React Course

Review of Javascript - ES6

Syntax

- `let` and `const`: they are local scope compared to `var`
- Template Literal:

```
let message = `${student.name} please see ${teacher.name} in
${teacher.room} to pick up your report card.`;
```

- Destructuring:

```
const point = [10, 25, -34];

const [x, y, z] = point;

console.log(x, y, z);
```

- for ... in loop:

```
Array.prototype.decimalfy = function() {
  for (let i = 0; i < this.length; i++) {
    this[i] = this[i].toFixed(2);
  }
};

const digits = [0, 1];

for (const index in digits) {
  console.log(digits[index]);
}
```

Above prints:

> 0
> 1
> function() {*
> for (let i = 0; i < this.length; i++) {*
> this[i] = this[i].toFixed(2);*
> }
> }

Explain:

The for...in loop can get you into big trouble when you need to add an extra method to an array (or another object). Because for...in loops loop over all enumerable properties, this means if you add any additional properties to the array's prototype, then those properties will also appear in the loop.

- For ... of loop: loop over any type of data that is iterable

```
const digits = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];

for (const digit of digits) {
  console.log(digit);
}
```

::For of loop will only loop through the values in object. The problem above with `decimalfy` function will not exist.::

- Spread Operator: gives you the ability to expand, or spread, utterable objects into multiple elements.

```
const books = ["Don Quixote", "The Hobbit", "Alice in Wonderland",
"Tale of Two Cities"];
console.log(...books);

//prints Don Quixote The Hobbit Alice in Wonderland Tale of Two Cities
```

- Concatenate multiple arrays

```
const fruits = ["apples", "bananas", "pears"];
const vegetables = ["corn", "potatoes", "carrots"];
const produce = fruits.concat(vegetables);
console.log(produce);
```

```
// use spread operator to simplify
const fruits = ["apples", "bananas", "pears"];
const vegetables = ["corn", "potatoes", "carrots"];
const produce = [...fruits, ...vegetables]
console.log(produce);

//print ["apple","bananas", "pears", "corns", "potatoes", "carrots"]
```

- Rest parameter: written with three consecutive dots (...), allows you to represent an indefinite number of elements as an array. This can be helpful in a couple of different situations.

```
const order = [20.17, 18.67, 1.50, “cheese”, “eggs”, “milk”, “bread”];
const [total, subtotal, tax, …items] = order;
console.log(total, subtotal, tax, items);
//print 20.17 18.67 1.5 ["cheese", "eggs", "milk", "bread"]
```

- Variadic functions: function takes indefinite number of arguments

```
printPackageContents('cheese', 'eggs', 'milk');
function printPackageContents(...items) {
  for (const item of items) { console.log(item); }
}
```

Function

Lesson1 - Why React

- Outline:
    1. Its compositional model

2. Its declarative nature
3. The way data flows through a Component
4. Composing functions to return UI
5. React is just javascript

1. Benefits of Composition

> Composition is to combine simple functions to build more complicated ones. So that each function only does "one thing"

- Key ingredients:
  1. simple functions
  2. combined to create another function

Example:

```
function getProfileLink (username) {
 return 'https://github.com/' + username
}

function getProfilePic (username) {
 return 'https://github.com/' + username + '.png?size=200'
}

function getProfileData (username) {
 return {
 pic: getProfilePic(username), //composition
 link: getProfileLink(username) //composition
 }
}
```

Do one thing:
* getProfileLink – just builds up a string of the user's GitHub profile link
* getProfilePic – just builds up a string the user's GitHub profile picture
* getProfileData – returns a new object

2. React & Composition

React makes use of the power of composition, heavily! React builds up pieces of a UI using **components**.

Simple components:

```
<Page />
<Article />
<Sidebar />
```

With composition to create a more complex component:

```
<Page>
 <Article />
 <Sidebar />
</Page>
```

3. Imperative Code

When JavaScript code is written *imperatively*, we tell JavaScript exactly **what** to do and **how** to do it. Think of it as if we're giving JavaScript *commands* on exactly what steps it should take. For example, I give you the humble for loop:

```
const people = ['Amanda', 'Farrin', 'Geoff', 'Karen', 'Richard',
'Tyler']
const excitedPeople = []

for (let i = 0; i < people.length; i++) {
 excitedPeople[i] = people[i] + '!'
}
```

Above is imperative code: I have to tell exactly what to do such as setting initial value i = 0, tell for loop when to stop and what to do inside.

