



PRENOTA LA TUA POLTRONA A TEATRO !

La nostra prima dapp.

Progetto Sistemi peer-to-peer

2018-2019

Informatica per il Management

Università di Bologna

Prof. Stefano Ferretti

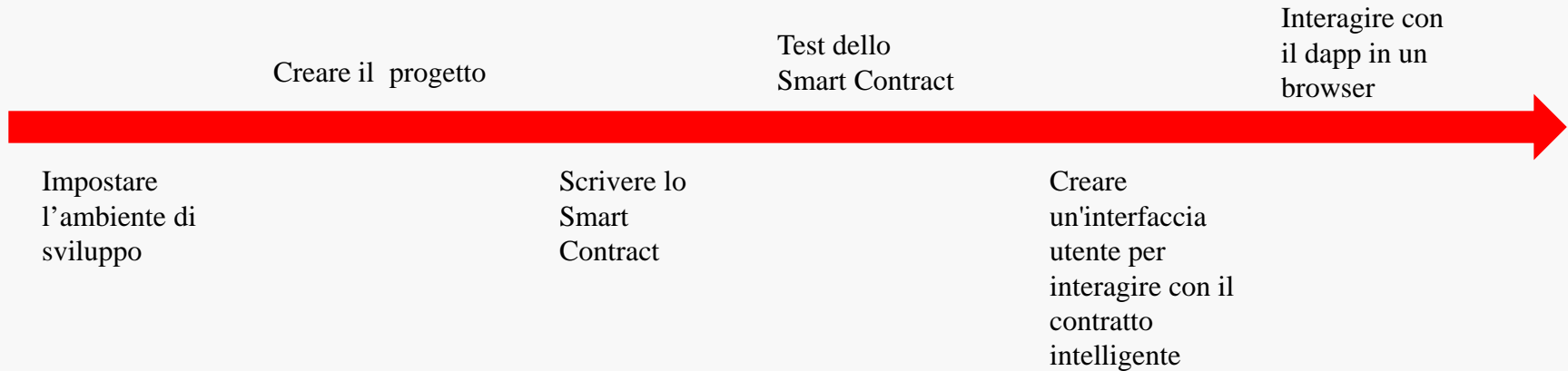
Di Marino Grezia 854009

Capuano Margherita 849862

1. INDICE

Queste slide mostrano il processo di creazione della nostra prima dapp, un sistema per la prenotazione di una poltrona presso il Teatro Comunale di Bologna.

Di seguito ci occuperemo di:



2. PRESENTAZIONE

Per gestire le prenotazioni delle poltrone del Teatro Comunale di Bologna utilizziamo la piattaforma Ethereum.

Ipotizziamo, per semplicità, che il teatro abbia a disposizione, in un dato momento, 16 poltrone prenotabili in tal modo e abbia già un database di poltrone.

Il nostro obiettivo è :

Vedere un utente che associa un indirizzo di Ethereum a una poltrona da prenotare.

Fondamentalmente progettiamo un software attraverso un contratto intelligente nella blockchain di Ethereum, la quale è avviata automaticamente utilizzando Ganache. Per eseguire direttamente la dapp nel browser facciamo uso di MetaMask.

➤ Ricordiamo che...

Etherum è una piattaforma decentralizzata del Web 3.0 per la creazione e gestione di Smart Contract. L'unità di conto è Ether.

Truffle Box, “scatola di tartufo”, contiene modelli utili, contratti e librerie Solidity, viste front-end..tutto ciò che rende il dapp unico.

Solidity è linguaggio di scrittura di programmazione orientato agli oggetti per la scrittura di smart contract.

Ganache permette di avviare automaticamente una blockchain di Ethereum

Dapp, è un acronimo per indicare una applicazione decentralizzata, ovvero un software creato attraverso i contratti intelligenti nella blockchain di Ethereum ,utilizzando il linguaggio Solidity .

