



Blockchain Basics: An Introduction to
Quorum[®], created by J.P. Morgan

Objective: Prepare yourself for this workshop, **Blockchain Basics: An Introduction to Quorum®**, created by J.P. Morgan where we will learn about Quorum, the blockchain, and how to create Smart Contracts to trade items in a digital marketplace.

About this Workshop

In this workshop you will teach participants about two things: the Quorum platform, and the Solidity language that can be used to write Smart Contracts for it. Quorum is an open source fork of the Ethereum blockchain based on the Go Ethereum implementation, developed by J.P. Morgan. The language Solidity was developed to create and execute these Smart Contracts.

Participants will come out of this workshop understanding major concepts of distributed ledger networks, how to create a Smart Contract, and how to run a Quorum app locally on their machine so that they can continue their learning later.

How to Prepare

The best way to prepare for this workshop is by familiarizing yourself with the slides and this presenter's guide. During the workshop you'll be able to use this guide as a reference of what to say and how to direct participants.

Before this workshop you should:

- Make sure you have Atom downloaded onto your computer, or a text editor of your choice. If you do not, Atom is available here: <https://atom.io>
- Install the necessary programs so that the day of you can concentrate on the participants. For this workshop, Docker, Node, and Truffle to run Quorum.
- Read over this guide to understand the fundamentals of distributed ledgers, the differences between Ethereum and Quorum, and the importance of Smart Contracts for future financial systems.
- Dig into the Quorum [wiki](#) a little bit!
- Attempt this workshop 2-3 times so that you feel confident and familiar with the material and equipped to help workshop participants. Be sure to reach out to localhost@mlh.io if you come across any parts where you need help when you're practicing.
- Make sure to customize any slides in the presentation that ask you to do so! Don't change any of the information on the other slides.

Introduction to Decentralized Ledgers

Objective: Help participants understand foundational ideas about the blockchain and Smart Contracts.

Activity: Ask the participants a theoretical question: if they were given fifty dollars, and they have an agreement to split the money with their neighbor, but their neighbor does not know how much they have, how much would they give?

But to complicate the scenario, the neighbor wants to make sure they are getting the right amount, so they find a **middle-man** and pay them five dollars to make sure that they are getting a fair share of the money.

After participants have a moment to think, then tell the participants a slightly different situation: they still have fifty dollars, but whenever they split it with any of their neighbors, this transaction is posted at the front of the room. With everyone in the room seeing the transaction, everyone can see it and check to make sure the deal is fair.

Try out a handful of splits. When a new transaction is posted to the front of the room, the old one is erased. Do this quickly!!

At the end, anyone in the room who can repeat back the right transactions can get a small prize, like some stickers.

This system makes it much harder to create unfair deals, and the second person does not need to pay a middle-man to check whether the amount they got was even.

Decentralized Ledgers

A **decentralized ledger** is a set of replicated databases, each containing the same list of past transactions. The database of transactions is held independently by each node (a computer in the network that has a copy of the blockchain). In the network there are nodes that re-run the same mathematical equations with the same inputs and confirm that the results match exactly before the transaction is added to the ledger.

Decentralized ledgers change dynamics of power. Instead of one person having control of a database and being able to say what is and what is not a fair trade, the entire room is there to verify that the money has been split evenly.

Understanding Blockchain

Many people use the term blockchain and decentralized ledger interchangeably, but there are important distinctions. Blockchains are one specific type of decentralized ledger. Data on a **blockchain** is grouped together and organized into blocks. The blocks are then linked together and secured using cryptography.

A blockchain is like a continuously growing list of records, stored chronologically and published publicly (usually). Each entry into this list is checked by many nodes on the network, ensuring that a majority of nodes have the same record of the transaction. This is how transactions are verified by a group rather than a single actor.

But what can you do with this kind of database? To understand this, we need to also understand Smart Contracts.

Introduction to Ethereum

Ethereum is a type of blockchain created specifically to run **Smart Contracts**. A Smart Contract is a protocol that digitally verifies and enforces a transaction. These contracts are trackable and irreversible. Smart Contracts self-execute without the need of a third party, which lowers the need for a lawyer or notary.