4. Declarative Code

- With declarative code, we don't code up all of the steps to get us to the end result. Instead, we *declare* what we **want** done, and JavaScript will take care of doing it.

Recall that the end goal that we want is an array of the same names but where each name ends with an exclamation mark:

```
[“Amanda!”, “Farrin!”, “Geoff!”, “Karen!”, “Richard!”, “Tyler!”]
```

To get us from the starting point to the end, we’ll just use JavaScript’s .map() function to declare what we want done.

```
const excitedPeople = people.map(name => name + ‘!’)
```

That’s it! Notice that with this code we haven’t:
* created an iterator object
* told the code when it should stop running
* used the iterator to access a specific item in the people array
* stored each new string in the excitedPeople array

All of those steps are taken care of by JavaScript’s .map() Array method.

- The below code shows how react is delcarative

```
<button onClick={activateTeleporter}>Activate Teleporter</button>
```

Notice that there’s just an onClick attribute on the button...we aren’t using .addEventListener() to set up event handling with all of the steps involved to set it up. ::Instead, we’re just declaring that we want the activateTeleporter function to run when the button is clicked.::

5. Unidirectional Data Flow

- Two ways data binding:

Front-end frameworks like Angular and Ember make use of two-way data bindings. In two-way data binding, the data is kept in sync throughout the app no matter where it is updated. If a model changes the data, then the data updates in the view. Alternatively, if the user changes the data in the view, then the data is updated in the model. ::Two-way data binding sounds really powerful, but it can make the application harder to reason about and know where the data is actually being updated.::

- React data flow: ~unidirectional data flow~

> In React, the data flows in one direction: from the parent component to a child component. Data updates are sent to parent component where the parent performs the actual changes.

[image:B71682F2-315A-42AB-87A0-C1C973781E21-526-00003126C9F06CCE/58BE6022-5590-4119-BFCD-F29DC93EEB4D.png]

Logics: Parent makes the changes

- The data lives in the parent component and is passed down to the child component. ::Even though the data lives in the parent component, both the parent and the child components can use the data.::
- If the data must be updated, then only the parent component should perform the update. ~If the child component needs to make a change to the data, then it would send the updated data to the parent component where the change will actually be made. Once the change *is* made in the parent component, the child component will be passed the data~ (that has just been updated!).

Having the data flow in one direction and having one place where the data is modified makes it much easier to understand how the application works.

Lesson2 - Rendering UI

React uses JavaScript objects to create React elements. We'll use these React elements to describe what we want the page to look like, and React will be in charge of generating the DOM nodes to achieve the result.

The React code that we write is declarative because we aren't telling React *what* to do; instead, we're writing React elements that describe what the page should look like, and React does all of the implementation work to get it done.

1. Creating Elements and JSX

1.1 Creating Elements

::createElement method takes in a description of an element and returns a plain JavaScript object.::

::After calling createElement, no real DOM element is created until rendering. Nothing is created in the DOM yet.::

```
* Virtual Dom: objects that describe real DOM nodes.
```

```
React.createElement( /* type */, /* props */, /* content */);
```

```
* Type - either a string (such as 'p', 'span', 'header') or a React Component
* Props - either null or an object
* content - null, a string or a React Element or a React Component
> Anything that you pass here will be the content of the rendered element. This can include plain text, JavaScript code, other React elements, etc.
```

Example1:

```
import React from 'react'
import ReactDOM from 'react-dom'

//div is the tag name
const element = React.createElement('div', {className: 'welcome-message'}, 'hello world')

ReactDOM.render(element, document.getElementById('root'))
```

**Question: how is the element being displayed in the page?

ReactDOM helps to render on the server, native device and VR environment.

