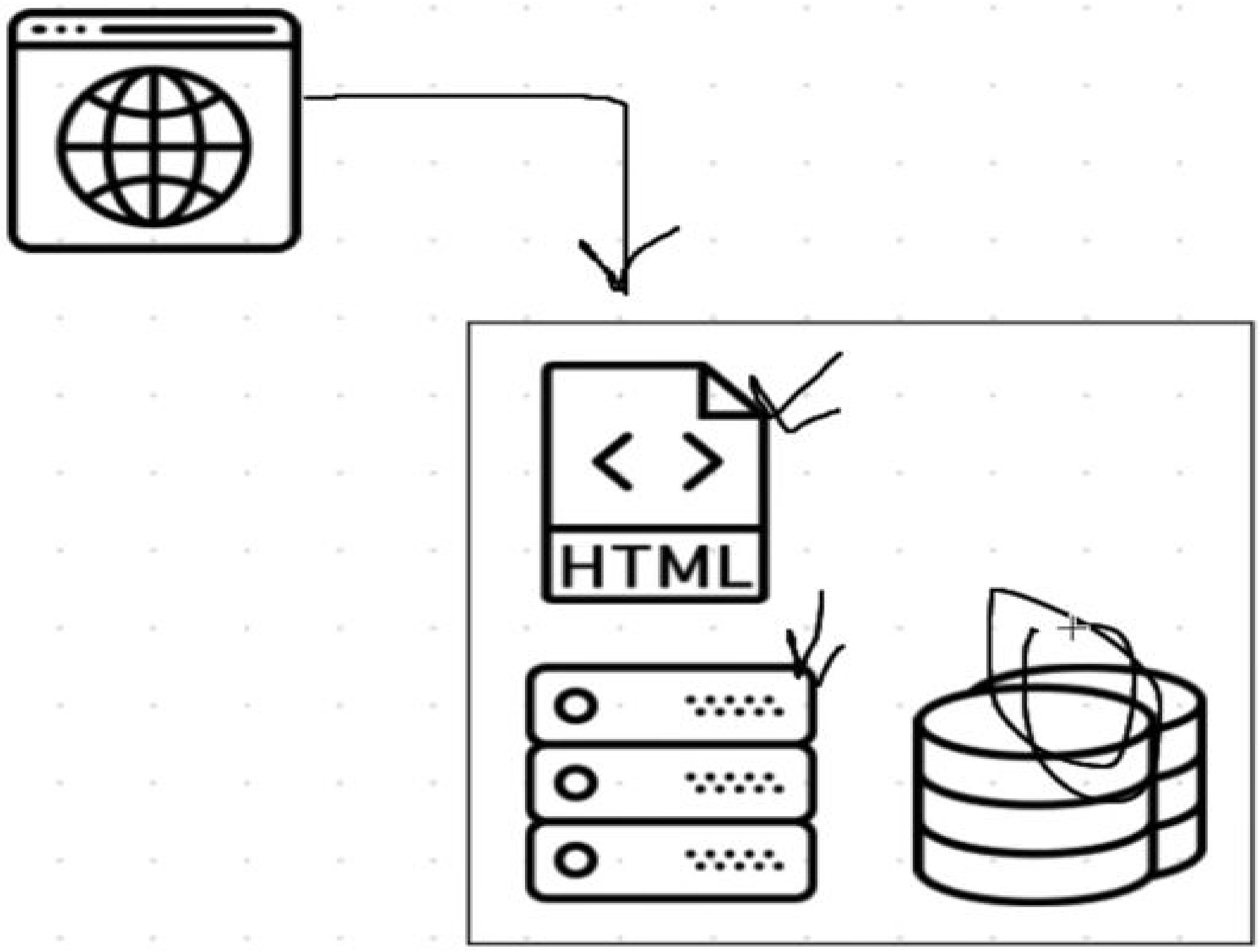
Build Blockchain App

Ethereum Todo List 2019

Todo list in the web application

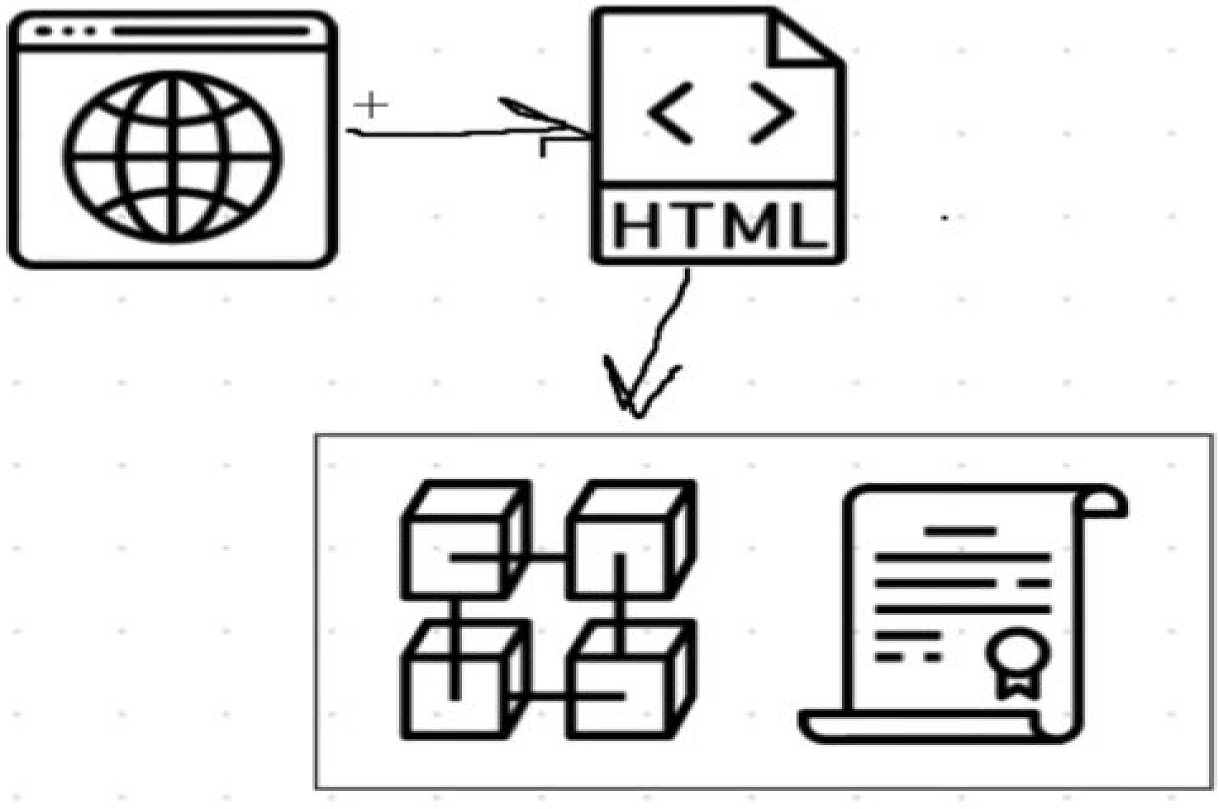
- To access the todo list, you would use a web browser that would communicate with a web server over the Internet. The server contains all of the code and data for the todo list.



- Here is a list of what you would find on the server:
    - Client side files in HTML, CSS, and JavaScript
    - Back end code responsible for the application's business logic
    - Database that stores the tasks in the todo list
- This server is a centralized entity that has full control over every aspect of the application. Anyone with full access to the server can change any part of the code or the data at any time.

Todo list in the blockchain application

- A blockchain application works quite differently. All of the code and the data to the todo list does not lie on a centralized server. Instead, it is distributed across the blockchain. All of the code and the data is shared and unchangeable on the blockchain.



- To access the blockchain todo list, we'll use a web browser to talk to the client side application, which will be written in HTML, CSS, and JavaScript. Instead of talking to a back end web server, the client side application will talk directly to the blockchain. In blockchain, it contains all the business login written in Ethereum Smart Contract for the todo list and all the todo list is stored in the blockchain itself.

Q. What is blockchain ??

Ans: A blockchain is a peer-to-peer network of computers, or nodes, that talk to one another. It's a distributed network where all of the participants share the responsibility of running the network. Each network participant maintains a copy of the code and the data on the blockchain. All of this data is contained in bundles of records called "**blocks**" which are "**chained together**" to make up the blockchain. All of the nodes on the network ensure that this data is secure and unchangeable, unlike a centralized application where the code and data can be changed at any time. That's what makes the blockchain so powerful!
Because the blockchain is responsible for storing data, it fundamentally is a database. And because it's a network of computers that talk to one another, it's a network.
We can think of it as a network and a database all in one.

Q. What is a Smart Contract ??

Ans: All of the code on the blockchain is contained in smart contracts, which are programs that run on the blockchain. They are the building blocks of blockchain applications.
Smart contracts are written in a programming language called Solidity, which looks a lot like JavaScript. All of the code in the smart contract is immutable, or unchangeable. Once we deploy the smart contract to the blockchain, we won't be able to change or update any of the code. This is a design feature that ensures that the code is trustless and secure.
They act as an interface for reading and writing data from the blockchain, as well as executing business logic. They're publicly accessible, meaning anyone with access to the blockchain can access their interface.

