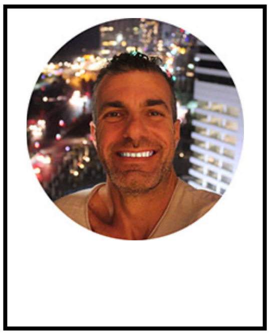
Essential ReactJS Interview Questions and Answers

Table of Contents

About the Author



Elad Elrom (https://elielrom.com/) is a technical coach. As a writer, he authored and co-authored several technical books. Elad consulted for a variety of clients, from large corporations such as AT&T, HBO, Viacom, NBC Universal, and Weight Watchers, to smaller startups. Aside from coding, Elad is also a PADI diving instructor and a pilot.

Introduction

First off, I would like to thank you for purchasing this book and making it a commitment to learn and improve.

React is an open-source JavaScript library for building user interfaces. It is maintained by Facebook & other contributors and rated as the most loved framework for two years in a row by surveys such as StackOverflow, at the time of writing at its 17 iteration.

> *The need for React developers also ballooned; according to Indeed.com, there are close to 10,000 React Developer open positions in the United States.*

Here are a few interesting statistics about React.

- React is a framework released by Facebook. Use case: Facebook, Airbnb, Uber, Netflix & Instagram.
- React developers who stared and liked the project on GitHub ballooned to close to 150k at the time of writing.

Preparing for an interview as the candidate as well as being an interviewer is a stressful time and requires much preparation from the interviewer as well as the interviewee.

In this short book, I curated specific questions that will check knowledge and ensure you pick the right candid. However, beyond that, there are specific concepts that every React must know to avoid getting frustrated writing React code.

In this book, not only I curated specific questions that can help you prepare as well as to conduct an interview for a React developer. Additionally, the book will help you make sure you are well equipped with the most important concepts when it comes to the world of React.

Who is this eBook for?

In this eBook, I am going to share with you questions that you can use to interview React candid as well as use to prepare yourself for an interview.

These questions in this eBook go way beyond just orchestrating an interview or preparing for one.

The information in this eBook can help you become a better developer or ensure your skills are sharpened.

Who is this eBook Not for?

If you are getting started with React, this book is useful to be able to watch and be aware of terms you will encounter during your development journey, but it will be overwhelming as a first React book. I am not giving full explanations here and have referenced some of my articles on Medium to expend on topics. If you are getting started, I recommend you purchase my React book from Apress first:

https://www.amazon.com/React-Libraries-Complete-Guide-Ecosystem/dp/1484266951

How this book is organized?

The questions in this book are classified into two main categories:

- JavaScript / TypeScript Questions
- React Core

When writing React code you can use either JavaScript (JS) or TypeScript (TS). Having a strong understanding in programming is as crucial as understanding React itself.

For that reason, for the first section of the book is dedicated to JS / TS questions.

In this book I compiled 60 questions and answers that you can use to prepare, or get ready to interview a React developer candid.

Let’s get started.

JavaScript / TypeScript Questions

This section can be broken down into six sub-sections;

- String
- Array
- Sorting
- Searching
- Binary Tree
- Linked Lists

When it comes to TypeScript (TS) you don’t need to run your code in your integrated development environment (IDE), you can also use the TS playground — https://www.typescriptlang.org/play.

String

Question:

What is a string?

Answer:

A string — is a sequence of characters, either as a literal / fixed constant or as some kind of variable. In plain English: zero or more characters written inside quotes.
We should use let or const, instead of var.

Why? let and const are block-scoped rather than function-scoped like var.

const vs let vs var

Question:

What is the difference between const, let and var?

Answer:

var variables can be updated and re-declared within its scope; let variables can be updated but not re-declared; const variables cannot be updated or re-declared.
Variable — elements are mutated and length changed, or it may be fixed (after creation).

Erase character from string

Question:

How would you erase a character from a string?

Answer:

```
‘hello world’.slice(1) // ello
world
‘hello world’.replace(‘o’, ‘@’)
// hell@ world
‘hello world’.replace(/o/g, ‘@’)
// hell@ w@rld — regex g=global
```

A String is a palindrome (left to right is equal to the right to left)

Question:

Write code that will perform a palindrome check.

Answer:

```
function palindrome(str: string)
{
 // lower case and clean special
char regex W
 var lowRegStr =
str.toLowerCase().replace(/[\W_]
/g, ‘’)
 var reverseStr =
lowRegStr.split(‘’).reverse().jo
in(‘’)
 return reverseStr === lowRegStr
}
palindrome(“A man, a plan, a
canal. Panama”) // true
```

Convert to strings

Question:

How would you convert a string to a inter, float and number type?

Answer:

```
parseInt('12') // string to
integer
parseFloat('1.2') // string to
float
Number('12') // string to a
number
```

Maximum occurring character in a given string using regex

Question:

Write code that finds out the maximum occurring character in a given string.

Answer:

```
let getMax = function (str:
string) {
  var max = 0,
    maxChar = '';

str.split('').forEach(function(c
har: string){
    if(str.split(char).length >
max) {
      max =
str.split(char).length;
```

```
      maxChar = char;
    }
  });
  return maxChar;
};
getMax('hello world') // return
'l'
```

First repeating character in a string

Question:

Write code to find repeating character in a string.

Answer:

```
function
firstRepeatingCharacter(str:
string) {
  for (let i = 0; i <
str.length; i++) {
    if
(str.indexOf(str.charAt(i)) !==
str.lastIndexOf(str.charAt(i)))
{
      return str.charAt(i)
    }
  }
  return 'no results found'
}
firstRepeatingCharacter('123455'
) // 5
```

To first a non-repeated character of a given string — same as before just remove the ‘!’ sign;

```
function
firstNoneRepeatingCharacter(str:
string) {
  for (let i = 0; i <
str.length; i++) {
    if
(str.indexOf(str.charAt(i)) ===
str.lastIndexOf(str.charAt(i)))
{
      return str.charAt(i)
    }
  }
  return 'no results found'
}
firstRepeatingCharacter('123455'
) // 1
```

Split a string

Question:

Write code to find all the words in a sentence.

