



UNIVERSITÀ PARTHENOPE DI NAPOLI

PROGETTO ESAME "RETI DI CALCOLATORI E LAB."

Ethvent

Alessio MADDALUNO (0124/1455), Salvatore NANNI(0124/1598)

Docente:
Prof. Alessio FERONE

Indice

1	Descrizione Progetto	3
1.1	Traccia progetto	3
1.2	Descrizione contratti	3
2	Descrizione e schemi dell'architettura	4
2.1	Blockchain	4
2.2	Ethvent	5
2.3	EOA - User e Owner	5
3	Descrizione e schemi del protocollo applicazione	6
3.1	Descrizione del protocollo applicazione	6
3.2	Schemi del protocollo	6
4	Dettagli implementativi	9
4.1	Contratti	9
4.1.1	CheckInEvent	9
4.1.2	RegisterEvent	11
4.2	Webapp	12
4.2.1	Struttura della webapp	12
4.2.2	Routes	12
4.2.3	Templating delle pagine web	13
4.2.4	Transazioni dal browser	13
5	Manuale Utente	15
5.1	Requisiti	15
5.2	Configurazione Ganache	15
5.3	Installare e configurare Ethvent	16
5.4	Creazione eventi	17
5.5	Gestione eventi e permessi Metamask	18
5.6	Attività sulla blockchain	20
6	Limitazioni e miglioramenti	21

Capitolo 1

Descrizione Progetto

1.1 Traccia progetto

Si vuole realizzare un sistema di prenotazione di eventi basato sulla blockchain di Ethereum facendo uso di smart contracts.

È data la possibilità ad un utente di registrare la propria presenza ad un evento, inviando una certa quantità di ether ad un contratto. Partecipando realmente all'evento, effettuando quindi un check-in (interagendo con un contratto apposito), la quantità di ether inviata in fase di registrazione è resa all'utente.

Se dopo aver effettuato la prenotazione, l'utente non effettua il check-in entro un limite temporale (si è assunto sia la data e l'ora d'inizio dell'evento), gli ether non restituiti saranno trasferiti al proprietario del contratto, fornitore di tale servizio.

È inoltre prevista la realizzazione di un interfaccia web che consenta di interagire con questo servizio.

1.2 Descrizione contratti

Il sistema è basato principalmente sull'utilizzo di due smart contracts: RegisterEvent e CheckInEvent.

- **RegisterEvent:**

- Verifica che la quantità¹ di ether inviati sia corretta;
- Verifica che l'indirizzo dell'utente non sia già registrato;
- Registra l'indirizzo dell'utente;
- Invia gli ether al CheckInEvent;

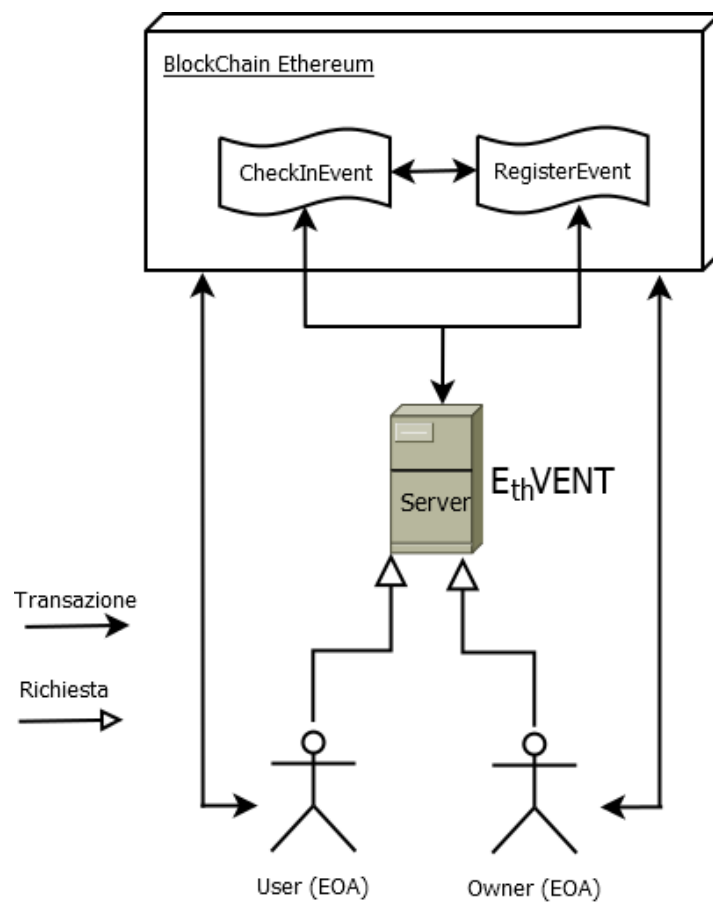
- **CheckinEvent:**

- Verifica se l'indirizzo dell'utente che sta eseguendo il check-in è registrato in RegisterEvent;
- In caso affermativo gli restituisce gli ether inviati;
- Alla scadenza del tempo massimo, trasferisce al proprietario del contratto gli ether non restituiti;

¹Tale quantità definisce quanti ether l'utente dovrà inviare al contratto per poter effettuare la prenotazione.

Capitolo 2

Descrizione e schemi dell'architettura



2.1 Blockchain

L'architettura del progetto utilizza la blockchain di **Ethereum**, per realizzare un'applicazione distribuita (*decentralized app*).

In tale blockchain, ogni nodo è un *peer* e contiene una copia di tutte le informazioni contenute nella blockchain.

Ogni modifica attuata da una transazione deve essere ricevuta e approvata dalla maggioranza dei nodi affinché venga realmente effettuata.

