

React js - library for building user interfaces

- Course will work with any version of React (16.8)
- Basics of JS reference



Basics of JavaScript

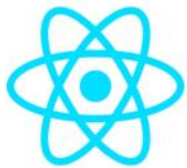
- Variables and types
- Objects and arrays
- Functions and classes
- Loops and conditionals

Learning JS:

- Book: jscomplete.com/beginning-js
- Labs: jscomplete.com/js-labs

- React js commonly faced problems reference <https://jscomplete.com/react-cfp>
 - Library vs Frameworks
 - Frameworks - Good for young teams and startup and most of the smart design decisions are already made and dev can focus on writing good app-level logic.
 - o Disadvantages -
 - Limited flexibility - We can do things only in a certain way defined by the framework
 - Limited flexibility - Hard to deviate
 - Large and full of features - Hard to customize for specialized cases
 - Large and full of features - Even if we require a small piece of the framework we need to use the whole thing.

Why React?



The "virtual" browser (vs. DOM API)

"Just JavaScript"

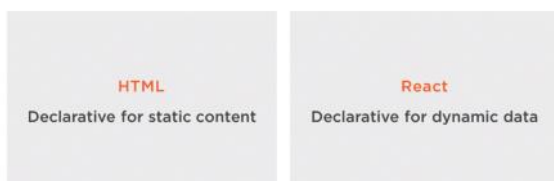
React Native (for the win)

Battle-tested

Declarative language (model UI and state)

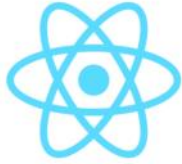
- React is declarative (It helps dev to declaratively describe their UI's and model the state of these UI's)
- Developers declaratively describe stateful user interfaces (This means instead of coming up with steps for the transactions on their interfaces, dev just describes the interfaces in terms of final state, like a function. When transaction happens to that declared state, React takes care of updating the user interfaces based on that.)

But Isn't HTML Already Declarative?



- React gives dev the ability to work with a virtual browser that is better than working with real browser.
- The performance of virtual DOM in React is an advantage.
- More of javascript and less of React API's
- React Native can be used to write native code for iOS / Android and code can also be shared between platforms.
- Battle tested on FB, so less of major production bugs.

React's Basic Concepts



- 1) Components
 - Like functions
 - Input: props, state | Output: UI
 - Reusable and composable
 - `<Component />`
 - Can manage a private state
- 2) Reactive updates
 - React will react
 - Take updates to the browser
- 3) Virtual views in memory
 - Generate HTML using JavaScript
 - No HTML template language
 - Tree reconciliation

- Components (UI interfaces are described using components)
 - o Simple react components are just vanilla js functions.
 - o Components receives Input in form of props, state and delivers the UI as output
 - o Components can be reused in multiple user interfaces and components can contain other components.
 - o We do not invoke the component but use it as a regular html element (`<Component />`)
 - o A react component can have a private state to hold any data that may change over the lifecycle of the component.
- Reactive updates
 - o When state of react component, the input changes then the output changes as well. This change in the description of the UI has to be reflected in the device we are working with.
 - o In browser we need to regenerate the HTML views in the DOM tree. With React we need not worry about how to reflect these changes or even manage when to take the changes to browser. React will automatically update the parts of the DOM that needs to be updated.
- Virtual views in memory
 - o Generates HTML using js
 - o No HTML template language
 - o Tree reconciliation (React uses the virtual DOM to compare versions of the UI in memory before it acts on them)
 - o The above works on the concept called **Reconciliation**

Reconciliation

- o React provides a declarative API so that you don't have to worry about exactly what changes on every update. This makes writing applications a lot easier, but it might not be obvious how this is implemented within React. This article explains the choices we made in React's "diffing" algorithm so that component updates are predictable while being fast enough for high-performance apps.

When you use React, at a single point in time you can think of the `render()` function as creating a tree of React elements. On the next state or props update, that `render()` function will return a different tree of React elements. React then needs to figure out how to efficiently update the UI to match the most recent tree.

- o There are some generic solutions to this algorithmic problem of generating the minimum number of operations to transform one tree into another. However, the state of the art algorithms have a complexity in the order of $O(n^3)$ where n is the number of elements in the tree.

If we used this in React, displaying 1000 elements would require in the order of one billion comparisons. This is far too expensive. Instead, React implements a heuristic $O(n)$ algorithm based on two assumptions:

1. Two elements of different types will produce different trees.
2. The developer can hint at which child elements may be stable across different renders with a `key` prop.

The Diffing Algorithm

When diffing two trees, React first compares the two root elements. The behavior is different depending on the types of the root elements.

Elements Of Different Types

- Whenever the root elements have different types, React will tear down the old tree and build the new tree from scratch. Going from `<a>` to ``, or from `<Article>` to `<Comment>`, or from `<Button>` to `<div>` - any of those will lead to a full rebuild.

When tearing down a tree, old DOM nodes are destroyed. Component instances receive `componentWillUnmount()`. When building up a new tree, new DOM nodes are inserted into the DOM. Component instances receive `componentWillMount()` and then `componentDidMount()`. Any state associated with the old tree is lost.

Any components below the root will also get unmounted and have their state destroyed. For example, when diffing:

```
<div>
  <Counter />
</div>
```

- ```

 <Counter />

```

This will destroy the old `Counter` and remount a new one.

## DOM Elements Of The Same Type

When comparing two React DOM elements of the same type, React looks at the attributes of both, keeps the same underlying DOM node, and only updates the changed attributes. For example:

```
<div className="before" title="stuff" />

<div className="after" title="stuff" />
```

- By comparing these two elements, React knows to only modify the `className` on the underlying DOM node.

When updating `style`, React also knows to update only the properties that changed. For example:

```
<div style={{color: 'red', fontWeight: 'bold'}} />

<div style={{color: 'green', fontWeight: 'bold'}} />
```

When converting between these two elements, React knows to only modify the `color` style, not the `fontWeight`.

After handling the DOM node, React then recurses on the children.

## Component Elements Of The Same Type

- When a component updates, the instance stays the same, so that state is maintained across renders. React updates the props of the underlying component instance to match the new element, and calls `componentWillReceiveProps()` and `componentWillUpdate()` on the underlying instance.

Next, the `render()` method is called and the diff algorithm recurses on the previous result and the new result.

Next, the `render()` method is called and the diff algorithm recurses on the previous result and the new result.

## Recurring On Children

By default, when recursing on the children of a DOM node, React just iterates over both lists of children at the same time and generates a mutation whenever there's a difference.

For example, when adding an element at the end of the children, converting between these two trees works well:

```
○
 first
 second

 first
 second
 third

```

React will match the two `<li>first</li>` trees, match the two `<li>second</li>` trees, and then insert the `<li>third</li>` tree.

If you implement it naively, inserting an element at the beginning has worse performance. For example, converting between these two trees works poorly:

```
○
 Duke
 Villanova

 Connecticut
 Duke
 Villanova

```

○ React will mutate every child instead of realizing it can keep the `<li>Duke</li>` and `<li>Villanova</li>` subtrees intact. This inefficiency can be a problem.

## Keys

In order to solve this issue, React supports a `key` attribute. When children have keys, React uses the key to match children in the original tree with children in the subsequent tree. For example, adding a `key` to our inefficient example above can make the tree conversion efficient:

```
○
 <li key="2015">Duke
 <li key="2016">Villanova

 <li key="2014">Connecticut
 <li key="2015">Duke
 <li key="2016">Villanova

```

Now React knows that the element with key '2014' is the new one, and the elements with the keys '2015' and '2016' have just moved.

In practice, finding a key is usually not hard. The element you are going to display may already have a unique ID, so the key can just come from your data:

```
<li key={item.id}>{item.name}
```

○

When that's not the case, you can add a new ID property to your model or hash some parts of the content to generate a key. The key only has to be unique among its siblings, not globally unique.

As a last resort, you can pass an item's index in the array as a key. This can work well if the items are never reordered, but reorders will be slow.

Reorders can also cause issues with component state when indexes are used as keys.

- Component instances are updated and reused based on their key. If the key is an index, moving an item changes it. As a result, component state for things like uncontrolled inputs can get mixed up and updated in unexpected ways.

## Tradeoffs

It is important to remember that the reconciliation algorithm is an implementation detail. React could rerender the whole app on every action; the end result would be the same. Just to be clear, rerender in this context means calling `render` for all components, it doesn't mean React will unmount and remount them. It will only apply the differences following the rules stated in the previous sections.

○

We are regularly refining the heuristics in order to make common use cases faster. In the current implementation, you can express the fact that a subtree has been moved amongst its siblings, but you cannot tell that it has moved somewhere else. The algorithm will rerender that full subtree.

Because React relies on heuristics, if the assumptions behind them are not met, performance will suffer.

- 
1. The algorithm will not try to match subtrees of different component types. If you see yourself alternating between two component types with very similar output, you may want to make it the same type. In practice, we haven't found this to be an issue.
  2. Keys should be stable, predictable, and unique. Unstable keys (like those produced by `Math.random()`) will cause many component instances and DOM nodes to be unnecessarily recreated, which can cause performance degradation and lost state in child components.

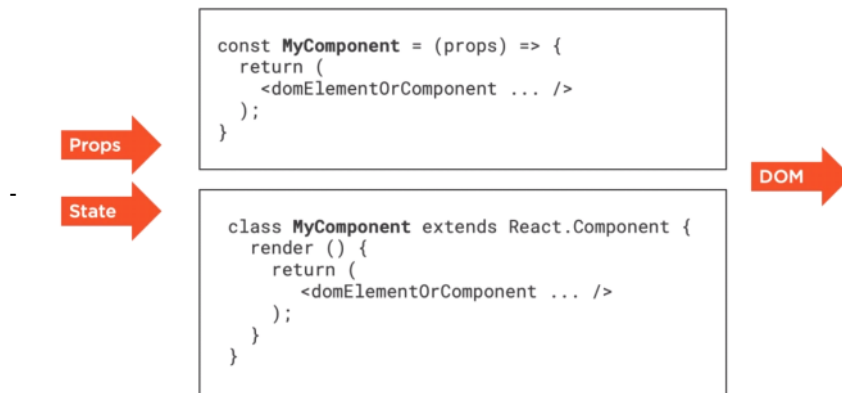
## React Components

### React Components

-



- Function Components are simpler when compared to class component but class components are powerful than the functional component.