Q. How Blockchain Todo List Works ?

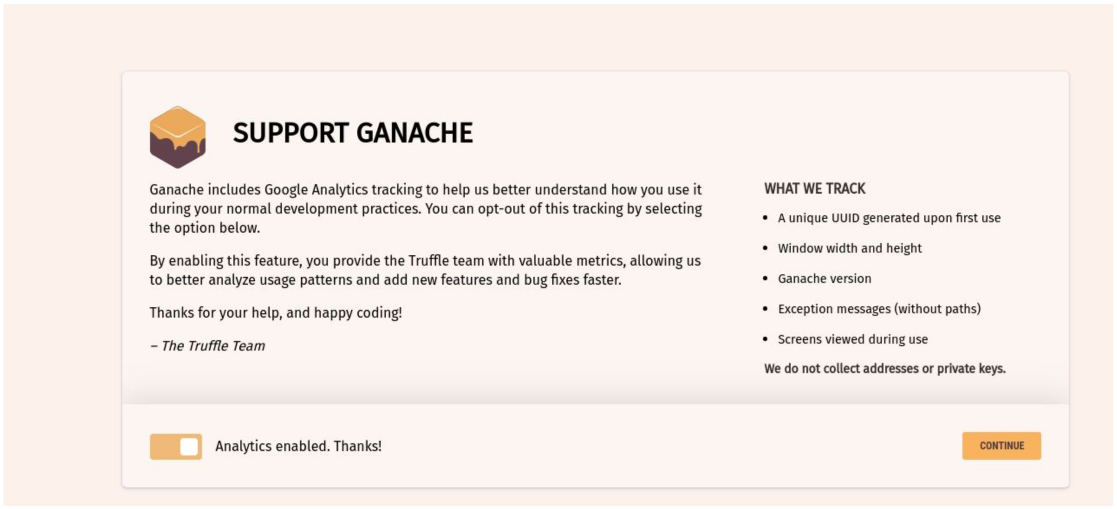
- a client side application for the todo list that will talk directly to the blockchain.
- the Ethereum blockchain, which we can access by connecting our client side application to a single Ethereum node.
- a smart contract in Solidity that powers the todo list.
- deploy it to the Ethereum blockchain.
- connect to the blockchain network with our personal account using an Ethereum wallet in order to interact with the todo list application.

Installing Dependencies

Install all of the dependencies we need to build our project.

Install Ganache

- In Ubuntu, open a browser and navigate to https://github.com/trufflesuite/ganache/releases
- Download the latest Linux release which will be the *.AppImage file. For example ganache-1.3.0-x86_64.AppImage.
- Once the download is complete, open a new terminal and change into the directory with the *.AppImage file.
- Use chmod to make the file executable:
```
chmod a+x ganache-1.3.0-x86_64.AppImage
```
- Now run the file
```
./ganache-1.3.0-x86_64.AppImage
```
- You will be prompted if you want to integrate the application into your system. For convenience, click Yes. This will allow you to launch Ganache later from Ubuntu Application menu.
- Ganache will launch and prompt if you want to enable Google Analytics tracking to help the developers improve the software. Toggle this off if you wish, then click Continue.


SUPPORT GANACHE

Ganache includes Google Analytics tracking to help us better understand how you use it during your normal development practices. You can opt-out of this tracking by selecting the option below.

By enabling this feature, you provide the Truffle team with valuable metrics, allowing us to better analyze usage patterns and add new features and bug fixes faster.

Thanks for your help, and happy coding!

– The Truffle Team

WHAT WE TRACK

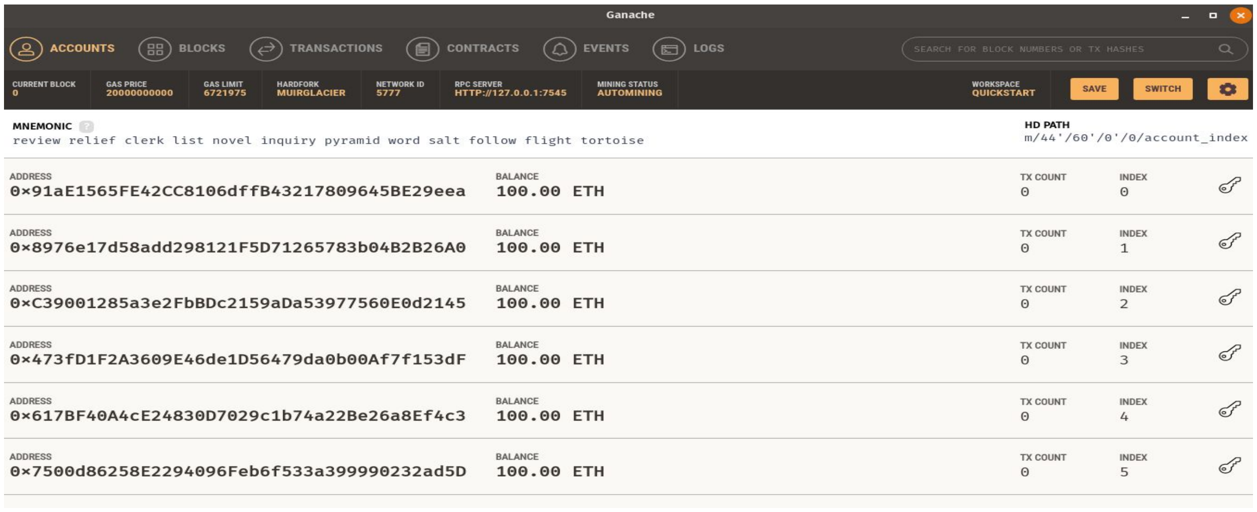
- A unique UUID generated upon first use
- Window width and height
- Ganache version
- Exception messages (without paths)
- Screens viewed during use

We do not collect addresses or private keys.