Inoltre, in tale blockchain per distribuire un servizio vengono definiti dei contratti (*Smart*

Contracts), che possono essere richiamati da *Externally Owned Accounts* (utenti esterni) per svolgere la funzione proposta, attraverso delle *transazioni*: veri e propri messaggi tra mittente e destinatario. Gli stessi contratti possono interagire tra di loro mediante transazioni, ponendosi sia come client nei confronti di un altro contratto, che farà da server, e viceversa.

In particolare in un sistema quale "Ethvent" con una componente decentralized, le informazioni relative ai contratti di prenotazione e checkin associate a ogni evento non saranno presenti su di un unico server, ma *distribuite* sulla blockchain.

In questo modo si garantisce un uso più *democratico* e *sicuro* delle informazioni, poichè ogni modifica dovrà essere approvata dalla maggioranza dei nodi affinchè vada a buon fine.

2.2 Ethvent

Ethvent fornisce un'*interfaccia web* per interagire con i contratti. In particolare fornisce tutte le informazioni necessarie all'utente affinchè possa effettuare le transazioni necessarie. Ethvent non si pone come intermediario tra la blockchain e l'utente, ciò poichè sarebbe necessario gestire informazioni sensibili degli utenti quali la loro chiave privata. In Ethvent è possibile creare degli eventi e per scopo didattico simularne la scadenza. Il gas utilizzato per la loro creazione è fornito dall'owner (il proprietario del servizio). Le uniche transazioni che effettua attivamente sono quelle relative alla gestione dei contratti (creazione e cancellazione).

2.3 EOA - User e Owner

Gli attori rappresentati nello schema dell'architettura¹, utilizzatori del servizio fornito, sono due account externally owned della blockchain:

- un "**Utente**", mero utilizzatore del servizio, può:
 - prenotarsi ad un evento;
 - svolgere il check in.
- il "**Proprietario**" (dell'intero servizio) può:
 - creare un nuovo evento;
 - effettuare la chiusura delle prenotazioni;
 - ricevere gli ether degli user che non hanno effettuato il checkin in tempo.

¹Figura a pagina 4

Capitolo 3

Descrizione e schemi del protocollo applicazione

3.1 Descrizione del protocollo applicazione

Il protocollo applicazione che si viene a creare, prevede la presenza di due attori principali: un account proprietario (EOA) che crea i contratti e un insieme di utenti (users) utilizzatori del servizio fornito dai contratti.

Nello specifico l'applicazione deve fornire un metodo per prenotarsi ad un evento, permettendo all'informazione relativa a tale prenotazione di risiedere sulla blockchain, essendo così salvata su ogni nodo di essa.

Tale servizio è offerto realizzando due contratti, "**RegisterEvent**" e "**CheckinEvent**", che cooperando permettono di:

- Creare un nuovo evento;
- Prenotarsi a un evento, mediante una transazione in cui è inviato il corretto ammontare di ether richiesti dalla prenotazione;
- Effettuare il check in, vedendosi restituiti gli ether spesi per la prenotazione.

3.2 Schemi del protocollo

All'interno della blockchain di Ethereum, per lo scambio di messaggi tra contratti e tra un contratto e un utente, sono realizzate delle transazioni che corrispondono a veri e propri *pacchetti*, in grado di inviare informazioni tra entità pari.

Una transazione è composta:

- **destinatario** del messaggio:
è il destinatario della transazione, contrassegnato da un indirizzo di 20 byte. Può essere un externally owned account oppure un contratto (contract account);
- firma del **mittente**:
è l'indirizzo dell'account che invia la transazione, può essere, come nel destinatario, un EOA o un Contract account;
- **valore** di Ether da trasferire: indica il numero di ether inviati dal mittente al destinatario;
- un campo **dati** opzionale: il campo dati è utile nelle decentralized apps per definire un comportamento dei contratti, inviando transazioni con dati si permette l'utilizzo dinamico dei contratti;

- un valore **gasLimit**: è il limite imposto sul consumo di gas che l'esecuzione può generare. Poichè Ethereum definisce un sistema **Turing completo**, l'esecuzione di problemi intrattabili viene scongiurata concludendo l'esecuzione al termine del gas.
- un valore **gasPrice**: è il prezzo in ether associato all'unità di gas, è da moltiplicare al gasLimit per determinare il costo in ether dell'esecuzione.

In particolare il protocollo applicazione prevede un iniziale scambio di messaggi, mediante una transazione, tra l'account utente (externally owned) che intende registrarsi a un evento ed il contratto "RegisterEvent".

In questa transazione, iniziale, l'utente specifica il numero di ether necessario alla registrazione (nel campo valore). Sarà compito di tale contratto la verifica della correttezza dell'ammontare inviato.

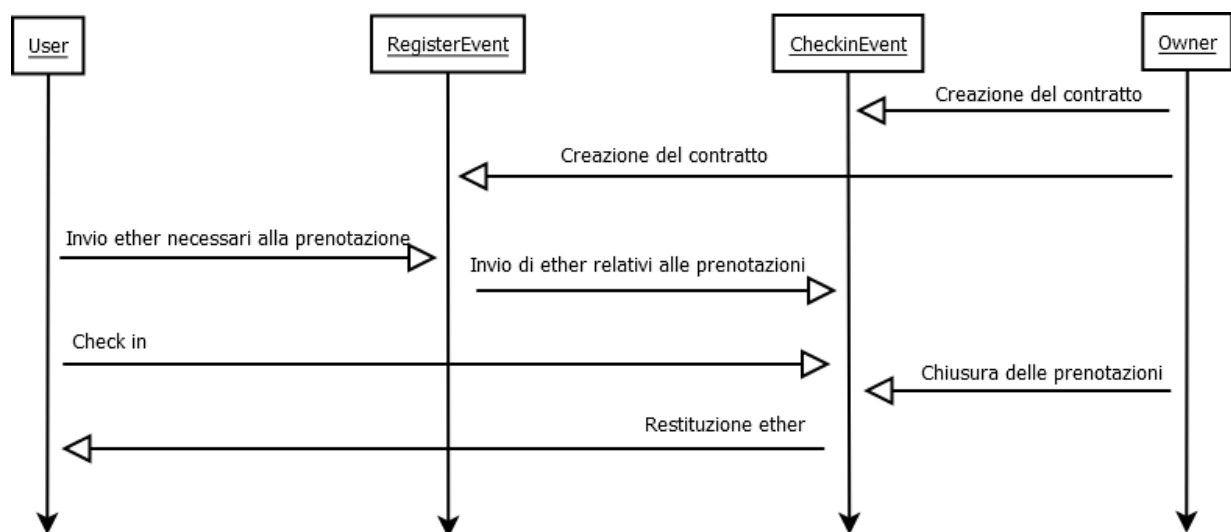
Mediante il campo mittente della transazione, il contratto è in grado di salvare l'indirizzo dell'account che ha fatto richiesta di registrazione.

