Complex Components

- Combining many individual simple components into one "super" component.
- Usefulness : `Composability`

```
class SrchResult extends React.Component { // extends declares SrchResult
        render() {                                  // as a subclass of
React.Component
                return (                                    // render defines DOM
elements to be                                                  // rendered.
Returns JSX.
                        <div>
                                <ResImage/>
                                <ResCaption/>
                                <ResLink/>
                        </div>
                );
        }
}
```

> `render` has become less popular with the introduction of react hooks like `useState` and `useEffect`.

```
class SrchResult extends React.Component {
     render() {
     return (
             <div>
                        <ResImage { ... this.props}/>
                        <ResCaption { ... this.props}/>
                        <ResLink { ... this.props}/>
               </div>
               );
         }
}
```

> `...` is called the **Spread** operator and its used to pass all the props of the parent class to the child components.

> `props` are immutable. When we need to change it the component needs to request the parent component for a new props object.

> Since the introduction of hooks this format of writing react has become dated. We no longer require to define a class and inherit from parents.

Component State

- Each instance of a react component is a collection of properties that control the properties of the component at a given time in its life cycle.
- States are used to track changes in these properties.

```
class MyClass extends React.Component {
        constructor(props)  {
        super(props);
        this.state = { attribute : "value" };
        }
}
```

- `constructor` runs when an instance of that class is created. It overwrites any other inherited constructors. Hence we *NEED* to call super inside the constructor *IF* we have a constructor.
- `super` calls the constructor of the parent class for proper initialization. All class variables of the parent become available across components.
- The class will run without hitches even without these 2 most of the time as the class calls a default constructor in the background and the super function (This is a function of only React not JS as a whole).
- `setState` is used to update states.

> When `setState` is called it automatically calls `render` to re-render the page.

Stateless Components

- No internal state management.
- Basically functions that take in regular parameters and return JSX.
- `Hooks` can be used to add state management to them.

Component LifeCycle

- Events a Component goes through :
  - Mounting
  - Updating
  - Unmounting
- Order of React Component LifeCycle
  - `constructor`
  - `componentDidMount` on both server and client side.
  - `componentWillUpdate` `
  - `componentWillReceiveProps`

- `shouldComponentUpdate` allows component to exit lifecycle if no update is required.
- `componentWillUpdate`
- `componentDidUpdate`
- `componentWillUnmount`

> `setState` can be performed inside the `componentWillUpdate` step of the lifecycle. React will not trigger a re-render immediately to avoid intermediate stages. Instead for example if you have multiple `setState` then it'll batch them all and re-render once.

Refs and Keys

- `ref` is used to return a reference to a node/element.
- It can be accessed using the `ref` attribute

```
// Method 1
this.refer = React.createRef()
// in element
<div ref={this.refer}/>
// access the reference
const node = this.refer.current
//Method 2: Callback Refs
// in the element
<div ref={this.setInput}/>
// class
this.setInput = element=>{
        this.target = element
}
// then do whatever with this.target
```

- Uses:
  - Bringing elements to focus
  - triggering imperative animations
  - Using 3rd party React DOM libraries.
- `key` is a property to be defined while returning an array of elements in react.
- It is used to identify whether specific elements have been changed or modified.
- It provides a static identity to the array elements.
- Error is generated in case its not specified.
- `map` : data collection that maps key-value pairs.

```
numbers = [1,2,3,4,5]
numbers.map((x)=>
```

```
                    <li key={x}>{x}</li>)
```

Event Handling

- For user interactivity.
- React's event handler : **Synthetic Events**
- Differences between traditional DOM Event handing and React
  - React passes function as event handler while DOM passes it as string.
  - React CamelCases the event name while DOM doesn't.
  - In React default behavior can be disabled only using the `event.preventDefault` or `event.stopPropagation` function whereas in DOM we can just `return false`
- Event objects passed to the handlers are `SyntheticEvent` objects which are a wrapper over the DOMEvent object.
- Event handlers are registered at the time of rendering.

Hooks

- Rules
  - Only defined in the function component
  - Only defined in the top level of the component
  - Can't be a conditional

```
[input,changeInput] = useState("")
```

- `input` is the current state
- `changeInput` is the function to change the current state

```
useEffect(function,dependency)
```

- Infinite re-render if `dependency` is blank
- One time render if `dependency` is `[]`
- Performs side-effects(functions other than rendering)