Analytics enabled. Thanks!

- The main Ganache window will open and you will notice there are already a number of addresses with a balance of 100 ETH each. Ganache provides a personal blockchain that you can start developing against immediately.

- The main Ganache window will open and you will notice there are already a number of addresses with a balance of 100 ETH each. Ganache provides a personal blockchain that you can start developing against immediately.



CURRENT BLOCK	GAS PRICE	GAS LIMIT	HARDFORK	NETWORK ID	RPC SERVER	MINING STATUS	WORKSPACE
0	20000000000	6721975	MUIRGLACIER	5777	HTTP://127.0.0.1:7545	AUTOMINING	QUICKSTART

MNEMONIC
review relief clerk list novel inquiry pyramid word salt follow flight tortoise

HD PATH
m/44'/60'/0'/0/account_index

ADDRESS	BALANCE	TX COUNT	INDEX
0×91aE1565FE42CC8106dffB43217809645BE29eea	100.00 ETH	0	0
0×8976e17d58add298121F5D71265783b04B2B26A0	100.00 ETH	0	1
0×C39001285a3e2FbBDc2159aDa53977560E0d2145	100.00 ETH	0	2
0×473fD1F2A3609E46de1D56479da0b00Af7f153dF	100.00 ETH	0	3
0×617BF40A4cE24830D7029c1b74a22Be26a8Ef4c3	100.00 ETH	0	4
0×7500d86258E2294096Feb6f533a399990232ad5D	100.00 ETH	0	5

Install NodeJS and npm

Install NodeJS and it’s package manager, npm:

```
sudo apt install nodejs npm
```

Install Tuffle

- Once npm is install, you can install Truffle with this command:

```
npm install -g truffle
```

- Verify the installation by checking the truffle version:

```
truffle version
```

Install Metamask Chrome Extension

To turn your web browser into a blockchain browser. Most major web browsers do not currently connect to blockchain networks, so we'll have to install a browser extension that allows them to do this.

Link:
https://chrome.google.com/webstore/detail/metamask/nkbihfbeogaeaoehlefnkodbefgpgknn?hl=en
Metamask will also allow us to manage our personal account when we connect to the blockchain, as well as manage our Ether funds that we'll need to pay for transactions.

Project Setup

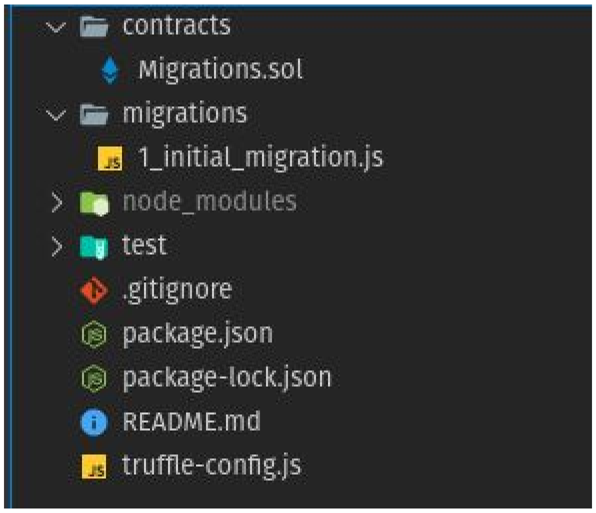
- create a project directory and navigate into the directory
- Now we initialize a new truffle project to develop our project like this:

```
$ truffle init
```

- The new project will include some folders and files:
    - contracts
    - migrations
    - test
    - truffle-config.js
- You can bootstrap all of the dependencies for your project by simply copy-and-pasting the code below into your **package.json** file:
- Now you can install the dependencies from the command line like this:

```
$ npm install
```

- Now that the dependencies are installed, the project directory structure that we just created:



```
contracts
    Migrations.sol
migrations
    1_initial_migration.js
node_modules
test
.gitignore
package.json
package-lock.json
README.md
truffle-config.js
```

- **contracts directory**: this is where all smart contacts live. We already have a Migration contract that handles our migrations to the blockchain.
- **migrations directory**: this is where all of the migration files live. These migrations are similar to other web development frameworks that require migrations to change the state of a database. Whenever we deploy smart contracts to the blockchain, we are updating the blockchain's state, and therefore need a migration.
- **node_modules directory**: this is the home of all of our Node dependencies we just installed.
- **test directory**: this is where we'll write our tests for our smart contract.
- **truffle-config.js** file: this is the main configuration file for our Truffle project, where we'll handle things like network configuration.

