
Blockchain Learning Group DApp Fundamentals

Feb 03, 2019

Contents

1	Contents	3
1.1	Course Prerequisites	3
1.2	Blockchain Fundamentals	6
1.3	Introduction to React	7
1.4	Introduction to DApp Development	32
1.5	Solidity Exercises	45

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](#)

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-vm-1.1.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-vm-1.1.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

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.

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

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

Note: Note if the keyring dialog pops up and the password is asked for simply click the cancel button, it may ask several times...

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
- SimpleStorage Payable
- Tic-Tac-Toe
- Voting
- Token

1.2.5 5. An Introduction to ReactJS

- ReactJS tutorial

1.2.6 Bonus

1. Blockchain Fundamentals Video Series

1. How it works: Hashes
2. Accounts and Digital Signatures
3. Consensus Algorithms
4. Proof of Work
5. Block Target
6. Decentralized Consensus

2. Deploy your token to a public Test Net(Kovan, Rinkeby, Ropsten) via Metamask!

- Ensure Metamask is installed, enabled and unlocked, you may install from [here](#)
- Ensure Metamask is connected to Kovan via the drop down in the top left corner
- Within remix under the run tab switch from Javascript VM to injected web3
- Refresh the browser
- Now re-deploy and the contract will be sent from your Metamask account.

Note:

- A Metamask window will pop-up for you to confirm the transaction
- Also **SAVE** the address the token was deployed at! You may need it later :)

3. 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](#)

1.3 Introduction to React

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!

1.3.1 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
```

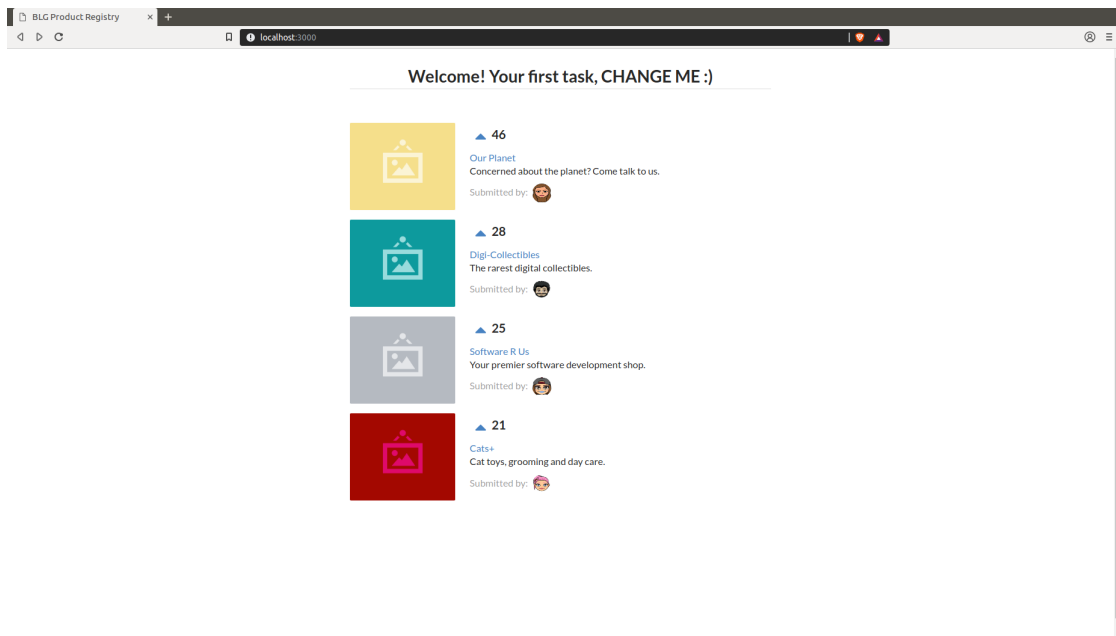
- *Example output:*

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

> product-registry-01@1.0.0 start /home/ajl/Desktop/blg/product-registry-01
  ↳ 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.

1.3.2 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

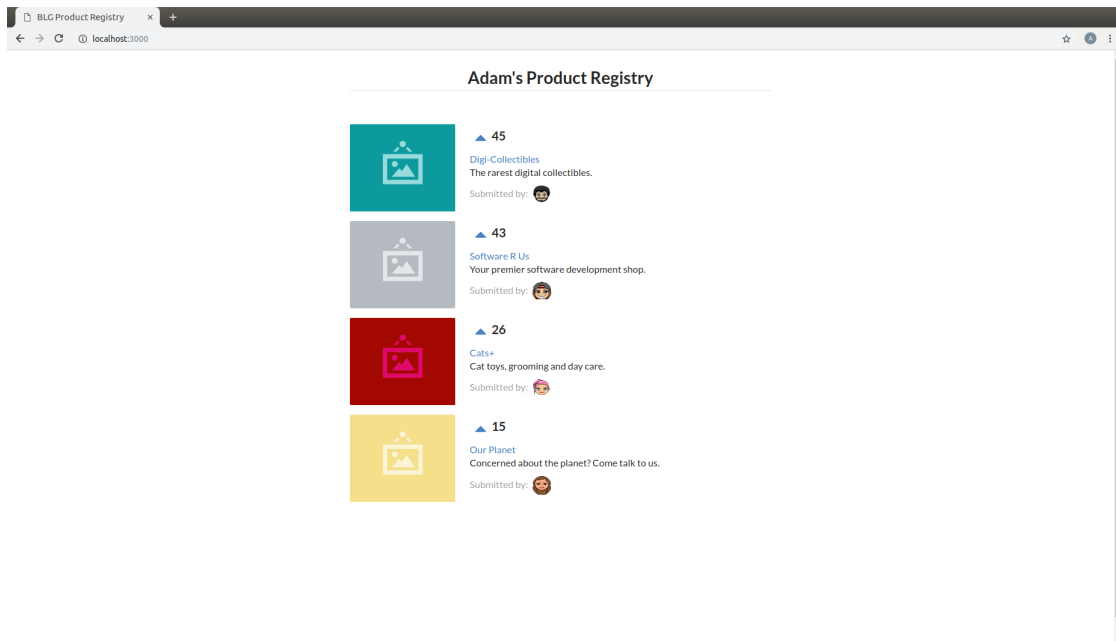
- View the contents of the file.
- Don't worry too much about what is being *linked* within the `<head>` of the file, the core to focus on is between the `<body></body>` tags beneath
- The core of the application may be simplified to the following:

```
<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>
```

•

1.3.3 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-contained 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 available 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:

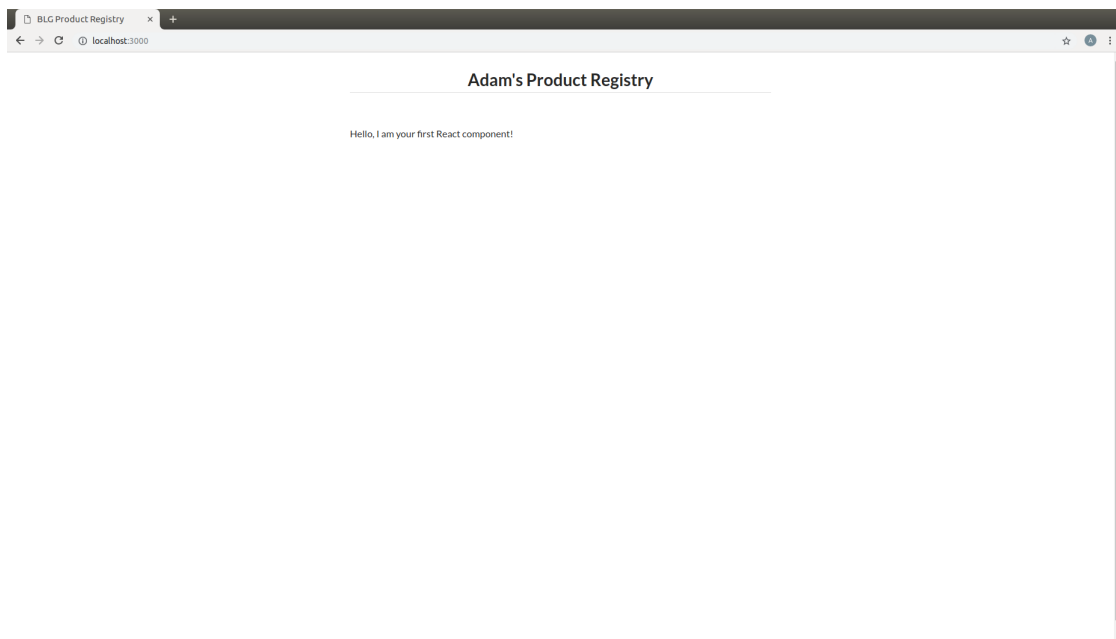
```
ReactDOM.render(
  <ProductRegistry />,
  document.getElementById('content')
);
```

- 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 attempting 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>
```

1.3.4 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!

<ProductList>

<Product>

<Product>

<Product>

<Product>

- 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”

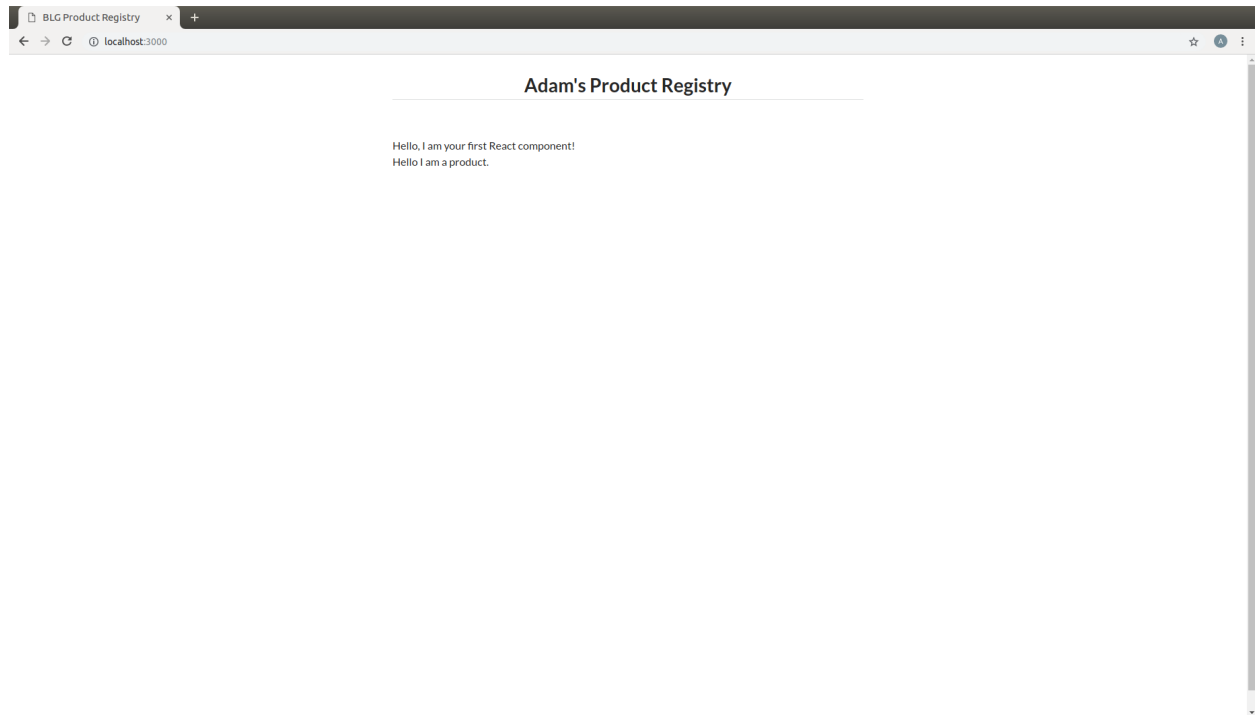
```
class Product extends React.Component {  
  render() {  
    return (  
      <div>Hello I am a product.</div>  
    );  
  }  
}
```