- Both components use Input: Props and State, Output: DOM
- Props input in an explicit one which is equal to the list of attributes an HTML element can have.
- State input is an internal one which is used to auto reflect the changes in the browser.
- State object can be changed but the props object value is fixed.
- Props are immutable.
- Components can only change their internal state, not their properties. (Core concept of React)

## JSX Is NOT HTML

```

class Hello extends React.Component {
 render () {
 return (
 <div className="container">
 <h1>Getting Started</h1>
 </div>
);
 }
}

ReactDOM.render(<Hello />, mountNode);

```

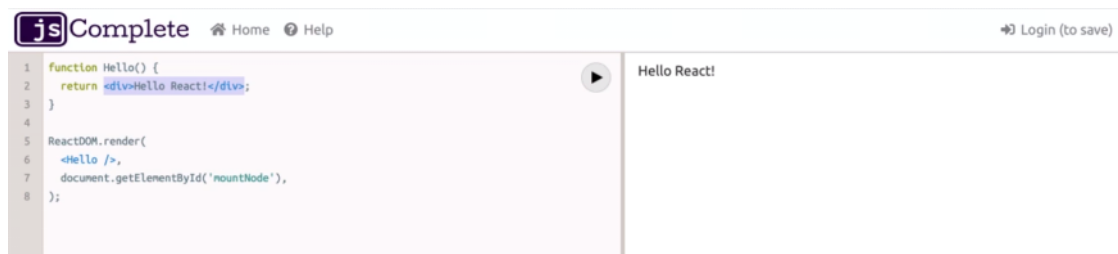
```

class Hello extends React.Component {
 render () {
 return (
 React.createElement("div", { className: "container"},
 React.createElement("h1", null, "Getting Started")
)
);
 }
}

ReactDOM.render(React.createElement(Hello, null), mountNode);

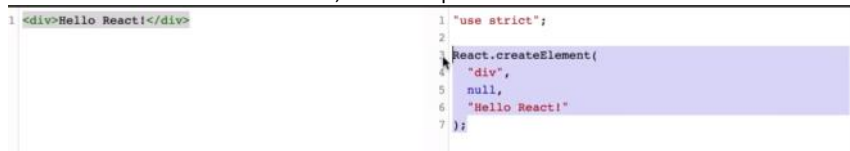
```

- Working with React Component



The above code is in JSX and will be compiled by Babel to convert JSX into React API calls. So the browser is not really executing the above piece of code.

In Babel if the above code is written, Babel compiles this into a call to the createElement function in the React top-level API as shown below.



Browser executes the following  
Without using JSX it can be written as below  
React.createElement will take a list of arguments

- first argument is on the element to be created (In this ex. It is a div)
- second argument is about the attributes the element needs to have (In this ex. No attributes)
- third argument is the children that the element needs to have(In this ex. It is a string "Hello React!")

```

1 function Hello() {
2 // return <div>Hello React!</div>;
3 return React.createElement('div', null, 'Hello React!!!!');
4 }
5
6 ReactDOM.render(
7 <Hello />,
8 document.getElementById('mountNode'),
9);

```

Hello React!!!!

Even <Hello /> can be written as above using React.createElement without JSX

```

6 ReactDOM.render(
7 // <Hello />,
8 React.createElement(Hello, null),
9 document.getElementById('mountNode'),
10);

```

Note: It is better to follow the standard to capitalize the function name

See next module for a detailed explanation of  
Array Destructuring

Explanation of Array Destructuring:

useState() returns array of two elements and they are destructuring the array set to the const variables.

onClick={functionRef}

onClick -> case sensitive and it should be camel case. The function reference(name) needs to be mentioned in curly braces {}.

{ } -> Expressions can be written inside the braces. Ex { () => setCounter(counter+1) }

() => -> Arrow function, which has the body of the function after the parenthesis and the arrow



```
1 function Button() {
2 const [counter, setCounter] = useState(0);
3 return <button onClick={() => setCounter(counter+1)}>{counter}</button>;
4 }
5
6 ReactDOM.render(
7 <Button />,
8 document.getElementById('mountNode'),
9);
```

useState(0) -> This useState function is called as a "Hook" in React. It is similar to a mixin or a module, but it is a stateful one that hooks this component into a state.

## React Hook

useState() results:

- a) state object (getter)
- b) updater function (setter)

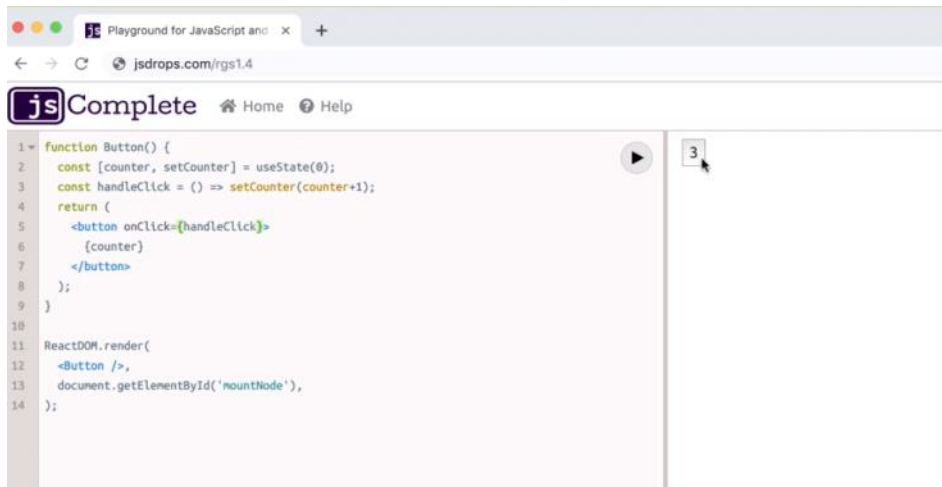
- useState returns an array with two elements as mentioned above.

```
const [currentStateValue, functionToSetNewStateValue] = useState(initialStateValue)
```

```
1 function Button() {
2 const [counter, setCounter] = useState(0);
3 return {
4 <button onClick={() => setCounter(counter+1)}>
5 {counter}
6 </button>
7 };
8 }
9
10 ReactDOM.render(
11 <Button />,
12 document.getElementById('mountNode'),
13);
```

- The above returns a function call and not an object. React.createElement returns a function call.





- We cannot concatenate two different react elements to be displayed as shown below

```
ReactDOM.render(
 <Button /><Display />,
 document.getElementById('mountNode'),
);
```

- The above does not work because each react elements will be translated into a function call.
- To make it work we can render the react elements as an array as shown below

```
ReactDOM.render(
 [<Button />, <Display />],
 document.getElementById('mountNode'),
);
```

- The above works and is a good solution when we have all the elements being rendered by the same component in a dynamic way.
- Other option is to wrap the elements inside a <div> as shown below

```
ReactDOM.render(
 <div>
 <Button />
 <Display />
 </div>,
 document.getElementById('mountNode'),
);
```

- If we don't want the react elements to be wrapped in a single div then we can use React.Fragment

```
ReactDOM.render(
 <React.Fragment>
 <Button />
 <Display />
 </React.Fragment>,
 document.getElementById('mountNode'),
);
```

- The above does the same as the div apart from adding an new DOM parent.
- We can also mention an empty tag as below, but this will be compiles as a React.Fragment if it is supported.

```
ReactDOM.render(
 <>
 <Button />
 <Display />
 </>,
 document.getElementById('mountNode'),
);
```

```

1 function Button(props) {
2 // const handleClick = () => setCounter(counter+1);
3 return (
4 <button onClick={props.onClickFunction}>
5 +1
6 </button>
7);
8 }
9
10 function Display(props) {
11 return (
12 <div>{props.message}</div>
13);
14 }
15
16 function App() {
17 const [counter, setCounter] = useState(42);
18 const incrementCounter = () => setCounter(counter+1);
19 return (
20 <div>
21 <Button onClickFunction={incrementCounter}>/>
22 <Display message={counter}>/>
23 </div>
24);
25 }
26

```

- Parent components can flow their data down to the children components using props. Props can hold functions as well. Check above code
- Parent components can flow their behavior to their children components.
- When we want to pass an number through props it should be mentioned in the attribute as in the below example
  - o `<Button increment={5} />` - This is to send a numeric value, when we specify in "" it will send it as a string

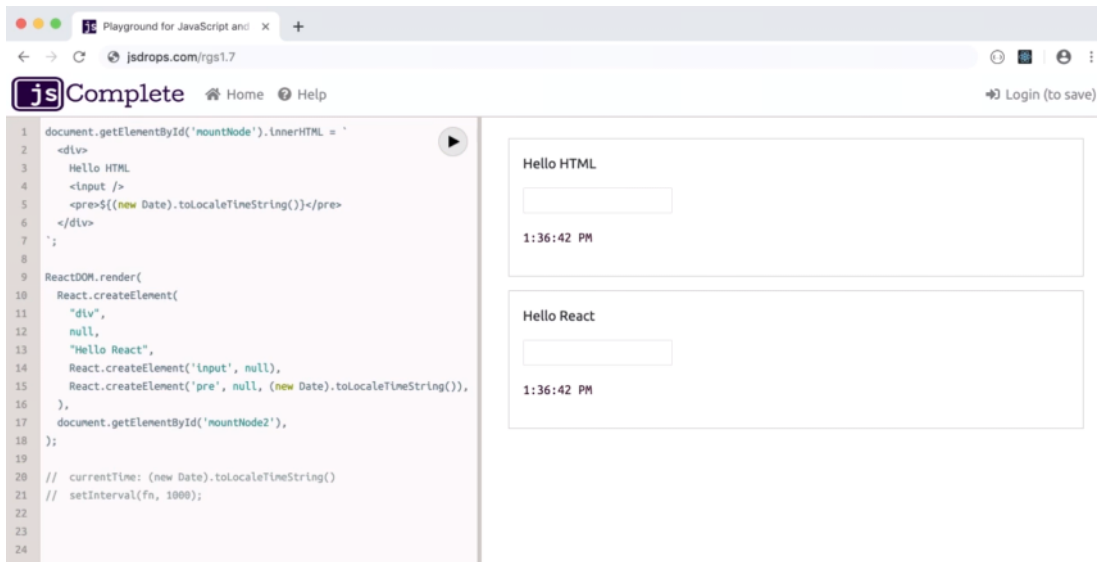
```

1 function Button(props) {
2 const handleClick = () => props.onClickFunction(props.increment);
3 return (
4 <button onClick={handleClick}>
5 {props.increment}
6 </button>
7);
8 }
9
10 function Display(props) {
11 return (
12 <div>{props.message}</div>
13);
14 }
15
16 function App() {
17 const [counter, setCounter] = useState(0);
18 const incrementCounter = (incrementValue) => setCounter(counter+incrementValue);
19 return (
20 <div>
21 <Button onClickFunction={incrementCounter} increment={1} />
22 <Button onClickFunction={incrementCounter} increment={5} />
23 <Button onClickFunction={incrementCounter} increment={10} />
24 <Button onClickFunction={incrementCounter} increment={100} />
25 <Display message={counter}>/>
26 </div>
27);
28 }
29

