

Versioning basato su Blockchain

Davide Tarasconi

12 luglio 2019

1 Introduzione

Sviluppo di un sistema di versioning basato su *blockchain* in cui è possibile caricare un file e i suoi successivi aggiornamenti salvando l'hash e altre informazioni di base all'interno di una blockchain **Ethereum** avvalendosi di un *web server* sviluppato con **Node.js**.

2 Architettura

Il progetto è stato sviluppato utilizzando una architettura **client-server**. Per quanto riguarda il **Server** gli strumenti utilizzati sono stati:

- **Ethereum** (www.ethereum.org)
- **Ganache** (www.trufflesuite.com/ganache)
- **Truffle** (www.trufflesuite.com/truffle)
- **Node.js** (<https://nodejs.org/it/>)

Il **Client** è stato sviluppato utilizzando:

- **HTML**
- **CSS**
- **Javascript**

2.1 Ethereum

La **blockchain** consiste in un registro condiviso e distribuito replicato su tutti i node della rete. Permette di salvare i dati delle *transazioni* raggruppati in blocchi che formano una lista *immutabile* sempre in crescita. La blockchain è costituita da :

- **Full node:** nodi della rete che contengono l'intera blockchain e possono creare nuovi blocchi.
- **Consensus:** protocolli per determinare quale nodo potrà pubblicare il prossimo blocco.
- **Wallet:** software per effettuare transazioni.

Nel nostro caso abbiamo utilizzato la blockchain *pubblica* **Ethereum**, un tipo di blockchain basata sugli **Smart Contract**, programmi salvati all'interno della blockchain che esprimono una logica contrattuale e possono implementare qualsiasi algoritmo. Gli smart contract possono interagire tra loro scambiandosi dei messaggi e forniscono agli utenti un'interfaccia che permette di salvare dati oppure restituirli. Le *interazioni* col contratto sono salvate nella blockchain come delle *transazioni*.

Ethereum permette di sviluppare e distribuire questi contratti sulla blockchain. La valuta principale è l'*Ether*, dove $1 \text{ ETH} = 10^9 \text{ GWEI}$; mentre per effettuare le transazioni dei contratti viene utilizzato il *Gas* ($1 \text{ Gas} = 20 \text{ GWEI}$).

2.1.1 Truffle

Ambiente per sviluppare e testare smart contract usando la *Ethereum Virtual Machine* (EVM). I contratti sono sviluppati in **Solidity** e viene anche fornita la possibilità di rilasciare i contratti oltre che su EVM anche sulla blockchain Ethereum o su una sua copia locale.

2.1.2 Ganache

Software che fornisce una copia locale della blockchain Ethereum che può essere utilizzata per il rilascio di contratti, lo sviluppo di applicazioni o per effettuare dei test.

2.2 Node.js

Piattaforma per sviluppare facilmente applicazioni veloci e scalabili. Il modello *event-driven* su cui è basato la rende adatta allo sviluppo di applicazio-

ni in tempo reale. La gestione degli eventi viene effettuata utilizzando delle **callback**, funzioni passate come parametro ad altre funzioni (in questo caso i *listener* degli eventi) che descrivono il comportamento da seguire in presenza proprio di quell'evento. L' I/O è considerato non bloccante in quanto non è eseguito direttamente da quasi nessuna funzione, per cui il rischio di processi bloccati è molto basso. Basato su JavaScript.

2.3 HTML

Linguaggio di markup utilizzato per descrivere la struttura di un documento web.

2.4 CSS

Fogli di stile esterni a HTML ma che permettono di descrivere come deve esserne rappresentato il contenuto.

2.5 JavaScript

Linguaggio di programmazione Web, utilizzato per programmare il comportamento delle pagine. Fornisce le API per poter lavorare con stringhe, array, date ed espressioni regolari. Può essere inserito direttamente all'interno delle pagine HTML nei tag `<script></script>` oppure sempre utilizzando quei tag è possibile effettuare il link di un file esterno con estensione `.js`.

2.5.1 AJAX

Asynchronous JavaScript and XML, è una tecnica per creare pagine web dinamiche permettendone l'aggiornamento asincrono effettuando dei collegamenti in background col server per scambiare piccole quantità di dati. E' possibile utilizzarlo creando in JavaScript l'oggetto `XMLHttpRequest()`.

2.5.2 jQuery

Libreria che permette di facilitare l'uso di JavaScript raggruppando insieme di righe di codice in un unico metodo. Permette di manipolare documenti HTML e CSS, permette la gestione degli eventi HTML, la creazione di effetti e animazioni e consente di utilizzare AJAX.

3 Sviluppo

Per sviluppare questa **DApp** (Decentralized Application) prima di tutto sono stati scaricati Ganache e Truffle, che consentono di creare e compilare uno Smart Contract e poi effettuare il deployment sulla blockchain. Dopodiché sono state sviluppate la parte back-end utilizzando Node.js, per costruire un server web che fornisca le pagine richieste e che operasse e comunicasse con la blockchain; e la parte front-end che consiste in un insieme di pagine web che nel nostro caso consentono di poter caricare un documento, aggiornarlo caricando una nuova versione, fornire informazioni sulle versioni precedenti, fornire l'elenco di tutti i file caricati ed effettuare una piccola iscrizione per poter collegare un account blockchain ad un individuo e sapere chi sta eseguendo le operazioni appena elencate.