MetaMask è un bridge che consente di eseguire le dapp di Ethereum direttamente nel browser

3. IMPOSTAZIONE DELL'AMBIENTE DI SVILUPPO

Abbiamo installato

1. Node.js v6 + lts e npm : Node.js® è un runtime JavaScript creato con il motore JavaScript V8 di Chrome.
2. Git : un sistema di controllo delle versioni distribuite, gratuito e open source, progettato per gestire tutto, dai progetti piccoli a quelli molto grandi.
3. Truffle : installato da terminale con il seguente comando, “npm install -g truffle”.

4. CREAZIONE DEL PROGETTO

Per creare il nostro progetto innanzitutto abbiamo :

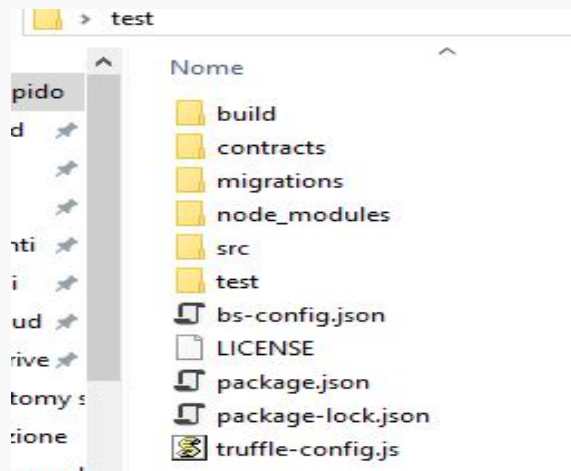
Creata una directory chiamata **test**, che include la struttura di base del progetto e il codice per l'interfaccia utente.

```
C:\Users\571g-719q>cd desktop  
C:\Users\571g-719q\Desktop>mkdir test  
C:\Users\571g-719q\Desktop>cd test  
C:\Users\571g-719q\Desktop\test>
```

➤ Struttura della directory

La struttura della directory contiene quanto segue :

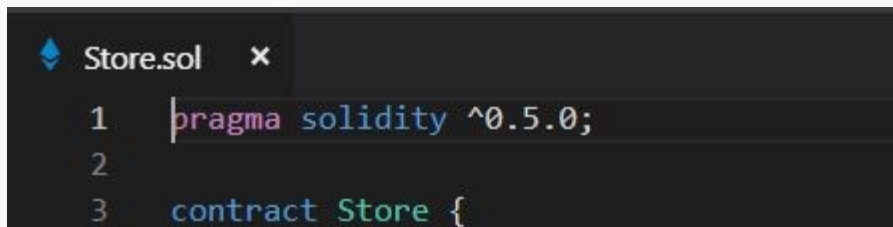
1. **contracts/**: contiene i file Solodity di origine per i nostri contratti intelligenti.
2. **migrations/**: Truffle utilizza un sistema di migration per gestire le distribuzioni di contratti intelligenti. Una migration è un ulteriore contratto intelligente speciale che tiene traccia delle modifiche.
3. **test/**: contiene i test Solidity per i nostri contratti intelligenti.
4. **truffle-config.js**: file di configurazione del Truffle.



4. SCRIVERE IL CONTRATTO INTELLIGENTE

Per scrivere lo Smart Contract ci siamo occupati di :

- Creare un nuovo file chiamato **Store.sol** nella **contracts/**directory.
- Aggiungere il seguente contenuto al file:



```
Store.sol x
1  pragma solidity ^0.5.0;
2
3  contract Store {
```

N.B.

La versione minima di Solidity richiesta si nota nella parte superiore del contratto: _

pragma solidity ^0.5.0; . Il comando pragma significa “*informazioni aggiuntive a cui interessa solo il compilatore*”; il simbolo del caret (^) identifica la versione indicata.

➤ Variable setup

Solidity è un linguaggio tipizzato staticamente.

Ha un tipo unico chiamato indirizzo . Gli indirizzi sono indirizzi Ethereum, memorizzati come valori di 20 byte.

Ogni account e contratto intelligente sulla blockchain di Ethereum ha un indirizzo e può inviare e ricevere Ether da e verso questo indirizzo.

Nel nostro caso definiamo una sola variabile: **prenotazioni**. Questa è un array di indirizzi di Ethereum.

