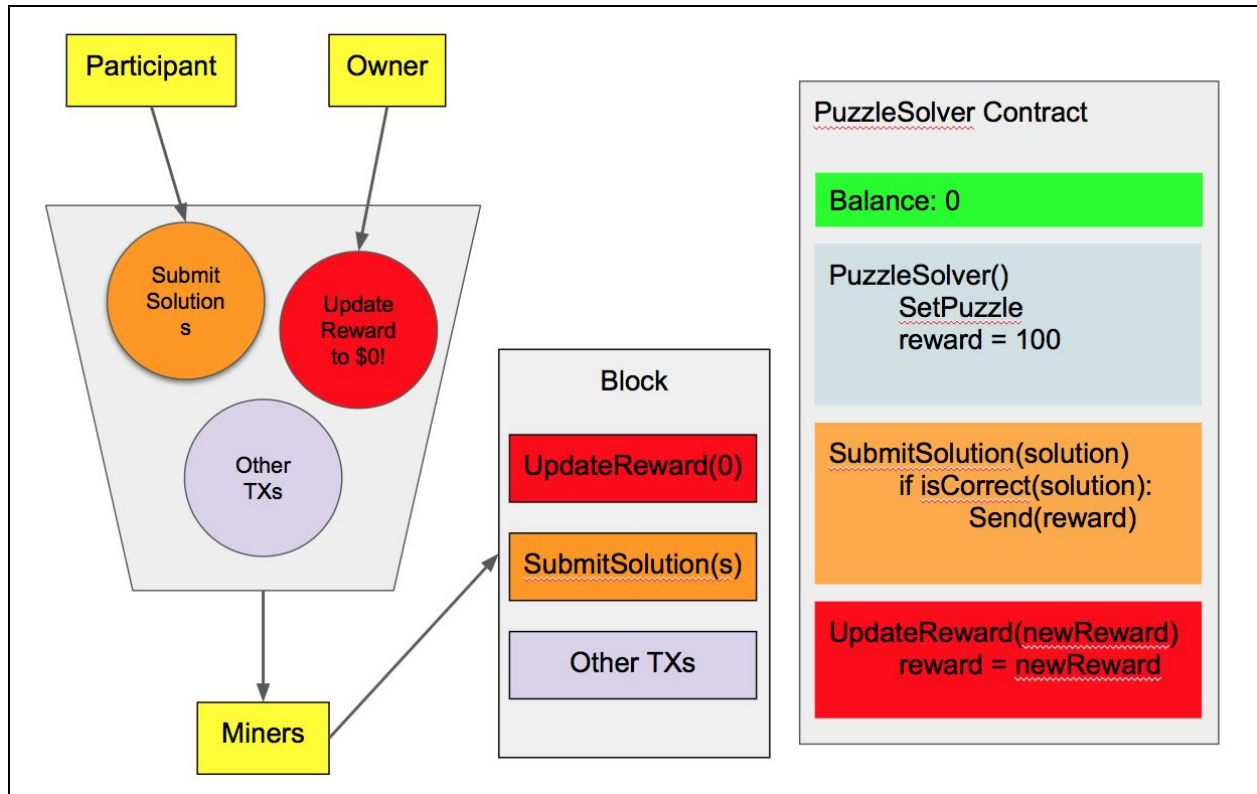


Figure 5: Scenario 2: Both submit-solution and update-reward are triggered for the Puzzle Smart Contract

In this unexpected scenario, the submit-solution transaction and the update-reward transaction, together with other irrelevant transactions are included in a block proposed by a miner. Recall that the Ethereum network allows miners to freely choose which transactions to include in their proposed block and to freely arrange the chosen transactions in any order at their will. The miner who wins the election for adding the block containing the puzzle contract transactions put the update-reward transaction before the submit-solution transaction in [Figure 5](#). The consequence of arranging these two transactions in this order is that the solution submitter gets nothing for solving the puzzle, despite the solution he submitted is correct. During the invocation of transactions in this scenario, the update-reward transaction transfers the reward amount 100 from the contract's account back to the puzzle owner's account before the submit-solution transaction is invoked. So the balance of the contract's account is 0 when the submit-solution transaction is invoked, and therefore regardless of the correctness of the solution submitted, the submitter gets nothing as reward.

The transaction-ordering dependence vulnerability illustrated in the puzzle smart contract example above shows that how the expectation of the state of a smart contract can be different from the actual invocation. This difference can be caused by coincidence, i.e. two transactions just happen at about the same time, and the miner happens to have arranged them in a undesirable order. However, it can be resulted from the manipulation of a malicious user too. For example, when the puzzle owner see a submit-solution transaction from the victim, he may submit the second transaction to update the reward to 0 immediately so that he can have his puzzle solved for free.

Timestamp Dependence

When a miner updates the blockchain by adding a newly proposed block, he needs to set a timestamp as the creation time of the new block. If the execution logic of a smart contract depends on this block timestamp, malicious miners may gain profit from the invocation of this smart contract by manipulating this block timestamp.

Block Timestamp as Random Seed: theRun Contract

[Figure 6](#) shows an example of a smart contract that is vulnerable to timestamp dependence attacks. In this example, the contract depends on the block timestamp to “randomly” generate the amount of money to be sent.

```
1 contract theRun {
2   uint private Last_Payout = 0;
3   uint256 salt = block.timestamp;
4   function random returns (uint256 result){
5     uint256 y = salt * block.number / (salt % 5);
6     uint256 seed = block.number / 3 + (salt % 300)
7                 + Last_Payout + y;
8     //h = the blockhash of the seed-th last block
9     uint256 h = uint256(block.blockhash(seed));
10    //random number between 1 and 100
11    return uint256(h % 100) + 1;
12  }
```

Figure 6: A real contract which depends on block timestamp to send out money [\[11\]](#). This code is simplified from the original code to save space.

Block Timestamp as Global Timestamp: lendGovernmentMoney Contract

[Figure 7](#) shows another example of a smart contract having timestamp dependence. In this example, the smart contract relies on the block timestamp to decide who is the last investor to

share the jackpot amount with the contract owner. This contract collects investments from users. Newly collected investments are paid to previous investors and added to the jackpot. After 12 hours of no-new-investment window, the last investor wins and shares the jackpot with the contract owner [12]. A malicious miner could profit from this contract by manipulating the timestamp of a block to make himself as the last investor.

```
1 function lendGovernmentMoney(address buddy)
2   returns (bool) {
3   uint amount = msg.value;
4   // check the condition to end the game
5   if (lastTimeOfNewCredit + TWELVE_HOURS >
6       block.timestamp) {
7       msg.sender.send(amount);
8       // Sends jacpot to the last creditor
9       creditorAddresses[creditorAddresses.length - 1]
10        .send(profitFromCrash);
11       owner.send(this.balance);
12
13       // Reset contract state
14       lastCreditorPayedOut = 0;
15       lastTimeOfNewCredit = block.timestamp;
16       profitFromCrash = 0;
17       creditorAddresses = new address[](0);
18       creditorAmounts = new uint[](0);
19       round += 1;
20       return false;
21   }}
```

Figure 7: PonziGovernmentMental contract, with over 1000 Ether, allows users to participate/profit from the creation/fall of a government [12].

Mishandled Exceptions

When running computer programs, an exception may be raised due to various reasons. As a computer program, smart contract is subject to exceptions when being executed on the blockchain. Exceptions being raised during the invocation of a smart contract include those specific to smart contracts such as “not enough gas” and “exceeding call stack limit”, and those common exceptions seen in many computer programs like “division by zero” and “array index out of bound”.

In Ethereum, a smart contracts can call another contract by sending instruction to the callee contract or directly invoking the callee contract’s function. When an exception is raised when executing a function in the callee contract, the callee contract terminates the execution, reverts

its state to that before the call, and returns false to mark the abnormal exit. If the call is made via direct invocation of functions in the callee contract, the exception gets propagated to the caller side. However, if the call is made via sending instruction, the caller has to explicitly check the return value in order to find out if the function in the callee contract has been executed properly. Thus, the exception is not handled properly if the caller fails to verify the return value explicitly. Similar to the case when exceptions are not properly handled in any computer program, mishandled exceptions in smart contracts may lead to unexpected and problematic execution results.

Exception Caused by “Not Enough Gas”

[Figure 8](#) shows an example of a smart contract with mishandled exceptions. This contract allows users to claim as king by paying the current king required amount of money. The difference between the price the user pays the current king and the price his successor pays to him after he becomes a king is the profit he earns.

```
1 contract KingOfTheEtherThrone {
2   struct Monarch {
3     // address of the king.
4     address ethAddr;
5     string name;
6     // how much he pays to previous king
7     uint claimPrice;
8     uint coronationTimestamp;
9   }
10  Monarch public currentMonarch;
11  // claim the throne
12  function claimThrone(string name) {
13    /.../
14    if (currentMonarch.ethAddr != wizardAddress)
15      currentMonarch.ethAddr.send(compensation);
16    /.../
17    // assign the new king
18    currentMonarch = Monarch(
19      msg.sender, name,
20      valuePaid, block.timestamp);
21  }}
```

Figure 8: A code snippet of a real contract which does not check the return value after calling other contracts [\[13\]](#).

When a user claims the throne, the KingOfTheEtherThrone contract sends an instruction to the compensation contract for the payment to the current king in line 15 without verifying the return value of the call. So regardless of the execution result of the compensation call, the user is assigned as the new king, and the current king may lose his throne without getting any compensation in the case that the compensation call does not execute properly. Note that it requires more gas to send money to a contract address than to directly send to the address of a normal account, because it involves additional contract execution of the callee contract. Hence, if the caller contract does not provide enough gas for the execution of the callee contract, not-enough-gas exception may arise during the execution of the callee contract easily. Since it is difficult for a caller contract to know the exact cost of the callee contract beforehand, not-enough-gas exceptions are commonly seen when invoking those contracts that send money to a contract address like the case described in the KingOfTheEtherThrone example.

Exception Caused by “Exceeding Call Stack Limit”

While the “not enough gas” problem described in previous section illustrates a vulnerability triggered by an exception raised from the callee function, a malicious user who claims the throne can deliberately cause the payment to the current king in line 15 to always fail regardless of the execution of the compensation contract. The Ethereum virtual machine limits the depth of the call-stack to 1024 frames. An exception arises if the call-stack’s depth limit is reached during the invocation of a smart contract [14]. A malicious user may prepare a contract to call itself 1023 times before sending a transaction to the KingOfTheEtherThrone contract to claim the throne. This way, the compensation contract will always fail to execute, and the malicious user can claim the throne without compensating the current king.

Reentrancy Vulnerability

Reentrancy is a well-studied vulnerability in smart contract platforms after the famous TheDao hack happens, during which the hacker steals over 3,600,000 Ether, or 60 million US Dollars from the Ethereum platform. A reentrancy problem occurs when a smart contract calls another contract and waits for the callee contract to finish its execution. A malicious callee contract may exploit the intermediate state of the caller contract to gain profit.

An Reentrancy Vulnerability Example: SendBalance Contract

[Figure 9](#) shows an example of a smart contract vulnerable to reentrancy attack. The SendBalance contract sends the current balance of a withdrawer that stored in the internal variable userBalances to the withdrawer by calling the default function of the withdrawer contract in line 12. A malicious withdrawer could keep calling its default function until the balance of the SendBalance contract’s account is 0 or there is not sufficient gas to compensate

further contract invocation, because the withdrawer's balance that stored in the internal variable `userBalances` is set to 0 only after the callee function completes its execution.

```
1 contract SendBalance {
2   mapping (address => uint) userBalances;
3   bool withdrawn = false;
4   function getBalance(address u) constant returns(uint){
5     return userBalances[u];
6   }
7   function addToBalance() {
8     userBalances[msg.sender] += msg.value;
9   }
10  function withdrawBalance(){
11    if (!(msg.sender.call.value(
12      userBalances[msg.sender]))()) { throw; }
13    userBalances[msg.sender] = 0;
14  }}
```

Figure 9: An example of the reentrancy bug. The contract implements a simple bank account[2].

Recommendations for Fixing Security Bugs in Smart Contracts

In this section, we discuss some of the solutions to the security issues discussed in previous section recommended by researchers.

Guarded Transactions (for Transaction-Ordering Dependence Bug)

The solution to the TOD bug is to guard transaction invocations by specifying a guard condition whenever a user sends a contract invocation transaction so that either the user gets an expected output or the transaction fails. To illustrate this solution, we use the Puzzle contract example discussed in [An TOD bug example: Puzzle Contract](#). To guard a submit-solution transaction, the solution submitter should specify a guard condition “reward == R where R is the current reward stored in the contract” when he sends the transaction to Ethereum. This way, either he gets the expected reward amount or the transaction fails and the owner fails to get the submitted solution.

Deterministic Timestamp

Research has shown that block timestamps generally serve two purposes in existing smart contracts: 1) deterministic random seed (in [theRun Contract](#)); 2) global timestamp in a distributed network (in [lendGovernmentMoney Contract](#), [15], [16]). On one hand, users may avoid using block timestamps as random seeds in smart contracts by using better random seeds on the blockchain [17, 18]. On the other hand, users may use block index to model the global

time on the blockchain, since the block index always increment by one. Avoiding block timestamps in smart contracts removes the flexibility for an attacker to bias the output of contract execution that depends on the block timestamp [2].

Better Exception Handling

A obvious solution to better handle the mishandled exceptions vulnerability is of course for users to verify the return value whenever calling another contract. This requires good programming practices from developers of smart contracts. A better solution is for the smart contract platform, Ethereum, to automatically propagate exceptions from callee to caller implemented at the Ethereum Virtual Machine level. In addition, the platform could offer the “throw” and “catch” mechanism for users to proper handle exceptions. However, unlike the previous solution, this fix requires an upgrade at the platform level on which all clients need to act. And even with the proper exception mechanism available, a malicious user could still deliberately plants a bug in a contract to gain profits [2].

Challenges of programming in blockchain

While most people focus on the cryptocurrencies, many administrative activities, authentication sensitive options and even everyday services can be done by utilizing blockchain technology.

The big step in the development of blockchain technology is the emergence of smart contract. Before the smart contract, in order to combine blockchain to their own existing business, one has to rebuild an entire blockchain infrastructure and modify some parts of it or add some new modules. This process is difficult because very few people in the world could fully master the blockchain system, mainly means the Bitcoin system.

Unlike Bitcoin’s blockchain, Ethereum blockchain is designed as a universal platform, through smart contract, one can deploy their own decentralized Apps into the blockchain without rebuilding the entire blockchain infrastructure. Smart contracts are pre-defined pieces of codes, to be integrated in the blockchain and executed as scheduled in every step of blockchain updates. In short, now blockchain is “programmable”. Just like the programmable computer changed the traditional industry, the programmable blockchain is going to change the direction of the internet.

Even with the smart contract, there are many differences between programming in blockchain and programming in traditional ways. Two most significant differences are unpatchable and open source.

Unpatchable

Data immutability is the one of the main characteristics of blockchain, through which the irreversibility of a transaction can be held. A smart contract is created through a transaction with specific input data. Thus, once a smart contract was deployed in the blockchain, the code can't be changed anymore. That's saying you have only one chance to deploy a bug-free program, which is not only a huge challenge during the develop process, but a huge challenge to the maintenance of your development. The old-way procedure of program development must be changed to suit this situation.

Open source

Another challenge of programming in blockchain is that all the programs you have deployed should be open source. Because, consensus is one of the most important characteristics of blockchain. How can i trust your program? An important way to let other people trust your contracts is to show them your code, through which they can verify the contracts indeed act as promised. Otherwise, even by the help of blockchain they cannot trust your code.

As you have to open source all your codes, there is nothing can be hidden in the lines, like a hard coded crypto function to enhance the security, and you should figure out another way to protect your work and build your business. Furthermore, as an open sourced program, if your contract has bugs, it's more likely to be found by others than traditional closed source programs.

To conclude, with these challenges we have talked about above, a contract must be designed carefully by experts to achieve good performance and avoid financial losses.

How to move services to blockchain

In this chapter, we are going to talk about how to move services that usually be done offline or by traditional internet means to blockchain. We are going to take the E-Voting system as an example to show how blockchain technology could benefit our daily life and what problems we will encounter when program in blockchain.

Starting with the e-voting system

The concept of E-Voting is significantly older than blockchain, all known examples so far used means of centralized computation and storage.

Disadvantages of centralized system

Vulnerable. The centralized solution, by its nature, create a single-point-of-failure and is open to hacking/hijacking attempts. For example, the Distributed Denial of Service (DDoS) attacks can harm the server or database and interrupt an ongoing election.

Transparency. The transparency of centralized system mainly depends on the administrator's consciousness, which could be influenced economically by the system itself, like corruption.

Easy-to-use. An easy-to-use system can significantly increase user engagement. Take the paper-to-box election system, one has to go the vote centre, get in a line and wait for his/her vote. It's takes a lots of time. For those who cannot attend some election meetings but have the willingness to vote, these prosperous electoral processes have become an obstacle to user participation. In the other hand, if we offer an easy-to-use product, like a mobile app or internet interface for user, they can vote their votes anywhere they want.

Implement a e-voting system in blockchain

This paper[\[10\]](#) tried to provide a secure voting environment and show that a reliable e-voting scheme is possible using blockchain. Their main aim is to provide an easy-to-use, transparent and low-cost blockchain-based e-voting system in a smaller scale used in university election process online such as, department chairs, university rector, or student councils elections.

They choose ethereum blockchain as their smart contract platform. Because, while Bitcoin is only intended to validate coinage transactions, Ethereum network provides a broader range of use cases, with the power of smart contracts. Many applications, that may normally require a web server, can be run through these smart contracts, without using a server. Thus, it is very hard, if not impossible, to manipulate or harm the source codes of the intended software.

These contracts are written in Solidity programming language, which is a combination of C++ and JavaScript. Smart contracts are executed by the peers of the Ethereum network in every 15 seconds, and they should be validated at least by 2 other users to be activated. After that, functions of contracts can be executed, and contracts can be shared with other candidates.

[Figure 5](#) shows the definition of the data structure and variables the contract use. The chairPerson used varify how has the right to initialize a new election and how can grant other people the right to vote, like a administrator. The vote is defined as a struct in solidity language and collected voters as an array. The data structure itself is pretty much self-explanatory. It contains information about whether a user has voted or not, whether he has the right to vote and who he voted for, etc.


```
address chairPerson;
struct Voter {
    bool isVoted;
    bool hasRightToVote;
    uint8 vote;
    address ID;
}
struct Proposal {
    uint voteCount;
}
```

Figure 10: Code block to define the structs and variables

To make the whole system manageable, the contract also exposed an interface called giveRightToVote to grant someone the right to vote. This function can only be called by the chairPerson, the code is shown in [Figure 5](#).

```
function giveRightToVote(address toVoter) public {
    if (msg.sender != chairPerson ||
        voters[toVoter].isVoted) {
        return;
    }
    else{
        voters[toVoter].hasRightToVote = true;
    }
}
```

Figure 11: Code block of function that initialize voters

The function for voting is show in [Figure 6](#). This function takes one parameter as input, which indicate who the user want to vote for. There are some verification in this function to make sure one person can only vote once and only the person who has the right to vote can cast a vote.


```

function vote(uint8 toProposal) public {
    Voter storage sender = voters[msg.sender];
    if (sender.isVoted || toProposal >=
proposals.length && !sender.hasRightToVote)
return;
    sender.isVoted = true;
    sender.vote = toProposal;
    proposals[toProposal].voteCount += 1;
}

```

Figure 12: Code block of defining the vote casting process

There is another function used to retrieve the result of the voting, as shown in following figure. There is a for loop in this function to find the candidate who has the most votes.

```

function winningProposal() public constant returns
(uint256 _winningProposal) {
    uint256 winningVoteCount = 2;
    _winningProposal=0;
    for (uint8 prop = 0; prop < proposals.length;
prop++)
    if (proposals[prop].voteCount > winningVoteCount)
    {
        winningVoteCount = proposals[prop].voteCount;
        _winningProposal = prop;
    }
}

```

Figure 13: Code block for returning the result of voting

The idea of Micro-Contract

Now, we have a glance of the implementation of the blockchain-version e-voting system. From programming perspective, it is very similar to the programming in traditional ways. The main difference is after the deployment what we can do to maintain this program.

For example, even in this e-voting system, there is a minor flaw that could weaken the transparency of this contract. The contract only store proposal's index in the blockchain, people vote for different proposal indexes. But, in some situation someone could change the correspondence between proposals and indexes, then the voting result could be changed too. The best way is to store the information of proposals into blockchain too.

If someone raises this problems, how can we patch this contract? Actually, we can not update this contract at all.

The only way we can do is to deploy another contract to replace the existing buggy one, which will raise another problem. If this service is public to thousands or millions of people, and we have made great efforts to promote our voting system. Many third-party platforms will also interface with our voting system. If we deploy a new contract, it means that everyone needs to update their contract address to keep their service. If someone does not update, then there will be consistency problems, which is not what we want to see. Making everyone update is a challenging task.

There may be others ways to solve this kind of consistency problems, but we reduce the chance of this happening by utilizing a different design schema, which is pretty much like microservice, micro-contract. We can split our tasks into many small modules, each module does a simple job, as simple as possible. Then we integrate these small modules into a more powerful, rich functional one. If we need to modify some functions, we don't have to deploy the whole contract, instead, we just need to update some modules.

Furthermore, we can reuse those modules in many applications, which is more efficient for both the programming and the blockchain infrastructure.

Advantage/disadvantage of those research

The advantage of [Recommendations for Fixing Security Bugs in Smart Contracts](#) by current research is that once those recommendations are implemented by Ethereum, the task of programming smart contracts will become less challenging for developers. However, most of the recommendations require a platform upgrade, thus, each client participating in the network needs to upgrade as part of the platform upgrade, since the cryptocurrency network is a decentralized system. For those recommendations regarding coding practices, it is difficult to make sure every smart contract developer to adopt those practices.

By studying the e-voting system implemented in blockchain, we summarize the advantage of utilizing the blockchain technology, more specifically the smart contract, as following: the decentralized blockchain network enables a more secure and transparent way of implementing applications. However, we also observe some challenges when adopting this technology, such as the unpatchability of the code, the open source requirement, and so on. These challenges increase the demands on programmers and also increase the chance of insecurity bugs which may cause loss of money.

Your solution to solve this problem

We propose a smart contract platform that serves as a middle layer between the blockchain infrastructure and public users who want to utilize the smart contract technology for their business needs.

First of all, our platform provides a catalog of high-quality smart contract templates for users to choose as the foundation of the smart contract applications they want to create. Users make this choice through an easy-to-use interface, such as a web interface. Once selected a specific template based on their business needs, users build a new contract by supplying appropriate input data, from which we concretize the chosen template into a deployment-ready contract.

Next, the platform deploys the prepared contract to blockchain when the contract owner triggers this through the same interface we discuss in previous step, from which the contract owner can also query the state of his/her contract.

Besides the interface through which the contract owner creates, deploys, and manages smart contracts, our platform provides a RESTful api service to help contract owners to integrate the smart contracts they create through our platform with their existing business applications. We may validate the incoming request before pass it to the blockchain for actual contract invocation. Figure 14 shows a general architecture of our design for the proposed solution we discussed above.