Answer:

```
(‘hello world’).split(‘ ‘) //
['hello', 'world']
```

Find duplicate characters in a string

Question:

Write code to find duplicate characters in a string

Answer:

```
const text = 'abcda'.split('')
text.some( (v, i, a) => {
  return a.lastIndexOf(v) !== i
}) // true
```

Check duplicate characters in a string isogram

Question:

Write code to check duplicate characters in a string

Answer:

isogram is a word with no repeating letters

```
'test'.split('').some((value,ind
ex, array) => {
  return
array.lastIndexOf(value) !==
index;
}) // true
```

Array

Question:

What is an array?

Answer:

An array is a special variable, which can hold more than one value at a time.

Find missing value in an integer array

Question:

Write code to find missing value in an integer array.

Answer:

```
const a = [3]
const count = 5
let missing = []

for (let i = 1; i <= count; i++)
{
  if (a.indexOf(i) === -1) {
    missing.push(i)
  }
}
console.log(missing) // [1, 2,
4, 5]
```

The largest and smallest number in an array

Question:

Write code to find the largest and smallest number in an array

Answer:

```
Math.max(...[1, 3, 10])
Math.min(...[1, 3, 10])
```

If it’s a couple of numbers and not an array, remember;

```
Math.max(1, 3, 10)
Math.min(10, 5, 15)
```

Missing number in an array

Question:

Write code to find a missing number in an array.

Answer:

```
let arr = [1, 2, 3, 5, 8]
let [min,max] = [Math.min(…arr),
Math.max(…arr)]
let out = Array.from(Array(max-
min),(v,i)=>i+min).filter(i=>!ar
r.includes(i))console.log(out)
// [4, 6, 7]
```

Reverse an array

Question:

Write code to reverse an array.

Answer:

```
[1, 2, 3].reverse() // [3, 2, 1]
```

Duplicate numbers in an array

Question:

Write code to find duplicate numbers in an array.

Answer:

```
let numArray: number[] = [1, 1,
2, 2,
3]numArray.filter((element,
index, array) =>
array.indexOf(element) !==
index) // [1, 2]
```

Remove duplicates from an array

Question:

Write code to remove duplicates numbers from an array.

Answer:

```
[...new Set([1, 1, 2, 2, 3])] //
[1, 2, 3]
```

Convert a byte array into a string

Question:

Write code to convert a byte array into a string.

Answer:

```
String.fromCharCode.apply(null,
[102, 111, 111]) // "foo"
```

Sorting

Question:

Part I: What is a sorting algorithm? Part II: Give me examples of stable and unstable sorting.

Answer:

A Sorting Algorithm is used to arranging the data of the list or array into some specific order.

When it comes to sorting there are many options. It is safe to categorize them into two categories stable and unstable.

Stable sorting algorithms preserve the relative order of equal elements, while unstable sorting algorithms don't. In other words, stable sorting maintains the position of two equals elements relative to one another.

Common stable sorting algorithms:

- Bubble Sort
- Merge Sort
- Timsort
- Counting Sort
- Insertion Sort

Examples of unstable sorts are

- Quicksort
- Heapsort
- Selection

There is no clear answer to which sorting algorithm you should be using it depends on the assumption you make about the data and many variables.

Some sorting algorithms come in both flavors. For example, binary sort algorithm.

Array built-in sort

Question:

Write code to sort through an array.

Answer:

```
const array = ['Banana',
'Orange', 'Apple', 'Mango']
array.sort() // ascending -
["Apple", "Banana", "Mango",
"Orange"]
array.sort().reverse() //
descending
```

This works on number array as well;

```
const numArray = [3, 5, 2, 1]
numArray.sort() // [1, 2, 3, 5]
```

Bubble sort

Question:

Write code to do a bubble sort.

Answer:

Bubble sort — For each item in the array we want to check if the next item is larger, if it is then swapping their indexes in the array

```
function bubbleSort(array:
number[]) {
  const len = array.length;
  const retArray = array;
  for (let i = 0; i < len; i++)
{
    for (let j = 0; j < len - i;
j++) {
      const a = array[j];
      if (a !== array[-1]) {
        const b = array[j + 1];
        if (a > b) {
          retArray[j] = b;
          retArray[j + 1] = a;
        }
      }
    }
  }
  return retArray;
}
bubbleSort([10, 9, 8, 7, 6, 5,
4, 3, 2, 1]) // [1, 2, 3, 4, 5,
6, 7, 8, 9, 10]
```

Insertion sort

Question:

Write code to do insertion sort.

Answer:

Insertion sort is the sorting mechanism where the sorted array is built having one item at a time. The array elements are compared with each other sequentially and then arranged simultaneously in some particular order. The analogy can be understood from the style we arrange a deck of cards.

```
const insertionSort = (array:
number[]) => {
  const { length } = array;
  const retArray = array;
  for (let i = 1; i < length;
i++) {
    const key = array[i];
    let j = i - 1;
    while (j >= 0 && array[j] >
key) {
      retArray[j + 1] =
array[j];
      j -= 1;
    }
    retArray[j + 1] = key;
  }
  return retArray;
};
insertionSort([10, 9, 8, 7, 6,
5, 4, 3, 2, 1]) // [1, 2, 3, 4,
5, 6, 7, 8, 9, 10]
```

Merge sort algorithm

Question:

Write code to do a merge sort.