L' array contiene un tipo e può avere una lunghezza fissa o variabile.

➤ Variable setup

...in base a quanto detto prima...

- Aggiungiamo la seguente variabile dopo **contract Store {** .

```
address [16] public prenotazioni;
```

N.B.

In questo caso l'array ha : tipo address e lunghezza 16.

➤ La prima funzione: prenotare una poltrona a teatro

```
//Prenotazione poltrone  
function prenota (uint Id) public returns (uint) {  
  require(Id >= 0 && Id <= 15);  
  prenotazioni[Id] = msg.sender;  
  return Id;  
}
```

N.B.

- In Solidity devono essere specificati i tipi di parametri di funzione e di output. In questo caso si prende un Id(intero) e si restituisce un intero.
- Il valore Id è compreso tra 0 e 15. Utilizziamo l'istruzione require() per garantire che l'Id sia compreso nell'intervallo. Se questo è compreso si aggiunge l'indirizzo che ha effettuato la chiamata al nostro prenotazioni array. L'indirizzo della persona o del contratto intelligente che ha chiamato questa funzione è indicato da msg.sender .
Infine, si restituisce Id come conferma.

➤ La seconda funzione: recuperare le poltrone prenotate

L'interfaccia utente deve aggiornare tutti gli stati di prenotazione delle poltrone.

Per fare ciò scriviamo una funzione che restituisce l'intero array.

```
// Recupero delle poltrone prenotate  
function getPrenotazione() public view returns (address[16] memory) {  
    return prenotazioni;  
}
```

N.B.

- prenotazioni è già stato dichiarato, possiamo semplicemente restituirlo.
- memory fornisce il percorso dei dati per la variabile.
- view, parola chiave nella dichiarazione di funzione, significa che la funzione non modificherà lo stato del contratto.

➤ Compilazione & Migrazione

Solidity è un linguaggio compilato, il che significa che dobbiamo compilare Solidity in bytecode per l'Ethereum Virtual Machine (EVM) da eseguire.

In un terminale, ci assicuriamo di essere nella radice della directory che contiene il dapp ed eseguiamo la compilazione:

```
C:\Users\571g-719q>cd desktop
C:\Users\571g-719q\Desktop>cd test
C:\Users\571g-719q\Desktop\test>truffle compile

Compiling your contracts...
=====
> Everything is up to date, there is nothing to compile.
```

Compilato con successo i contratti, è tempo di migrarli alla blockchain!

Una migrazione è uno script di distribuzione che modifica lo stato dei contratti dell'applicazione, spostandolo da uno stato a quello successivo.

➤ Compilazione & Migrazione

Il file JavaScript, nella migrations/directory, 1_initial_migration.js, gestisce l'implementazione del Migrations.sol, contratto per osservare successive migrazioni di contratti intelligenti e garantisce che non si esegua una doppia migrazione dei contratti. Dopo di che :

- creiamo un nuovo file chiamato 2_deploy_contracts.js nella migrations/directory.
- aggiungiamo

```
1  var Store = artifacts.require("Store");
2
3  module.exports = function(deployer) {
4    deployer.deploy(Store);
5  };
```

Prima di poter eseguire la migrazione del contratto alla blockchain, è necessario avere una blockchain in esecuzione.

➤ Compilazione & Migrazione

Pertanto abbiamo scaricato Ganache e avviato l'applicazione. Questo genererà una blockchain in esecuzione su sever locale sulla porta 9545 , con hostname 12.0.0.1 e network id 5777.