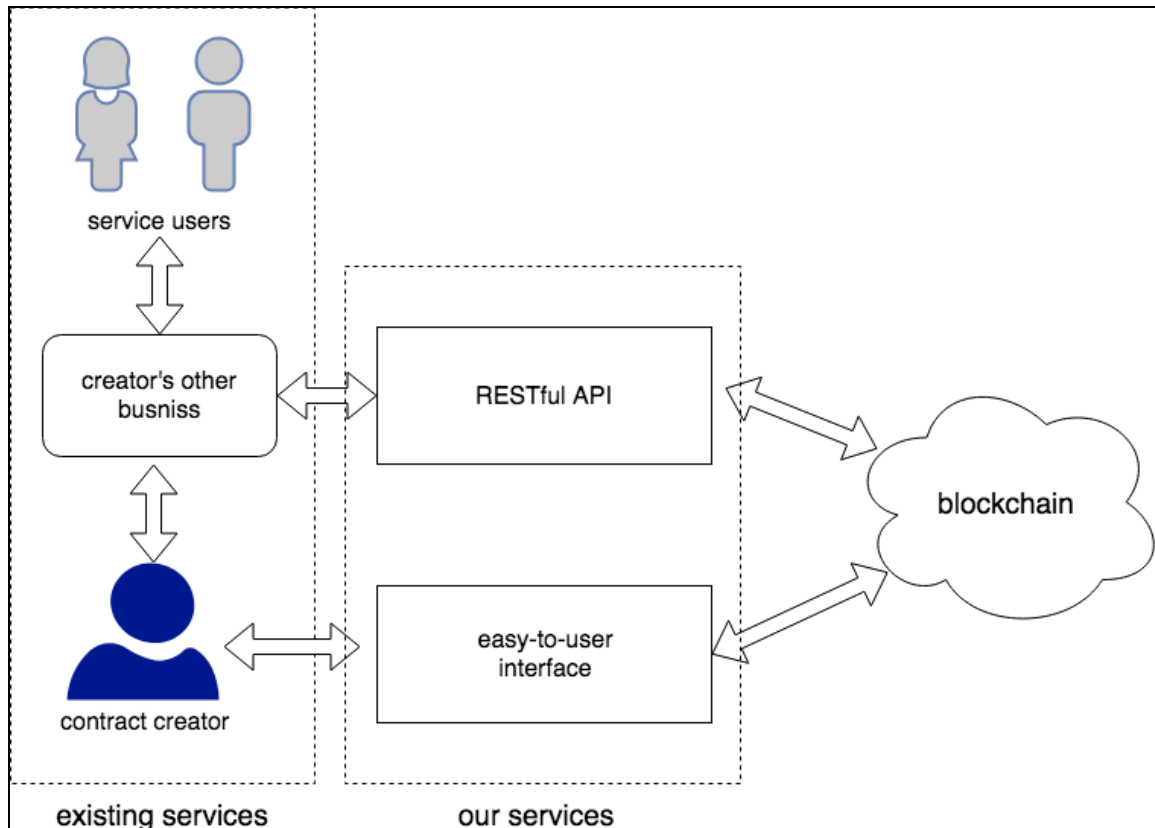


Figure 14: Design of the smart contract platform we propose.

Where your solution different from others

The problem we try to solve in this project is to support easier development of quality smart contracts. Quality smart contracts should be with correct business logic, secure, and efficient in terms of both running time and cost of gas. To address these problems, existing research provides recommendations mainly on two areas.

The first area of recommendations focus on improving the programming skills of the smart contract developers, since better coding practices produce higher-quality applications, in particular, those coding practices specific to the smart contract development. Examples of this type of recommendations for Solidity developers to avoid known vulnerabilities observed in Ethereum include: to always verify the return value of the function call from the callee contract whenever call another contract in the caller contract, to use guard conditions to ensure expected contract invocation result, and to avoid using the block timestamp as the random seed or global time [2]. Most of these coding practices are specific to the smart contract development. Developers have to fully understand the sophisticated semantics of the smart contract platform, i.e. Ethereum, in order to understand why these techniques help in specific circumstances, and therefore to apply these techniques accordingly. It is a relatively high

requirement for new contract developers, and time consuming even for experienced developers. What's more, these recommendations are not even relevant for non-programmer users who are willing to adopt the blockchain technology.

Another area of improvements that existing research recommends is the general improvement of the smart contract platform. For example, it is recommended for Ethereum to automatically propagate exceptions from callee contract to the caller contract when one contract invokes another, similar to how other programming languages handle the exception propagation such as Java and C++ [2]. It is also recommended to return block index in place of block timestamp for the `TIMESTAMP` instruction defined in Ethereum to avoid timestamp dependence bugs [2]. However, to implement these recommendations, we have to upgrade all clients participating in the Ethereum network.

Why your solution is better

To tackle the challenges face new smart contracts developers and users, we propose a smart contract platform that allows anyone to create, deploy and manage smart contracts on the Ethereum blockchain with simple configurations. As our platform relies on domain expertise to guarantee the high-quality of the contract templates on top of which new smart contracts are created, the quality of smart contracts developed using our platform are easier to control.

Comparing to the first area of recommendations by existing research we discussed in previous section which focuses on better coding practices of contract developers, our approach have two advantages. First, it does not limit the usage of smart contracts to developers only. Even non-programmers could use the easy-to-use interface provided by our platform to create smart contracts from the templates. Second, we can get the same benefits of adopting these recommendations using our approach with less effort. We just need to make sure that the domain experts who create contract templates for our platform adopt all the best practices, a much smaller group of developers to advocate than the group of all the contract developers in general. Moreover, the domain experts generally have a better understanding of the semantics of the underlying contracts platform than many contract developers, especially those new to the contract development domain.

The second area of recommendations by existing research focus on a better design of the smart contract platform. To make any newly designed feature of the network to work, such as the automatic exception propagation feature, we need to upgrade all clients on the smart contract network. Comparing to the huge amount of efforts required for this approach, our approach does not touch the underlying blockchain network, and therefore is more lightweight.

Providing efficiency is another advantage of our approach. The current practice for creating blockchain-based applications is usually to build a new solution from scratch for every new

business need. To build the same amount of contracts, this model requires more development resources, longer development time and higher financial cost than following our approach. For each type of application, we only design and implement once for the corresponding contract template. The reuse of the existing contract templates reduces the collective cost of developing smart contract applications.

To sum up, our approach provides a secure and easy-to-use platform for anyone to enjoy the benefits brought by blockchain-based applications without worrying about how to write secure and quality smart contracts.

4. Hypothesis

The aim of this project is to provide a platform, through which anyone, programmer and non-programmer, can deploy their blockchain-based apps. Basically, there are two part:

Part 1: An middle layer software between blockchain and web service, through which we can deploy contracts, interact with contracts automatically.

Part2: A catalog of common contract templates offered to users. In this part, limited by time and resources, we will focus on just one contract template in this project, which is the Survey system, to show our solution will work as we expect. We can scale up to many different contract templates easily as long as we work through this one.

Once we build the middle layer software, we can easily offer an interface for user to interact with the smart contract. But, for limit of time, we will only focus on the creation, deployment, management of the contract in our web interface. We will leave the business related services to the contract creator himself/herself. Take the survey service for example, we will help to create, deploy and manage the contracts, but the contract creators have the responsible for providing a interface for his/her user to complete the survey. The creator's service can send the user's ID, questions and answers to each question to us through our RESTful API service. The data sent to our server will be verified against the data stored in the blockchain to make sure no modifications have been made to any question and any answer by any person.

5. Methodology

How to generate/collect input data

Since we are providing a cloud service, we do not need to collect any big set of input data.

How to solve the problem

In this section, we will demonstrate our solution to the problem by showing how a user leverage our platform to create, deploy, manage a smart contract for storing survey responses, and to integrate our service with the existing survey applications.

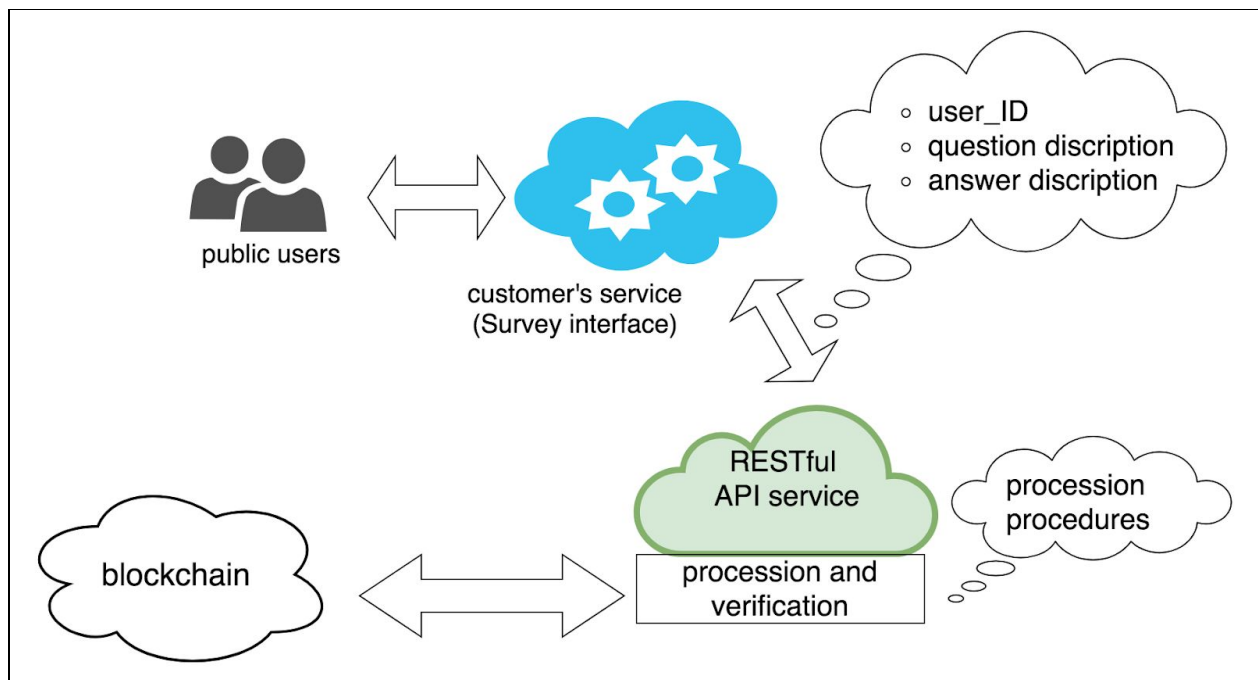


Figure 14: flowchart of our service for survey type of contracts

Figure 14 shows the flowchart of developing the survey type of contracts using our platform. In the contract owner's view, he/she will first complete a registration on our website, offer an Ethereum address, which will be used as the owner of the contract, selection one of the catalog from our service list, which is the survey, customize parameters of the contract template, which include the survey questions and choices for each questions, and then click to submit.

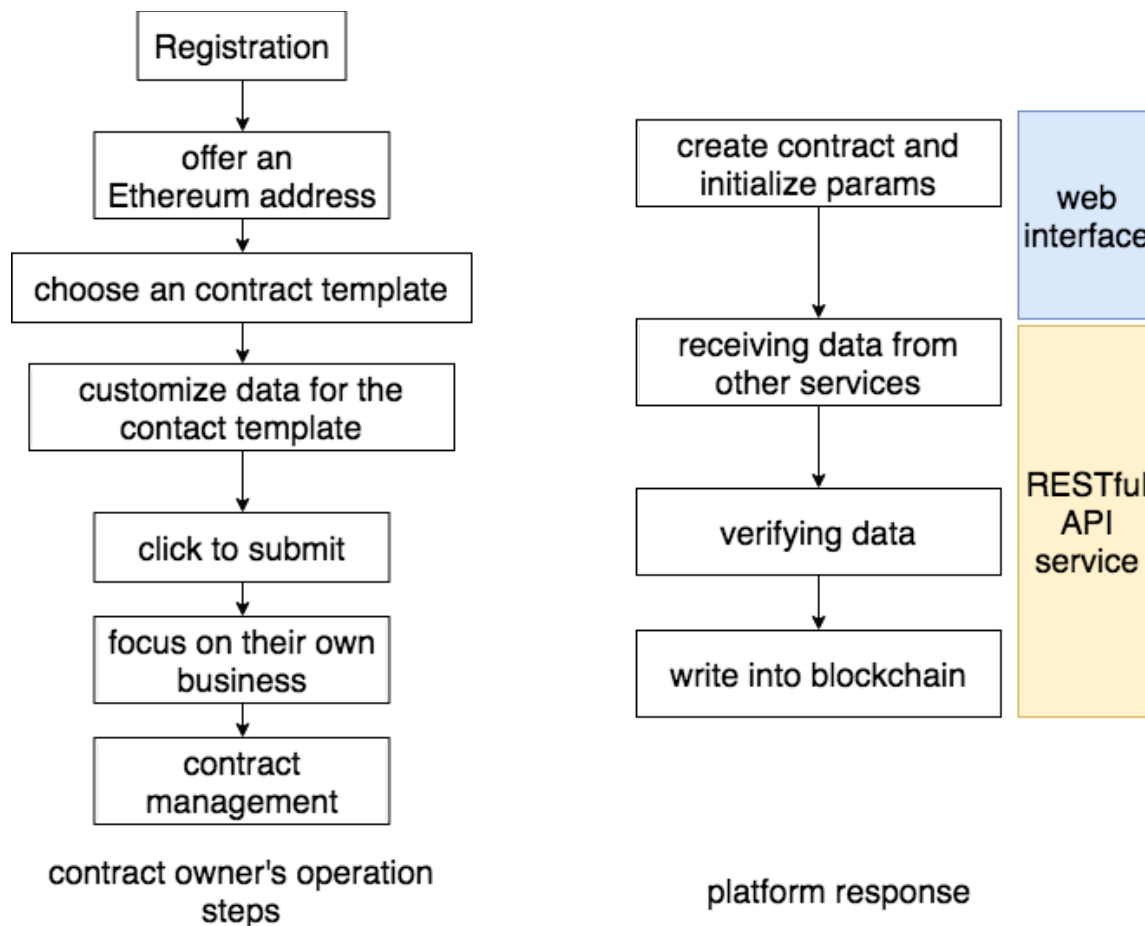


Figure 14: flowchart of operations

Then, our web service will create a new smart contract for this customer from the contract template he/she choose and initialize critical data of the contract such as, the owner of the contact, the question hashes and the hashes of all the answers to each questions.

In the other hand, the contract owner can build his/her own service for his/her own users, for example he/she will provide a interface for public to complete the survey. Once a user complete a survey and click to submit the result, the contract owner's service will invoke our API to interact with the smart contract, more specifically, it should send the user's ID, question and answers in raw data to us through our RESTful API.

When we receive the data, we will first calculate the hashes of the questions and answers to each question, and check whether or not these hashes are existing in the blockchain, as shown the flowchart of Figure 15. Through this way we can make sure, both the questions and answers are not modified by any person in any way. Once this verification is passed, we will then calculate a KEY and VALUE from the submitted data and write the KEY-VALUE pair into blockchain.

In the meantime, the contract creator can manage his/her contract through our web service such as parameters modification, authority management and querying statistical data and so on.

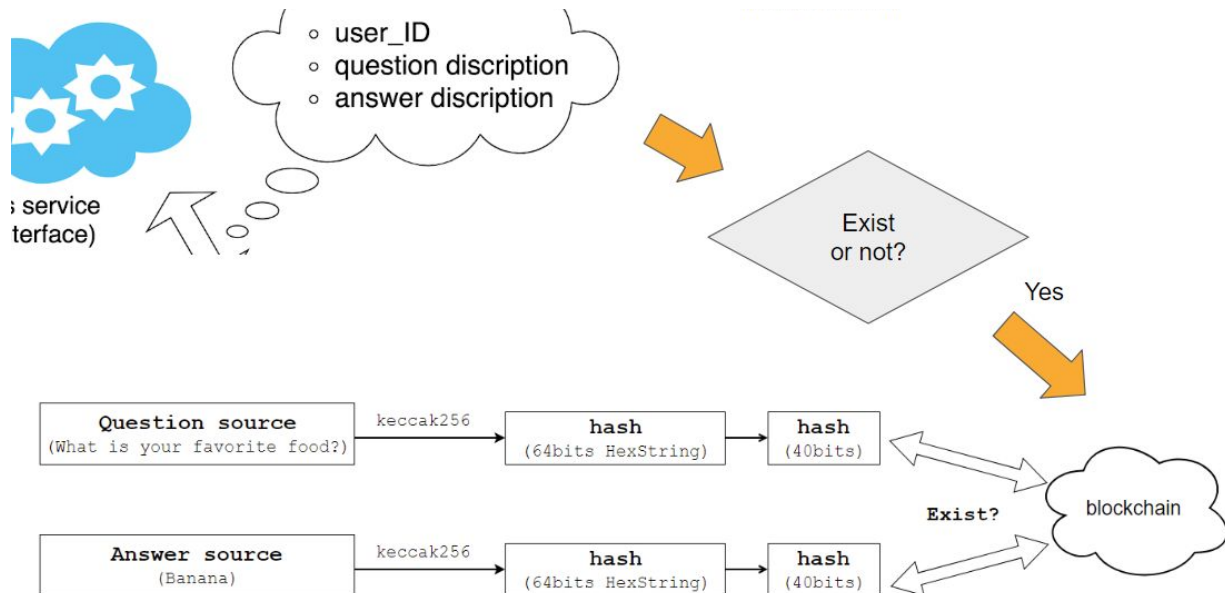


Figure 15: flowchart of validation when receiving data from public user

Algorithm design

As our goal is to build a platform, and the main work of this project is the contract design and the middle layer software development, instead of discussing specific algorithms, we discuss several design considerations in this section.

Transparency

As every transaction will be recorded in blockchain, every answer user submit will be recorded in the blockchain too. Through this way, we can make sure the answer user submitted can be not modified by other people or even by himself/herself.

Security

Design smart contract templates following the microservice model to promote the reuse of code. Leverage domain expertise to tackle the security related challenges.

Privacy

Store hashes instead of raw data on blockchain. For example, we store hashes of user id, question, answer for the survey type of contracts.

Simplicity

We offer an easy-to-use interface, through which user just need to choose a contract template and input some params to create a contract by configuration.

Language used

We use solidity to develop smart contracts. And use javascript to develop the middle layer software and the website interface. The RESTful API service will be implemented through the 'EXPRESS' module in NodeJS. We use another module, mocha, to do the unit tests.

Tools used

Smart contract development

For smart contract development, there is a web IDE programmed in javascript, remix, which is the best IDE so far. In remix, you can develop, compile, debug, analyze and deploy your smart contract. It's also very convenient to interact with contract using Remix.

Truffle is popular smart contract framework, it's a NodeJS package. It offers many commonly used contracts with careful design, and scripts to test, debug and deploy contracts.

Interact with smart contract

Basically, to interact with a smart contract, you just need a ethereum wallet that connected to the ethereum network with a given contract address and a list function definitions of that contract, more specifically the Application Binary Interface(ABI).

Myetherwallet and Metamask are two most popular ethereum wallets. Myetherwallet is a decentralized wallet, it offers interface to manage your wallet without storing your private keys in a server. Metamask is lightweight ethereum wallet as a extension of chrome browser.

This extension injects the Ethereum web3 API into every website's javascript context, so that dapps can read from the blockchain. This is a very helpful tool for during development.

Middle layer software development

Ethereum development team offers many API tools helping developer to interact with the blockchain such as, web3.py for python and web3.js for javascript. In the project, we are going to use web3.js. Because we can use the same language to development both front-end web interface and backend middle layer software.

Web interface and api

The backend and frontend will be two different apps. The backend app is responsible for the interacting with smart contracts, data processing and job management. The frontend is responsible for the interacting with user, processing user's input sending/querying data to/from the backend through RESTful APIs.

We are planning to setup a API service for both our own need and for other developers or organizations who want to use our services, through the *EXPRESS*. Express is a very popular framework to setup a api service with high scalability.

We also need to setup a user management system. There are multiple choices there such as raw PHP, ReactJS framework and so on.

Here is list of tools we are going to use in this project.

1. Solidity online IDE: <http://remix.ethereum.org/>
2. Ethereum online wallet: <https://www.myetherwallet.com/>
3. Metamask:<https://metamask.io/>
4. Truffle framework:<http://truffleframework.com/>
5. Web3 interface: <https://github.com/ethereum/web3.js>
6. Web server: <https://nodejs.org>
7. Web interface: <https://reactjs.org/>
8. RESTful API: <https://expressjs.com/en/4x/api.html>
9. NodeJS backend for SurveyJS library and Editor (for our survey-focused prototype):
<https://github.com/surveyjs/surveyjs-nodejs>

A prototype

Due to the limited time frame given for this project, we are unable to implement a full-blown contract platform that we proposed. Instead, we implement a reduced-scope of the platform that focus on a single category of smart contracts only. Inspired from the e-voting applications that we reviewed when studying existing research in the blockchain domain, we select the tamper-proof survey applications as our main focus for our proof-of-concept system.

Like the originally proposed platform, the core components of our prototype includes a backend application offering RESTful API services and a frontend application providing a user interface for them to use our services. While the backend is responsible for interacting with the blockchain, i.e. the smart contracts platform - Ethereum, processing input data, and managing jobs, the frontend is responsible for interacting with users and acting as the middle layer to pass requests/responses between the users and the backend.

Specific to the survey prototype, we implement a frontend application for survey owners to create or edit surveys and to view results for the surveys. Note that the survey-running service, on which end users could take the survey and submit survey results, should not be part of our contract platform, because users should host their ready-to-run surveys on their own servers to serve their end users, i.e. the survey participants, in whatever way they want. However, the frontend application we implement for this prototype integrates the survey-running functionality to allow users to take surveys right from the same interface for demonstration purpose only. Due to the short timeframe, we implement the most essential functionalities only for demonstration, which means we ignore those non-essential ones such as user authentication/authorization.

We also implement the core component of our proposed contract platform, a backend application, to actually store the key data of the created surveys and survey results on blockchain to achieve the tamper-proof characteristic of the surveys. The simplest use case of our prototype is when the frontend application receives a contract-creation request from a contract owner. The contract owner just needs to provide a valid Ethereum address, e.g. 0xE6F7028d239d061C4265630fe108c4B3128e559B, when creating a new survey via the frontend. The frontend sends a contract-creation request to the backend via the RESTful API with the owner's address as a parameter, e.g. `http://surveyapi.{hosting server}/survey/create?creator=0xE6F7028d239d061C4265630fe108c4B3128e559B`. The backend then creates a new contract for this new survey with the given creator address and sends back the contract address after the contract-creation transaction is successfully broadcasted to the contract network, i.e. Ethereum. An important note to take is that a creation transaction may require some time (typically 15~60 seconds) to be fully processed on the decentralized network, unlike the case in a centralized system, because all nodes need to update their copy of the blockchain in order to accept this transaction.

Implementation draft

Here is a draft of implementation for the survey system, as shown in the following figure. Details of this figure will be discussed in the following sections.