The combination of Smart Contracts within a system that verifies them makes it possible to create systems where people can conduct many different kinds of transactions in ways where the transactions are executed automatically and in a way that can be verified and in turn trusted. All of a sudden you do not have just one person making sure you split money evenly with your neighbor: you have everyone in the room.

Developers can use Ethereum to create whole new kinds of systems, which brings us to the practical applications of blockchains and Smart Contracts.

Decentralized Applications

Decentralized applications utilize the ideas of decentralized ledgers and smart contracts to create potentially revolutionary new systems. Today most applications are organized, maintained, and controlled by a single entity's server, and share information with people that request it. Decentralized applications work differently, on something called **peer-to-peer networks**, where the rights and responsibilities of creating and maintaining

an application are split amongst different actors in a group. This is why they're called **decentralized**.

The users of these networks have greater authority over changes to the network instead of a central body. In healthcare, a specific healthcare-centered **DApp** (decentralized application) could help hospitals around the world to store, access, and share patient records in a way that is secure and can't be tampered with. In politics, DApps have the potential to create more secure elections if each person's vote is added to a blockchain.

Introduction to Quorum

Quorum is one kind of permissioned blockchain. However, Quorum is different from Ethereum in three key ways: privacy, permissioning, and performance.

1. **Private transactions:** With a **privateFor** parameter, Quorum makes it so users do not have to reveal specifics on how much money is transferred between nodes.
2. **Network Permissioning:** Only verified nodes can connect to each other. This makes it less likely that nefarious actors can access information that they are not supposed to.
3. **Higher Performance:** Quorum uses RAFT and IBFT consensus algorithms instead of traditional Proof of Work algorithm which makes it much faster.

Quorum was built for the financial sector where privacy of transactions is incredibly important.

Putting It Together: The Transaction Life Cycle

Now that you and participants understand the general ideas behind Quorum and Ethereum, let's dive into what one full transaction looks like on the network.

A public transaction occurs the same way standard Ethereum Transactions do. There are additional steps for private transactions, which you can read about below:

Step 1: A Sender submits a transaction to their Quorum node (a machine connected to the Quorum network). The transaction's payload (the information about the transaction) should specify whether the transaction is private for specific parties.

Step 2: The Sender's Quorum node passes the transaction to their Transaction Manager.

Step 3: The Transaction Manager makes a call to its Enclave to validate the sender of the transaction.

Step 4: The Enclave validates and encrypts the transaction's payload.

Step 5: The Enclave performs the Transaction Conversion, a series of steps that returns necessary information to the Sender's Transaction Manager (including the Receiver's public key) .

Step 6: The Sender's Transaction Manager passes all of this information to the Receiver's node via HTTPS.

Step 7: The Receiver responds with an Ack or Nack response. If the response is Nack, the transaction is propagated to the network.

Step 8: A block is created containing the information created in previous steps and is distributed to all nodes.

Step 9: All nodes on the network will attempt to process the Transaction. However, only parties specified in the privateFor parameter are actually able to process it. Parties that can process it will make a call to their respective Enclaves and pass all of this information to it.

Step 10: The Enclave validates the signature and decrypts the Transaction Payload.

Step 11: The Enclave validates and encrypts the transaction's payload and returns it to the Transaction Manager.

Step 12: The Sender and Receiver's Transaction Managers send the payload to the EVM for contract execution.

Step 13: The Quorum node's state is updated, and the transaction is complete.

Building a Network

Objective: Use Quorum to build a TechMarketplace

Now we understand major ideas of distributed ledgers, smart contracts, and the differences between Quorum and Ethereum. The workshop will now teach participants how to create a run the Quorum network and use it to run a DApp.

The live version of the finished product is available here:

<http://mlhlocal.host/quorum-demo>

Setting Up Your Environment

The workshop assumes that participants already have the Atom text editor set up on their computers. If they do not, have them download it now: <https://atom.io>

We will also have to download three different packages in order to run the Quorum network.