Answer:

Merge sort is one of the most efficient sorting algorithms. It works on the principle of Divide and Conquers. Merge sort repeatedly breaks down a list into several sub-lists until each sub-list consists of a single element and merging those sub-lists in a manner that results in a sorted list.

1. The Merge sort algorithm repeatedly divides the array into smaller chunks until we no longer divide the array into chunks.
2. Then it repeatedly merges the chunks of the array to get a sorted array.

```
function merge(left: number[],
right: number[]) {
  const resultArray: number[] =
[]
  let leftIndex = 0
  let rightIndex = 0

  while (leftIndex < left.length
&& rightIndex < right.length) {
    if (left[leftIndex] <
right[rightIndex]) {

resultArray.push(left[leftIndex]
)
      leftIndex+=1
    } else {

resultArray.push(right[rightInde
x])
      rightIndex+=1
    }
  }
  return resultArray
.concat(left.slice(leftIndex))

.concat(right.slice(rightIndex))
}
function
mergeSort(unsortedArray:
number[]): number[] {
  if (unsortedArray.length <= 1)
{
    return unsortedArray;
  }
  const middle =
Math.floor(unsortedArray.length
/ 2);
  const left =
unsortedArray.slice(0, middle);
  const right =
unsortedArray.slice(middle);
  return merge(
    mergeSort(left),
mergeSort(right)
  )
}
mergeSort([10, 9, 8, 7, 6, 5, 4,
3, 2, 1]) // [1, 2, 3, 4, 5, 6,
7, 8, 9, 10]
```

Iterative quicksort algorithm

Question:

Write code to do an iterative quicksort.

Answer:

I left the hardest for last. The 'quickSort' function selects a pivot element using the partition function, then recursively sorts the left side of the array (low to pivot-1) and the right side of the array (pivot+1 to high), returning the full sorted array once it’s done. The partition function is where most of the work is done.

```
const swap = (arr: number[], i:
number, j: number) => {
  const tmp = arr[i]
  const retArr = arr
  retArr[i] = arr[j]
  retArr[j] = tmp
  return retArr
};
const partition = (arr:
number[], low: number, high:
number) => {
  let q = low; let i;
  for (i = low; i < high; i++) {
    if (arr[i] < arr[high]) {
      swap(arr, i, q)
      q += 1
    }
  }
  swap(arr, i, q)
  return q
};
const quickSort = (arr:
number[], low: number, high:
number) => {
  if (low < high) {
    const pivot = partition(arr,
low, high)
    quickSort(arr, low, pivot -
1)
    quickSort(arr, pivot + 1,
high)
    return arr
  }
  return []
}
```

```
quickSort([9, 8, 7, 6, 5, 4, 3,
2, 1], 4, 9) // [9, 8, 7, 6,
undefined, 1, 2, 3, 4, 5]
```

Searching

Question:

Write code to do a searching sort.

Answer:

Searching Algorithms are designed to retrieve an element from any data structure where it is used. Just like sorting the search can be stable or unstable.

Term search

Question:
Write code to do a term search.

Answer:

Searching Algorithms are designed to retrieve an element from any data structure where it is used.

```
'Hello world'.search('lo') //
return where found 3
```

Binary search algorithm

Question:

Write code to a binary search.

Answer:

Binary Search (AKA half-interval search) is used to find a specific element located in an array this only works with sorted arrays. It comes in two flavors stable and unstable.
Here is an example of an unstable binary search to find the location of an element in the array;

```
function
binarySearchIndex(array:
number[], target: number, low =
0, high = array.length - 1):
number {
  if (low > high) {
    return -1
  }
  const midPoint =
Math.floor((low + high) / 2)
  if (target < array[midPoint])
{
    return
binarySearchIndex(array, target,
low, midPoint - 1)
  } if (target >
array[midPoint]) {
    return
binarySearchIndex(array, target,
midPoint + 1, high)
  }
  return midPoint
}
binarySearchIndex([1, 2, 3, 4,
5, 6, 7, 8, 9, 10], 9) // 8
```

Note: An unstable binary search would return the first one found, which may not be the chronological first. Stable binary searches only require one relational comparison of a given pair of data elements per iteration, where unstable binary searches require two comparisons per iteration.

Binary Trees

Question:

Explain what is a binary tree?

Answer:

What is a Binary Tree and why should you care about it?

A binary tree is made of nodes, where each node contains a “left” reference, a “right” reference, and a data element. Take a look;

```
         1
       /   \
      2     3
     / \   / \
    4   5 6   7
```

- **root** — the topmost node in the tree is called the root
- **Leaves (external nodes)** — Nodes with no children are called: leaves, or external nodes.
- **Internal nodes** — nodes that are not leaves are called internal nodes.
- **Sibling/grandchild** — Nodes with the same parent are called siblings and children of these are grandchild.

- **Degree** — the number of children a node has is a degree.
- **Level d** — All the nodes with the same depth d are in a set called level d.
- **A** binary tree — is a tree where the maximum degree of any internal node is 2. Thus, a node may have 0, 1, or 2 children.

A tree can be evaluated using a bottom-up manner (i.e., from leaf nodes to the root) top-down manner (root to leaf).

```
Level 0                 1
                      / | \
Level 1              2  3  4
                          / \
Level 2                  5   6
```

Tree nodes are for storing some kind of keyed data.

Typically, this data is a record, like a student record with key=SSN, user Id, etc, but we can simply consider trees where the data is a key, as an integer, but they can be of any type. Because of that TypeScript makes perfect sense and has an advantage since TS has generics.

TypeScript generics, if you are starting with TS read this article. Also, if you are a bit rusty on TS check out my TS Programming Reference.

Generics

Additionally, TS provides Generics declarations.