Let’s Code …

- Now let's start developing the smart contract that will manage our todo list. We can do this by creating a new file in the contracts directory like this:
```
$ touch ./contracts/TodoList.sol
```
- Inside here, let's develop our todo list smart contract. First, we'll start by specifying the version like this:
```
pragma solidity ^0.5.0;
```
- Now we can declare the smart contract like this:
```
pragma solidity ^0.5.0;
contract TodoList {
// Code goes here…
}
```

We create a smart contract called TodoList followed by curly braces. We'll add all of the code for the smart contract inside of them. The thing we'll do is just keep track of the number of tasks inside the todo list. This will allow us to write some simple code that will help us ensure that the project is set up properly, and that our code is working on the blockchain.

Compile and Run the application

- let's compile the smart contract

  ```
  $ truffle compile
  ```

  You should notice that a new file was generated whenever you compiled the smart contract at the following path: `./build/contracts/TodoList.json`. This file is the smart contract ABI file, which stands for "Abstract Binary Interface". This file has many responsibilities, but two that I will highlight here:

  - It contains the compiled bytecode version of the Solidity smart contract code that can be run on the Ethereum Virtual Machine (EVM), i.e., an Ethereum Node.
  - It contains a JSON representation of the smart contract functions that can be exposed to external clients, like client-side JavaScript applications.

To talk to the smart contract on the personal blockchain network inside the Truffle console, we must do a few things:

- Update our project's configuration file to specify the personal blockchain network we want to connect to (Ganache).
- Create a migration script that tells Truffle how to deploy the smart contract to the personal blockchain network.
- Run the newly created migration script, deploying the smart contract to the personal blockchain network.

First, we'll update the project configuration file to specify the personal blockchain network we want set up in the first section. Find the file truffle-config.js and paste the code here.

Note: these should match the default settings provided by the Ganache personal blockchain network. If you changed any settings inside the Ganache settings page, like the port, those should be reflected here.

- Next, we'll create a migration script inside the migrations directory to deploy the smart contract to the personal blockchain network.

  Q. What does this file do?

  Any time we create a new smart contract, we are updating the state of the blockchain. Remember, that a blockchain fundamentally is a **database**. Hence, whenever we permanently change it, we must migrate it from one state to another. This is very similar to a database migration that you might have performed in other web application development frameworks.

Notice that we number all of our files inside the migrations directory with numbers so that Truffle knows which order to execute them in. Inside this newly created migration file, you can use this code to deploy the smart contract.

Let's run this migration script from the command line like this:

```
$ truffle migrate
```

We have successfully migrated the smart contract to the personal blockchain network.

List Tasks

Let's start listing out the tasks in the todo list. Here are all of the steps to list the tasks.

- Write code to list tasks in the smart contract
- List tasks in the client side application
- Write a test for listing tasks

In order to list the tasks inside the smart contract, we'll need a way to model a task in solidity. Solidity allows you to define your own data types with structs. We can model any arbitrary data with this powerful feature. We'll use a struct to model the task for our todo list like this:

```
// 1. Version of solidity Programming language that we want to use
pragma solidity ^0.5.0;
// 2. Declare the smart contract
contract TodoList {
      uint public taskCount = 0; //State

      struct Task {
        uint id;
        string content;
        bool  completed;
       }
}
```

Now that we've modeled a task, we need a place to put all of the tasks in the todo list! We want to put them in storage on the blockchain so that the state of the smart contract will be persistent.

A mapping in Solidity is a lot like an associative array or a hash in other programming languages. It creates key-value pairs that get stored on the blockchain. We'll use a unique id as the key. The value will be the task itself.

Create a function for creating tasks. This will allow us to add new tasks to the todo list by default so that we can list them out in the console. To add one task to the todo list whenever the smart contract is deployed to the blockchain so that it will have a default task that we can inspect in the console. We can do this by calling the createTask() function inside of the smart contract's constructor function like this:

```
// Smart Contract for Todo-List

// 1. Version of solidity Programming language that we want to use
pragma solidity ^0.5.0;
```

```
// 2. Declare the smart contract
contract TodoList {
 uint public taskCount = 0; //State

 struct Task {
   uint id;
   string content;
   bool  completed;
 }

 // Storage
 mapping(uint => Task) public tasks;

 // Constructor fn to initialize the task
 constructor() public {
   createTask("Complete BCT Notes website new features");
   createTask("Complete pending work");
 }

 function createTask(string memory _content) public {
   taskCount ++;
   tasks[taskCount] = Task(taskCount, _content, false);
 }

}
```

HTML Code

Sample HTML code for our todo list in **src** folder.
Sample Code
The file pulls in all of the dependencies for the project like the bootstrap templating framework that will allow us to create nice-looking UI elements without having to write too much CSS. It also uses the Truffle Conract library that will allow us to interact with the todo list smart contract with JavaScript.