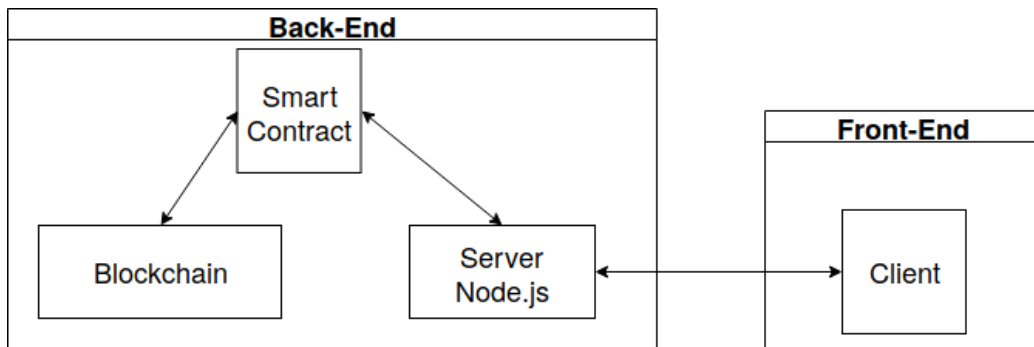


Figura 1: Architettura del sistema sviluppato.

3.1 Truffle e Ganache, sviluppo Smart Contract

3.1.1 Ganache

Dopo aver scaricato e installato Ganache lo avviamo, la schermata iniziale permette di creare un workspace, ovvero permette di creare una copia della blockchain con le caratteristiche scelte da noi durante la configurazione. Nel nostro caso abbiamo mantenuto le caratteristiche standard impostando come porta di comunicazione la 7575.

The screenshot shows the 'ACCOUNTS & KEYS' tab in the Ganache application. The window has a dark header with the title 'Ganache' and standard window controls. Below the header is a navigation bar with tabs: 'WORKSPACE', 'SERVER', 'ACCOUNTS & KEYS' (which is active and highlighted in orange), 'CHAIN', 'ADVANCED', and 'ABOUT'. To the right of the navigation bar are two buttons: 'CANCEL' and 'SAVE WORKSPACE'. The main content area is titled 'ACCOUNTS & KEYS' and contains four configuration sections, each with a label, a form field, and a description:

- ACCOUNT DEFAULT BALANCE:** A text input field containing '100'. Description: 'The starting balance for accounts, in Ether.'
- TOTAL ACCOUNTS TO GENERATE:** A text input field containing '10'. Description: 'Total number of Accounts to create and pre-fund.'
- AUTOGENERATE HD MNEMONIC:** A checkbox that is currently checked. Description: 'Turn on to automatically generate a new mnemonic and account addresses on each run.'
- Mnemonic:** A text input field containing the mnemonic 'nuclear legal toss arrange enjoy mountain friend puzzle phone fine sudden daught'. Description: 'Enter the Mnemonic you wish to use.'
- LOCK ACCOUNTS:** A checkbox that is currently unchecked. Description: 'If enabled, accounts will be locked on startup.'

Figura 2: Configurazione account.

The screenshot shows the 'CHAIN' tab in the Ganache application. The window has a dark header with the title 'Ganache' and standard window controls. Below the header is a navigation bar with tabs: 'WORKSPACE', 'SERVER', 'ACCOUNTS & KEYS', 'CHAIN' (which is active and highlighted in orange), 'ADVANCED', and 'ABOUT'. To the right of the navigation bar are two buttons: 'CANCEL' and 'SAVE WORKSPACE'. The main content area is titled 'CHAIN' and contains three configuration sections, each with a label, a form field, and a description:

- GAS LIMIT:** A text input field containing '6721975'. Description: 'Maximum amount of gas available to each block and transaction. Leave blank for default.'
- GAS PRICE:** A text input field containing '20000000000'. Description: 'The price of each unit of gas, in WEI. Leave blank for default.'
- HARDFORK:** A dropdown menu with 'Petersburg' selected. Description: 'The hardfork to use. Default is Petersburg.'

Figura 3: Configurazione GAS

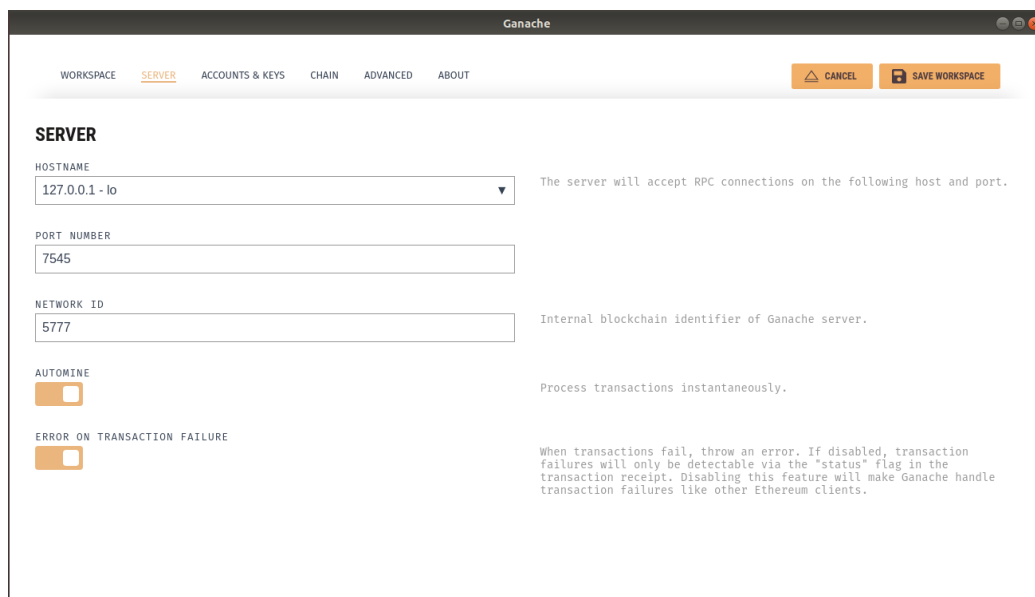


Figura 4: Configurazione informazioni server come porta e indirizzo.