```

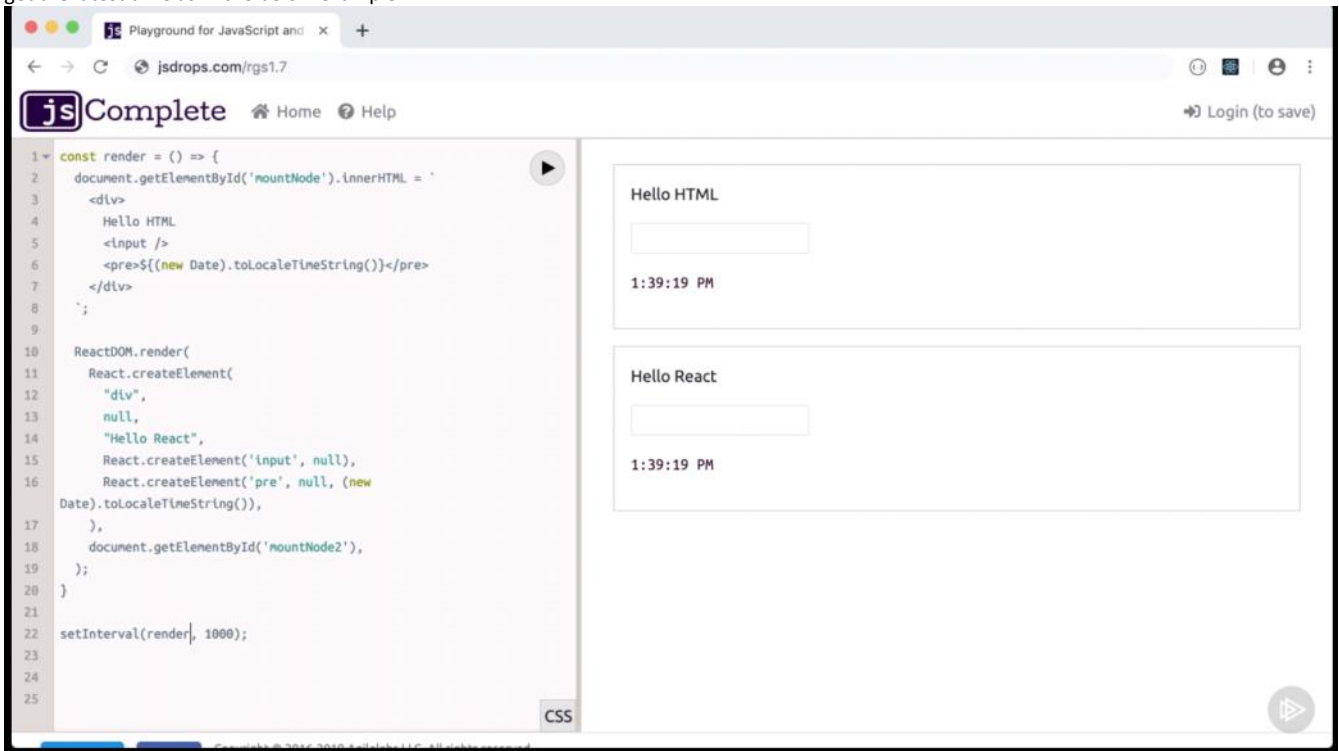
### Tree Reconciliation

- In the below example it shows the difference between using html template to generate an HTML and using JSX to generate HTML

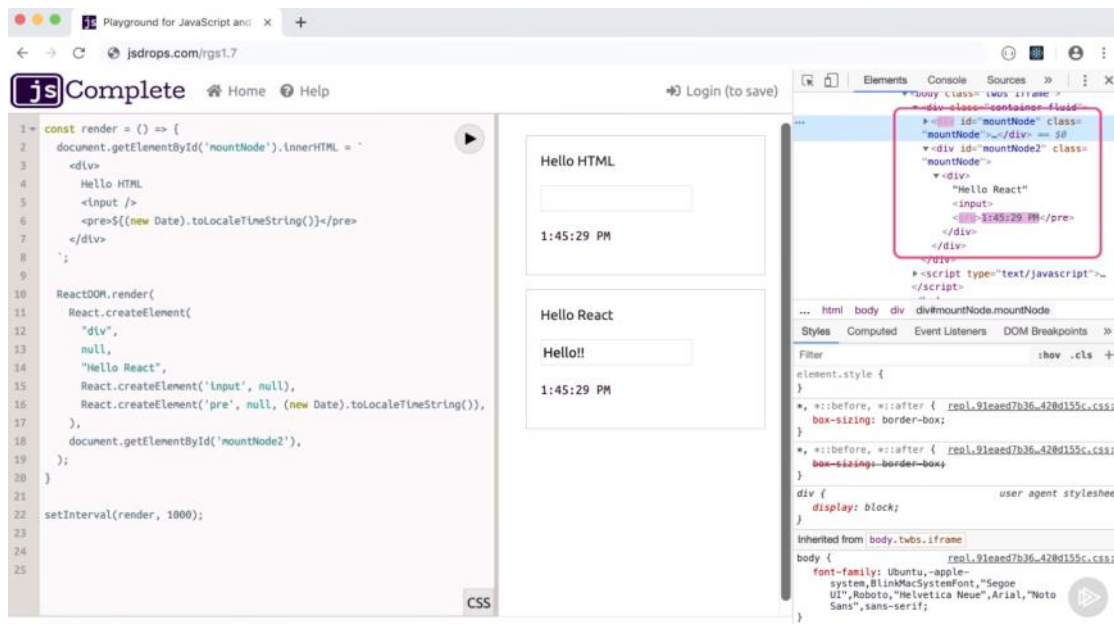


- Advantages of using React over normal HTML

- The advantage is not in rendering the first HTML view but it is about what we need to do to update any existing views in DOM.
- This can be seen in the below example where we try to update the DOM, in this example we try to use a setInterval web timer API to update the time.
- To use the web timer API we move the entire code in the above example into a single function so that we can execute the function every second to get the latest time as in the below example.



- In the above example if we try updating the input field, it won't be able to update it. This is expected as we are basically throwing away the whole DOM node every second and regenerating it.
- But when we try to update the input element rendered by React API, we will be able to update the value.
- The above works with React, because the whole React rendering code is within the ticking timer and React is changing only the timestamp text and not the whole DOM node. Due to this reason the text input is not regenerated and we are able to type in the value.
- In the below screenshot it can be seen that HTML div element is completely regenerated but only the React pre element tag containing the time stamp text is being regenerated



- React works on the Diffing Algorithm which updates only the updated elements and not the whole DOM.
- The diffing process is possible because of the React's virtual DOM and the fact that we have the representation of our User Interface in memory because it is written in JS.
- For every second in the above example, React keeps the last UI version in memory and when it has a new one to take to the browser that new UI version will also be in memory so that React can compute the difference between the new and the old versions.
- Based on the above React will instruct the browser to update only the computed element and not the whole DOM node.

## ECMA Script and TC 39

### React.js Beyond The Basics / Introduction

Modern JavaScript has a lot of features. If you want a short list, here is one that includes only the features that are especially relevant to React:

- The new object literal syntax
- Template strings
- Block scopes and the use of let and const
- Arrow functions
- Destructuring objects and arrays in variables and arguments
- Default function arguments and the REST/ SPREAD operator
- Classes (which are used slightly in defining component, but to be avoided otherwise)
- Static and instance properties for a class. The new class-field syntax to define functions with the arrow syntax and avoid the binding problem
- Promise objects and how to use them with `async / await`
- Importing and exporting modules. Default exports and named exports

#### Variables and Block Scopes

##### ○ {{{}}}

- Is this valid JS code? - Yes
- These are nested block scopes.
- We can write as below and access the v variable outside

```

{{{ var v = 42; }}}
|

```

- Problem with var scope is that it can be accessed outside the block as in the below example

```

for (var i = 1; i <= 10; i++) {
 // Block Scope
}

```

"i" can be accessed outside the for loop and it prints 11, to overcome this issue we can use "let".

- If we replace var with let in the above for loop and try to access "i" outside the for loop, we get an error "i is not defined".

```

{
 // Block Scope
 {
 // Nested Block Scope!
 // let, const
 }
}

```

- Within each level of a block scope in a Nested block scopes the scope will protect the variables in it as long as we use the let and const keyword.
- We use const when the reference assigned to a variable is meant to be a constant one. References assigned with const cannot be changed.

```

// Scalar values
const answer = 42;
const greeting = 'Hello';

// Arrays and Objects
const numbers = [2, 4, 6];
const person = {
 firstName: 'John',
 lastName: 'Doe',
};

```

- The scalar values like integer or string cannot be mutated.
- But when const used for an array or object can be mutated as reference will be pointed to the same array or object.

```

const answer1 = 42;

/*
A big program here...
*/

answer1; // is still 42

// vs

let answer2 = 42;

/*
A big program here...
*/

answer2; // might have changed

```

- When using const and if we write some code after that and check the value of the const it will be the same, we don't have to confirm if it changed but when using let we have to make sure that it has not changed.

#### - Arrow functions

```

const X = function () {
 // "this" here is the caller of X
};

const Y = () => {
 // "this" here is NOT the caller of Y

 // It's the same "this" found in Y's scope
};

```

```

/*
Regular functions give access to their "calling" environment while arrow
functions give access to their "defining" environment

The value of the "this" keyword inside a regular function depends on HOW the
function was CALLED (the OBJECT that made the call).

The value of the "this" keyword inside an arrow function depends on WHERE the
function was DEFINED (the SCOPE that defined the function).
*/

```

- An arrow function will close over the value of the this keyword for its scope at the time it was defined.
- This makes it great for delayed execution cases like events and listeners because it gives easy access to the defining environment, not the calling environment.
- In the below example **this** refers to `{id: "REPL", main: f}`

```

const testerObj = {
 func1: function () {
 console.log('func1', this);
 },
 func2: () => {
 console.log('func2', this);
 },
};

testerObj.func1();
testerObj.func2();

```

☐ Preserve log
 ☒ Eager evaluation

☐ Selected context only
 ☒ Autocomplete from history

☒ Group similar

Console was cleared VM487:116  
 func1 ▶ {func1: f, func2: f} VM487:119  
 func2 ▶ {id: "REPL", main: f} VM487:122  
 >

- For arrow functions that has only a single line it can be written without braces as below

```

const square1 = (a) => {
 return a * a;
};

```

```

const square2 = (a) => a * a;

```

```

const square3 = a => a * a;

```

## - Object Literals

```

/*
 const obj = {
 key: value
 };
*/

```

```

const obj = {
 p1: 10,
 p2: 20,
 f1() {},
 f2: () => {},
};