You can declare types with special characters that will be declared at run-time instead of compile time. For example, if my function takes one argument and returns a value. If I know that the type is a string, it would look like this;

```
function myFunction<T>(arg:
string): string {
 return arg
}
```

If we want to build a flexible software, we can leave it abstract and it will be declared on run-time, that way it can be anything but still, get checked instead of using ‘any’ or ‘never’;

```
function myFunction<T>(arg: T):
T {
 return arg
}
```

The T stands for Type in this example. What if we need to pass two arguments? not a problem

```
function myFunction<T>(arg: T,
arg2: U): T {
 return arg
}
```

As you see I am depreciating the types of the arguments, and we can use any letter (or combination of valid alphanumeric names), so you will often see <U>, <G>, or whatever letter you wish — no significance to letter, other than conventional purposes.

Normally we would key off a type of string of a number, but in case we would like to later support other types why not built that into our code from the get-go?!

Binary search tree

Question:

What is a binary search tree?

Answer:

Binary search tree — is a tree where the maximum degree of any internal node is 2. Thus, a node may have 0, 1, or 2 children.

```
Level 0                      1
                            / \
Level 1                    2   3
                          /   /
\
Level 2                  4   5
6

\
Level 3
7
```

A complete binary tree

Question:

What is a complete binary tree?

Answer:

- Every internal node has a degree of exactly two, and
- Every leaf node occurs at the same level.

```
          1
        /   \
       2     3
      / \   / \
     4   5 6   7
```

Complete binary trees are important because they allow us to explore certain aspects of binary trees in a simple context.

For example, we can compute how many leaf nodes are there in a complete binary tree of height h?

For a complete binary tree, we saw above. That would be the following formula.

height = 2, nodes = $2^{3}-1 = 7$

A detailed explanation can be found in this school article: https://www.cs.utexas.edu/users/djimenez/utsa/cs1723/lecture8.html

Question:

When should you use a binary tree?

Answer:

The major advantage of binary search trees over other data structures is that the related sorting algorithms and search algorithms such as in-order traversal can be very efficient which fits perfectly when you need to compute things for animations and graphics.

Finding the depth of a binary tree

<u>Question:</u>

Write code to find the depth of a binary tree.

<u>Answer:</u>

- The depth of a node x is the number of nodes between the root and x.
- The root is said to be at depth 0;
- any children of the root are at depth 1, etc.
- The height of a tree is the maximum depth of any node.
- For convenience, the height of an empty tree is defined as -1.

All the nodes with the same depth d are in a set called level d.

Let’s create a Binary Search Tree in TypeScript;

```
export default class
BinaryTree<U> {

  private root: INode<U> |
undefined
```

```
  createNewNode = (value: U):
INode<U> => {
    return {
      value,
      left: null,
      right: null,
    }
  }

  insert = (value: U) => {
    const currentNode =
this.createNewNode(value)
    if (!this.root) {
      this.root = currentNode
    } else {

this.insertIntoCurrentNode(curre
ntNode)
    }
    return this
  }

  private insertIntoCurrentNode
= (currentNode: INode<U>) => {
    const { value } =
currentNode
    const traverse = (node:
INode<U>) => {
      if (value > node.value) {
        if (!node.right) {
          node.right =
currentNode
        } else
traverse(node.right)
      } else if (value <
node.value) {
        if (!node.left) {
          node.left =
currentNode
        } else
traverse(node.left)
      }
    }
    traverse(this.root as
INode<U>)
  }

interface INode<U> {
  value: U
  left: INode<U> | null
  right: INode<U> | null
}
```

Notice that I am using TS generics but, in my interface, and for the value. That will allow me to be ready for any type of data to come my way.

The method used by the traverse method is recursive (a function that calls itself) and that allows me to avoid loops.

Let’s implement;

```
const binaryTree = new
BinaryTree()binaryTree.insert(10
).insert(20).insert(30).insert(5
).insert(8).insert(3).insert(9)
```

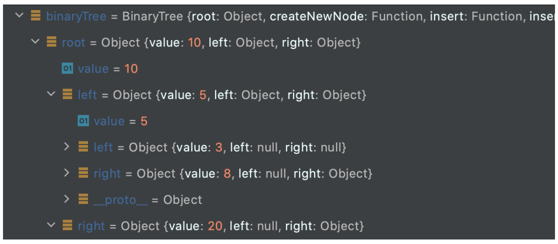
What’s happening, on every add method called, the node is added starting parent, then children's — right side first then left child.

```
              #1
             /   \
           #4   #2
          / \     \
          #6  #5     #3
                \
                 #7
```

Now let’s plug in the actual values;

```
     10
   /    \
  5      20
 / \       \
3   8       30
     \
      9
```

To confirm, place a line breakpoint in your IDE, to watch the results take a look;



```
⌄ binaryTree = BinaryTree {root: Object, createNewNode: Function, insert: Function, inse
  ⌄ root = Object {value: 10, left: Object, right: Object}
      value = 10
    ⌄ left = Object {value: 5, left: Object, right: Object}
        value = 5
      › left = Object {value: 3, left: null, right: null}
      › right = Object {value: 8, left: null, right: Object}
      › __proto__ = Object
    ⌄ right = Object {value: 20, left: null, right: Object}
```

Figure 1: Binary tree output.

Binary Search tree — contained, min, max

Question:

Write code to do a binary tree search to find if a node is contained as well as the min and max node value.