- Una volta avviato Ganache, migriamo il contratto con il seguente comando da terminale:

```
C:\Users\571g-719q\Desktop\test>truffle migrate

Starting migrations...
=====
> Network name:    'development'
> Network id:     5777
> Block gas limit: 6721975

1_initial_migration.js
=====

  Replacing 'Migrations'
  -----
  > transaction hash: 0x074c6966480ae81cc6f4b8efe0c9e13
  > Blocks: 0        Seconds: 0
  > contract address: 0xB6BCf7AACCc15c5A26911e8f89b8E0a
  > account:         0xaa66025610160C758B60AF4F1DF5EE7
  > balance:         99.99430312
  > gas used:        284844
  > gas price:       20 gwei
  > value sent:      0 ETH
  > total cost:      0.00569688 ETH

  > Saving migration to chain.
  > Saving artifacts
  -----
  > Total cost:      0.00569688 ETH
```

```
2_deploy_contracts.js
=====

  Replacing 'Store'
  -----
  > transaction hash: 0x444257a0725a7f306960411cfc37e6c0a215c0d2fe2090ca7e741d076403f853
  > Blocks: 0        Seconds: 0
  > contract address: 0xd963078c8a707974998856cf82a5b69741070847
  > account:         0xaa66025610160C758B60AF4F1DF5EE7
  > balance:         99.98838504
  > gas used:        253020
  > gas price:       20 gwei
  > value sent:      0 ETH
  > total cost:      0.0050764 ETH

  > Saving migration to chain.
  > Saving artifacts
  -----
  > Total cost:      0.0050764 ETH

Summary
=====
> Total deployments: 2
> Final cost:       0.01077328 ETH
```

5. TEST DEL CONTRATTO INTELLIGENTE

Una volta scritto il contratto intelligente e averlo implementato in una blockchain in esecuzione reale, interagiamo con esso per assicurarci che faccia ciò che vogliamo.

Pertanto :

- Creiamo un nuovo file chiamato **TestStore.sol** nella **test/directory**.
- Aggiungiamo il seguente contenuto al file **TestStore.sol** :

```
1  pragma solidity ^0.5.0;
2
3  import "truffle/Assert.sol";
4  import "truffle/DeployedAddresses.sol";
5  import "../contracts/Store.sol";
6
7  contract TestStore{
8
9      // L'indirizzo del contratto da testare
10     Store store = Store(DeployedAddresses.Store());
11
12     // L'id della poltrona che verrà utilizzato per il test
13     uint expectedId =8;
14
15     // Indirizzo del cliente atteso della poltrona prenotata
16     address expectedPrenota = address(this);
17 }
```


N.B.

- Assert.sol: verifica elementi come l'uguaglianza, la disuguaglianza etc., affermando se il nostro test sia pass o fail .
- DeployedAddresses.sol: durante l'esecuzione dei test, Truffle distribuirà una nuova istanza del contratto sottoposto a test sulla blockchain. Questo contratto intelligente ottiene l'indirizzo del contratto distribuito.
- Store.sol: il contratto intelligente che vogliamo testare.

Definiamo tre variabili a livello di contratto: in primo luogo, una contenente il contratto intelligente da testare, chiamando DeployedAddresses, contratto intelligente per ottenere il suo indirizzo.

In secondo luogo, Id della poltrona che verrà utilizzata per testare le funzioni di prenotazione.

In terzo luogo, poiché il contratto TestStore invierà la transazione, impostiamo l'indirizzo di prenotazione previsto su questa , una variabile a livello di contratto che ottiene l'indirizzo del contratto corrente.

➤ Test della funzione prenota()

Per testare la funzione **prenota()** vediamo se ci viene restituito un Id che sia corretto confrontando il valore di ritorno della funzione.

Aggiungi la seguente funzione all'interno del contratto **TestStore.sol**, dopo la dichiarazione di **Store**:

```
//Test della funzione prenota (Store.sol)
function testPrenotazione() public {
    uint returnedId = store.prenota(expectedId);

    Assert.equal(returnedId, expectedId,
        "La prenotazione attesa non corrisponde alla poltrona.");
}
```

N.B.

- Chiamiamo il contratto intelligente che abbiamo dichiarato in precedenza con l'Id di expectedId.
- Infine, passiamo il valore attuale, il valore atteso e un messaggio di errore (che viene stampato sulla console se il test non passa) ,Assert.equal().