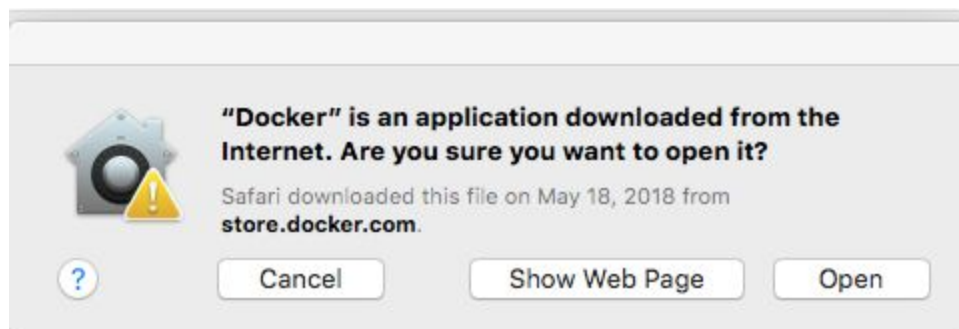
First Step: Docker

First we need to download **Docker**. Docker is a platform that helps developers build, ship, and run any application in any environment. Docker is really exciting for developers because it makes it easier to write one application and know that it will run the same on a friend's laptop, a major company's servers, or a Raspberry Pi. This makes product development a lot easier. Easier development means faster development time and in turn faster time to market before any competitors.

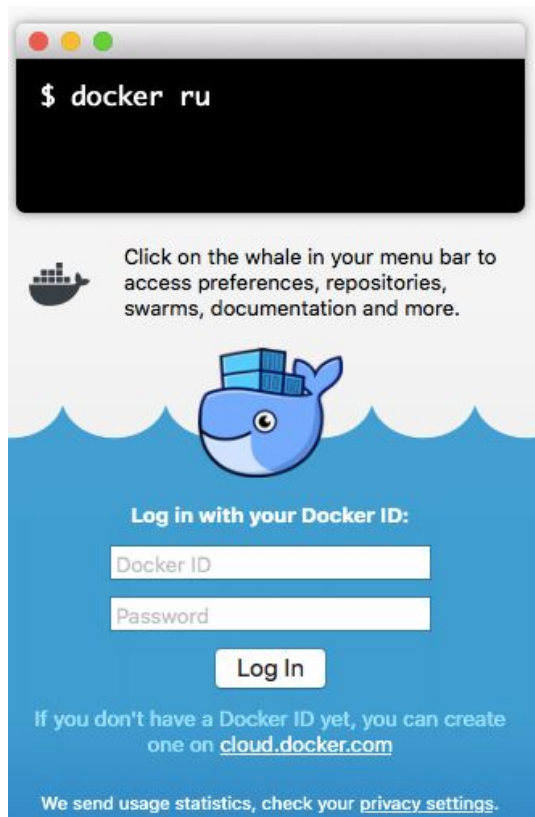
Docker for Mac: <https://docs.docker.com/docker-for-mac/install/>

Docker for Windows: <https://docs.docker.com/docker-for-windows/install/>

When Docker has downloaded, move it into your Applications folder, then open it.