```

- Object properties can hold integers, strings, function or an arrow function as in the above example.
- Modern object literals also support dynamic properties using this syntax [mystery]: 42
- Javascript will evaluate the variable inside the square brackets and add the value dynamically.
- Interview question:

```

const mystery = 'answer';

const obj = {
 p1: 10,
 p2: 20,
 f1() {},
 f2: () => {},
 [mystery]: 42,
};

console.log(obj.mystery);

```

- Ans: **undefined** (mystery is not a key, it is a variable which will be evaluated)
- console.log(obj.answer); Ans: 42**

- Shortcuts if writing a key-value pair when both have the same name
- Ex.

```

/*
 const obj = {
 key: value
 };
*/

```

```

const mystery = 'answer';
const InverseOfPI = 1 / Math.PI;

```

```

const obj = {
 p1: 10,
 p2: 20,
 f1() {},
 f2: () => {},
 [mystery]: 42,
 InverseOfPI: InverseOfPI,
};

```

- The above can be written as below

```
const mystery = 'answer';
const InverseOfPI = 1 / Math.PI;

const obj = {
 p1: 10,
 p2: 20,
 f1() {},
 f2: () => {},
 [mystery]: 42,
 InverseOfPI,
};
```

## - De-structuring and Rest/Spread

- De-structuring will work with both arrays and objects.
- Example of De-structuring is as below

```
// const PI = Math.PI;
// const E = Math.E;
// const SQRT2 = Math.SQRT2;

const {PI, E, SQRT2} = Math;
```

- De-structuring can be used in arguments of function when we do not want the whole object but instead need only some of the properties
- Example:

```
const circle = {
 label: 'circleX',
 radius: 2,
};

const circleArea = ({radius}) =>
 (PI * radius * radius).toFixed(2);

console.log(
 circleArea(circle)
);
```

- In the above example, we passed circle object to the function, but de-structured the function argument to take only radius.
- De-structured arguments can also be defined with defaults as in the below example

```
const circle = {
 label: 'circleX',
 radius: 2,
};

const circleArea = ({radius, {precision = 2}} =>
 (PI * radius * radius).toFixed(precision);

console.log(
 circleArea(circle)
);
```

- De-structured arguments can also be defined with defaults and made optional as in the below example

```
const circle = {
 label: 'circleX',
 radius: 2,
};

const circleArea = ({radius, {precision = 2} = {}} =>
 (PI * radius * radius).toFixed(precision);

console.log(
 circleArea(circle)
);
```

- De-structural arguments offers a good alternative using named arguments instead of positioning.
- Array de-structuring can also be done as below
- We can assign the array values to local values, but we can also notice that we can skip assigning to a local variable as we did for 30 in the below example

```
const [first, second, , forth] = [10, 20, 30, 40];
```

- Array de-structuring is useful when used with rest/spread operator. Below is an example

```
const [first, ...restOfItems] = [10, 20, 30, 40];

console.log(first);
console.log(restOfItems);
```



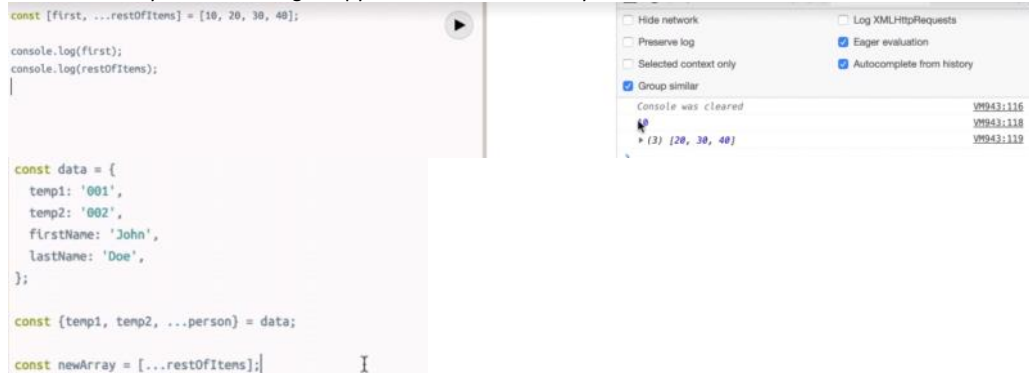
- The above is useful when we have to filter an array or split an array.
- We can also use rest operator for an object to filter out certain properties from the object. Below is an example



```
const data = {
 temp1: '001',
 temp2: '002',
 firstName: 'John',
 lastName: 'Doe',
};

const {temp1, temp2, ...person} = data;
```

- In above example it filters out temp1, temp2 and then we have a separate object person for the rest of the properties.
- Just like the three dots of rest, we can use Spread operator to spread one array or object to a new array or object. Below is an example where it shows newArray will be creating a copy of the restOfItems array which was de-structured



- Spread operator is useful for copying arrays and objects.
- Similarly for an object, we can also spread the key-value pairs of an object into a new object as in the below example

```
const data = {
 temp1: '001',
 temp2: '002',
 firstName: 'John',
 lastName: 'Doe',
};

const {temp1, temp2, ...person} = data;

const newArray = [...restOfItems];

const newObject = {
 ...person
};
```

- These are shallow copies.

#### - Template strings

```
const greeting = "Hello World";

const answer = 'Forty Two';

const html = `
<div>
 ${Math.random()}
</div>
`;

html
```

- We can use back ticks for multiple lines as in the above example

#### - Classes

```

class Person {
 constructor(name) {
 this.name = name;
 }
 greet() {
 console.log(`Hello ${this.name}!`);
 }
}

class Student extends Person {
 constructor(name, level) {
 super(name);
 this.level = level;
 }
 greet() {
 console.log(`Hello ${this.name} from ${this.level}`);
 }
}

const o1 = new Person("Max");
const o2 = new Student("Tina", "1st Grade");
const o3 = new Student("Mary", "2nd Grade");
o3.greet = () => console.log('I am special!');

o1.greet();
o2.greet();
o3.greet();

```

- We have Person class and Student class in the above example
- Student can extend the Person class. So, every student is also a Person.
- Both classes define a constructor function.
- The constructor function is called every time we instantiate an object out of the class, which we do using the new keyword.
- As in the above example we are instantiating one object from the person class and two objects from the student class.
- Arguments passed while instantiating using the new keyword will be passed to the constructor function of the class.
- The Person class expects a name argument, and it stores that value on the instance using **this** keyword.
- The Student class expects a name argument and level argument.
- It stores the level value on its instance and since the class extends Person class, it will call the super method with the name argument, which will invoke the Person class constructor function and store the name as well.
- Both classes define a greet function that uses the values stored on each instance.

#### - Promises and Async/Await

- For asynchronous operations we use promise objects.

```

const fetchData = () => {
 fetch('https://api.github.com').then(res => {
 res.json().then(data => {
 console.log(data)
 });
 });
};

fetchData();

```

- Promise is an object that might deliver data at a later point in the program.
- Web API fetch returns a promise. So, to consume the promise we use a .then call on the result of fetch and supply a callback function.
- The callback function will receive data from the API.
- To parse the data as JSON received from the fetch API which gives a raw response, we need to use a json method on the response object.
- The json() method is again an asynchronous one, hence it returns a promise as well.
- To get the data, we need another .then call on the result of the json method, and in the callback of that, we can access the data.
- The syntax might get complicated with more nesting of asynchronous operations or when we need to combine this with any looping logic.
- We can simplify the nesting here by making each callback return the promise object, but the whole .then syntax is a bit less readable than the modern way to consume promises in javascript, which is using async/await.
- We just await on the asynchronous calls that returns a promise which gives us the response object. Then we can await on the json method to access the JSON data.

```

const fetchData = async () => {
 const res = await fetch('https://api.github.com');
 const data = await res.json();
 console.log(data);
};

fetchData();

```

- To make the await calls we need to make the function as async.
- The async await syntax is just a way for us to consume promises without having to nest .then calls.
- Once we make the function as async, the function itself will become an asynchronous one and it will return a promise object.

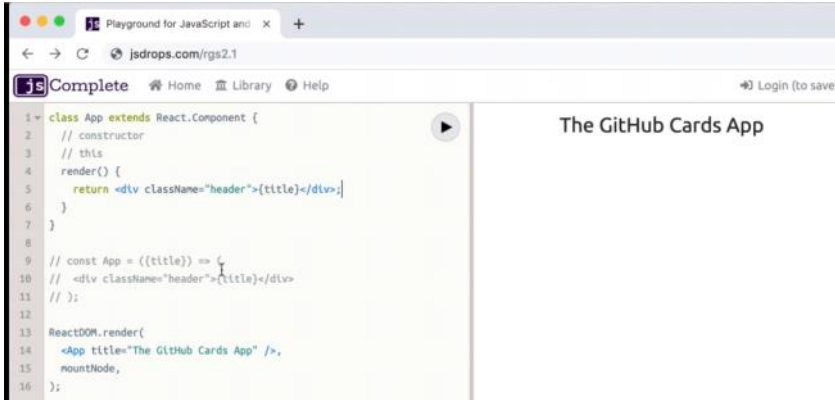
#### The Github Cards app (using Class components)

## - React Class Components

- First decision to be made in a React application is on the component structure.
- We need to know how many components to use and what each component should describe.
- Use top to bottom approach if we know what are all the components in the tree.

```
class App extends React.Component {
 // constructor
 // this
 render() {}
}
```

- Each React component will have a render function and there will be only one and is required.



- However receiving props as arguments, in class components, the state and props are used/managed through the instance of the class.
- So title props will be used as **this.props.title**

```
class App extends React.Component {
 // constructor
 // this
 render() {
 return <div className='header'>{this.props.title}</div>;
 }
}
```

- Below is an alternative way to style



- The above is called Javascript styling and is good for conditional styling.
- Ex.



- With Javascript style objects it is difficult to write media queries.
- Babel-plugin-css-with-js plugin will help dev with media queries in javascript.
- React-native-web plugin will also help dev with media queries in javascript if we are using react-native to develop.

