Blockchain Learning Group DApp Fundamentals

May 24, 2019

Contents

An immersive, hands on bootcamp in Blockchain technology, with a focus on decentralized application (DApp) development. From the basics of Blockchain technology to cutting edge smart contract development patterns and deployment techniques. Additionally, a deep dive into utilizing the Ethereum platform to build powerful and impactful applications.

- Participants leave with a strong working knowledge foundation of full DApp development.

Note: For further information please navigate to The Blockchain Learning Group

---

CHAPTER 1

Contents

1.1 Course Prerequisites

1.1.1 1.0 Course Resources

- A quick(3 minute video) primer on how software is made

Note: Familiarity beforehand **recommended**.

---

1. JavaScript Basics
    a. Object Destructuring
    b. Arrow Functions
2. The Command Line
3. ReactJS

Note: Familiarity beforehand **nice to have**.

---

4. Truffle Framework
5. Material UI

1.1.2 2.0 Machine Specs

Attention: Participants are required to bring their own laptops.

1. 4GB of memory and some disk space(4GB+) recommended.
2. Operating System: Mac preferred, Ubuntu 16.04-17.04 and Windows 7+ OK.

1.1.3 3.0 Virtual Machine Setup

A customized virtual machine has been provided for all students that is fully configured.

Important: Virtualization must be enabled! If not by default, this will need to be updated in the computer's BIOS.

Note: View tutorial video [1-2]

---

1. Please download the virtual machine(vm) by clicking here

- The download will begin immediately or you may need to simply confirm to `Download Anyway` depending on your browser. Note the file is quite large ~2GB so it will take a few minutes to complete.
- By default the file will be named `blg-starter-vm.ova` and will be saved within your `Downloads` folder.

2. Install VirtualBox-5.2 for your respective operating system, select the correct package for Mac, Windows and Linux distributions below

- **Mac** users may download directly from here
- **Windows** users may download directly from here
- **Linux** users will need to find the correct package for their distribution here

---

Note: View tutorial video [3-5]

---

3. Once downloaded double-click on the package to open it and follow the simple steps to complete the installation

4. Once installed open VirtualBox

- For Mac users VirtualBox will be located within the Applications folder and may be opened by clicking on the icon
- Otherwise you may search for `VirtualBox` on linux or Windows machines to locate the installed application

5. Import the downloaded vm

- Within the VirtualBox application click on `File` in the top left corner
- In the dropdown menu click "Import Appliance…", this will open a dialog window

- In the dialog window, click the file icon beside the bottom text field to search for the location the vm was downloaded to, it will be named `blg-starter-vm.ova` by default and will be located in your `Downloads` folder, click on the vm to select it
- Click the `Import` button in the bottom bar, this will take a few minutes
- Once completed a new VM will be present within VirtualBox that is currently stopped with the name `blg-starter-vm`

Note: View tutorial video [6-7]

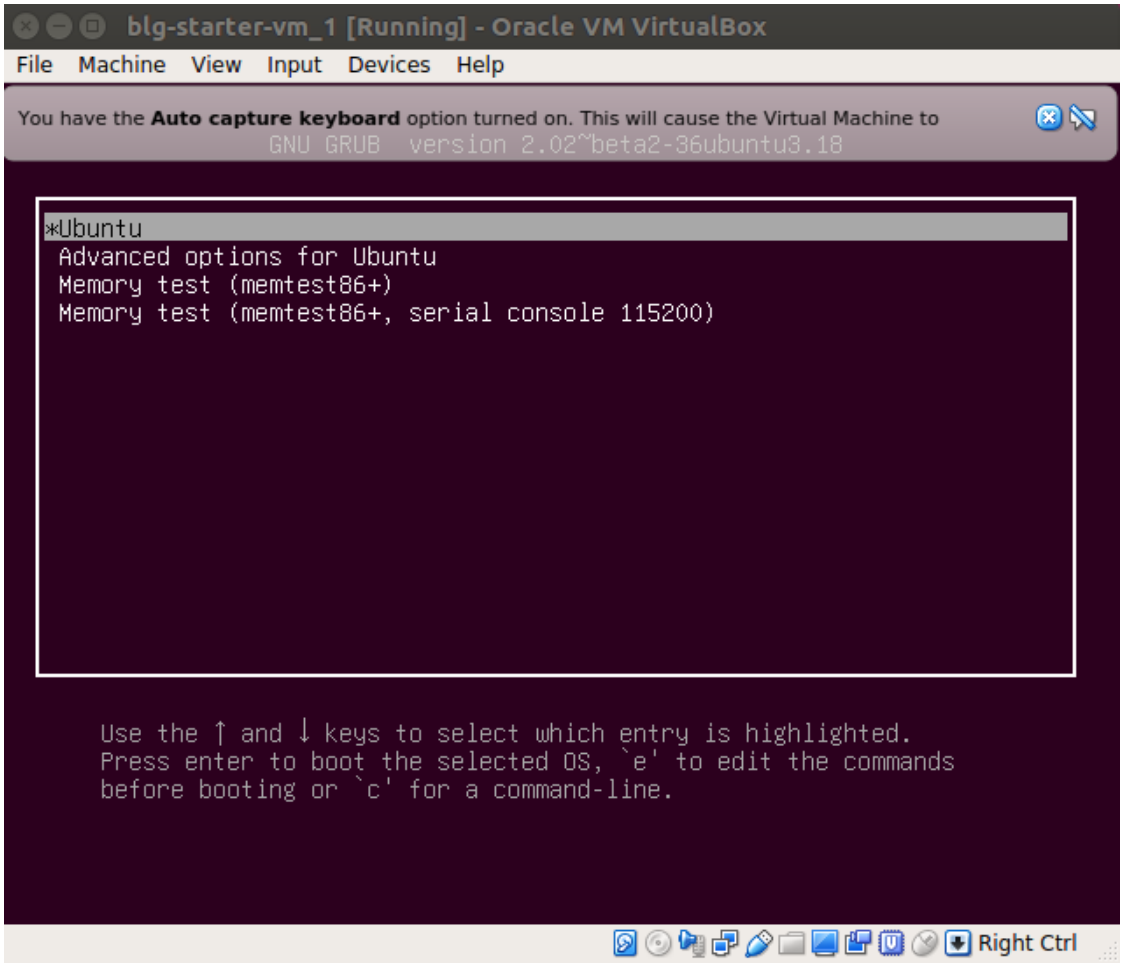
---

6. Start the vm!

- Click on the new vm and then click the `Start` button
- In the dropdown click `Normal Start`
- This will take a few minutes, your vm is starting!
- **Note ONLY if the VM fails to start**

Note: It is most likely due to virtualization not being enabled on your machine, this will need to be accomplished in the BIOS of your machine. The process to boot into BIOS is highly dependent on your specific computer and instructions will need to be sourced online. In many cases restarting the computer and holding the esc button may do so.

- You will be prompted with the dialog below, simply ensure `*Ubuntu` is highlighted and hit the `enter` key



7. Finally you will have a brand new Ubuntu 16.04 virtual machine up and running with all the required dependecies

Well done, you are all set!

8. Stop the vm

- Prior to the beginning of the course you may wish to stop the vm
- Within VirtualBox right-click on the vm that is running
- In the dropdown menu click “Stop”, this will stop your vm which you can easily start back up when the course begins by following step 6 above

1.2 Blockchain Fundamentals

1.2.1 1. ethstats.net

1.2.2 2. etherscan.io

1.2.3 3. ethernodes.org

1.2.4 4. Solidity Exercises

SimpleStorage

Important: The below exercises will be completed within REMIX. Navigate to: https://remix.ethereum.org

---

Video Tutorial

1 Define the compiler version, line 1

```
pragma solidity 0.4.24;
```

2 Create the SimpleStorage contract, line 3

- Video Tutorial

```
contract SimpleStorage {}
```

3 Compile and deploy, view the deloyed contract instance within Remix

- Video Tutorial

4 Add a first storage variable, `storedData`, line 4

- Video Tutorial

```
uint256 storedData;
```

5 Compile and deploy, view the deloyed contract instance

Note: Is the storage variable, `storedData`, available in the interface?

---

6 Update the storage variable's visibility to `public`, line 4

- Video Tutorial

```
uint256 public storedData;
```

7 Compile and deploy, view deloyed contract instance

Note: Is the storage variable, `storedData`, available in the interface now?

Important: Note the changes made between 4 and 7 and the impact of the visibility modification.

- The difference between default(internal) visibility and public visibility.

---

8 Create the SimpleStorage contract's first function to set the value of the storage variable, line 6-8

- Video Tutorial

```
function set(uint256 x) {
    storedData = x;
}
```

9 Compile and deploy the contract again, test the set function

- Video Tutorial
- Read `storedData`
- Call `set` to update the value of storedData, note default visibility
- Read `storedData`, did the value change successfully?
- Expand the transactional data within the evm console and investigate

10 Change the visibility of storedData to private, line 4

- Video Tutorial

```
uint256 private storedData;
```

Note: Storage variable is no longer accessible, let's write a function to fix that!

---

11 Create a function to get the value of `storedData`, line 10-12

- Video Tutorial

```
function get() returns (uint256) {
    return storedData;
}
```

12 Compile and deploy, test the get function

Note: Could you get the value of storedData? What did the get function return? Was gas consumed? Was a transaction sent? Or a call?

---

13 Update the get function's mutability, line 10

- Video Tutorial

```
function get() view returns (uint256) {
    return storedData;
}
```

14 Compile and deploy, test the set and get functions

- Get the initial value, what was returned this time? a transaction or a call?
- Set the value
- View it has changed
- Investigate the evm console transactional details along the way

The final solution may be found here

Important: All done? We recommend reviewing the complementary video series found here.

SimpleStorage Payable

Important: The below exercises will be completed within REMIX. Navigate to: https://remix.ethereum.org

---

- Video Lecture

1 Add an `acceptEther` function, line 6-8

- Video tutorial

```
function acceptEther() public payable {
    storedData = this.balance;
}
```

2 Compile and run, test the `acceptEther` function

- Call the function and send value
- Get the value of `storedData`, was it updated?
- Note the value has moved from the EOA to the contract

3 Add a function to withdraw the ether from this contract into the calling account, line 18-20

```
function withdraw() {
    msg.sender.transfer(this.balance);
}
```

4 Add a function to read the balance of the simple storage contract, line 22-24

```
function getMyBalance() returns(uint256) {
    return this.balance;
}
```

Important: Forgetting something? Don't forget these functions need to be marked `view` to return the value. Go ahead and modify the function with the `view` mutability modifer.

---

5. Add the `view` modifier to the `getMyBalance` function, line 22

```
function getMyBalance() view returns(uint256) {
    return this.balance;
}
```

6. Test the ability to send and withdraw Ether from the simple storage contract

- Read the contract's balance along the way, by calling `getMyBalance`

The final solution may be found here

Important: All done? We recommend reviewing the complementary video series found here.

Tic Tac Toe

Important: The below exercises will be completed within REMIX. Navigate to: https://remix.ethereum.org

---

- Solidity Types Video Tutorial

1.0 Contract Creation and Basic Functionality

1. Create the contract and initial storage variables
    - Empty Contract Video Tutorial
    - Storage Variables Video Tutorial

```
pragma solidity 0.4.24;


contract TicTacToe {

    address public player1_;
    address public player2_;
    mapping(address => uint256) public playerNumbers_;  // Map ugly address to number for simpler inspection of game board

    /** The game board itself
     * 0, 1, 2
     * 3, 4, 5
     * 6, 7, 8
     */
    uint256[9] private gameBoard_;
}
```

2. Create a function to allow a game to be started

```
function startGame(address _player1, address _player2) external {
    player1_ = _player1;
    playerNumbers_[_player1] = 1;

    player2_ = _player2;
    playerNumbers_[_player2] = 2;
}
```

3. Now players need to be able to take a turn, specifying where they want to place their x or o
    - create a function to allow this

```
/**
* @notice Take your turn, selecting a board location
* @param _boardLocation Location of the board to take
*/
function takeTurn(uint256 _boardLocation) external {}
```

4. Mark the board, within the `takeTurn` function update the `gameBoard` array

```
gameBoard_[_boardLocation] = playerNumbers_[msg.sender];
```

    - Result:

```
function takeTurn(uint256 _boardLocation) external {
    gameBoard_[_boardLocation] = playerNumbers_[msg.sender];
}
```

5. Add a function to return the contents of the game board

```
function getBoard() external view returns(uint256[9]) {
    return gameBoard_;
}
```

6. Give it a shot! Try starting a game and taking turns, watch as the game board's indexes are filled

---

- Completed code:

```
pragma solidity 0.4.24;


contract TicTacToe {

    address public player1_;
    address public player2_;
    mapping(address => uint256) public playerNumbers_;  // Map ugly address to number for simpler inspection of game board

    /** The game board itself
    * 0, 1, 2
    * 3, 4, 5
    * 6, 7, 8
    */
    uint256[9] private gameBoard_;

    function startGame(address _player1, address _player2) external {
            player1_ = _player1;
            playerNumbers_[_player1] = 1;

            player2_ = _player2;
            playerNumbers_[_player2] = 2;
    }

    /**
    * @notice Take your turn, selecting a board location
    * @param _boardLocation Location of the board to take
    */
    function takeTurn(uint256 _boardLocation) external {
        gameBoard_[_boardLocation] = playerNumbers_[msg.sender];
    }

    function getBoard() external view returns(uint256[9]) {
        return gameBoard_;
    }
}
```

- Now take a look, what problems do you notice?
- Did you have some time to play with the contract?
- Any big issues come up?

Important: What problems currently exist with this?

- Anyone can take turns!

- A player can overwrite a spot that has already been taken
- A player may take many turns in a row, alternating must be enforced

Let's tackle these problems first!

