# Parity for Private Ethereum Blockchain Network

## Introduction

In the previous document, steps for parity installation and configuration were discussed for a private network Ethereum Blockchain. The document also focused on the operations involving smart contract development and deployment using Parity UI. Furthermore Web3.js was also discussed and how it can be used to perform certain operations on the blockchain and how it can interact with a Smart Contract deployed on Ethereum blockchain. This document will also focus on the same components as discussed in the previous document with further detailed and simplified instructions. So, this document is an extension of a previous document shedding more light on major components such as **Parity, Parity UI**, **Web3.js** and **Smart Contract**.

## System Configuration

The operations highlighted in the document are performed on MacOS and all the configuration of parity and parity UI is done accordingly.

## Assumptions

Before going in to details and discussion about Parity and how to use Parity UI and Web3.js, it is assumed that following steps have already been performed and the mentioned components are already installed and available. Refer to the previous document in case of installation steps and instructions.

* Parity and Parity UI are already installed on the system
* Node JS is already installed on the system
* Rust is already installed on the system

# Working with Parity and Parity UI

This section of the document focuses on the steps involved in configuration of Parity for private network Ethereum Blockchain and how Parity UI can be used to interact with Ethereum Blockchain

***Note:*** Running Parity as default mode will configure the Ethereum Blockchain in a **public network** (Foundation mode) where the node will automatically connect and sync with other external nodes involved in the same public network.

## Configuring Private Network Ethereum Blockchain through Parity

As mentioned earlier, Parity runs in public mode on default settings. To restrict the Ethereum Blockchain to run on private network, Parity needs to be launched in **dev** mode which will initialize the Ethereum Blockchain in private network environment and node will not connect and sync with other external nodes without proper permission and authorization. Firstly, network should be configured as private network with only specific nodes. Nodes must also be registered in the network.

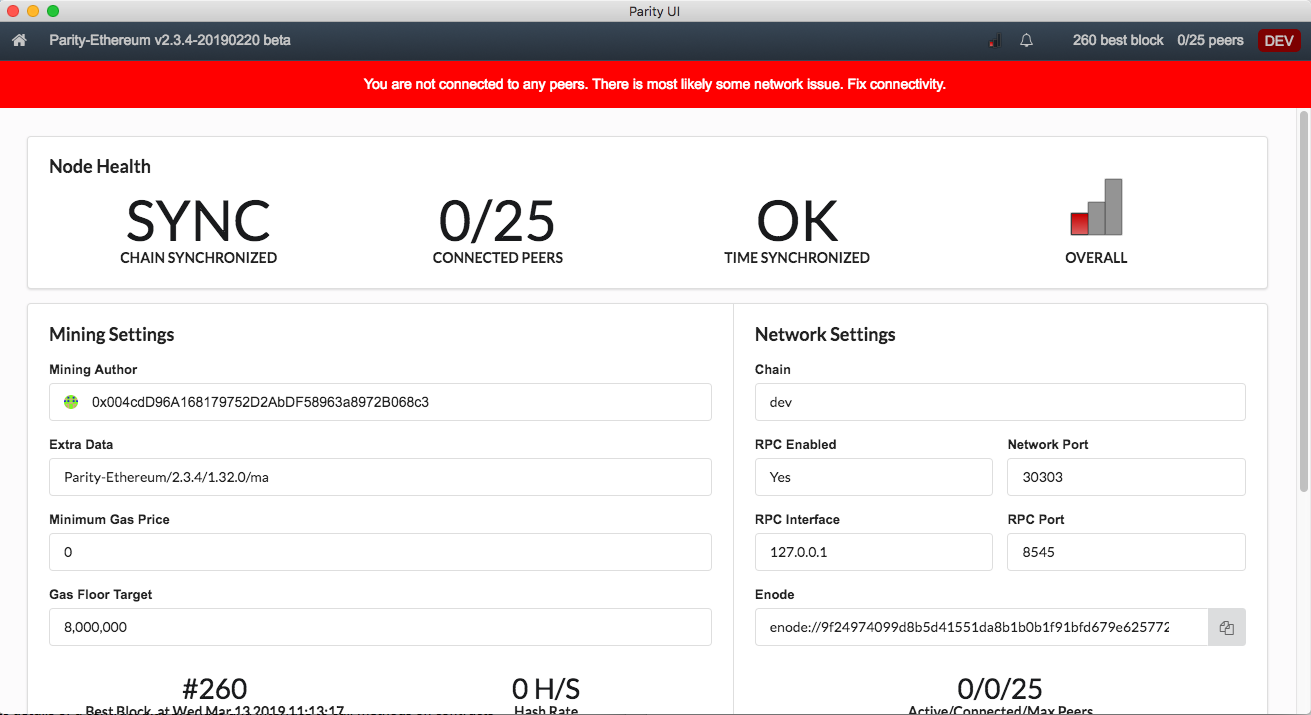
## Registering nodes in Private Network Ethereum Blockchain

To register nodes and allowing them to join private network, they must be registered in a single file. An empty file (say, reservedNodes.txt) is created which will contain the addresses of all the nodes to be involved in private network and listing them as peers in private network Ethereum Blockchain. The address of any particular node running parity can be seen in Node Status in Parity UI which is also running on that particular node.

## 

*Figure 1: Parity UI home page*

After clicking on Node Status, information about that particular node will be displayed in Network Settings shown in Figure 2 below.



*Figure 2: Node Status in Parity UI*

The field **Enode** in Network Settings contain the address of that particular node on which the Parity and Parity UI is configured and running.