➤ Test per recuperare il proprietario di una singola poltrona

- Aggiungiamo questa funzione sotto la funzione aggiunta in precedenza in **TestStore.sol**.

```
// Test del recupero della singola prenotazione
function testGetPrenotazione() public {
    address prenotazione = store.prenotazioni(expectedId);

    Assert.equal(prenotazione, expectedPrenota,
        "Prenotazione prevista dal contratto");
}
```

➤ Test per recuperare tutti i proprietari delle poltrone

- Aggiungiamo questa funzione sotto la funzione aggiunta in precedenza in **TestStore.sol**.

```
//Test per recuperare tutti i proprietari delle poltrone
function testGetPrenotazioneArray() public {
    address [16] memory prenotazioni = store.getPrenotazione();

    Assert.equal(prenotazioni[expectedId], expectedPrenota,
        "Prenotazione prevista dal contratto (Array)");
}
```

N.B.

L'attributo memory dice a Solidity di memorizzare temporaneamente il valore in memoria, anziché salvarlo nella memoria del contratto. Poiché prenotazioni è un array e dal primo test di prenotazione sappiamo che abbiamo prenotato una poltrona, expectedId, confrontiamo l'indirizzo dei contratti di test con la posizione expectedId nell'array.

➤ Esecuzione dei test

A questo punto procediamo con l'esecuzione dei test.

Nel terminale inviamo il seguente comando :

```
C:\Users\571g-719q\Desktop\test>truffle test
Using network 'development'.

Compiling your contracts...
=====
> Compiling .\test\TestStore.sol
> Artifacts written to C:\Users\571G-7~1\AppData\Local\Temp\test-119211-16532-1yko0hu.mfo6
> Compiled successfully using:
   - solc: 0.5.0+commit.1d4f565a.Emscripten.clang

TestStore
  ✓ testPrenotazione (121ms)
  ✓ testGetPrenotazione (120ms)
  ✓ testGetPrenotazioneArray (275ms)

3 passing (12s)
```

Dobbiamo verificare che passino tutti i test!

6. CREAZIONE DI UN'INTERFACCIA UTENTE PER INTERAGIRE CON IL CONTRATTO INTELIGENTE

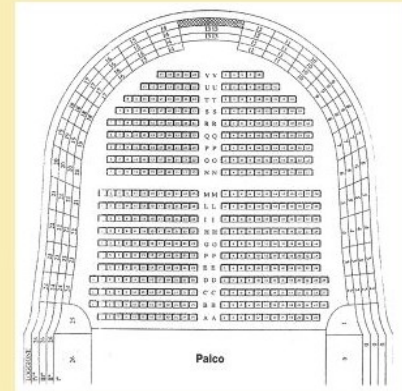
Ricapitolando fin ora abbiamo creato il contratto intelligente, lo abbiamo implementato nella nostra blockchain di test locale e confermato che possiamo interagire con esso tramite la console, è ora di creare un'interfaccia utente.

Nella cartella **test/src** creiamo un file **index.html** dove scriviamo il codice html per creare la pagina web del teatro .

Così facendo la struttura della dapp è pronta, dobbiamo solo compilare le funzioni uniche per Ethereum.



PRENOTA LA TUA POLTRONA A TEATRO



Prima di procedere dobbiamo dire che...

La rete Ethereum è composta da nodi, ognuno dei quali contiene una copia della blockchain. Quando si desidera chiamare una funzione su un contratto intelligente, è necessario interrogare uno di questi nodi e dirgli:

1. L'indirizzo del contratto intelligente.
2. La funzione che si desidera chiamare.
3. Le variabili che si desidera passare a quella funzione.

I nodi di Ethereum parlano solo un linguaggio chiamato JSON-RPC , che non è molto leggibile. Una query per indicare al nodo che si desidera chiamare una funzione su un contratto è scritta in un modo quasi “incomprensibile”, ma fortunatamente Web3.js permette di interagire con un’interfaccia JavaScript, facilmente leggibile.

Detto ciò possiamo procedere!