Important:

- Tic-Tac-Toe Video Tutorial

---

7. Require that only player 1 or player 2 may take turns, within the `takeTurn` function

```
require(msg.sender == player1_ || msg.sender == player2_, "Not a valid␣
↪player.");
```

8. Add a pre condition check to confirm the spot on the board is not already taken, within the `takeTurn` function

```
require(gameBoard_[_boardLocation] == 0, "Spot taken!");
```

   - Result:

```
function takeTurn(uint256 _boardLocation) external {
    require(msg.sender == player1_ || msg.sender == player2_,
↪"Not a valid player.");
    require(gameBoard_[_boardLocation] == 0, "Spot taken!");

    gameBoard_[_boardLocation] = playerNumbers_[msg.sender];
}
```

9. Add a storage variable, at the top of the contract beneath `mapping(address => uint256) public playerNumbers_;` to track who just took a turn

```
address public lastPlayed_;
```

   - Result:

```
address public player1_;
address public player2_;
mapping(address => uint256) public playerNumbers_;  // Map ugly␣
↪address to number for simpler inspection of game board
address public lastPlayed_;
```

10. Following a turn being taken update the storage variable, within the `takeTurn` function

```
lastPlayed_ = msg.sender;
```

11. Check that the same player is not trying to take another turn, within the `takeTurn` function

```
require(msg.sender != lastPlayed_, "Not your turn.");
```

   - Result:

```
function takeTurn(uint256 _boardLocation) external {
    require(msg.sender == player1_ || msg.sender == player2_, "Not a␣
↪valid player.");
    require(gameBoard_[_boardLocation] == 0, "Spot taken!");
    require(msg.sender != lastPlayed_, "Not your turn.");

    gameBoard_[_boardLocation] = playerNumbers_[msg.sender];
    lastPlayed_ = msg.sender;
}
```

Try taking turns now! More restricted / protected?

Important: Happy?

What else do we need to fix?

How about a conclusion to the game?

Let's look into how we can compute a winner

- Completed code:

```
pragma solidity 0.4.24;


contract TicTacToe {

    address public player1_;
    address public player2_;
    mapping(address => uint256) public playerNumbers_;  // Map ugly␣
↪address to number for simpler inspection of game board
    address public lastPlayed_;

    /** The game board itself
    * 0, 1, 2
    * 3, 4, 5
    * 6, 7, 8
    */
    uint256[9] private gameBoard_;

    function startGame(address _player1, address _player2) external {
            player1_ = _player1;
            playerNumbers_[_player1] = 1;

            player2_ = _player2;
            playerNumbers_[_player2] = 2;
    }

    /**
    * @notice Take your turn, selecting a board location
    * @param _boardLocation Location of the board to take
    */
    function takeTurn(uint256 _boardLocation) external {
        require(msg.sender == player1_ || msg.sender == player2_, "Not a␣
↪valid player.");
```

(continues on next page)

(continued from previous page)

```
        require(gameBoard_[_boardLocation] == 0, "Spot taken!");
        require(msg.sender != lastPlayed_, "Not your turn.");

        gameBoard_[_boardLocation] = playerNumbers_[msg.sender];
        lastPlayed_ = msg.sender;
    }

    function getBoard() external view returns(uint256[9]) {
        return gameBoard_;
    }
}
```

2.0 Finding a Winner!

12. First define which combinations within the game board, which indexes, define a "win"

```
/**
 * Winning filters:
 * 0, 1, 2
 * 3, 4, 5
 * 6, 7, 8
 *
 * 3 in a row:
 * [0,1,2] || [3,4,5] || [6,7,8]
 *
 * 3 in a column:
 * [0,3,6] || [1,4,7] || [2,5,8]
 *
 * Diagonals:
 * [0,4,8] || [6,4,2]
 */
```

13. Create a function to compute a winner and implement these combintations as filters to filter the board with

```
function isWinner(address player) private view returns(bool) {
    uint8[3][8] memory winningFilters = [
        [0,1,2],[3,4,5],[6,7,8],  // rows
        [0,3,6],[1,4,7],[2,5,8],  // columns
        [0,4,8],[6,4,2]           // diagonals
    ];
}
```

14. Create a for loop to iterate over each filter, within the `isWinner` function

```
for (uint8 i = 0; i < winningFilters.length; i++) {
    uint8[3] memory filter = winningFilters[i];
}
```

- Result:

```
function isWinner(address player) private view returns(bool) {
    uint8[3][8] memory winningFilters = [
        [0,1,2],[3,4,5],[6,7,8],  // rows
        [0,3,6],[1,4,7],[2,5,8],  // columns
        [0,4,8],[6,4,2]           // diagonals
```

(continues on next page)

(continued from previous page)

```
    ];

    for (uint8 i = 0; i < winningFilters.length; i++) {
        uint8[3] memory filter = winningFilters[i];
    }
}
```

15. Add a storage variable at the top of the contract beneath `address public lastPlayed_;` to define the winner

```
address public winner_;
```

16. Within the above `for loop` compare each filter against the game board and see if the player has won with their latest turn

```
if (
    gameBoard_[filter[0]]==playerNumbers_[player] &&
    gameBoard_[filter[1]]==playerNumbers_[player] &&
    gameBoard_[filter[2]]==playerNumbers_[player]
) {
    return true;
}
```

- Result:

```
function isWinner(address player) private view returns(bool) {
    uint8[3][8] memory winningFilters = [
        [0,1,2],[3,4,5],[6,7,8],  // rows
        [0,3,6],[1,4,7],[2,5,8],  // columns
        [0,4,8],[6,4,2]           // diagonals
    ];

    for (uint8 i = 0; i < winningFilters.length; i++) {
        uint8[3] memory filter = winningFilters[i];

        if (
            gameBoard_[filter[0]]==playerNumbers_[player] &&
            gameBoard_[filter[1]]==playerNumbers_[player] &&
            gameBoard_[filter[2]]==playerNumbers_[player]
        ) {
            return true;
        }
    }
}
```

17. At the end of the `takeTurn` function, after each turn is taken see if there is a winner, update the storage variable if there is a winner

```
if (isWinner(msg.sender)) {
    winner_ = msg.sender;
}
```

- Result:

```
function takeTurn(uint256 _boardLocation) external {
    require(msg.sender == player1_ || msg.sender == player2_, "Not a␣
↪valid player.");
```

(continues on next page)

(continued from previous page)

```
    require(gameBoard_[_boardLocation] == 0, "Spot taken!");
    require(msg.sender != lastPlayed_, "Not your turn.");

    gameBoard_[_boardLocation] = playerNumbers_[msg.sender];
    lastPlayed_ = msg.sender;

    if (isWinner(msg.sender)) {
        winner_ = msg.sender;
    }
}
```

Try it out! See if the winner is set if 3 in a row is found

Important: Are we done?

... still a few problems

- Turns can still continue to be taken, no notification that the game has ended
- What happens in the case of a draw?

- Completed code:

```
pragma solidity 0.4.24;


contract TicTacToe {

    address public player1_;
    address public player2_;
    mapping(address => uint256) public playerNumbers_;  // Map ugly␣
↪address to number for simpler inspection of game board
    address public lastPlayed_;
    address public winner_;

    /** The game board itself
    * 0, 1, 2
    * 3, 4, 5
    * 6, 7, 8
    */
    uint256[9] private gameBoard_;

    function startGame(address _player1, address _player2) external {
            player1_ = _player1;
            playerNumbers_[_player1] = 1;

            player2_ = _player2;
            playerNumbers_[_player2] = 2;
    }

    /**
    * @notice Take your turn, selecting a board location
    * @param _boardLocation Location of the board to take
```

(continues on next page)

(continued from previous page)

```
    */
    function takeTurn(uint256 _boardLocation) external {
        require(msg.sender == player1_ || msg.sender == player2_, "Not a␣
↪valid player.");
        require(gameBoard_[_boardLocation] == 0, "Spot taken!");
        require(msg.sender != lastPlayed_, "Not your turn.");

        gameBoard_[_boardLocation] = playerNumbers_[msg.sender];
        lastPlayed_ = msg.sender;

        if (isWinner(msg.sender)) {
            winner_ = msg.sender;
        }
    }

    function getBoard() external view returns(uint256[9]) {
        return gameBoard_;
    }

    function isWinner(address player) private view returns(bool) {
        uint8[3][8] memory winningFilters = [
            [0,1,2],[3,4,5],[6,7,8],  // rows
            [0,3,6],[1,4,7],[2,5,8],  // columns
            [0,4,8],[6,4,2]           // diagonals
        ];

        for (uint8 i = 0; i < winningFilters.length; i++) {
            uint8[3] memory filter = winningFilters[i];

            if (
                gameBoard_[filter[0]]==playerNumbers_[player] &&
                gameBoard_[filter[1]]==playerNumbers_[player] &&
                gameBoard_[filter[2]]==playerNumbers_[player]
            ) {
                return true;
            }
        }
    }
}
```

18. Add a storage variable to signify the game has ended, beneath `address public winner_;`

```
bool public gameOver_;
```

19. If a winner was found update that the game has ended, within the `takeTurn` function

```
gameOver_ = true;
```

- Result:

```
if (isWinner(msg.sender)) {
    winner_ = msg.sender;
    gameOver_ = true;
}
```

20. Add a storage variable to count how many turns have been taken, beneath `bool public gameOver_;`, will use this variable to define if a draw has occured

```
uint256 public turnsTaken_;
```

21. After a turn is taken update the turns taken storage variable, within the `takeTurn` function

```
turnsTaken_++;
```

22. Add a conditional that if 9 turns have been taken the game has ended with no winner, within the `takeTurn` function

```
else if (turnsTaken_ == 9) {
    gameOver_ = true;
}
```

- Result:

```
function takeTurn(uint256 _boardLocation) external {
    require(msg.sender == player1_ || msg.sender == player2_,
↪"Not a valid player.");
    require(gameBoard_[_boardLocation] == 0, "Spot taken!");
    require(msg.sender != lastPlayed_, "Not your turn.");

    gameBoard_[_boardLocation] = playerNumbers_[msg.sender];
    lastPlayed_ = msg.sender;

    if (isWinner(msg.sender)) {
        winner_ = msg.sender;
        gameOver_ = true;
    } else if (turnsTaken_ == 9) {
        gameOver_ = true;
    }

    turnsTaken_++;
}
```

23. Add a last pre condition check that the game is still active, within the `takeTurn` function

```
require(!gameOver_, "Sorry game has concluded.");
```

Try it out!!

1. Start a game with account 1 and 2
2. **Take turns back and forth**
    - view turns taken are updating the game board
    - view no winner yet
    - view game has not ended
3. View that the winner has been set
4. View that the game has ended
5. Try and take another turn and view the output in Remix's console

---

- Completed code:

```
pragma solidity 0.4.24;


contract TicTacToe {

    address public player1_;
    address public player2_;
    mapping(address => uint256) public playerNumbers_;  // Map ugly address to number for simpler inspection of game board
    address public lastPlayed_;
    address public winner_;
    bool public gameOver_;
    uint256 public turnsTaken_;

    /** The game board itself
    * 0, 1, 2
    * 3, 4, 5
    * 6, 7, 8
    */
    uint256[9] private gameBoard_;

    function startGame(address _player1, address _player2) external {
            player1_ = _player1;
            playerNumbers_[_player1] = 1;

            player2_ = _player2;
            playerNumbers_[_player2] = 2;
    }

    /**
    * @notice Take your turn, selecting a board location
    * @param _boardLocation Location of the board to take
    */
    function takeTurn(uint256 _boardLocation) external {
        require(msg.sender == player1_ || msg.sender == player2_, "Not a valid player.");
        require(gameBoard_[_boardLocation] == 0, "Spot taken!");
        require(msg.sender != lastPlayed_, "Not your turn.");
        require(!gameOver_, "Sorry game has concluded.");

        gameBoard_[_boardLocation] = playerNumbers_[msg.sender];
        lastPlayed_ = msg.sender;

        if (isWinner(msg.sender)) {
            winner_ = msg.sender;
            gameOver_ = true;
        } else if (turnsTaken_ == 9) {
            gameOver_ = true;
        }

        turnsTaken_++;
    }

    function getBoard() external view returns(uint256[9]) {
        return gameBoard_;
    }
```

(continues on next page)

(continued from previous page)

```
    function isWinner(address player) private view returns(bool) {
        uint8[3][8] memory winningFilters = [
            [0,1,2],[3,4,5],[6,7,8],  // rows
            [0,3,6],[1,4,7],[2,5,8],  // columns
            [0,4,8],[6,4,2]           // diagonals
        ];

        for (uint8 i = 0; i < winningFilters.length; i++) {
            uint8[3] memory filter = winningFilters[i];

            if (
                gameBoard_[filter[0]]==playerNumbers_[player] &&
                gameBoard_[filter[1]]==playerNumbers_[player] &&
                gameBoard_[filter[2]]==playerNumbers_[player]
            ) {
                return true;
            }
        }
    }
}
```

3.0 A High Stakes Game

OK how about a friendly wager!

24. Add a storage variable to hold the placed wagers, beneath `uint256 public turnsTaken_;`

```
mapping(address => uint256) public wagers_;
```

25. Add a function to allow the players to place a wager

```
function placeWager() external payable {
    require(msg.sender == player1_ || msg.sender == player2_, "Not a valid player.
↪");
    wagers_[msg.sender] = msg.value;
}
```

26. Update the logic if a winner is found to transfer all the value to them, within the `takeTurn` function

```
msg.sender.transfer(address(this).balance);
```

- Result:

```
if (isWinner(msg.sender)) {
    winner_ = msg.sender;
    gameOver_ = true;
    msg.sender.transfer(address(this).balance);
} else if (turnsTaken_ == 9) {
    gameOver_ = true;
}
```

27. Update the logic to refund the value if a draw has occured, within the `takeTurn` function

```
player1_.transfer(wagers_[player1_]);
player2_.transfer(wagers_[player2_]);
```

- Result:

```
if (isWinner(msg.sender)) {
    winner_ = msg.sender;
    gameOver_ = true;
    msg.sender.transfer(address(this).balance);
} else if (turnsTaken_ == 9) {
    gameOver_ = true;
    player1_.transfer(wagers_[player1_]);
    player2_.transfer(wagers_[player2_]);
}
```

Go play! Earn some ETH.

- `As above` Final solution may be found here

Homework!

- What happens when a new game wants to be started in the same contract?
- How to allow this? When to allow this? Reset storage variables?

Important: All done? We recommend reviewing the complementary video series found here.