To register a node in private network, the address mentioned in **Enode** field should be written in reservedNodes.txt file in the same format as mentioned. To register multiple nodes, the address of the other node should be written in the same format but in a new line of the file and so on

***enode:// [Replace with Address of the Node]*** *// First Node*

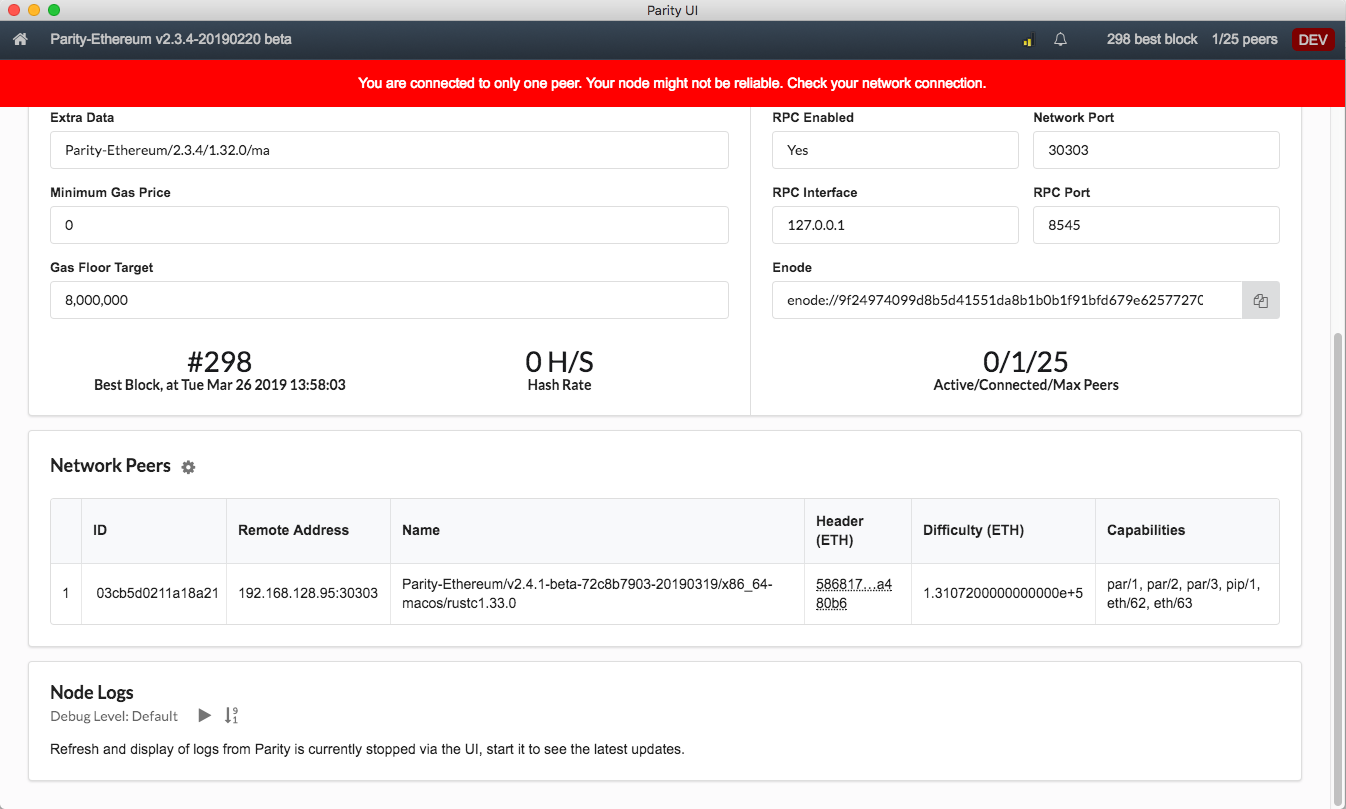
***enode:// [Replace with Address of the Node]*** // Second Node

From then on it is assumed that reservedNodes.txt contains all other nodes which are to be involved in the private network Ethereum blockchain. From now on, Parity can use the reservedNodes.txt file to launch parity in private network (dev mode) and register the mentioned nodes in private network Ethereum Blockchain by the following command

***parity --reserved-peers [path to the reservedNodes.txt file] --config dev --allow-ips private***

The **--config dev** keywork in the command will automatically configure the Ethereum Blockchain to run in Private Network Environment and **--allow-ips private** will block the other external nodes to connect with your node. Nodes can further be added or removed by simply modifying the reservedNodes.txt file and launching the parity again with the same command mentioned above.

After configuring the network for Private Ethereum Blockchain and registering nodes using reservedNodes.txt file, information about all connected nodes can be seen in **Network Peers** field located inside Node Status.