➤ Instantiating Web3

1. Creiamo il file .js, `/src/js/app.js`.
2. Esaminando il file notiamo che esiste `App`, il quale è un oggetto globale per gestire la nostra applicazione, carichiamo i dati della poltrona `init()` e chiamiamo la funzione `initWeb3()`. La libreria JavaScript Web3 interagisce con la blockchain di Ethereum, la quale può recuperare account utente, inviare transazioni, interagire con contratti intelligenti e altro.
3. All'interno della funzione `initWeb3` riportiamo quanto segue:

```
initWeb3: async function() {  
  // Browser moderni dapp...  
  if (window.ethereum) {  
    App.web3Provider = window.ethereum;  
    try {  
      // Richiesta d'accesso all'account  
      await window.ethereum.enable();  
    } catch (error) {  
      //accesso negato all'account dell'utente  
      console.error("Accesso negato all'account")  
    }  
  }  
  // I browser obsoleti di dapp...  
  else if (window.web3) {  
    App.web3Provider = window.web3.currentProvider;  
  }  
  //Se non viene rilevata nessuna istanza web3, torna a Ganache  
  else {  
    App.web3Provider = new Web3.providers.HttpProvider('http://localhost:9545');  
  }  
  web3 = new Web3(App.web3Provider);  
}
```


N.B.

1. Controlliamo se stiamo usando i moderni browser dapp o le versioni più recenti di MetaMask in cui ethereum è mandato in esecuzione permettendo di accedere al nostro sito e richiediamo l'accesso agli account con ethereum.enable().
2. Se l'oggetto ethereum non esiste, controlliamo quindi l'istanza web3 . Se esiste, indica che stiamo usando un browser più vecchio (una versione precedente di MetaMask). Se è così, otteniamo il suo provider e lo usiamo per creare il nostro oggetto web3.
3. Se non è presente un'istanza web3, creiamo il nostro oggetto web3 in base al nostro provider locale.

➤ Istanziare il contratto

Ora possiamo interagire con Ethereum tramite **web3**.

Dobbiamo creare un'istanza del nostro contratto intelligente in modo che **web3** sappia dove trovarlo e come funziona. Truffle ha una libreria che permette di fare ciò , **truffle-contract**; questa mantiene le informazioni sul contratto in sincronia con le migrazioni, quindi non è necessario modificare manualmente l'indirizzo distribuito del contratto.

- In **/src/js/app.js**, nella funzione **initContract** si riporta quanto segue:

```
initContract: function() {  
  $.getJSON('Store.json', function(data) {  
    //Si prende il file degli artefatti necessari del contratto e si crea un'istanza con truffle-contract  
    var Prenotazione = data;  
    App.contracts.Store = TruffleContract(Prenotazione);  
  
    //Imposta il provider per il nostro contratto  
    App.contracts.Store.setProvider(App.web3Provider);  
  
    //Usare il contratto per recuperare e contrassegnare le poltrone prenotate  
    return App.markPrenota();  
  });  
}
```

N.B.

- Per prima cosa recuperiamo il file degli artefatti del nostro contratto intelligente. Gli artefatti sono informazioni sul nostro contratto come il suo indirizzo distribuito e l'Application Binary Interface (ABI) . L'ABI è un oggetto JavaScript che definisce come interagire con il contratto incluse le sue variabili, funzioni e i relativi parametri.

- Poi passiamo a TruffleContract(). Questo crea un'istanza del contratto con cui possiamo interagire.

Con il nostro contratto istanziato, impostiamo il provider web3 utilizzando il valore App.web3Provider che abbiamo archiviato in precedenza durante la configurazione di web3.

- Chiamiamo markPrenota(), funzione dell'app nel caso in cui alcune poltrone siano già state prenotate da una visita precedente. L'abbiamo incapsulato in una funzione separata poiché dovremo aggiornare l'interfaccia utente ogni volta che apportiamo una modifica ai dati del contratto intelligente.

7. INTERAGIRE CON IL DAPP IN UN BROWSER

Interagiamo con la dapp in un browser attraverso MetaMask.