In the JavaScript code to list the todo, We'll add code to the newly created app.js file.

```
App = {
 loading: false,
 contracts: {},

 load: async () => {
   // load the application
   console.log("app loading ...");
   await App.loadWeb3();
   await App.loadAccount();
   await App.loadContract();
   await App.render();
 },

//https://medium.com/metamask/https-medium-com-metamask-breaking-change-in
jecting-web3-7722797916a8
 loadWeb3: async () => {
   if (typeof web3 !== "undefined") {
     App.web3Provider = web3.currentProvider;
     web3 = new Web3(web3.currentProvider);
   } else {
     window.alert("Please connect to Metamask.");
   }
   // Modern dapp browsers...
   if (window.ethereum) {
     window.web3 = new Web3(ethereum);
     try {
       // Request account access if needed
       await ethereum.enable();
       // Accounts now exposed
       web3.eth.sendTransaction({
         /* ... */
       });
     } catch (error) {
       // User denied account access...
     }
   }
   // Legacy dapp browsers...
   else if (window.web3) {
     App.web3Provider = web3.currentProvider;
     window.web3 = new Web3(web3.currentProvider);
     // Accounts always exposed
     web3.eth.sendTransaction({
       /* ... */
     });
   }
   // Non-dapp browsers...
   else {
     console.log(
       "Non-Ethereum browser detected. You should consider trying
MetaMask!"
     );
   }
 },

 loadAccount: async () => {
   App.account = web3.eth.accounts[0];
 },

 loadContract: async () => {
   const todoList = await $.getJSON("TodoList.json");
   App.contracts.TodoList = TruffleContract(todoList);
   App.contracts.TodoList.setProvider(App.web3Provider);
   App.todoList = await App.contracts.TodoList.deployed();
 },
 render: async () => {
   if (App.loading) {
     return;
   }

   $("#account").html(App.account);
   // Render Task
   await App.renderTask();

   App.setLoading(false);
 },

 renderTask: async () => {
   // Load the total task count from the blockchain
   const taskCount = await App.todoList.taskCount();
   // Render out each task with a new template
   const $taskTemplate = $(".taskTemplate");
   for (let i = 1; i <= taskCount; i++) {
     // Fetch the task data from the blockchain
     const task = await App.todoList.tasks(i);
     const taskId = task[0].toNumber();
     const taskContent = task[1];
     const taskCompleted = task[2];
     // TASK HTML
     const $newTaskTemplate = $taskTemplate.clone();
     $newTaskTemplate.find(".content").html(taskContent);
     $newTaskTemplate
       .find("input")
       .prop("name", taskId)
       .prop("checked", taskCompleted)
       .on("click", App.toggleComplete);
     // Filter task acc to status
     if (taskCompleted) {
       $("#completedTaskList").append($newTaskTemplate);
     } else {
      $("#taskList").append($newTaskTemplate);
    }
    // Show the task
    $newTaskTemplate.show();
  }
 },

 setLoading: (boolean) => {
   App.loading = boolean;
   const loader = $("#loader");
   const content = $("#content");
   if (boolean) {
     loader.show();
     content.hide();
   } else {
     loader.hide();
     content.show();
   }
 },
};

$(() => {
 $(window).load(() => {
   App.load();
 });
});
```

We have created a new App object that contains all the functions we need to run the JavaScript app. The explanation of important functions:

- `loadWeb3()` web3.js is a JavaScript library that allows our client-side application to talk to the blockchain. We configure web3 here. This is default web3 configuration specified by Metamask. Do not worry if you don't completely understand what is happening here. This is a copy-and-paste implementation that Metamask suggests.
- `loadContract()` This is where we load the smart contract data from the blockchain. We create a JavaScript representation of the smart contract with the Truffle Contract library. Then we load the smart contract data with web3. This will allow us to list the tasks in the todo list.

- `renderTasks()` This is where we actually list the tasks in the todo list. Notice that we create a for loop to access each task individually. That is because we cannot fetch the entire task mapping from the smart contract. We must first determine the taskCount and fetch each task one-by-one.

Start Web Server

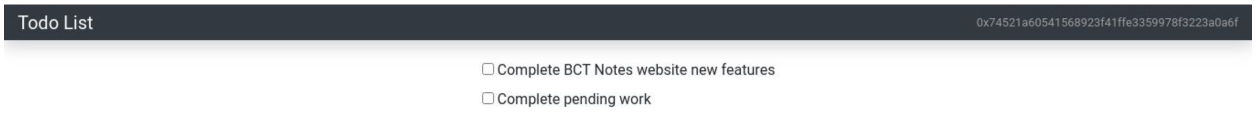
Start the web server and ensure that the project will load in the browser.

```
$ npm run dev
```

Yay! 🎉 You have successfully loaded the client side application. 😃

Notice that your application says **"Loading..."**. That's because we're not logged in to the blockchain yet! In order to connect to the blockchain, we need to import one of the accounts from Ganache into Metamask.