Answer:

Next, we need to implement some basic methods that we would need to check if a value is contained as well as find the min and max of values inserted. Take a look;

```
//
src/utils/BinaryTree.tsxcontains
= (value: U) => {
  // eslint-disable-next-line
consistent-return
  const traverse = (node:
INode<U>): undefined | boolean
=> {
    if (!node) return false
    if (value === node.value) {
      return true
    }
    if (value > node.value) {
      return traverse(node.right
as INode<U>)
    }
    if (value < node.value) {
      return traverse(node.left
as INode<U>)
    }
  }
  const rootNode: INode<U> |
undefined = this.root
  return traverse(rootNode as
INode<U>)
}

// find the leftmost node to
find the min value of a binary
tree
findMin = () => {
  const traverse = (node:
INode<U>): INode<U> | U => {
    return !node.left ?
node.value : traverse(node.left)
```

```
  }
  const rootNode: INode<U> =
this.root as INode<U>
  return traverse(rootNode)
}

// find the rightmost node to
find the max value of a binary
tree
findMax = () => {
  const traverse = (node:
INode<U>): INode<U> | U => {
    return !node.right ?
node.value :
traverse(node.right)
  }
  const rootNode: INode<U> =
this.root as INode<U>
  return traverse(rootNode)
}
```

To Implement;

```
console.log(binaryTree.contains(
30)) // true
console.log(binaryTree.findMin()
) // 3
console.log(binaryTree.findMax()
) // 30
```

‘preorder’, ‘inorder’, and ‘postorder’ traversals searches

It’s important to know leaf Nodes, so we can calculate the total nodes and do our searches.

Visiting all of the nodes in a binary search tree is called doing a tree traversal.

- Visit everything in the left subtree.
- Visit the data or root node.
- Visit everything in the right subtree.

Their main searches are ‘preorder’, ‘inorder’, and ‘postorder’. Let’s take a look.

Binary Search – Preorder

Question:

Write code to do a preorder binary search.

Answer:

preOrder is a type of depth-first traversal that tries to go deeper in the tree before exploring siblings. It returns the shallowest descendants first.
1) Display the data part of the root element (or current element)
2) Traverse the left subtree by recursively calling the pre-order function.
3) Traverse the right subtree by recursively calling the pre-order function.

```
//
src/utils/BinaryTree.tsxpreOrder
= () => {
  let result: U[]
  // eslint-disable-next-line
prefer-const
  result = []
  const traverse = (node:
INode<U>) => {
    result.push(node.value)
    node.left &&
traverse(node.left)
```

```
    node.right &&
traverse(node.right)
   }
  const rootNode: INode<U> |
undefined = this.root as
INode<U>
  traverse(rootNode)
  return result
}
```

Inorder

Question:

Write code to do an Inorder binary search.

Answer:

If we search everything on the left first, the current data or root, then everything on the right. This is called doing an inorder traversal.

inOrder traversal is a type of depth-first traversal that also tries to go deeper in the tree before exploring siblings. however, it returns the deepest Descendents first

1) Traverse the left subtree by recursively calling the pre-order function.
2) Display the data part of the root element (or current element)
3) Traverse the right subtree by recursively calling the pre-order function.

```
//
src/utils/BinaryTree.tsxinOrder
```

```
= () => {
  let result: U[]
  // eslint-disable-next-line
prefer-const
  result = []
  const traverse = (node:
INode<U>) => {
    node.left &&
traverse(node.left)
    result.push(node.value)
    node.right &&
traverse(node.right)
  }
  const rootNode: INode<U> |
undefined = this.root as
INode<U>
  traverse(rootNode)
  return result
}
```

Postorder

Question:

Write code to do a Postorder binary search.

Answer:

postOrder traversal is a type of depth-first traversal that also tries to go deeper in the tree before exploring siblings. however, it returns the deepest Descendents first;

1) Traverse the left subtree by recursively calling the pre-order function.
2) Display the data part of the root element (or current element)

3) Traverse the right subtree by recursively calling the pre-order function.

```
//
src/utils/BinaryTree.tsxpostOrde
r = () => {
  let result: U[]
  // eslint-disable-next-line
prefer-const
  result = []
  const traverse = (node:
INode<U>) => {
    node.left &&
traverse(node.left)
    node.right &&
traverse(node.right)
    result.push(node.value)
  }
  const rootNode: INode<U> |
undefined = this.root as
INode<U>
  traverse(rootNode)
  return result
}
```

Let’s give it a try, shall we;

```
// [ 10, 5, 3, 8, 9, 20, 30 ]
console.log('preorder',
binaryTree.preOrder()) // [ 3,
5, 8, 9, 10, 20, 30 ]
console.log('inorder',
binaryTree.inOrder()) // [ 3, 9,
8, 5, 30, 20, 10]
console.log('postorder',
binaryTree.postOrder())
```

LinkedList

Question:

Part I: What is a LinkedList and when should you use it? Part II: write a LinkedList code using TS.

Answer:

So far, we have dealt with binaries trees that we have the maximum degree. If we don't know the maximum degree, then we can use a linked list of siblings, each of which has a link to a linked list of children.

When should you use LinkedList programming?

The primary benefit of linked lists is that they can contain an arbitrary number of values while using only the amount of memory necessary for those values.

Linked lists are linear data structures that hold data in individual objects called nodes. These nodes hold both the data and a reference to the next node in the list.

Linked lists are often used because of their efficient insertion and deletion.

Good use case when time predictability is absolutely critical, you don't know how many items will be on the list. A good use case would a Blockchain.

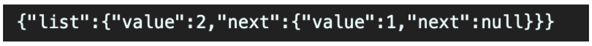
Two main ways to implement: Iterative or Recursive. Let’s create a recursive list;

```
export class Node<U> {

  private readonly value: U

  next: Node<U> | null |
undefined

  constructor(value: U) {
    this.value = value
    this.next = null
  }

  getValue(): U {
    return this.value
  }
}

export class LinkedList<U> {

  private list: Node<U> | null

  constructor()
  constructor(link: Node<U>)
  constructor(link: Node<U>,
list?: LinkedList<U>)
  constructor(link?: Node<U>,
list?: LinkedList<U>) {
    if (!link) {
      this.list = null
    } else {
      this.list = link
      this.list.next =
list?.list
    }
  }

  insert(value: U): void {
    const currentList =
this.list
    if (!currentList) {
      this.list = new
Node<U>(value)
    } else {
      let newNode: Node<U>
      newNode = new
Node<U>(value)
      newNode.next = currentList
      this.list = newNode
    }
  }
}
```

Once again, I am using TypeScript generics as I have done with the binary tree so we can pass any type.