1. Installiamo nel nostro browser MetaMask.
2. Lo avviamo.
3. Connettiamo MetaMask alla blockchain creata da Ganache.

The screenshot shows the Ganache desktop application. At the top, there's a navigation bar with icons for ACCOUNTS, BLOCKS, TRANSACTIONS, and LOGS. Below this is a status bar displaying various network metrics: CURRENT BLOCK (51), GAS PRICE (2000000000), GAS LIMIT (6721975), NETWORK ID (5777), RPC SERVER (HTTP://127.0.0.1:9545), and MINING STATUS (AUTOMINING). The main area displays two transaction details. Each entry includes a TX HASH, FROM ADDRESS, TO CONTRACT ADDRESS, GAS USED, and VALUE. The first transaction has a hash starting with 0xaecdeb4b28f45b32f127e3368d3aeca8aa6dc2f3c1ab963a68f7dea4975d3647 and a gas used of 42120. The second transaction has a hash starting with 0xefc537a4fed1526656cde3366aba86492061db812394996a9756dde177cff93d and a gas used of 35646. A 'CONTRACT CALL' button is visible next to each transaction entry.

The screenshot shows a MetaMask Notification window. It features the MetaMask fox logo at the top. Below the logo, the text reads 'Bentornato!' (Welcome back!) and 'Il web decentralizzato ti attende' (The decentralized web awaits you). There is a 'Password' input field. Below the password field is a large orange button labeled 'ACCEDI' (Log in). At the bottom, there is a link that says 'Ripristina da una frase seed' (Restore from a seed phrase) and 'Importa account con frase seed' (Import account with seed phrase).

➤ Usare il dapp

Ora possiamo avviare un server web locale e usare il dapp, installando e configurando da terminale la libreria lite-server.

Avviamo il server web locale:

```
C:\Users\571g-719q\Desktop\test>npm run dev
> theater@1.0.0 dev C:\Users\571g-719q\Desktop\test
> lite-server

** browser-sync config **
{ injectChanges: false,
  files: [ '**/*.html', '**/*.css', '**/*.js' ],
  watchOptions: { ignored: 'node_modules' },
  server: {
    baseDir: [ './src', './build/contracts' ],
    middleware: [ [Function], [Function] ] } }
[Browsersync] Access URLs:
  Local: http://localhost:3000
  External: http://192.168.56.1:3000
  UI: http://localhost:3001
  UI External: http://localhost:3001
[Browsersync] Serving files from: ./src
[Browsersync] Serving files from: ./build/contracts
[Browsersync] Watching files...
19.03.11 12:12:48 200 GET /index.html
19.03.11 12:12:48 200 GET /css/bootstrap.min.css
19.03.11 12:12:48 200 GET /images/pianta.jpeg
19.03.11 12:12:48 200 GET /images/teatro.jpg
19.03.11 12:12:48 200 GET /js/bootstrap.min.js
19.03.11 12:12:48 200 GET /js/app.js
19.03.11 12:12:48 200 GET /js/web3.min.js
19.03.11 12:12:48 200 GET /images/poltrona.jpg
19.03.11 12:12:48 200 GET /js/truffle-contract.js
19.03.11 12:12:49 404 GET /favicon.ico
19.03.11 12:12:49 200 GET /poltrone.json
```