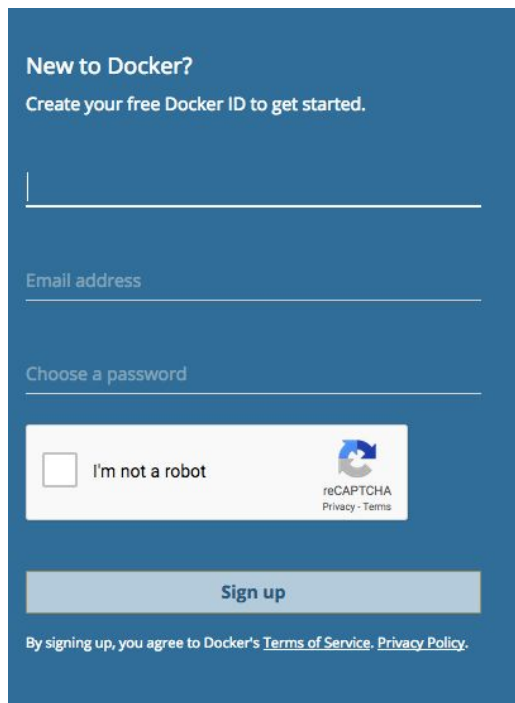


You will likely be asked for permission. Click **“Open.”**



A screenshot of a terminal window showing the command `$ docker ru`. Below the terminal, there is a message: "Click on the whale in your menu bar to access preferences, repositories, swarms, documentation and more." accompanied by a small ship icon. The Docker whale logo is prominently displayed in the center. Below the logo, the text "Log in with your Docker ID:" is followed by two input fields labeled "Docker ID" and "Password". A "Log In" button is positioned below these fields. At the bottom, a note states: "If you don't have a Docker ID yet, you can create one on [cloud.docker.com](\"http://cloud.docker.com\")". A footer line reads: "We send usage statistics, check your [privacy settings](\"#\")."

If you have a Docker ID already, log in now. If you do not, go to cloud.docker.com and sign up.



A screenshot of the Docker sign-up form. The header reads "New to Docker?" followed by "Create your free Docker ID to get started." Below this are three input fields: a blank one for the Docker ID, and two others labeled "Email address" and "Choose a password". A reCAPTCHA widget is present, featuring a checkbox labeled "I'm not a robot" and the reCAPTCHA logo with links to "Privacy" and "Terms". A "Sign up" button is located at the bottom of the form. A final line of text at the very bottom states: "By signing up, you agree to Docker's [Terms of Service](\"#\"), [Privacy Policy](\"#\")."

Choose an ID and a password. You will then need to verify the your account through your email. Next, log in to your new Docker account.

Second Step: Install Node

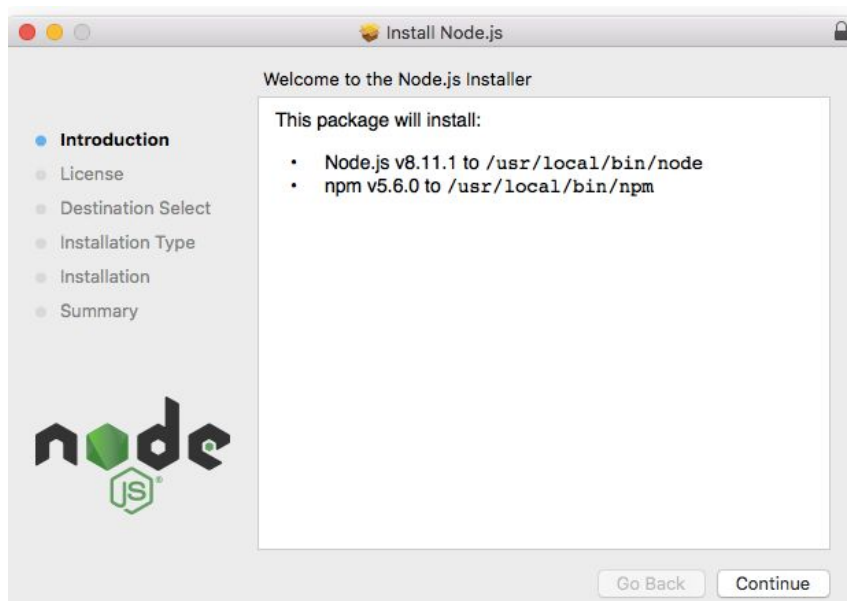
You will need to install Node to run the TechMarketplace application locally. Node is a Javascript runtime. It allows participants to run Javascript applications.

Node for Mac: <http://mlhlocal.host/node-mac>

Node for PC: <http://mlhlocal.host/node-pc>

Node for Linux: <http://mlhlocal.host/node-linux>

You will see this installation window. Follow the instructions.



To check whether Node is running, type the command `node --version` into your terminal:

```
node --version
v8.11.2
```

If you get back anything less than **v8.0.0**, try restarting your terminal or making sure that you are downloading the current version of Node.JS. You should have version **v8.11.2** at the time of this writing.