Il compito del "RegisterEvent" diventa, dopo aver salvato i campi relativi all'utente, di inviare gli ether ricevuti e l'indirizzo del mittente al secondo contratto: "CheckinEvent".

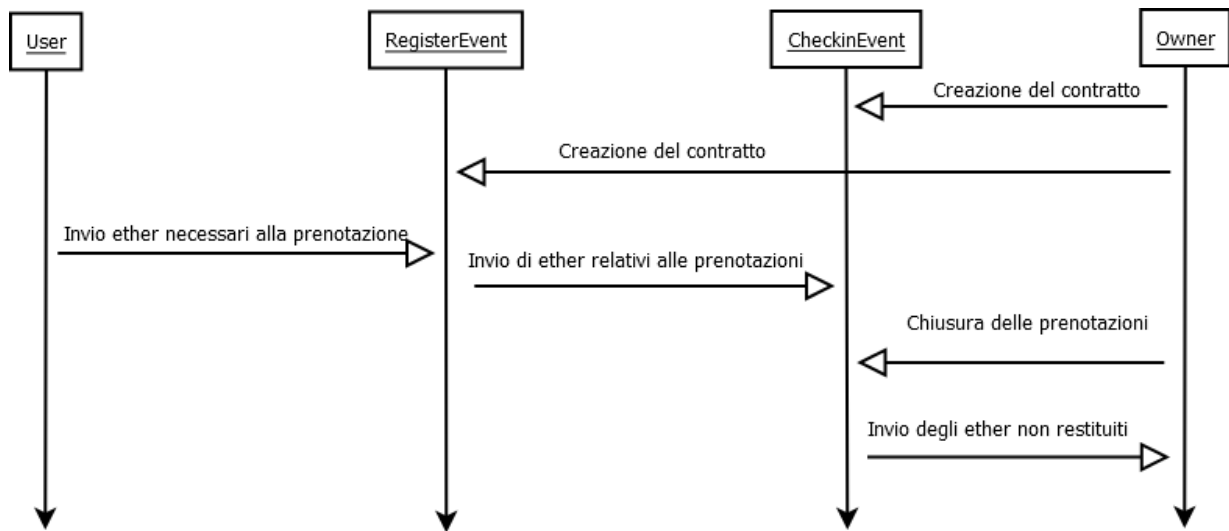
Tale contratto deterrà gli ether, associati alle prenotazioni per un singolo evento, fino alla scadenza del tempo di prenotazione (coincidente con l'inizio dell'evento).

A quel punto gli utenti che avranno effettuato il check in, avviando una transazione verso il contratto "CheckinEvent", vedranno restituiti gli ether di cauzione.

Per fare ciò sarà il contratto a creare una transazione verso l'account dell'user (EOA) con l'ammontare di ether spesi per la prenotazione, restituendoli.



Per gli utenti, invece, che non avranno effettuato il checkin in tempo utile, l'ammontare non sarà restituito bensì il contratto "CheckinEvent" creerà una transazione verso il proprietario del contratto stesso, che avrà come campo valore l'ammontare di ether non restituiti. Il contratto riceverà una transazione (temporizzata) da parte del proprietario volta alla distruzione dei due contratti associati all'evento e l'esecuzione dell'operazione.



Capitolo 4

Dettagli implementativi

4.1 Contratti

Gli smart contracts sono scritti in **Solidity**, un linguaggio ad alto livello atto proprio alla loro scrittura. È un linguaggio largamente ispirato al C++, Python e Javascript. Compilare un file .sol significa tradurre ciò che è scritto in Solidity in **bytecode** eseguibile dalla EVM (Ethereum Virtual Machine), in maniera molto simile a ciò che accade con Java. Eseguire operazioni con la EVM cambia lo stato globale di Ethereum (essendo lo stato condiviso, in quanto blockchain). Inoltre, ogni operazione eseguita costa del **gas**, ovvero del "carburante" acquistato con la valuta della blockchain. In questo modo è scongiurata la possibilità che un deadlock possa bloccare l'intero sistema. Possibili esecuzioni di codice malevolo atto a bloccare il sistema si ridurranno al solo spreco di gas da parte dell'esecutore dell' "attacco"; ne consegue che anche il tentativo di bloccare per un tempo considerevole il sistema si traduce, in un'attività economicamente dispendiosa.