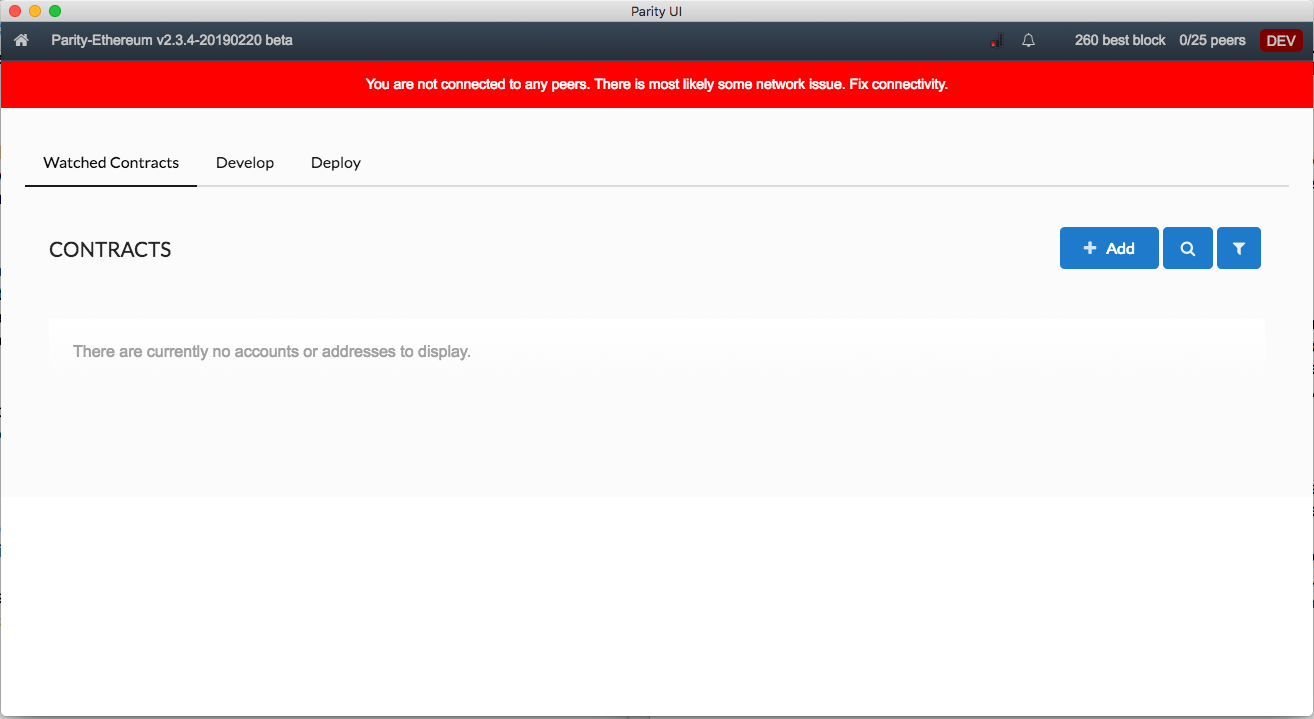
*Figure 3: Network Peers Information*

# Developing Smart Contracts

Parity UI provides the capability to directly write and deploy smart contracts on the Ethereum Blockchain. Furthermore, the deployed smart contract can also be executed within Parity UI. This section will discuss step by step approach to compile smart contracts, deploy smart contracts on private network Ethereum Blockchain and how to interact with deployed smart contract with Parity UI functionality.

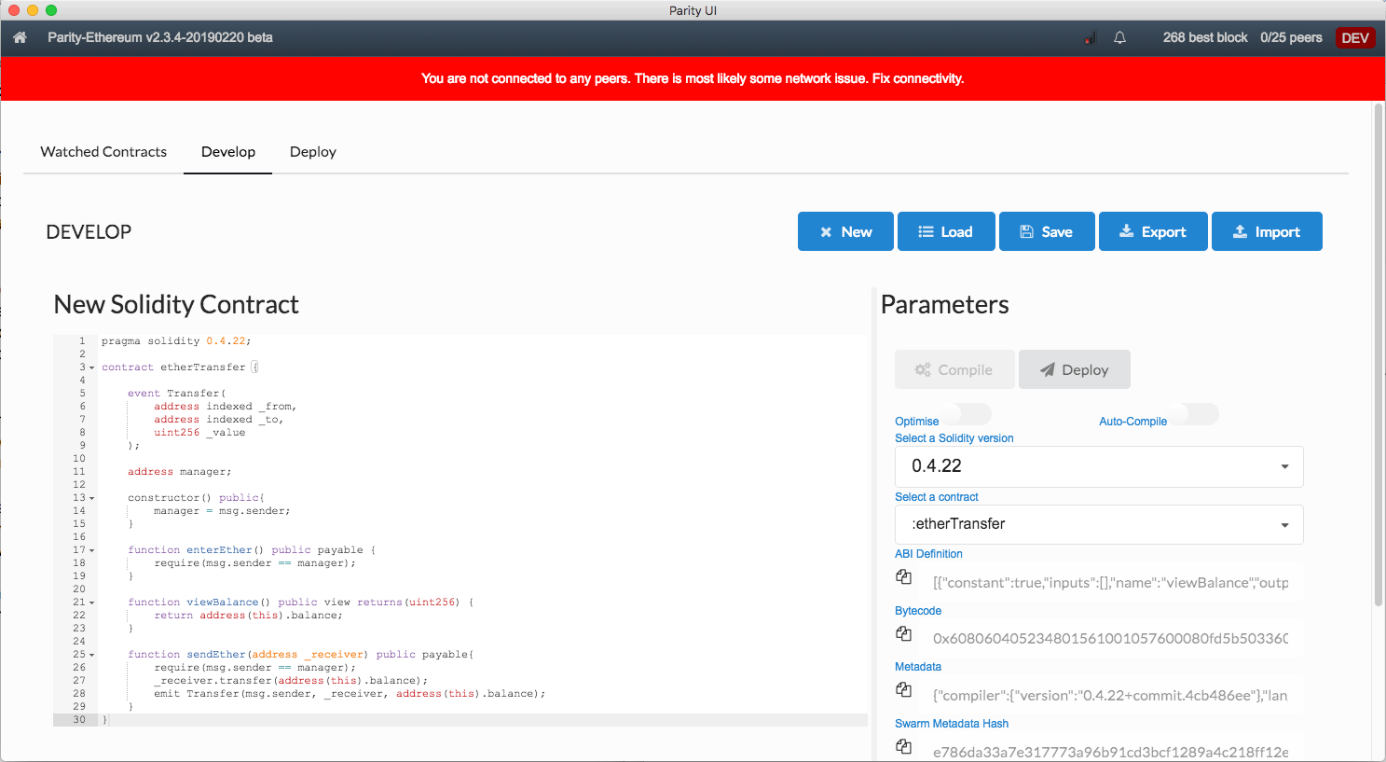
## Writing Smart Contracts in Parity UI

Parity UI allows smart contract writing and deployment with the specific account. All contracts which are already developed and deployed on Ethereum Blockchain will be shown in **Contracts** field. At start there will be no Contract showing because no smart contract has yet been deployed on privately configured Ethereum Blockchain.



*Figure 4: Smart Contract Development and Deployment*

Smart Contract writing can be started by clicking on **Develop** tab as shown in Figure 4



*Figure 5: Writing a Smart Contract*

A simple smart contract example **etherTransfer** has been written as shown in Figure 5. The programming language in writing smart contract for Ethereum Blockchain is **Solidity**. The smart contract in the Figure 5 simply transfers the overall balance of smart contract to a specific address. This smart contract also gives the authority to the deployer of the contract and only allows deployer to execute the restricted contract functions. The working of the smart contract is following:

* ***pragma solidity 0.4.22;***

Pragma is a keyword that set the instructions for the compiler on how to treat that source code. In this smart contract it mentions that the solidity version for this smart contract is **0.4.22** and only the compilers with matching or higher version can compile the smart contract.