- Completed code:

```
pragma solidity 0.4.24;


contract TicTacToe {

    address public player1_;
    address public player2_;
    mapping(address => uint256) public playerNumbers_;  // Map ugly␣
↪address to number for simpler inspection of game board
    address public lastPlayed_;
    address public winner_;
    bool public gameOver_;
    uint256 public turnsTaken_;
    mapping(address => uint256) public wagers_;

    /** The game board itself
    * 0, 1, 2
    * 3, 4, 5
    * 6, 7, 8
    */
    uint256[9] private gameBoard_;

    function startGame(address _player1, address _player2) external {
            player1_ = _player1;
            playerNumbers_[_player1] = 1;

            player2_ = _player2;
            playerNumbers_[_player2] = 2;
    }
```

(continues on next page)

(continued from previous page)

```
    /**
     * @notice Take your turn, selecting a board location
     * @param _boardLocation Location of the board to take
     */
    function takeTurn(uint256 _boardLocation) external {
        require(msg.sender == player1_ || msg.sender == player2_, "Not a␣
↪valid player.");
        require(gameBoard_[_boardLocation] == 0, "Spot taken!");
        require(msg.sender != lastPlayed_, "Not your turn.");
        require(!gameOver_, "Sorry game has concluded.");

        gameBoard_[_boardLocation] = playerNumbers_[msg.sender];
        lastPlayed_ = msg.sender;

        if (isWinner(msg.sender)) {
            winner_ = msg.sender;
            gameOver_ = true;
            msg.sender.transfer(address(this).balance);
        } else if (turnsTaken_ == 9) {
            gameOver_ = true;
            player1_.transfer(wagers_[player1_]);
            player2_.transfer(wagers_[player2_]);
        }

        turnsTaken_++;
    }

    function getBoard() external view returns(uint256[9]) {
        return gameBoard_;
    }

    function isWinner(address player) private view returns(bool) {
        uint8[3][8] memory winningFilters = [
            [0,1,2],[3,4,5],[6,7,8],  // rows
            [0,3,6],[1,4,7],[2,5,8],  // columns
            [0,4,8],[6,4,2]           // diagonals
        ];

        for (uint8 i = 0; i < winningFilters.length; i++) {
            uint8[3] memory filter = winningFilters[i];

            if (
                gameBoard_[filter[0]]==playerNumbers_[player] &&
                gameBoard_[filter[1]]==playerNumbers_[player] &&
                gameBoard_[filter[2]]==playerNumbers_[player]
            ) {
                return true;
            }
        }
    }
}
```

1.2.5 5. Bonus

Blockchain Fundamentals Video Series

a. How it works: Hashes
b. Accounts and Digital Signatures
c. Consensus Algorithms
d. Proof of Work
e. Block Target
f. Decentralized Consensus

An Introduction to ReactJS

- ReactJS tutorial

1.3 DApp Development

1.3.1 1. Introduction to ReactJS

Important: In this exercise, you're going to get a crash course on the React library by building a simple product registry and voting application.

You will become familiar with the fundamentals of React front-end development and be able to build an interactive React app from scratch!

---

1. Starting the Application

Note: Begin instructions from within the VM(via VirtualBox) that was configured and run in step 6 of the prerequisites.

Open a new terminal window

- Click on the terminal icon in the left dock

Start the app

- Change directory into the `blg/product-registry-01` folder on the Desktop

```
cd Desktop/blg/product-registry-01
```

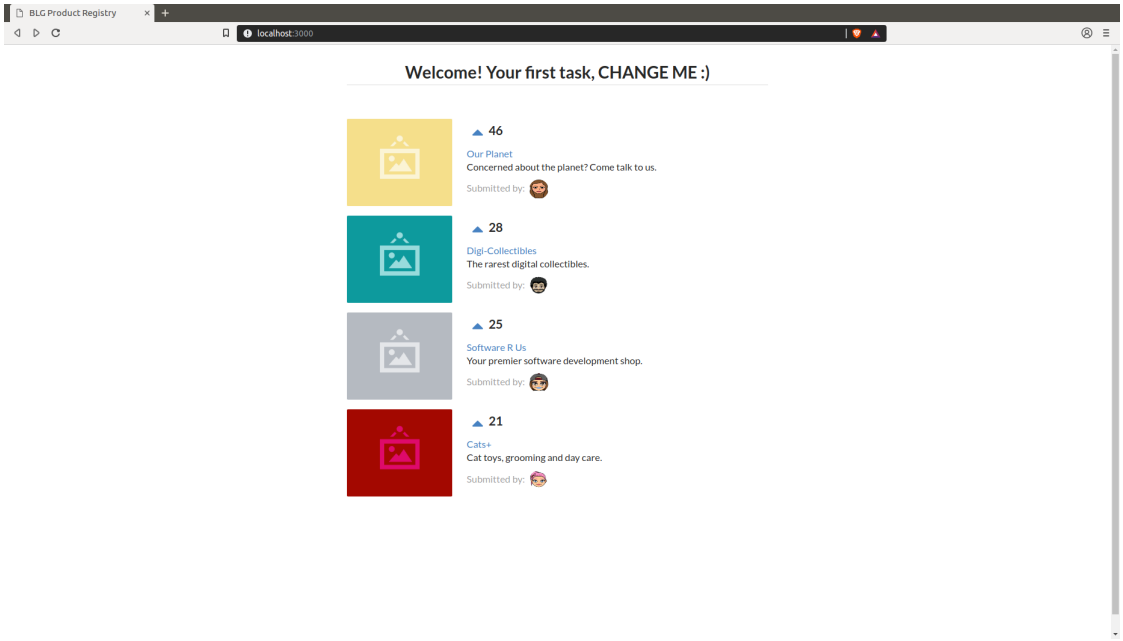
- Start the server

```
npm start
```

- *Example output:*

```
ajl@x1c:~/Desktop/blg/product-registry-01$ npm start

> product-registry-01@1.0.0 start /home/ajl/Desktop/blg/product-registry-
↪01
> live-server --host=localhost --port=3000 --middleware=./libraries/
↪disable-browser-cache.js

Serving "/home/ajl/Desktop/blg/product-registry-01" at http://
↪localhost:3000 (http://127.0.0.1:3000)
Ready for changes
```

- Chrome should automatically be opened and the application rendered! Otherwise navigate to http://localhost:3000 in your browser. You should see some basic text and your first task!



Note: The `npm start` command that you executed ran the `start` script in the included `package.json` file:

```
{
  [...],
  "scripts": {
    "start": "live-server --host=localhost --port=3000 --middleware=./
↪libraries/disable-browser-cache.js"
  },
  "devDependencies": {
    "live-server": "https://github.com/acco/live-server/tarball/master"
  }
}
```

This ran a very light-weight server that will host the code for your browser to access. The server will also detect changes in the code base and automatically re-render the browser. Therefore you will not need to

restart the server at all during development and every time that you change and save a file the browser will render the updated page for you.

---

2. Understanding and Updating the Application

Open the application's code in the Sublime text editor

- Open the Sublime text editor by clicking on the Sublime icon in the left dock.
- From within Sublime open the *product-registry-01* folder.
- Click on `File` in the top left corner and select `Open Folder...` in the menu.
- Select `Desktop/blg/product-registry-01` to open, and we can get to coding!

Open up the `index.html` file within the `product-registry-01` folder

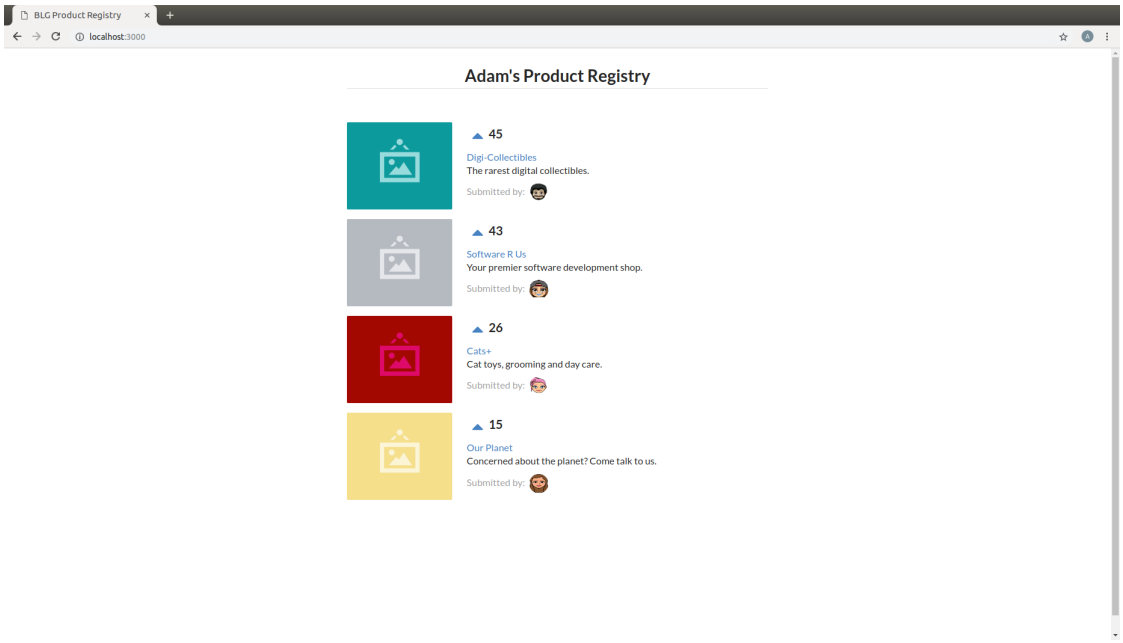
- View the contents of the file.
- Don't worry too much about what is being *linked* within the `<head>` of the file, the core to focus on is between the `<body></body>` tags beneath
- The core of the application may be simplified to the following:

```
<div>
  <h1>Welcome! Your first task, CHANGE ME :)</h1>
  <div id="content"></div>
</div>
```

  - Simply a title `<h1>` and one `<div>` that contains the *content* of the application. Remember this *content* `<div>` as we will see it again soon!
- Update the title `<h1></h1>` where your first task is noted
- Update the title to be **your** Product registry, for example `Adam's Product Registry`
- Example Code:

```
<h1 class="ui dividing centered header">Adam's Product Registry</h1>
```

- Save the file! This may be done by selecting the File menu in the top left corner and selecting save, or with the keyboard shortcut `ctrl + s`
- View the updated title in the browser!



-

Reverting to a Blank `app.js` file to get started!

- Note within the open `index.html` file that `app-complete.js` is linked in a `<script>` tag within the `<body>`
- Update this to link `app.js` instead of `app-complete.js`, which is the blank template you will begin with.
- Don't forget to save!
- Example Code:

```
<script
  type="text/babel"
  data-plugins="transform-class-properties"
  src="app.js"
></script>
```

-

3. Your First Component!

Note: Components

- React components are entirely comprised of components. A component can be thought of as a UI element within an application, generally within your browser.
- Components may be thought of as small self contained building blocks that may effectively be reused and combined within other to build up complete applications.
- The layout, logic, and specific styles are all housed within the given self-contanied component.

Taking a look into `app.js` and a first component

- The remainder of coding for this exercise will occur in the `app.js` file. Go ahead and open that one up in the Sublime text editor.
- It should contain the following *component*:

```
class ProductRegistry extends React.Component {
  render() {
    return (
      <div className='ui unstackable items'>
        Hello, I am your first React component!
      </div>
    );
  }
}
```

- A React component is simply a JavaScript class, one which extends, or inherits from, the base React Component class
- The `React` object is availble globally as the `React` library was in fact linked in the `<head>` of the `index.html` file:

```
<script src="libraries/react.js"></script>
```

- The class, which we will refer to as a component moving forward, `ProductRegistry` has only a single function, `render()`. This is a required function and is used to determine what the component will render within the browser.
- However, the return value doesn't look like traditional JavaScript, and you are right as we are actually using `JSX (JavaScript eXtension syntax)`, an extension for JavaScript. JSX allows us to write the markup for our component views in a familiar, HTML-esq syntax.
- Note the familiar looking `<div>` section within the return statement. These are the elements that will be rendered in the browser.
- Also note that although this file is now linked in your `index.html` it is not currently displayed in the browser. The text *"Hello, . . . "* is not present

Rendering your component

- You now have your first component defined and it is even linked in your `index.html` file. . . but it is not being rendered on the page. . . let's fix that.

```
<script src="app.js"></script>
```

- Remember that *content* `<div>`? Yes, we want to render our JSX component within that `<div>` on our page.
- Add the following lines at the bottom of your `app.js` file:

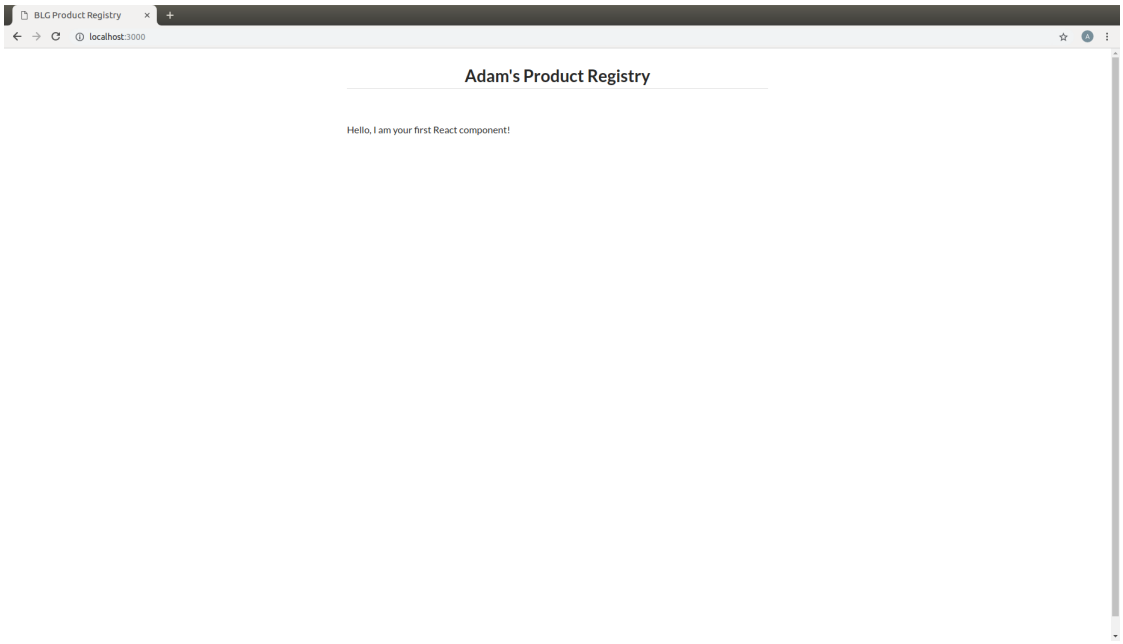
```
ReactDOM.render(
  <ProductRegistry />,
  document.getElementById('content')
);
```

- Save the file and have a look at your browser. Is a warm hello from your component present?