4.1.1 CheckInEvent

Il primo contratto di cui è fatto il deploy ad ogni nuovo evento è CheckInEvent. Nella fase di deploy è necessario passare al costruttore del contratto la data massima di check in.

```
1 constructor(uint256 date) public{
2     owner = msg.sender; //set the owner of the contract
3     limitDate = date;    //set the limit date of the event
4 }
```

Tale data è in formato **Unix time**, il fuso di riferimento è UTC. È **necessario** quindi convertire tale data in questo formato affinché l'operazione, legata al limite temporale del contratto, sia correttamente sincronizzata. Il costruttore è richiamato da chi effettua il deploy del contratto in quanto include delle istruzioni da eseguire oltre quelle necessarie al deploy stesso; di conseguenza il **msg.sender** sarà il proprietario del contratto.

Per poter identificare il RegisterEvent, è necessario creare un metodo richiamabile da quest'ultimo che consenta di memorizzare l'indirizzo del contratto: quando il RegisterEvent eseguirà il metodo sarà il **msg.sender** della transazione.

```
1 function setRegister(address _contract) external {
2     if(_registerContract == address(0x0)){
3         _registerContract = _contract;
```

```

4         registerContract = BookingManagerContract(_registerContract);
5     }
6 }

```

Per rendere il contratto pagabile da un altro contratto, è necessario fare un override della funzione "function ()", una funzione di fallback. Tale funzione accetterà pagamenti solo del RegisterEvent associato:

```

1 function () external payable {
2     require(
3         address(registerContract) == msg.sender,
4         'Solo il Register associato puo inviare ether a tale contratto'
5     );
6 }

```

Per poter effettuare il *check in* è necessario verificare che la prenotazione sia valida. Ciò è verificabile solo all'interno di RegisterEvent: verrà quindi eseguito un metodo di quest'ultimo. Se la prenotazione è valida, gli ether sono restituiti all'utente. La quantità restituita sarà leggermente **inferiore** in quanto queste operazioni - che coinvolgono l'esecuzione di diverse istruzioni - sono pagate attraverso del gas dall'utente che effettua la transazione:

```

1 function checkIn() public {
2     //Check if msg.sender have a valid booking in the register contract
3     if(!registerContract.checkBooking(msg.sender)){
4         revert('Prenotazione non valida');
5     }
6     //Refound the Ethereum
7     msg.sender.transfer(registerContract.getEventDeposit());
8 }

```

Non essendo possibile programmare un'esecuzione *time-driven* all'interno di una blockchain e non potendo effettuare **polling**, il limite temporale del checkIn e le operazione che ne conseguono sono eseguite attraverso una transazione che deve essere **necessariamente temporizzata all'esterno della blockchain**. Per evitare un'autodistruzione involontaria sono fatti controlli temporali al momento dell'esecuzione ed è eseguita la "distruzione" sia di CheckIn-Event che di RegisterEvent:

```

1 function bookingTimeout() public {
2     require(
3         owner == msg.sender,
4         'Questa funzione e\' abilitata solo per il proprietario del
5         contratto'
6     );
7     require (
8         now >= limitDate,
9         'Limite per effettuare il check-in non ancora raggiunto'
10    );
11    // Destroy Register
12    registerContract.bookingTimeout();
13    selfdestruct(owner);
14 }

```

4.1.2 RegisterEvent

Un contratto di tipo RegisterEvent è inserito sulla blockchain immediatamente dopo uno di tipo CheckInEvent, in quanto necessita, come uno parametri del costruttore, l'indirizzo del contratto CheckInEvent associato. Oltre quest'ultimo è necessario fornire anche il prezzo della prenotazione dell'evento:

```
1 constructor(uint256 eventBookingRate,address payable _checkInContract)
  public {
2     owner = msg.sender;
3     eventRate = eventBookingRate;
4     _checkIn = _checkInContract;
5     checkin = CheckInContract(_checkIn);
6     checkin.setRegister(address(this));
7 }
```

La prenotazione è effettuata inviando una transazione al metodo book(), **payable**, che effettuerà controlli sulla somma inviata di una prenotazione precedente e provvederà all'invio della somma appena ricevuta al contratto CheckInEvent:

```
1 function book() public payable{
2     // Checks if the amount of Ethereum is correct
3     require(
4         msg.value == eventRate,
5         'Quantita\' di ether errata per la prenotazione'
6     );
7     // Checks if the sender address is already booked up
8     if(checkAlreadyBooked(msg.sender)){
9         revert('E\' gia\' presente una prenotazione per questo wallet');
10    }
11    // Registers the new address
12    registerNewBooking(msg.sender);
13    // Transfers the Eth to the checkIn Contract
14    _checkIn.transfer(msg.value);
15 }
```

Il controllo della prenotazione è fatto utilizzando una map booleana avente come chiave gli indirizzi dei wallet dei partecipanti. Tale metodo permette inoltre che una prenotazione sia valida una sola volta per il check-in:

```
1 function checkBooking(address _booked) external returns (bool){
2     if(booked[_booked]==true){
3         booked[_booked] = false;
4         return true;
5     }
6     return false;
7 }
```

4.2 Webapp

Premessa

La webapp che consente di simulare il servizio offerto da Ethvent è da definirsi un prototipo. Sono presenti una serie di limitazioni da un punto di vista progettuale e per ciò che concerne la sicurezza di chiavi private e transazioni. È da considerarsi un puro esperimento e da utilizzare solo ed esclusivamente su una blockchain locale. Il suo scopo è solo di simulare il servizio.