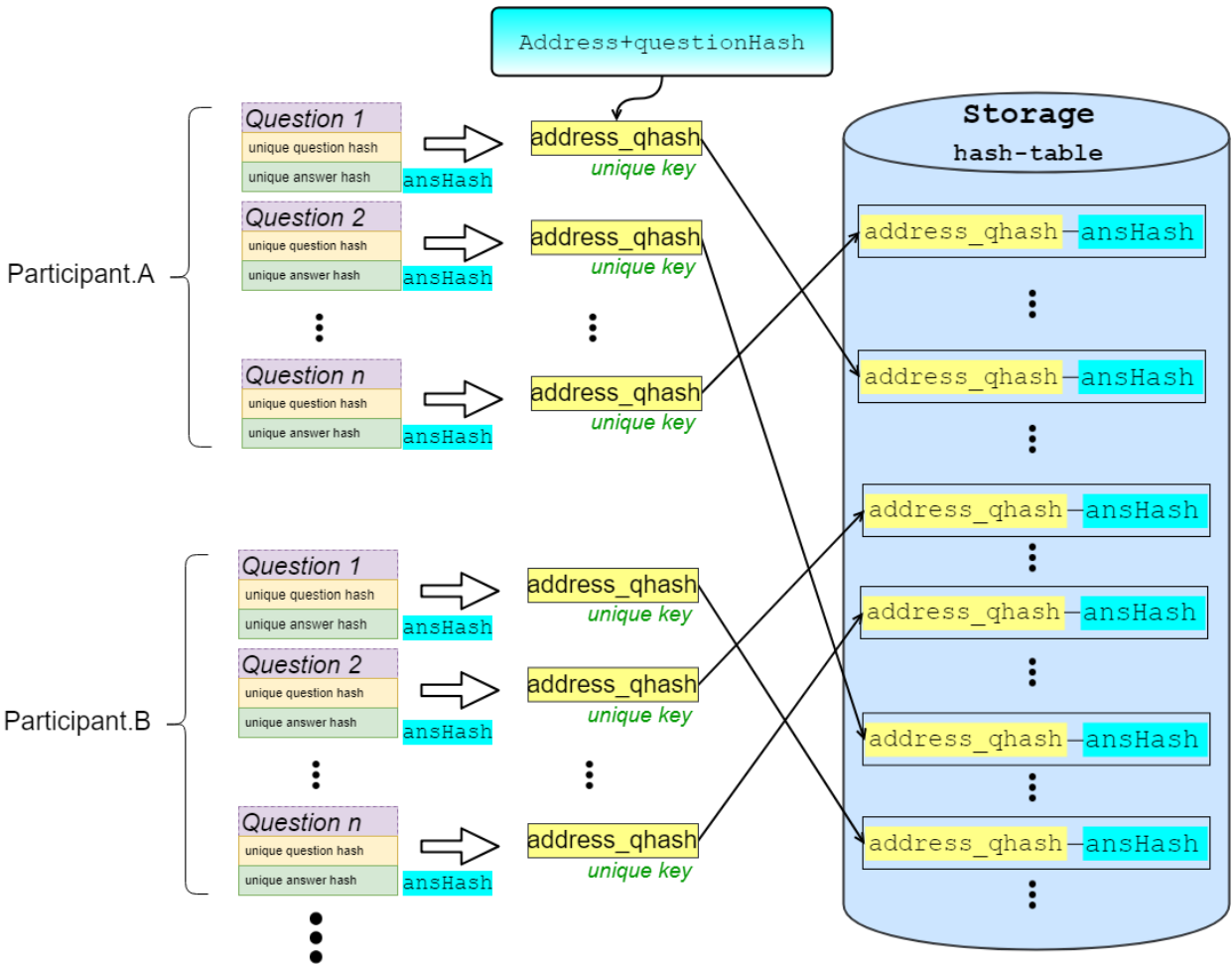


Figure 16: data structure and data flow of the back-end contract management for the survey-focused prototype

User identification

In our system, user are identified by a specific address, more specifically an Ethereum address. The address will be assigned to the user when he/she register in our website. The participant.A and participant.B in [Figure 14](#) are referred to different user who have different addresses.

An ethereum address is a 40 bits hexadecimal string, like:

0xE6F7028d239d061C4265630fe108c4B3128e559B

Data storage scheme

To demonstrate how we store the results of a survey, let's take a single question as an example.

Suppose user A 's address is :

0xE6F7028d239d061C4265630fe108c4B3128e559B (40bits HexString)

For a question as below:

Q1: What is your favorite food?

1. Banana
2. Apple
3. Orange
4. Noodle

We use keccak256[19][20] as hash function. Keccak is a secure hash algorithm, also known as SHA-3, which is also used as the hash function in Ethereum's PoW(Poof of Work) mechanism.

For question Q1, **question_hash** is: **keccak256**("What is your favorite food?") = [0xbb672009c7d71169f611592d39d799bdb151d59dc5228cbe6c5f4bf50707a2d1](#) (64bits)

For the 4 different answers of Q1, the **answer_hash** is:

1: **keccak256**("What is your favorite food?:Banana") = [0xfaf500900993d9b2913a88dfd062083e5f589829af52c0d7fff98ef631f8cfac](#) (64bits)

2: **keccak256**("What is your favorite food?:Apple") = [0x4839cc650c13173a7b3538f4ad79ffe2b16df6da4b93dd285bc8fd9c1d470438](#)

3: **keccak256**("What is your favorite food?:Orange") = [0xf327381623e8026f5260607b9d3b686791f20b66c26df3b46e8652cf8f684938](#)

4: **keccak256**("What is your favorite food?:Noodle") = [0x05e74322aad3496af6ad1624570ae7d0912b9a52b6eefbb7c61fea2f2dcd7fef](#)

First, we store user's address to userList, which is hashmap storage in smart contract.

Then, according to the implementation draft, before we can store the answer to a question into blockchain, we need to generate a KEY from user's address and hash of the question. For the convenience of storage in blockchain, the purpose of multiple verification and storage saving, we restrict the key to 64 bits, which can represent a 32 bytes large number.

Append the first 24 characters of the question hash to the 40 bit long address, then we get a new 64 bit hex string, we use this new string as the key in the hashmap storage.

Key = address + question_hash[:16]

Value = answer_hash[:24]

The keccak256 function will generate a 64 bits hexadecimal string, we use the first 24 bits as the value in the hashmap storage.

Storage[Key] = Value

How to rebuild the data

Step1: read userList from contract by calling function getUsers()

userList = getUsers()

Step2: read questionList from contract by calling function getQuestions()

questionList = getQuestions()

Step3: generate Key through which we can get the answer

Keys = [user+question[:24]] for user in userList for question in questionList

Step4: get answers from contract by calling function getAnswer()

answerStoredInBlockchain = []

for key in keys:

Ans = getAnswer(key)

answerStoredInBlockchain.append(Ans)

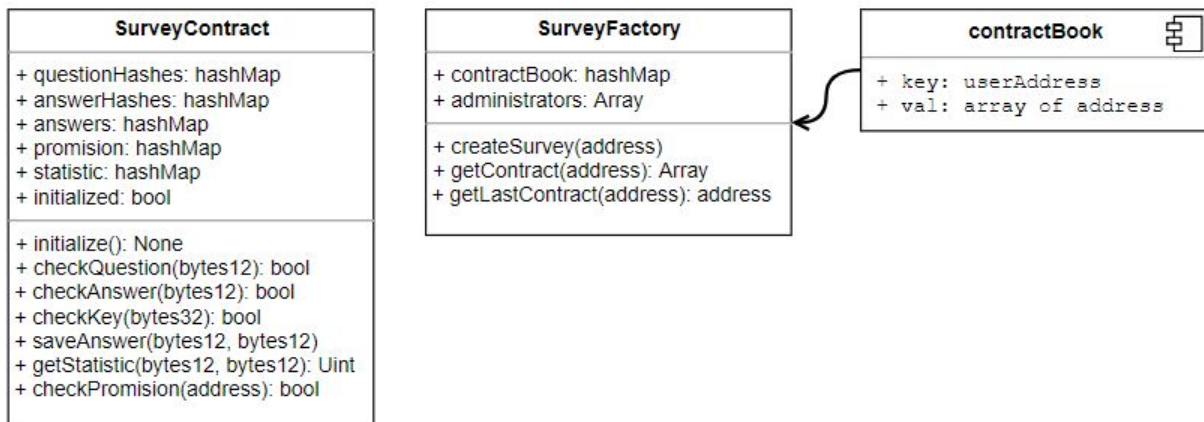


Figure 17: main interface and variables of contracts

Verify answers according to question source

We don't store question source and answer source (Q: What is your favorite food?, A: Apple) into blockchain, instead, we store them into our database, here is how we do the verification.

Let questionSource be the list where all question source are stored, and let answerSource be the list where all answer source are stored.

Then,

```
questionHash = [keccak256(question): question for question in questionSource]
```

```
answerHash = {keccak256(answer):answer for answer in answerSource}
```

As for userA, who's address is addressA:

```
Keys = [addressA+question[:24] for question in questionHash ]
```

```
answerStoredInBlockChain =[]
```

```
for key in Keys:
```

```
    Ans = getAnswer(key)
```

```
    answerStoredInBlockChain .append(Ans)
```

```
for item in answerStoredInBlockChain:
```

```
    if item not in answerHash:
```

```
        print("ERROR: answer not found, verification failed!")
```

```
    else:
```

```
        print("Answer stored in blockchain is: {}".format(answerHash(item)))
```

The program should print:

[What is your favorite food?:Banana](#)

If a user choose **Banana** for question "What is your favorite food?"

Otherwise, we will know the answer stored in the blockchain can't be find in our answer source, either the answer is changed, or the answer source or question source are changed after the survey. Thus, user can verify whether data stored in blockchain are their choices or not, and the verification is clear and concrete.

Meanwhile, user's privacy are also kept. Because, in the blockchain, we only store hashes, other people don't know user's answer to a specific question unless they know both the question source and answer source, which the creator can choose to open to public or keep private.

How to generate output

The output of our system will be some contracts that can hold data of surveys. The contracts will be generated automatically according to the creator's inputs and stored data coming from each user's choices. All these input and output operations are triggered by user and executed by the middle layer software automatically.

How to verify if we meet our goals

As our goal is to provide an easy-to-use smart contract platform for users to create and manage smart contracts, we can verify the success of our project by checking if a user is able to deploy a functional survey (the experiment category of smart contracts we are going to develop for our prototype) contract to the Ethereum platform without writing any source code.

6. Implementation

Code

Contracts

For the survey system, there are two contracts, Survey Contract and Survey Factory. The Survey Factory contract contains the whole copy of the survey contract and expose an interface to create a new survey contract instance, through this way we don't need to keep a copy of the contract source file in our system and can deploy a contract much more efficiently.

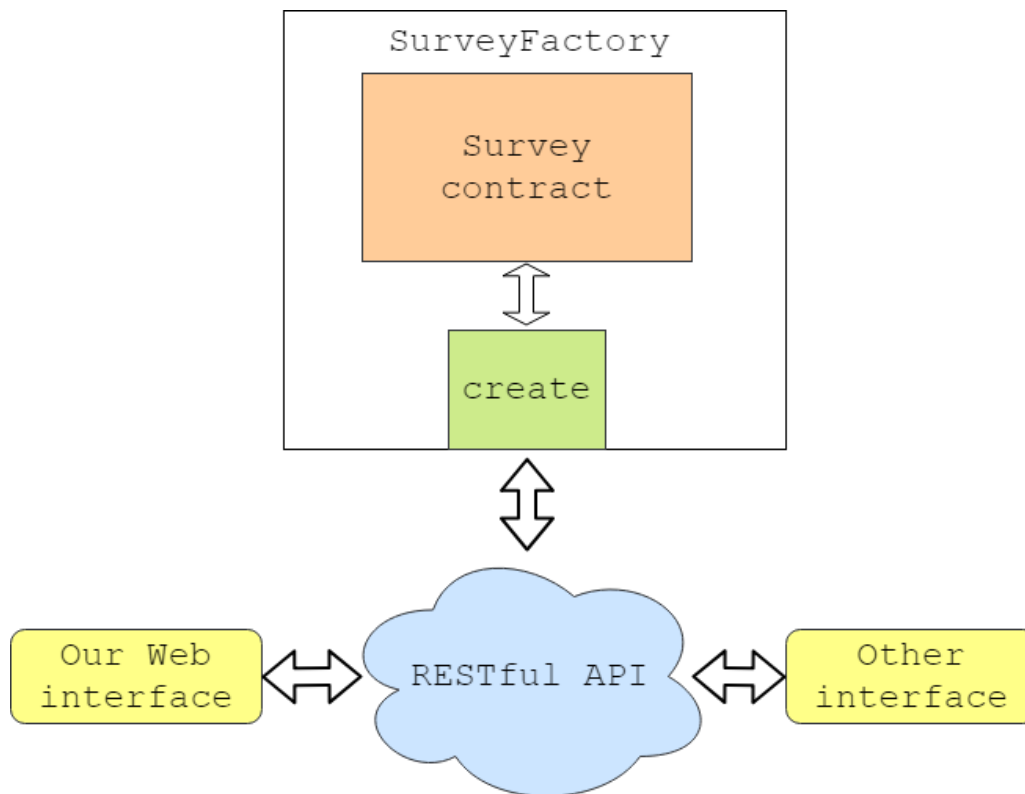


Figure xx. The structure of the contracts in survey system.

Survey Contract

We use a hashMap to store who has the permission to access the key data in this contract. The **questionHashes** and **answerHashes** are the other two hashMap the store the question hashes and answer hashes of a severity, the **ans** is a hashMap to store the key-value pair.

We also define an event called **InitializedBy**. According to the documents of solidity, events allow the convenient usage of the EVM logging facilities, which in turn can be used to “call” JavaScript callbacks in the user interface of a dapp, which listen for these events.

Events are inheritable members of contracts. When they are called, they cause the arguments to be stored in the transaction’s log - a special data structure in the blockchain. These logs are associated with the address of the contract and will be incorporated into the blockchain and stay there as long as a block is accessible (forever as of Frontier and Homestead, but this might change with Serenity). Log and event data is not accessible from within contracts (not even from the contract that created them).

The boolean variable initialized is defined to make sure one survey contract can only be initialized once.

```
mapping(address => bool) internal promision;
mapping(bytes12 => bool) internal questionHashes;
mapping(bytes12 => bool) internal answerHashes;
mapping(bytes32 => bytes12) internal ans;
mapping(bytes12 => mapping(bytes12 => uint96)) statistic;
event DataSaved(address indexed user, bytes12 indexed key, bytes12 indexed value);
event InitializedBy(address indexed by);
bool initialized;
```

A contract is an object which is very similar to the class in C++/JAVA. The **constructor**, is the constructor of this contract, and will be executed once when the contract is created. We can put some initialization work in the constructor, here we add the creator and administrator into the **promision** array.

```
constructor(address creator, address[] admin) public {

    promision[msg.sender] = true;
    promision[creator] = true;
    for(uint i = 0; i < admin.length; i++) {
        promision[admin[i]] = true;
    }
    initialized = false;
}
```

The modifier in solidity offers a more clear and convenient way to restrict the use of a function. Here we defined a modifier, ***needPromision***, to make sure only people have permission can call the function, we will show the usage of this modifier later in detail.

```
modifier needPromision() {
    require(promision[msg.sender]);
    _;
}
```

The function initialize require two parameters, both of them are an array of hashes, the length of with is bytes12. Each hash in the array will be write to blockchain in the for loop. This function can only be executed once, after the first execution, the ***initialized*** will be set to true.

```
function initialize(bytes12[] questionHashArray, bytes12[]
answerHashArray) external needPromision {
    require(!initialized);
    for(uint i = 0; i < questionHashArray.length; i++) {
        questionHashes[questionHashArray[i]] = true;
    }

    for(uint j = 0; j < answerHashArray.length; j++) {
        answerHashes[answerHashArray[j]] = true;
    }
    initialized = true;
    emit InitializedBy(msg.sender);
}
```

The following three function, checkQuestion, checkAnswer and checkKey is pretty simple. They are used to check whether a given input is in the contract or not.

The input of checkQuestion and checkAnswer is the first 12 bytes of the hash of original source. And the input of the checkKey is a 32 bytes data, which can be encoded to 64 bits hex string. The structure of the KEY can be found in next section.

```
function checkQuestion(bytes12 questionHash) public view returns(bool exist) {
    exist = questionHashes[questionHash];
}
```

```

}
function checkAnswer(bytes12 answerHash) public view returns(bool exist) {

    exist = answerHashes[answerHash];
}
function checkKey(bytes32 key) public view returns(bool exist) {
    exist = ans[key] != 0;
}
}

```

The following function, saveAnswer is key function in the contract. We use this function to save each user-question-answer combination to the blockchain. The key contains the information about user's address and the question hash, the value is just part of the answer hash. Through this way, we can make sure each answer to each question from each user can be verified in blockchain.

```

function saveAnswer(bytes32 key, bytes12 value) external needPromision {
    require(!checkKey(key));           //make sure no duplicated key
    require(checkAnswer(value));       // make sure answer is valid
    bytes12 q = last12bytesOf32(key);
    require(checkQuestion(q));         // make sure question is valid
    ans[key] = value;
    statistic[q][value] += 1;
    emit DataSaved(address(bytes20(key)), q, value);
}

```

In the function, the first step is to check the permission of the caller. Only the people whose address is in the contract can call this function, otherwise the call will be reverted in the first time.

Then, we also make sure the key, the question hash and the answer hash are all valid, means they can be found in the contract, otherwise the function will reverted too.

If all requirement are met, we save key-value pair to the blockchain, and update the statistics. The statistics are stored as a two dimensional hashMap, as shown in the function of getStatistic, given question hash and answer hash, the count number of one particular answer to some particular question will be returned. Also, we make this function can only be accessed by people with permission.

```

function getStatistic(bytes12 questionHash, bytes12 answerHash) external
view needPromision returns(uint count) {
    count = statistic[questionHash][answerHash];
}

```

The function checkPromision is used to test whether the given address is in the contract or not.

```

function checkPromision(address user) external view returns(bool
havePromision) {
    require(user != address(0));
    havePromision = promision[user];
}

```

We also have two helper functions, *last12bytesOf32* and *front12bytesOf32*, get get the first 12 bytes or last 12 bytes of a given 32 bytes input data.

```

/**
 * @notice : an helper function to get the last 12 bytes of an 32 bytes data
 * @param src : the original 32 bytes data
 * @return rtn : return the last 12 bytes of the input
 */
function last12bytesOf32(bytes32 src) internal pure returns(bytes12 rtn) {
    rtn = bytes12(src<<(32-12)*8);
}

/**
 * @notice : an helper function to get the first 12 bytes of an 32 bytes
data
 * @param src : the original 32 bytes data
 * @return rtn : return the first 12 bytes of the input
 */
function front12bytesOf32(bytes32 src) internal pure returns(bytes12 rtn) {
    rtn = bytes12(src);
}

```

Survey factory

The **surveyFactory** contract is shown in the following code block. The key function **createSurvey** is pretty simple, it creates a new survey contract instance first, and then put the contract address into a storage, **contractBook**, which can be read by the function **getContract** and **getLastContract** later. Actually, the function can return a value to the caller, but this return value can not be received directly, the contract address will be returned as part of the receipt when the transaction is confirmed.

```
contract SurveyFactory {
    mapping(address => address[]) contractBook;
    address[] public administrators;
    event InstanceCreated(address indexed creator, address indexed
contractAddress);

    constructor(address administrator) public {
        require(administrator != address(0));
        administrators.push(administrator);
    }

    /**
     * @param creator : the creator has the promise to saveAnswer and
getStatistic
     */
    function createSurvey(address creator) external {
        require(creator != address(0));
        address contractAddress = new SurveyInstance(creator, administrators);
        contractBook[creator].push(contractAddress);
        emit InstanceCreated(creator, contractAddress);
    }

    /**
     * @param creator : the creator whose last contract will be return
     * @dev : return all the surveyInstance contract addresses the creator
created
     * as an array
    */
}
```



```

    */
    function getContract(address creator) external view returns(address[]
addresses){
        require(creator != address(0));
        addresses = contractBook[creator];
    }

    /**
     * @param creator : the creator whose last contract will be return
     * @dev : return the surveyInstance contract address the creator created
recently
     */
    function getLastContract(address creator) external view returns (address
contractAddress) {
        require(creator != address(0));
        address[] storage addresses = contractBook[creator];
        contractAddress = addresses[addresses.length-1];
    }
}

```

Unit test for contract

To make sure the contract will act as we designed, we need to fully test every function in every possible conditions.

We use **mocha**, which is a Nodejs module, to create test cases. In every test case of this section, we interact with blockchain directly by using the web3.js lib. The following code shows a portion of the test case procedure. We create a test case named “**can createSurvey**” first, inside the function we call the *createSurvey* function of the contract with correct input parameters. And we expect this call will return some result as we designed, then we make some assertions to make sure the returned result is exactly what we expected.

We also need to set the timeout attributes, because we need to wait 15~60 second, or even longer in some cases, otherwise the test case will fail because of timeout.

```

it('can createSurvey', async () => {

```

```

    try {
      const creation = await
surveyFactory.methods.createSurvey(creatorAddress)
      .send({
        from: accounts[0],
        gas: gaslimit,
        gasPrice: gasprice
      });
      assert.ok(creation['transactionHash'], 'transactionHash is null,
createSurvey failed');
    } catch (error) {
      console.log(error);
    }
  }).timeout(timeoutSec);

it('can createSurvey multiple times', async () => {
  const creation = await
surveyFactory.methods.createSurvey(creatorAddress).send({
    from: accounts[0],
    gas: gaslimit,
    gasPrice: gasprice
  });
  assert.ok(creation['transactionHash'], 'transactionHash is null,
createSurvey failed');
}).timeout(timeoutSec);

it('can getContract', async () => {

  addresses = await
surveyFactory.methods.getContract(creatorAddress).call();
  assert(addresses.length > 0);
  assert.ok(addresses[addresses.length - 1], 'contract address is null,
getContract failed');
})

```

Test case on rinkeby network

We provide 30 test cases for the contract in total. The following block shows the result of the one test run on windows 10, which takes about 10 mins to complete.