**Question: where is root coming from?

Apps built with React typically have a single root DOM node. For example, an HTML file may contain a

with the following:

```
<div id='root'></div>
```

So, together with above React code, the element in the console will be like:

```
<div id='root'>
   <div class='welcome-message'>hello world</div>
</div>
```

```
* All Supported HTML Attributes: [DOM Elements – React]
(https://reactjs.org/docs/dom-elements.html#all-supported-html-
attributes)
```

Example2: nesting

```
import React from 'react'
import ReactDOM from 'react-dom'

const element = React.createElement('ol', null,
React.createElement('li', null, 'tyler'), React.createElement('ol',
null, 'karen'), React.createElement('ol', null, 'richard'))

ReactDOM.render(element, document.getElementById('root'))
```

Renders:

```
1. tyler
2. kren
3. richard
```

Example3: display array element as list

```
import React from 'react'
import ReactDOM from 'react-dom'

const people = [{name: 'tyler'}, {name:
karen'}]

const element = React.createElement('ol', null,
    people.map((person) => (React.createElement('li', null,
person.name))))

ReactDOM.render(element, document.getElementById('root'))
```

In the above code, things will be displayed however, with a console warning: Each child in an array or iterator should have a unique 'key' prop. Remember that in the `createElement` function, the second argument is the prop field. Simply give person.name a unique key like:

```
const element = React.createElement('ol', null,
    people.map(person) => (React.createElement('li', {key:
person.name}, person.name))))
```

::Remember that if you are looping through array and are creating an element for each item in the array, you will need a unique key prop.::

1.2 JSX

JSX is a syntax extension to JavaScript that lets us write JS code that looks little bit more like HTML.

```
const element = <ol><li>{people[0].name}</li></ol>
```

With map function:

```
const element = <ol>
    {people.map((person) => (
      <li key={person.name}>{person.name}</li>
    ))}
</ol>
```

```
* Make sure you don't forget unique key props.
* In the page source, JSX is still translated to createElement
function.
* ::JXS must only return a single element.:: This element may have
many dependents though.
```

2. Intro to Components

`createElement()` and `JSX` can help us produce HTML. But Components in React can help us construct UI.

::> Components refers to reusable pieces of code ultimately responsible for returning HTML to be rendered onto pages.::

```
* Think Components as the factory generating custom elements.
* The only method requires in Component class is `render` And it's to
return the GSX or the elements that this component renders.
* By grouping together all elements in a single Component, we can
treat them as a single element.
* Components represents **Modularity** and **Reusability**
```

Below example is still to render/display a list of items:

```
class ContactList extends React.Component {
  render() {
    const people = [
      {name: 'Tyler'}, {name: 'Karen'}, {name: 'Richard'}
    ]

    return <ol> {people.map((person) => (<li key={person.name}>
{person.name}</li>))}</ol>
  }
}

//render ContactList rather than element now

ReactDOM.render(
  <ContactList />
```

```
  document.getElementById('root')
)
```

Declaring in another format:

```
import React, { Component } from 'react'

class ContactList extends Component {}
```

2. Create React APP

JSX is awesome, but it does need to be transpiled into regular JavaScript before reaching the browser. We typically use a transpiler like Babel to accomplish this for us. We can run Babel through a build tool, like Webpack which helps bundle all of our assets (JavaScript files, CSS, images, etc.) for web projects.
To streamline these initial configurations, we can use Facebook's Create React App package to manage all the setup for us! This tool is incredibly helpful to get started in building a React app, as it sets up everything we need with *zero configuration*!

[image:919CC3C3-5C6A-4115-89C8-35901E5F1DD9-526-00006690455F6E63/Screen Shot 2020-01-10 at 9.12.12 PM.png]

```
npm install -g create-react-app
```

`-g` makes sure we install it globally. `npm list -g` can find out where global packages are installed.

Next, run command `create-react-app contacts` to install `react`, `react-dom` and `react-scripts`

`react-scripts` installs:

```
1. Babel: we can use JSX and latest JS
2. Webpack: we can generate the build
3. Webpack dev server: we have auto-reload behaviour
```

Next, cd to the directory and run `yarn start`

```
*  Yarn is a package manager that's similar to NPM. Yarn was created from the ground up by Facebook to improve on some key aspects that are slow or lacking in NPM.* [Yarn](https://yarnpkg.com/)
* For most command with yarn, you can swap yarn with npm.
```

3. Composing Components

Components benefit from two points:

```
1. Each component is independent. No effect on others when developing one.
2. Composition rather than inheritance: ::Instead of extending base components to add more UI or behavior, we compose elements in different ways using nesting and props.:: You ultimately want your UI components to be independent, focused, and *reusable*.
```