4.2.1 Struttura della webapp

Il progetto segue una struttura molto simile a quella definita da NPM con delle leggere modifiche. I principali elementi sono i seguenti.

```
ethvent
├── build - contratti compilati, abi e bytecode per la EVM
├── contracts - sorgenti dei contratti in Solidity
├── public - File statici (.css, immagini) e un json che memorizza gli eventi
├── views - views dell'applicazione in ejs
├── package.json - informazioni generiche sul package
├── package-lock.json - dipendenze dei moduli
├── settings.js - alcuni parametri dell'applicazione
├── server.js routes manager e creazione dell'istanza del server
└── contract-manager.json modulo per la gestione dei contratti e degli eventi
```

4.2.2 Routes

Trattandosi di una webapp è necessario gestire quali saranno le response alle varie request HTTP che verranno fatte al server. Tali richieste si differenzieranno in base al metodo di richiesta (GET,POST) e alla risorsa richiesta. Trattandosi di un webserver è possibile esporre una serie di risorse definite come "route" (o endpoint), intercettare le richieste fatte ad esse e servirle in maniera opportuna. Ethvent espone i seguenti endpoint:

- POST / - Root della webapp, renderizza la view "index" mostrando tutti gli eventi disponibili.
- GET /createEvent - Renderizza semplicemente la view "createEvent".
- POST /createEvent - Riceve i valori dal form, crea l'evento effettuando il deploy dei contratti e salva le informazioni relative nel file events.json.
- GET /event - Viene passato un id come parametro, verrà renderizzata la pagina relativa all'evento selezionato, con tutti i suoi dettagli.
- GET /simulateTimeout - Simula il timeout per effettuare il check in. **NB: Per poter simulare facilmente il timeout dell'evento, al momento della creazione dei contratti la data massima passata come argomento è 0. In questo modo si può,**

in qualsiasi momento, simulare la scadenza, senza aspettare realmente la data indicata.

4.2.3 Templating delle pagine web

Avendo la necessità di dover generare pagine web in maniera dinamica, con informazioni che cambiano in base all'operato degli utenti, è necessario utilizzare un linguaggio di templating che consenta di aggiungere e aggiornare le informazioni presenti nelle pagine web prima di essere inviate al client. La gestione di queste informazioni dinamiche è affidata all'utilizzo di **EJS** (Embedded JavaScript templating). Tale linguaggio consente di modificare il contenuto di una pagina web con informazioni dinamiche attraverso l'ausilio di costrutti d'iterazione provenienti da Javascript. Un esempio lampante è presente proprio nella pagina ricevuta in seguito alla richiesta di root (/) della webapp: gli eventi mostrati in questa pagina sono aggiunti dinamicamente in EJS prima di inviare la pagina all'utente, questo è un semplice snippet di esempio semplificato presente in "views/index.ejs":

```
1      <% if(events.length == 0) { %>
2          <h2>Non ci sono eventi disponibili</h2>
3      <% } %>
4
5      <% for(var i=0; i < events.length ; i++) { %>
6          <div> events[i] </div>
7      <% } %>
```

Tra i tag "<%" e "%>" è possibile inserire codice javascript che verrà eseguito dal template engine prima di inviare la pagina web al client. In questo esempio specifico, passando a questa pagina l'insieme degli eventi è possibile verificare se ce ne sono (in caso negativo stamperà il messaggio "Non ci sono eventi disponibili") e quindi di inserirli dinamicamente nella pagina.

4.2.4 Transazioni dal browser

Per svincolare il server dalla responsabilità di effettuare le transazioni, dovendo gestire le chiavi private degli utenti, tutte le operazioni che non coinvolgono il proprietario del servizio sono effettuate dai client che utilizzano la webapp direttamente dal proprio browser. Utilizzando client wallet già esistenti, come Metamask è possibile - previo consenso - effettuare le transazioni (che nel nostro caso oltre che pagare il costo dovuto alla prenotazione, coinvolge anche il tentativo di fare il checkin) sfruttando la libreria web3, utilizzata anche lato backend per alcune interazioni con i contratti. Trattandosi di codice che deve essere eseguito lato client e utilizzando alcuni parametri che dipendono dalla configurazione dell'applicazione, lo stesso codice javascript subisce una leggera modifica dal template engine, in modo da ricevere informazioni come il provider della blockchain e gli indirizzi dei contratti stessi. Ciò è presente nella pagina richiesta all'endpoint "/event". Per poter effettuare delle transazioni utilizzando l'estensione Metamask (o simili) è necessario verificarne la presenza e verificare che l'utente abbia dato il consenso a tale estensione di accedere alle informazioni presenti nella pagina.

Solo dopo aver assicurato l'accesso al wallet alla pagina è possibile effettuare operazioni che coinvolgono l'account selezionato nell'estensione. In Ethvent queste operazioni sono necessarie solo nella pagina dell'evento. Quando la pagina è completamente renderizzata dal browser (e quindi quando è sollevato l'evento "load") è controllata la presenza e il consenso di un wallet, nel nostro caso Metamask. Questo è uno snippet esplicativo di ciò che accade:

```

1  window.addEventListener('load', async () => {
2      //Apri la il popup di Metamask
3      if (window.ethereum) {
4          window.web3 = new Web3('<%= config['provider'] %>');
5          try {
6              await ethereum.enable();
7              bookingHandler();
8              checkinHandler();
9          }catch (err) {
10             //gestione errori
11         }
12     }
13     //verifica la presenza di un provider (e' gia' stato dato il consenso)
14     else if (window.web3) {
15         window.web3 = new Web3(web3.currentProvider)
16     }
17     else {
18         // se viene eseguito questo blocco : o non e' stato dato il consenso o
19         // non e' installato Metamask
20     }
21 })

```

bookingHandler() e checkinHandler() sono le due function che gestiscono le transazioni (prenotazione e checkin). Sono delle funzioni associate ad un Event Listener (ovvero vengono eseguite solo quando si verifica un determinato evento, nel nostro caso il click sui bottoni). Richiamandole nello script precedente vengono attivate e rimangono in attesa dell'evento. In questo modo, se non c'è consenso o se non è installato Metamask non verranno generate inutili transazioni che fallirebbero in ogni caso (Non avendo un wallet da cui far partire la transazione).