3.1.2 Truffle

Truffle si scarica da riga di comando con il comando:

```
npm install truffle -g
```

Una volta installato ci spostiamo in una cartella ed utilizziamo (sempre da shell) il comando `truffle init`. Questo comando crea 3 cartelle:

1. **contracts:** cartella contenente lo smart contract sviluppato in solidity.
2. **migration:** contiene i file per effettuare il deployment del contratto.
3. **test:** contiene i file per valutare il corretto funzionamento del contratto.

Dopo aver sviluppato il nostro contratto possiamo compilarlo con il comando `truffle compile`. Successivamente possiamo creare dei file di test in solidity che ci consentono di valutarne il corretto funzionamento, questi file devono essere salvati nella cartella *test* e per eseguirli si utilizza il comando `truffle test`.

Infine, dopo aver sviluppato e testato il contratto, dobbiamo effettuare il deployment sulla blockchain di Ganache. Per farlo dobbiamo creare un file `2_deploy_contracts.js` nella cartella *migrations* e andare a sostituire il contenuto del file `truffle-config.js` con il seguente codice:

```
module.exports = {
  networks: {
    development: {
      host: "127.0.0.1",
      port: 7575,
      network_id: "*",
      gas: 5500000
    }
  }
};
```

Quando abbiamo sostituito il codice andiamo ad eseguire il comando **truffle migrate**. Dopo averlo eseguito notiamo che nella blockchain sono presenti delle transazioni. Solitamente la **terza** transazione è quella che fornisce l'indirizzo dello smart contract.

2_deploy_contracts.js:

```
var Versioning = artifacts.require("Versioning");

module.exports = function(deployer) {
  deployer.deploy(Versioning);
};
```

ACCOUNTS

BLOCKS

TRANSACTIONS

CONTRACTS

EVENTS

LOGS

SEARCH FOR BLOCK NUMBERS OR TX HASHES

CURRENT BLOCK

4

GAS PRICE

20000000000

GAS LIMIT

6721975

HARDFOK

PETERSBURG

NETWORK ID

5777

RPC SERVER

HTTP://127.0.0.1:7590

MINING STATUS

AUTOMINING

WORKSPACE

EXAMPLE

SWITCH

TX HASH

0x5173f28e19f70782de1d4168cef14834f9773ad3b675a3f53106d718ea2fb4a3

FROM ADDRESS

0x0bfc6959f80bfbeFBA3821a881292E9A13A6E84E

TO CONTRACT ADDRESS

0xB311aeD8339D5Dd0648BC65157d80D1AE7568314

GAS USED

27034

VALUE

0

CONTRACT CALL

TX HASH

0x7edc2ccb948cee0d0a37dc8f568399bbe7442a6bd2f9cf92ad31d6a476504c

FROM ADDRESS

0x0bfc6959f80bfbeFBA3821a881292E9A13A6E84E

CREATED CONTRACT ADDRESS

0x4CF5B0FE6db4eFfc2b1510CF48Ae6968fa8ee86

GAS USED

1156198

VALUE

0

CONTRACT CREATION

TX HASH

0x38339517bf1005df8748e97358115e84531762d2331db37bfdaaedecd28d59b6

FROM ADDRESS

0x0bfc6959f80bfbeFBA3821a881292E9A13A6E84E

TO CONTRACT ADDRESS

0xB311aeD8339D5Dd0648BC65157d80D1AE7568314

GAS USED

42034

VALUE

0

CONTRACT CALL

TX HASH

0xc7b5b399b4b75049c82d5b618200c40f16268361c2464242adc341ba302c319

FROM ADDRESS

0x0bfc6959f80bfbeFBA3821a881292E9A13A6E84E

CREATED CONTRACT ADDRESS

0xB311aeD8339D5Dd0648BC65157d80D1AE7568314

GAS USED

284908

VALUE

0

CONTRACT CREATION

Figura 5: La transazione segnata in *rosso* è quella del contratto mentre l'indirizzo segnato in *blu* è l'indirizzo del contratto.

3.1.3 Smart Contract

Come già accennato in precedenza i contratti sono sviluppati in **Solidity**, un linguaggio di alto livello e orientato agli oggetti progettato per EVM.

Per mantenere le informazioni riguardanti i documenti e le loro versioni è stata utilizzata una mappa (ID, Document[]) dove il secondo parametro è un'array di **Document**, una struttura creata per contenere le informazioni dei documenti come:

- indirizzo di chi crea o modifica il file
- hash del file (salvare tutto il file costerebbe troppi Gas)
- data di upload del file o della nuova versione
- numero di versione

```
//information about documents
struct Document{
    address creator;    //creator or modifier
    string value;       //document hash
    uint creation;      //date of creation
    uint8 version;      //document version
}

mapping (uint256 => Document[]) public doc;    //list of
the documents and relative "branch"
```

I metodi, che in questo caso vengono eseguiti come transazioni, consentono di creare nuovi documenti, aggiornarli e richiedere le informazioni di determinate versioni (o il numero totale delle versioni presenti).