***Note:*** The compiler version used for the example smart contract is also 0.4.22. Selecting a compiler of lower version will result in compilation errors and won’t be able to deploy the smart contract

* **contract etherTransfer {…**

Next line in the smart contract simply takes the keyword **contract** along with the name of the contract

* **address manager;**

**constructor () public { manager = msg.sender; }**

Event Transfer is skipped for now and will be explained later in the document. **address** is a data type in solidity which is used to store addresses. In this example smart contract, a variable **manager** of address type is defined.

The constructor function specifies that the deployer of the smart contract will be the manager of the smart contract. **msg** is a global namespace and is used to provide information about the Ethereum Blockchain. **msg.sender** holds the address of the deployer of the smart contract and is stored in manager variable.

* **function enterEther() public payable {require (msg.sender == manager) ;}**

The function **enterEther** allows the manager of the contract to send ethers to the smart contract. Smart contract will hold the value as deposit. The keyword **payable** makes this function available to accept ethers and will create a transaction in Ethereum Blockchain whenever manager send ethers to the smart contract through this function. The **require** keyword specifies the condition and function will not execute if the mentioned condition is not fulfilled. The above function only authorizes the deployer of contract as manager and only allows the manager to execute that function and no other account has permission to send ethers to the smart contract.

* **function viewBalance() public view returns(uint256) { return address(this).balance;}**

The function **viewBalance** is only a viewable function specified by **view** keyword. This function only returns the overall balance of the smart contract. It can be observed that viewBalance function doesn’t contain any condition or restriction and can be called from any account. **address(this)** holds the address of smart contract stored in Ethereum Blockchain and **balance** is a property of the address returning the overall balance of a particular address.

* **function sendEther(address \_receiver) public payable**

**{require(msg.sender == manager);**

**\_receiver.transfer(address(this).balance);**

**emit Transfer(msg.sender, \_receiver, address(this).balance);**

**}**

The last function **sendEther** is the main function of the smart contract as it transfers the overall balance of a smart contract to a specific address provided by the manager. This function is also **payable** and will also result in transaction in Blockchain upon every time it is executed by the manager. This function also holds a condition and only authorizes the manager to execute this function. **\_receiver** variable holds the address of the receiving account and **transfer** keyword transfers the overall balance of smart contract to the account provided by the manager.

The last line of the function emits an event named **Transfer** with the address of executor (manager) of the function, address of the receiver and overall balance of the smart contract.

**Events** are inheritable members of contracts. Upon emitting an event they result in causing the arguments to be stored in transaction’s logs which is a special data structure in the blockchain. Events allow tracking and logging of transactions and are customized according to the requirements of the user.

***Note:*** Events can be subscribed by any end user. By subscribing to a specific event, subscriber will be notified every time whenever a function is called and results in emitting that specific event.

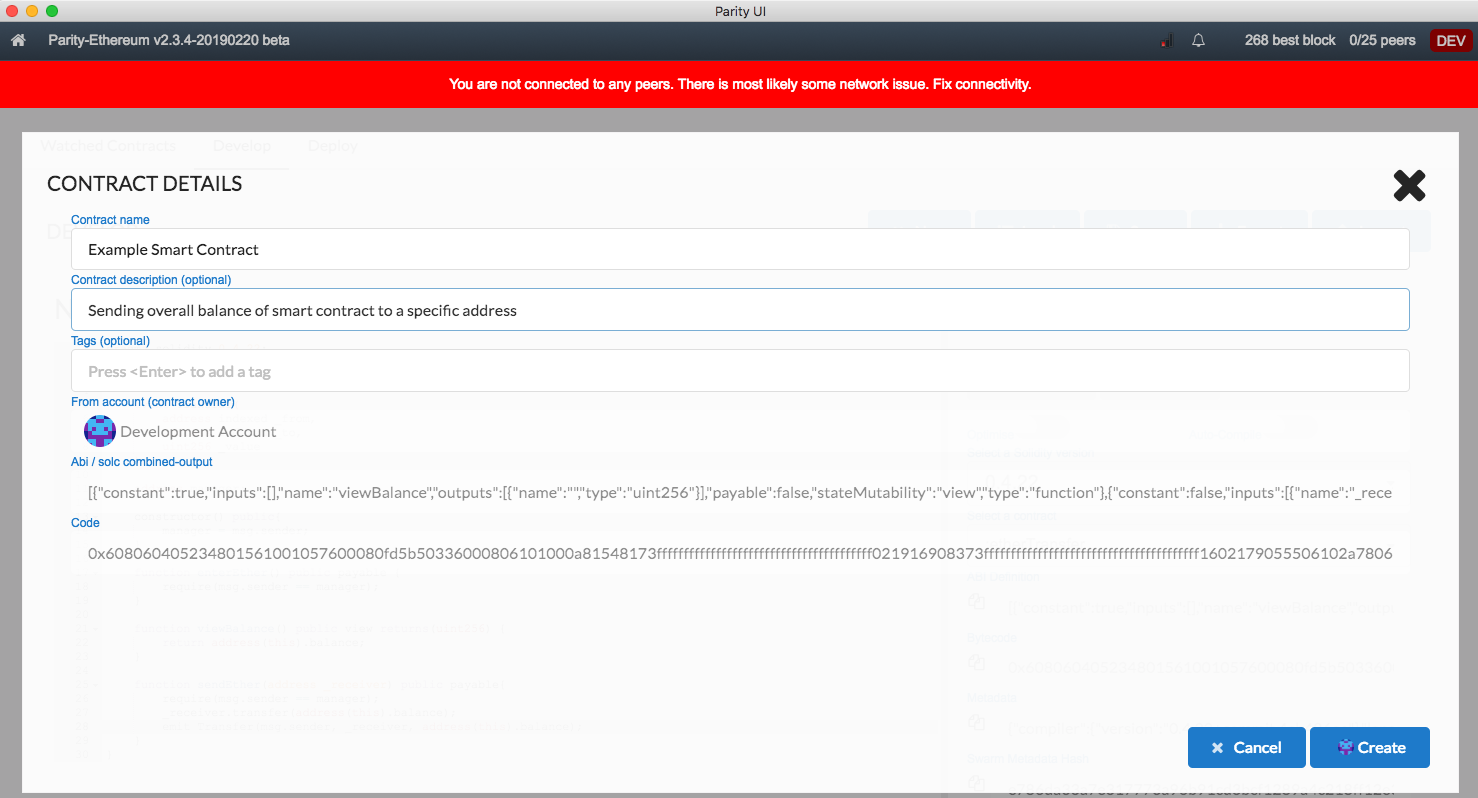
At start of the smart contract an event named **Transfer** is described.