Ad ogni transazione è associato una receipt, ovvero una ricevuta, che contiene eventuali messaggi di errore definiti nel contratto (ad esempio quelli dei "require"). È possibile accedere a tali messaggi e visualizzarli nella pagina web a runtime.

Capitolo 5

Manuale Utente

Il progetto dovrebbe essere compatibile con qualsiasi sistema Windows e Unix-like a patto che si possa utilizzare Node.

5.1 Requisiti

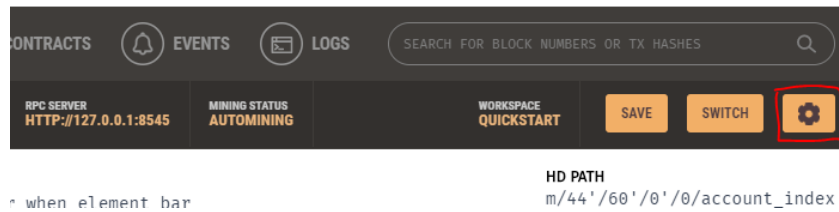
- Assicurarsi che il sistema sia connesso ad Internet. (È necessario per alcuni CDN presenti nel progetto, come Bootstrap, JQuery e FontAwesome).
- Installare **NodeJS** e il suo package manager **NPM** (<https://nodejs.org/en/>)
- Installare il client **Ganache** (<https://www.trufflesuite.com/ganache>). Tale software ci consente di avere una blockchain locale e di vederne blocchi e transazioni.
- Installare l'estensione **Metamask** (<https://metamask.io/>) per avere la possibilità di gestire wallet Ethereum dal browser.

5.2 Configurazione Ganache

Per poter avere una blockchain locale facendo uso di Ganache è necessario aprire il software appena installato. Fare click su "Quickstart":

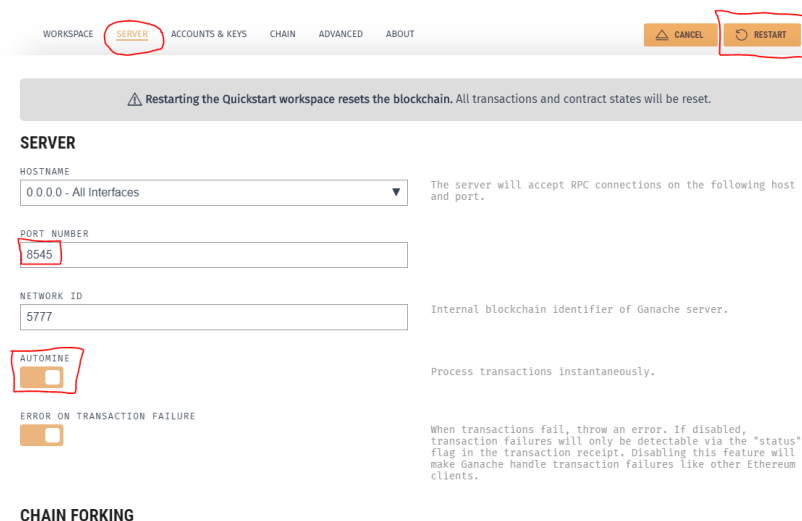


Verranno generati degli account a partire da una singola chiave privata. È necessario configurare il client di Ganache in modo tale da permettere a Metamask ed Ethvent di potervi accedere.



Andare nella sezione "Server" :

- Impostare come Port Number: 8545 (porta di default per la rete di testing locale di Metamask).
- Assicurarsi di avere la blockchain in Automine. (I blocchi verranno immediatamente minati, in questo modo possiamo liberatamente simulare il servizio).
- Cliccare su "Restart" per applicare le modifiche e resettare la blockchain.



Da questo momento in poi, avere Ganache in funzione significa avere una blockchain in locale in ascolto che risponde all'indirizzo 127.0.0.1:8545.

5.3 Installare e configurare Ethvent

- Per poter configurare e installare tutte le dipendenze di Ethvent è necessario aver a disposizione il suo codice sorgente. Qualora non si avesse a disposizione è sufficiente fare un clone della sua repository Github:

```
1 $ git clone https://github.com/AlessioMaddaluno/ethvent.git
```


- Per poter installare tutte le dipendenze è sufficiente da terminale entrare nella directory **ethvent** e installare i moduli dal file *package-lock.json* :

```
1 $ npm install
```

- Terminata l'installazione, è necessario aprire il file *settings.json*.
La porta dell'applicazione e l'url_provider della blockchain hanno già un valore di default che non è necessario modificare.
È importante però impostare la chiave privata del proprietario del servizio. Tale entità sarà anche il proprietario dei contratti relativi agli eventi.
È possibile utilizzare un qualsiasi account messo a disposizione da Ganache: è sufficiente cliccare sull'icona della chiave per l'account che intendiamo utilizzare e impostare quindi nel file json tale valore.

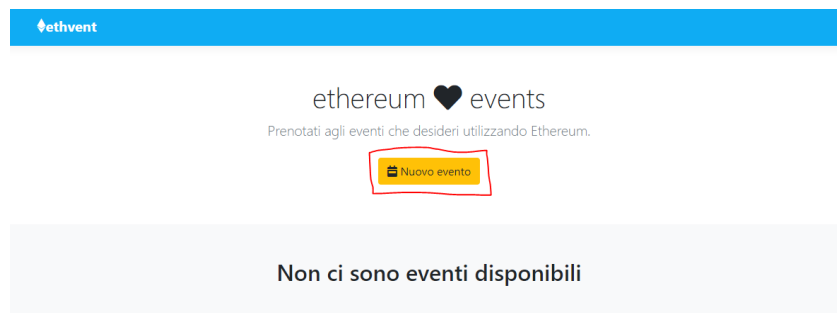
È possibile dunque lanciare l'app da terminale con

```
1 $ node server.js
```

A questo punto è possibile tramite il browser (con l'estensione Metamask installata), visitare l'indirizzo *localhost:3000* e interagire con l'app.