Let's take it for a test drive;

```
const list = new LinkedList()
list.insert(1)
list.insert(2)
console.log(JSON.stringify(list)
)
```



```
{"list":{"value":2,"next":{"value":1,"next":null}}}
```

Figure 2: LinkList results

Convert Binary Tree to Linked List

<u>Question:</u>

Write code to convert Binary Tree to LinkedList.

Answer:

Now to have our binary tree and the linked list we can convert the Binary Tree into a Linked list.

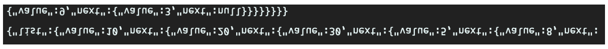
That is really simple since we have the searches implemented.
For example, if we want to display a linear list of the data in the tree, all we need to do is call inOrder method and use JSON.stringify;

```
//
src/utils/BinaryTree.tsxtoString
= () => {
  return JSON.stringify(
this.inOrder() )
}
```

Since we get an array back, we can just iterate through the array and insert the results into our linked list, take a look;

```
//
src/utils/BinaryTree.tsximport {
LinkedList } from
'./LinkedList'convertToLinkedLis
t = (orderType: 'inOrder' |
'postOrder') => {
  const list = new LinkedList()
  const ordered = orderType ===
'inOrder' ? this.inOrder() :
this.postOrder()
  ordered.forEach(value => {
    list.insert(value)
  })
  return list
}
```

Notice that I am setting the method with order type: ‘inOrder’ | ‘postOrder’.
Here are the results;



```
{"list":{"value":10,"next":{"value":20,"next":{"value":30,"next":{"value":5,"next":{"value":8,"next":
{"value":9,"next":{"value":3,"next":null}}}}}}}}
```

Figure 3: Convert Binary Tree to Linked List results

Let’s Recap

In this section I covered my top used;

- String
- Array
- Sorting
- Searching Expressions.

I shared code to handle; Palindrome, isogram, stable and unstable sorting algorithm, half-interval search, bubble sort, insertion sort, merge sort algorithm, quicksort algorithm, binary search algorithm, and more.

The complete code can be found here: https://gist.github.com/EliEladElrom/450ce67da8f7a9322b87dee5a269b212#file-typescript-programming-reference-tsx

Knowing how to work with Binary Trees and Linked Lists are great tools to put in your programming arsenal and can come in handy when working with complex data. TypeScript is great because checking the types and using generics adds an extra layer to ensure the data integrity as well as create tests.

I kept the code simple, however, there may be a need in the project to implement methods such as;

- Reversed order
- Remove Node elements
- Merge two Binary Trees / Linked Lists

Download the source code: https://github.com/EliEladElrom/react-tutorials

React Core

What I will be covering in this article?
In this article, part I of II, I will be covering the following:

- React vs Angular
- React advantages
- React limitations
- DOM vs VDOM.
- JSX
- ES6 vs ES5
- Toolchain
- Automate React App

- Writing comments in JSX
- Everything is a component
- Functional and class components
- Embed multiple child components into the parent
- Function Component with Hooks
- Component StrictMode
- States vs props
- Arrow functions
- Avoid binding
- Component lifecycle
- Action once on init class vs function component
- Update state class vs function component
- Higher-Order Components (HOCs)
- Modularize with lazy load
- Mixins
- Keys
- Refs
- Avoid Prop drilling

- Testing Shallow vs Mount vs Render
- Styling a React application
- Stateful vs stateless components
- Controlled vs uncontrolled components
- Event & synthetic events in React
- Forms
- Optimize slow App
- React vs React Native

React vs Angular

Question:

What's the different between React and Angular, which one is better?

Answer:

React enables developers to render a user interface. To create a full front-end application, developers need other pieces and libraries such as state management.

Angular is a full mature framework and fits both small and large team and can be scaled easily, however, the learning curve is higher, Angular using more resources, slower and you are more limited in the number of developers that will be available at your disposal and that will be eager to learn and migrate to Angular.

React library, is lighter in terms of resource usage, faster, and is more favorite among developers to be used and to be learned. React is easier to learn but not as easy to master as one would think; with all the good, keep in mind, if your team is not following a strict best practices protocol, when it time to scale, you may find yourself with an app that holds a technology debt that it won’t be easy to overcome or bugs that will hunt you.

There is no better or worse here, it depends on the individual developers working on the project, the size of the project, scope of the project, countries it needs to be deployed at, the experience of the team, and the timeline needed to complete. It’s more about deciding whether to use a library or framework. If you ask a React or Angular fanboys clubs, they will argue both ways claiming their side is the winner.

However, using Framework is more suitable for larger teams and usually suitable for projects with more budget. With that being said, React can be set with tools and libraries that will make it fit a larger enterprise-level project, and the proof is the companies using React today.

Planning a project is everything. As wisely said, “measure twice cut once”.

I recommend to pay close attention to all the variables and moving parts of your project before jumping into a sprint and marrying one technology stack over the other
React advantages

Question:

Name some of React’s main advantages?

Answer:

- The usage of the VDOM to expedite the real DOM.
- React can be used as a single-page-application (SPA ie CRA) or server-side rendering (SSR ie Gatsby.js, Next.js)
- It can follow uni-directional data flow or data binding.
- Increases application’s performance
- Due to JSX, better code’s readability.
- React is easy and made to integrate with other frameworks.

- Writing UI test cases are easy to implement since in React everything is a component.

Question:

Name some of React’s main limitations?