- 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>  
    );  
  }  
}
```



1.3.5 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_url`

```
<a href="https://react.semantic-ui.com/" target="_blank">Semantic UI</a>
```

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 its `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'>
            <span>Submitted by:</span>
          </div>
        </div>
      </div>
    )
  }
}
```

(continues on next page)

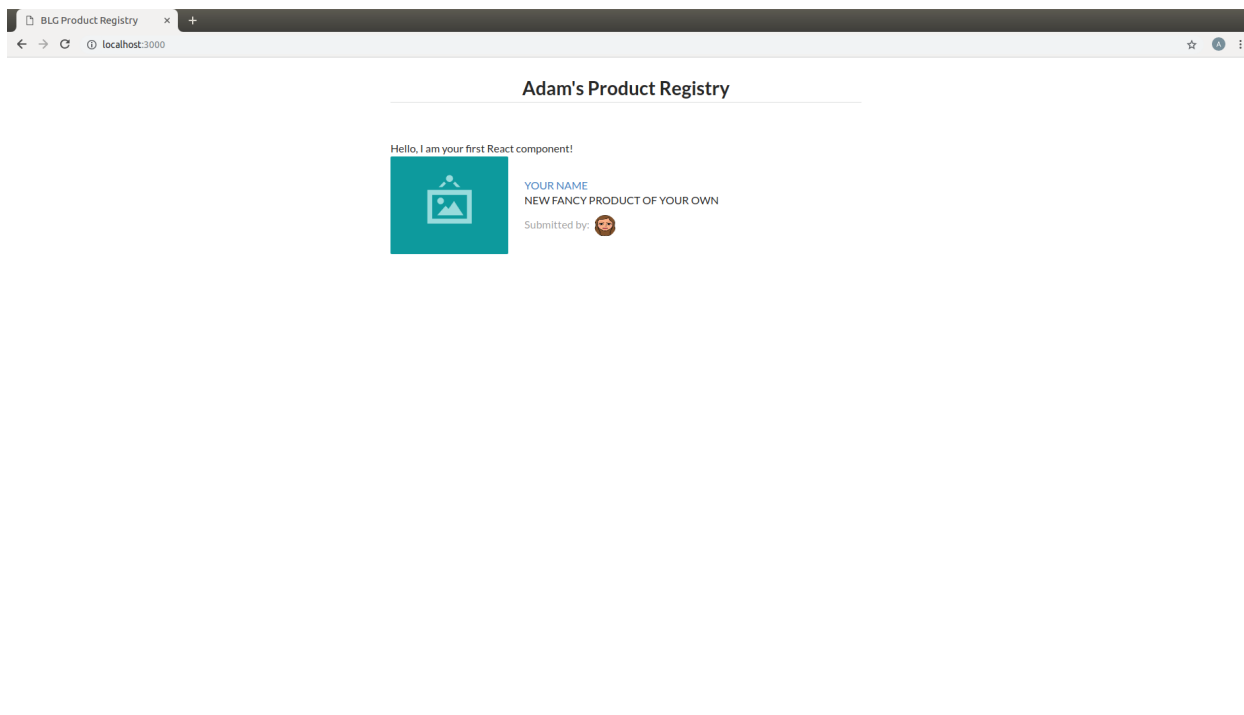
(continued from previous page)

```

        <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: `` and the submitted image: `` 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!



•

1.3.6 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 DESCRIPTION.',
  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 its 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 accept 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 this 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.imageUrl} />
</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.
-

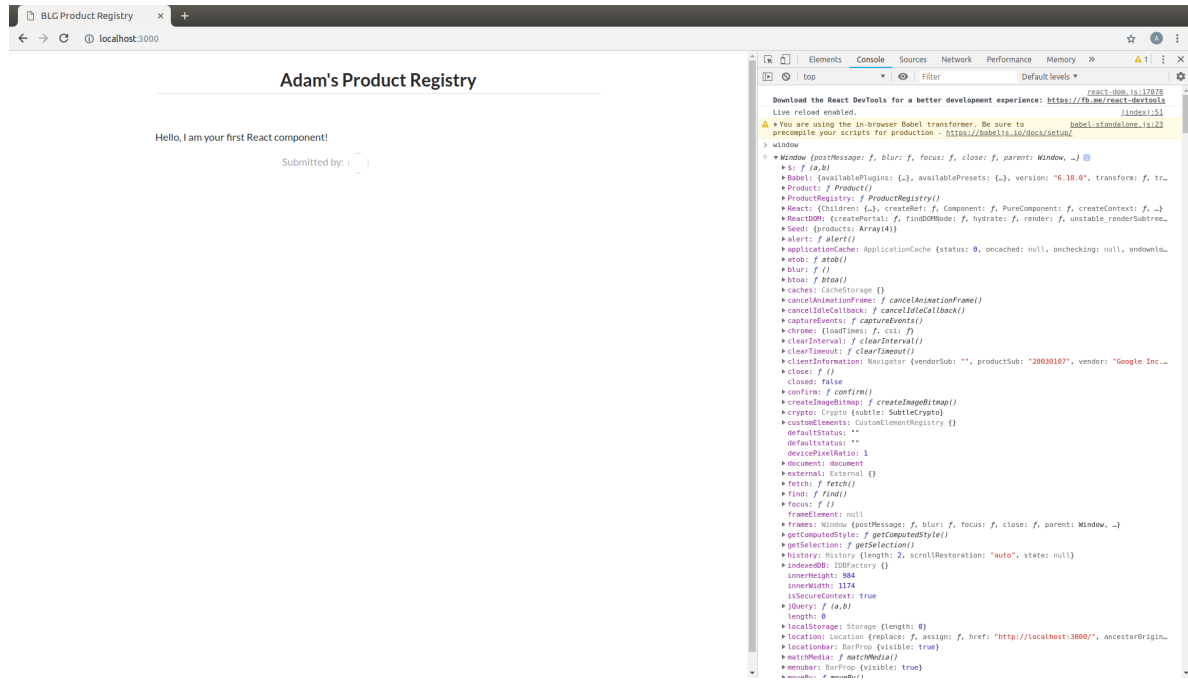
Understanding seed.js and the window

- 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 available 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:

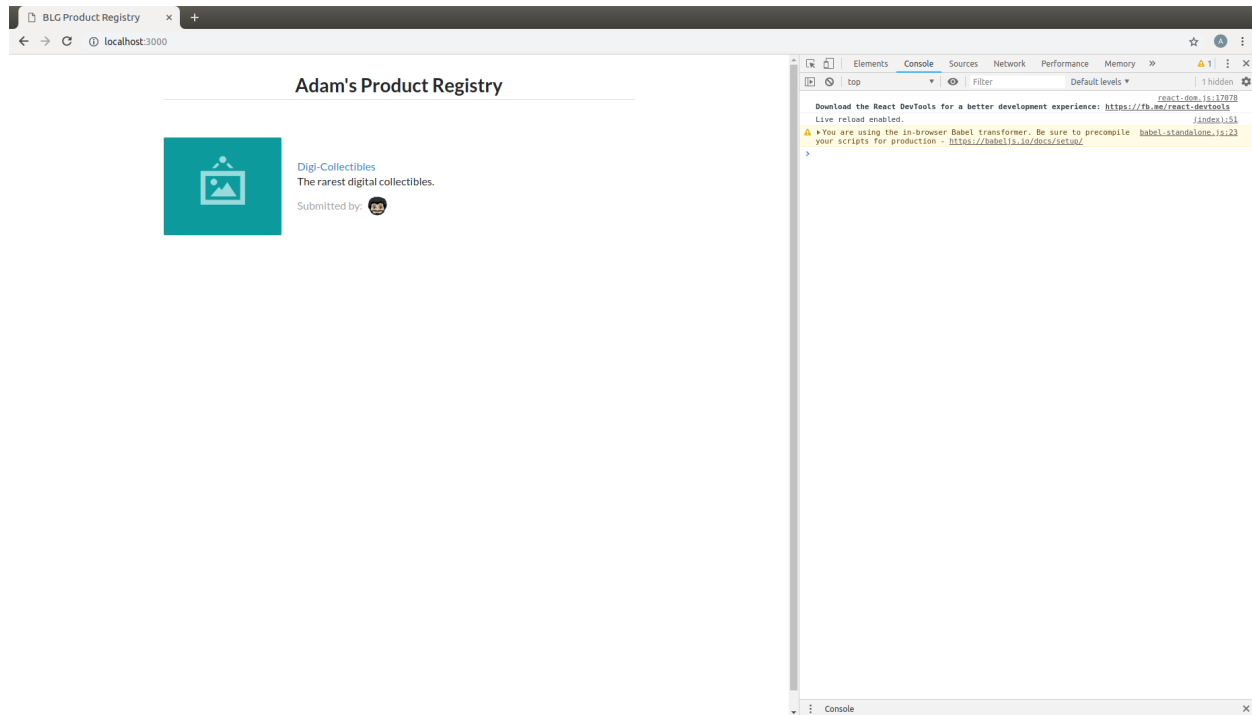
```
<Product
  title={product.title}
  description={product.description}
  submitterAvatarUrl={product.submitterAvatarUrl}
  productImageUrl={product.productImageUrl}
/>
```

- 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>
    );
  }
}
```