NodeJS

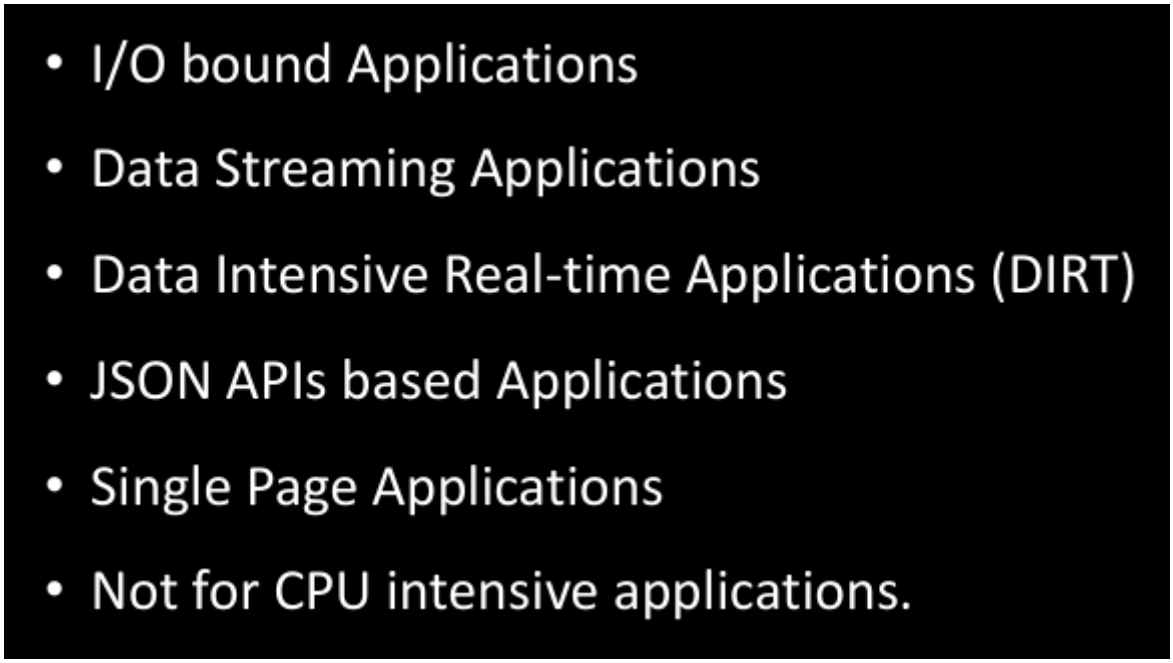
- JS runtime built on the V8 engine. Supports non-blocking(`async`) I/O.

> History on the V8 engine: before this browsers interpreted code line by line which lead to low performance. V8 brought in JIT compilation that translates the code entirely to machine code just before execution which makes it much faster.

- Single Process/ Single Threaded.

- No Buffering
- After completion of a given task Node uses Callbacks to send responses to the server after an `async` process reaches completion.

Multi-Threaded	Asynchronous Event-driven
Lock application / request with listener-workers threads	Only one thread, which repeatedly fetches an event
Using incoming-request model	Using queue and then processes it
Multithreaded server might block the request which might involve multiple events	Manually saves state and then goes on to process the next event
Using context switching	No contention and no context switches
Using multithreading environments where listener and workers threads are used frequently to take an incoming-request lock	Using asynchronous I/O facilities (callbacks, not poll/select or O_NONBLOCK) environments



- `fs` module for file IO

```
// print stats
fs.stat('U4L2.js', (err,stats)=>{
    if (err) throw err;
    console.log('Stats of U4L2.js',JSON.stringify(stats))
})
// rename file
fs.rename('U4L2.js',"NewU4L2.js",(err)=>{
    console.log("Rename Succesfull")
})
// Non-Blocking Read
```

```
const fs=require('fs')
fs.readFile('NewU4L2.js','UTF-8' ,(err,data)=>{
    if(err) throw err
    console.log("Contents:",data)
})
console.log("Reading the Contents");
// Blocking Read
const fs=require('fs')
const data=fs.readFileSync("NewU4L2.js",'UTF-8')
console.log("Reading the file contents...")
console.log("data:",data)
```

- presence of a callback is usually an indication of Non-Blocking nature.

NodeJS Datatypes

- They are:
  - Boolean
  - String
  - Number
  - NULL
  - Undefined
- Comes with REPL(Read.Eval.Print.Loop) for easier debugging.

Node Modules

- Types:
  - Builtin
  - Local
- `require` keyword to include a module
- It returns a JS objects based on what the module returns

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'}); // Http Headers
  res.write(req.url); // after the hostname
  res.end(“Example 1”);
}).listen(8080);
```

```
// url parser
var url = require("url")
var myurl = url.parse(request.url)
var pathname = myurl.pathname; // returns the route
var query = request.query; // returns an object with query items
var host = myurl.host; // returns the hostname
```

```
fs.writeFile(“requestlog.txt”, host + pathname + query, function(err){
});
```

```
var ass = require("assert")
ass(2==2) // outputs nothing
ass(2==3) // prints assertion error
```

```
// user defined modules
var yay = ()=>{
        console.log("printing from my module")
}
module.exports(yay)
```

```
// timer module
// cancelling an immediate object(an operation scheduled in the next
iteration of the event loop)
clearImmediate(timerObject)

// clearing interval object
clearInterval(timerObject)