Once you're connected with Metamask, you should see all of the contract and account data loaded.



Todo List 0x74521a60541568923f41ffe3359978f3223a0a6f

☐ Complete BCT Notes website new features

☐ Complete pending work

Create Tasks

We've already created a function for creating tasks, but it is not complete just yet. That's because I want to trigger an event any time that new task is created. Solidity allows us to trigger arbitrary events which external consumers can subscribe to. It will allow us to listen for these events inside client side applications, etc...

```
// Event
event TaskCreated(
  uint id,
  string content,
  bool completed
);
```

```
function createTask(string memory _content) public {
   taskCount ++;
   tasks[taskCount] = Task(taskCount, _content, false);

   emit TaskCreated(taskCount, _content, false);
}
```

Now let's deploy a new copy of the smart contract to the blockchain since the code has changed:

```
$ truffle migrate --reset
```

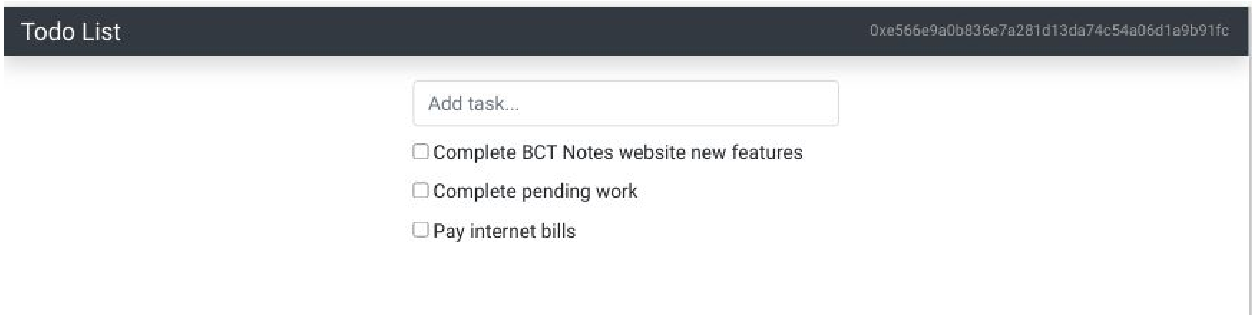
Add a form in the client side code to enter the tasks.

```
<form onSubmit="App.createTask(); return false;">
  <input id="newTask" type="text" class="form-control" placeholder="Add
task..." required>
  <input type="submit" hidden="">
</form>
```

we'll add a createTask() function in the app.js file like this:

```
createTask: async () => {
   App.setLoading(true);
   const content = $('#newTask').val();
   await App.todoList.createTask(content);
   // reload the page after adding the new task
   window.location.reload();
}
```

Now you should be able to add new tasks from the client side application! Once you do, you'll see a Metamask confirmation pop up. You must sign this transaction in order to create the task.



☐ Complete BCT Notes website new features

☐ Complete pending work

☐ Pay internet bills

Complete Tasks

"check off" the tasks in the todo list. Once we do, they will appear in the "completed" list, striked through. First, we'll update the smart contract. We'll add a TaskComplted() event, and trigger it inside a new toggleCompleted() function like this:

```
event TaskCompleted(
  uint id,
  bool completed
);
```

```
function toggleCompleted(uint _id) public {
  Task memory _task = tasks[_id];
  _task.completed = !_task.completed;
  tasks[_id] = _task;

  emit TaskCompleted(_id, _task.completed);
}
```

Now let's deploy a new copy of the smart contract to the blockchain since the code has changed:

```
$ truffle migrate --reset
```

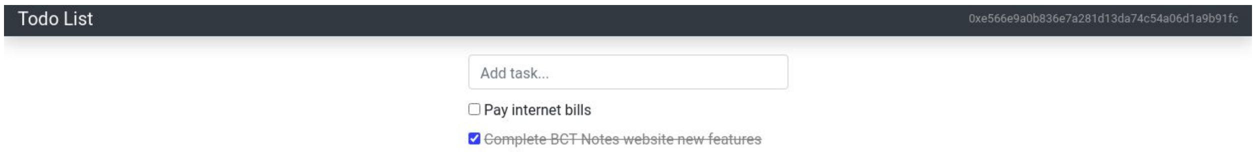
update the client side code. Add an event listener to update the task.

```
.on('click', App.toggleCompleted)
```

add a toggleCompleted() function in the app.js file like this:

```
toggleCompleted: async (e) => {
   App.setLoading(true);
   const taskId = e.target.name;
   await App.todoList.toggleCompleted(taskId);
   window.location.reload();
 },
```

Now, find a task in the client side application and click the checkbox. Once you sign this transaction, it will check off the task from the todo list!