## - Working with data

- To get github user data we can use the below link

```

{
 "login": "yrkapil",
 "id": 1701346,
 "node_id": "MDQ6VXN1c3E3MDEzNDY=",
 "avatar_url": "https://avatars3.githubusercontent.com/u/1701346?v=4",
 "gravatar_id": "",
 "url": "https://api.github.com/users/yrkapil",
 "html_url": "https://github.com/yrkapil",
 "followers_url": "https://api.github.com/users/yrkapil/followers",
 "following_url": "https://api.github.com/users/yrkapil/following{/other_user}",
 "gists_url": "https://api.github.com/users/yrkapil/gists{/gist_id}",
 "starred_url": "https://api.github.com/users/yrkapil/starred{/owner}{/repo}",
 "subscriptions_url": "https://api.github.com/users/yrkapil/subscriptions",
 "organizations_url": "https://api.github.com/users/yrkapil/orgs",
 "repos_url": "https://api.github.com/users/yrkapil/repos",
 "events_url": "https://api.github.com/users/yrkapil/events{/privacy}",
 "received_events_url": "https://api.github.com/users/yrkapil/received_events",
 "type": "User",
 "site_admin": false,
 "name": null,
 "company": null,
 "blog": null,
 "location": null,
 "email": null,
 "hireable": null,
 "bio": null,
 "public_repos": 13,
 "public_gists": 0,
 "followers": 1,
 "following": 0,
 "created_at": "2012-05-03T06:28:34Z",
 "updated_at": "2017-09-15T07:51:59Z"
}

```

- When we need to send props to a component we can use a spread operator with an object in the React component all the properties of that object will become props for the component as in the below example

```

const testData = [
 {name: "Dan Abramov", avatar_url: "https://avatars0.githubusercontent.com/u/810438?v=4", company: "Facebook"},
 {name: "Sophie Alpert", avatar_url: "https://avatars2.githubusercontent.com/u/6820?v=4", company: "Facebook"},
 {name: "Sebastian Markbåge", avatar_url: "https://avatars2.githubusercontent.com/u/63648?v=4", company: "Facebook"},
];

const CardList = (props) => (
 <div>
 <Card {...testData[0]} />
 <Card {...testData[1]} />
 </div>
);

class Card extends React.Component {
 render() {
 const profile = this.props;
 return (
 <div className="github-profile">

 <div className="info">
 <div className="name">{profile.name}</div>
 <div className="company">{profile.company}</div>
 </div>
 </div>
);
 }
}

```

- We know that objects can be instantiated from classes and we call every object an instance.
- An instance in a React application - Everytime we use a class component we are doing it here twice in the below example

```

const CardList = (props) => (
 <div>
 <Card {...testData[0]} />
 <Card {...testData[1]} />
 </div>
);

```

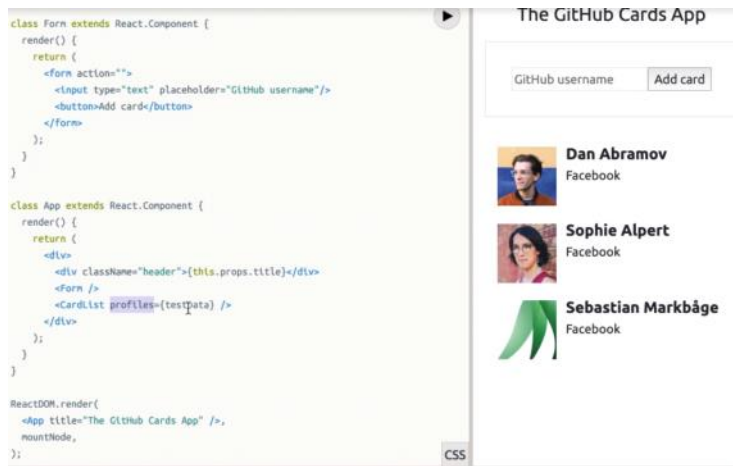
- So React internally creates an instance from the component and uses it to render the element. This instance is something that React keeps in memory for each rendered element. It is not really in the browser, the browser is only the result of the DOM operation React came up with using that instance render method which came from the component.
- "this" in each rendered component refers to that instance.
- Instead of mentioning each array of the element and each Card for the array item we can make it dynamic as mentioned below

```

const CardList = (props) => (
 <div>
 {testData.map(profile => <Card {...profile}/>)}
 </div>
);

```

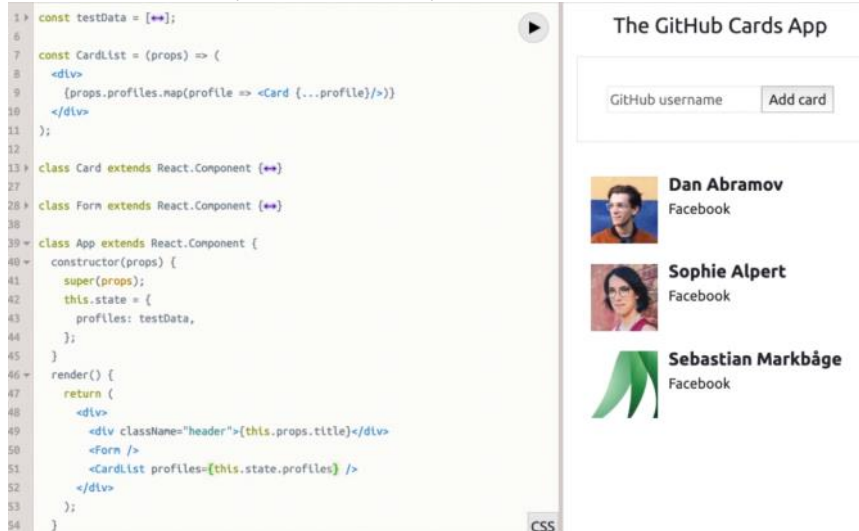
- Initializing and reading the state object



- In a class component the state is also managed on the in-memory instance that React associates with every rendered component.
- To initialize a state object for the App component we need to tap into the native class constructor method, which gets called for every instantiated object.
- This constructor method receives the instance props as well. It has to call the javascript super method to honor the link between the App class and the class that it extends from (React.Component)
- We should also pass the props as an argument to the super method.
- Once inside the constructor we will have access to the special state object that React manages for each class component.
- We can initialize this inside a constructor using this.state as shown in the below example

```
class App extends React.Component {
 constructor(props) {
 super(props);
 this.state = {};
 }
 render() {
 return (
 <div>
 <div className="header">{this.props.title}</div>
 <Form />
 <CardList profiles={testData} />
 </div>
);
 }
}
```

- Unlike useState in function components, this state instance property has to be an object in the class components and it can't be an integer or string.
- To use this state to have profiles and read the profiles we need to use it as mentioned below



- The constructor call can also be replaced as below

```
class App extends React.Component {
 // constructor(props) {
 // super(props);
 // this.state = {
 // profiles: testData,
 // };
 // }

 state = {
 profiles: testData,
 };
}
```

- This is not yet official in the javascript language but will be.

- These work in the playground as it uses babel to transpile things to normal javascript that the browser will understand.

#### - Taking input from the user

- Inputs can be referenced using React instead of using document.getElementById
- React has a special property called "Ref" that we can use to reference the element. This is kind of like a fancy ID that React keeps in memory and associates with every rendered element.

```
<form onSubmit={this.handleSubmit}>
 <input type="text" placeholder="Github username" ref={} required />
 <button>Add card</button>
</form>
```

- To use a Ref we need to instantiate an object which can be named anything but we need to do a React.createRef() call.

```
class Form extends React.Component {
 userNameInput = React.createRef();
 handleSubmit = (event) => {
 event.preventDefault();
 };
 render() {
 return (
 <form onSubmit={this.handleSubmit}>
 <input type="text" placeholder="Github username" ref={this.userNameInput}
 required />
 <button>Add card</button>
 </form>
);
 }
}
```

- The current value is stored under .current and the reference is used as below and this refers to the html input itself

```
class Form extends React.Component {
 userNameInput = React.createRef();
 handleSubmit = (event) => {
 event.preventDefault();
 console.log(
 this.userNameInput.current
);
 };
 render() {
 return (
 <form onSubmit={this.handleSubmit}>
 <input
 type="text"
 placeholder="Github username"
 ref={this.userNameInput}
 required
 />
 </form>
);
 }
}
```

- So to get the value from the element we refer to "this.userNameInput.current.value"

```
class Form extends React.Component {
 userNameInput = React.createRef();
 handleSubmit = (event) => {
 event.preventDefault();
 console.log(
 this.userNameInput.current.value
);
 };
 render() {
 return (
 <form onSubmit={this.handleSubmit}>
 <input
 type="text"
 placeholder="Github username"
 ref={this.userNameInput}
 required
 />
 </form>
);
 }
}
```

- Rather than reading values from DOM elements, we have another method to read values directly through React itself. This method is often labelled as "**Controlled components**" and it has some advantages over the simple ref property.
- To use the controlled components we use the state object we define an element to handle the input value of the username field and initialize with an empty string as in the below example
- And we use it as the value of the element

```

class Form extends React.Component {
 state = { userName: '' };
 handleSubmit = (event) => {
 event.preventDefault();
 console.log()
 };
 render() {
 return (
 <form onSubmit={this.handleSubmit}>
 <input
 type="text"
 value={this.state.userName}
 placeholder="Github username"
 required
 />
 <button>Add card</button>
 </form>
);
 }
}

```

- The above create a controlled element.
- Now as React controls the value, we need an onChange event, so that the DOM can tell React that something has changed in this input and needs to be reflected in UI as well.
- To set the value typed in the input box, we use `this.setState` and pass it an object that has the new state.

```

class Form extends React.Component {
 state = { userName: '' };
 handleSubmit = (event) => {
 event.preventDefault();
 console.log()
 };
 render() {
 return (
 <form onSubmit={this.handleSubmit}>
 <input
 type="text"
 value={this.state.userName}
 onChange={event => this.setState({ userName: event.target.value })}
 placeholder="Github username"
 required
 />
 <button>Add card</button>
 </form>
);
 }
}

```

- To use the value, we can access it using `"this.state.userName"`

```

class Form extends React.Component {
 state = { userName: '' };
 handleSubmit = (event) => {
 event.preventDefault();
 console.log(this.state.userName);
 };
 render() {
 return (
 <form onSubmit={this.handleSubmit}>
 <input
 type="text"
 value={this.state.userName}
 onChange={event => this.setState({ userName: event.target.value })}
 placeholder="Github username"
 required
 />
 <button>Add card</button>
 </form>
);
 }
}

```

- The difference between using Ref and Controlled Components is that the state tracks on each character typed in the input box whereas when Ref is used React will not be aware of what is being typed in the input box.
  - So Controlled Components can be used for live feedback of what the user types. (Ex. Password Strength indicator/Character count in text area)
- Working with Ajax calls
- In this example we use axios to fetch data from the github api.
  - Axios returns a promise, so to use this we need to use the await method and to use await we need to name the function as `async` as in the below example



```

class Form extends React.Component {
 state = { userName: '' };
 handleSubmit = async (event) => {
 event.preventDefault();
 const resp = await
 axios.get('https://api.github.com/users/${this.state.userName}');
 console.log(resp);
 };
 render() {
 return (
 <form onSubmit={this.handleSubmit}>
 <input
 type="text"
 value={this.state.userName}
 onChange={event => this.setState({ userName: event.target.value })}
 placeholder="GitHub username"
 required
 />
 <button>Add card</button>
 </form>
);
 }
}