// cancelling timeout object
clearTimeout(timerObject)

// activate timerObject
timerObject.ref()
//deactivate the timerObject
timerObject.unref()
setImmediate(callback,args) // immediate execution of callback
setInterval(callback,delay,args) // every "delay" milliseconds
setTimeout(callback,delay,args) // after "delay" milliseconds
```

```
// validator functions
isEmail(<data>)
isLowercase(<data>)
isEmpty(<data>)
// chalk functions (adds styles to terminal output)
blue(text)
red(text)
```

- `Nodemon` scans for changes in `src` and auto restarts server.
- Globally accessible Objects :
  - `Buffer` to deal with binary data
  - `console` to print to `stdout` and `stderr`

    - `global` makes a variable globally accessible
- `package.json` -> project metadata,dependencies

Buffers

- Global class to work with raw data(binary)

```
Buffer.allocUnsafe(size)// without initializing memory
Buffer.alloc(10,'a','UTF-8') // slower but never contains old data
// Typeerror if size if not a number
```

```
buf.write(string,offset,length,encoding)
```

- `offset` defaults to 0
- `length` defaults to `Buffer.length`
- `encoding` defaults to `UTF-8`
- Returns the number of bytes written. if length is not enough then it writes part of it

```
buf.toString(encoding,start, end)
```

- `start` default to 0
- `end` default to end of buffer

```
buf.compare(target[, targetStart[, targetEnd[, sourceStart[,
sourceEnd]]]])
```

- `target` to which buffer must be compared
- `targetstart` & `sourceStart` default to 0
- `targetEnd` & `sourceEnd` default to buf.length otherwise index(non inclusive)
- returns integer
    - 0 if both are equal
    - -1 if buf comes before target
    - 1 if buf comes after target

```
buf.copy(target[, targetStart[, sourceStart[, sourceEnd]]])
```

- copied from buffer into target
- `sourceEnd` is non inclusive
- returns number of bytes copied

Streams

- Way to input and output data in small chunks
- Give us power of **Composability**
  - Memory Efficiency
  - TIme Efficiency
- Types of Streams:
  - Writable : `createWriteStream`
  - Readable: `createReadStream`
  - Duplex : `net.Socket`
  - Transform : file compressions
- Streams are `EventEmitter` instances
  - `data` event: when data is available to read
  - `end` : when no more data to read
  - `error` : error in receiving or writing
  - `finish` : when all data is flushed to stream

FileSystem Module

- File IO using POSIX wrappers.

```
fs.open(path, flags[, mode], callback)  // callback params
(err,filedescriptor)
```

- `path` : path to the file
- `flags` : read/write flags
- `mode` : access mode
- `callback` function

```
fs.writeFile(filename, data[, options], callback) // callback params (err)
```

- `options` : encoding,mode,flags

```
fs.read(fd, buffer, offset, length, position, callback) //
(err.BytesRead,buffer)
```

- `fd` : returned by open function

```
fs.unlink() // deleted file
fs.close(fd, callback) // returns file descriptor
fs.truncate(fd, len, callback) // reduces size of file
```

Method	Description
fs.readFile(fileName [,options], callback)	Reads existing file.
fs.writeFile(filename, data[, options], callback)	Writes to the file. If file exists then overwrite the content otherwise creates new file.
fs.open(path, flags[, mode], callback)	Opens file for reading or writing.
fs.rename(oldPath, newPath, callback)	Renames an existing file.
fs.chown(path, uid, gid, callback)	Asynchronous chown.
fs.stat(path, callback)	Returns fs.stat object which includes important file statistics.
fs.link(srcpath, dstpath, callback)	Links file asynchronously.
fs.symlink(destination, path[, type], callback)	Symlink asynchronously.
fs.rmdir(path, callback)	Renames an existing directory.
fs.mkdir(path[, mode], callback)	Creates a new directory.
fs.readdir(path, callback)	Reads the content of the specified directory.
fs.utimes(path, atime, mtime, callback)	Changes the timestamp of the file.
fs.exists(path, callback)	Determines whether the specified file exists or not.
fs.access(path[, mode], callback)	Tests a user's permissions for the specified file.
fs.appendFile(file, data[, options], callback)	Appends new content to the existing file.

HTTP Module

- Web App:
  - Client- frontend. makes request to webserver
  - Server- receives requests. passes response
  - Business- for processing
  - Data- databases
- *See examples in Node Modules*
- FetchAPI
  - include `node-fetch`
  - `fetch(url,options)`
    - text()
    - json()
    - status/statusText
    - ok
    - headers

```
fetch("/data", {
  method: "POST",
  headers: {
    "Content-Type": "application/json",
  },
  body: JSON.stringify({ data: input }),
```

```
    })
      .then((response) => response.json())
      .then((data) => {
        console.log("Server Response: ", data);
        setData(data);
      })
      .catch((error) => {
        console.error("Errr", error);
      });
```