To run the test case, we just need to run: `npm test`

```

> surveyapi@0.0.0 test
C:\Users\xzhu\Dropbox\blockchain\eth\projects\surveyapi
> mocha ./test/survey.test.js

SurveyFactory
  ✓ deploys surveyFactory
  ✓ checkaccount
  ✓ have administrator in administrators (280ms)
  ✓ can createSurvey (21102ms)
  ✓ can createSurvey multiple times (30066ms)
  ✓ can getContract (269ms)
  ✓ can getLastContract (269ms)

SurveyInstance
  HavePromision
    ✓ creator has promision (269ms)
    ✓ administrator has promision (562ms)
    ✓ factorycontract has promision (287ms)
  Initialization
    ✓ can be initialized by administrator (42080ms)
    ✓ can only be initialized once by administrator (29063ms)
    ✓ create a new surveyInstance by creator (75069ms)
    ✓ can be initialized by creator (60083ms)
    ✓ can only be initialized once by creator (45098ms)
  checkQuestion
    ✓ valid question should be in the contract (269ms)
    ✓ invalid question should not be in the contract (276ms)
  checkAnswer
    ✓ valid answer should be in the contract (1124ms)
    ✓ invalid answer should not be in the contract (1085ms)
  saveAnswer
    ✓ people have promision should be able to saveAnswer (85357ms)
    ✓ same people-question-answer should not be allowed to save multiple
times (29346ms)
    ✓ same people-question different answer should not be allowed to
saveAnswer (44369ms)
    ✓ people dont't have promision should not be able to saveAnswer
(807ms)
    ✓ invalid question or answer should not be saved in the contract
  getStatistic
    ✓ can getStatistic (271ms)
    ✓ submit more answers from different users (103355ms)
    ✓ statistic should be updated (535ms)
    ✓ default count in statistic should be 0 (288ms)
    ✓ only promisioned people have access to getStatistic (366ms)
  contract address
0x4501B28941CE10D1f252E5B68b5bd9180c1e43CB

```

```
✓ last generated surveyInstance address
```

```
30 passing (10m)
```

The following figure shows the result of the one test run on my MacBook Pro, which takes about 10 mins too.

```
1. bash
> surveyapi@0.0.0 test /Users/zhuxiao/Documents/cloud/P3/surveyapi
> mocha ./test/survey.test.js

SurveyFactory
  ✓ deploys surveyFactory
  ✓ have administrator in administrators (260ms)
  ✓ can createSurvey (18153ms)
  ✓ can createSurvey multiple times (60165ms)
  ✓ can getContract (253ms)
  ✓ can getLastContract (263ms)

SurveyInstance
  HavePromision
    ✓ creator has promision (248ms)
    ✓ administrator has promision (273ms)
    ✓ factorycontract has promision (249ms)
  Initialization
    ✓ can be initialized by administrator (28215ms)
    ✓ can only be initialized once by administrator (28663ms)
    ✓ create a new surveyInstance by creator (45103ms)
    ✓ can not be initialized by people who has no promision (514ms)
    ✓ can be initialized by creator (29062ms)
    ✓ can only be initialized once by creator (30083ms)
  checkQuestion
    ✓ valid question should be in the contract (247ms)
    ✓ invalid question should not be in the contract (244ms)
  checkAnswer
    ✓ valid answer should be in the contract (1007ms)
    ✓ invalid answer should not be in the contract (1026ms)
  saveAnswer
    ✓ people have promision should be able to saveAnswer (71412ms)
    ✓ same people-question-answer should not be allowed to save multiple times (44323ms)
    ✓ same people-question different answer should not be allowed to saveAnswer (29365ms)
    ✓ people dont't have promision should not be able to saveAnswer (804ms)
    ✓ invalid question or answer should not be saved in the contract (104371ms)
  getStatistic
    ✓ can getStatistic (255ms)
    ✓ submit more answers from different users (103862ms)
    ✓ statistic should be updated (506ms)
    ✓ default count in statistic should be 0 (247ms)
    ✓ only promisioned people have access to getStatistic (249ms)
  contract address
current instance address: 0xE8201A4e1BABaa187D9a44FD90e5D046BCd4b039
  ✓ last generated surveyInstance address

30 passing (10m)

zhuxiaos-MacBook-Pro:src zhuxiao$
```

Figure x. A result of one run of the contract test case

RESTful API service

The RESTful API service is constructed by utilizing a module named “EXPRESS” in NodeJS. We just need to define some router and the corresponding callback functions.

The routers are shown as below. The callback functions are defined in the controller module. Take the save operation as an example, when user post data to `surveyapi.skypigr.info/survey/save`, the `/save` router will be activated, and the corresponding callback function, which is `surveyController.save`, will be called to execute some logic operations.

```
const express = require('express');
const router = express.Router();
const surveyController = require('../controller/surveyController');
/* GET home page. */
router.get('/', surveyController.index);
router.post('/', surveyController.index);

router.get('/save', surveyController.save);
router.post('/save', surveyController.save);

/**
 * checkQuestion
 */
router.get('/checkq', surveyController.checkQuestion);
router.post('/checkq', surveyController.checkQuestion);

/**
 * checkAnswer
 */
router.get('/checka', surveyController.checkAnswer);
router.post('/checka', surveyController.checkAnswer);
...

```

As we need to deal with different condition checkings and several exceptions, the code in the callback function is pretty long.

```

exports.save = async function save(req, res, next) {
  let answer;
  let question;
  let contractAddress;
  let userAddress;
  // console.log(req.query);
  if (req.query.answer) { answer = req.query.answer; }
  else { res.status(400).json({ success: false, msg: 'answer field is
required' }); return; }

  if (req.query.question) { question = req.query.question; }
  else { res.status(400).json({ success: false, msg: 'question field is
required' }); return; }

  if (req.query.contractAddress) { contractAddress =
req.query.contractAddress; }
  else { res.status(400).json({ success: false, msg: 'contractAddress field is
required' }); return; }

  if (req.query.userAddress) { userAddress = req.query.userAddress; }
  else { res.status(400).json({ success: false, msg: 'userAddress field is
required' }); return; }

  if (!web3.utils.isAddress(contractAddress)) {
    res.status(400).json({ success: false, msg: 'contractAddress is invalid'
}); return;
  }

  console.log('incoming SAVE request, query:', req.query);

  if (!contractContainer.has(contractAddress)) {
    //create new surveyInstance
    console.log('New SurveyInstance need to be created');
    surveyInstance = await new
web3.eth.Contract(JSON.parse(instanceInterface), contractAddress);
    contractContainer.set(contractAddress, surveyInstance);
  }
}

```

```

} else {
    surveyInstance = contractContainer.get(contractAddress);
}

qhash = web3.utils.soliditySha3(question);
ahash = web3.utils.soliditySha3(answer);

//generate key and value
key = utils.makeKey(userAddress, qhash);
console.log(key);

// in case of the input hash is bytes32, we just need the first 12 bytes.
// that's a 24 bits hexString + '0x', 26 bits in total.
value = ahash.slice(0, 26); //0x9e563769c9 1d99840a2f 4956, 24bits

if (accounts.length == 0) {
    console.log('reloading accounts');
    accounts = await web3.eth.getAccounts();
}
// console.log(accounts);
let hash;
try {
    surveyInstance.methods.saveAnswer(key, value).
        send({
            from: accounts[0],
            gasPrice: gasprice,
            gas: gaslimit
        })
        .on('transactionHash', txhash => {

// return the transaction id first, because it will take 15~60 seconds
// to confirm this transaction, we don't have to wait.
            msg = 'pending'
            hash = txhash;
            transactionContainer.set(hash, msg);
            res.json({

```



```

        success: true,
        result: msg,
        msg: hash,
        query: 'save'
    })
    console.log('transactionHash:', hash);
})
.on('receipt', async function (receipt) {
    // when we get the receipt finally, we save the contract address
    // in memory for later use.
    msg = receipt['blockHash'];
    hash = receipt['transactionHash'];
    transactionContainer.set(hash, msg);
    console.log('SAVE complete for tx:', hash);
})
.on('error', (error) => {
    // this happens when the transaction is reverted. usually because
    // the operation violates the condition of the contract function or
    // the transaction itself has not been successfully created.
    console.error(error);
    if (hash) {
        msg = 'false';
        transactionContainer.set(hash, msg);
        console.error('SAVE error for tx:', hash);
    }
    else {
        //critical error, even don't get a transaction id
        res.status(400).json({
            success: false,
            result: error
        })
        console.error('SAVE encounter fatal error');
    }
})
} catch (error) {
    console.log(error);
}

```

```
    res.json({
      success: false,
      result: error,
      msg: question + ', ' + answer,
      query: 'save'
    })
  }
}
```

Unit test for api service

After we set up our RESTful api service, everyone can access this service. We also need to make sure the interface we are offering will act as we designed. So, we developed some test cases to make sure everything is ok. And, this test case is also a good show case of our api service for our customer.

By the same method as we have shown in contract unit section, we developed 13 test cases for api service test, as shown in the following block. The difference is, in this case, we don't interact with blockchain directly, we call the api service instead. We submit our data to corresponding interface and then we grasp return value from it. The api service will deal with everything that needs to interact with blockchain.

To run the api test case, we just need to run: `npm run apitest`

In this test case, we submit many answers from different user address, although each submit need 15~60 second to be confirmed, we don't need to wait for each submit. Instead, we submit all the answer, store the txid and check the txids in the end. All of the transactions will be included in the same block, in the case, we just need to wait for one confirm period, which is about 15~60 second.

```
> surveyapi@0.0.0 apitest C:\Users\xzhu\Dropbox\blockchain\eth\projects\surveyapi
> mocha ./test/api.test.js

create contract
  ✓ create contract with invalid creator address, should be reverted immediately (219ms)
  ✓ create contract with pending (312ms)
  ✓ waiting for tx confirm, should be true (30748ms)
  ✓ initialize instance with invalid address, should be reverted immediately (182ms)
  ✓ initialize instance with pending (250ms)
  ✓ waiting for tx confirm, should be true (29359ms)
```

- ✓ save data with invalid question, should be reverted immediately (217ms)
- ✓ save data with invalid answer, should be reverted immediately (294ms)
- ✓ save data with pending (219ms)
- ✓ waiting for tx confirm, should be true (29308ms)

```
{ success: true,  
  result: 'pending',  
  msg: '0xaba018781963cde2c6d5056b5e3553a10f25dcff2f346f1181e1f68148b5ce70',  
  query: 'save' }  
{ success: true,  
  result: 'pending',  
  msg: '0x5dca8f72ef48d082542d1df6b3b653dea80d8958686f6faa335e3af5b50da18f',  
  query: 'save' }  
{ success: true,  
  result: 'pending',  
  msg: '0x2d585438b770120f90c4af1f18f633ac32eec647596659c4b3761a1ecd9209da',  
  query: 'save' }  
{ success: true,  
  result: 'pending',  
  msg: '0x8c3524467041d1b9fa83dc865f7650e698286964da52a7dbbb616609390d9cc1',  
  query: 'save' }  
{ success: true,  
  result: 'pending',  
  msg: '0x84ba117fd26aed49167dc94da9fceed58085b02ec4bbf5041f5ee9b1a2af9cb9',  
  query: 'save' }  
{ success: true,  
  result: 'pending',  
  msg: '0xcf9e743188b7d5ec8a2d5b4d9b8d469b656130fa39d2f1ed84c38827ac9f9f1b',  
  query: 'save' }  
{ success: true,  
  result: 'pending',  
  msg: '0x308c3e1018341e20de0a2a0e79de1c5f84ce238cff645d889e70ef3c7a54292e',  
  query: 'save' }  
{ success: true,  
  result: 'pending',  
  msg: '0x843a55475e9dad15b9c42fdafbab1ce1b97abad6fb7adcb84abf0046a6f32bf1',  
  query: 'save' }  
{ success: true,  
  result: 'pending',  
  msg: '0xdb51dcb6fae586b9b5597bc48490be7a41d02c8a419b9ea0867236db73c5086e',  
  query: 'save' }  
{ success: true,  
  result: 'pending',  
  msg: '0x710d48ef7d7a1aa5dd6ba8fc9fe1199c64e137a79bdb584ddba9367b63402d85',  
  query: 'save' }
```

```
{ success: true,
  result: 'pending',
  msg: '0x1d4ec7bd9a9d1ac707e8320f6be27327bb59d683b8ef276e21b162c64481e732',
  query: 'save' }
{ success: true,
  result: 'pending',
  msg: '0x55d31ad1a068e4d5537684371997cc937f29b1672224d3011883b27c70700934',
  query: 'save' }
{ success: true,
  result: 'pending',
  msg: '0x0581fb7a08c09db0c58576cedac267c05c48d6cbfbc07a3427963fcf75c34b10',
  query: 'save' }
{ success: true,
  result: 'pending',
  msg: '0xf23260908b88ff22672ea20464d4504b6e6faf59e4ca5580c5d5a5f07dfb39c2',
  query: 'save' }
```

✓ save multiple data with pending (6802ms)

total count: 14

```
0xb403c213aebec774817c9cefd629ea2ab8fc58f4e578c0e0518dd5b129b8687 5
0xb403c213aebec774817c9cefd629ea2ab8fc58f4e578c0e0518dd5b129b8687 6
0xb403c213aebec774817c9cefd629ea2ab8fc58f4e578c0e0518dd5b129b8687 7
0xb403c213aebec774817c9cefd629ea2ab8fc58f4e578c0e0518dd5b129b8687 8
0xb403c213aebec774817c9cefd629ea2ab8fc58f4e578c0e0518dd5b129b8687 9
0xb403c213aebec774817c9cefd629ea2ab8fc58f4e578c0e0518dd5b129b8687 10
0xb403c213aebec774817c9cefd629ea2ab8fc58f4e578c0e0518dd5b129b8687 11
0xb403c213aebec774817c9cefd629ea2ab8fc58f4e578c0e0518dd5b129b8687 12
0xb403c213aebec774817c9cefd629ea2ab8fc58f4e578c0e0518dd5b129b8687 13
0xb403c213aebec774817c9cefd629ea2ab8fc58f4e578c0e0518dd5b129b8687 14
0xb403c213aebec774817c9cefd629ea2ab8fc58f4e578c0e0518dd5b129b8687 15
0xb403c213aebec774817c9cefd629ea2ab8fc58f4e578c0e0518dd5b129b8687 16
0xb403c213aebec774817c9cefd629ea2ab8fc58f4e578c0e0518dd5b129b8687 17
0xb403c213aebec774817c9cefd629ea2ab8fc58f4e578c0e0518dd5b129b8687 18
```

✓ waiting for all tx confirm, should be true (33272ms)

```
{ success: true,
  result: '14',
  msg: '0xfbb236712beb41e86af5bbcf,0x9d886c24dbbc130ac49e3e3f',
  query: 'getStatic' }
```

✓ get stat of answer[2], should be the same with the total count in prev step (207ms)

13 passing (2m)

Web Interface

As part of our prototype, we build a simple web interface for survey owners to build and manage surveys from the “survey templates” provided by our backend application via RESTful API service. According to our original design of the contract templating services, once a contract owner successfully created a new contract on our platform, they could integrate their own systems with our platform via RESTful API service to further interact with their contracts, i.e. perform transactions on their contracts. For example, in the case of survey contracts, the survey owner could host their own survey participating interface for the end users to submit responses to the survey, but with the submitted survey results being redirected to the blockchain by sending submission requests to our RESTful API service. However, for demonstration purpose, we integrate a survey participating service in addition to the survey management interface in the same web interface as part of our prototype.

As our research focus was put on the interaction with the blockchain, we build our web interface by modifying the open-source NodeJS project, “Sample NodeJS backend for SurveyJS library and Editor” [22], instead of building it from scratch. This open-source survey portal project was built on top of a JavaScript Survey Engine called SurveyJS [23], which is a library for developers to add modern and feature-rich survey management system into their own web applications or popular content management systems. In particular, the sample survey portal makes use of the visual survey builder and the service for survey and survey results storage and analysis provided by the SurveyJS Library [24]. The codebase for the example survey portal was hosted on the public Git repository on GitHub [25], we started our coding by forking the codebase to our private Git repository hosted on GitLab [26]. Note that we have both frontend and backend implemented for this NodeJS-based web portal. The backend for the web portal application is different from the “backend application” for our platform that is described in previous sections.

Key Modifications

We can classify all the code modifications we performed in the forked codebase into two main categories: web interface modifications on the frontend of the sample NodeJS application, and data store modifications on the backend of the sample NodeJS application.

Frontend Modifications

We simplified the user interface of the sample survey portal application to provide only the following features by modifying the frontend html/javascript files located in the survey-portal/public source code directory:

- list all surveys created by a survey owner
- create a new empty survey given a valid survey owner address that satisfies the address format required by Ethereum
- add questions/answers to a newly created survey
- run an existing survey by answering all the survey questions by survey participants
- view survey results by survey owner

Backend Modifications

The most important changes to the original source code of the sample NodeJS application for our prototype happens in the backend of this application. The sample application allows developers to customize the data store by implementing their own database adaptor. Key functions implemented in the default in-memory database adaptor include the following:

1. addSurvey(name, callback): to create a new survey instance
2. storeSurvey(id, json, callback): to initialize a survey by storing its questions/answers defined in the survey editor
3. postResults(postId, json, callback): to store survey results when a survey participant completes a survey
4. getResults(postId, callback): to query the survey results collected from survey participants so far
5. changeName(id, name, callback): disabled in our prototype
6. deleteSurvey(surveyId, callback): disabled in our prototype, because anything stored on blockchain is not deletable

We implemented functions 1 to 4 in our own blockchain-based database adaptor (code in blockchaindbadaptor.js file) by storing non-sensitive data as in memory JSON objects (similar as the in-memory database adaptor), but storing sensitive data (user's survey results) on blockchain only. In the original JSON object used by the default in-memory database adaptor to represent the datastore of the survey application, we defined both surveys and results fields in the demo-surveys.js file.

```
var surveys = {  
};  
var results = {
```

```
};  
module.exports = {  
  surveys: surveys,  
  results: results  
};
```

However, in our blockchain-demo-surveys.js file, we define a JSON object to store only surveys but not results to contain the plain text questions and answers defined for surveys, because survey results are stored directly on blockchain. We also store hashes of questions and answers for each survey on blockchain, as discussed in previous sections on our backend system.

```
var surveys = {  
};  
var users = {  
};  
module.exports = {  
  surveys: surveys,  
  users: users  
};
```

Note that we also store a “users” field in addition to surveys field in our blockchain-based in-memory JSON object. The reason is that we automatically generate a new valid user address for each survey participation that runs via our web interface for demonstration purpose. In this users field, we store the last index used by our user address generation function to ensure the uniqueness of the user addresses used for each survey participation.

In the remaining part of this section, we highlight some of the key code changes we made to the survey portal.

On the survey listing page, i.e. the home page of the web portal, we have added text inputs to allow the survey owner to enter an address that satisfies the requirement of a valid address format used in blockchain and to enter a name for the survey. We have also added a spinner to this page to show a waiting status while we executing the initializing transaction on blockchain at the backend.

```
<section>  
  <p>Survey Owner Address (must be a valid Ethereum address  
format): <input type="text" id="creator-address" size="60"></p>
```

```

    <p>New Survey Name: <input type="text" id="contract-name"
size="60" value="NewSurvey"></p>
    <button data-bind="click: function() {
createSurvey(document.getElementById('contract-name').value,
document.getElementById('creator-address').value, 'spinner',
loadSurveys); }">Add</button>
    <i id='spinner' class="fa fa-spinner fa-spin fa-2x"
style='display: none'></i>
</section>

```

We concatenate the input survey name and a unique identifier of the survey, a contract address generated by the backend system of our platform, to form the non-editable survey name, implemented in the addSurvey function in the blockchainbadapter.js file.

```

function addSurvey(name, creatorAddress, callback) {
    console.log('Creating new survey with creator address: ' +
creatorAddress);
    let rtn = postto('create', { 'creator': creatorAddress })
    .then(async (res) => {
        console.log(res);
        let done = false;
        while (!done) {
            await postto('gettx', { txid: res.msg })
            .then((txres) => {
                if (txres.result != 'pending') {
                    address = txres.result;
                    console.log('');
                    console.log(txres);
                    console.log('Newly created contract address: ' +
address);

                    var table = getTable("surveys");
                    var newObj = {
                        name: name + '-' + address,
                        json: "{}"
                    };
                    table.push(newObj);
                    callback(newObj);
                }
            });
        }
    });
}

```



```

        done = true;
    } else {
        process.stdout.write('.');
    }
})
.catch((error) => {
    console.log('Failed to confirm the survey creation: ' +
error);
    callback({});
});
    await sleep(delaySec);
}
})
.catch((error) => {
    console.log('Failed to start the survey creation: ' + error);
callback({});
});
}

```

The above addSurvey backend function receives the creator address that is entered by the survey owner on home page to identify himself/herself and sends a contract creation request to the backend system of our platform. On a decentralized data store system like the blockchain network, a transaction needs some time to execute, because all nodes on the network needs to agree on the new state change to the blockchain. As a result, we adopt an async mechanism to interact with the backend system of our platform who directly interacts with the blockchain network. We first send a “create” request to the RESTful api service to start the transaction, the RESTful api service returns a transaction id in the response. We then keep checking the status of this transaction by sending another “gettx” request to the RESTful api service with the given transaction id as a parameter. When we finally get the transaction execution result from the blockchain, we consider the survey has been created as a new and empty contract on the blockchain, after which we store the concatenated survey name in our in-memory data store and call the callback function to inform the frontend about the completion of the transaction. The spinner that we added on the survey listing page will then become invisible and the new survey will be visible on that page.

After a survey is created, the survey owner needs to edit the survey with questions and answer options for each question. This can be done on the survey editing page. On this page, we have disabled the “change name” feature by deleting the following code from editor.html. The

reason for disabling this feature is that we append the address of a survey contract on blockchain to the survey name entered by contract owners when they first create their surveys.

```
<span class="edit-survey-name" onclick="startEdit()" title="Change Name">
  
</span>
```

On this same survey editing page, we have disabled the auto save feature when survey owners edit their surveys on this page by modifying the editor.js file, because the survey content (questions/answers) as a contract can be submitted to blockchain only once. Of course, in a real application, we can allow a survey editor to store all the intermediate changes to the survey in the node application itself, and only submit the final version of a survey to blockchain when survey owners confirm that the survey is finalized. However, we do not have time to develop such good-to-have features in our prototype product due to the limited timeframe for this project. Therefore, we simply make survey owners to save their changes to the survey only once.

```
editor.isAutoSave = true;
```

When a survey owner finishes editing the survey and click the “Save Survey” button, the storeSurvey function defined in our blockchain database adaptor (blockchaindbadaptor.js file) will be called.

```
function storeSurvey(id, json, callback) {
  let address = id.substr(id.lastIndexOf("-") + 1);
  let data = JSON.parse(json)['pages'];
  console.log("Initializing survey with contract address: " + address);
  let questions = new Array();
  let answers = new Array();
  let questionHashes = new Array();
  let answerHashes = new Array();
  for (let i = 0; i < data.length; i++) {
    var elements = data[i]['elements'];
    for (let j = 0; j < elements.length; j++) {
      var element = elements[j];
```

```

    if (element['type'] == 'radiogroup') {
      var question = element['name'];
      questions.push(question);
      questionHashes.push('0x' + sha3(question).slice(0, 24));
      var choices = element['choices'];
      for (let k = 0; k < choices.length; k++) {
        var answer = choices[k];
        if (typeof choices[k] === 'object') {
          answer = choices[k]['value'];
        }
        answer = question + '-' + answer;
        answers.push(answer);
        answerHashes.push('0x' + sha3(answer).slice(0, 24));
      }
    }
  }
}

console.log('Storing questions [' + questions + '] as: [' +
questionHashes + '] on blockchain.');
```

```

  console.log('Storing answers: [' + answers + '] as: [' +
answerHashes + '] on blockchain.');
```

```

data = {
  contractAddress: address,
  questionlist: JSON.stringify(questionHashes),
  answerlist: JSON.stringify(answerHashes)
}

var table = getTable("surveys");
var result = table.filter(function (item) {
  return item.name === id;
})[0];
postto('init', data)
  .then(async (res) => {
    console.log(res);
    let done = false;
    while (!done) {
      await postto('gettx', { txid: res.msg })
        .then((txres) => {
          if (txres.result != 'pending') {
```

```

        console.log('');
        console.log(txres);
        if (txres.result.length > 40) {
            console.log('Stored survey with contract address: '
+ address);
            if (!!result) {
                result.json = json;
            } else {
                result = {
                    name: id,
                    json: json
                };
                table.push(result);
            }
            callback && callback(result);
        } else {
            console.log('Survey with contract address [' +
address + '] has already been initialized, ignore current request. ');
            callback && callback(result);
        }
        done = true;
    } else {
        process.stdout.write('.');
    }
})
.catch((error) => {
    console.log('Failed to confirm survey initialization: '
+ error);
    callback && callback(result);
});
    await sleep(delaySec);
}
})
.catch((error) => {
    console.log('Failed to start survey initialization: ' +
error);
    callback && callback(result);
});

```

```
}
```

In the `storeSurvey` function, we first create hash values for all questions and answers for the backend system of our platform to store on the blockchain, and at the same time store the plain text questions and answers in our in-memory datastore. This way, the backend system can verify the validity of any survey results submitted later on.