Example: reuse components with different prop

```
import React, { Component } from 'react';


class ContactList extends Component {
  render() {
    const people = this.props.contacts
    return <ol>
    {people.map((person) => (<li key={person.name}>{person.name}
</li>))}
    </ol>

  }
}

class App extends Component {
  render() {
    return (
      <div className="App">
        <ContactList contacts={[{name: 'tyler'}, {name: 'karen'}]}/>
        <ContactList contacts={[{name: 'gabriel'}, {name: 'gu'}]}/>
      </div>
```

```
    )
  };
}

export default App;
```

^ note the line: `const people = this.props.contacts` and `<ContactList contacts={[{name: 'tyl'}, {name: 'kar'}]}/>`

Lesson3 - Managing States

Three concepts of React:
* Props: allow you t o pass data into your components
* Functional Components: an alternative, and probably more intuitive approach to creating components
* Controlled Components: allow you to hook up the forms in your application to your component state

1. Stateless Functional Components

You no longer have access to `this.props`, so make it an function argument.

[image:64E5D2F4-89ED-4C0C-BB4D-5D061364009B-526-00009FEF1C711146/Screen Shot 2020-01-13 at 5.58.51 PM.png]

With above code, you don't need `Component` but you still need `export default ListContacts`

::**If your component does not keep track of internal state (i.e., all it really has is just a render() method), you can declare the component as a Stateless Functional Component.**::

2. Add State To A Component

```
* `props` represents `read-only` data that are immutable.
* `state`represents `mutable` data that ultimately affects what is rendered on the page.
```

State is managed internally by the component itself and is meant to change over time, commonly due to user input (e.g., clicking on a button on the page).

This is one of the key benefits of using React to build UI components: when it comes to re-rendering the page, we just have to think about updating state. We don't have to keep track of exactly which parts of the page change each time there are updates. We don't need to decide how we will efficiently re-render the page. React compares the previous output and new output, determines what has changed, and makes these decisions for us. This process of determining what has changed in the previous and new outputs is called ::Reconciliation::.

```
class User extends React.Component {
 state = {
   username: 'Tyler'
 }
}
```

```
class User extends React.Component {
 constructor(props) {
   super(props);
   this.state = {
     username: 'Tyler'
   };
 }
}
```

Don't write `props` in `state` field, like

```
this.state = {
 user: props.user
}
```

> In the above example, if props are ever updated, the current state will not change unless the component is "refreshed." Using props to produce a component's initial state also leads to duplication of data, deviating from a dependable "source of truth."
> [image:7A146E2B-6EA7-4460-B322-CE0975D54935-526-

> 0000A1977E04E198/Screen Shot 2020-01-13 at 6.29.15 PM.png]

2. Change State

You cannot mutate state variable directly as React won't be able to track what state is changed.
Thus, we use `setState()` function in two ways:

```
1. Pass in a function:
  * function will takes in the previous state
  * function produces new state that will be merged with current state
to form new state of component
```

```
this.setState((prevState) => ({count: prevState.count + 1}))
```

```
2. Pass in an object
  * the object will merge with current state to form the new state of
the component
```

```
this.setState({username: 'Tyler'})
```

> Choose pass in a function when new state depends on the previous one

Exercise Code:

Add a function that updates state by remove target row of data to display
[image:9020E6D7-D10E-4DF8-A0BC-A14D2297E2FE-435-000033CBB54C1249/Screen Shot 2020-01-16 at 12.20.58 AM.png]

In the app component, when pass to ListContacts component, pass the function called onDeleteContact
[image:3653CE2D-0CD8-4B2B-BBD1-6B741075D45D-435-000033D660071207/Screen Shot 2020-01-16 at 12.22.15 AM.png]

When click on button, trigger removeContact function by giving the target row of data as argument
[image:B000CA87-9A12-48E5-8DA7-04133E1DF640-435-000033E4FA883C68/Screen Shot 2020-01-16 at 12.22.51 AM.png]

2. PropTypes

> PropTypes is a package that lets us define the data type we want to see right from the get-go and warn us during development if the prop that's passed to the component doesn't match what is expected.

to use it, we install by `npm install –save prop-types` or `yarn add prop-types`

Now in the target component file: ListContact function, add following codes:
[image:38DEF5CB-9785-47FF-BE99-E11D0E97B400-435-0000347299B5D0F1/Screen Shot 2020-01-16 at 12.35.53 AM.png]
[image:08E31866-7ECA-46A5-A822-A671ADC4EA70-435-00003484792168C0/Screen Shot 2020-01-16 at 12.36.48 AM.png]
Below code is same thing but add PropTypes to components:
[image:F3DD73FF-CDEF-4FC8-BA37-4413B2741C38-435-0000368AAFB933FD/Screen Shot 2020-01-16 at 1.13.59 AM.png]

3. Handle forms in React: Controlled Component

> Controlled Component renders a form but the source of truth for the forms state lives inside of your component state rather than inside of the DOM

> React controls the state of form

`event.target` returns a reference to the object on which the event originally occurred.
`event.target.value` is the value property of some DOM elements such as text entered in the search input.