Install Truffle

Truffle is a development framework for Ethereum.

We will run a few Truffle commands later to get the Quorum network up and running, but for now all we need to do is install it.

Type **npm install -g truffle** into your Terminal.

```
npm install -g truffle
+ truffle@4.1.5
updated 1 package in 6.122s
```

NPM stands for **Node Package Manager**. We are using NPM to install our new package, Truffle. NPM allows developers to create dynamic pages on the server before sending it to a user's web page.

The command **-g** means Global. We are installing Truffle across your entire computer and not just this directory.

Install Tech Marketplace

The final component of our set up is to download the Tech Marketplace source code. Go to this url: <http://mlhlocal.host/quorum-app>

A zip file of the code will be automatically downloaded. Unzip the file and open it in your text editor.

To do so, type the following commands into your Terminal or PowerShell:

```
cd Downloads\mlh-localhost-tech-
marketplace-master

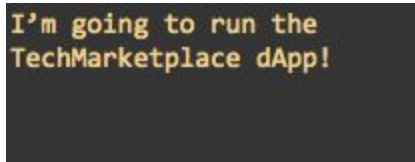
npm install
## takes 2-3 minutes
```

These commands use the Node Package Manager to install the Marketplace DApp.

Setting Up the Environment

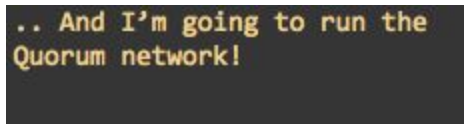
From here on out you should have two terminal windows open. You will need to migrate back and forth between them.

The first terminal window will be running the Tech Marketplace.



```
I'm going to run the  
TechMarketplace dApp!
```

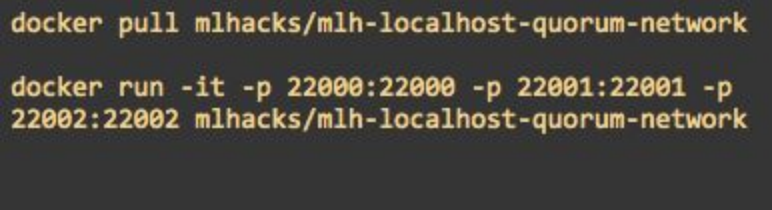
The second one will be running the Quorum network.



```
.. And I'm going to run the  
Quorum network!
```

Run the Quorum Network

In your Quorum network window, type the commands below:



```
docker pull mlhacks/mlh-localhost-quorum-network  
  
docker run -it -p 22000:22000 -p 22001:22001 -p  
22002:22002 mlhacks/mlh-localhost-quorum-network
```

This code retrieves the Quorum code from the Docker repository and begins the network with several nodes.

Then participants will need to initialize the quorum accounts and keystores of the initial nodes in the network. Type the command `./raft-init.sh` into the terminal.

```
./raft-init.sh

[*] Cleaning up temporary data directories
[*] Configuring node 1 (permissioned)
INFO [04-03|04:46:01] Allocated cache and file handles
database=/home/vagrant/quorum-network/qdata/dd1/g
eth/chaindata cache=16 handles=16
. . .
```

After the nodes are initialized, they need to be launched. Next, type in `./raft-start.sh` which starts the nodes.

```
./raft-start.sh

[*] Starting Constellation nodes
constellation-node --url=https://127.0.0.1:9001/
--port=9001 --workdir=qdata/c1 --socket=tm.ipc
--publickeys=tm.pub --privatekeys=tm.key
--othernodes=https://127.0.0.1:9001/ >>
qdata/logs/constellation1.log 2>&1
. . .
```

Compile and Migrate Contracts

Return to the other terminal (the TechMarketplace window). Type `truffle compile`, wait, and then `truffle migrate` into the console. These two commands will compile the `Market.sol` and the `Migrate.sol` folders. This will migrate the contracts running on your computer to the Quorum network.

```

truffle compile

Compiling ./contracts/Market.sol...
Compiling ./contracts/Migrations.sol...
Writing artifacts to ./build/contracts

truffle migrate
Using network 'development'.

Running migration: 1_initial_migration.js
Deploying Migrations...