5.4 Creazione eventi

Inizialmente non sono presenti eventi. Per poterne creare uno è sufficiente cliccare su "Nuovo evento".



Compilare il form relativo per creare l'evento. Una volta creato è possibile verificare la sua creazione ritornando alla homepage, cliccando sul logo.

Crea un nuovo evento

Compila il form e crea un nuovo evento

Titolo Evento	<input type="text" value="HackForNeeds Hackaton 2020"/> <small>es. WCC 2020, Comicon 2020</small>
Data	<input type="text" value="13/02/2020"/>
Ora	<input type="text" value="10:15"/>
Luogo	<input type="text" value="Piazza Carlo Borbone 12, San Giorgio a Cremano (NA)"/>
Descrizione Evento	<input type="text" value="HackForNeeds è il primo hackaton organizzato dalla città di San Giorgio a Cremano. Lo scopo è quello di sensibilizzare riguardo il tema della disabilità."/>
Costo prenotazione	<input type="text" value="0.1"/> <small>In Ethereum</small>
<input type="button" value="Crea Evento"/>	

5.5 Gestione eventi e permessi Metamask

Per poter utilizzare Metamask sulla webapp è necessario innanzitutto importare gli account di Ganache e assicurarsi di utilizzare la stessa blockchain (nel nostro caso, locale). Fare click sull'estensione e selezionare "Import using account seed phrase". Come seed phrase utilizzare quella di Ganache sotto la voce "Mnemonic", in questo modo possiamo importare contemporaneamente tutti gli account. È importante prima di procedere di selezionare come rete di riferimento di Metamask, la rete locale (su cui lavora Ganache).

METAMASK Localhost 8545

< Back

Restore your Account with Seed Phrase

Enter your secret twelve word phrase here to restore your vault.

Wallet Seed


champion protect vote box caught cruel label shoe
near when element bar

New Password (min 8 chars)

Confirm Password

Restore

A questo punto dalla home della nostra webapp possiamo selezionare un evento e accedere alla sua pagina in dettaglio.

 Nuovo evento

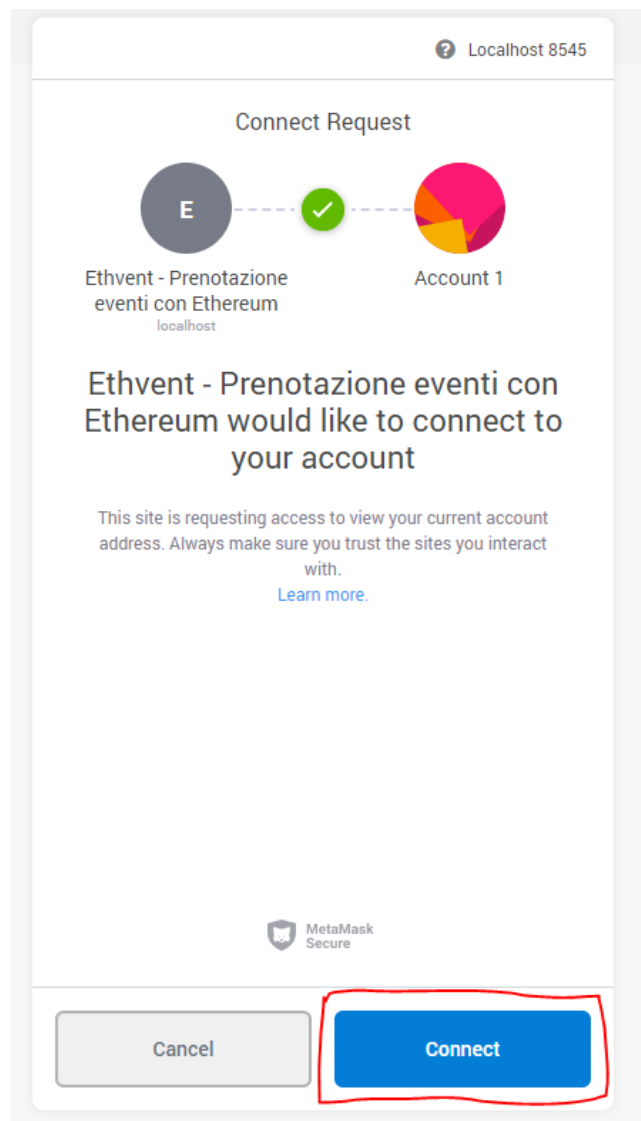
HackForNeeds Hackaton 2020

HackForNeeds è il primo hackaton organizzato dalla città di San Giorgio a Cremano. Lo scopo è quello di sensibilizzare riguardo il tema della disabilità.

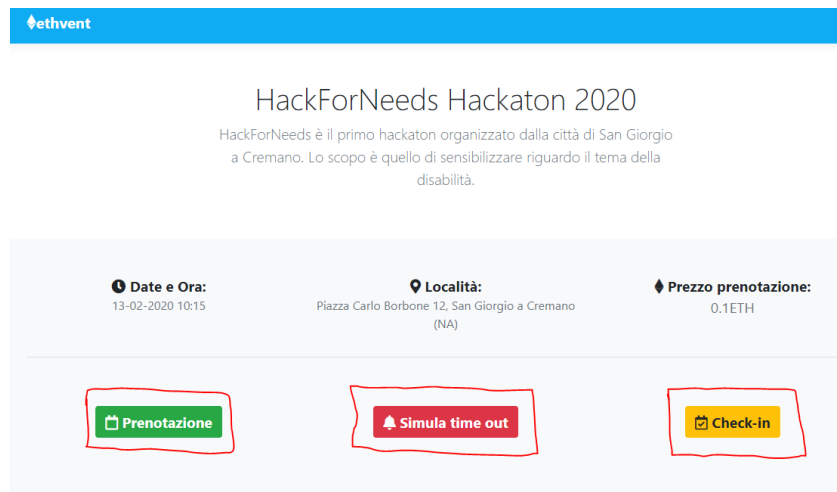
dove e quando: **Piazza Carlo Borbone 12, San Giorgio a Cremano (NA)**
13-02-2020 - 10:15

0.1


Prima di effettuare qualsiasi transazione è necessario dare i permessi a Metamask di accedere alla pagina.