* **event Transfer(address indexed \_from, address indexed \_to, uint256 \_value);**

This event will result in emitting out the address of the caller, address of the receiver and amount of transferred ethers. This event is mentioned in **sendEther** function and logs the address of the caller restricted to manager, address of the receiving account provided by the manager and the total balance of the smart contract at that particular state.

Further information about solidity and its documentation can be found [here](https://solidity.readthedocs.io/en/v0.4.24/).

After writing the smart contract and compiling it successfully, the next step is to deploy the smart contract on Ethereum Blockchain. Then click on the **Deploy** tab as shown in Figure 5.

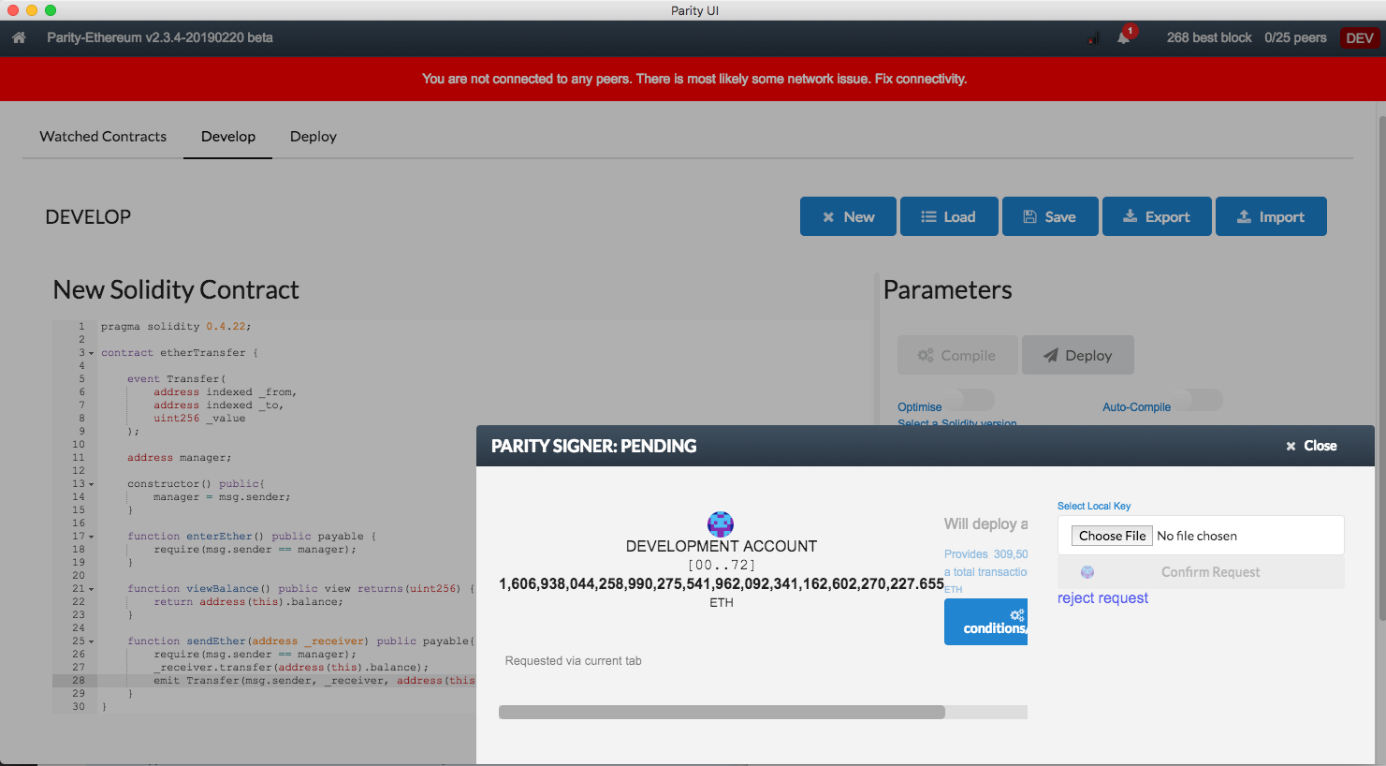


*Figure 6: Deploying Smart Contract*

**Contract Name** must be mentioned but Contract description and Tags are optional. The **from account** specifies the account who will deploy the contract and will own that contract. The selected account will be authorized to send ethers to the smart contract and then transfer the overall balance to a specific address.

**Abi** of the contract is a Contract Application Binary Interface and it provides a standard way in Json format to interact with smart contract and invoke functions described in smart contract. Abi allows the contract to be called from either outside of the Ethereum blockchain or within the Ethereum blockchain. The Abi of a contract is automatically generated on compiling the contract successfully and it contains all the methods and events along with other relevant information about a smart contract. The interaction with a smart contract using Web3.js is discussed in detail in next section.