- Great, you have rendered your first React component!
- *ReactDOM* is a package that the React library provides to essentially allow direct interaction with the elements defined in your `index.html`.
- Above you told React to locate the element on the current page(document) with the id *content* and to render the `<ProductRegistry />` component within it. Telling React *what* you want to render and *where* you want to render it, and voila it appeared beneath your title as is defined in your `index.html`.

  Effectively resulting in the following:

```
<div>
  <h1>Product Registry</h1>
  <div id="content">
    <ProductRegistry />
  </div>
</div>
```



-

Important: Understanding and how our browser is able to understand your new JSX component.

Modern browsers' execution engines do not natively understand the JSX language. JSX is an extension to standard JavaScript, which browsers do understand. We therefore need to *translate* this JSX code to standard JavaScript so our browser can understand it. Essentially your component is speaking Espanol while our browser only understands English.

Babel is here to solve this problem for us!

Babel is a JavaScript *transpiler*, or in more familiar English language a translator. Babel understands JSX and is capable of translating your JSX into standard JavaScript. You simply need to instruct the browser to use Babel prior to attemtping to execute the JSX code.

The Babel library has been included in your `index.html`:

```
<script src="libraries/babel-standalone.js"></script>
```

Finally the browser may be instructed to use Babel directly where the `app.js` file is linked in your `index.html`:

```
<script
  type="text/babel"
  data-plugins="transform-class-properties"
  src="app.js"
></script>
```

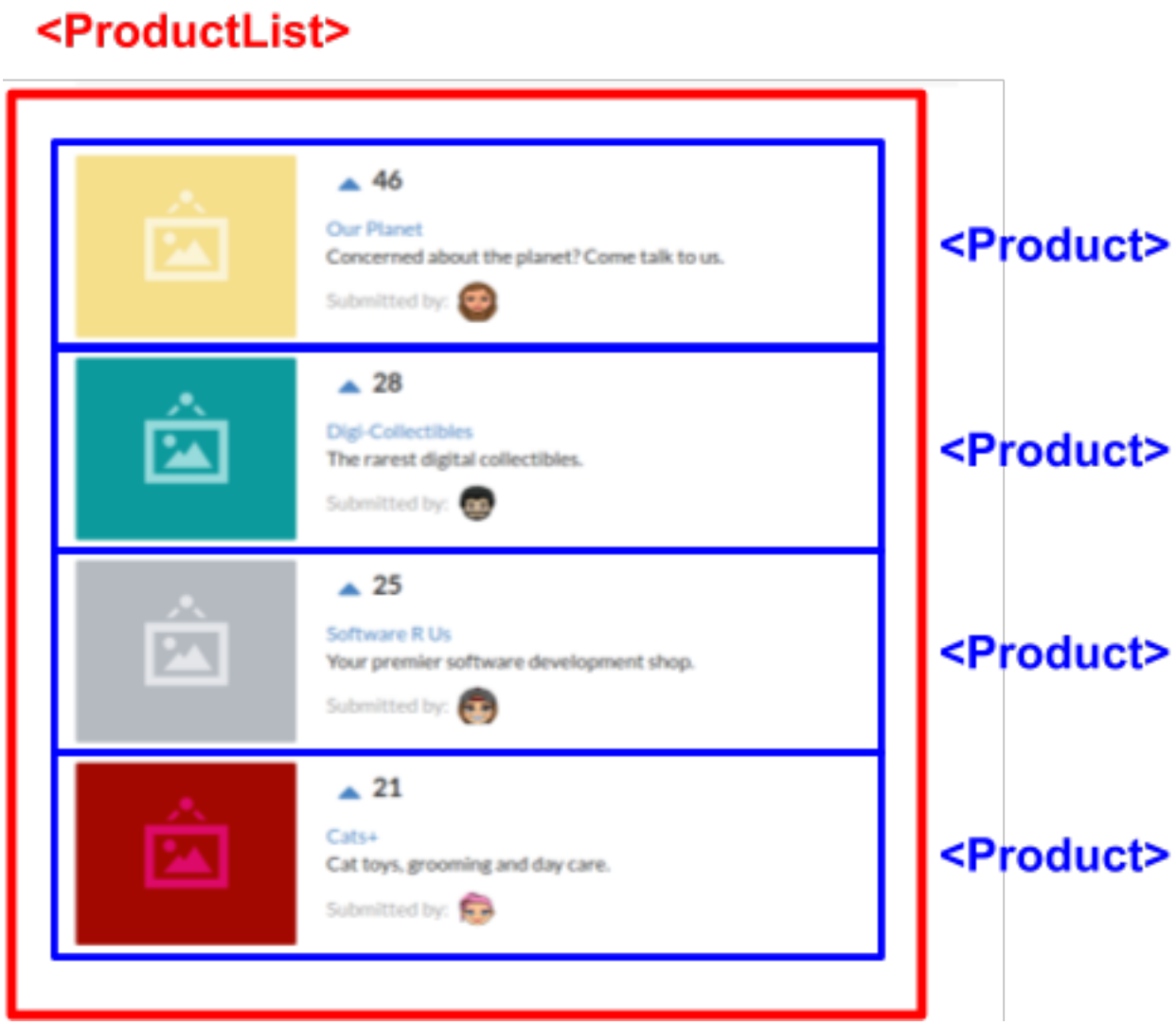
4. Creating the `<Product>` Component

Note: A best practise in designing web application front ends, and especially when utilizing the ReactJS library, is to breakdown the final design into modular, portable and reusable components.

-

Take a second and think about the components that you could break this up into. Remembering that thus far we have defined the `<ProductRegistry>`.

- Can the interface be simplified to a `<ProductRegistry>` of `<Products>`? We think so!



- Navigate back to your open `app.js` file in your text editor. It should currently look like this:

```
class ProductRegistry extends React.Component {
  render() {
    return (
      <div className='ui unstackable items'>
        Hello, I am your first React component!
      </div>
    );
  }
}

ReactDOM.render(
  <ProductRegistry />,
  document.getElementById('content')
);
```

- Begin by defining a brand new component, JavaScript class, beneath the existing `<ProductRegistry>` component

```
class Product extends React.Component {}
```

- This is a completely empty component that will not render anything and in fact will throw an error as a `render()` function is required for each component. This is the function that defines what is to be rendered by the browser and by default empty components are not allowed.
- Add a `render()` function to the `<Product>` component to return a simple `<div>` saying "hello"

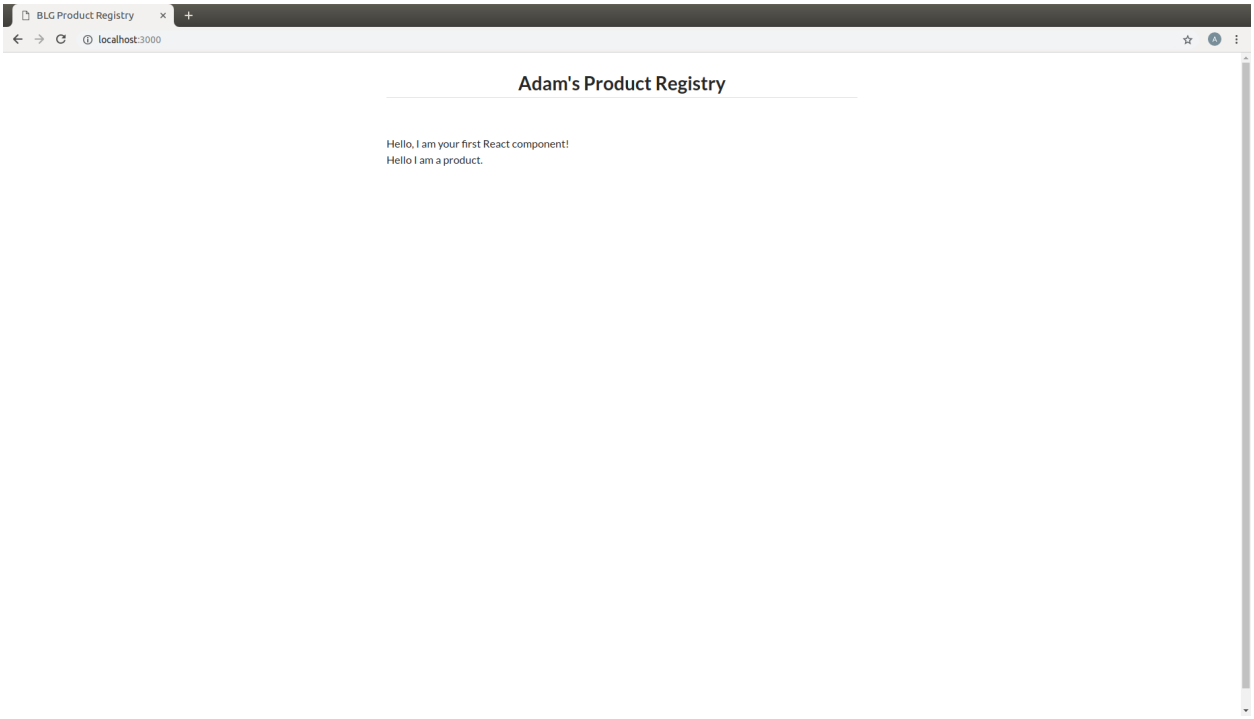
```
class Product extends React.Component {
  render() {
    return (
      <div>Hello I am a product.</div>
    );
  }
}
```

- Now remember what is currently being rendered to the page:

```
ReactDOM.render(
  <ProductRegistry />,
  document.getElementById('content')
);
```

- Therefore the `<Product>` component is not being rendered yet and will not be present in the browser
- Add the `<Product>` component to the components that are returned by your `<ProductRegistry>` within the `app.js` file

```
class ProductRegistry extends React.Component {
  render() {
    return (
      <div className='ui unstackable items'>
        Hello, I am your first React component!
        <Product />
      </div>
    );
  }
}
```



-

5. Building out Your Products

- Great work so far! However, the application is not looking overly interesting just yet. In this section you will build out the `<Product>` component.

Note:

Semantic is being used as a styling library which has been linked in your `index.html` file for you. Therefore, the `className` reference in several of the elements below are in fact utilizing classes and styling that is provided by Semantic.

- Begin by extending the content that is returned by the component in it's `render()` function. You will add an image, a title, a description, and an image of who submitted the product.

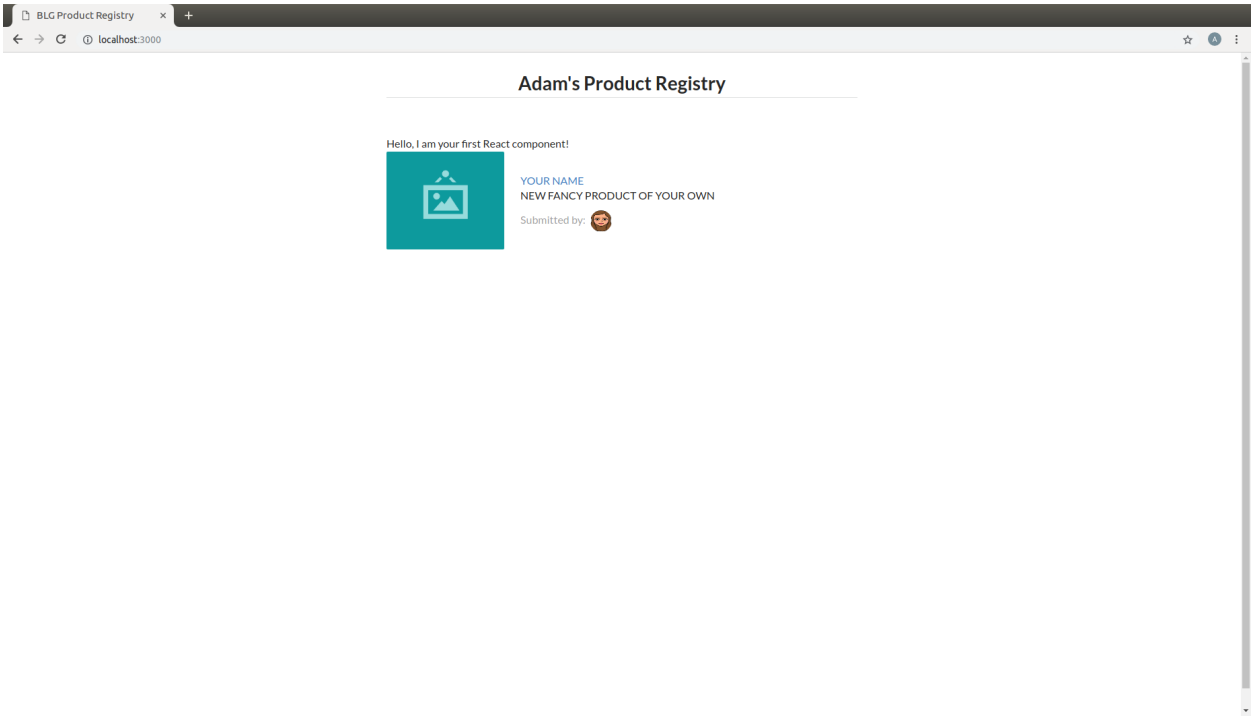
```
class Product extends React.Component {
  render() {
    return (
      <div className='item'>
        <div className='image'>
          <img src='images/products/image-aqua.png' />
        </div>
        <div className='middle aligned content'>
          <div className='description'>
            <a>YOUR PRODUCT NAME</a>
            <p>NEW FANCY PRODUCT OF YOUR OWN</p>
          </div>
          <div className='extra'>
```

(continues on next page)

(continued from previous page)

```
            <span>Submitted by:</span>
            <img className='ui avatar image' src='images/avatars/liz.png' />
          </div>
        </div>
      </div>
    );
  }
}
```

- The image links in the component above map to the images that have been provided for you within the `/images` folder.
- The product image: `<img src='images/products/image-aqua.png' />` and the submitted image: `<img className='ui avatar image' src='images/avatars/liz.png' />` both exist within their respective folders `images/products/` and `images/avatars/`, feel free to add new images of your own and update these paths accordingly!