After the survey content is finalized in the previous step, end users start to participate in the newly created survey. For demonstration purpose, we kept the “run survey” feature provided by the sample web portal so that we can submit survey results through the same web interface, triggered by the “run survey” button on the survey listing page. When this button is clicked, users will be redirected to the “run survey” page defined in `public/survey.html` file. We enabled the “show data saving status” feature in the frontend to show users up-to-date transaction execution result on blockchain, because it may take a while to complete the data store process involving blockchain. This change is done in the `public/survey.js` file.

```
var model = new Survey.Model({ surveyId: surveyId, surveyPostId:
surveyId, surveyShowDataSaving: true, css: css });
```

And in the backend of the web portal, we submit the survey results to blockchain by implementing the `postResults` function in the `blockchaindbadapter.js` file. Note that, we automatically generate a user address here through the `nextUserAddress` to simulate the uniqueness of survey results.

```
function postResults(postId, json, callback) {
  let contractAddress = postId.substr(postId.lastIndexOf("-") + 1);
  let userAddress = nextUserAddress(postId);
  console.log('Posting results for survey with contract address: ['
+ contractAddress + '] for user with address [' + userAddress + ']');
  let data = JSON.parse(json);
  let questions = Object.keys(data);
  postDoneCount = questions.length;
  postAllSuccessful = true;
  for (let i = 0; i < questions.length; i++) {
    let question = questions[i];
    postResult(contractAddress, userAddress, postId, json,
question, question + '-' + data[question], callback);
  }
}
```

```

function postResult(contractAddress, userAddress, postId, json,
question, answer, callback) {
    console.log('Posting question [' + question + '], answer [' +
answer + '] to survey with contract address: [' + contractAddress +
'] from user with address [' + userAddress + ']');
    let qa = {
        contractAddress: contractAddress,
        userAddress: userAddress,
        question: question,
        answer: answer
    }
    postto('save', qa)
        .then(async (res) => {
            console.log(res);
            let done = false;
            while (!done) {
                await postto('gettx', { txid: res.msg })
                    .then((txres) => {
                        if (txres.result !== 'pending') {
                            address = txres.result;
                            console.log('');
                            console.log(txres);
                            let isSuccessful = false;
                            if (txres.result.length > 40) {
                                console.log('Posted question [' + question + '],
answer [' + answer + '] to survey with contract address: [' +
contractAddress + '] from user with address [' + userAddress + ']');
                                isSuccessful = true;
                            } else {
                                console.log('Failed to post question [' + question + '],
answer [' + answer + '] to survey with contract address: [' +
contractAddress + '] from user with address [' + userAddress + ']');
                            }
                            donePostResult(isSuccessful, contractAddress,
userAddress, postId, json, callback);
                            done = true;
                        } else {

```

```

        process.stdout.write('.');
    }
})
.catch((error) => {
    console.log('Failed to confirm posting question [' +
question + '], answer [' + answer + '] to survey with contract
address: ' + contractAddress + ': ' + error);
    donePostResult(false, contractAddress, userAddress,
postId, json, callback);
});
    await sleep(delaySec);
}
})
.catch((error) => {
    console.log('Failed to start posting question [' + question +
'], answer [' + answer + '] to survey with contract address: ' +
contractAddress + ': ' + error);
    donePostResult(false, contractAddress, userAddress, postId,
json, callback);
});
}

function donePostResult(isSuccessful, contractAddress, userAddress,
postId, json, callback) {
    postDoneCount -= 1;
    if (!isSuccessful) {
        postAllSuccessful = false;
    }
    if (postDoneCount == 0) {
        if (postAllSuccessful) {
            console.log('Results for user with address [' + userAddress +
'] are posted to survey with contract address: ' + contractAddress);
            var newObj = {
                postId: postId,
                json: json
            };
            callback(newObj);
        } else {

```

```

    console.log('Failed to post some of the results for user with
address [' + userAddress + '] to survey with contract address: ' +
contractAddress);
    callback({ postId: postId, json: {} });
  }
}
}

```

In the `postResults` function, we submit the answer for each question as individual requests to the RESTful api service provided by the backend system of our platform. The survey result was sent as `<question hash, answer hash>` pairs. We design a mechanism to wait for the completion of all post result transactions before calling the callback function to inform the frontend of the web portal. In this step, we do not store any result in our in-memory JSON object because we consider the survey results as sensitive information, and should be store on blockchain only.

Finally, we modify the “view survey results” feature of the sample web portal to show only survey result statistics on the results page, which are implemented in `public/results.html` and `public/results.js` in the frontend. In the original sample survey portal, the survey results displayed are grouped by users, meaning that user 1’s answers for each question are displayed before the next user’s answers for each question. [Figure 17](#) is an example of the original survey results page for a survey of 2 questions. There are 2 users who submitted results for this survey, and one of them chooses `item1` for both questions while the other chooses `item2` for both questions.

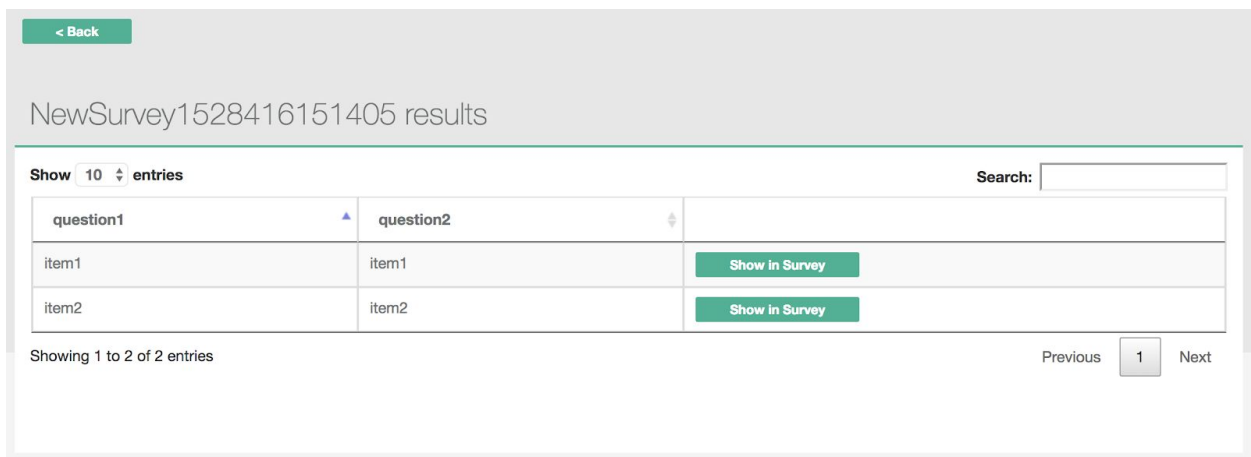


Figure 17: The survey results page of the original sample SurveyJS application showing results for a survey from 2 users.

However, in our blockchain based implementation, we do not want to differentiate answers from different users, so what we display on the results page are only the total counts of answers chosen by users for each question. [Figure 18](#) shows the same results as shown in [Figure 17](#), however, we cannot see here that one user selects item1 for both questions, and the other choose item2 for both questions anymore.

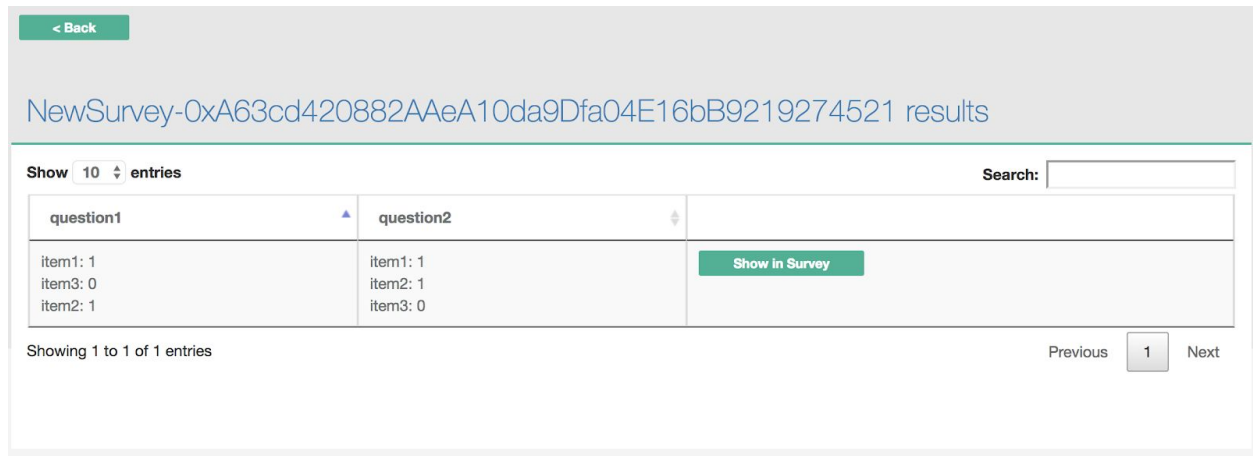


Figure 17: The survey results page of our survey web interface showing results for a survey from 2 users with the same answers selected as in [Figure 17](#).

As the format of results on the display page are the same, what we modified here are just the data returned to the frontend, which was implemented in the `getResults` function in our blockchain database adaptor (the `blockchaindbadapter.js` file).

```
function getResults(postId, callback) {
  let address = postId.substr(postId.lastIndexOf("-") + 1);
  console.log("Getting survey results with contract address: " +
address);
  getSurveys((result) => {
    let data = JSON.parse(result[postId].json)['pages'];
    let qas = new Array();
    answerCounts = {};
    for (let i = 0; !!data && i < data.length; i++) {
      let elements = data[i]['elements'];
      for (let j = 0; j < elements.length; j++) {
        let element = elements[j];
        if (element['type'] == 'radiogroup') {
          let question = element['name'];
          let questionText = element['name'];

```

```

        if (element['title']) {
            questionText = element['title'];
        }
        answerCounts[question] = {};
        let choices = element['choices'];
        for (let k = 0; k < choices.length; k++) {
            let answer = choices[k];
            let answerText = choices[k];
            if (typeof choices[k] === 'object') {
                answer = choices[k]['value'];
                answerText = choices[k]['text'];
            }
            qas.push([question, answer, questionText, answerText]);
        }
    }
}
}
let failed = false;
getDoneCount = qas.length;
getAllSuccessful = true;
for (let i = 0; failed !== true && i < qas.length; i++) {
    let question = qas[i][0];
    let answer = qas[i][1];
let questionText = qas[i][2];
let answerText = qas[i][3];
    data = {
        contractAddress: address,
        question: question,
        answer: question + '-' + answer
    }
    postto('stat', data)
        .then((res) => {
            console.log(res);
            if (res.result !== 'pending' && res.success) {
                answerCounts[question][answerText] = res.result;
                doneGetResult(true, postId, callback);
            }
        })
})

```


In the `getResults` function, we follow the same asynchronized mechanism to wait for all “query statistics” transaction to complete before send back response to the frontend and display the total counts of each answer for each question in the result display page. Note that we only store question and answer hashes on blockchain, so the statistics response we received from the RESTful API service is hash based. In the `getResults` function, we query our own in-memory JSON object to get the plain texts for questions and answers, and translate the hash-based statistics result to a human-readable one before returning it back to the frontend to be displayed on the web interface.

Design document and flowchart

We develop our project in JavaScript, based on NodeJS. We use web3.js lib, which is provided by Ethereum Dev. Group, to interact with blockchain infrastructure. INFURA offers an scalable blockchain infrastructure, which is very convenient for developer to use. Without this service, developer must maintain a full node of ethereum mainnet or rinkeby net to get a data source for web3 library.

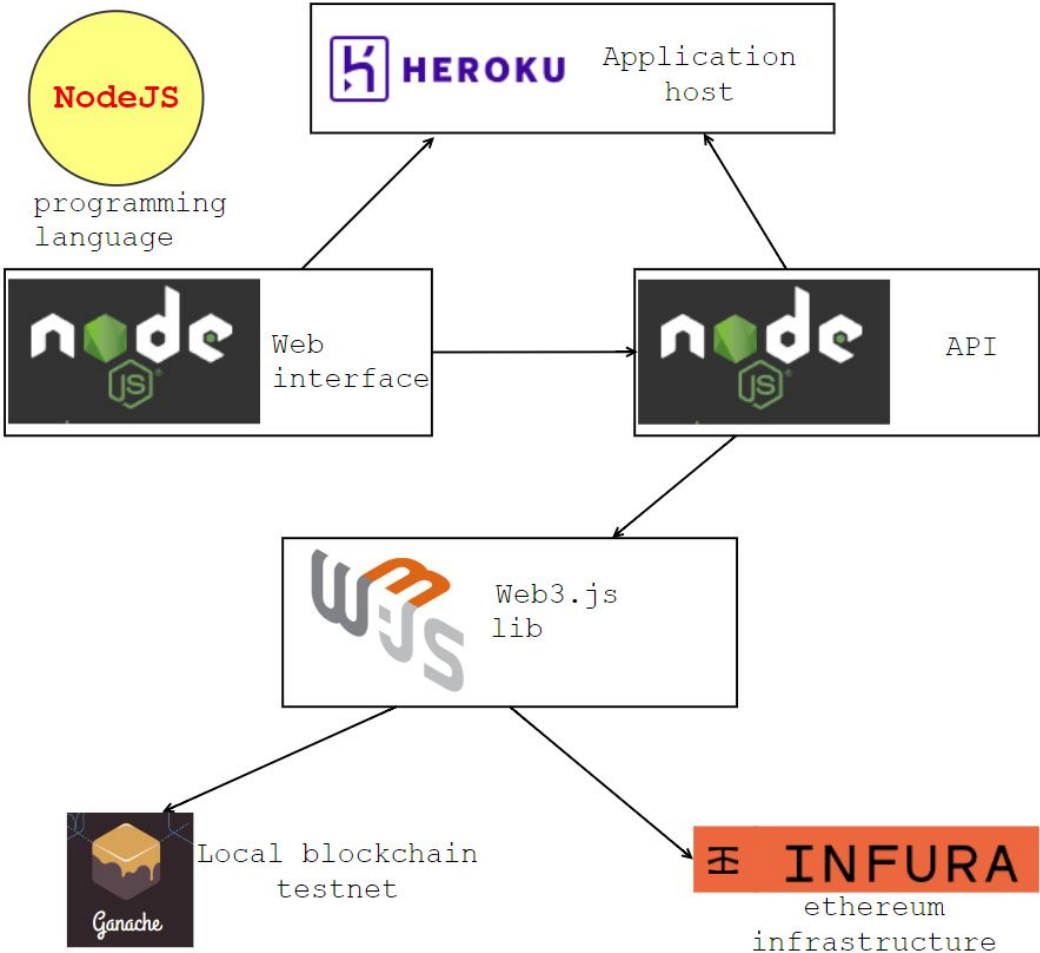


Figure 18: implement structure of this project

We also use Ganache to establish a testnet for ethereum blockchain, on which blocks are mined automatically and transactions are confirmed automatically, which could save huge developing time.

Data structure

As we have described in chapter 5, we need to store two kinds of data into blockchain, the initial questions and answers and the user-question-answer pair. To implement this, we design our data structure as so:

1. Generate 24bits (12 bytes) hash string from question source and answer source.
2. Saving these 24 bits hash into the blockchain
3. Generate key by combining the user address and 24bits question hash, the value is the 24 bits answer hash
4. Save key-value pair into blockchain

The following figure shows the procedure to generate the 12 bytes hash string from question source and answer source.

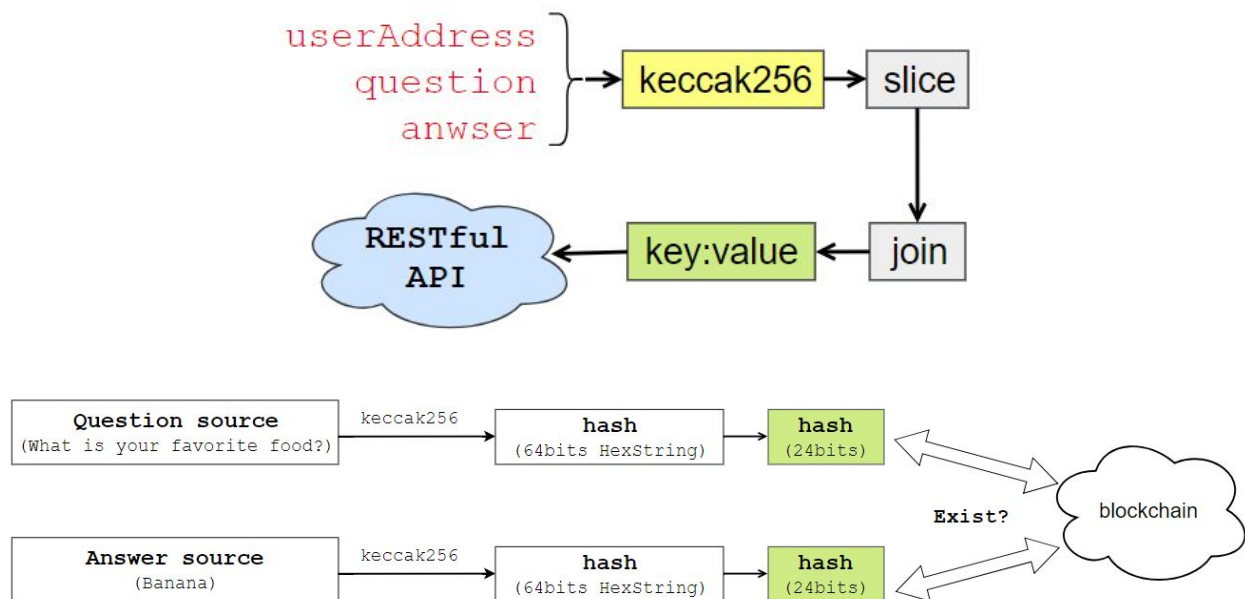


Figure 18: Question and answer generate procedure

As we want to identify different user's answers to each question, thus we combine the user's address and the question hash as the key. As a result, any user who has done the survey could checkout his/her answer to any question, it is recorded uniquely in the blockchain.

Because each answer has a unique key in the storage and also has a unique transaction id in the blockchain, and we build our statistic data upon these unique, traceable and invertible records, thus, the result of the survey has high transparency and high security.

The following figure shows how to generate the key and value from user's address, question hash and answer hash.

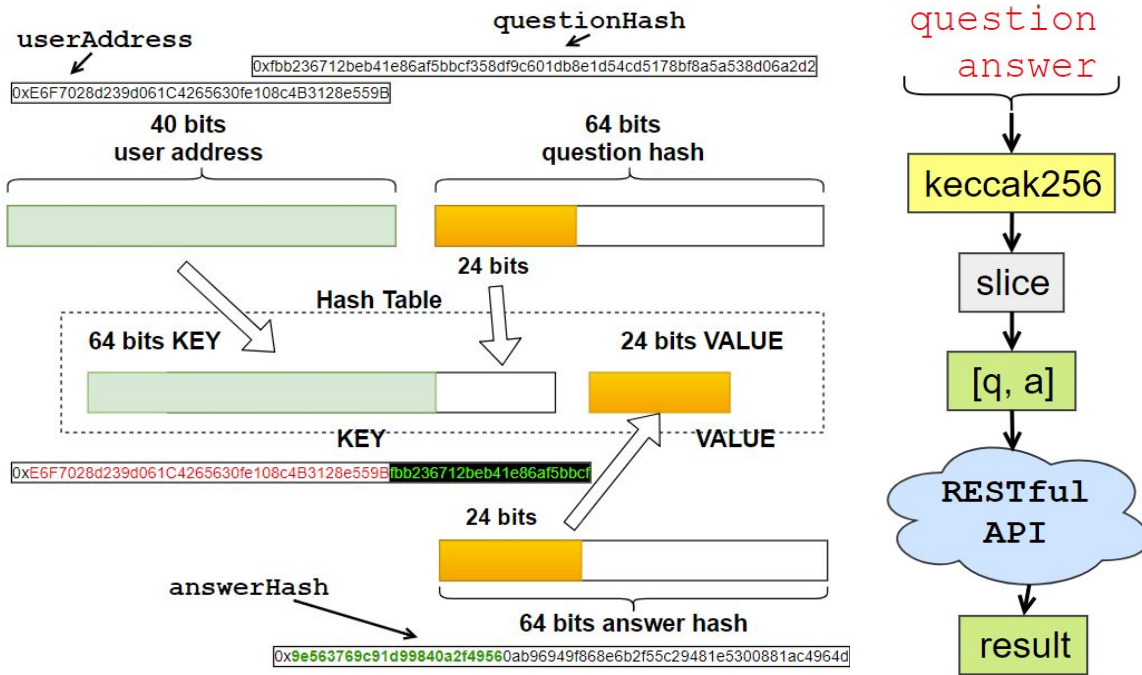


Figure 19: Key-value generate procedure and result fetching procedure

Contract structure

The contract structure are shown as the following figure. We use a contract factory to generate other survey contract instance. Each survey contract is independent after they have been created. Once we get the contract address, we can init, check questions or answers, save answers or get statistic data through our RESTful api service.

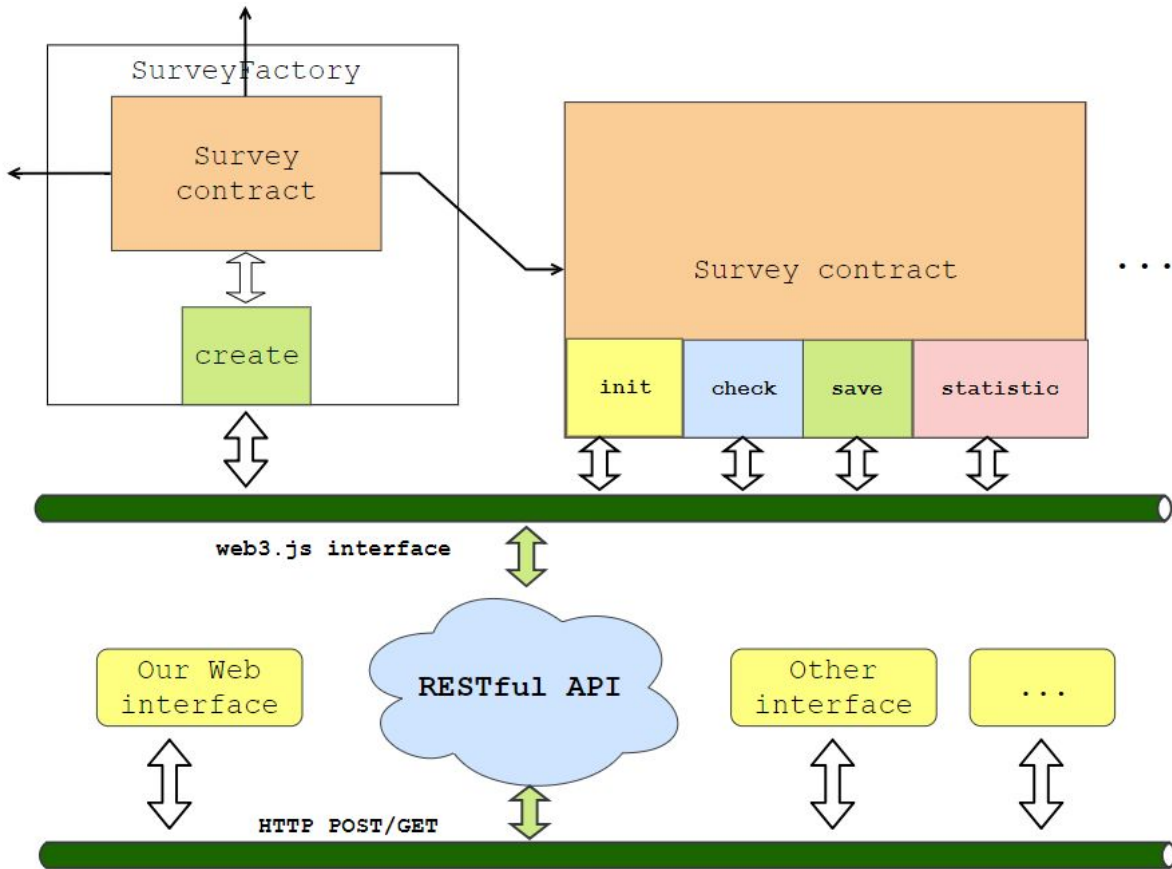


Figure 20: The abstract structure of cloud service

RESTful API service

In the backend, to interact with the contract, our program must deal with the HTTP GET/POST, requests from users, get the parameters, check the input parameters and then interact with contracts through web3.js, and send corresponding result to user through the HTTP GET/POST method again, as shown in the figure above.