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



•

1.3.7 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>
);
```

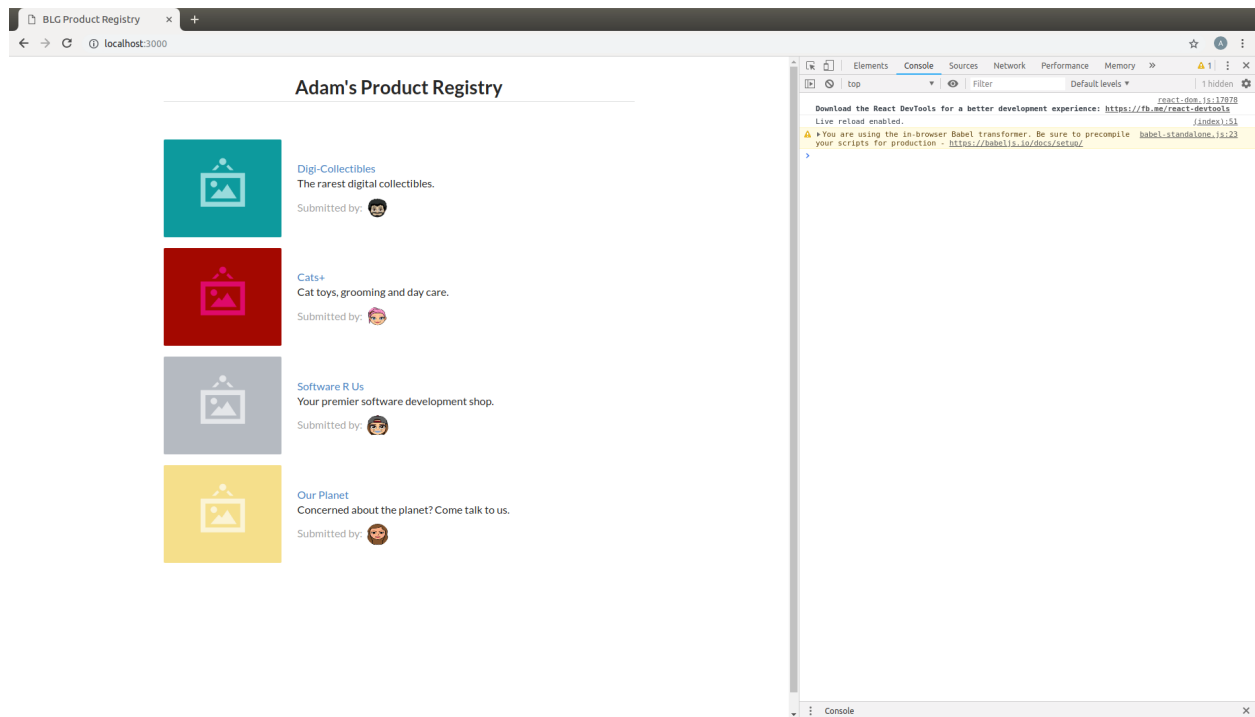
- Now you will notice an error in the browser console stating: Warning: Each child in an array or iterator should have a unique "key" prop.
- The use of the key prop is something that the React framework uses to identify each instance of the Product component.
- For the time being just remember that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

- And all seeded products should be rendered!



1.3.8 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'>
        <div className='image'>
          <img src={this.props.productImageUrl} />
        </div>
        <div className='middle aligned content'>
          <div className='header'>
            <a>
```