[image:AC321FA6-C8E7-407C-9C62-56FFD995990B-435-000037048054869E/Screen Shot 2020-01-16 at 1.22.55 AM.png]

Analysis: for `line9` we see the value of search bar is just whatever the state's email is. And it has dynamic method on change that changes the state value when user input texts.

Good Example:

[image:F5F2436F-DCA2-468C-8C4F-3AC935F9D45A-435-000037357D4BA616/Screen Shot 2020-01-16 at 1.25.39 AM.png]
[image:44D78889-26B4-414B-9801-361122ABA54A-435-00003736E9B5D53F/Screen Shot 2020-01-16 at 1.26.27 AM.png]
[image:CC627EFE-C8C4-406A-9172-69D75DC03EDC-435-0000385417388532/Screen Shot 2020-01-16 at 1.46.55 AM.png]

Next, based on the input text, lets see how we can use this state to filter information

```
1. Install following package
*  [escape-string-regexp](https://www.npmjs.com/package/escape-string-regexp)
*  [sort-by](https://www.npmjs.com/package/sort-by)
```

`npm install –save escape-string-regexp sort-by`

[image:294FA3D0-758D-4F14-8136-DA12737B4686-435-00006F3E08F54D3C/Screen Shot 2020-01-18 at 4.35.21 PM.png]
Next, show the display count:
[image:6607340C-267E-4CDA-8DC6-000F3338D311-435-00006F3830ACF443/BB00C6B9-9BC4-4594-9D5C-187480BB1611.png]

Lesson4 - Lifecycle Events

> Data should not be fetched in the `render` method. `render` method should only be used to render that component rather than making HTTP, API requests or fetch data that's used to display the content or alter the DOM. `render` function should not call any other function that does that. We put them in **lifecycle events.**
> [image:DBDEB33E-8F6D-44BE-9654-8F5DA3F8FBE9-435-000075F7A13358C5/Screen Shot 2020-01-18 at 6.38.35 PM.png]

Lifecycle Events

Lifecycle events are specially named methods in a component. These methods are automatically bound to the component instance, and React will call these methods naturally at certain times during the life of a component. There are a number of different lifecycle events, but here are the most commonly used ones.

[image:82654AC2-1D7D-4021-AF30-D3B8AD68B6A2-435-000070D17B82CFB7/Screen Shot 2020-01-18 at 5.04.13 PM.png]

::ComponentDidMount: calls after component is mounted(which means after it is rendered), it is popularly used for api fetch data or ajax request::

Adding to the DOM

The following lifecycle events will be called in order when a component is being added to the DOM:
1 constructor()
2 getDerivedStateFromProps()
3 render()
4 componentDidMount()

Re-rendering

The following lifecycle events will be called in order when a component is re-rendered to the DOM:
1 getDerivedStateFromProps()
2 shouldComponentUpdate()
3 render()
4 getSnapshotBeforeUpdate()(specific use cases)
5 componentDidUpdate()

Removing from the DOM

This lifecycle event is called when a component is being removed from the DOM:

- componentWillUnmount()
    - You'll sometimes see `shouldComponentUpdate()` in React apps as well. It returns true by default. This means that whenever a component's state

(or its parent's state) is updated, the component re-renders.