The RESTful API service(backend) are totally separated from web interface(frontend). In our customer's view, they can focus on the design of own web interface without taking care of technique details about how to interact with blockchain, they program as they usually do.

As the service provider, we design the middle layer software carefully, with consideration of the contract security, transaction cost efficiency, manageability, exception handling and the blockchain related issues, establish a uniform application interface for each category of our services.

Web Interface

The flowchart of a typical use case of our web interface is shown in Figure 21 below.

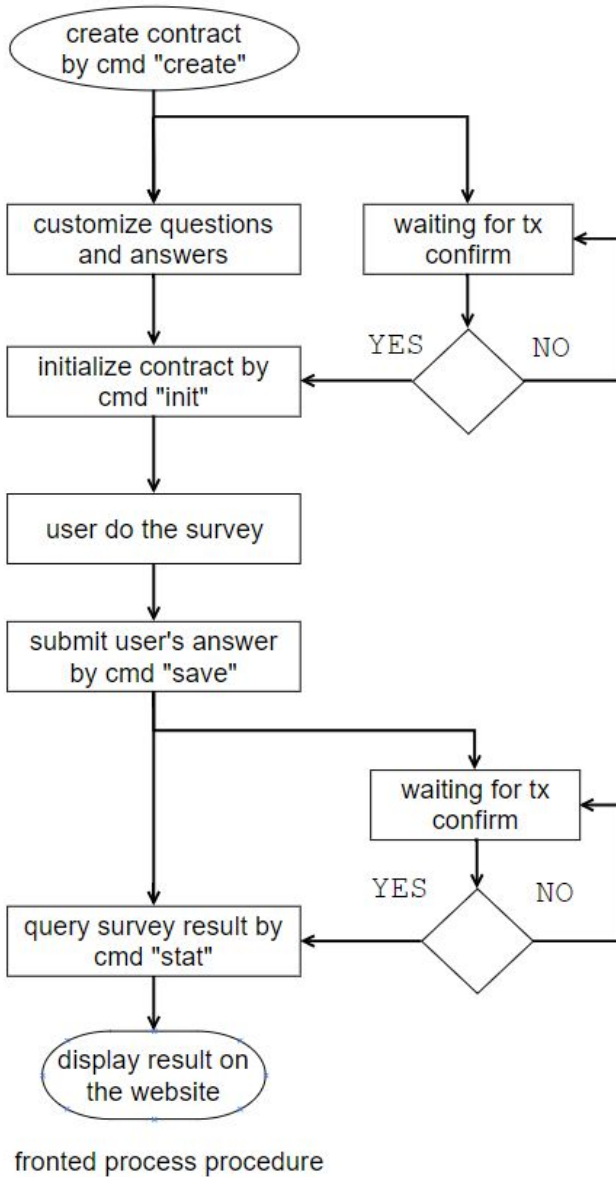


Figure 21: The process procedure of a typical web interface.

As introduced in the [Web Interface](#) section under the previous Code section, we implement our web interface by customizing an existing open-source NodeJS survey management application. The architecture of our web interface application is shown in Figure 22 below.

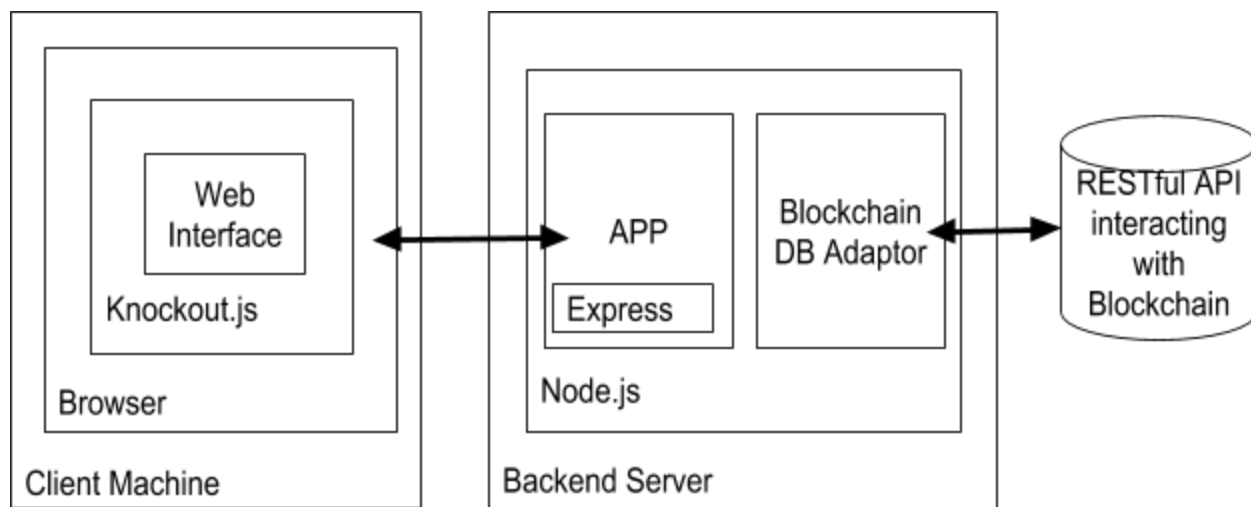


Figure 22: Architecture diagram of the survey web portal.

From this diagram, we can see that we have both backend and frontend for the web interface alone, both implemented in NodeJS. Through implementing a blockchain-based DB adaptor, we successfully integrated the SurveyJS (the JavaScript Survey Engine) backend with the backend application of our own platform.

7. Data analysis and discussion

As our goal for creating the prototype is to test the feasibility of creating blockchain-based surveys from the pre-built survey type of contract templates. Our approach to validate whether we have met our goal is by carefully validating the correctness of all the transactions generated by our system and sent to the blockchain for a typical use case, i.e. when a survey owner creates a new survey through our web interface and when more than one end users submit their answers to the survey questions through our web interface. In this section, we will describe how we extract all those transactions from both our frontend and backend Node applications and validate them by analyzing their connections. We also analyzed the creation time of all the transactions to get a feel of the performance of a blockchain-based survey system when working on the described validation approach. One thing we noticed from the process is that the transaction execution is very time-consuming due to its decentralized acknowledgement nature (and so we implemented asynchronized mechanisms in our system to work with these transactions), so we can safely conclude that a blockchain-based survey system will be much slower than a non-blockchain-based system without a thorough performance test. Therefore, we skip the conventional testing mechanism that is based on a large amount of automatically generated input data set due to the limited time given for this project.

Output generation

Following our validation approach described at the beginning of this section, we started our data validation by manually completing a typical use case and validating all the transactions generated and sent to blockchain associated with this use case. Since we have deployed our prototype to Heroku [27], all steps described below were performed on [the online system](#) hosted there.

Step 1: create a new contract

Methods: get, post

Params:

creator: the address of the creator, which will be stored in the survey contract. This address should be valid ethereum address, otherwise the request will be rejected by the contract factory.

Example using GET method

<http://surveyapi.skypigr.info/survey/create?creator=0xE6F7028d239d061C4265630fe108c4B3128e559B>

return:

```
{
  "success": true,
  "result": "pending",
  "msg":
  "0x70ea8621a490d804efc01a4fb9a014f181539f8676fc27a95a0c218a48a3c924",
  "query": "create"
}
```

This methods will return an txid in the 'msg' field as soon as the transaction is broadcasted to the network. But it needs 15~60 seconds to be confirmed, until then we can get the contract address of the new survey instance we just created, as shown in the "result" field of this method, the status is pending. We can check the status of this transaction through "gettx" method according to the txid we just received.

The following result is the return value of a 'gettx' call to api service with txid as its parameter. If the given tx is confirmed by the network, the result field should be the contract address, otherwise the result field is 'pending'.

```
{
  "success": true,
  "result": "0x455f56ad388ddf8a68571Ad4652E388Bee3666f8",
  "msg":
  "0x70ea8621a490d804efc01a4fb9a014f181539f8676fc27a95a0c218a48a3c924",
  "query": "gettx"
}
```

Remind that only when we get the address of a contract can we do further operation on the contract.

In our web interface, this can be done by just input two params: address and a name for the survey, as shown in the following picture.

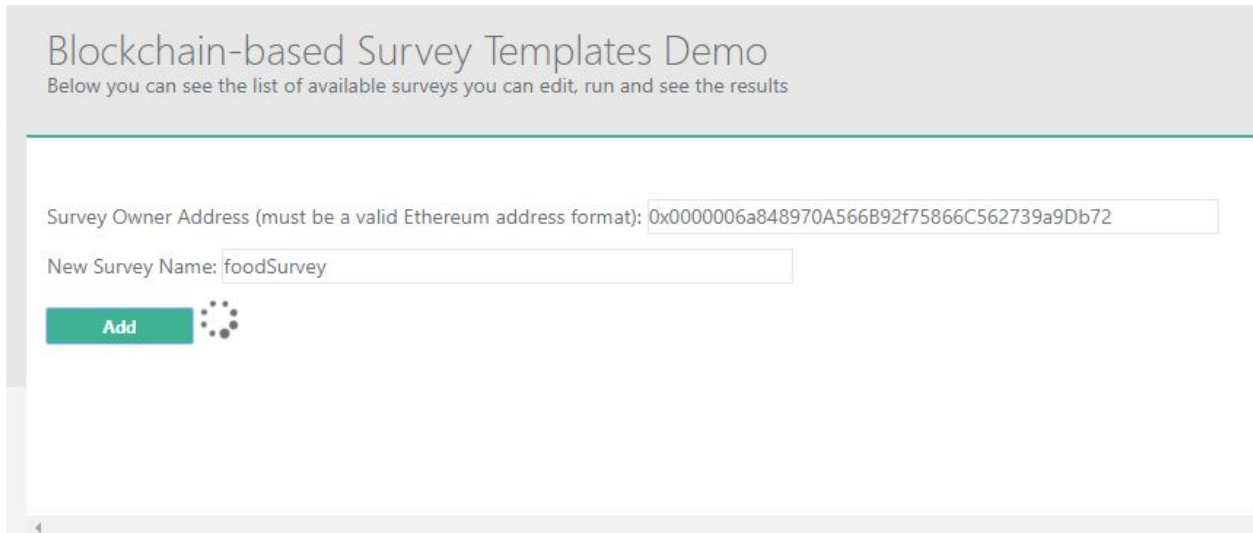


Figure 23: web interface for creating an survey.

After about 15~60 seconds, when the transaction is confirmed by the blockchain network, the page will return a contract list as shown below. As survey named:

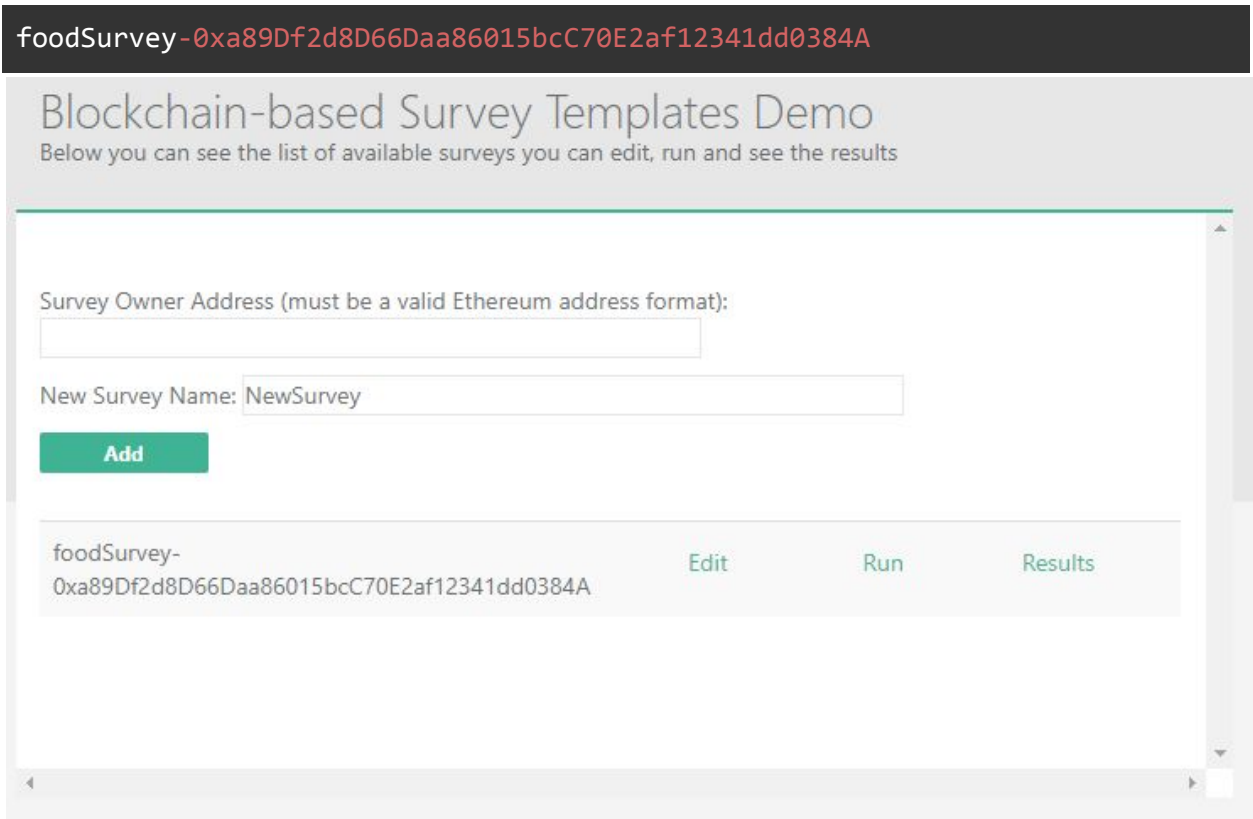


Figure 24: The contracts lists of web interface

We can customize this contract by 'Edit' this contract, complete a survey by 'Run' this contract and check the result by click 'Results'.

Step 2: initialize the contract

After a contract is created, the first thing the user should do is to customize his contract, take the survey category as an example, the use will customize his survey questions and answers to each question by click the 'Edit' link in the contract list. And we will come to customization page as shown below.

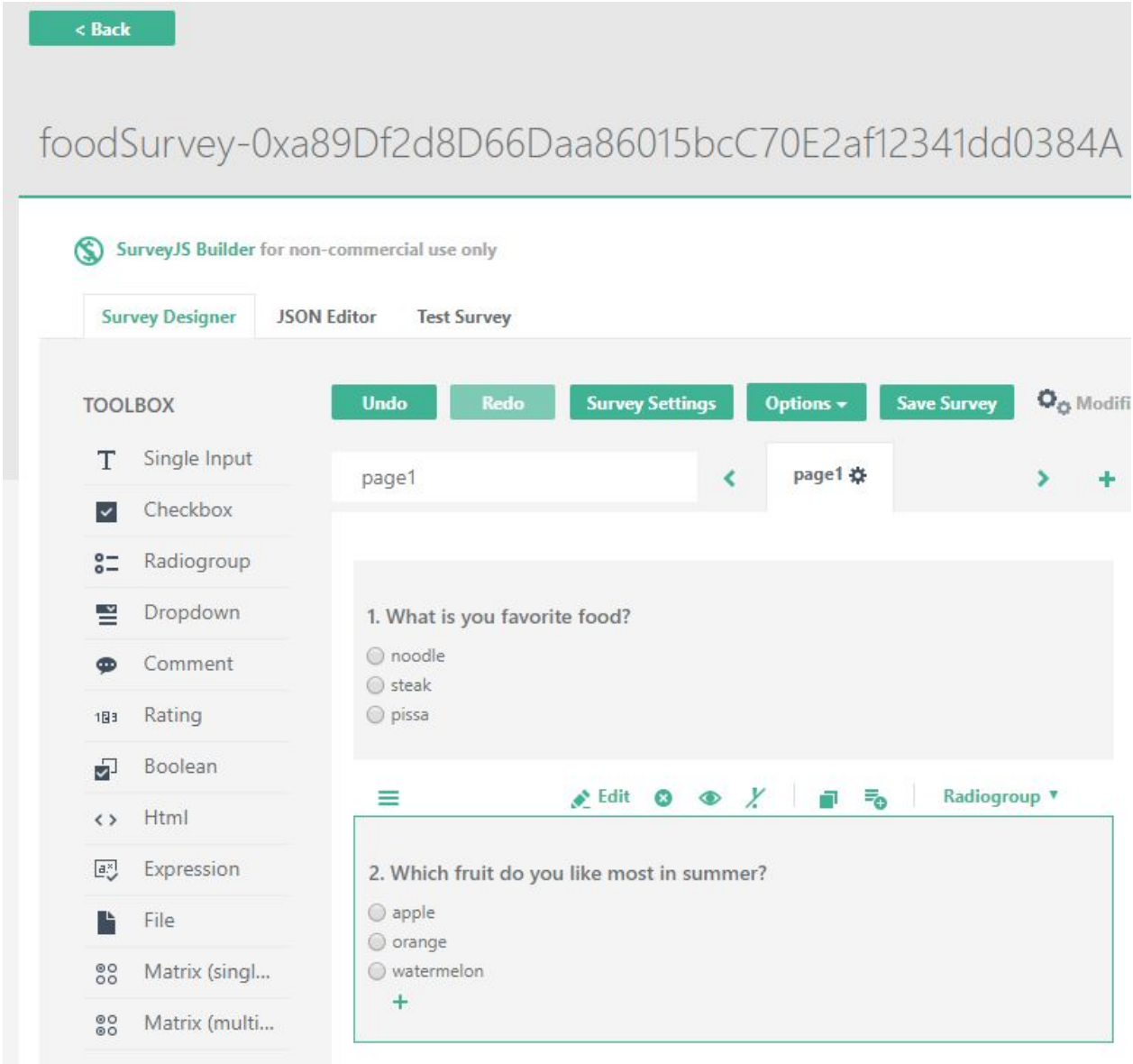


Figure 25: Customize the content of the survey.

This demo only support radiogroup element for now to demonstrate the our service.

User just needs to drag the Radiogroup to the center of this page and modify the content of the question and answers. Multiple questions and multiple answers to each questions are supported. As shown in the picture above, we created two question for the food Survey.

In the end, when user completes all the questions and answers, he just need to click the 'Save Survey' button in the upper right corner.

Then the contents of this pages, the question sources and answer sources, will be hashed and sliced and then sent to the API server by 'init' method.

methods: get,post

params:

1. contractAddress
2. questionlist
3. answerlist

An example using GET method:

[http://surveyapi.skypigr.info/survey/init?questionlist=\["0x9e563769c91d99840a2f49560ab96949f868e6b2f55c29481e5300881ac4964d","0xbe83a48bf35c1402f9efaa687c50a92fb78c5e4323b4379af5cf699171e608c6","0x9d886c24dbbc130ac49e3e3f87a9dd3b71a23548da8973674d2cb5b2678ab767","0xd6d2bdabef5123f0dc370bbc761941f57371fe50899d69248ebcc2d9aa9d7d8c"\]&answerlist=\["0x44c3bbc6a84dd2c76cf7878c313211195d144470990b68336d9180445fe607b7","0x455f56ad388ddf8a68571Ad4652E388Bee3666f8"\]&contractAddress=0x455f56ad388ddf8a68571Ad4652E388Bee3666f8](http://surveyapi.skypigr.info/survey/init?questionlist=[)

the gestionlist and answerlist is an array of question hashes.

return:

```
{
  "success": true,
  "result": "pending",
  "msg":
  "0x44c3bbc6a84dd2c76cf7878c313211195d144470990b68336d9180445fe607b7",
  "query": "init"
}
```

The same as we did in the "create" part, we also need to wait about 15~60 seconds for the transaction to be confirmed by ethereum network. We can check the status of this transaction by query 'gettx' with the txid as a parameter, which is shown in the 'msg' field of the return result above, OR, we can check the status by exploring the public blockchain explorer, take this transaction as an example, we can access the following link in a browser:

<https://rinkeby.etherscan.io/tx/0x44c3bbc6a84dd2c76cf7878c313211195d144470990b68336d9180445fe607b7>

The result is shown as below.

The screenshot shows the Etherscan interface for a transaction on the Rinkeby testnet. The transaction hash is 0x44c3bbc6a84dd2c76cf7878c313211195d144470990b68336d9180445fe607b7. The transaction is marked as 'Success' and occurred at block height 2398201, which has 26051 block confirmations. It was sent 4 days and 12 hours ago (on June 3, 2018, at 04:24:53 PM UTC). The sender's address is 0x1aca1282970840a5814f060a7921e3f90ce63a4c and the recipient's address is 0x455f56ad388ddf8a68571ad4652e388bee3666f8. The transaction value is 0 Ether (\$0.00). The gas limit is 6,000,000, and the gas used is 157,696. The gas price is 0.000000002 Ether (2 Gwei), resulting in an actual transaction cost of 0.000315392 Ether (\$0.000000). The nonce is 578. The input data field contains a long hexadecimal string, and there is a 'Convert To UTF8' button below it.

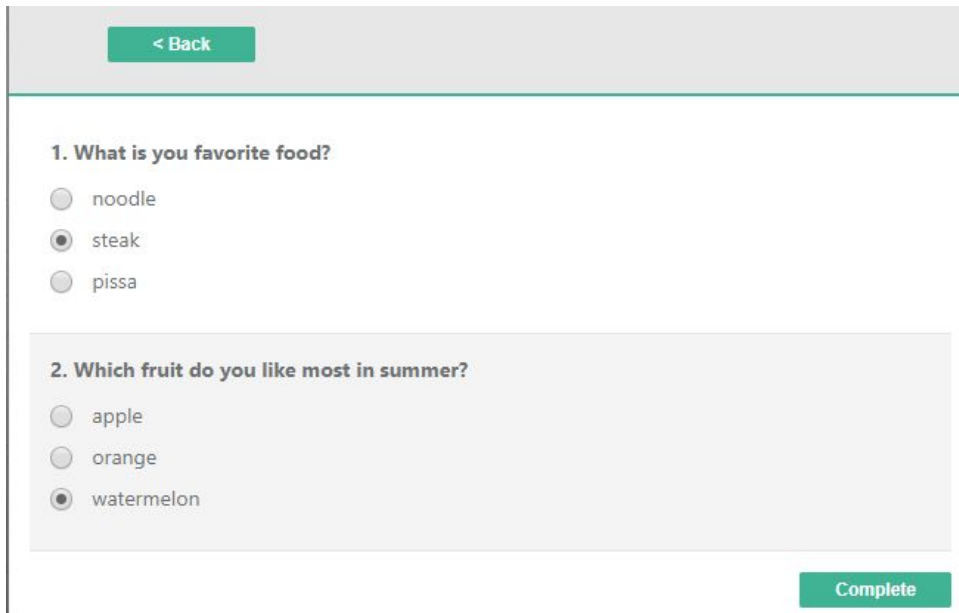
Figure 26: check the transaction in a public blockchain explorer

One thing should be reminded, a newly created survey can only be initialized once, this is a designed feature. If you try to modify or add more questions to a survey which has been saved once, the next save operation will be reverted by the API server and return a message with error.