A questo punto è possibile simulare tutte le azioni dei contratti come la prenotazione all'evento, il check-in e simulare la scadenza. È da tener presente che le azioni di prenotazione e di check-in sono fatte da Metamask e quindi riguardano l'account selezionato nell'estensione.



5.6 Attività sulla blockchain

Tutte le attività sulla blockchain come la creazione dei contratti e tutte le transazioni coinvolte sono visibili dal client di Ganache.

CURRENT BLOCK	GAS PRICE	GAS LIMIT	HARDFORK	NETWORK ID	RPC SERVER	Mining Status	WORKSPACE	QUICKSTART	SAVE	SWITCH	⚙
BLOCK 5	20000000000	6721975	PETERSBURG	5777	HTTP://127.0.0.1:8545	AUTOMINING					
BLOCK 5	MINED ON	2019-12-30 17:10:33				GAS USED 50460					1 TRANSACTION
BLOCK 4	MINED ON	2019-12-30 17:10:33				GAS USED 24137					1 TRANSACTION
BLOCK 3	MINED ON	2019-12-30 17:10:31				GAS USED 50460					1 TRANSACTION
BLOCK 2	MINED ON	2019-12-30 16:23:32				GAS USED 609544					1 TRANSACTION
BLOCK 1	MINED ON	2019-12-30 16:23:32				GAS USED 573838					1 TRANSACTION
BLOCK 0	MINED ON	2019-12-30 14:59:07				GAS USED 0					NO TRANSACTIONS

CURRENT BLOCK	GAS PRICE	GAS LIMIT	HARDFORK	NETWORK ID	RPC SERVER	Mining Status	WORKSPACE	QUICKSTART	SAVE	SWITCH	⚙
TX HASH	0xf6f839d24b17d9c1044ace75f0ef45ab9b642d443c6ea3761cf983f3a3a7ad										CONTRACT CALL
FROM ADDRESS	0x443E49c657c89C95123656522bab681883112615										
TO CONTRACT ADDRESS	0x6d868d4EC7e5EF57D23c51e055BAACe446ED8450										
GAS USED	50460										
VALUE	10000000000000000000										
TX HASH	0xf4f282e84c0f41deefa3e36cd525b7fe14a828c59e8445eb1fdf63462cabe046										CONTRACT CALL
FROM ADDRESS	0x443E49c657c89C95123656522bab681883112615										
TO CONTRACT ADDRESS	0x6d868d4EC7e5EF57D23c51e055BAACe446ED8450										
GAS USED	24137										
VALUE	0										
TX HASH	0x385c10bb3d98bdb83d0153c560cc544e09543db8ca4b23e02d9144a434dc133										CONTRACT CALL
FROM ADDRESS	0x443E49c657c89C95123656522bab681883112615										
TO CONTRACT ADDRESS	0x6d868d4EC7e5EF57D23c51e055BAACe446ED8450										
GAS USED	50460										
VALUE	10000000000000000000										
TX HASH	0xf31fe574892cc86f8d5492f542be1bafa3161a2745fa028c9635597eba1ae067										CONTRACT CREATION
FROM ADDRESS	0x443E49c657c89C95123656522bab681883112615										
CREATED CONTRACT ADDRESS	0x6d868d4EC7e5EF57D23c51e055BAACe446ED8450										
GAS USED	609544										
VALUE	0										
TX HASH	0x568edf0e08225ba865180cb9f54ab7737b1469e07f42ab42798358dc6f06dca										CONTRACT CREATION

Capitolo 6

Limitazioni e miglioramenti

Tale progetto è da considerarsi un prototipo. Non è stato realizzato nell'ottica di un caso reale, ma sfruttato da noi studenti come occasione per poter affrontare argomenti solitamente non trattati in maniera canonica durante i corsi del CdL (Blockchain, Ethereum, sviluppo web con NodeJS). Per quanto funzionante, non è esente da problematiche riguardo una scorretta progettazione di un servizio web e mancanza di strumenti che in altri contesti sarebbero vitali. In seguito verranno elencate una serie di elementi che possono essere liberamente sviluppati e aggiunti per poter migliorare il progetto:

- L'inserimento di un database e quindi di un DBMS per gestire accessi multi-thread. A tal proposito, lavorando su file che solitamente hanno estensione JSON l'ideale potrebbe essere MongoDB (<https://www.mongodb.com/>).
- Un refactoring del codice che sfrutti i punti di forza di NodeJS e che non blocchi il sistema con chiamate sincrone.
- L'utilizzo di design pattern orientati al web. Esporre quindi delle API per poter sviluppare facilmente un applicazione mobile che effettui lo stesso servizio.
- Uno script (magari in Bash) che consenta una configurazione veloce del progetto in fase di installazione
- Evitare l'uso di CDN (Content Delivery Network) per rendere il servizio indipendente dai provider di framework quali Bootstrap o FontAwesome
- Generalizzare la possibilità di effettuare transazioni, proponendo diverse alternative al solo utilizzo di estensioni come Metamask
- Sollevare l'utente dei costi relativi alle transazioni di Prenotazione e Check in, non restituibili, presenti in aggiunta al costo di prenotazione.