- **create** : in base all'id fornito creiamo quell'elemento nella mappa aggiungendo all'array di documenti con il metodo **push()** la prima versione del documento. Il risultato del metodo è un evento che può essere gestito da codice JavaScript nella parte back-end e sia il metodo che l'evento restituiranno l'id, il numero di versione e l'indirizzo del creatore. Se il documento fornito è vuoto la transazione non viene eseguita.

```
function create(uint256 id, string memory docv)
public returns (uint256 docId, uint256 ver,
address aCreator){
    require(bytes(docv).length != 0, "No document");
    //If document is empty don't execute
    //require(doc[id][0].value != 0, "Document
    already exist"); //wrong id

    //Create new document
    doc[id].push(Document({
        creator: msg.sender,
        value: docv,
        creation: now,
        version: 0
    }));

    emit CreateDocument(id, 0, msg.sender);

    return (id, 0, msg.sender);
}
```

- **update** : permette di aggiungere una nuova versione dell'id fornito, inserendo un nuovo elemento nell'array. L'evento emesso fornisce l'indirizzo di chi ha aggiornato, il valore del nuovo documento e il nuovo numero di versione. Il metodo viene eseguito solo se il documento fornito non è vuoto e se il nuovo documento non è uguale a quello della versione precedente.

```
function update(uint256 id, uint256 ver, string
memory docv) public returns (uint256 newVer){
    require(keccak256(abi.encodePacked(doc[id][ver].
        value)) != keccak256(abi.encodePacked(docv)),
        "Same file");    //Check if new file is equal
        to old file
    require(bytes(doc[id][ver].value).length > 0, "
        Document doesn't exist");    //Check if selected
        document exist

    //add new version
    doc[id].push(Document({
        creator: msg.sender,
        value: docv,
        creation: now,
        version: doc[id][ver].version+1
    }));

    emit ChangeDocument(doc[id][doc[id][ver].version
        +1].creator, docv, doc[id][ver].version+1);

    return doc[id][ver].version+1;
}
```

- **get** : fornisce creatore, data di aggiornamento e valore della versione del documento fornita in input.

```
function get(uint256 id, uint256 ver) public view
returns (address creator, string memory value,
uint creation, uint8
version){
    Document memory d = doc[id][ver];
    return (d.creator, d.value, d.creation, d.version
        );
}
```

- `getNumVer` : fornisce il numero di versioni totali del documento richiesto.

```
function getNumVer(uint256 id) public view returns (
    uint256 num){
    return doc[id].length;
}
```

3.2 Back-end

La parte back-end è stata sviluppata creando un web server con Node.js. Per farlo sono stati utilizzati i seguenti moduli:

- `fs` : modulo per poter leggere nel file system
- `express` : modulo che permette di creare un web server con più funzionalità rispetto al modulo `http`, nel nostro caso possiamo leggere il contenuto dei form delle pagine web e possiamo salvare anche dei file.
- `body-parser` : insieme ad `express` permette di leggere il contenuto dei form.
- `cookie-parser` : permette di gestire e creare i cookie sui client.
- `multer` : permette di caricare i file dai form e salvarli sul web server.
- `web3` : modulo fondamentale che si occupa del collegamento con la blockchain e permette di eseguire i metodi dello smart contract.

3.2.1 Creazione Web Server

Per poter creare il web server dobbiamo prima di tutto creare un oggetto `express()` (nel nostro caso la costante `app`). Prima di tutto utilizzando il metodo `use()` definiamo le *funzioni middleware* che deve avere il nostro server. Le nostre funzioni sono: la codifica dei dati da leggere dai *form* sfruttando il modulo `body-parser`, la gestione dei cookie per avere delle sessioni di login, fornire automaticamente al client i fogli di stile CSS e di script JS.

```
//Parser for reading form data
app.use(parser.urlencoded({ extended: true }));

//send automatically style and script files
app.use('/CSS',express.static('../client/CSS'));
app.use('/JS',express.static('../client/JS'));
```

Listing 1: Esempio Funzioni Middleware

Dopo aver impostato le funzioni dobbiamo gestire le richieste GET e POST che potrebbero arrivare, utilizzando i metodi `get()` e `post()`. In questo progetto quasi tutte le richieste GET sono richieste di pagine web, mentre le richieste POST solitamente forniscono dati da elaborare che sono stati letti da un form.

I metodi `get()` e `post()` richiedono in input il path che indica quale funzione middleware invocare e una callback che permette di elaborare i parametri della richiesta (`req`), e fornire risposte al richiedente utilizzando il valore di risposta (`res`) fornito proprio dalla callback.

```
//login request
app.get("/login.html" || "/login", function(req, res){
    sendFile(res, '../client/HTML/login.html', 'text/html');
});
```

Listing 2: Esempio GET

```
//data from subscribe
app.post('/subscribe', (req, res) => {
    console.log("Get POST SUB request");
    //add user to user.json document
    readUser.user.push(new User((readUser.user.length),req.
        body.first, req.body.last, req.body.pwd, req.body.wal,
        req.body.usr));
    res.redirect("/login.html");
});
```

Listing 3: Esempio POST

Quando abbiamo gestito tutte le richieste è possibile avviare il server su una porta scelta da noi utilizzando il metodo `listen()`.

```
app.listen(8000, function(){
    setContract(); //create the contract
    readStartingData();
    console.log("Server running at http://127.0.0.1:8000/\n")
    ;
});
```

Listing 4: Avvio server web

3.2.2 Collegamento con blockchain e creazione contratto

La parte fondamentale del progetto. Prevede il collegamento del server con la blockchain e la creazione del contratto che permette di effettuare le transazioni usando i metodi creati nello smart contract.

Prima di tutto creiamo l'oggetto `web3()` passandogli come parametro l'indirizzo e la porta della blockchain (nel nostro caso ciò che abbiamo impostato su Ganache).

```
const web3js = new web3(new web3.providers.HttpProvider("http://127.0.0.1:7575"));
```

Per creare il contratto (l'oggetto `web3js.eth.Contract()`) dobbiamo conoscere il suo indirizzo nella blockchain (trovato precedentemente durante la migrazione) e l'**ABI**. ABI non è altro che la descrizione *json* del contratto che viene creata nel file `Versioning.json` durante la compilazione. Permette la creazione di un oggetto adattato al modello del contratto creato, ovvero il contratto JavaScript fornisce gli stessi metodi del contratto Solidity. Per semplificare il tutto è stata creata una funzione `setContract()` che inizializza il contratto e viene richiamata nella callback di creazione del server.

```
function setContract(){
    contractABI = [ /* ABI contract */ ];

    contractAddress = '0
        x585a79b73b9644546675401Ef44a45Bfa05dB268';

    contract = new web3js.eth.Contract(contractABI,
        contractAddress);
}
```