This is because, later when a user submits an answer to the contract, we need to check whether this answer and question are stored in the contract or not. Through this way, we can make sure only the predefined questions and answers will be counted in the statistic result, thus the transparency can be guaranteed.

Step 3: Save data to contract

User can run the survey easily on this demo, the following picture shows the interface of a survey we created in step2. User can select an answer for each question, then click “complete” to submit his result.



The screenshot shows a mobile survey interface. At the top left, there is a green button with a white arrow pointing left and the text "< Back". Below this, the first question is "1. What is you favorite food?". It has three radio button options: "noodle", "steak", and "pissa". The "steak" option is selected. Below the first question is a light gray box containing the second question: "2. Which fruit do you like most in summer?". It has three radio button options: "apple", "orange", and "watermelon". The "watermelon" option is selected. At the bottom right of the survey area, there is a green button with the text "Complete".

Figure 27: run the survey in the demo

Then, each question-answer pair will be sent to the API server through “save” method.

methods: get, post

server: <http://surveyapi.skypigr.info/survey/save>

params:

1. contractAddress
2. question
3. answer
4. userAddress

example:

<http://surveyapi.skypigr.info/survey/save?contractAddress=0x455f56ad388ddf8a68571Ad4652E388Bee3666f8&question=How old are you?&answer=I'am 2&userAddress=0x49C57Da993072C72570ED251fc108126B126B6d9>

return:

```
{  
  "success": true,  
  "result": "pending",
```

```
"msg":  
"0xc702c2bc0b531b9e3707295224ca0877d2ddd588a6441028fa642727613d74f4",  
"query": "save"  
}
```

Each question-answer will be sent separately and will receive an unique txid. Also, we can check the status of each tx by querying the “gettx” method or check them in a public ethereum block explorer.

Step 4: View statistics

We can view the statistics by clicking the “Results” link. The results will be updated automatically as shown below in Figure 28.



Figure 28: view the statistics of a survey

We don't store any results in our server, we get the result by querying the API server with “stat” command.

methods: get, post

server: <http://surveyapi.skypigr.info/survey/stat>

params:

1. contractAddress
2. question
3. answer

An example using get method:

<http://surveyapi.skypigr.info/survey/stat?answer=I'am 2&contractAddress=0x455f56ad388ddf8a68571Ad4652E388Bee3666f8&question=How old are you?>

return:

```
{
  "success": true,
  "result": "1",
  "msg": "0xfbb236712beb41e86af5bbcf,0x9e563769c91d99840a2f4956",
  "query": "getStatic"
}
```

The request will be processed immediately by the API server and return a number in the 'result' field, which is the count number of a given question-answer pair selected by users. By querying each question-answer pair, we can build the statistic results totally from the data stored in the blockchain.

Output analysis

Since our focus is on the manual walk-through of a typical use case to validate the correctness of the transactions we sent to the blockchain, we do not generate a large set of output data for this project, so the analysis is simply the comparison between our manual input and the expected output which shall be discussed in the next section.

Compare output against hypothesis

As we can see from the previous section on output generation, we carefully verified each transaction generated by our platform for the manual walk-through of a typical use case. The output of the survey result statistics we get in step 4 fully matched with our answers to the survey questions submitted in step 3 (see Figure 27 and Figure 28).

Abnormal case explanation (the most important task)

We observed one abnormal case during our manual validation of the correctness of all the transactions when multiple users submit results for the same survey at the same time, i.e. we click "run survey" on homepage to start submitting results for a previously submitted survey. Note that our web portal will generate a new user identifier (as the user's contract address when submitting results to blockchain) each time when we click "run survey" to simulate survey submissions from different users. We generate the new user address by incrementing a counter variable which points to the last used address on a list of available addresses we pre-generated and stored in memory. When we start the second survey submission before the previous post results transaction is confirmed, we found that the user identifier returned by our program stays the same as the previous survey submission. After countless tests, we figured out the reason behind this abnormal behavior of our program. Because we adopt a async mechanism to

handle requests in our web portal, the callback function corresponding to the first survey results posting transaction is not called until the first transaction is executed on blockchain, which often takes a while. We send back response to the web frontend in this important callback function so that the outcome of the transaction is communicated back to the web portal user. For example, for the survey submission case, the “run survey” page will inform the user whether the survey results have been successfully stored upon the completion of the post results transaction. Although we increment the user address counter before calling our backend API service, however since the user address counter is stored in our session-aware in-memory JSON datastore, managed by the NodeJS framework, the session-aware data is not updated until the callback function is called. Therefore, the root cause of this problem is due to the slow confirmation of blockchain transactions. Of course, we could fix this issue by digging into the SurveyJS framework and implemented a different approach for generating the user address. But since we generate the user address to run surveys just for demonstration purpose, we do not want to spend time on fixing this issue. After all, the survey owners shall host their own survey running interfaces for survey participants to complete the survey. The survey owners just need to submit the survey results via the RESTful API service we provided as part of our platform.

Another related issue we observed during our manual validation of the correctness of our system is the staleness of the web frontend of our survey interface. For example, we implemented a spinner on the homepage to indicate the waiting status while creating the survey and waiting for the creation transaction to complete. Sometimes when it takes too long for the creation transaction to finish, we observed that the spinner will be spinning on the web page forever, even if the transaction has already completed at the backend. The root cause for this issue is the same as the one described in previous paragraph, which is the slow execution of blockchain transactions. There is a timeout set in the SurveyJS framework, apparently the callback function failed to be invoked when the backend take too much time to complete a request. In this case, the response is not propagated to the web frontend in time, the web frontend will not refresh the content of the web page accordingly. That is how the stale content of a web page formed. We observed the same issue on the survey submission page. When a user completes the survey, we will show the results saving status on the web page in a timely manner, if the post results transaction does not take too long to complete, the user will see a message informing that data saved successfully on the submission page. However, if the post results transaction takes too long to complete, the user will see a “failed to store the survey results” error message on the submission page, because the SurveyJS framework considered a timeout response as failure. However, in the backend, the survey results are successfully stored in blockchain, just after the timeout event happened.

8. Conclusions and recommendations

Summary and conclusions

As we discussed in the chapter 4, the hypothesis of this project is trying to provide a platform, through which anyone, programmer and non-programmer, can deploy their blockchain-based apps.

After the discussion of the above sections, it is summarized as follows:

First, after investigating the characteristics and application of the blockchain, we discussed the security of smart contract, studied the common vulnerabilities of smart contracts and how to deal with them.

Then, given the high threshold of blockchain technology and unpatchable feature, we propose to build a secure and easy-to-use blockchain cloud service through which more users can easily combine their existing business with blockchain technology.

We discussed the project design ideas, structure, implementation methods in chapter 4 and chapter 5, and details the implementation details in the chapter 6.

The main results of this project is:

1. We designed an easy-to-deploy smart contract template
2. We designed an easy-to-use user interface to utilizing our cloud service
3. A middle layer software between blockchain infrastructure and public user was constructed, tested and verified by this project.
4. An open RESTful API service are also provided for user to interact with our blockchain-based service.
5. An easy-to-use web interface for user to interact with the contracts was provided.

Through those topics we discussed, the tests and the analysis, we have the following conclusions:

1. The feasibility of providing cloud services based on blockchain was verified.
2. Blockchain is good solution for transparency sensitive applications.
3. The scalability and capacity of blockchain networks are the bottlenecks that restrict their further penetration.

Recommendations for future studies

Although, it is the best smart contract blockchain infrastructure by the time of writing, the Ethereum network have bottlenecks that restrict its further penetration. The 15s average block time is still too long for commercial use, although it is must faster than bitcoin's 10 mins.

As we have talked about in the previous chapters, we need to wait 15~60 secs to make sure our data DO write flush into the blockchain. Although we can take some technical measures to reduce the waiting time of users, this will, to a certain extent, reduce the transparency and security of our services.

Therefore, based on this consideration, for follow-up studies, we recommend the following directions :

1. Give priority to applications that require less real-time performance.
2. Use other technical means to hide user's waiting time.
3. Try other emerging blockchain platform for a better performance.

Although we design our system to store data totally on blockchain, we do should try and find a way to combine the existing mature technology with blockchain application, by which we can solve the existing problems of real-time and capacity while keeping all the advantages brought by blockchain technology. From this perspective, there are some other directions we can dive in:

1. Creating a side-layer software for existing blockchain infrastructure to deal with huge transactions in a secure but more efficient way and integrate the result of the transactions in this layer to main blockchain. This idea also known as side-chain [\[21\]](#) technology.
2. Build a hybrid system which contains the decentralized blockchain network and a efficient centralized system. I believe a well designed hybrid system can provide service capabilities that are comparable to existing large-scale mature systems while maintaining the advantages of blockchain technology.

9. Bibliography

- [1] Satoshi Nakamoto. "Bitcoin: A peer-to-peer electronic cash system." bitcoin.org, 2009.
- [2] Luu, Loi, et al. "Making Smart Contracts Smarter." Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS'16, 2016, doi:10.1145/2976749.2978309.
- [3] Popper, Nathaniel. "Hacker May Have Taken \$50 Million From Cybercurrency Project." The New York Times, June 2016.
- [4] Ethereum Foundation. "Ethereum's white paper." <https://github.com/ethereum/wiki/wiki/White-Paper>, 2014.
- [5] Miller, Andrew, et al. "Permacoin: Repurposing Bitcoin Work for Data Preservation." 2014 IEEE Symposium on Security and Privacy, 2014, doi:10.1109/sp.2014.37.
- [6] Use case for factom: The world's first blockchain operating system (bos). <http://kencode.de/projects/ePlug/Factom-Linux-Whitepaper.pdf>, Feb 2015.
- [7] Sudhir Khatwani. "How Is Ethereum Blockchain Different From Bitcoin's Blockchain?" <https://coinsutra.com/ethereum-blockchain-vs-bitcoins-blockchain/>, Dec 2017.
- [8] Corbett, James C., et al. "Spanner." ACM Transactions on Computer Systems, vol. 31, no. 3, Jan. 2013, pp. 1–22., doi:10.1145/2518037.2491245.
- [9] Jason, Baker, et al. "Megastore: Providing scalable, highly available storage for interactive services." Proceedings of the Conference on Innovative Data System Research (CIDR), pp. 223–234, 2011.
- [10] Towards secure e-voting using ethereum blockchain[C]//Digital Forensic and Security (ISDFS), 2018 6th International Symposium on. IEEE, 2018: 1-7.
- [11] "The Run smart contract." <https://etherscan.io/address/0xcac337492149bdb66b088bf5914bedfbf78ccc18>.
- [12] "GovernMental Smart Contract." <http://governmental:github.io/GovernMental/>.
- [13] "KingOfTheEtherThrone smart contract." <https://github.com/kieranlby/KingOfTheEtherThrone/blob/v0:4:0/contracts/KingOfTheEtherThrone.sol>.
- [14] Andrew Miller, Brian Warner, and Nathan Wilcox. "Gas economics." <https://github.com/LeastAuthority/ethereum-analyses/blob/master/GasEcon.md>.

- [15] "Protect The Castle Contract." <http://protect-the-castle:ether-contract.org/>.
- [16] "Lottopolo smart contract."
<https://etherchain.org/account/0x0155ce35fe73249fa5d6a29f3b4b7b98732eb2ed>.
- [17] "Random number generator contract." <https://github.com/randao/randao>.
- [18] Joseph Bonneau, Jeremy Clark, and Steven Goldfeder. "On Bitcoin as a public randomness source." Cryptology ePrint Archive, Report 2015/1015, 2015. <http://eprint.iacr.org/>.
- [19] "An online keccak256 hash tool", https://emn178.github.io/online-tools/keccak_256.html
- [20] "Keccak in wikipedia", <https://zh.wikipedia.org/wiki/SHA-3>
- [21] "Side Chain", <https://en.wikipedia.org/wiki/Side-chain>
- [22] "Sample NodeJS backend for SurveyJS library and Editor",
<https://github.com/surveyjs/surveyjs-nodejs>
- [23] "SurveyJS", <https://surveyjs.io/Examples/Service/>
- [24] "SurveyJS Library", <https://surveyjs.io/Overview/Library>
- [25] "GitHub", <https://github.com/>
- [26] "GitLab", <https://about.gitlab.com/>
- [27] "Heroku", <https://www.heroku.com/>
- [28] "Infura - Scalable Blockchain Infrastructure", <https://infura.io/>

10. Appendices

Program flowchart

In order to achieve better results, we try to analyze the charts as much as possible in each chapter.

Program source code with documentation

Different from an algorithm, we developed two APPs, which contains several thousands of lines of codes, it's not necessary to put them all here. Here we only list some key parts of our programs, the full codes can be viewed in gitlab:

API Service project: <https://gitlab.com/skypigr/surveyapi>

Web-portal project: <https://gitlab.com/zhyl321/survey-portal>

Source code of contracts

surveyInstance and surveyFactory.

```
pragma solidity ^0.4.22;
//
http://remix.ethereum.org/#optimize=false&version=soljson-v0.4.24+commit.e67f0147.js
// import "./SafeMath.sol";
// import "./Ownable.sol";

/**
 * @title Survey Instance Contract
 * @author xiao zhu
 * @notice This contract will be call in another contract.
 *
 * The contract takes one param, the address of the creator, and
 * grant that address as while as the msg.sender the promise to
 * protected operations.
 */
contract SurveyInstance {
    mapping(address => bool) internal promise;
```

```

mapping(bytes12 => bool) internal questionHashes;
mapping(bytes12 => bool) internal answerHashes;
mapping(bytes32 => bytes12) internal ans;

/*
    -answer 1: 87
question -> |-answer 2: 6
    -answer 3: 7
*/
mapping(bytes12 => mapping(bytes12 => uint96)) statistic;
event DataSaved(address indexed user, bytes12 indexed key, bytes12 indexed
value);
event InitializedBy(address indexed by);
bool initialized;

/**
 * @param creator : the creator of this contract, the user who triggled this
command
 * @param admin : our address that will be used to interact with the
 * contract in the RESTful API service, this is an array of address.
 * @dev : the only chance to set the promision data is in this constructor
 *
 */
constructor(address creator, address[] admin) public {

    promision[msg.sender] = true;
    promision[creator] = true;
    for(uint i = 0; i < admin.length; i++) {
        promision[admin[i]] = true;
    }
    initialized = false;
}

/**
 * @dev Throws if called by any account without promision.

```

```

    */
    modifier needPromision() {
        require(promision[msg.sender]);
        _;
    }

    /**
     * @param questionHashArray : an array of qestion hashes;
     * @param answerHashArray : an array of answer hashes;
     * This function can only be called once. After initialized, the internal
     * storage "questionHashes" and "answerHashes" can not be modified anymore.
     * the state variable "initialized" will be set to true.
     */
    function initialize(bytes12[] questionHashArray, bytes12[] answerHashArray)
external needPromision {
    require(!initialized);
    for(uint i = 0; i < questionHashArray.length; i++) {
        questionHashes[questionHashArray[i]] = true;
    }

    for(uint j = 0; j < answerHashArray.length; j++) {
        answerHashes[answerHashArray[j]] = true;
    }
    initialized = true;
    emit InitializedBy(msg.sender);
}

    /**
     * @param questionHash : the keccak256 hash of an question
     * @return exist: true if questionHash exist, otherwise return false
     */
    function checkQuestion(bytes12 questionHash) public view returns(bool exist)
    {
        // require(questionHash != 0);

```

```

    exist = questionHashes[questionHash];
}

/**
 * @param answerHash : the keccak256 hash of an answer to some question
 * @return exist : true if answerHash exist, otherwise return false
 */
function checkAnswer(bytes12 answerHash) public view returns(bool exist) {
    // require(answerHash != 0);
    exist = answerHashes[answerHash];
}

/**
 * @param key : the key is a combination of address and questionHash
 * @return exist : true if key exist, otherwise return false
 */
function checkKey(bytes32 key) public view returns(bool exist) {
    // require(key != 0);
    exist = ans[key] != 0;
}

/**
 * @param key : the key is a combination of address and questionHash
 * @param value : answerHash
 * @notice only addresses those have promision can access this function
 * this function require that the input key does exist, that means we don't
 * overwrite, that's because we want to make sure the answer we stored
 * can't be modified in any way.
 */
function saveAnswer(bytes32 key, bytes12 value) external needPromision {
    require(!checkKey(key)); //make sure no duplicated key
    require(checkAnswer(value)); // make sure answer is valid
    bytes12 q = last12bytesOf32(key);
    require(checkQuestion(q)); // make sure question is valid

    ans[key] = value;
}

```

```

    statistic[q][value] += 1;
    emit DataSaved(address(bytes20(key)), q, value);
}

/**
 * @param questionHash : front 12 bytes of question hash
 * @param answerHash : front 12 bytes of answer hash
 * @return count : the number of this question-answer
 * @dev for every question-answer pair return the count. for example:
 * 100 people took this survey, for question A and choice 1,2,3,4
 * the count number for every choice are:
 * A-1: 23
 * A-2: 39
 * A-3: 18
 * A-4: 20
 * the count number is exactly this function will return.
 */
function getStatistic(bytes12 questionHash, bytes12 answerHash) external
view needPromision returns(uint count) {
    count = statistic[questionHash][answerHash];
}

/**
 * @param key : the key is a combination of address and questionHash
 * @return value : the answer hash to the given key
 */
function getAnswer(bytes32 key) external view returns(bytes32 value){
    require(checkKey(key));
    value = ans[key];
}

function checkPromision(address user) external view returns(bool
havePromision) {
    require(user != address(0));
    havePromision = promision[user];
}

```

```

/**
 * @notice : an helper function to get the last 12 bytes of an 32 bytes data
 * @param src : the original 32 bytes data
 * @return rtn : return the last 12 bytes of the input
 */
function last12bytesOf32(bytes32 src) internal pure returns(bytes12 rtn) {
    rtn = bytes12(src<<(32-12)*8);
}

/**
 * @notice : an helper function to get the first 12 bytes of an 32 bytes
data
 * @param src : the original 32 bytes data
 * @return rtn : return the first 12 bytes of the input
 */
function front12bytesOf32(bytes32 src) internal pure returns(bytes12 rtn) {
    rtn = bytes12(src);
}
}

contract SurveyFactory {
    mapping(address => address[]) contractBook;
    address[] public administrators;

    event InstanceCreated(address indexed creator, address indexed
contractAddress);

    constructor(address administrator) public {
        require(administrator != address(0));
        administrators.push(administrator);
    }

    /**
 * @param creator : the creator has the promision to saveAnswer and

```

```

getStatistic
    */
    function createSurvey(address creator) external {
        require(creator != address(0));
        address contractAddress = new SurveyInstance(creator, administrators);
        contractBook[creator].push(contractAddress);
        emit InstanceCreated(creator, contractAddress);
    }

    /**
     * @param creator : the creator whose last contract will be return
     * @dev : return all the surveyInstance contract addresses the creator
    created
     * as an array
     */
    function getContract(address creator) external view returns(address[]
addresses){
        require(creator != address(0));
        addresses = contractBook[creator];
    }

    /**
     * @param creator : the creator whose last contract will be return
     * @dev : return the surveyInstance contract address the creator created
    recently
     */
    function getLastContract(address creator) external view returns (address
contractAddress) {
        require(creator != address(0));
        address[] storage addresses = contractBook[creator];
        contractAddress = addresses[addresses.length-1];
    }
}

```

Implementation of API methods

The file of surveyapi/controller/surveyController.js

```
const utils = require('../src/utils');
const web3 = require('../src/web3/web3');
const instanceContract = require('../src/build/SurveyInstance.json')
const instanceInterface = instanceContract['interface'];

const contractfile = require('../src/build/SurveyFactory.json')
const factoryInterface = contractfile['interface'];

const deployed = require('../src/contracts/deployed.json')
const factoryAddress = deployed['address'];

contractContainer = new Map();
let accounts = new Array();
let gaslimit = '6000000';
let gasprice = web3.utils.toWei('2', 'gwei');
let timeoutSec = 10 * 60 * 1000;

transactionContainer = new Map();

function checkFailure(str) {
  let result = false;
  strStr = str.toString();

  if (strStr.search('revert')) return true;
  if (strStr.search(`Transaction ran out of gas`)) return true;
}

/**
 * index page of server/survey/
 * we just return a simple information about our team
 */
exports.index = function index(req, res, next) {
```



```

res.json({
  title: 'RESTful API Service',
  service: 'survey',
  version: '0.0.2',
  author: 'xzhu',
  email: 'xzhu@scu.edu'
})
}

/**
 * save
 * require param: contractAddress, questionlist, answerlist and userAddress
 * @dev questionlist is a list of bytes12 of the question hash, so to the
 answerlist
 */
exports.save = async function save(req, res, next) {
  let answer;
  let question;
  let contractAddress;
  let userAddress;
  // console.log(req.query);
  if (req.query.answer) { answer = req.query.answer; }
  else { res.status(400).json({ success: false, msg: 'answer field is
required' }); return; }

  if (req.query.question) { question = req.query.question; }
  else { res.status(400).json({ success: false, msg: 'question field is
required' }); return; }

  if (req.query.contractAddress) { contractAddress =
req.query.contractAddress; }
  else { res.status(400).json({ success: false, msg: 'contractAddress field is
required' }); return; }

  if (req.query.userAddress) { userAddress = req.query.userAddress; }
  else { res.status(400).json({ success: false, msg: 'userAddress field is

```

```

required' }); return; }

    if (!web3.utils.isAddress(contractAddress)) {
        res.status(400).json({ success: false, msg: 'contractAddress is invalid'
}); return;
    }

    console.log('incoming SAVE request, query:', req.query);

    if (!contractContainer.has(contractAddress)) {
        //create new surveyInstance
        console.log('New SurveyInstance need to be created');
        surveyInstance = await new
web3.eth.Contract(JSON.parse(instanceInterface), contractAddress);
        contractContainer.set(contractAddress, surveyInstance);
    } else {
        surveyInstance = contractContainer.get(contractAddress);
    }

    qhash = web3.utils.soliditySha3(question);
    ahash = web3.utils.soliditySha3(answer);

    //generate key and value
    key = utils.makeKey(userAddress, qhash);
    console.log(key);

    // in case of the input hash is bytes32, we just need the first 12 bytes.
    // that's a 24 bits hexString + '0x', 26 bits in total.
    value = ahash.slice(0, 26); //0x9e563769c9 1d99840a2f 4956, 24bits

    if (accounts.length == 0) {
        console.log('reloading accounts');
        accounts = await web3.eth.getAccounts();
    }
    // console.log(accounts);
    let hash;