```

- React has only one way flow of data and the component can't change the state of the parent directly.
- Below is the complete example

```

class Form extends React.Component {
 state = {addUserInput : ''};
 handleSubmit = async (event) => {
 const resp = await
 axios.get('https://api.github.com/users/${this.state.addUserInput}');
 this.props.onSubmit(resp.data);
 this.setState({ addUserInput: '' });
 }
 render() {
 return (
 <div className="add-user-form">
 <input type="text" placeholder="Type the name of the user" value=
 {this.state.addUserInput} onChange={event => this.setState({addUserInput:
 event.target.value})}/>
 <button onClick={this.handleSubmit}>Add user</button>
 </div>
);
 }
}

```

```

const CardList = (props) => {
 return (
 <React.Fragment>
 {props.profiles.map(profile => <Card {...profile} />)}
 </React.Fragment>
);
}

class Card extends React.Component {
 render() {
 const profile = this.props;
 return (
 <div className="github-profile">

 <div className="info">
 <div style={{fontWeight: 'bold'}}>{profile.name}</div>
 <div className="company">{profile.company}</div>
 </div>
 </div>
);
 }
}

```

```

class App extends React.Component {
 state = {
 profiles: []
 };
 addUserProfile = (profile) => {
 this.setState(prevState => ({
 profiles: [...prevState.profiles, profile]
 }));
 };
 render() {
 return (
 <React.Fragment>
 <div className="header">{this.props.title}</div>
 <Form onSubmit={this.addUserProfile}/>
 <CardList profiles={this.state.profiles} />
 </React.Fragment>
)
 }
}

ReactDOM.render(
 <App title="My App" />,
 mountNode
);

```

- Key property is required for dynamic lists in React which should be unique, below is an example

```

const CardList = (props) => {
 <div>
 {props.profiles.map(profile => <Card key={profile.id} {...profile}/>)}
 </div>
};

```

```

<div className="github-profile">
 Handle Errors
 a) Invalid Input
 b) Network Problems

```

- Code for the above example

```

class Form extends React.Component {
 state = {addUserInput: ''};
 handleSubmit = async (event) => {
 const resp = await axios.get(`https://api.github.com/users/${this.state.addUserInput}`);
 this.props.onSubmit(resp.data);
 this.setState({ addUserInput: '' });
 }
 render() {
 return (
 <div className="add-user-form">
 <input type="text" placeholder="Type the name of the user" value={this.state.addUserInput} onChange={event => this.setState({ addUserInput: event.target.value})}/>
 <button onClick={this.handleSubmit}>Add user</button>
 </div>
);
 }
}

```

```

const CardList = (props) => {
 return (
 <React.Fragment>
 {props.profiles.map(profile => <Card {...profile} />)}
 </React.Fragment>
);
};

```

```

}

class Card extends React.Component {
 render() {
 const profile = this.props;
 return (
 <div className="github-profile">

 <div className="info">
 <div style={{fontWeight: 'bold'}}>{profile.name}</div>
 <div className="company">{profile.company}</div>
 </div>
 </div>
);
 }
}

class App extends React.Component {
 state = {
 profiles: []
 };
 addUserProfile = (profile) => {
 this.setState(prevState => ({
 profiles: [...prevState.profiles, profile]
 }));
 };
 render() {
 return (
 <React.Fragment>
 <div className="header">{this.props.title}</div>
 <Form onSubmit={this.addUserProfile}/>
 <CardList profiles={this.state.profiles} />
 </React.Fragment>
)
 }
}

ReactDOM.render(
 <App title="My App" />,
 mountNode
);

```

#### Miscellaneous

- ES6 Array.from()

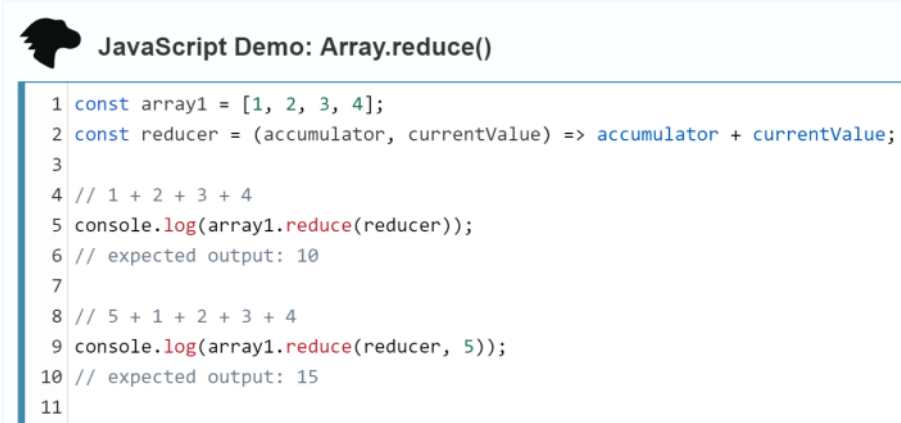
```

// Generate a sequence of numbers
// Since the array is initialized with `undefined` on each position,
o // the value of `v` below will be `undefined`
Array.from({length: 5}, (v, i) => i);
// [0, 1, 2, 3, 4]

```

- Array.reduce()

The `reduce()` method executes a **reducer** function (that you provide) on each element of array, resulting in a single output value.



```

1 const array1 = [1, 2, 3, 4];
2 const reducer = (accumulator, currentValue) => accumulator + currentValue;
3
4 // 1 + 2 + 3 + 4
5 console.log(array1.reduce(reducer));
6 // expected output: 10
7
8 // 5 + 1 + 2 + 3 + 4
9 console.log(array1.reduce(reducer, 5));
10 // expected output: 15
11

```


- Usage of `useState` in StarMatch app



```

const StarMatch = () => {
 const [stars, setStars] = useState(utils.random(1, 9));
 return (
 <div className="game">
 <div className="help">
 Pick 1 or more numbers that sum to the number of stars
 </div>
 <div className="body">
 <div className="left">
 {utils.range(1, stars).map(starId =>
 <div key={starId} className="star" />
)}
 </div>
 <div className="right">
 {utils.range(1, 9).map(number =>
 <button key={number} className="number">{number}</button>
)}
 </div>
 </div>
 <div className="timer">Time Remaining: 10</div>
 </div>
);
};

```



```

const [stars, setStars] = useState(utils.random(1, 9));
const [availableNums, setAvailableNums] = useState(utils.range(1, 9));
const [candidateNums, setCandidateNums] = useState([]);

```

- Using Number constructor to convert string to number

- `const [stars, setStars] = useState(utils.random(Number("1"), 9));`



```

const PlayNumber = props => (
 <button className="number" onClick={() => console.log('Num', props.number)}>
 {props.number}
 </button>
);

```

**Question:** What JavaScript concept enabled 9 different click handlers to report their numbers

(Pause and think)

- Answer - Javascript closures
  - Each `onClick` function here closes over the scope of its owner number and gives us access to its props.
  - We have 9 `onClick` handlers and all have different closures closing over different scopes. This is important to understand and remember when working with stateful function components React as they depend on closures.
- Note: Don't place on the state anything that could be computed from the other things that you place on the state.
  - In this example below we can have different types of numbers : `candidateNums` (numbers clicked), `usedNums`, `availableNums`, `wrongNums`

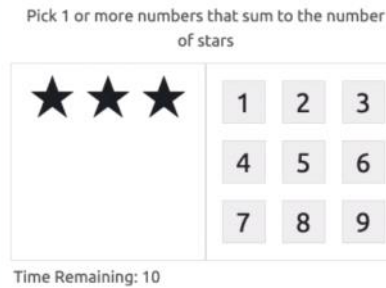
```

const StarsDisplay = props => {
 <>
 {utils.range(1, props.count).map(starId =>
 <div key={starId} className="star" />
)}
 </>
};

const PlayNumber = props => {
 <button className="number" onClick={() => console.log('Num', props.number)}>
 {props.number}
 </button>
};

const StarMatch = () => {
 const [stars, setStars] = useState(utils.random(1, 9));
 // candidateNums
 // wrongNums
 // usedNums
 // availableNums
 return (
 <div className="game">
 <div className="help">
 Pick 1 or more numbers that sum to the number of stars
 </div>
 <div className="body">
 <div className="left">
 <StarsDisplay count={stars}/>

```



- wrongNums should not be placed in the state as wrongNums are also numbers clicked but do not match to the sum of the numbers clicked with the stars displayed.
- Also usedNums is an inverse of availableNums, if number is used it is not available so we can use either of them in the state.
- Side effects of Hooks in React
  - Till now we have used only 1 hook in the above examples (useState), but there are other hooks available in React
    - React.useEffect - Used to introduce some side effect to the component.
    - It takes in a function and it will run this function every time the owner component renders itself.
    - Below is an example, where the log will be generated everytime the component is done rendering.
    - So when we click on the number the component has to re-render itself because the state has changed, and when it is done rendering again it logs the message.

```

17);
18
19 const PlayAgain = props => {
20 <div className="game-done">
21 <button onClick={props.onClick}>Play Again</button>
22 </div>
23 };
24
25 const StarMatch = () => {
26 const [stars, setStars] = useState(utils.random(1, 9));
27 const [availableNums, setAvailableNums] = useState(utils.range(1, 9));
28 const [candidateNums, setCandidateNums] = useState([]);
29 const [secondsLeft, setSecondsLeft] = useState(10);
30 // setTimeout
31 useEffect(() => {
32 console.log('Rendered...');
33 });
34

```



- Best Practice: Whenever we create a side effect, you have to clean that side effect when it is no longer needed.
- For cleaning the side effect of the useEffect() hook lies with in the returned value of this callback function in the side effect.
- We can return a function here and react will invoke this function when it is about to unmount the component (when it is about to re-render the component/when ever there is a change in the component)
- Below is an example, where the initial rendering log ("Done Rendering") is displayed, and when the component is re-rendered/changed it displays the next log("Component is going to re-render") followed by the initial log again.

```

 });

 const PlayAgain = props => {
 <div className="game-done">
 <button onClick={props.onClick}>Play Again</button>
 </div>
 };

 const StarMatch = () => {
 const [stars, setStars] = useState(utils.random(1, 9));
 const [availableNums, setAvailableNums] = useState(utils.range(1, 9));
 const [candidateNums, setCandidateNums] = useState([]);
 const [secondsLeft, setSecondsLeft] = useState(10);

 useEffect(() => {
 // If (secondsLeft > 0) {
 // setTimeout(() => {
 // setSecondsLeft(secondsLeft - 1);
 // }, 1000);
 // }
 console.log('Done rendering')
 return () => console.log('Component is going to rerender');
 });
 };

```

Pick 1 or more numbers that sum to the number of stars

Time Remaining: 10

Hide network Log XMLHttpRequest

Preserve log Eager evaluation

Selected context Autocomplete

Group similar

Console was cleared VM644:116

Done rendering VM644:143

Component is going to rerender VM644:144

Done rendering VM644:143

Component is going to rerender VM644:144

Done rendering VM644:143

- Also when we unmount the component, React will call all the clean ups that we have. It is a lot cleaner way to implement Play Again Component.