-

6. Loading Your Product List

- Currently your `<Product>` is completely hardcoded and although it may seem quite portable where you can place it just about anywhere, and may create many instances of it, that is where the functionality ends. That `<Product>` will currently always have the exact same images and text. This section will take you through the process of making the content of your components dynamic and allow them to be passed in as variables!
- Review your existing `<Product>` and have a look at the data fields that are present:
  1. Product Image URL:

```
<div className='image'>
  <img src='images/products/image-aqua.png' />
</div>
```

2. Product Title:

```
<div className='description'>
  <a>YOUR PRODUCT NAME</a>
  ...
</div>
```

3. Product Description:

```
<div className='description'>
  ...
  <p>NEW FANCY PRODUCT OF YOUR OWN</p>
</div>
```

4. Submitted Image URL:

```
<div className='extra'>
  <span>Submitted by:</span>
  <img className='ui avatar image' src='images/avatars/liz.png' />
</div>
```

- Therefore a minimal representation of the data fields that are required for a `<Product>` at this time are:

```
product = {
  title:              'YOUR PRODUCT NAME',
  description:        'YOUR PRODUCT DESCRIPTINON.',
  submitterAvatarUrl: 'images/avatars/adam.jpg',
  productImageUrl:    'images/products/image-aqua.png',
}
```

- Open up the file `seed.js` and have a look around. Observe the definition of the array, or list, of products.
- Note there are 2 additional fields we did not previously define, `id` and `votes`, which we will see in action shortly.

```
const products = [
  {
    id: 1,
    title: 'Digi-Collectibles',
    description: 'The rarest digital collectibles.',
    votes: generateVoteCount(),
    submitterAvatarUrl: 'images/avatars/adam.jpg',
    productImageUrl: 'images/products/image-aqua.png',
  },
  ...
]
```

- This file defines a list of components that are to populate, or seed, the application when it initially renders. This will take a few steps, first you will need to update your `<Product>` component to allow data for it's fields to be passed in.

Making your `<Product>` dynamic and data-driven

Note: In order to do this we must introduce another ReactJS concept, that is **Props**.

Components are allowed to accpet data passed to them from their *parents* meaning the components that contain them. In your case the parent is the `<ProductRegistry>` and it may have many child `<Product>` s. Therefore the `<Product>` components may accept data passed to them by the `<ProductRegistry>`, and it is this data, passed from parent to child, that is referred to as *props*. Essentially the input parameters that a component may accept are referred to as *props*.

Also `this` is a special keyword in JavaScript. For the time being we can assume his will be bound to the React component class. Therefore `this.props` inside the component is accessing the `props` attribute on the component class.

- Time to update your `<Product>` component to accept some props!

  1. Update Product Image URL:

```
<div className='image'>
  <img src='images/products/image-aqua.png' />
</div>
```

     to >>

```
<div className='image'>
  <img src={this.props.productImageUrl} />
</div>
```

  2. Product Title:

```
<div className='description'>
  <a>YOUR PRODUCT NAME</a>
  ...
</div>
```

     to >>

```
<div className='description'>
  <a>{this.props.title}</a>
  ...
</div>
```

  3. Product Description:

```
<div className='description'>
  ...
  <p>NEW FANCY PRODUCT OF YOUR OWN</p>
</div>
```

     to >>

```
<div className='description'>
  ...
  <p>{this.props.description}</p>
</div>
```

  4. Submitted Image URL:

```
<div className='extra'>
  <span>Submitted by:</span>
  <img className='ui avatar image' src='images/avatars/liz.png' />
</div>
```

to >>

```
<div className='extra'>
  <span>Submitted by:</span>
  <img className='ui avatar image' src={this.props.submitterAvatarUrl} />
</div>
```

- The resulting component should look like the following:

```
class Product extends React.Component {
  render() {
    return (
      <div className='item'>
        <div className='image'>
          <img src={this.props.productImageUrl} />
        </div>
        <div className='middle aligned content'>
          <div className='description'>
            <a>{this.props.title}</a>
            <p>{this.props.description}</p>
          </div>
          <div className='extra'>
            <span>Submitted by:</span>
            <img className='ui avatar image' src={this.props.submitterAvatarUrl} /
↪>
          </div>
        </div>
      </div>
    );
  }
}
```

- This may look odd at first, seeing JavaScript directly inline with html, and that is in fact the beauty of JSX! The `{}` braces identify that what is within them is a JavaScript expression. Therefore the props object of the `<Product>` component(remember just a JavaScript class).
- You will notice immediately that the rendered `<Product>` is currently empty as there are no props being passed in to the component now to every data field is in fact empty. Let's fix that and get your hands on some data.
-

Understanding `seed.js` and the `window`

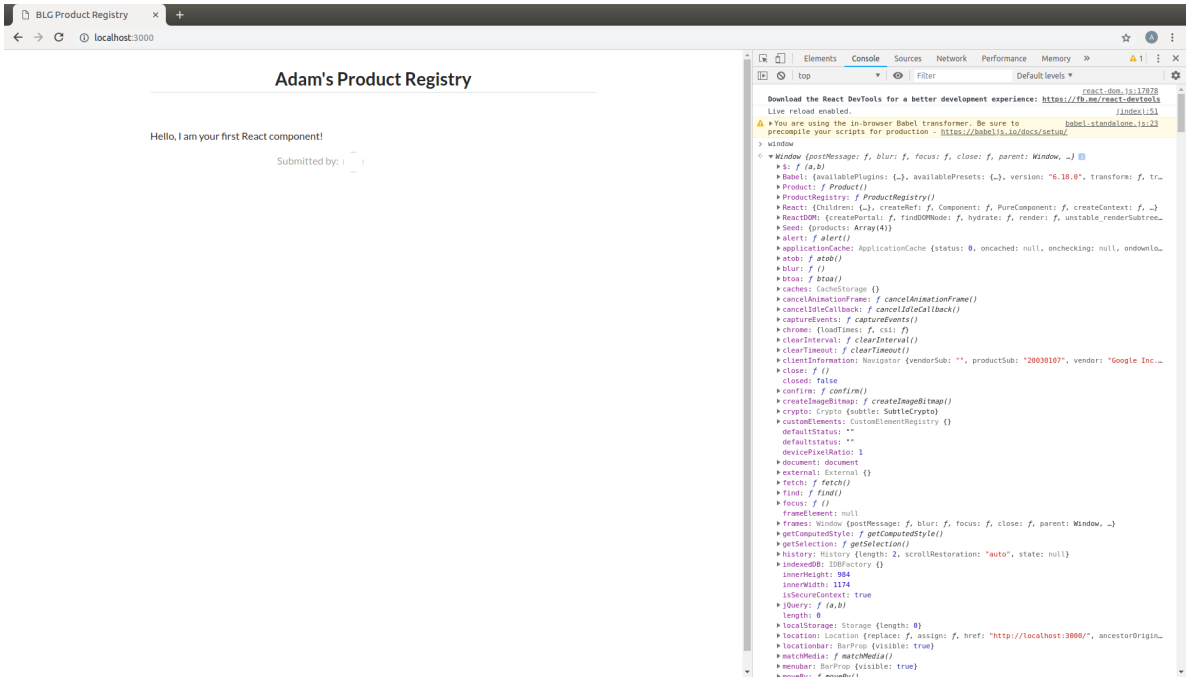
- Take a look back at the `seed.js` file that should be open at this time, if it is not do open it now. This is where our data is going to come from!
- Let's walk through this file one step at a time . . .

Note: JavaScript exposes a top level object accessible with the `window` keyword which represents the browser window. This object contains many functions and properties about the given browser session and also allows

you as a developer to bind data to the window that may be globally accessible.

Let's have a look at this `window` object directly in the browser to see what it is all about.

- Right-click in your browser and in the dropdown menu select *inspect*.
- Within the console type `window` and hit enter.
- You will see a reference to the global `window` object, expand this and have a look around. This is the top level object that JavaScript natively exposes.



- The first line of `seed.js` is in fact accessing this `window` object and adding an attribute to it: `Seed`.

```
window.Seed = (function () {...});
```

- This attribute is set to a `function()` and this function returns an object: `{ products: products }`. Resulting in: `window.Seed = { products: products }`
- Where `products` is your array of product data fields:

```
const products = [
  {
    id: 1,
    title: 'Digi-Collectibles',
    description: 'The rarest digital collectibles.',
    votes: generateVoteCount(),
    submitterAvatarUrl: 'images/avatars/adam.jpg',
    productImageUrl: 'images/products/image-aqua.png',
  },
  ...
]
```

- Have a look at this attribute back in the browser.
- In the browser console type `window.Seed` and you will see the result!

- Yes, all of your data has been loaded and is availble globally attached to the `window` object.
- In fact every `window` attribute is accessible without the `window` keyword as they are added to the global scope of the application. Therefore this object may simply be accessed globally via just `Seed`.

```
> window.Seed
  > {products: Array(4)}

> Seed
  > {products: Array(4)}
```

Loading a product from the Seed

- Now it is time to utilize this seed data to populate your `<Product>`
- Within the `<ProductRegistry>` component load the first product of the Seed data into a local variable. Remember it is the parent that must pass the data to the child, `<Product>`, as props.

```
class ProductRegistry extends React.Component {
  render() {
    const product = Seed.products[0];

    return (
      <div className='ui unstackable items'>
        Hello, I am your first React component!
        <Product />
      </div>
    );
  }
}
```

- Pass the seed data to the `<Product>` component as props:

```
<Product
  title={product.title}
  description={product.description}
  submitterAvatarUrl={product.submitterAvatarUrl}
  productImageUrl={product.productImageUrl}
/>
```

- You can also get rid of the "Hello…" line that is currently in the `<ProductRegistry>` resulting with the following component:

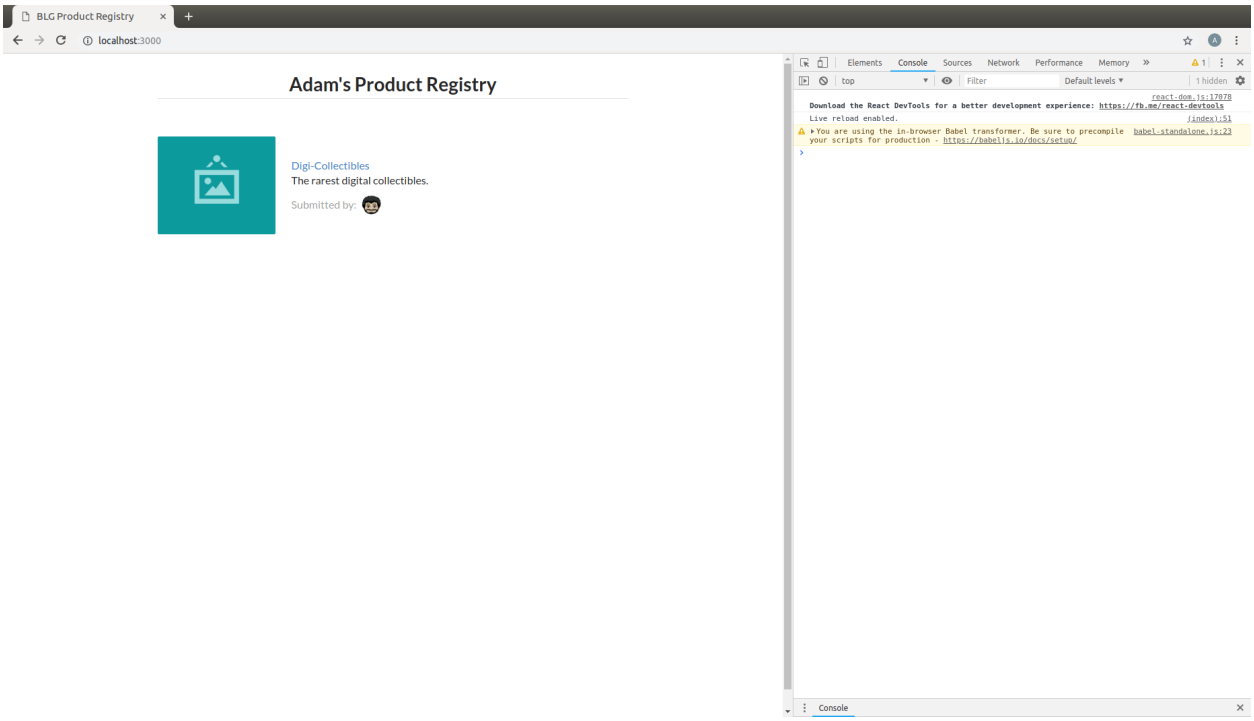
```
class ProductRegistry extends React.Component {
  render() {
    const product = Seed.products[0];

    return (
      <div className='ui unstackable items'>
        <Product
          title={product.title}
          description={product.description}
          submitterAvatarUrl={product.submitterAvatarUrl}
          productImageUrl={product.productImageUrl}
        />
      </div>
    );
```

(continues on next page)

(continued from previous page)

```
    }
}
```

- You should see the data from the first product in the `seed.js` file rendered!



-

7. Loading the Entire Product Registry

- This section will aid to highlight the beauty and power of the portable and modular nature of React components!
- You will now load all of the seeded products into the registry.
- Within the `<ProductRegistry>` component now instead of just loading the first product in the array iterate over the entire list. In order to do this you will leverage the internal `map` function of the JavaScript language.

Note: JavaScript's `map` function

`array.map(function(currentValue, index, arr), func())`

`map` is a function that is accessible on every array object. This function essentially allows efficient iteration over all of the array components.