```

Now you must run the seed file. Type `npm run seed` into your window. This command runs the seed file, which populates the marketplace from the `items.json` file.

```

npm run seed

truffle exec scripts/seed.js --network
development

Using network 'development'.

[ [ BigNumber { s: 1, e: 0, c: [Array] },
  '0x0000000000000000000000000000000000000000',

```

You start the application then by typing `npm start` into the console. This starts the local application.

```

npm start

truffle exec scripts/seed.js --network
development

Using network 'development'.

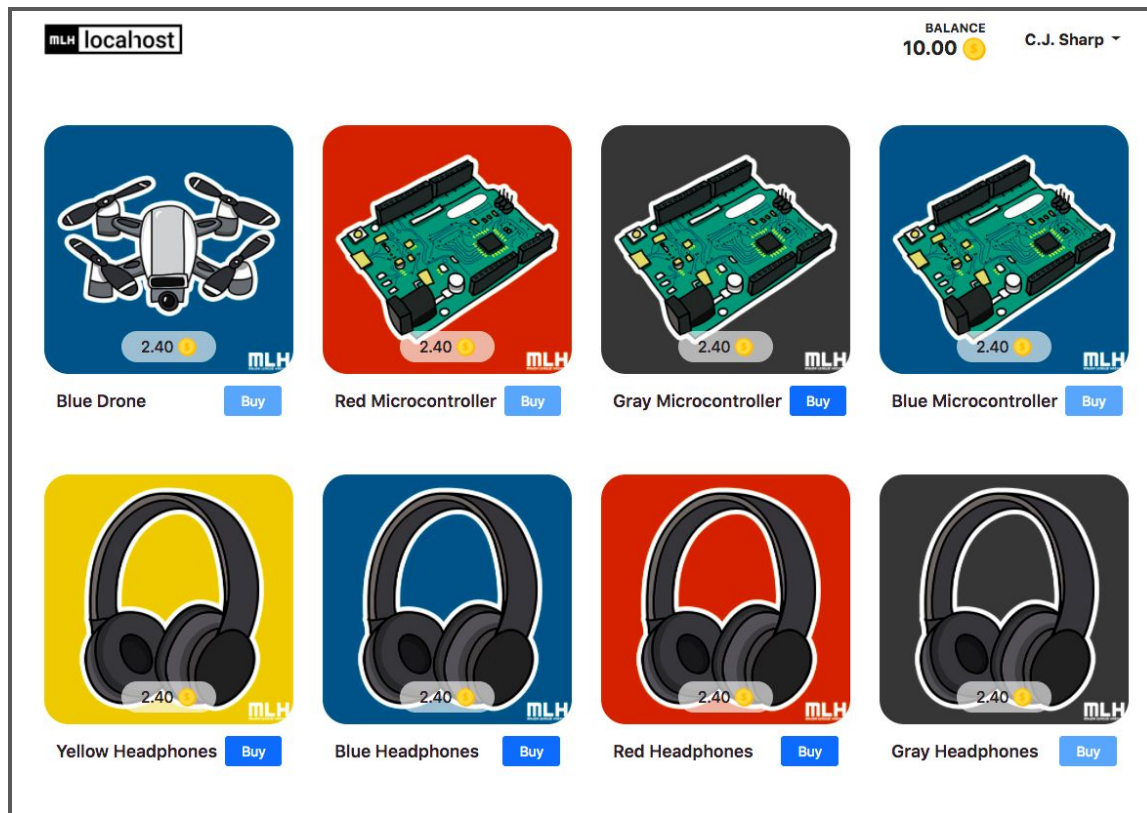
[ [ BigNumber { s: 1, e: 0, c: [Array] },
  '0x0000000000000000000000000000000000000000',

```

First Test

Check out the TechMarketplace. You can see the finished product by opening a browser window and navigating to `localhost:3000`.