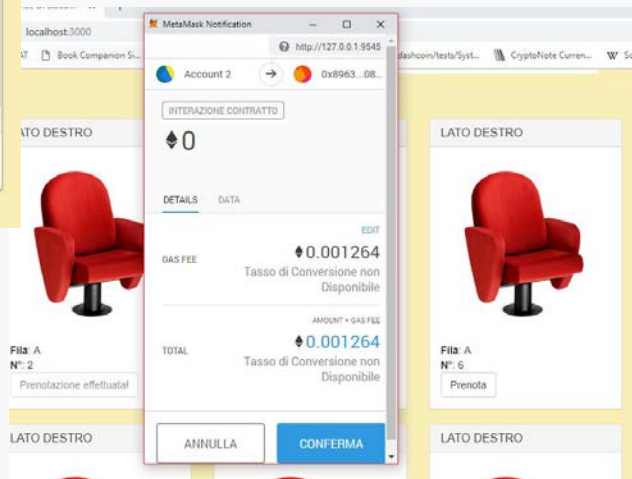
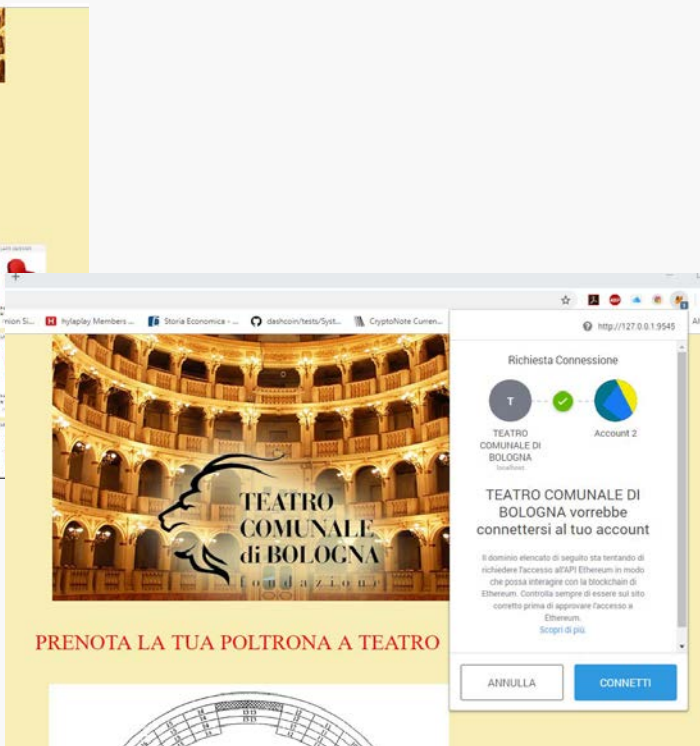
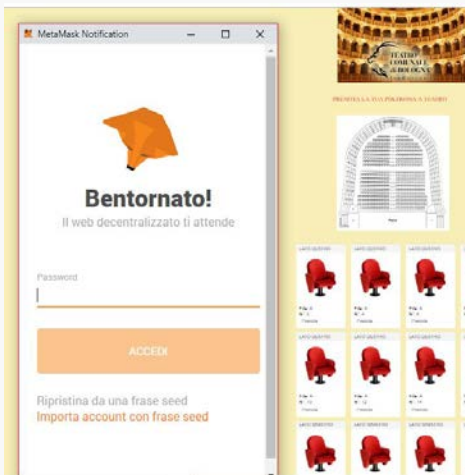
➤ Usare il dapp

Il server di sviluppo avvierà e aprirà automaticamente una nuova scheda del browser contenente il dapp.

Per utilizzare il dapp, facciamo clic sul pulsante **Prenota** della poltrona a nostra scelta.

Ci verrà automaticamente richiesto di approvare la transazione tramite MetaMask.

In MetaMask vediamo effettivamente la transazione ed è possibile vedere la stessa elencata in Ganache.



TX HASH: BxhrcRzQ7aBcRzBfzhhaCdfKdKdKdRk7fcQQKRdeD7K1Z7rfLfchd1a7a71171ffcc77Q CONTRACT CREATION

[illegible]

MESSAGE	HD PATH
message siren oyster thing bracket quarter notice short thought auto blame evidence	n/44/60/0/0/account_index

ADDRESS	BALANCE	TX COUNT	INDEX	
0xaa66025610160C758B60AF4FD5EE7a0881D40b	99.67 ETH	28	0	
ADDRESS	BALANCE	TX COUNT	INDEX	
0x7a6e781738F46a8929aC4ca597a46A0eEfC20a6	100.00 ETH	0	1	
ADDRESS	BALANCE	TX COUNT	INDEX	
0xb4b4e6Fa6a8E36440eD07aa420eD7d8F7755804	100.00 ETH	0	2	

A digital illustration of a theater stage. The stage is framed by heavy, draped red curtains that are pulled back to reveal a light-colored, textured floor. In the center of the stage, the word "FINE!" is written in a large, white, serif font. The foreground shows the backs of several rows of red theater seats, suggesting an audience's perspective. The lighting is soft, with a slight glow around the text.

FINE!