3.2.3 Gestione utenti e documenti

Per gestire documenti e utenti sono stati creati 3 tipi di oggetti:

- `User()`: traccia gli utenti iscritti, ne salva nome, cognome, username, password e indirizzo dell'account Ethereum. Durante la creazione del server viene letto il file `user.json` contenente i dati degli utenti già iscritti. Questo file viene salvato periodicamente.
- `Document()`: fornisce le informazioni di tutti i documenti già esistenti leggendo il file `documents.json` al lancio del server e ne permette l'aggiunta di nuovi salvando le informazioni periodicamente. Salva id del documento, l'utente che lo aggiunge, la versione attuale, il percorso di salvataggio del file, data di creazione e una piccola descrizione.

- `BlockDoc()`: permette di salvare tutte le informazioni presenti nell'oggetto `Document()` con l'aggiunta del valore di `hash`. Questa classe è stata creata per salvare le informazioni fornite dalla blockchain interagendo con lo smart contract.

3.2.4 Nuovo documento

Quando viene ricevuta la richiesta di salvataggio di un nuovo documento andiamo a leggere i dati forniti dal form nella parte front-end. Tra quei dati è presente anche il documento che salviamo in una cartella definita durante la creazione del server, e che utilizziamo per calcolare l'hash che salviamo nella blockchain. Per effettuare la codifica utilizziamo il modulo `crypto` che richiameremo tutte le volte per creare un seme diverso. La codifica usata è `sha256`.

```
//hash of file
//We create the seed every time to avoid recreation error
var hash = require('crypto').createHash('sha256').update(
    file.path).digest('hex');
```

Dopo aver effettuato questa elaborazione aggiungiamo il file ad un array della classe `Document()` che, come già spiegato, contiene tutti file caricati. Come ultima cosa utilizziamo il contratto per effettuare una transazione e aggiungere i dati del documento nella blockchain. Per farlo richiamiamo il metodo `create()` a cui passiamo come parametri l'ID del documento e il suo hash. Se il documento non è valido la transazione non viene eseguita e viene fornito un messaggio di errore.

```
contract.methods.create(id,hash).send({from: req.cookies.
    wall, gas: 4712388, gasPrice: 100000000000})
.on('confirmation', (confirmationNumber, receipt) => {
    //save data and return to home page
    var dataD = JSON.stringify(listDoc);
    fs.writeFileSync('data/documents.json', dataD);
    res.redirect("/");
})
.on('error', ()=>{
    res.redirect("/error.html");
});
```

La quantità di GAS indicata ed il suo prezzo sono quelle standard mentre l'indirizzo dell'account Ethereum di chi effettua la transazione è letto dai cookie del client richiedente.

3.2.5 Aggiornamento documento

Molto simile al caricamento di un nuovo file con la differenza che il metodo del contratto usato è `update()`, a cui passiamo ID, versione attuale e hash. Nel caso in cui la transazione vada a buon fine viene emesso un evento dalla blockchain che ci fornisce i valori di ritorno del metodo. Usiamo questi valori per aggiornare la lista di documenti salvati.

```
//update document's info and save into blockchain
contract.methods.update(id,version,hash).send({from: req.
  cookies.wall, gas: 4712388, gasPrice: 100000000000})
.on('confirmation', (confirmationNumber, receipt) => {

  console.log(receipt.events.ChangeDocument.returnValues);

  listDoc.doc[id].version = receipt.events.ChangeDocument.
    returnValues.version.toNumber();
  listDoc.doc[id].description = req.body.desc;
  //listDoc.doc[id].description = req.body.desc;
  var dataD = JSON.stringify(listDoc);
  fs.writeFileSync('data/documents.json', dataD);
  res.redirect("/");
}).on('error', ()=>{
  res.redirect("/updError");
});
```

3.2.6 Gestione richieste AJAX e jQuery

Vengono effettuate una richiesta AJAX e una jQuery.

La richiesta AJAX è una richiesta *get* effettuata dalla pagina principale del client per richiedere la lista di documenti presenti.

```
app.get("/list", function(req, res){
  console.log("getListDoc");
  readStartingData();
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.write(JSON.stringify(listDoc));
  res.end();
});
```

La richiesta jQuery è una richiesta *post* effettuata dal client per richiedere tutte le versioni di un determinato documento. In questo caso interagiamo con la blockchain utilizzando il metodo `get()` del contratto, che fornisce tutte le informazioni di un documento. Queste informazioni vengono salvate in un oggetto `BlockDoc()`. In questo caso la transazione viene eseguita come `call()` invece che `send()` in quanto la `call()` non apporta modifiche

alla EVM e quindi esegue transazioni che non spendono GAS e non sono salvate nella blockchain, mentre le transazioni `send()` vengono salvate e viene utilizzato GAS. E' stata utilizzata la `call()` perché `get()` è utilizzata solo a scopo informativo.

```
app.post('/version', (req, res) => {
  console.log("Get POST request");

  //parse id and version to int
  var id = parseInt(req.body.ID, 10);
  var version = parseInt(req.body.ver);

  //use call because this transaction it's informative and it
  //hasn't to use currency
  contract.methods.get(id, version).call({from: req.body.wall,
    gas: 4712388, gasPrice: 1000000000000})
    .then((result) => {
      res.send(JSON.stringify(new BlockDoc(id, result.creator,
        listDoc.doc[id].path, new Date(result.creation.
          toNumber()*1000).toUTCString(), result.version, "",
          result.value )));
    })
    .catch(err => {
      res.redirect("/error.html");
    });
});
```


3.3 Front-end

Per la parte front-end sono state sviluppate delle pagine web che permettono di registrarsi al servizio, effettuare login, visualizzare i documenti presenti, le loro informazioni e caricarne di nuovi.