#### ○ Unmounting and Remounting Components

- We can specify a key attribute to the component, which is a unique identifier that React uses to identify a component element.
- So if the key attribute changes from 1 to 2, react will see it as a completely different component.
- So when we change the key attribute value of a component, react will unmount the component and mount a new component.
- And when React unmounts the component with key={1}, it will reset all the side effects and when it mounts the new component with key={2} it will be initialized with the new states.

```

const StarMatch = () => {
 const [gameId, setGameId] = useState(1);
 return <Game key={gameId} startNewGame={() => setGameId(gameId + 1)} />;
}

```

#### - Using custom hooks

- In the above app, we have managed the states, side effects of hooks and rendering logic in the same function.
- We can separate the management of states(Initialization and Transactions on states) and side effects of hooks(React.useEffect()) in a different function by defining a custom hook.
- Best Practice: Name of the custom hook should be pre-fixed with use, to know that the function contains react hooks. Also the custom hook should follow the rule of hooks.
- Rule of Hooks: We should always use the React Hooks function in the same order, so we cannot define them conditionally. We cannot have an if statement for one of the hook.

## #1 Rule of Hooks Don't call Hooks inside loop or conditions

```

return { stars, availableNums, candidateNums, secondsLeft, setGameState };
};

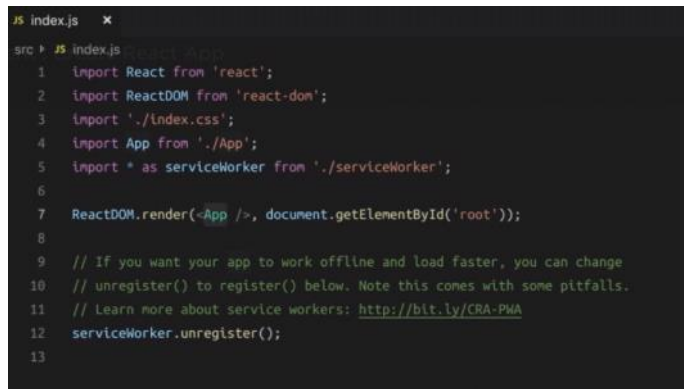
const Game = (props) => {
 const {
 stars,
 availableNums,
 candidateNums,
 secondsLeft,
 setGameState,
 } = useGameState();
}

```

- The custom hook returns all the required states and hooks to the component and the component can use it.

#### - Setting up a Development Environment

- \$ npm i -g create-react-app && create-react-app cra-test
- i -> install
- g -> global
- npm i -g <create-react-app-package-name> && <create-react-app-command> <app-name>
- A better way to do it is using it via the below command using npx <create-react-app-command> <app-name>
- npx create-react-app cra-test
- The x in npx is for execute, so we are actually executing a npm package here
- If we do not have the package, npm will download it and cache it locally, but it will also download a new one if needed when we run the same command again.
- So create-react-app will take over and install its own dependencies (react, react-dom and react-scripts)
- Once the app is created, we can see the index.js as below which renders the top-level App component

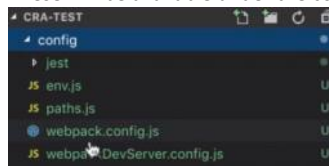


```

1 import React from 'react';
2 import ReactDOM from 'react-dom';
3 import './index.css';
4 import App from './App';
5 import * as serviceWorker from './serviceWorker';
6
7 ReactDOM.render(<App />, document.getElementById('root'));
8
9 // If you want your app to work offline and load faster, you can change
10 // unregister() to register() below. Note this comes with some pitfalls.
11 // Learn more about service workers: http://bit.ly/CRA-PWA
12 serviceWorker.unregister();
13

```

- We need to import the required into the js file and none of the react services are global as in the react playground.
- Webpack tool is internally used in react, so that we can import the css files needed as in the above screenshot.
- So with all defaults in app we don't need to worry on how JSX is compiled. It is all abstracted internally in the create-react-app package. However, we can eject the application from that internal application using the "npm run eject" command and this action is permanent.
- This will copy all the configurations and scripts used by this tool locally to your project, and we can modify the according to the requirement.
- These will be available under the config folder as in the below screenshot



- Installing environmental dependencies required for react app
  - <https://jscomplete.com/learn/1rd-reactful>
  - npm init - will help us create the package.json
  - npm i express - express is a framework to create a Node.js web server. Web server is the first step to do server side rendering of the react app
    - Npm would download express and place it under node\_modules folder(this will be created by npm itself) and the command also saves this dependency to package.json
  - npm i react - to include the react core library
  - npm i react-dom - to render a react app to dom and also for server side rendering
    - The above can also be written as "npm i react react-dom"
  - npm i webpack webpack-cli - when we ship everything(react, react-dom, express) to the browser, we ship it as a single bundle as browsers do not know on how to work with modules yet. For that we need to use webpack to bundle all the modules/applications in a single file and ship that to the browser.
    - Webpack - core library
    - Webpack-cli - command to invoke the webpack
  - npm i babel-loader @babel/core @babel/node @babel/preset-env @babel/preset-react
    - Babel is the package that compiles JSX into regular React API calls
    - We need to hook babel in to the webpack process as webpack is going to bundle our system.
    - So we need to inform webpack that while it is bundling the system if it sees any JSX stuff it should invoke Babel to convert this JSX stuff in to React API calls. The package to do that is "babel-loader".
    - The other packages are used for configuring babel.
      - @babel/core - core package for babel compiler
      - @babel/node - node package used for server-side rendering and we need node to understand JSX for the server side rendering
      - @babel/preset-env - To use any modern JS and also target old browsers. (Ex. Preset-env can help compile the code to not use any modern JS that old browsers cant understand). We can also mention which browsers to target.
      - @babel/preset-react - minimum configuration required to tell babel to compile React
  - npm i -D nodemon
    - -D -> to save under devDependencies section in package.json
    - Nodemon is used to run the node server in a wrapper process that monitors the main process and automatically restarts when the files are saved to the disk
    - Nodemon is a watcher on top of the node command and it automatically restarts when changes are saved to the disk.
  - npm i -D eslint babel-eslint eslint-plugin-react
    - ESLint is a code quality tool
    - To configure ESLint we need to add a .eslintrc.js file to the root of the project.
    - Provides feedback/checks as you type the code (Ex. Using single quotes through out the app, unused variables, undefined variables etc.)
  - npm i -D jest babel-jest react-test-renderer
    - Jest is a testing library used with react

## - Configuring dependencies



```

.eslintrc.js

module.exports = {
 parser: 'babel-eslint',
 env: {
 browser: true,
 commonjs: true,
 es6: true,
 node: true,
 jest: true,
 },
 extends: ['eslint:recommended', 'plugin:react/recommended'],
 parserOptions: {
 ecmaFeatures: {
 jsx: true,
 },
 sourceType: 'module',
 },
 plugins: ['react'],
 rules: {
 // You can do your customizations here...
 // For example, if you don't want to use the prop-types package,
 // you can turn off that recommended rule with: 'react/prop-types': ['off']
 },
};

```

## - Initial directory structure

### 4. Creating an Initial Directory Structure

This really depends on your style and preferences, but a simple structure would be something like:

```

fulljs/
├── dist/
│ └── main.js
├── src/
│ ├── index.js
│ └── components/
│ ├── App.js
│ └── server/
│ └── server.js

```

- The above is recommended as these are the defaults(folders and files) which webpack will look for.
- Webpack is going to take the files in the src directory and is going to write them into the dist folder

## - Configuring webpack and babel

```

babel.config.js

module.exports = {
 presets: ['@babel/preset-env', '@babel/preset-react'],
};

```

- We have already installed the preset-env and preset-react using npm

```

webpack.config.js

module.exports = {
 module: {
 rules: [
 {
 test: /\.js$/,
 exclude: /node_modules/,
 use: {
 loader: 'babel-loader',
 },
 },
],
 },
};

```

- We did not specify the entry point or output point for webpack because we are using the defaults.



Webpack has certain defaults on which JavaScript file to start with. It looks for a `src/index.js` file. It'll also output the bundle to `dist/main.js` by default. If you need to change the locations of your `src` and `dist` files, you'll need a few more configuration entries in `webpack.config.js`.

- An example if we use the entry and output is as below

```
module.exports = {
 entry: {
 'app': ['./app/index.ts', './app/styles/style.scss'],
 'vendor': './vendor.ts'
 },
 output: {
 path: path.resolve(__dirname, 'dist'),
 filename: '[name].js',
 chunkFilename: '[name].js'
 },
};
```

- We have to mention babel-loader in loader in the module rules
  - This is to tell webpack to invoke babel on any file that ends with js. As the app is a react app and we will be writing JSX in js files, we will need Babel to run on those files before webpack picks them up and includes them in the bundle.

webpack.config.js

```
module.exports = {
 module: {
 rules: [
 {
 test: /\.js$/,
 exclude: /node_modules/,
 use: {
 loader: 'babel-loader',
 },
 },
],
 },
};
```

- So as in the above webpack config rules we tell webpack to bundle all the js files which are not under the node\_modules folder and use babel loader on them.
- Now babel picks up the configuration as mentioned below and compiles all the modern JS into something that the browser understands.

babel.config.js

```
module.exports = {
 presets: ['@babel/preset-env', '@babel/preset-react'],
};
```

## - Configuring npm scripts and development

### 6. Creating npm Scripts for Development

You need 2 commands to run this environment. You need to run your web server and you need to run Webpack to bundle the frontend application for browsers. You can use **npm scripts** to manage these.

In your `package.json` file you should have a `scripts` section. If you generated the file with the `npm init` defaults you'll have a placeholder "test" script in there. You should change that to work with Jest:

```
// In package.json
scripts: {
 "test": "jest"
}
```

Add 2 more scripts in there. The first script is to run the server file with Nodemon and make it work with the same Babel configuration above. You can name the script anything. For example:

```
"dev-server": "nodemon --exec babel-node src/server/server.js --ignore dist/"
```

- In package.json we can mention the scripts that needs to be run by npm as mentioned below

```
"scripts": {
 "dev-server": "nodemon --exec babel-node src/server/server.js --ignore dist/",
 "dev-bundle": "webpack -wd"
},
```

- webpack -wd -> w is for watch mode and d is for development bundle
  - -p is for production

```
$ npm run dev-server
$ npm run dev-bundle
```

- The above is to run the dev-server and create a bundle.
- Above example is available in github under [jscomplete/rgs-template](#)

At this point, you are ready for your own code. If you followed the exact configurations above, you'll need to place your `ReactDOM.render` call (or `.hydrate` for SSR code) in `src/index.js` and serve `dist/main.js` in your root HTML response.