Answer:

React limitations

- On its own React is just a UI library, not a full-blown framework like Angular.
- The library is small in size but large in features and takes time to understand.
- It’s harder for novice programmers to grasp
- Coding can get complex as it uses inline templating and JSX vs pure JS.

Read my Medium article (https://medium.com/react-courses/angular-9-vs-react-16-a-2020-showdown-2b0b8aa6c8e9) to find out more.

DOM vs VDOM

Question:

What is the difference between the DOM and VDOM?

Answer:

Before I start talking about the Virtual DOM (VDOM), you need to have a great grasp of the Document Object Model (DOM).

DOM is the API for HTML and even XML documents such as SVG images. API describes the document by including interfaces that define the functionality of the HTML's elements as well as any interfaces and types they rely on.

The HTML document includes support and access to various features such as interacting with users via event handlers, focus, copy & paste, etc.

The DOM document consists of a hierarchical tree of nodes (https://developer.mozilla.org/en-US/docs/Web/API/Node) and the node interface allows access, not just to the document but each element (node).

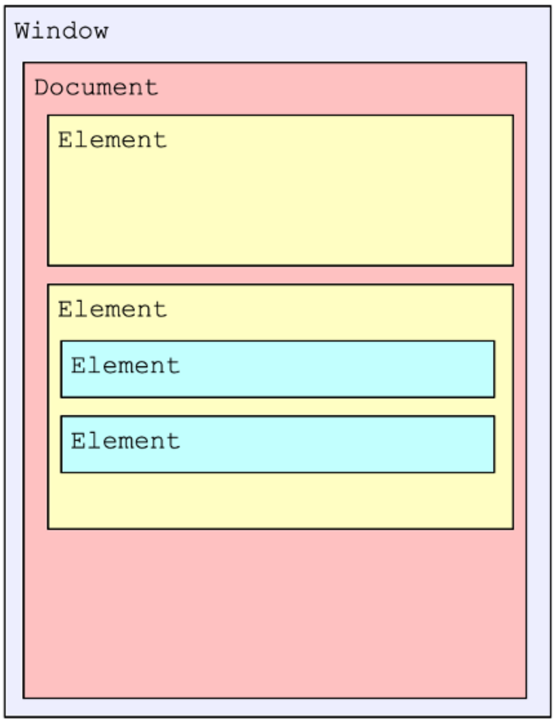
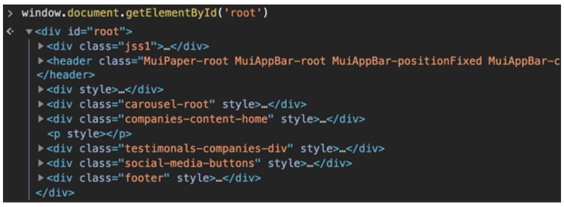


Figure 4: DOM document hierarchical tree.

Most developers have spent some time debugging using browser tools such as the Chrome inspector. However, I encourage you to learn it better as well as try some of Chrome DevTools Extensions specific for React (check my article here).

For example, did you know that if you type this in the console of a CRA; window.document.getElementById('root'), you will get the App Root document;



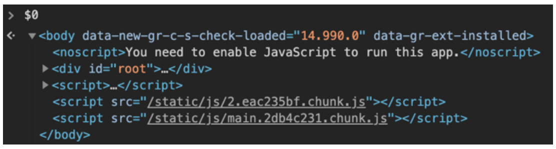
```
> window.document.getElementById('root')
<· ▾<div id="root">
    ▸<div class="jss1">…</div>
    ▸<header class="MuiPaper-root MuiAppBar-root MuiAppBar-positionFixed MuiAppBar-c
    </header>
    ▸<div style>…</div>
    ▸<div class="carousel-root" style>…</div>
    ▸<div class="companies-content-home" style>…</div>
     <p style></p>
    ▸<div class="testimonals-companies-div" style>…</div>
    ▸<div class="social-media-buttons" style>…</div>
    ▸<div class="footer" style>…</div>
   </div>
```

Figure 5: App Root document.

The reason for that is that CRA sets the root as the id of the root element;
// index.html<div id="root"></div>

And that's matches what's happing in our code too;

// index.tsxconst rootElement = document.getElementById('root')
On any website out there typing $0 gives the whole body of the document;



```
> $0
<· ▾<body data-new-gr-c-s-check-loaded="14.990.0" data-gr-ext-installed>
     <noscript>You need to enable JavaScript to run this app.</noscript>
    ▸<div id="root">…</div>
    ▸<script>…</script>
     <script src="/static/js/2.eac235bf.chunk.js"></script>
     <script src="/static/js/main.2db4c231.chunk.js"></script>
   </body>
```

Figure 6: the body of the document.

Here are some resources that can help you sharpen your DOM sword;

- https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction

- https://developers.google.com/web/tools/chrome-devtools/dom
- https://reactjs.org/docs/faq-internals.html

Now the VDOM is a programming concept where an ideal, or “virtual”, representation of a UI is kept in the user’s memory and synced with the “real” DOM by ReactDOM library. The React VDOM aims to speed up things.

React hold a copy of the HTML DOM (that’s the virtual DOM), once a change is needed, React first make the change to the Virtual DOM, then it syncs the actual HTML DOM in a process called “Reconciliation” and avoiding the need to update the entire HTML DOM speeding up the rendering and updating process.

JSX

Question:

What is JSX?

Answer:

JavaScript XML (JSX) is a type of file used by React, that utilizes the expressiveness of JavaScript along with HTML like template syntax. This makes the HTML file really easy to understand.

Browsers can only read JS objects, but JSX is not a regular JS object. Thus to enable a browser to read JSX, first, we need to transform the JSX file into a JS object using JSX transformers like Babel and then pass it to the browser.