Your page should look something like this:



Give participants a minute or two in order to dive into the web page themselves and understand what does and does not work. They will soon notice that the Sell function does not work. In order to fix it, we will need to understand how to create a Smart Contract.

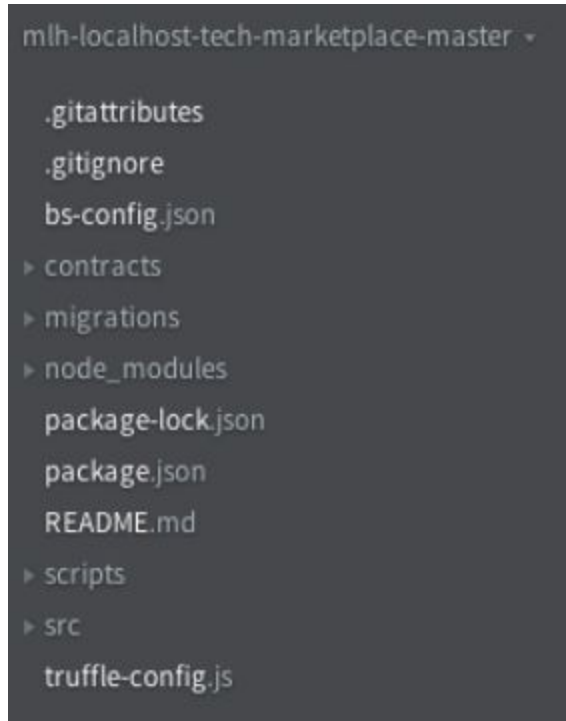
To do this, we'll need to stop your local environment. In the tab where you ran the npm commands, type **[CTRL] C**. Then press enter. This stops the local server from running.



To create a Smart Contract participants need to understand the language they run on, Solidity.

To get started, go to the [mlh-localhost-tech-marketplace-master](http://mlhlocalhost-tech-marketplace-master) folder that you downloaded from this link: <http://mlhlocal.host/quorum-app>

This folder should have several different folders.



Click on the folder called [contracts/](#). It has two files in it: [Market.sol](#) and [Migrations.sol](#). The `.sol` ending means that the code has been written in Solidity.

Introducing Solidity

A Smart Contract both sets up the terms of the agreement (like a transaction) and also enforces those terms. They are a necessary part of completing transactions on the Quorum network.

Solidity is a statically typed language used to create Smart Contracts. It is called a “contract-oriented” programming language. It was written specifically for making Smart Contracts. Solidity is influenced by C++, Java, and Python. To read more: mlhlocal.host/solidity-docs

You’ll start your participants out with the traditional function called [HelloWorld](#). Your participants do not need to code this; you are simply using this to explain the basic syntax of a Solidity function.

```

01 pragma solidity ^0.4.11;
02
03 contract helloWorld {
04     function renderHelloWorld() returns (string) {
05         return 'Hello World';
06     }
07 }
08

```

Line 1 specifies the version of Solidity used. Line 3 creates a new Smart Contract called `helloWorld`. Line 4 declares a function called `renderHelloWorld` that takes no parameters and returns a string. Line 5 returns the string "Hello World."

Data Types

In order to create Smart Contracts you need to understand four different data types. There are `addresses` that are needed for transactions - an address is like an account number. There are traditional strings. There are also `uints`. These are unsigned integers up to 256 bits. These integers will be used later to demarcate what prices items in the marketplace sell for.

There are `strings` that are data types that contain text. Finally there is the `struct` that is a custom-defined type that can include other data types listed here.

Using Smart Contracts

Let's put all of the lessons we just learned about smart contracts into practice by understanding the `Market.sol` contract. Line 3 of this function declares a new contract named `Market`. In Line 4 you create an array with an `Item` in it that can in turn include multiple things, such as an image or price. Line 5 is then given an integer that acts as an address to send it to, and a person who owns that address.