Clicking on create button after providing the correct information will result in following prompt

****

*Figure 7: Deploying Smart Contract*

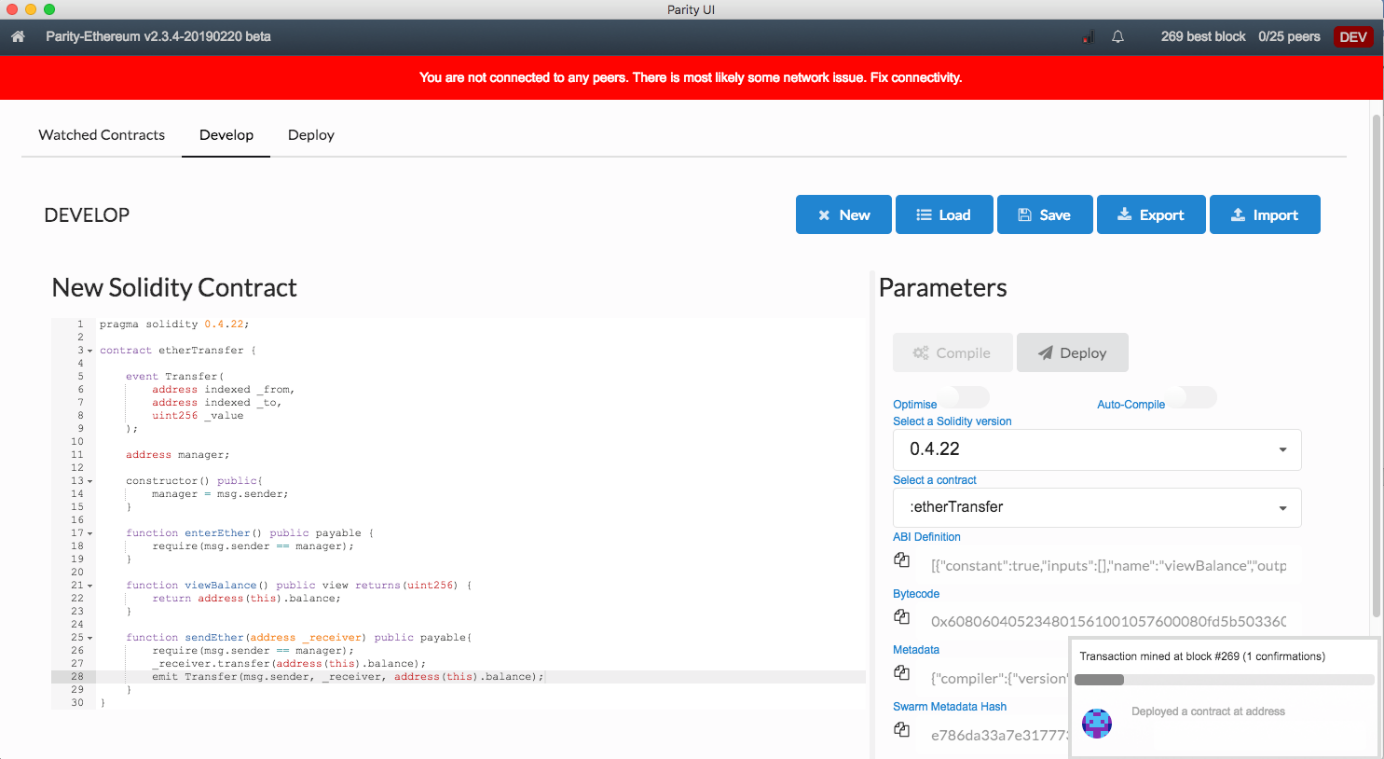
The prompt requires a **Private Key** of the deployer account to deploy the smart contract on Ethereum Blockchain. The keys and information about the blockchain are stored in a special folder named **io.parity.ethereum** which is automatically created on launching the parity.

Navigation from the root to the folder can done by typing the following command in terminal

* ***cd /Library/Application Support/io.parity.ethereum***

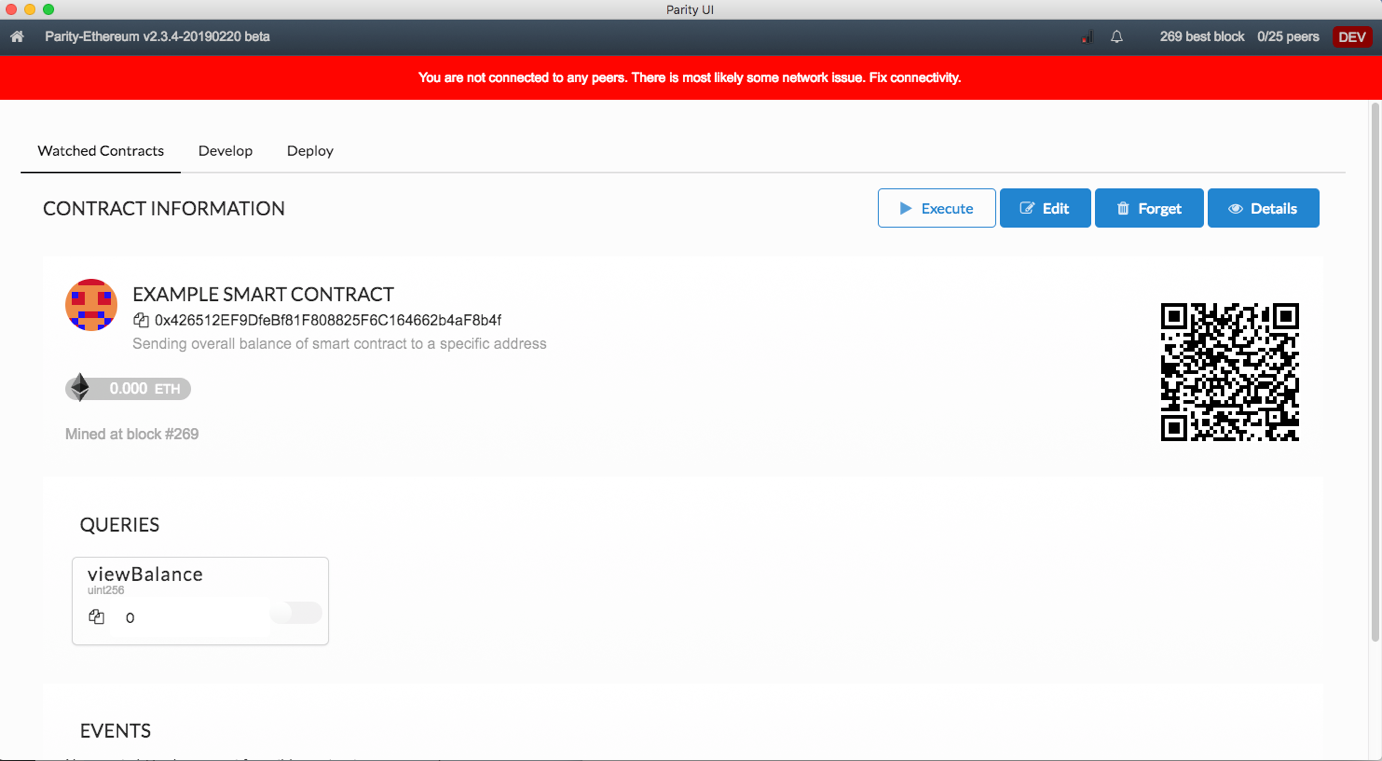
The folder named **chains** contain the information of all the Ethereum blockchains which are configured or launched at least once on the particular node. The **keys** folder holds all the private keys of all the accounts involved in Ethereum Blockchain network. Two folders will be shown named **DevelopmentChain** and **ethereum**. **DevelopmentChain** folder holds information for privately configured Ethereum Blockchain and private key for a specific account can be accessed from there.