3.3.1 Registrazione e login

La pagina web della *registrazione* è composta da un form che deve essere compilato con le informazioni personali dell'utente e l'indirizzo del suo account Ethereum.

Il *login* viene effettuato inserendo username e password, il server dopo aver verificato l'esistenza dell'utente impostano i cookie che verranno letti dalle altre pagine web per permettere all'utente determinate operazioni ed effettuare modifiche alla struttura della pagina se l'utente è già connesso. il *logout* è effettuato utilizzando uno script JavaScript che imposta la "data di scadenza" dei cookie precedente a quella attuale.

```
function logout(){
  document.cookie = "ID= ; expires = Thu, 01 Jan 1970
    00:00:00 GMT";
  document.cookie = "wall= ; expires = Thu, 01 Jan 1970
    00:00:00 GMT";
  document.cookie = "name= ; expires = Thu, 01 Jan 1970
    00:00:00 GMT";
}
```

Listing 5: Logout

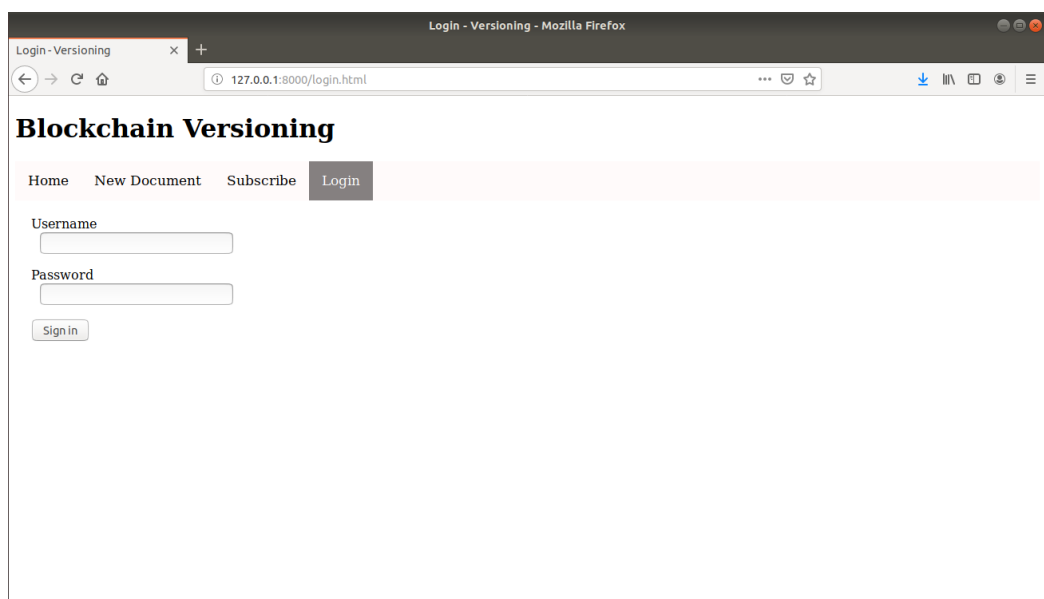


Figura 6: Pagina di Login.

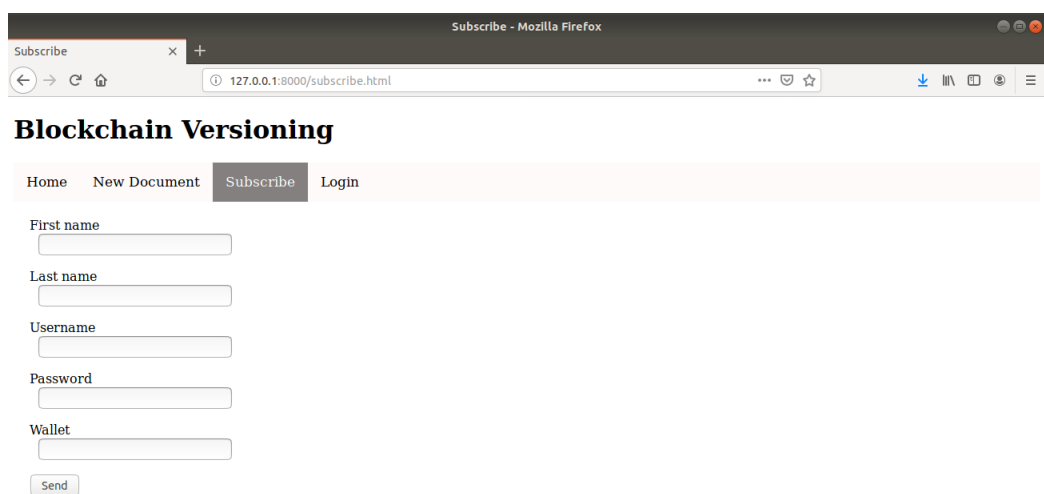
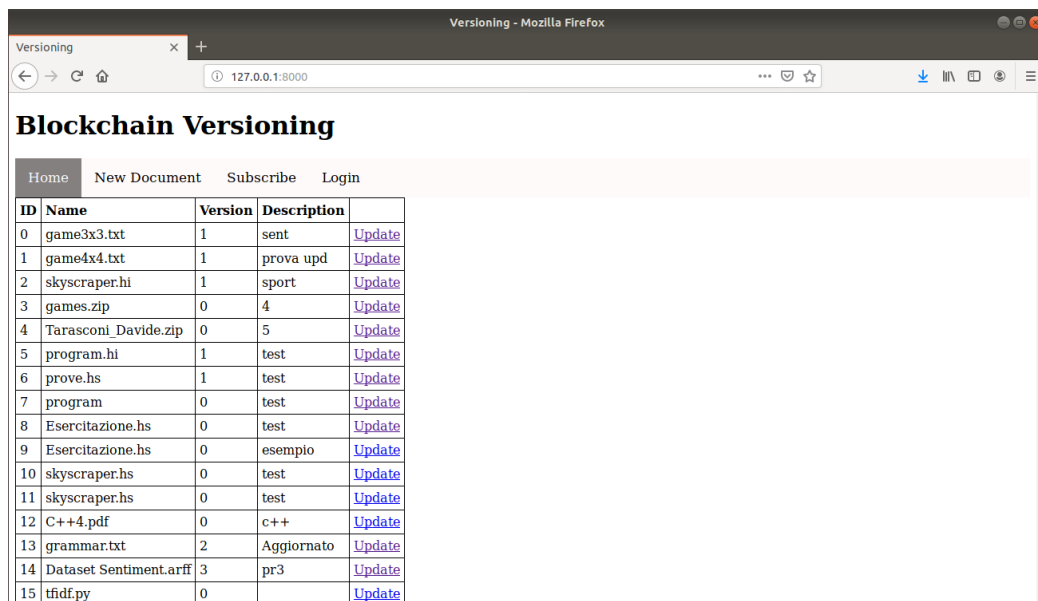