(continues on next page)

(continued from previous page)

```

        <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.

```

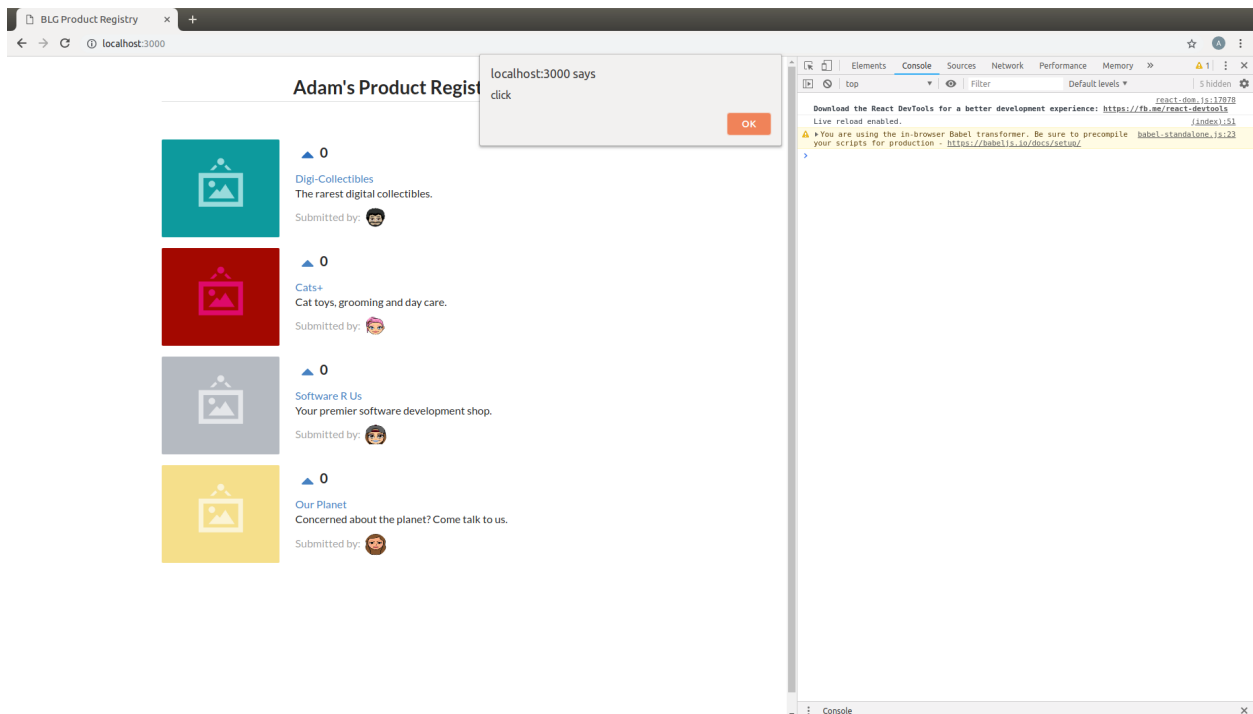
<div className='header'>
  <a onClick={() => alert('click')}>
    <i className='large caret up icon' />
  </a>
  {this.props.votes}
</div>

```

- 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 (  
      <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 occurred 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
  };

```

(continues on next page)

(continued from previous page)

```

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 {
    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() {
```

(continues on next page)

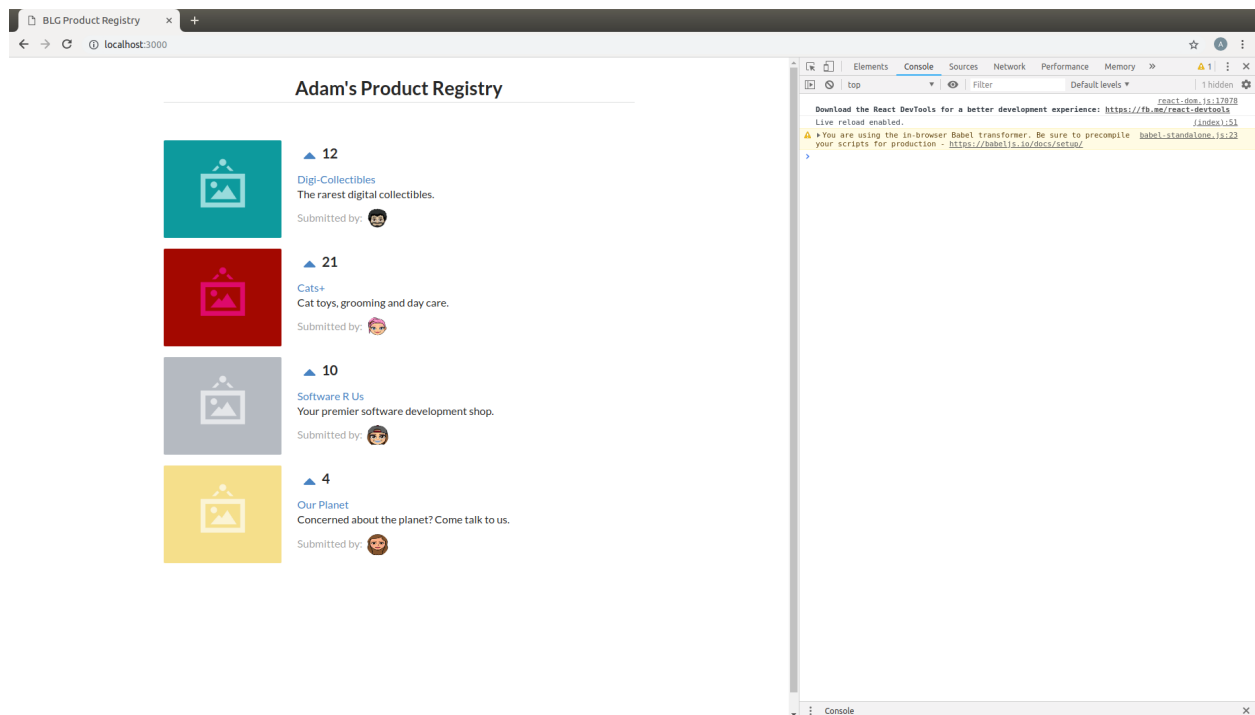
(continued from previous page)

```

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!



1.4 Introduction to DApp Development

[View Completed Wallet Demo](#)

1.4.1 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