After providing the correct **Private Key** and clicking on Confirm Request. A transaction will be created in Ethereum Blockchain deploying the smart contract with a specific contract address stored in Ethereum Blockchain.



*Figure 8: Transaction Confirmation of contract deployment*

After deploying the smart contract successfully, all deployed contracts along with contract address and information can be viewed in **Watched Contracts** tab



*Figure 9: Deployed Contracts Information*

# Working and interacting with smart contract using Web3.js

Web3.js is a collection of libraries which allows to interact with a local or remote ethereum node using a HTTP or IPC connection. It is also used to interact with smart contracts deployed on Ethereum Blockchain and to invoke functions of smart contracts.

This section of the documentation will focus on the steps involving installing Web3.js and how it can be used to perform various operations on Ethereum Blockchain and smart contract interaction.

To add Web3.js to the system, the following command is executed using terminal

* npm install web3

From then, Web3.js can be set up for privately configured Ethereum network by typing the following commands in order. The code is written in a file for testing purposes and then executed with node.

* var Tx = require('ethereumjs-tx')
* Web3 = require('web3')
* web3 = new Web3('ws://localhost:8546')

Web3.js is now configured for a private network Ethereum Blockchain. Port **8546** is where Parity Web Sockets are configured and allow Web3.js to query Ethereum Blockchain without the involvement of Parity UI. The network configuration can be seen in network settings as shown in Figure 2.

**ethereumjs-tx** is an Ethereum library which is used in creating Transaction objects, signing and broadcasting the transactions and query transactions data stored in Ethereum Blockchain.

Next step is to create a contract object which allows to interact with smart contract and provides various functionalities to interact with smart contract and execute smart contract functions described in the contract.

* var contract = new web3.eth.Contract(Abi, address)

Contract method takes at least two arguments which are **Abi** (Application Binary Interface) and **contract** **address**. Contract Abi and Contract Address can be seen in contract details as shown in Figure 9.

After creating the contract object, it enables the user to interact with smart contract deployed on Ethereum Blockchain and perform operations. Next step includes executing the deployed smart contract function with Web3.js without the involvement of Parity UI.

***Note:*** Calling a viewable smart contract function will not change the state of the Ethereum Blockchain but calling a payable function will create a transaction and updates the Ethereum Blockchain

The code to call a payable function **sendEther** described in example smart contract using Web3.js is following:

First of all create a transaction object which contains all the information about the transaction to be performed. The transaction is this case is executing a smart contract function **sendEther.**

web3.eth.getTransactionCount(devaccount, (err, txCount) =>

var txObject = {

    nonce: web3.utils.toHex(txCount),

    from:devaccount,

    to: contractAddress,

    data:mycontract.methods.sendEther(testac count).encodeABI(),

    gasLimit:web3.utils.toHex(web3.utils.toWei("0.00000000 00001", 'ether')),

    gasPrice:web3.utils.toHex(web3.utils.toWei("0.00000001 0", 'ether'))

}

**txObject** is a transaction object with transaction information to be performed. Other keywords and their functionalities are described below

* ***Nonce:*** Number of transactions sent from a given address. Transactions in Ethereum Blockchain are mined in a certain order and nonce allows to keep check of transactions sent from a specific address. A transaction with nonce of 3 cannot be mined before the transaction with nonce of 2. It also eradicates the possibility of double spending
* ***from:*** Takes the address of the caller of the function
* ***to:*** Takes the address of the smart contract
* ***data:*** Contains the command required to call a smart contract function and encodeABI() returns the function signature along with parameters
* **gasLimit:** Amount required to execute the function and carry out the transaction. Transaction will roll back if it runs out of gas
* **gasPrice:** Amount in units provided to run a transaction or smart contract on Ethereum Blockchain

***Note:*** The above transaction object is encapsulated in the **getTransactionCount** function. This returns the number of transactions (**txCount**)that have been sent from a particular address. In Ethereum Blockchain, ordering of transactions matter a lot and transactions are mined on the basis of their nonce values as discussed earlier. **txCount** is used in the nonce value while setting up transaction object **txObject**. getTransactionCount takes care of all this process and makes sure that transaction with same hash won’t be generated again due to different value of nonce every time. If nonce is not set up incrementally, the transaction will go the transaction queue and will remain a **pending Transaction** until its nonce number matches **txCount.**

After setting up the transaction object it is then initialized. Transaction still needs to be signed by deploying account along with private key

* privateKey=Buffer.from('4d5db4107d237df6a3d58ee5f70ae63d73d7658d4026f2eefd2f204c81682cb7', 'hex')
* tx = new Tx(txObject)
* tx.sign(privateKey)

Private Key must be buffered and is extracted from the account key file stored in **io.parity.ethereum** folder. Navigation to this folder and then developmentChain folder is described above in the document. Keys of accounts are stored in developmentChain folder inside **keys** folder. The key of a particular account can be accessed inside that folder. Private key of the account is then decrypted by using Web3.js by using the following command typed in **terminal**

* web3.eth.accounts.decrypt(Key Data, Account Password)

Key Data is located inside the key file of a particular account. Open the key file with text editor or other relevant application. **Key Data** takes the data inside the key file and **Account Password** takes the password of the particular account. After providing the mentioned information, run the command in terminal and output will provide a field name **privatekey**. Remove the **0x** at start of the private key and store the rest of the information in **privateKey** variable in code file in **hex** format.