Example:

```
import React, { Component } from 'react';
import fetchUser from '../utils/UserAPI';

class User extends Component {
 constructor(props) {
   super(props);

   this.state = {
     name: '',
     age: ''
   };
 }

 componentDidMount() {
   fetchUser().then((user) => this.setState({
     name: user.name,
     age: user.age
   }));
 }

 render() {
   return (
     <div>
       <p>Name: {this.state.name}</p>
       <p>Age: {this.state.age}</p>
     </div>
   );
   }
}

export default User;
```

You'll notice that this component has a componentDidMount() lifecycle event. This component seems pretty straightforward, but let's walk through the order of how it works:

1. The render() method is called which updates the page with a

that has one paragraph for the name and one paragraph for the age. What's important to realize is that this.state.name and this.state.age are empty strings (at first), ::so the name and age *don't actually display*:: 2. Once the component has been mounted, the componentDidMount() lifecycle event occurs * The fetchUser request from the UserAPI is run which sends a request to the user database * When the data is returned, setState() is called and updates the name and age properties 3. Since the state has changed, render() gets called again. This re-renders the page, but now this.state.name and this.state.age have values

```
* Demo in production: fetch information through API
(note `ContactsAPI.getAll()` is utility helper function)
    [image:C2C39A7A-EFDF-4B8D-94F4-EF1A8C3263FB-410-
000000D051312F5C/Screen Shot 2020-01-29 at 1.49.25 PM.png]

* Demo to remove a contact by adding a function
     [image:EED458AA-FC45-4D4F-9A4E-EFE8FCCA16F7-410-
000000CCF0CA7065/Screen Shot 2020-01-29 at 1.49.11 PM.png]
The last line `ContactAPI.remove()` function makes us sync with our
server which means the db actually removes a contact.
```

Routing in React: build React as Single-Page Application

- **Single Page App**: When user browses different pages, the browser does not need to go back to server to retrieve the page content.
- **Single Page App Dev Plan:**
    - pre-downloading the entire site contents all at once, so no need to refresh the page
    - downloading everything that's needed to render the page the user requested. Then when the user navigates to a new page, asynchronous

JavaScript requests are made for *just* the content that was requested.

- **Pros of Single Page App:**

Another key factor in a good single-page app is that the URL controls the page content. Single-page applications are highly interactive, and users want to be able to get back to a certain state using just the URL. Why is this important? Bookmarkability!

React Router:

> React Router is a collection of **navigational components** that compose declaratively with your application.
>
> Below is an easy way to do conditional rendering based on a state called **short-circuit evaluation**

```
{this.state.screen === 'list' && (
  <ListContacts
  contacts={this.state.contacts}
  onDeleteContact={this.removeContact}
  />
)};

{this.state.screen === 'create' && (
  <CreateContact />
)}
```

Demo： in App.js, pass new component a function onNavigate. In
[image:B86B5BE1-F7CC-4711-96E7-DE9626C09FA0-410-0000033FC2AFDB5D/Screen Shot 2020-01-29 at 2.34.02 PM.png]
In ListContact.js, add a clickable text, once clicked, trigger onNavigate function which is passed as a props. [we add this text on the search bar as routing to a new component render function]
[image:C7839666-07E5-44C5-82F2-2DE4E271D907-410-00000343B05D287A/Screen Shot 2020-01-29 at 2.34.21 PM.png]
::However, the back button does not work for now.:: Using state to control content display is not enough.

::We need router to keep our UI and URL in Sync::

How to use React Router

```
npm install –save react-router-dom
```

:::: Listen for changes in URL and make sure the correct screen shows up

To use it, change in the index.js file. Wrap main component inside the BrowserRouter.

```
import { BrowserRouter } from 'react-router-dom'

ReactDOM.render(
  <BrowserRouter>
    <App />
  </BrowserRouter>, document.getElementById('root'));
registerServiceWorker();
```

Let's look into the source code of React Router for better illustration:

```
class BrowserRouter extends React.Component {
  static propTypes = {
    basename: PropTypes.string,
    forceRefresh: PropTypes.bool,
    getUserConfirmation: PropTypes.func,
    keyLength: PropTypes.number,
    children: PropTypes.node
  }

  history = createHistory(this.props)

  render() {
    return <Router history={this.history} children=
{this.props.children} />
  }
}
```

> When you use BrowserRouter, what you're really doing is rendering a Router component and passing it a history prop. Wait, what is history? history comes from the history library (also built by React Training). The whole purpose of this library is it abstracts away the differences in various environments and provides a minimal API that lets you manage the history stack, navigate, confirm navigation, and persist state between sessions.

> So in a nutshell, when you use BrowserRouter, you're ::creating a history object which will listen to changes in the URL:: and make sure your app is made aware of those changes.