The beginning of the contract looks like this:

```

03 contract Market {
04     Item[] public items;
05     mapping(address => uint256) public balances;
06
07     struct Item {
08         address owner;
09         string name;
10         string image;
11         uint256 price;
12         string nickname;
13     }

```


On Line 7, a custom object, a **struct**, is created called Item. To compare this to typical online shopping, the person is posting a thing to sell. While on Ebay or Amazon this thing could be anything from a baseball to a book, our example uses tech merchandise.

On Line 8, a variable called "owner" is created of the type "address." This address identifies the owner of the item. On Line 9, the item is given a variable named "name" that is a type "string" (this means plain text). In the online shopping metaphor, the participant is creating a Seller profile.

Line 10 gives each item a variable named Image of type string. This allows a participant to upload an image to showcase the item they are selling. Line 11 gives each item a unit256 (a number) that will be the item's price. Line 12 gives the item a text (string) nickname.

```
15  function Market(address nodeA, address nodeB, address nodeC) public {
16      balances[nodeA] = 10 ether;
17      balances[nodeB] = 10 ether;
18      balances[nodeC] = 10 ether;
19  }
20
```

Line 15 creates a public function called **Market()** that takes three addresses called **nodeA**, **nodeB**, and **nodeC**.

Lines 16 through 18 gives each node (participant) a balance of 10 ether that they can now use to buy and sell items on the market.

```
31  function buyItem(uint id) public returns (uint) {
32      require(id >= 0 && id <= items.length);
33      require(msg.sender != items[id].owner);
34      require(balances[msg.sender] >= items[id].price);
35
36      balances[msg.sender] -= items[id].price;
37      items[id].owner = msg.sender;
38
39      return id;
40  }
```

Line 31 declares a public function **buyItem()** that takes a **uint** called **id** and returns a **uint**. Line 32 requires that **id** be greater than or equal to zero. Line 33 requires that the sender (the person buying the item) is not the same as the owner of the item. Line 34 requires the balance of the message sender be greater than or equal to the price of the item being bought.

Line 36 subtracts the cost of the item from the message sender's balance. Line 37 changes the owner of the item to be the message sender. Line 39 returns the `id` of the item purchased.

Last Steps

Now we're going to put all of the pieces we've been learning together into a complete transaction called the `sellItem()` function. We are going to see whether the seller of the item is the owner of the item, add the price of the item to the sender's wallet, and then delete the id of the item's current owner.

The code will look like this:

```
42 function sellItem(uint id) public returns (uint) {
43     require(id >= 0 && id <= items.length);
44     require(msg.sender == items[id].owner);
45
46     balances[msg.sender] += items[id].price;
47     delete items[id].owner;
48
49     return id;
50 }
```

Give participants a few minutes to try to solve this problem on their own before revealing the solution.

Let's go ahead and run the application.

Type these commands into your terminal:

```
pwd
mlh-localhost-tech-marketplace-master/
truffle compile
truffle migrate --reset
npm run seed
npm start
```

You should be able to verify that the message sender is the owner of the item, add the price of the item to the message sender's wallet, and delete the owner of the item.

Circling Back

Today participants learned about distributed ledgers and Smart Contracts that allow people to conduct automatic, irreversible trades that are entered into a shared database and verified by the other actors in the network.

We learned about Quorum, a blockchain network created for private financial transactions, and the language of Solidity, used to create Smart Contracts.

To connect the work done back to larger concepts and to make sure this knowledge sticks, provide your participants a follow-up quiz: <http://mlhlocal.host/quiz>

Next Steps

You might want to wrap up the workshop by asking participants whether this workshop has changed their opinion at all about how blockchain and Smart Contracts can be utilized in the financial sector. What are some other sectors they might want to see decentralized applications used in?

If participants are curious about the intricacies of Quorum, there is a Quorum wiki: <https://github.com/jpmorganchase/quorum/wiki>. If they are feeling very confident, they can even submit issues they find to the Quorum repo: <https://github.com/jpmorganchase/quorum-examples>.

Encourage your participants to take their new skills and keep learning, because these ideas are complicated. There are many emerging jobs related to Quorum, and so going further with their projects might pay off!