Figura 7: Pagina di iscrizione.

3.3.2 Home

Pagina principale in cui è presente una tabella contenente la lista di documenti presenti indicandone nome, descrizione e versione. Da qui è possibile selezionare un documento da aggiornare con una nuova versione, oppure è possibile selezionare un'altra area del servizio tra quelle del menù situato in ogni pagina. Viene effettuata una richiesta AJAX per richiedere la lista di documenti presenti.



ID	Name	Version	Description	
0	game3x3.txt	1	sent	Update
1	game4x4.txt	1	prova upd	Update
2	skyscraper.hi	1	sport	Update
3	games.zip	0	4	Update
4	Tarasconi_Davide.zip	0	5	Update
5	program.hi	1	test	Update
6	prove.hs	1	test	Update
7	program	0	test	Update
8	Esercitazione.hs	0	test	Update
9	Esercitazione.hs	0	esempio	Update
10	skyscraper.hs	0	test	Update
11	skyscraper.hs	0	test	Update
12	C++4.pdf	0	c++	Update
13	grammar.txt	2	Aggiornato	Update
14	Dataset Sentiment.arff	3	pr3	Update
15	tfidf.py	0		Update

Figura 8: Pagina principale.

3.3.3 Nuovo documento

Contiene un form in cui è possibile selezionare un nuovo documento e fornirne una breve descrizione. E' possibile eseguire l'operazione soltanto se si è loggati al sistema e se il documento fornito è valido. Per controllare se il login è stato effettuato viene eseguito un codice JavaScript durante la creazione della pagina che verifica l'esistenza di cookie. Una volta inserito il documento si torna alla pagina principale dove sarà possibile notare la nuova voce nella lista.

```
function checkLogged(){
    var menu = document.getElementById("menu");

    if(getCookie("ID") != ""){
        menu.innerHTML += '<a href="/" onclick="logout()">Logout
                           </a>';
    }
    else{
        var logi = document.createElement("A");
        logi.appendChild(document.createTextNode("Login"));
        logi.setAttribute("href", "login.html");

        var subs = document.createElement("A");
        subs.appendChild(document.createTextNode("Subscribe"));
        subs.setAttribute("href", "subscribe.html");

        menu.appendChild(subs);
        menu.appendChild(logi);
    }
}
```

Listing 6: Controllo login

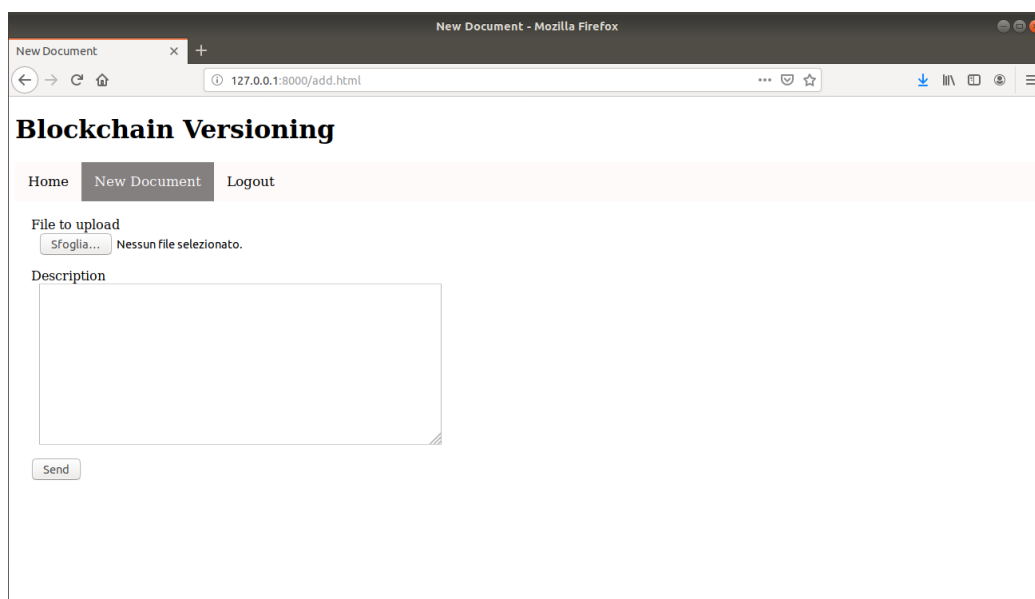


Figura 9: Pagina che permette di inserire un nuovo documento.