For example:

```
> const myArray = [1,2,3,4]
> myArray.map(arrayItem => console.log(arrayItem))
1
2
3
4
```

- Instead of loading just the first product from the seed now iterate over all the products and define a `<Product>` component to be rendered for each:

  remember anything between `{}` *allows you to use native JavaScript*

```
return (
  <div className='ui unstackable items'>
    {
      Seed.products.map(product =>
        <Product
          title={product.title}
          description={product.description}
          submitterAvatarUrl={product.submitterAvatarUrl}
          productImageUrl={product.productImageUrl}
        />
      )
    }
  </div>
);
```

- Now you will notice an error in the browser console stating: `Warning: Each child in an array or iterator should have a unique "key" prop.`
- The use of the `key` prop is something that the React framework uses to identify each instance of the Product component.
- For the time being just remember that this attribute needs to be unique for each React component.
- Add a `key` and `id` prop to the `<Product>` component:

```
<Product
  key={'product-'+product.id}
  id={product.id}
  title={product.title}
  description={product.description}
  submitterAvatarUrl={product.submitterAvatarUrl}
  productImageUrl={product.productImageUrl}
/>
```

- Resulting with the final `<ProductRegistry>` component:

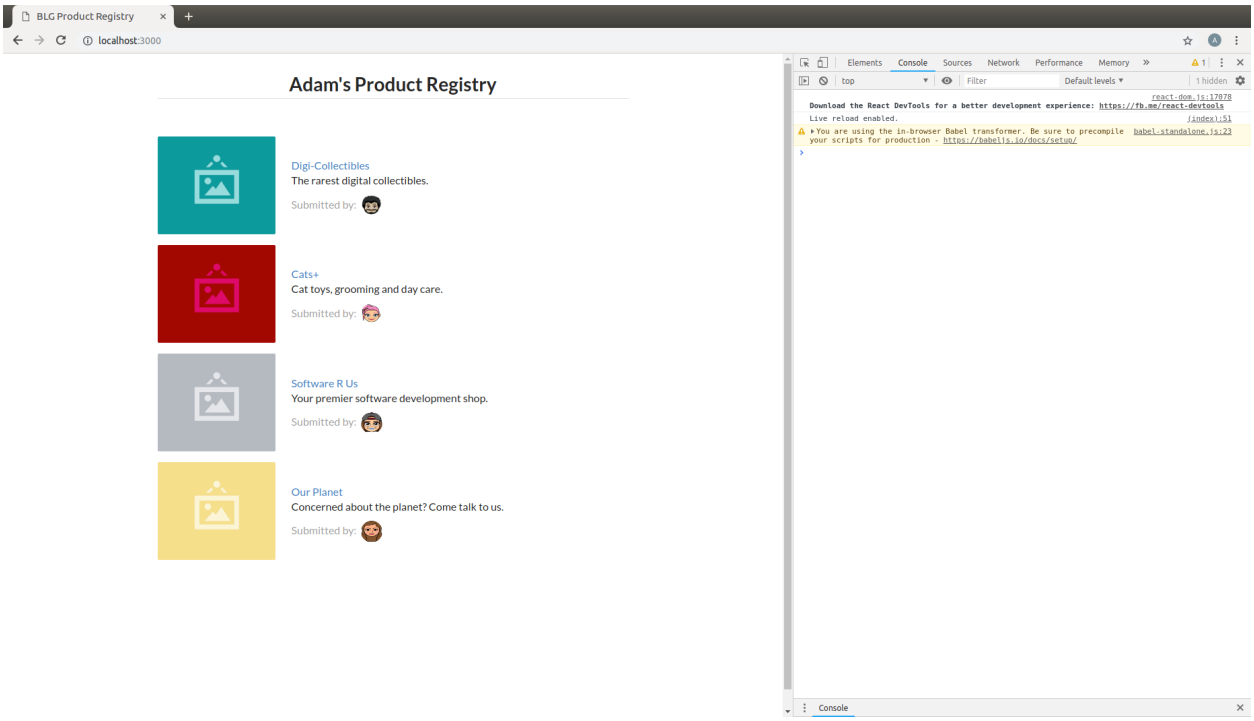
```
class ProductRegistry extends React.Component {
  render() {
    return (
      <div className='ui unstackable items'>
        {
          Seed.products.map(product =>
            <Product
              key={'product-'+product.id}
              id={product.id}
              title={product.title}
              description={product.description}
              submitterAvatarUrl={product.submitterAvatarUrl}
              productImageUrl={product.productImageUrl}
            />
          )
        }
      </div>
    );
```

(continues on next page)

(continued from previous page)

```
    }
}
```

- And all seeded products should be rendered!



-

8. Voting for a Product - Dynamically Updating the UI

- This section will allow you to vote on your favourite products. Interacting with the application and dynamically up it!
- You will learn how to manage interaction with your components and how to dynamically update data that is stored in a component's *state*.
- Begin by updating the product component to show its current number of votes as well as a button to click on to vote for that product.

```
<div className='header'>
  <a>
    <i className='large caret up icon' />
  </a>
  {this.props.votes}
</div>
```

- Resulting in the following `<Product>` component:

```
class Product extends React.Component {
  render() {
    return (
      <div className='item'>
```

(continues on next page)

(continued from previous page)

```
        <div className='image'>
          <img src={this.props.productImageUrl} />
        </div>
        <div className='middle aligned content'>
          <div className='header'>
            <a>
              <i className='large caret up icon' />
            </a>
            {this.props.votes}
          </div>
          <div className='description'>
            <a>{this.props.title}</a>
            <p>{this.props.description}</p>
          </div>
          <div className='extra'>
            <span>Submitted by:</span>
            <img className='ui avatar image' src={this.props.submitterAvatarUrl} /
↪>
          </div>
        </div>
      </div>
    );
  }
}
```

- Notice that `this.props.votes` is being accessed but is not currently being passed in by the parent `<ProductRegistry>`.
- Update the `<ProductRegistry>` to also pass in votes as a prop:

```
votes={product.votes}
```

- Resulting in the complete `<Product>` definition:

```
<Product
  key={'product-'+product.id}
  id={product.id}
  title={product.title}
  description={product.description}
  submitterAvatarUrl={product.submitterAvatarUrl}
  productImageUrl={product.productImageUrl}
  votes={product.votes}
/>
```

Time for some interaction!

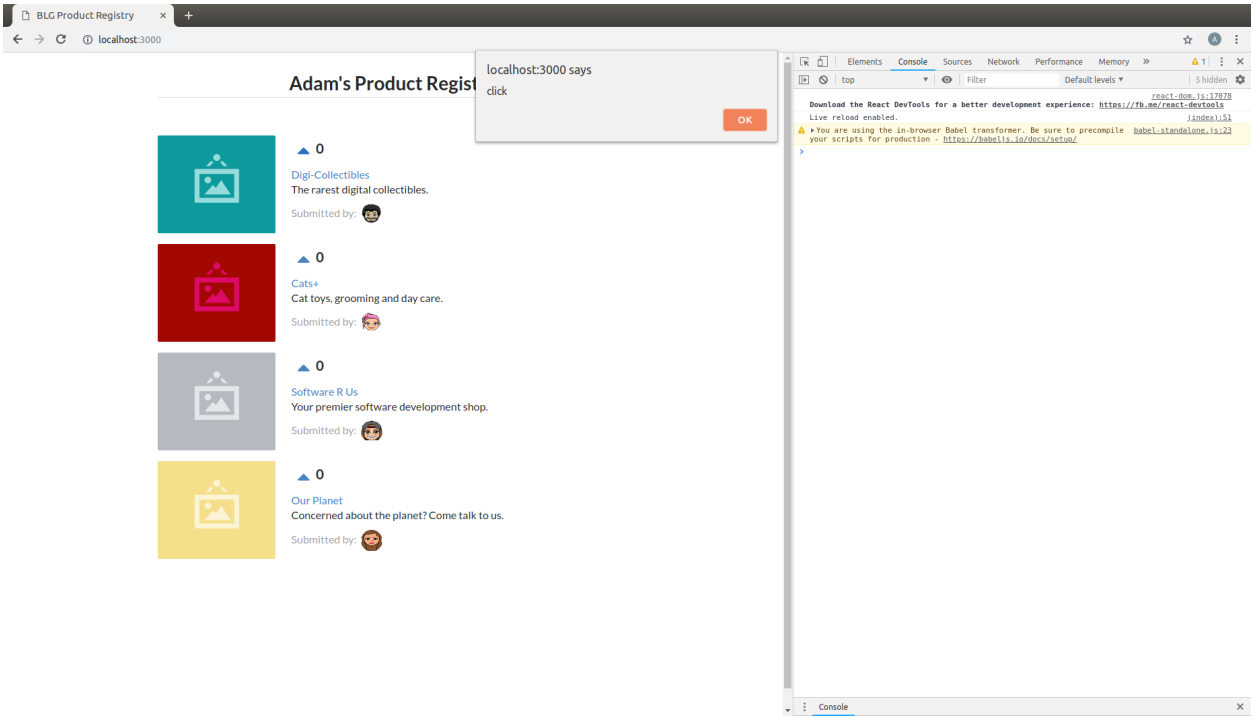
- When the voting caret is clicked we want to increment the product's total vote count.
- In order to do this we need to register the event when the given product is clicked.
- React features many built-in listeners for such events. In fact an `onClick` prop exists that we can access directly.
- Within the definition of the caret in the `<Product>` component add the `onClick` prop and create an alert whenever a click occurs.

```
<div className='header'>
  <a onClick={() => alert('click')}>
    <i className='large caret up icon' />
  </a>
  {this.props.votes}
</div>
```

- Resulting in the following `<Product>` component:

```
class Product extends React.Component {
  render() {
    return (
      <div className='item'>
        <div className='image'>
          <img src={this.props.productImageUrl} />
        </div>
        <div className='middle aligned content'>
          <div className='header'>
            <a onClick={() => alert('click')}>
              <i className='large caret up icon' />
            </a>
            {this.props.votes}
          </div>
          <div className='description'>
            <a>{this.props.title}</a>
            <p>{this.props.description}</p>
          </div>
          <div className='extra'>
            <span>Submitted by:</span>
            <img className='ui avatar image' src={this.props.submitterAvatarUrl} /
↪>
          </div>
        </div>
      </div>
    );
  }
}
```

- Try it out!



- Now we need to update the number of votes that the clicked on product currently has every time that caret is clicked.

Note: The props of a given component are not *owned* by the child component itself but instead are treated as immutable, or permanent, at the child component level and owned by the parent.

So the way you currently have your components setup, parent `<ProductRegistry>` passing in the `votes` prop to child `<Product>` means that the `<ProductRegistry>` must be the one to update the given value.

Therefore, the first order of business is to have this click event on the `<Product>` propagated upwards to the `<ProductRegistry>`. React allows you to not only pass data values as props but functions as well to solve this problem!

- Add a function within your `<ProductRegistry>` component to handle the event when a vote is cast:

```
handleProductUpVote = (productId) => {
  console.log(productId);
}
```

- Pass this function to each `<Product>` as a new prop called `onVote`

```
onVote={this.handleProductUpVote}
```

- Resulting in the complete `<ProductRegistry>`:

```
class ProductRegistry extends React.Component {
  handleProductUpVote = (productId) => {
    console.log(productId);
  }

  render() {
    return (
```

(continues on next page)

(continued from previous page)

```
      <div className='ui unstackable items'>
        {
          Seed.products.map(product =>
            <Product
              key={'product-'+product.id}
              id={product.id}
              title={product.title}
              description={product.description}
              submitterAvatarUrl={product.submitterAvatarUrl}
              productImageUrl={product.productImageUrl}
              votes={product.votes}
              onVote={this.handleProductUpVote}
            />
          )
        }
      </div>
    );
  }
}
```

- Update the `<Product>` to no longer raise the alert but instead call its `onVote` prop, pass the id of the clicked component in order to determine where the event occured to cast the vote correctly:

```
<a onClick={() => this.props.onVote(this.props.id)}>
```

- Resulting in the complete `<Product>`:

```
class Product extends React.Component {
  render() {
    return (
      <div className='item'>
        <div className='image'>
          <img src={this.props.productImageUrl} />
        </div>
        <div className='middle aligned content'>
          <div className='header'>
            <a onClick={() => this.props.onVote(this.props.id)}>
              <i className='large caret up icon' />
            </a>
            {this.props.votes}
          </div>
          <div className='description'>
            <a>{this.props.title}</a>
            <p>{this.props.description}</p>
          </div>
          <div className='extra'>
            <span>Submitted by:</span>
            <img className='ui avatar image' src={this.props.
↪submitterAvatarUrl} />
          </div>
        </div>
      </div>
    );
  }
}
```

- Try it out! Noting the id of the product logged to the browser developer console, 1,2,3 or 4, and successfully the

event has been propagated upward to the parent component!

-

Introducing: The State!

Note: Props as we defined earlier are seen as immutable by a component and owned by a it's parent. State is instead owned by the component itself private to that component. The state of a component is in fact mutable and accessible via a function provided by the `React.Component` base class called `this.setState()`. And it is with the call of `this.setState()` that the component will also no to re-render itself with the new data!

- Begin by defining the initial state of the `<ProductRegistry>`:

```
state = {
  products: Seed.products
};
```

- Update the `render` function to now read from the component's state instead of the seed file directly:
- Resulting in the complete `<ProductRegistry>`:

```
class ProductRegistry extends React.Component {
  state = {
    products: Seed.products
  };

  handleProductUpVote = (productId) => {
    console.log(productId);
  }

  render() {
    return (
      <div className='ui unstackable items'>
        {
          this.state.products.map(product =>
            <Product
              key={'product-'+product.id}
              id={product.id}
              title={product.title}
              description={product.description}
              submitterAvatarUrl={product.submitterAvatarUrl}
              productImageUrl={product.productImageUrl}
              votes={product.votes}
              onVote={this.handleProductUpVote}
            />
          )
        }
      </div>
    );
  }
}
```

Important: Never modify state outside of `this.setState()`!

State should NEVER be accessed directly, i.e. this.state = {}, outside of its initial definition.

`this.setState()` has very important functionality built around it that can cause odd and unexpected behaviour if avoided. Always use `this.setState()` when updating the state of a component.