***Note:*** Password set for the account explained in this document is set to an empty string

Transaction is then serialized, and raw transaction is created

* serializedTransaction = tx.serialize()
* raw = '0x' + serializedTransaction.toString('hex')

At the end, transaction is broadcasted using **sendSignedTransaction** on private network Ethereum Blockchain returning transaction hash in return

web3.eth.sendSignedTransaction(raw, (err, txHash)=>

{

     console.log('txHash', txHash)

})

# Subscribing to an event with Web3.js

Web3.js has some pre-configured events for Ethereum Blockchain and can be subscribed to. In addition, Web3.js also allows to subscribe to a custom event described in smart contract and this will be described in this section. It is assumed that the contract object is already configured with contract Abi and contract address.

As mentioned above an event named **Transfer** is described in example smart contract and can be subscribed using following lines on code

myevent= mycontract.events.Transfer( {

     address: contractAddress

}, (error, event) => {console.log(event);})

.on('changed', (event) => {

})

.on('error', console.error);

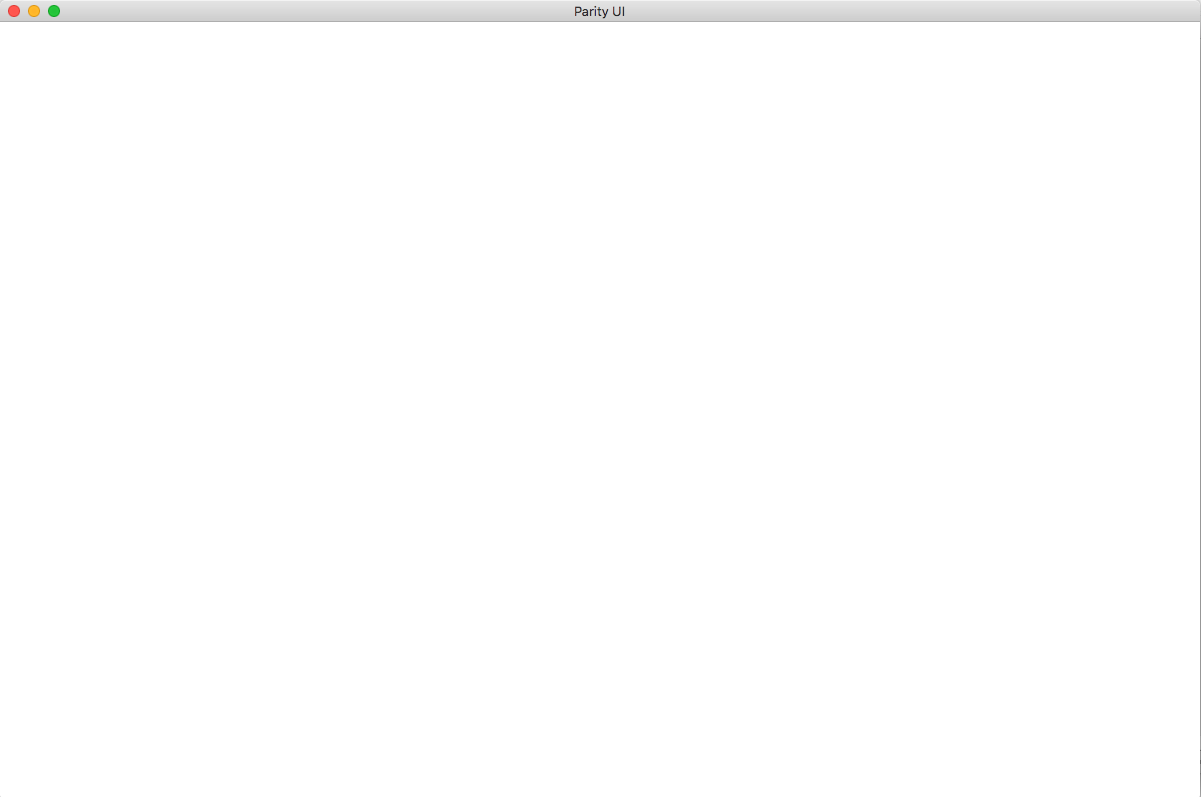
**myevent** variable takes the information about the event in smart contract. The custom events in smart contract can be subscribed by calling **[contract object].events.[name of event]** as mentioned in above code. After subscribing, Transfer event will notify the subscriber whenever manager of the smart contract will execute the sendEther function.

***Note:*** Further fields can be also be mentioned and configured in setting up a transaction object or subscribing to a smart contract event. However in the above examples only relevant and important fields are mentioned and discussed. Further information about Web3.js and its usage can be found [here](https://web3js.readthedocs.io/en/1.0/web3-eth-subscribe.html#subscribe)

# Pitfalls and Observations

Although Parity UI provides an interesting interface to interact with Ethereum Blockchain, but it has its limitations and bugs which have been observed so far

* Running parity in dev mode does not provide information about Ethereum Blockchain and Smart contracts deployed on **TestNet** network. TestNet is an Ethereum Blockchain explorer working exactly as Main Ethereum network but all transactions which are carried on TestNet network are fake and are only for testing purposes. So, no real ethers are actually transferred or consumed in TestNet environment. Parity UI does not work as a block explorer and does not provide any information about Ethereum Blockchain and transactions stored in Blockchain. Web3.js is used to query the Ethereum Blockchain running in dev mode.
* Parity UI doesn’t give any confirmation or error messages on sending invalid transactions or whenever a transaction fails. Instead the whole interface of Parity UI crashes and only white screen can be seen. It can be recovered on quitting the Parity UI and launching it again. The Parity UI will save the transaction as an incomplete transaction and it can be removed or sent again after correcting the transaction and providing the correct corresponding key for the account which is sending the transaction



*Figure 10: Parity UI Behavior on invalid transaction or transaction fail*

* Smart Contract compilation can only be done through certain number of compilers. Parity UI shows a list of compilers to compile a Smart Contract but most of them result in errors or compilation issues. This is done by hit and trial method until a correct solidity compiler is selected. The solidity compiler used in this example smart contract has version **0.4.22**