The link Component

link component is the way that user navigates through the App. When clicking a link, it talks to browserRouter to update the URL.

to use it first import it: `import { BrowserRouter } from 'react-router-dom'`

Change the below code:
[image:6E1DA11B-31F5-46EA-A4F9-D107C26B08EC-410-000004B22F936680/Screen Shot 2020-01-29 at 3.00.33 PM.png]
to:
[image:07FCB029-E69B-4CFE-A95A-851A23629867-410-000004BA89651BDB/Screen Shot 2020-01-29 at 3.01.11 PM.png]

We do not need herf but just need to add affix for url and we also don't need onClick function as React Router later can handle it.

When clicking on the button now, the URL will have `/create` being appended.

::As you've seen, Link is a straightforward way to provide declarative, accessible navigation around your application. By passing a to property to the Link component, you tell your app which path to route to.::

`<Link to="/about">About</Link>`

If you're experienced with routing on the web, you'll know that sometimes our links need to be a little more complex than just a string. For example, you can pass along query parameters or link to specific parts of a page. What if you wanted to pass state to the new route? ::To account for these scenarios, instead of passing a string to Links to prop, you can pass it an object like this,::

```
<Link to={{
 pathname: '/courses',
 search: '?sort=name',
 hash: '#the-hash',
 state: { fromDashboard: true }
}}>
 Courses
</Link>
```

An object that can have any of the following properties:
* pathname: A string representing the path to link to.
* search: A string representation of query parameters.
* hash: A hash to put in the URL, e.g. #a-hash.
* state: State to persist to the location.

Last thing we need is `<Route path="/create" />` It takes a path that may match the URL, if so it will render the UI. Since now we are checking URL instead of state, the browser back button will still work.

In the App.js:

```
1. We need to import it `import { Route } from 'react-router-dom'`
```

[image:2C5D7035-6110-4308-95C2-1EB69A550ECD-410-0000056BF4EC3690/Screen Shot 2020-01-29 at 3.13.52 PM.png]

We wrap two components in two Route attribute.

::`<Route exact` is important as otherwise, `/create` is actually matching both routes.::

The first Route uses `render` that takes in a function because we also need to pass some props. While the second one uses `component` that takes a component solely as no props needed to be passed.

Finalize the Create Contact Demo Page:

In CreateContact.js

1. add input box and

   [image:B2B5A30E-C1E0-49C3-A675-276C92330BAA-410-000005C890D7C28E/Screen Shot 2020-01-29 at 3.20.31 PM.png]
   https://github.com/udacity/reactnd-contacts-app/commit/f876f2d17b338e57ec80e8f67abbb3efa83bff2a

2. inside `form` add onSubmit function. So that we can actually submit those input inside

   to state and server
   [image:EC18CB3B-09E9-441D-B257-51A35514BCF2-410-00000651351A9369/Screen Shot 2020-01-29 at 3.30.15 PM.png]

3. Create handleSubmit function.
   - e.preventDefault() is to prevent URL to add those parameters.

   > We call this behaviour as `serialize the form data`. On default, our form will serialize the values from user input and add them as query string to our URL.
   > We would like to add additional functionality by having our app serialize those form fields on its own, To do so, we install the form-serialize package `npm install –save form-serialize` and then import it `import serializeForm from 'form-serialize'`

- ::then we use the function `serializeForm` function. passes it the form by `event.target` and then set the `hash` as `true` in order to get back the serialized object.::
- OnCreateContact will be a function written in `app.js` and passed to this file. It will actually update the state and server.

[image:2660ADA0-6896-42B1-A44C-2E0DF6C67264-410-0000068F19FCD7E8/Screen Shot 2020-01-29 at 3.34.43 PM.png]

In App.js

1. Create the function and call ContactAPI to update server and then set the state based on previous state.
   [image:F61D6564-9D5A-41BE-B796-4BB74F4CC1E0-410-000006B7912C715B/Screen Shot 2020-01-29 at 3.37.37 PM.png]
2. We need to pass the onCreateContact function to component CreateContact.

It is important to notice that instead of writing `onCreateContact={this.createContact(contact)}`, we explicitly write a function call and also add something as `history.push('/')` and explicitly asking to take in a parameter `history`. ::Those are added so that when the user called the `onCreateContact` function, the page will automatically go back to home page `/`::
[image:D43014BE-3CDF-4C80-A5B3-A6E5692ADB80-410-000006C51B9F1B43/Screen Shot 2020-01-29 at 3.38.34 PM.png]

React Routing Official Document: [React Router: Declarative Routing for React.js]