- Now although we noted earlier that props are seen as immutable from the given component and state mutable a slight variation to that definition must be explained
- Yes, the state may be updated, but the current state object is said to be immutable, meaning that the state object should not be updated directly but instead replaced with a new state object
- For example directly updating, mutating, the current state is bad practise!

```
// INCORRECT!
this.state = { products: [] };
this.state.products.push("hello");
```

- Instead a new state object is to be created and the state update to the new object.

```
// CORRECT!
this.state = { products: [] };
const newProducts = this.state.products.concat("hello");
this.setState({ products: products });
```

- Therefore when we want to update the state when a vote has been cast we need to:

1. Create a copy of the state
   - Map will return a copy of each item in the array it will not reference the existing.

```
const nextProducts = this.state.products.map((product) => {
  return product;
});
```

2. Determine which product was voted for

```
if (product.id === productId) {}
```

3. Mutate the copy of the state incrementing the product's vote count
   - Create a new product Object via `Object.assign` and update the `votes` attribute of that object to +1 of the existing product

```
return Object.assign({}, product, {
  votes: product.votes + 1,
});
```

4. Set the state to the new object

```
this.setState({ products: nextProducts });
```

- Resulting in the following segment added within the `handleProductUpVote` function of the `<ProductRegistry>` to update the vote count of a selected product identified by its `id`:

```
const nextProducts = this.state.products.map((product) => {
  if (product.id === productId) {
    return Object.assign({}, product, {
      votes: product.votes + 1,
    });
  } else {
```

(continues on next page)

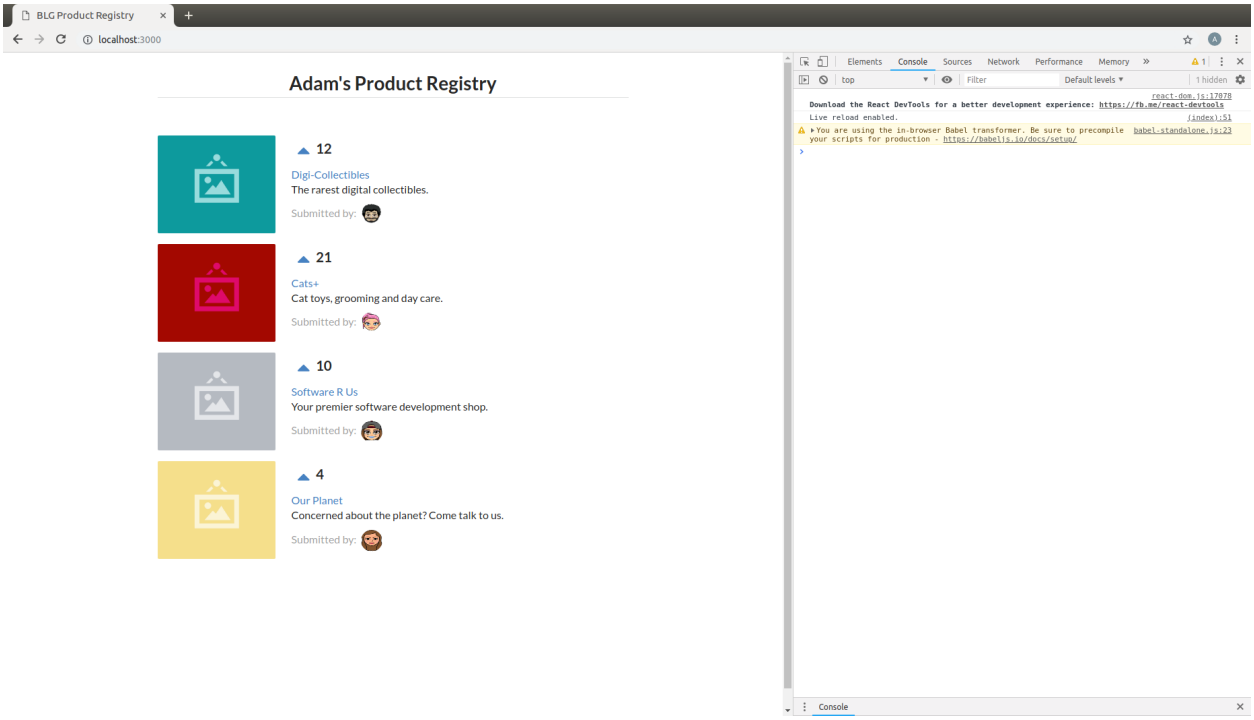
(continued from previous page)

```
    return product;
  }
});
```

- Resulting in the following complete `<ProductRegistry>`:

```
class ProductRegistry extends React.Component {
  state = {
    products: Seed.products
  };

  handleProductUpVote = (productId) => {
    const nextProducts = this.state.products.map((product) => {
      if (product.id === productId) {
        return Object.assign({}, product, {
          votes: product.votes + 1,
        });
      } else {
        return product;
      }
    });

    this.setState({ products: nextProducts });
  }

  render() {
    return (
      <div className='ui unstackable items'>
        {
          this.state.products.map(product =>
            <Product
              key={'product-'+product.id}
              id={product.id}
              title={product.title}
              description={product.description}
              submitterAvatarUrl={product.submitterAvatarUrl}
              productImageUrl={product.productImageUrl}
              votes={product.votes}
              onVote={this.handleProductUpVote}
            />
          )
        }
      </div>
    );
  }
}
```

- Give it a shot!



-

Important: All done? We recommend reviewing the complementary video series found here.

1.3.2 2. Introduction to DApps

View Completed Wallet Demo

Important:

Stage 1: Starting the Application

Note: Begin instructions from within the VM(via VirtualBox) that was configured and run in step 6 of the prerequisites.

Video Tutorial [1-2]

1. Open a new terminal window
   - Click on the terminal icon in the left dock
2. Start the app

2.1 Change directory into the `blg/wallet-template` folder

```
cd Desktop/blg/wallet-template
```

2.2 Start the server

Note: If interested, not vital, an introduction to npm can be found here.

```
npm start
```

- *Example output:*

```
$ npm start
Starting the development server...

Compiled successfully!

You can now view my-app in the browser.

  Local:            http://localhost:3000/
  On Your Network:  http://172.17.0.2:3000/

Note that the development build is not optimized.
To create a production build, use yarn build.
```

- Chrome should automatically be opened and the application rendered!

Stage 2: Token Deployment

Important: The following videos may make note of the use of something called "Docker" and "containers", but do note that Docker has since been omitted. The same commands that are mentioned may be run directly on the machine without entering into the noted container.

Video Tutorial

1. Start up your Ethereum node, ganache
   - Back within your terminal window where the `npm start` command was run, create a new terminal window, or tab, to run your blockchain

---

Note: While within the terminal window select File -> Open Terminal to create a new window.

To create a new tab from within a terminal window:

```
ctrl+shft+t
```

   - *Example output: Result is a new empty terminal, in the same directory you were in.*

```
adam@adam:~/Desktop/blg/wallet-template$
```

   - Then run the blockchain emulation with the below command:

```
ganache-cli
```

- Example output:

```
# ganache-cli
Ganache CLI v6.0.3 (ganache-core: 2.0.2)
[...]
Listening on localhost:8545
```

2. Open the application's code in the Sublime text editor, time to get to some coding...

- Open the Sublime text editor by clicking on the Sublime icon in the left dock.
- From within Sublime open the *wallet-template* folder. Click on `File` in the top left corner and select `Open Folder...` in the menu. Select `Desktop/blg/wallet-template` to open, and we can get to coding!

Video Tutorial

Note:

- A default, and required, initial migration script(src/migrations/1_initial_migration.js), has been included. Do *not* remove this script.

---

3. Write the Deployment Script
   - Create a new file in order to deploy the token, `src/migrations/2_deploy_contracts.js`
     - Simply right-click on the migrations directory and select `New File` from the drop down menu to create the new file.
     - Select `File` => `Save As` and then enter the name of the file: `2_deploy_contracts.js`, click the `Save` button in the bottom right corner.

   Time to add some code to this new file!

   - Import the token's artifacts, line 1

```
const Token = artifacts.require("./Token.sol");
```

   - Define the owner account, line 2 - Note that the `truffle migrate` command exposes the `web3` object globally which is connected to your ganache node

     and therefore has access to all of the default 10 accounts.

```
const owner = web3.eth.accounts[0];
```

   - Utilize truffle's deployer object in order to deploy an instance of the token, line 4-6 - This deployer is a utility that `truffle` provides within its framework to make sending `contract creation` transactions as easy as the one line below! - This function will be exported so that it may be executed by the `truffle migrate` command

```
module.exports = deployer => {
  deployer.deploy(Token, { from: owner });
}
```

Important: Don't forget to save!

This may be done either by selecting `File` => `Save` or via `ctrl+s` on the keyboard.

4. Deploy your Token

Important: Now back within your terminal window where the `truffle test` command was recently run...

Ensure you are still in the `src` directory!

```
truffle migrate --reset
```

- *Example output:*

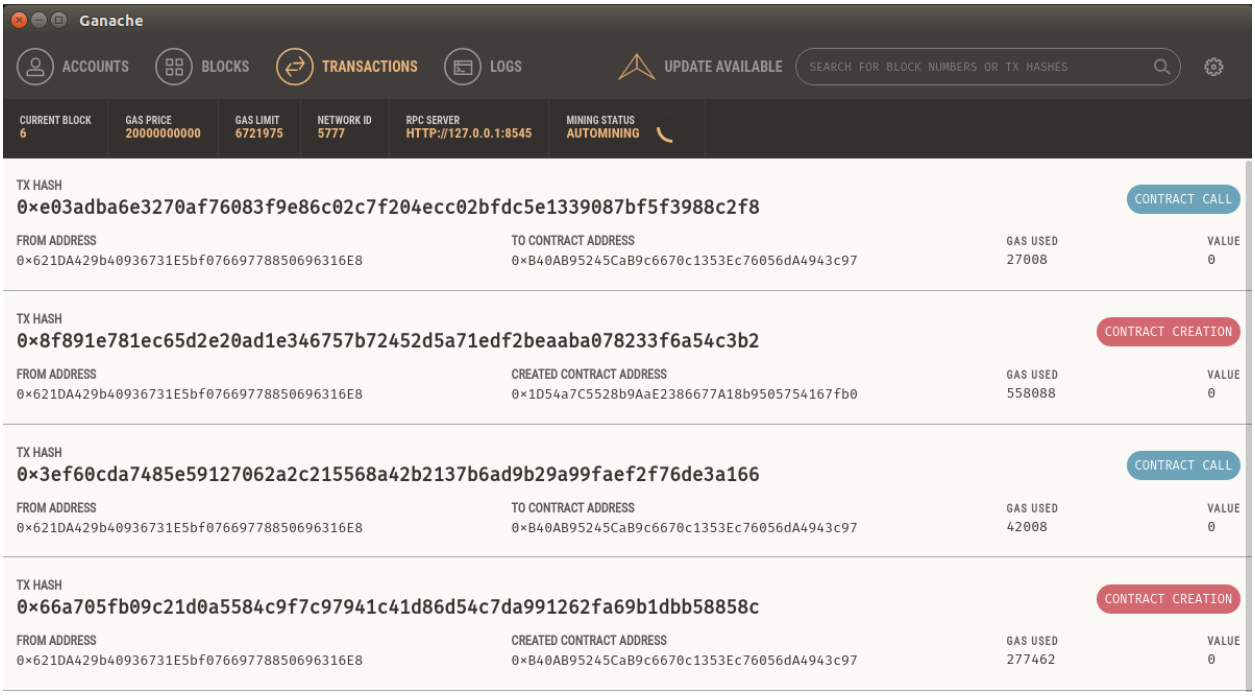
```
# truffle migrate --reset
Using network 'development'.

Running migration: 1_initial_migration.js
  Deploying Migrations...
  ... 0x26ff3f480502a228f34363e938289c3164edf8bc49c75f5d6d9623a05da92dbf
  Migrations: 0x3e47fad1423cbf6bd97fee18ae2de546b0e9188a
Saving successful migration to network...
  ... 0x19a7a819df452847f34815e2573765be8c26bac43b1c10d3b7528e6d952ac02c
Saving artifacts...
Running migration: 2_deploy_contracts.js
  Deploying Token...
  ... 0x4a69e7840d0f96067964fb515ffea1a04a98fc5759849d3308584af4770c8f7b
  Token: 0xd58c6b5e848d70fd94693a370045968c0bc762a7
Saving successful migration to network...
  ... 0xd1e9bef5f19bb37daa200d7e563f4fa438da60dbc349f408d1982f8626b3c202
Saving artifacts...
#
```

Important: You just sent your first *contract creation* transaction via the `truffle` framework, well done!

As above, the `Token` contract has been created at address: `0xd58c6b5e848d70fd94693a370045968c0bc762a7`, note that yours will almost certainly be created at a different address!

Also the `Migrations` contract was created which is nothing to worry about but just a utility the `truffle` framework uses to monitor the status of your transactions.

If you have a look back at ganache and select on the `Transactions` tab you will see the `Contract Creation` transactions that were sent and lots of other data too!



Stage 3: Token Interface

Now you will *connect* your web application to your newly created `Token` contract!

Video Tutorial

Important: Now you should transition back to your Sublime text editor to begin editing your application code.

All of the application development will take place within the `src/App.js` file.

Please begin by opening this file which may be done so simply by double-clicking on it in the left file tree under the `src` folder.

Note you may also close the other files you may still have open `test_buy.js` or `2_deploy_contracts.js`.

Time to start wiring it up!

1. Import the web3 library, `src/app.js` #line 5

```
import Web3 from 'web3';
```

2. Import the token build artifacts into the application, app.js#line 14

```
import tokenArtifacts from './build/contracts/Token.json';
```

3. Create a web3 connection to the local Ethereum node(ganache), app.js#line 26

```
this.web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8545"));
```

4. Detect the current network id that is connected, app.js#line 28-30

Note: This needs to be after the `this.web3 = ...` line above and within the `componentDidMount(){}` function

- i.e.

```
async componentDidMount() {
  this.web3 = new Web3(new Web3.providers.HttpProvider("http://
↪localhost:8545"));

  this.web3.version.getNetwork(async (err, netId) => {

  });
}
```

```
this.web3.version.getNetwork(async (err, netId) => {

});
```

5. Extract the recently deploy token address from the build artifacts, app.js#line 29

Important: This must be placed within the callback of the above `getNetwork` function call where `netId` is available!

i.e.

```
this.web3.version.getNetwork(async (err, netId) => {
  const tokenAddress = tokenArtifacts.networks[netId].address;
});
```

```
const tokenAddress = tokenArtifacts.networks[netId].address;
```

6. Create a client side reference to the contract and save it in the application's state, app.js#line 30-32

Important: This must be placed within the callback of the above `getNetwork` function call where `netId` is available!

i.e.

```
this.web3.version.getNetwork(async (err, netId) => {
  const tokenAddress = tokenArtifacts.networks[netId].address;
  const token = this.web3.eth.contract(tokenArtifacts.abi).at(tokenAddress);
  this.setState({ token });
  console.log(token);
});
```

```
const token = this.web3.eth.contract(tokenArtifacts.abi).at(tokenAddress);
this.setState({ token });
console.log(token);
```

7. Refresh your chrome browser and open up the developer console

This can be accomplished by right-clicking anywhere in the chrome browser and in the dropdown selecting `inspect` or `inspect element` or by utilizing the shortcut: `ctrl+shift+i`.