```
yarn start
```

- *Example output:*

```
$ yarn 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!

3. 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 *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!

END Stage 1: Starting the Application!

1.4.2 Stage 2: Testing Your Token

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. Create a new tab in your terminal window or a new terminal window for our Ethereum node

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+shift+t
```

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

```
adam@adam: ~/Desktop/blg$
```

2. Start up your Ethereum node, ganache-cli

```
ganache-cli
```

- *Example output:*

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

3. Create a new window or tab for our Truffle commands

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+shift+t
```

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

```
adam@adam: ~/Desktop/blg$
```

4. Create the Test Case

Note:

- contracts/Token.sol has been provided.
- Also one test file template has been provided in order to test the buy method was implemented correctly.

- Open the test file within Sublime, src/test/test_buy.js
- Import the token's build artifacts, src/test/test_buy.js line 2

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

- Define the owner account, note truffle test exposes the accounts array for us, line 6

```
const owner = accounts[0]
```

- Create a new instance of the token contract, line 10

```
const token = await Token.new({ from: owner })
```

- Specify the wei value of tokens you wish to purchase, line 13

```
const value = 100
```

- Send the transaction to the token's buy method, line 16

```
const txResponse = await token.buy({ from: owner, value })
```

- Pull the rate from the token, line 19

```
const rate = await token.rate()
```

- Compute the token amount to be minted to the buyer, line 22

```
const tokenAmount = value * rate
```

- Access the event object from the transaction receipt, line 25

```
const event = txResponse.logs[0]
```

- Assert the correct values were emitted, line 28-31

```
assert.equal(event.event, 'TokensMinted', 'TokensMinted event was not emitted.')
assert.equal(event.args.to, owner, 'Incorrect to was emitted.')
assert.equal(event.args.value, value, 'Incorrect value was emitted.')
assert.equal(event.args.totalSupply.toNumber(), tokenAmount, 'Incorrect totalSupply_
↳was emitted.')
```

Ensure the state of the contract is updated correctly

- Assert the buyer's balance is correct, line 34-35

```
const balance = await token.balanceOf(owner)
assert.equal(balance.toNumber(), tokenAmount, 'Incorrect token balance.')
```

- Assert the total supply is correct, line 38-39

```
const supply = await token.totalSupply()
assert.equal(supply.toNumber(), tokenAmount, 'Incorrect total supply.')
```

5. Execute the Test Case

- Change into the `src` directory

```
cd src
```

- Execute the test

```
truffle test
```

- *Example output:*

```
$ truffle test
Using network 'development'.
Contract: Token.buy()
  ✓ should buy new tokens. (133ms)
1 passing (148ms)
#
```

END Stage 2: Testing Your Token

1.4.3 Stage 3: Token Deployment

Video Tutorial

Note:

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

1. 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 create the new file.
- Import the token's artifacts, line 1

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

- Define the owner account, note `truffle migrate` exposes the `web3` object, line 2

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

- Utilize `truffle`'s `deployer` object in order to deploy an instance of the token, line 4-6

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


2. Deploy your Token

```
truffle migrate
```

- *Example output:*

```
# truffle migrate
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...
#
```

END Stage 3: Token Deployment

1.4.4 Stage 4: Token Interface

Video Tutorial

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

- Open `src/app.js` in the Sublime text editor

```
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-cli), `app.js` #line 26

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

4. Check if the connection was successful, `app.js` #line 28-30

```
if (this.web3.isConnected()) {  
  
}
```

5. Detect the current network id that is connected, app.js#line 29-31

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

6. Extract the recently deploy token address from the build artifacts, app.js#line 30-33

```
// Create a reference object to the deployed token contract  
if (netId in tokenArtifacts.networks) {  
  const tokenAddress = tokenArtifacts.networks[netId].address  
}
```

7. Create a client side reference to the contract and save it in the application's state, app.js#line 33-35

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

8. 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]}
```

END Stage 4: Token Interface

1.4.5 Stage 5: Load Available On-chain Accounts

Video Tutorial

1. Get the available accounts from the web3 connection, this is to wrap the existing token interface code, line 29 & 39

```
this.web3.eth.getAccounts((err, accounts) => { // Line 29
  })                                           // Line 39
```

2. Set the default account to use, line 30

```
const defaultAccount = this.web3.eth.accounts[0]
```

3. Load the available accounts into the user interface

- Import the Material UI MenuItem, line 8

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

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

```
availableAccounts: [],
```