Here is a sample server-side ready React application that you can test with:

`src/components/App.js`

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

export default function App() {
 const [count, setCount] = useState(0);
 return (
 <div>
 This is a sample stateful and server-side
 rendered React application.

 Here is a button that will track
 how many times you click it:

 <button onClick={() => setCount(count + 1)}>{count}</button>
 </div>
);
}
```

`src/index.js`

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

import App from './components/App';

ReactDOM.hydrate(
 <App />,
 document.getElementById('mountNode'),
);
```

src/server/server.js

```
import express from 'express';
import React from 'react';
import ReactDOMServer from 'react-dom/server';
import App from '../components/App';

const server = express();
server.use(express.static('dist'));

server.get('/', (req, res) => {
 const initialMarkup = ReactDOMServer.renderToString(<App />);

 res.send(`
 <html>
 <head>
 <title>Sample React App</title>
 </head>
 <body>
 <div id="mountNode">${initialMarkup}</div>
 <script src="/main.js"></script>
 </body>
 </html>
 `);
});

server.listen(4242, () => console.log('Server is running...'));
```

#### Reactful

- Package for all the pre-configured configurations as in the above example
- `$ npx reactful star-match`

# Star Match Game - without Custom Hooks

Monday, July 22, 2019 10:17 PM

```
// STAR MATCH - V2
const NumberCard = (props) => {
 return (
 <button
 className="number"
 style={{background: colors[props.status]}}
 onClick={() => props.onClick(props.number, props.status)}
 >
 {props.number}
 </button>
);
};

const Star = (props) => {
 return (
 <React.Fragment>
 {utils.range(1, props.count).map(starId =>
 <div key={starId} className="star" />
)}
 </React.Fragment>
);
};

const StarMatch = (props) => {
 const [gameId, setGameId] = useState(0);
 return (
 <Game key={gameId} resetGame={() => setGameId(gameId + 1)} />
);
}

const PlayAgain = props => {
 return (
 <div>
 <div style={{color: (props.status === 'lost') ? 'red' : 'blue'}}>
 {(props.status === 'lost') ? (Sorry! You Lost) : (Yay! You won)}
 </div>
 Wanna Play Again
 <button onClick={props.reset}> Play Again </button>
 </div>
);
}

const Game = (props) => {
 const [stars, setStars] = useState(utils.random(1, 9));
 const [availableNums, setAvailableNums] = useState(utils.range(1, 9));
 const [candidateNums, setCandidateNums] = useState([]);
 const [countDown, setCountDown] = useState(10);
```

```

useEffect(()=>{
 if(countDown > 0 && availableNums.length > 0) {
 const timerId = setTimeout(()=>{
 setCountDown(countDown-1);
 }, 1000);
 return () => {clearTimeout(timerId);}
 }
})

const areCandidatesWrong = utils.sum(candidateNums) > stars;
const gameStatus = (availableNums.length === 0) ? 'won' : ((countDown === 0) ? 'lost' : 'active');
console.log('sum::'+utils.sum(candidateNums));
console.log('stars::'+stars);
const getNumberStatus = (num) => {
 if(!availableNums.includes(num)) {
 return 'used';
 }
 if(candidateNums.includes(num)) {
 return areCandidatesWrong ? 'wrong' : 'candidate';
 }
 return 'available';
};
const getNumberClicked = (number, status) => {
 if (status === 'used' || gameStatus !== 'active') {
 return;
 }
 const newCandidateNums = status === 'available' ? candidateNums.concat(number) :
candidateNums.filter(cn => cn !== number);
 if (utils.sum(newCandidateNums) !== stars) {
 setCandidateNums(newCandidateNums);
 } else {
 const newAvailableNums = availableNums.filter(
 n => !newCandidateNums.includes(n)
);
 console.log(newAvailableNums);
 setAvailableNums(newAvailableNums);
 setStars(utils.randomSumIn(newAvailableNums, 9));
 setCandidateNums([]);
 }
}

return (
 <div className="game">
 <div className="help">
 Pick 1 or more numbers that sum to the number of stars
 </div>
 <div className="body">
 <div className="left">
 {gameStatus !== 'active' ? (<PlayAgain status={gameStatus} reset={props.resetGame}/>) : (<Star
count={stars} />)}
 </div>
 <div className="right">

```

```

 {utils.range(1, 9).map(number =>
 <NumberCard key={number} status={getNumberStatus(number)} number={number}
onClick={getNumberClicked} />
)}
 </div>
 </div>
 <div className="timer">Time Remaining: {countDown}</div>
</div>
);
};

```

```

// Color Theme
const colors = {
 available: 'lightgray',
 used: 'lightgreen',
 wrong: 'lightcoral',
 candidate: 'deepskyblue',
};

```

```

// Math science
const utils = {
 // Sum an array
 sum: arr => arr.reduce((acc, curr) => acc + curr, 0),

```

```

 // create an array of numbers between min and max (edges included)
 range: (min, max) => Array.from({ length: max - min + 1 }, (_, i) => min + i),

```

```

 // pick a random number between min and max (edges included)
 random: (min, max) => min + Math.floor(max * Math.random()),

```

```

// Given an array of numbers and a max...
// Pick a random sum (< max) from the set of all available sums in arr
randomSumIn: (arr, max) => {
 const sets = [[]];
 const sums = [];
 for (let i = 0; i < arr.length; i++) {
 for (let j = 0, len = sets.length; j < len; j++) {
 const candidateSet = sets[j].concat(arr[i]);
 const candidateSum = utils.sum(candidateSet);
 if (candidateSum <= max) {
 sets.push(candidateSet);
 console.log('sets:' + sets);
 sums.push(candidateSum);
 }
 }
 }
 return sums[utils.random(0, sums.length)];
},
};

```

```

ReactDOM.render(<StarMatch />, mountNode);

```



# Star Match Game - using custom hooks

Monday, July 22, 2019 10:38 PM

```
// STAR MATCH - V2
const NumberCard = (props) => {
 return (
 <button
 className="number"
 style={{background: colors[props.status]}}
 onClick={() => props.onClick(props.number, props.status)}
 >
 {props.number}
 </button>
);
};

const Star = (props) => {
 return (
 <React.Fragment>
 {utils.range(1, props.count).map(starId =>
 <div key={starId} className="star" />
)}
 </React.Fragment>
);
};

const StarMatch = (props) => {
 const [gameId, setGameId] = useState(0);
 return (
 <Game key={gameId} resetGame={() => setGameId(gameId + 1)} />
);
}

const PlayAgain = props => {
 return (
 <div>
 <div style={{color: (props.status === 'lost') ? 'red' : 'blue'}}>
 {(props.status === 'lost') ? (Sorry! You Lost) : (Yay! You won)}
 </div>
 Wanna Play Again
 <button onClick={props.reset}> Play Again </button>
 </div>
);
}

const useGameState = () => {
 const [stars, setStars] = useState(utils.random(1, 9));
 const [availableNums, setAvailableNums] = useState(utils.range(1, 9));
 const [candidateNums, setCandidateNums] = useState([]);
 const [countDown, setCountDown] = useState(10);
}
```

```

useEffect(()=>{
 if(countDown > 0 && availableNums.length > 0) {
 const timerId = setTimeout(()=>{
 setCountDown(countDown-1);
 }, 1000);
 return () => {clearTimeout(timerId);}
 }
});

const setGameState = (newCandidateNums) => {
 if (utils.sum(newCandidateNums) !== stars) {
 setCandidateNums(newCandidateNums);
 } else {
 const newAvailableNums = availableNums.filter(
 n => !newCandidateNums.includes(n)
);
 console.log(newAvailableNums);
 setAvailableNums(newAvailableNums);
 setStars(utils.randomSumIn(newAvailableNums, 9));
 setCandidateNums([]);
 }
}

return {stars, availableNums, candidateNums, countDown, setGameState};
}

const Game = (props) => {
 const {stars, availableNums, candidateNums, countDown, setGameState} = useGameState();
 const areCandidatesWrong = utils.sum(candidateNums) > stars;
 const gameStatus = (availableNums.length === 0) ? 'won' : ((countDown === 0) ? 'lost' : 'active');
 console.log('sum::'+utils.sum(candidateNums));
 console.log('stars::'+stars);
 const getNumberStatus = (num) => {
 if(!availableNums.includes(num)) {
 return 'used';
 }
 if(candidateNums.includes(num)) {
 return areCandidatesWrong ? 'wrong' : 'candidate';
 }
 return 'available';
 };
 const getNumberClicked = (number, status) => {
 if (status === 'used' || gameStatus !== 'active') {
 return;
 }
 const newCandidateNums = status === 'available' ? candidateNums.concat(number) :
 candidateNums.filter(cn => cn !== number);
 setGameState(newCandidateNums);
 }

 return (
 <div className="game">
 <div className="help">

```

```

 Pick 1 or more numbers that sum to the number of stars
 </div>
 <div className="body">
 <div className="left">
 {gameStatus !== 'active' ? (<PlayAgain status={gameStatus} reset={props.resetGame}/>) : (<Star
count={stars} />)}
 </div>
 <div className="right">
 {utils.range(1, 9).map(number =>
 <NumberCard key={number} status={getNumberStatus(number)} number={number}
onClick={getNumberClicked} />
)}
 </div>
 </div>
 <div className="timer">Time Remaining: {countDown}</div>
</div>
);
};

```

```

// Color Theme
const colors = {
 available: 'lightgray',
 used: 'lightgreen',
 wrong: 'lightcoral',
 candidate: 'deepskyblue',
};

```

```

// Math science
const utils = {
 // Sum an array
 sum: arr => arr.reduce((acc, curr) => acc + curr, 0),

 // create an array of numbers between min and max (edges included)
 range: (min, max) => Array.from({ length: max - min + 1 }, (_, i) => min + i),

 // pick a random number between min and max (edges included)
 random: (min, max) => min + Math.floor(max * Math.random()),

```

```

// Given an array of numbers and a max...
// Pick a random sum (< max) from the set of all available sums in arr
randomSumIn: (arr, max) => {
 const sets = [[]];
 const sums = [];
 for (let i = 0; i < arr.length; i++) {
 for (let j = 0, len = sets.length; j < len; j++) {
 const candidateSet = sets[j].concat(arr[i]);
 const candidateSum = utils.sum(candidateSet);
 if (candidateSum <= max) {
 sets.push(candidateSet);
 console.log('sets:' + sets);
 sums.push(candidateSum);
 }
 }
 }
}

```

```
 }
 return sums[utils.random(0, sums.length)];
 },
};

ReactDOM.render(<StarMatch />, mountNode);
```