3.3.4 Nuova versione

La pagina che permette di aggiornare la versione del documento fornisce le informazioni della versione attuale e dà la possibilità di selezionare un nuovo file e di scrivere una nuova descrizione. Il nuovo file e la sua descrizione diventeranno la nuova versione attuale solo se: la nuova versione è differente dalla precedente (hash diverso), se il nome del documento è uguale e se l'utente è collegato. In fondo alla pagina è presente una lista contenente tutte le versioni precedenti all'attuale, una volta selezionata una voce verranno mostrate le informazioni di quella versione come numero di versione, creatore, valore e data di creazione.

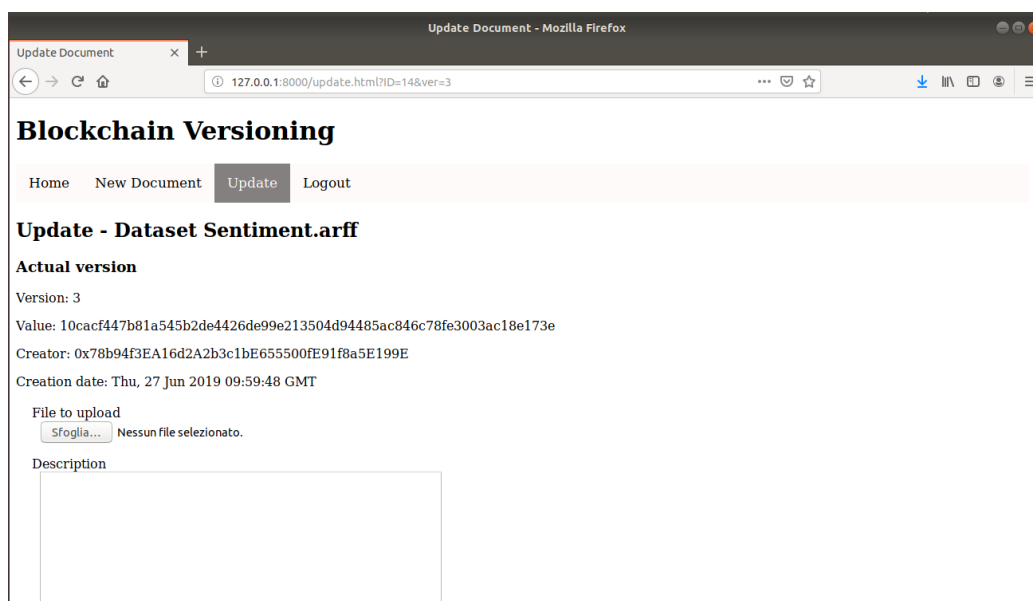


Figura 10: Pagina che permette di aggiornare un documento esistente.

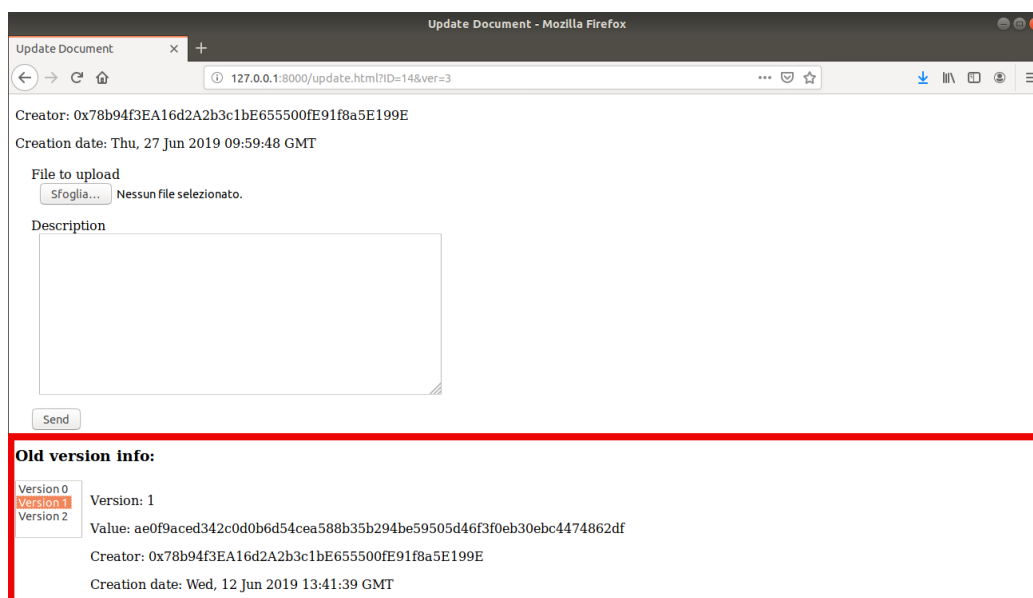


Figura 11: La sezione segnata in *rosso* è quella che fornisce le informazioni di tutte le altre versioni precedenti di quel documento

4 Funzionamento

Dopo aver effettuato il set up di Ganache e avviato il web server (comando `node server.js`) dal browser ci colleghiamo all'indirizzo `http://127.0.0.1/`. Dalla schermata di home selezioniamo la voce *Subscribe*. Nel form di iscrizione inseriamo i dati richiesti e scegliamo un account Ethereum tra quelli forniti da Ganache.

Dopo aver effettuato l'iscrizione selezioniamo la pagina *Login* ed effettuiamo il login.

Se sono presenti documenti, dalla pagina principale possiamo aggiornarli selezionando la voce **Update** dalla tabella, altrimenti andiamo sulla pagina *New Document* e ne carichiamo uno nuovo fornendo il documento e una sua breve descrizione (opzionale).

Per visualizzare le informazioni della versione attuale e di quelle precedenti di un documento selezioniamo la voce **Update** dalla pagina *Home* come se volessimo effettuare un aggiornamento.