- Append all accounts into the UI dropdown menu, line 34-41

```
// 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]} />
    )
  })
}
```

4. Set the default account

- Add a defaultAccount variable to the state, line 22

```
defaultAccount: 0,
```

- Set the defaultAccount in the state when the dropdown value changes, line 86

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

END Stage 5: Load Available Accounts

1.4.6 Stage 6: 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,
```

- Load the token's symbol, line 52-55

```
// Set token symbol below
token.symbol((err, tokenSymbol) => {
  this.setState({ tokenSymbol })
})
```

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

```
rate: 1,
```

- Load the token's rate, line 58-61

```
// Set wei / token rate below
token.rate((err, rate) => {
  this.setState({ rate: rate.toNumber() })
})
```

END Stage 6: Token Interaction - GET

1.4.7 Stage 7: Load Account Balances

Video Tutorial

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

- Confirm the token has been loaded, line 73-75

```
if (this.state.token) {
}
```

- Add tokenBalance to the state, line 24

```
tokenBalance: 0,
```

- Set the token balance, line 75-78

```
// Set token balance below
this.state.token.balanceOf(account, (err, balance) => {
  this.setState({ tokenBalance: balance.toNumber() })
})
```

- Add ethBalance to the state, line 23

```
ethBalance: 0,
```

- Set the eth balance, line 81-84

```
// Set ETH balance below
this.web3.eth.getBalance(account, (err, ethBalance) => {
  this.setState({ ethBalance })
})
```

- Call the loadAccountBalances method on load, line 67

```
this.loadAccountBalances(defaultAccount)
```

- Also load the balances whenever a new account is selected in the dropdown, line 111

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

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

END Stage 7: Load Available Account Balances

1.4.8 Stage 8: Purchasing Tokens

Video Tutorial

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

```
amount: 0,
```

2. Complete the method to buy tokens, sending a transaction to the token contract, line 99-104.

```
this.state.token.buy({
  from: this.web3.eth.accounts[this.state.defaultAccount],
  value: amount
}, (err, res) => {
  err ? console.error(err) : console.log(res)
})
```

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!

END Stage 8: Purchasing Tokens

1.4.9 Stage 9: Events

Video Tutorial

1. Add an event to listen for when tokens are transferred and reload the account's balances, line 94-99

```
// Watch tokens transfer event below
this.state.token.Transfer({ fromBlock: 'latest', toBlock: 'latest' })
.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 66

```
this.loadEventListeners()
```

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

END Stage 9: Events

1.4.10 Stage 10: Transfer Tokens

Try this portion on your own! [Solution noted at the bottom]

The required components included:

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

Finally transfer tokens between accounts and review balances.

END Stage 10: Transfer Tokens

1.4.11 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 150-161.

```
<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 }) }}
  />
  <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 117-124.

```
if (amount > 0) {
  // Execute token transfer below
  this.state.token.transfer(user, amount, {
    from: this.web3.eth.accounts[this.state.defaultAccount]
  }, (err, res) => {
    err ? console.error(err) : console.log(res)
  })
}
```

Complete Wallet Solution

- git clone https://github.com/Blockchain-Learning-Group/wallet-eod2.git
- cd wallet-eod2
- git checkout tags/2.0

1.4.12 Bonus: Extend Your Wallet

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

1.1 Add the routing components

Video Tutorial

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

```
yarn add react-router-dom@4.3.1
```

- *Example output:*

```
wallet-template$ yarn add react-router-dom@4.3.1
yarn add v1.2.0
[1/4] Resolving packages...
[...]
Done in 5.34s.
wallet-template$
```

- Import the router components into the app, line 2

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

- Wrap components with the router, ~line 172 & line 179

```
<BrowserRouter>
</BrowserRouter>
```

- Add a button to navigate to the exchange route, ~line 137-139

```
<Link to={'exchange'}>
  <RaisedButton label="">>> Exchange" secondary={true} fullWidth={true}/>
</Link>
```

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

1.2. Create the exchange component and the routes

Video Tutorial

- Add a template exchange component with a link back to the wallet, line 173-177

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

- Add a default route, line 186

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

- And an exchange route, line 187

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

2. Metamask Integration

- Ensure Metamask is installed, unlocked and connected to the local client(localhost:8545). - Metamask may be installed [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.

3. 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](#)
-

1.5 Solidity Exercises

1.5.1 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 delayed 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 delayed 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 delayed 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](#).

1.5.2 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 modifier.

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](#).

1.5.3 Tic Tac Toe

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

- Solidity Types Video Tutorial
- Tic-Tac-Toe Part 1 of 2 Video Tutorial

1. Create the contract and initial storage variables, line 1-15

- Empty Contract Video Tutorial
- Storage Variables Video Tutorial

```
pragma solidity 0.4.24;

contract TicTacToe {

    address public player1_;
    address public player2_;

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

2. Create a function to allow a game to be started, line 16-19

- Video Tutorial

```
function startGame(address _player1, address _player2) external {
    player1_ = _player1;
    player2_ = _player2;
}
```

Important:

- Video Tutorial [3-7]
-

3. Now players need to be able to take a turn, specifying where they want to place their x or o, line 21-25

- 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, line 26

```
gameBoard[_boardLocation] = msg.sender;
```

5. Add a function to return the contents of the game board, line 29-31

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

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

- 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 Part 2 of 2 Video Tutorial
 - Video Tutorial [8-12]
-

7. Require that only player 1 or player 2 may take turns, within the `takeTurn` function line 26

```
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 line 27

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

9. Add a storage variable to track who just took a turn, line 8

```
address public lastPlayed_;
```

10. Following a turn being taken update the storage variable, within the `takeTurn` function line 31

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

11. Check that the same player is not trying to take another turn, within the `takeTurn` function line 29

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

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

Important:

- Video Tutorial [13-17]

12. First define which combinations within the game board, which indexes, define a “win”, line 35-49

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

(continues on next page)

(continued from previous page)

```
*
* 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 combinations as filters to filter the board with, line 50-56

```
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 line 57-59

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

15. Add a storage variable to define the winner, line 9

```
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, line 60-66

```
if (
    gameBoard_[filter[0]]==player &&
    gameBoard_[filter[1]]==player &&
    gameBoard_[filter[2]]==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, line 35-37

- Video Tutorial

```
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?