View in the developer console the token instance is now present

- *Example output:*

```
Contract {_eth: Eth, transactionHash: null, address:
↪"0xd58c6b5e848d70fd94693a370045968c0bc762a7", abi: Array[20]}
```

-

Stage 4: Load Available On-chain Accounts

Video Tutorial

1. Get the available accounts from the web3 connection, this is to go beneath the `getNetwork` block where the token reference was created, line 35 & 37

```
this.web3.eth.getAccounts((err, accounts) => {

});
```

2. Load the available accounts into the user interface

- Import the Material UI MenuItem, line 12

```
import MenuItem from 'material-ui/MenuItem';
```

- Add an availableAccounts arrary into the app's state, line 21

```
availableAccounts: [],
```

- Append all accounts into the UI dropdown menu, line 38-45

Important: The code block below is to go within the callback of the `getAccounts` call above.

i.e.

```
this.web3.eth.getAccounts((err, accounts) => {
  // Append all available accounts
  for (let i = 0; i < accounts.length; i++) {
    this.setState({
      availableAccounts: this.state.availableAccounts.concat(
        <MenuItem value={i} key={accounts[i]} primaryText={accounts[i]} />
      )
    });
  }
});
```

```
// Append all available accounts
for (let i = 0; i < accounts.length; i++) {
  this.setState({
    availableAccounts: this.state.availableAccounts.concat(
      <MenuItem value={i} key={accounts[i]} primaryText={accounts[i]} />
    )
  });
}
```

3. Set the default account

- Add a defaultAccount variable to the state, line 22

```
defaultAccount: 0,
```

Note: The constructor and state should now look like this:

```
constructor(props) {
  super(props)
  this.state = {
    availableAccounts: [],
    defaultAccount: 0,
    token: null, // token contract
  };
}
```

- Set the defaultAccount in the state when the drowdown value changes, this is to be placed within the `handleDropDownChange` function, line 76

```
this.setState({ defaultAccount });
```

-

Stage 5: Token Interaction - GET

Video Tutorial

1. Load the token metadata from the contract
   - Add the token's symbol to the state, line 23

```
tokenSymbol: 0,
```

Important: The calls to the `token` object beneath must be nested within the callback of the `getNetwork` call where the `token` reference was created.

i.e.

```
this.web3.version.getNetwork(async (err, netId) => {
  const tokenAddress = tokenArtifacts.networks[netId].address;
  const token = this.web3.eth.contract(tokenArtifacts.abi).
↪at(tokenAddress);
  this.setState({ token });
  console.log(token);

  // Set token symbol below
  const tokenSymbol = await token.symbol();
  this.setState({ tokenSymbol });

  // Set wei / token rate below
  const rate = await token.rate();
  this.setState({ rate: rate.toNumber() });
});
```

   - Load the token's symbol, line 39-41

```
// Set token symbol below
const tokenSymbol = await token.symbol();
this.setState({ tokenSymbol });
```

- Add the token's rate to the state, line 23

```
rate: 1,
```

- Load the token's rate, line 43-45

```
// Set wei / token rate below
const rate = await token.rate();
this.setState({ rate: rate.toNumber() });
```

-

Stage 6: Load Account Balances

Video Tutorial

1. Load the default account's ETH and Token balances, completing the `loadAccountBalances` function

- Add tokenBalance to the state, line 24

```
tokenBalance: 0,
```

- Add ethBalance to the state, line 23

```
ethBalance: 0,
```

Important: This code below **must** be placed within the `loadAccountBalances` function

i.e.

```
// Load the accounts token and ether balances.
async loadAccountBalances(account) {
  const balance = await this.state.token.balanceOf(account);
  this.setState({ tokenBalance: balance.toNumber() });

  const ethBalance = await this.web3.eth.getBalance(account);
  this.setState({ ethBalance: ethBalance });
}
```

- Set the token balance, line 64-65

```
const balance = await this.state.token.balanceOf(account);
this.setState({ tokenBalance: balance.toNumber() });
```

- Set the eth balance, line 67-68

```
const ethBalance = await this.web3.eth.getBalance(account);
this.setState({ ethBalance: ethBalance });
```

- Call the `loadAccountBalances` function when the page initially renders within the `componentDidMount` function, line 49

```
this.loadAccountBalances(this.web3.eth.accounts[0]);
```

Important: This code must be placed in the `componentDidMount` function after the `token` object has been created.

i.e.

```
async componentDidMount() {
  // Create a web3 connection
  this.web3 = new Web3(new Web3.providers.HttpProvider("http://
↪localhost:8545"));

  this.web3.version.getNetwork(async (err, netId) => {
    const tokenAddress = tokenArtifacts.networks[netId].address;
    const token = this.web3.eth.contract(tokenArtifacts.abi).
↪at(tokenAddress);
    this.setState({ token });
    console.log(token);

    // Set token symbol below
    const tokenSymbol = await token.symbol();
    this.setState({ tokenSymbol });

    // Set wei / token rate below
    const rate = await token.rate();
    this.setState({ rate: rate.toNumber() });

    this.loadAccountBalances(this.web3.eth.accounts[0]);

    [...]
  }
```

- Also load the balances whenever a new account is selected in the dropdown, place this line within the `handleDropDownChange` function, line 94

```
this.loadAccountBalances(this.state.availableAccounts[index].key);
```

  - i.e.

```
handleDropDownChange = (event, index, defaultAccount) => {
  this.setState({ defaultAccount });
  this.loadAccountBalances(this.state.availableAccounts[index].key);
}
```

2. View the default account balances and token information in your browser!

-

Stage 7: Purchasing Tokens

Note: Time to start sending some transactions!

You have already sent a `contract creation` transaction via `truffle migrate` now it is time to send some

`calls`, or transactions that *call* functions of the contract.

Video Tutorial

1. Add token amount to the state, line 21.

```
amount: 0,
```

2. Complete the function to buy tokens, sending a transaction to the token contract, line 82-84. - From within the `buy()` function...

```
const sender = this.web3.eth.accounts[this.state.defaultAccount];
const transactionHash = await this.state.token.buy({ from: sender, value:␣
↪amount });
console.log(transactionHash);
```

Important: This must be added within the `buy()` function.

i.e.

```
async buy(amount) {
  const sender = this.web3.eth.accounts[this.state.defaultAccount];
  const transactionHash = await this.state.token.buy({ from: sender,␣
↪value: amount });
  console.log(transactionHash);
}
```

3. In the GUI buy tokens with several available accounts.

Note: Note transaction hash in the developer console

Example transaction hash: `0x4b396191e87c31a02e80160cb6a2661da6086c073f6e91e9bd1f796e29b0c983`

4. Refresh the browser or select a different account and come back, and view the account's balance of shiny new tokens!

-

Stage 8: Events

Video Tutorial

1. Add an event listener for when tokens are transferred and reload the account's balances when this event is caught, line 76-79 - Add this code within the `loadEventListeners()` function

```
this.state.token.Transfer().watch((err, res) => {
  console.log(`Tokens Transferred! TxHash: ${res.transactionHash} \n ${JSON.
↪stringify(res.args)}`);
  this.loadAccountBalances(this.web3.eth.accounts[this.state.defaultAccount]);
});
```

2. Load the contract events, line 51 - The `loadEventListeners` function must be called initially to create these listeners - Do so from within the `componentDidMount()` function

```
this.loadEventListeners();
```

Important: This call must come after the `token` object is created!

i.e

```
async componentDidMount() {
  // Create a web3 connection
  this.web3 = new Web3(new Web3.providers.HttpProvider("http://
↪localhost:8545"));

  this.web3.version.getNetwork(async (err, netId) => {
    const tokenAddress = tokenArtifacts.networks[netId].address;
    const token = this.web3.eth.contract(tokenArtifacts.abi).
↪at(tokenAddress);
    this.setState({ token });
    console.log(token);

    // Set token symbol below
    const tokenSymbol = await token.symbol();
    this.setState({ tokenSymbol });

    // Set wei / token rate below
    const rate = await token.rate();
    this.setState({ rate: rate.toNumber() });

    this.loadAccountBalances(this.web3.eth.accounts[0]);
    this.loadEventListeners();

    [...]
  }
```

---

3. Buy tokens and view the log confirmation in the developer console and token and ETH balance updated dynamically!

•

Stage 9: Transfer Tokens

Try this portion on your own! [Solutions noted in the following solutions section...]

The required components included:

1. Add the `transferAmount` and `transferUser` to the app's state.
2. Add the React form component in order to transfer tokens.
3. Complete the `transfer` method to send the transfer transaction.

Finally transfer tokens between accounts and review balances.

Stage 10: Basic Routing

1. Add basic routing to render navigate between a new component and the existing wallet component

   Video Tutorial

- Change into the `wallet-template` directory
- Add the `react-router-dom` package to the project

```
npm install --save react-router-dom@4.3.1
```

- *Example output:*

```
wallet-template$ npm install --save react-router-dom@4.3.1
[..]
+ react-router-dom@4.3.1
updated 1 package and audited 13712 packages in 9.686s
found 40 vulnerabilities (31 low, 6 moderate, 3 high)
run `npm audit fix` to fix them, or `npm audit` for details
wallet-template$
```

- Import the router components into the app, within `App.js` at line 2

```
import { BrowserRouter, Route, Link } from 'react-router-dom';
```

- Wrap the existing `{component}` in `App.js` at line 150 with the router, ~line 150 & line 152

```
<BrowserRouter>
  <div>
    {component}
  </div>
</BrowserRouter>
```

Important: The two routes below must go within the `<div>` with the `<BrowserRouter>`.

i.e.

```
<BrowserRouter>
  <div>
    <Route exact={true} path="/" render={() => component}/>
    <Route exact={true} path="/newPage" render={() => <h1>New Page!</
↪h1>} />
  </div>
</BrowserRouter>
```

- Add a *route* for the default page, replace the existing `{component}` line 151 with the route below on line 151

```
<Route exact={true} path="/" render={() => component}/>
```

- Add a secondary route for the new page you are going to create, line 152

```
<Route exact={true} path="/newPage" render={() => <h1>New Page!</h1>} />
```

Note: Give the new route a try! Directly in the browser in the search bar add the path `newPage` to the url resulting in: `http://localhost:3000/newPage`

- Add a button to navigate to the new page from the wallet, add this link at the top of your `component`, ~line 110-112

```
<Link to={'newPage'}>
  <RaisedButton label=">>> New Page" secondary={true} fullWidth={true}/>
</Link>
```

  - It should result in the following:

```
component = <div>
  <Link to={'newPage'}>
    <RaisedButton label=">>> New Page" secondary={true} fullWidth=
↪{true}/>
  </Link>
  <h3>Active Account</h3>
  [...]
```

- Confirm selection of the new button will change the route in the url to `/newPage`

---

2. Create your new page!

- Add a basic component with a link back to the wallet to begin with, add this component beneath the existing `component` just before the return state, line 147-151

```
const newPage = <div>
  <Link to={'/'}>
    <RaisedButton label="Wallet <<<" primary={true} fullWidth={true}/>
  </Link>
</div>
```

- Update your `newPage` route to now render this component, line 162

```
<Route exact={true} path="/newPage" render={() => newPage} />
```

-

Important: All done? We recommend reviewing the complementary video series found here.

Solutions

Stage 10: Transfer Tokens

Video Tutorial

1. Add the transferAmount and transferUser to the app's state, line 28 & 29.

```
transferAmount: '',
transferUser: '',
```

2. Add the React transfer tokens form component, line 128-139.

```
<div>
  <h3>Transfer Tokens</h3>
  <TextField floatingLabelText="User to transfer tokens to." style={{width: 400}}␣
↪value={this.state.transferUser}
    onChange={(e, transferUser) => { this.setState({ transferUser }) }}
```

(continues on next page)

(continued from previous page)

```
  />
  <TextField floatingLabelText="Amount." style={{width: 100}} value={this.state.
↪transferAmount}
    onChange={(e, transferAmount) => { this.setState({ transferAmount })}}
  />
  <RaisedButton label="Transfer" labelPosition="before" primary={true}
    onClick={() => this.transfer(this.state.transferUser, this.state.transferAmount)}
  />
</div>
```

3. Complete the transfer method to send the transfer transaction, line 94-96.

```
const sender  = this.web3.eth.accounts[this.state.defaultAccount];
const transactionHash = await this.state.token.transfer(user, amount, { from: sender }
↪);
console.log(transactionHash);
```

-

-

Bonus

- **Metamask Integration**
  - Ensure Metamask is installed, unlocked and connected to the local client(localhost:8545). - Metamask may be insalled here
  - Fund your Metamask account!

```
$ truffle console
truffle(development> web3.eth.sendTransaction({ from: web3.eth.accounts[0], to:
↪'METAMASK_ADDRESS', value: 1e18 })
```

  - Transfer tokens to your metamask account(from within the application).
  - Add a conditional to use the Metamask web3 provider if present, wallet-template/src/App.js#L35

```
if (window.web3)
    this.web3 = new Web3(window.web3.currentProvider)
else
```

  - Refresh the browser and connect to your Metamask account. View your Metamask account now available within the application.
- **Sync an Ethereum node of your own**

Note: Look to setup a node locally or via Azure. Azure is a nice option to begin with as a node locally can be quite heavy and resource intensive.

  - Getting Started With Azure
  - Sync a Parity node to Kovan
    - Instructions to deploy to Azure here

    * Parity Homepage
- Sync a Geth node to Rinkeby
    * Instructions here
    * Geth Homepage