Todo List 0xe566e9a0b836e7a281d13da74c54a06d1a9b91fc

Add task...

☐ Pay internet bills

☑ ~~Complete BCT Notes website new features~~

Testing

Now let's write a basic test to ensure that the todo list smart contract works properly. First, let me explain why testing is so important when you're developing smart contracts. We want to ensure that the contracts are bug free for a few reasons:

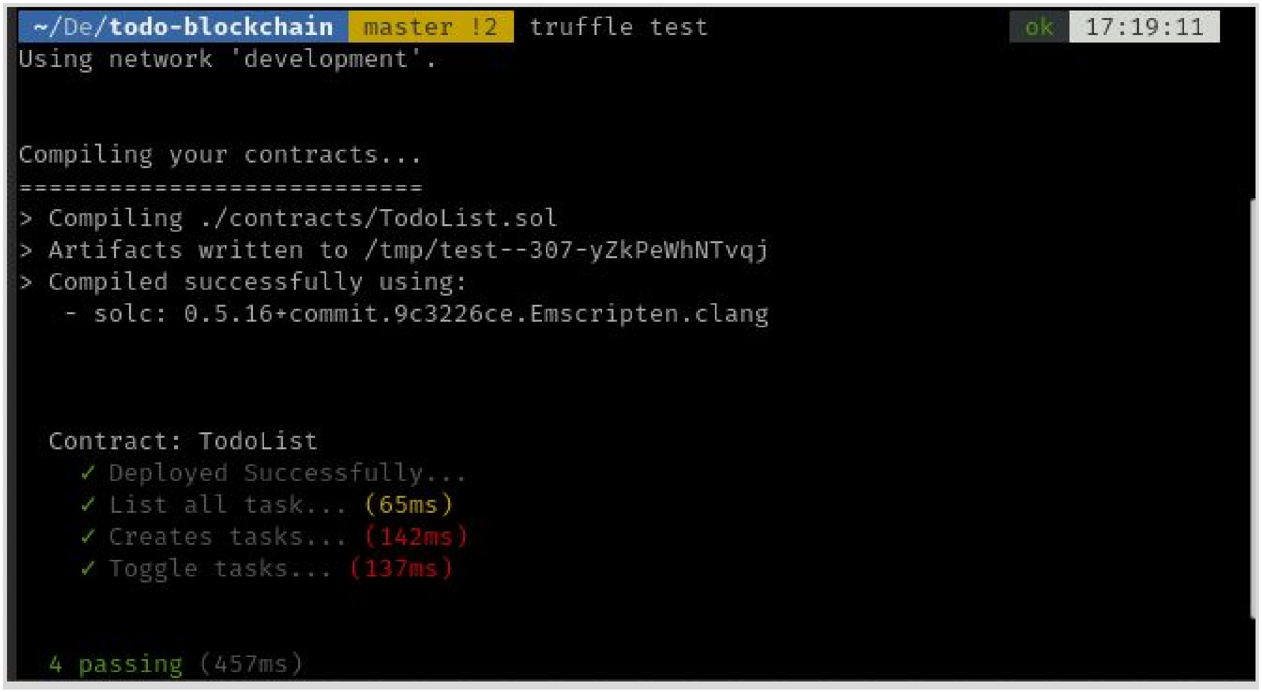
- All of the code on the Ethereum blockchain is immutable; it cannot change. If the contract contains any bugs, we must disable it and deploy a new copy. This new copy will not have the same state as the old contract, and it will have a different address.
- Deploying contracts costs gas because it creates a transaction and writes data to the blockchain. This costs Ether, and we want to minimize the amount of Ether we ever have to pay.
- If any of our contract functions that write to the blockchain contain bugs, the account who is calling this function could potentially waste Ether, and it might not behave the way they expect.

We'll write all our tests in Javascript inside this file with the Mocha testing framework and the Chai assertion library. These come bundled with the Truffle framework. We'll write all these tests in Javascript to simulate client-side interaction with our smart contract, much like we did in the console.

Here is all the code for the tests.

Run the tests from the command line like this:

```
$ truffle test
```



```
~/De/todo-blockchain  master !2  truffle test                                    ok  17:19:11
Using network 'development'.


Compiling your contracts...
===========================
> Compiling ./contracts/TodoList.sol
> Artifacts written to /tmp/test--307-yZkPeWhNTvqj
> Compiled successfully using:
   - solc: 0.5.16+commit.9c3226ce.Emscripten.clang



  Contract: TodoList
    ✓ Deployed Successfully...
    ✓ List all task... (65ms)
    ✓ Creates tasks... (142ms)
    ✓ Toggle tasks... (137ms)


  4 passing (457ms)
```

Congratulations! 🎉 🎉