```

```

try {

  surveyInstance.methods.saveAnswer(key, value).
    send({
      from: accounts[0],
      gasPrice: gasprice,
      gas: gaslimit
    })
    .on('transactionHash', txhash => {

      // return the transaction id first, because it will take 15~60
seconds

      // to confirm this transaction, we don't have to wait.
      msg = 'pending'
      hash = txhash;
      transactionContainer.set(hash, msg);
      res.json({
        success: true,
        result: msg,
        msg: hash,
        query: 'save'
      })
      console.log('transactionHash:', hash);
    })
    .on('receipt', async function (receipt) {
      // when we get the receipt finally, we save the contract address
      // in memory for later use.
      msg = receipt['blockHash'];
      hash = receipt['transactionHash'];
      transactionContainer.set(hash, msg);
      console.log('SAVE complete for tx:', hash);
    })
    .on('error', (error) => {
      // this happens when the transaction is reverted. usually
because

      // the operation violates the condition of the contract function

```

```

or

    // the transaction itself has not been successfully created.
    console.error(error);
    if (hash) {
        msg = 'false';
        transactionContainer.set(hash, msg);
        console.error('SAVE error for tx:', hash);
    }
    else {
        //critical error, even don't get a transaction id
        res.status(400).json({
            success: false,
            result: error
        })
        console.error('SAVE encounter fatal error');
    }
})

} catch (error) {
    console.log(error);
    res.json({
        success: false,
        result: error,
        msg: question + ',' + answer,
        query: 'save'
    })
}
}

/**
 * checkQuestion
 * require param: contractAddress, question
 */
exports.checkQuestion = async function checkQuestion(req, res, next) {
    let question;
    let contractAddress;

```

```

    if (req.query.question) { question = req.query.question; }
    else { res.status(400).json({ success: false, msg: 'question field is
required' }); return; }

    if (req.query.contractAddress) { contractAddress =
req.query.contractAddress; }
    else { res.status(400).json({ success: false, msg: 'contractAddress field is
required' }); return; }

    if (!web3.utils.isAddress(contractAddress)) {
        res.status(400).json({ success: false, msg: 'contractAddress is invalid'
}); return;
    }

    console.log('incoming checkQuestion request, query:', req.query);

    if (!contractContainer.has(contractAddress)) {
        //create new surveyInstance
        console.log('New SurveyInstance need to be created');
        surveyInstance = await new
web3.eth.Contract(JSON.parse(instanceInterface), contractAddress);
        contractContainer.set(contractAddress, surveyInstance);
    } else {
        surveyInstance = contractContainer.get(contractAddress);
    }

    qhash = web3.utils.soliditySha3(question).slice(0, 26);

    try {
        let rtn = await surveyInstance.methods.checkQuestion(qhash).call();
        res.json({
            success: true,
            result: rtn,
            msg: qhash,
            query: 'checkQuestion'

```

```

    })
  } catch (error) {
    console.log(error);
    res.json({
      success: false,
      result: error,
      msg: qhash,
      query: 'checkQuestion'
    })
  }
}

/**
 * checkAnswer
 * require param: contractAddress, answer
 */
exports.checkAnswer = async function checkAnswer(req, res, next) {
  let answer;
  let contractAddress;
  if (req.query.answer) { answer = req.query.answer; }
  else { res.status(400).json({ success: false, msg: 'answer field is
required' }); return; }

  if (req.query.contractAddress) { contractAddress =
req.query.contractAddress; }
  else { res.status(400).json({ success: false, msg: 'contractAddress field is
required' }); return; }

  if (!web3.utils.isAddress(contractAddress)) {
    res.status(400).json({ success: false, msg: 'contractAddress is invalid'
}); return;
  }

  console.log('incoming checkAnswer request, query:', req.query);

  if (!contractContainer.has(contractAddress)) {

```

```

    //create new surveyInstance
    console.log('New SurveyInstance need to be created');
    surveyInstance = await new
web3.eth.Contract(JSON.parse(instanceInterface), contractAddress);
    contractContainer.set(contractAddress, surveyInstance);
  } else {
    surveyInstance = contractContainer.get(contractAddress);
  }

  ahash = web3.utils.soliditySha3(answer).slice(0, 26);
  try {
    let rtn = await surveyInstance.methods.checkAnswer(ahash).call();
    res.json({
      success: true,
      result: rtn,
      msg: ahash,
      query: 'checkAnswer'
    })
  } catch (error) {
    console.log(error);
    res.json({
      success: false,
      result: error,
      msg: ahash,
      query: 'checkAnswer'
    })
  }
}

/**
 * getStatic
 * require param: contractAddress, question, answer
 */
exports.getStatic = async function getStatic(req, res, next) {

```

```

let answer;
let question;
let contractAddress;

if (req.query.answer) { answer = req.query.answer; }
else { res.status(400).json({ success: false, msg: 'answer field is
required' }); return; }

if (req.query.question) { question = req.query.question; }
else { res.status(400).json({ success: false, msg: 'question field is
required' }); return; }

if (req.query.contractAddress) { contractAddress =
req.query.contractAddress; }
else { res.status(400).json({ success: false, msg: 'contractAddress field is
required' }); return; }

// console.log('contractContainer size:', contractContainer.size);

if (!web3.utils.isAddress(contractAddress)) {
  { res.status(400).json({ success: false, msg: 'contractAddress is
invalid' }); return; }
}

console.log('incoming getStatic request, query:', req.query);

if (!contractContainer.has(contractAddress)) {
  //create new surveyInstance
  console.log('New SurveyInstance need to be created');
  surveyInstance = await new
web3.eth.Contract(JSON.parse(instanceInterface), contractAddress);
  contractContainer.set(contractAddress, surveyInstance);
} else {
  surveyInstance = contractContainer.get(contractAddress);
}

```



```

if (accounts.length == 0) {
  console.log('reloading accounts');
  accounts = await web3.eth.getAccounts();
}

qhash12 = web3.utils.soliditySha3(question).slice(0, 26);
ahash12 = web3.utils.soliditySha3(answer).slice(0, 26);
try {
  surveyInstance.methods.getStatistic(qhash12, ahash12)
    .call({ from: accounts[0] })
    .then((result) => {
      res.json({
        success: true,
        result: result,
        msg: qhash12 + ', ' + ahash12,
        query: 'getStatic'
      })
    })
} catch (error) {
  console.log(error);
  res.json({
    success: false,
    result: error,
    msg: qhash12 + ', ' + ahash12,
    query: 'getStatic'
  })
}
}

/**
 * createInstance
 * require param: creator
 * return: instance address
 */

```

```

exports.create = async function createInstance(req, res, next) {

  // console.log(req.query);

  let creator;
  if (req.query.creator) { creator = req.query.creator; }
  else { res.status(400).json({ success: false, msg: 'creator field is
required' }); return; }

  if (!web3.utils.isAddress(creator)) {
    res.status(400).json({ success: false, msg: 'creator address is invalid'
}); return;
  }

  console.log('incoming CREATE request, creator:', creator);
  if (!contractContainer.has(factoryAddress)) {
    //create new surveyInstance
    console.log('rebuilt factory contract');
    factoryContract = await new
web3.eth.Contract(JSON.parse(factoryInterface), factoryAddress);
    contractContainer.set(factoryAddress, factoryContract);
  } else {
    factoryContract = contractContainer.get(factoryAddress);
  }

  if (accounts.length == 0) {
    console.log('reloading accounts');
    accounts = await web3.eth.getAccounts();
  }
  try {
    let hash;
    const creation = factoryContract.methods.createSurvey(creator).
      send({
        from: accounts[0],
        gas: gaslimit,

```

```

        gasPrice: gasprice
    })
    .on('transactionHash', txhash => {

        // console.error(error);
        msg = 'pending'
        hash = txhash;
        transactionContainer.set(hash, msg);
        res.json({
            success: true,
            result: msg,
            msg: hash,
            query: 'create'
        })
        console.log('transactionHash:', hash);
    })

    .on('receipt', async function (receipt) {
        // console.log(receipt);
        await factoryContract.methods.getLastContract(creator).call()
            .then((address) => {
                hash = receipt['transactionHash'];
                transactionContainer.set(hash, address);
                console.log('CREATE complete for tx', hash);
            })
    })

    .on('error', (error) => {

        if (hash) {
            msg = 'false';
            transactionContainer.set(hash, msg);
            console.error(error);
            console.log('CREATE error for tx', hash);
        }
        else {
            res.status(400).json({

```

```

        success: false,
        result: error
    })
    console.error('CREATE encounter fatal error');
}
})
}
catch (error) {
    console.log(error);
    res.json({
        success: false,
        result: error,
        msg: creator,
        query: 'create'
    })
}
}
}

/**
 * createInstance
 * require param: contractAddress, questionlist and answerlist
 * return: true/false
 * questionlist is a stringified array, for example:
 * q = ['a', 'b'];
 * qestionlist = JSON.stringify(q);
 */
exports.init = async function initialize(req, res, next) {
    let answerlist;
    let questionlist;
    let contractAddress;
    let qlist, alist

    if (req.query.contractAddress) { contractAddress =
req.query.contractAddress; }
    else { res.status(400).json({ success: false, msg: 'contractAddress field is

```

```

required' }); return; }

    if (!web3.utils.isAddress(contractAddress)) {
        res.status(400).json({ success: false, msg: 'contractAddress is invalid'
}); return;
    }

    if (req.query.answerlist) { answerlist = req.query.answerlist; }
    else { res.status(400).json({ success: false, msg: 'answerlist field is
required' }); return; }

    if (req.query.questionlist) { questionlist = req.query.questionlist; }
    else { res.status(400).json({ success: false, msg: 'questionlist field is
required' }); return; }

    console.log('incoming INIT request, query:', req.query);

    try {
        qlist = JSON.parse(questionlist);
        alist = JSON.parse(answerlist);
    } catch (error) {
        console.log('ERROR, invalid json parameter, returned 400');
        res.status(400).json({ success: false, msg: 'invaidd json parameter ' });
        return;
    }

    if (!contractContainer.has(contractAddress)) {
        //create new surveyInstance
        console.log('New SurveyInstance need to be created');
        surveyInstance = await new
web3.eth.Contract(JSON.parse(instanceInterface), contractAddress);
        contractContainer.set(contractAddress, surveyInstance);
    } else {

```

```

    surveyInstance = contractContainer.get(contractAddress);
}

if (accounts.length == 0) {
    console.log('reloading accounts');
    accounts = await web3.eth.getAccounts();
}

// console.log(qlist, alist);
try {
    let hash;
    const creation = surveyInstance.methods.initialize(qlist, alist).
        send({
            from: accounts[0],
            gas: gaslimit,
            gasPrice: gasprice
        })
        .on('transactionHash', txhash => {

            // console.error(error);
            msg = 'pending'
            hash = txhash
            transactionContainer.set(hash, msg);
            res.json({
                success: true,
                result: msg,
                msg: hash,
                query: 'init'
            })
            console.log('transactionHash:', hash);
        })
        .on('receipt', async function (receipt) {
            // console.log(receipt);
            msg = receipt['blockHash'];
            hash = receipt['transactionHash'];

```

```

        transactionContainer.set(hash, msg);
        console.log('INIT complete for tx:', hash);
    })
    .on('error', (error) => {
        console.error(error);
        if (hash) {
            msg = 'false';
            transactionContainer.set(hash, msg);

            console.error('INIT error for tx:', hash);
        }
        else {
            //critical error, even don't get a transaction id
            res.status(400).json({
                success: false,
                result: error
            })
            console.error('INIT encounter fatal error');
        }
    })
})

}
catch (error) {
    console.log('#####', error);
    res.json({
        success: false,
        result: error,
        msg: '',
        query: 'init'
    })
}
}

exports.gettx = async function getTransaction(req, res, next) {
    let txid;

```

```

    if (req.query.txid) { txid = req.query.txid; }
    else { res.status(400).json({ success: false, msg: 'txid field is required'
}); return; }

    console.log('incoming GETTX request, txid:', req.query.txid);
    if (transactionContainer.has(txid)) {
        res.json({
            success: true,
            result: transactionContainer.get(txid),
            msg: txid,
            query: 'gettx'
        })
    }
    else {
        res.json({
            success: false,
            result: null,
            msg: txid,
            query: 'gettx'
        })
    }
}
}

```

Input/output listing

Live demo of survey-portal: <http://survey.skypigr.info>

Live demo of api server: <http://surveyapi.skypigr.info/>

create

methods: get,post

server: <http://surveyapi.skypigr.info/survey/create>

params:

1. creator: the address of the creator, which will be stored in the surveyContract

example:

<http://surveyapi.skypigr.info/survey/create?creator=0xE6F7028d239d061C4265630fe108c4B3128e559B>

return:

```
{
  "success": true,
  "result": "pending",
  "msg": "0x70ea8621a490d804efc01a4fb9a014f181539f8676fc27a95a0c218a48a3c924",
  "query": "create"
}
```

This methods will return an txid in the msg field as soon as the transaction is broadcasted to the network. But it need 15~60 seconds to be confirmed, until then we can get the contract address of the new survey instance we just created, as shown in the “result” field of this method, the status is pending. We can check the status of this transaction through “**gettx**” method according to the txid we just received.

Remind, only when we get the address of a contract can we do further operation on the contract.

gettx

methods: get,post

server: <http://surveyapi.skypigr.info/survey/create>

params:

1. txid : the transaction id we want to check.

the txid may from different method(create,init, save), which will return a different type of result.

for create: the gettx will return the contract address in the result field when tx is confirmed.

for init and save: the gettx will return the blockhash in the “result” field when the transaction is confirmed, otherwise it will return pending in “result” field.

return:

```
{
  "success": true,
  "result": "pending",
  "msg": "0x70ea8621a490d804efc01a4fb9a014f181539f8676fc27a95a0c218a48a3c924",
  "query": "gettx"
}
```

After a few seconds, when the transaction is confirmed, the result field will be replaced by a blockhash in which the transaction was included, or a contract address. We can use this to determine if the transaction has been confirmed.

```
{
  "success": true,
  "result": "0x455f56ad388ddf8a68571Ad4652E388Bee3666f8",
  "msg": "0x70ea8621a490d804efc01a4fb9a014f181539f8676fc27a95a0c218a48a3c924",
}
```

```
"query": "gettx"  
}
```

In addition, the transaction id returned by method of create, init and save, can be used in other public blockchain explorer, like <https://rinkeby.etherscan.io>. For example, instead of using our “gettx” method, you can check the state of a transaction by querying: <https://rinkeby.etherscan.io/tx/0x70ea8621a490d804efc01a4fb9a014f181539f8676fc27a95a0c218a48a3c924>

which includes more information about this transaction.

The screenshot shows the Etherscan Rinkeby Testnet interface. At the top, there is a search bar with the text "Search by Address / Txhash / Block / Token / Ens" and a "GO" button. Below the search bar are navigation tabs: HOME, BLOCKCHAIN (selected), TOKEN, CHART, and MISC. The main heading is "Transaction 0x70ea8621a490d804efc01a4fb9a014f181539f8676fc27a95a0c218a48a3c924". Below this, there are breadcrumb links: Home / Transactions / Transaction Information. The transaction details are shown in a table-like format:

TxHash:	0x70ea8621a490d804efc01a4fb9a014f181539f8676fc27a95a0c218a48a3c924
TxReceipt Status:	Success
Block Height:	2398121 (34 block confirmations)
TimeStamp:	8 mins ago (Jun-03-2018 04:04:53 PM +UTC)
From:	0x1aca1282970840a5814f060a7921e3f90ce63a4c
To:	Contract 0x63768c6a762a8c20358888dcfb86b0fa1f2da51e
Value:	0 Ether (\$0.00)
Gas Limit:	6000000
Gas Used By Txn:	433882
Gas Price:	0.000000002 Ether (2 Gwei)
Actual Tx Cost/Fee:	0.000867764 Ether (\$0.000000)
Nonce & (Position):	577 (4)
Input Data:	<pre>0x28ae1d6200000000000000000000000000000000e6f7028d239d061c4265630fe108c4b3128e559b</pre> Convert To UTF8

init

methods: get,post

server: <http://surveyapi.skypigr.info/survey/init>

params:

4. contractAddress
5. questionlist
6. answerlist

example:

[http://surveyapi.skypigr.info/survey/init?questionlist=\["0xfbb236712beb41e86af5bbcf358df9c601db8e1d54cd5178bf8a5a538d06a2d2"\]&answerlist=\["0x9e563769c91d99840a2f49560ab96949f868e6b2f55c29481e5300881ac4964d","0xbe83a48bf35c1402f9efaa687c50a92fb78c5e4323b4379af5cf699171e608c6","0x9d886c24dbbc130ac49e3e3f87a9dd3b71a23548da8973674d2cb5b2678ab767","0xd6d2bdabef5123f0dc370bbc761941f57371fe50899d69248ebcc2d9aa9d7d8c"\]&contractAddress=0x455f56ad388ddf8a68571Ad4652E388Bee3666f8](http://surveyapi.skypigr.info/survey/init?questionlist=[)

the questionlist and answerlist is an array of question hashes.

return:

```
{
  "success": true,
  "result": "pending",
  "msg": "0x44c3bbc6a84dd2c76cf7878c313211195d144470990b68336d9180445fe607b7",
  "query": "init"
}
```

the contract received a bytes12[] array, but you can still submit a bytes32[] array, the contract will use the first 12 bytes only. or you can cut the remaining 20 bytes to save traffics. let take a simple question-answer as an example. As you can see, the example link given above use the full bytes32 as param to the init method, we can also take the bytes12 as in input param.

like: [http://surveyapi.skypigr.info/survey/init?questionlist=\["0xfbb236712beb41e86af5bbcf"\]&answerlist=\["0x9e563769c91d99840a2f4956","0xbe83a48bf35c1402f9efaa68","0x9d886c24dbbc130ac49e3e3f","0xd6d2bdabef5123f0dc370bbc"\]&contractAddress=0x455f56ad388ddf8a68571Ad4652E388Bee3666f8](http://surveyapi.skypigr.info/survey/init?questionlist=[)

	source	Keccak-256(bytes32)	bytes12
question	How old are you?	0xfbb236712beb41e86af5bbcf358df9c601db8e1d54cd5178bf8a5a538d06a2d2	0xfbb236712beb41e86af5bbcf
answer	I'am 2	0x9e563769c91d99840a2f49560ab96949f868e6b2f55c29481e5300881ac4964d	0x9e563769c91d99840a2f4956
	I'am 4	0xbe83a48bf35c1402f9efaa687c50a92fb78c5e4323b4379af5cf699171e608c6	0xbe83a48bf35c1402f9efaa68
	I'am 6	0x9d886c24dbbc130ac49e3e3f87a9dd3b71a23548da8973674d2cb5b2678ab767	0x9d886c24dbbc130ac49e3e3f
	I'am 8	0xd6d2bdabef5123f0dc370bbc761941f57371fe50899d69248ebcc2d9aa9d7d8c	0xd6d2bdabef5123f0dc370bbc

return:

```
{
  "success": true,
  "result": "pending",
  "msg": "0xb9f44109cba22dcbbba38aa90c1a06d4b5cc5958b607f7abfb7318c4265421306",
  "query": "init"
}
```

after a while, gettx(txid) =>

```
{
  "success": true,
  "result": "0x80acd875b63c2697f0cb12e1b415e11bca2b367fd10e696d587edd67ead26381",
  "msg": "0xb9f44109cba22dcbbba38aa90c1a06d4b5cc5958b607f7abfb7318c4265421306",
  "query": "gettx"
}
```

this method will return a transaction id in the msg field as soon as the transaction is broadcasted to the network. we can check the status of this transaction by querying the txid through our “gettx” method or in any public blockchain explorer, like rinkeby.etherscan.io.

Because each contract can only be initialized once, thus, if you try to submit two “init” request for the same contract, the result the second request will be false, although it will return a different txid. when this tx is confirmed, the gettx method will return the following result for this txid:

```
{
  "success": true,
  "result": "false",
  "msg": "0x0703f2e9b4b5d84c855abca05f42019e942ece943d7eb599bcb23291f137bf7d",
  "query": "gettx"
}
```

the result field of the return is “false”, which a valid “init” operation will return a blockhash in the result field of gettx.

checkQuestion

description: return true if given question is in the contract, otherwise return false

methods: get, post

server: <http://surveyapi.skypigr.info/survey/checkq>

params:

1. contractAddress
2. question

example:

<http://surveyapi.skypigr.info/survey/checkq?question=How old are you?&contractAddress=0x455f56ad388ddf8a68571Ad4652E388Bee3666f8>

```
return:
{
  "success": true,
  "result": true,
  "msg": "0xfbb236712beb41e86af5bbc",
  "query": "checkQuestion"
}
```

The "msg" field of this query is the first 12 bytes of the hash of the given question source.

If the given question source is in the contract, the result field of the return is true, otherwise it will return false. For example, if we change "How" to "how", the request will return a false result.

<http://surveyapi.skypigr.info/survey/checkq?question=how old are you?&contractAddress=0x455f56ad388ddf8a68571Ad4652E388Bee3666f8>

```
{
  "success": true,
  "result": false,
  "msg": "0x0aba6f22398a23c6e66d4298",
  "query": "checkQuestion"
}
```

checkAnswer

description: return true if given answer is in the contract, otherwise return false

methods: get, post

server: <http://surveyapi.skypigr.info/survey/checka>

params:

1. contractAddress
2. answer

example:

<http://surveyapi.skypigr.info/survey/checka?answer=I'am 2&contractAddress=0x455f56ad388ddf8a68571Ad4652E388Bee3666f8>

```
return:
{
  "success": true,
  "result": false,
  "msg": "0x315caea8e93dc85e38af2745",
  "query": "checkAnswer"
}
```

save

description: save question-answer into blockchain.

methods: get, post

server: <http://surveyapi.skypigr.info/survey/save>

params:

5. contractAddress
6. question
7. answer
8. userAddress

example:

<http://surveyapi.skypigr.info/survey/save?contractAddress=0x455f56ad388ddf8a68571Ad4652E388Bee3666f8&question=How old are you?&answer=I'am 2&userAddress=0x49C57Da993072C72570ED251fc108126B126B6d9>

return:

```
{
  "success": true,
  "result": "pending",
  "msg": "0xc702c2bc0b531b9e3707295224ca0877d2ddd588a6441028fa642727613d74f4",
  "query": "save"
}
```

in the msg field, a txid will be return.

after a while, gettx(txid) =>

```
{
  "success": true,
  "result": "0xd44a26170d90af793f567ecc1cd36cfbb5c063a9e1219dc842ffae0ff07065f",
  "msg": "0xc702c2bc0b531b9e3707295224ca0877d2ddd588a6441028fa642727613d74f4",
  "query": "gettx"
}
```

getStatistic

description: given question-answer pair, return the count number

methods: get, post

server: <http://surveyapi.skypigr.info/survey/stat>

params:

4. contractAddress
5. question
6. answer

example:

<http://surveyapi.skypigr.info/survey/stat?answer=I'am 2&contractAddress=0x455f56ad388ddf8a68571Ad4652E388Bee3666f8&question=How old are you?>

return:

```
{
  "success": true,
  "result": "1",
  "msg": "0xfbb236712beb41e86af5bbcf,0x9e563769c91d99840a2f4956",
  "query": "getStatic"
}
```

the result field is the count of this answer chosen by user.

Gas consume

contract	function	gas
surveyFactory	createInstance	437752
surveyInstance	initialize	156196
	saveAnswer	25284