Important:

- [Video Tutorial \[19-24\]](#)

18. Add a storage variable to signify the game has ended, line 10

```
bool public gameOver_;
```

19. If a winner was found update that the game has ended, within the `takeTurn` function line 38

```
gameOver_ = true;
```

20. Add a storage variable to count how many turns have been taken, will use this variable to define if a draw has occurred, line 11

```
uint256 public turnsTaken_;
```

21. After a turn is taken update the turns taken storage variable, within the `takeTurn` function line 36

```
turnsTaken_++;
```

22. Add a conditional that if 9 turns have been taken the game has ended with no winner, within the `takeTurn` function line 41-43

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

23. Add a last pre condition check that the game is still active, within the `takeTurn` function line 30

```
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

OK how about a friendly wager!

Important:

- [Video Tutorial \[25-26\]](#)
-

24. Add a storage variable to hold the placed wagers, line 12

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

25. Add a function to allow the players to place a wager, line 82-86

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

Important:

- [Video Tutorial \[27-28\]](#)
-

26. Update the logic if a winner is found to transfer all the value to them, within the `takeTurn` function line 43

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

27. Update the logic to refund the value if a draw has occurred, within the `takeTurn` function line 46-47

```
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?

1.5.4 Voting

Voting Exercise

- [View Final Solution Demo](#)

[Video Tutorial\[1 - 3\]\[no audio\]](#)

1 Copy the exercise over to remix.

2 Define the duration of the vote, Line 7

```
uint256 public constant VOTE_DURATION = 2 minutes;
```

3 Complete the castVote method, beginning on Line 36

- 3a When a vote has been cast increment that candidates total, Line 41

```
candidateTotals[_candidate] += 1;
```

- 3b Create an event for when a vote is cast, Line 18

```
event VoteCast(address voter, string votedFor);
```

- 3c Emit an event that a new vote has been cast, Line 46

```
emit VoteCast(msg.sender, candidateIds[_candidate]);
```

- 3d Run the contract, deploying within remix and test the castVote method.

[Video Tutorial\[4 - 6\]\[no audio\]](#)

4 Complete the tallyVote method, starting at Line 59

- 4a Add a for loop to find the winner of the vote, Lines 61 - 65

```
for (uint8 i; i < candidates_.length; i++) {
  if (candidateTotals[i] > candidateTotals[currentWinner]) {
    currentWinner = i;
  }
}
```

- 4b Set the winner, Line 70

```
winner_ = candidateIds[currentWinner];
```

- 4c Emit an event that the vote has completed, Line 75

```
emit VoteComplete(winner_);
```

5 Add other candidates to the vote, Line 32

```
candidates_.push("YOUR NAME");  
candidateIds_[1] = "YOUR NAME";
```

6 Run the contract, deploying within remix and test the castVote method and tallyVote methods

- Confirm candidates
 - Cast several votes and after each confirm the total for the candidate has increased
 - Tally the vote before the duration has elapsed
 - Tally the vote after the duration has and view the winner
 - Attempt to cast votes after the duration has elapsed
-

Solution

Voting Exercise Solution

1.5.5 Token

Token Exercise

- [View Final Solution Demo](#)

[Video Tutorial \[1-6\]\[no audio\]](#)

1 Copy the exercise over to remix.

2 Compile and deploy the contract. Confirm variables and methods are available.

3 Update the contract metadata to be your own! Line 8 & 9.

```
string public constant symbol = 'YOUR NAME';  
string public constant name = 'YOUR NAME Token';
```

4 Specify the rate for the purchase of your token, line 14

```
uint public constant rate_ = 2; // rate of token / wei for purchase
```

5 Complete the buy method.

- May purchase only with > 0 ETH, line 46

```
require(msg.value > 0, 'Cannot buy with a value of <= 0, Token.buy()');
```

- Compute the amount of tokens to mint, line 49

```
uint256 tokenAmount = msg.value * rate_;
```

- Update the total supply and the user's balance, line 52 & 53

```
totalSupply_ += tokenAmount; // NOTE overflow
balances_[msg.sender] += tokenAmount; // NOTE overflow
```

- Finally emit events to notify the outside world, line 56 & 57

```
emit TokensMinted(msg.sender, msg.value, totalSupply_);
emit Transfer(address(0), msg.sender, msg.value);
```

6 Compile, deploy and confirm you can purchase your token. Confirm balance updated in balances mapping.

Video Tutorial [7-10][no audio]

7 Complete the transfer method.

- Ensure from address has a sufficient balance, line 70

```
require(balances_[msg.sender] >= _value, 'Sender balance is insufficient, ↩Token.transfer()');
```

- Update the from and to balances, line 73 & 74

```
balances_[msg.sender] -= _value; // NOTE underflow
balances_[_to] += _value; // NOTE overflow
```

- Finally emit an event of the transfer, line 77

```
emit Transfer(msg.sender, _to, _value);
```

8 Compile and deploy and confirm buy and transfer working.

9 Note error output if insufficient balance and other errors correct.

10 Usage

1. Purchase of tokens
2. Transfers

Important: But how can you get your hard earned ETH out of the contract that has been accumulating as tokens have been sold?!

11 Add a withdraw method, and claim the ETH sent to the contract! Line 102

- Solution below...

Important: Save this contract to disk if you wish to use it again! However a completed token will be made available should you wish.

Solutions

Token Exercise Solution

11 Token Withdraw method

- Confirm only the owner may withdraw, line 104

```
require(msg.sender == owner_, "only the owner may withdraw");
```

- Transfer the balance of the contract(this) to the wallet, line 107

```
_wallet.transfer(address(this).balance);
```