In fact, React become so popular that starting from Babel 8, React gets a special function, you guessed it... jsx to replace the render. Compiling JSX to these new functions. It also automatically imports "react" (or other libraries that support the new API) when needed, so you don't have to manually include them anymore if you write to babel directly.

```
This input;function Foo() {
  return <div />;
}
```

Will turn the JSX code into this;

```
import { jsx as _jsx } from
"react/jsx-runtime";
function Foo() {
  return _jsx("div", ...);
}
```

ES6 vs ES5

Question:

What is the difference between ES6 and ES5?

Answer:

Babel is the library that transpiling ES6 to ES5.

The JSX code we write is just a terser way to write the React.createElement() function declaration. Every time a component uses the render function, it outputs a tree of React elements or the virtual representation of the HTML DOM elements the component outputs.

ES5 (ES — ECMAScript version 5) is plain old “regular JavaScript.” finalized in 2009. It is supported by all major browsers. ES6 is the next version, it was released in 2015, and added syntactical and functional additions. ES6 is almost fully supported by all major browsers at the time of writing. In fact, the React team in version 17 made many changes to be more consistent and compatible with ES6.

We want to take advantage of ES6 functionality, however, at the same time, we want to be backward compatible with the old ES5 so we can be compatible in all versions of browsers. To do that we use Babel.

Babel is the library that transpiling ES6 to ES5 (that’s needed for Browsers that don’t support ES6). The ReactDOM.render(), as the name suggests, renders the DOM. The render function is expected to return a virtual DOM (representation browser DOM elements).

ES5 and ES6 Syntax is different. We want to take advantage of ES6 without waiting for a 100% adaptation. Take a look at some examples;

```
// ES5
var React = require('react');//
ES6
import React from 'react';// ES5
module.exports = Component;//
ES6
export default Component;
component and function// ES5
var MyComponent =
React.createClass({
  render: function() {
    return    <h3>Hello
World</h3>
    ;
  }
});// ES6
class MyComponent extends
React.Component {
  render() {
    return    <h3>Hello
World</h3>
    ;
  }
}// ES5
var App = React.createClass({
  propTypes: { name:
React.PropTypes.string },
  render: function() {
    return    <h3>Hello,
{this.props.name}!</h3>
    ;
  }
});// ES6
```

```
class App extends
React.Component {
  render() {
    return    <h3>Hello,
{this.props.name}!</h3>
    ;
  }
}// ES5
var App = React.createClass({
  getInitialState: function() {
    return { name: 'world' };
  },
  render: function() {
    return    <h3>Hello,
{this.state.name}!</h3>
    ;
  }
});// ES6
class App extends
React.Component {
  constructor() {
    super();
    this.state = { name: 'world'
};
  }
  render() {
    return    <h3>Hello,
{this.state.name}!</h3>
    ;
  }
}
```

Toolchain

Question:

What are the primary pillars in the JavaScript toolchain?

Answer:

The bleeding edge JavaScript toolchain can seem quite complex, and it's very important to feel confident in the toolchain and to have a mental picture of how the pieces fit together.

There are a couple of primary pillars in the JavaScript toolchain:

- File events watcher — this is the heavy lifting and most important development task. Build tools such as WebPack, Watchify/Browserify, Broccoli, or Gulp watch for file events such as add, or edit files. After this occurs, the build tool is configured to carry out a group of sequential or parallel tasks.

- Dependency Management — for JavaScript projects, most people use other packages from npm; some plugins exist for build systems (e.g. Webpack) and compilers (e.g. Babel) that allow automatic installation of packages being imported or required.

- Static Analysis Tools — Style-checking — a linter like ESlint is used to ensure the source code is following a certain structure and style. Linting tools like ESLint can be used with plugins such as eslint-plugin-jsx-a11y to analyze React projects at a component level.

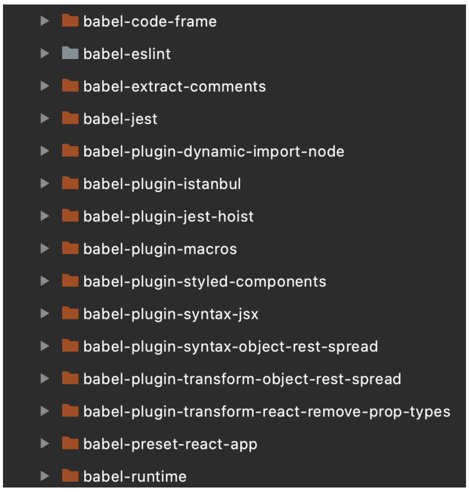
- Transpilation — a specific sub-genre of compilation, transpilation involves compiling code from one source version to another, only to a similar runtime level (e.g. ES6 to ES5)

- Compilation — specifically separate from transpiling ES6 and JSX to ES5, is the act of including assets, processing CSS files as JSON, or other mechanisms that can load and inject external assets and code into a file. Besides, there are all sorts of build steps that can analyze your code and even optimize it for you.

- Minification and Compression — typically part of — but not exclusively controlled by — compilation, is the act of minifying and compressing a JS file into fewer and/or smaller files

- Source-Mapping — another optional part of the compilation is building source maps, which help identify the line in the source code that corresponds with the line in the output code (i.e. where an error occurred)

For React, there are specific build tool plugins, such as the babel plugins to optimize that involves compiling code into a format that optimizes React or a plugin to extract comments, etc.



```
babel-code-frame
babel-eslint
babel-extract-comments
babel-jest
babel-plugin-dynamic-import-node
babel-plugin-istanbul
babel-plugin-jest-hoist
babel-plugin-macros
babel-plugin-styled-components
babel-plugin-syntax-jsx
babel-plugin-syntax-object-rest-spread
babel-plugin-transform-object-rest-spread
babel-plugin-transform-react-remove-prop-types
babel-preset-react-app
babel-runtime
```

Figure 7: babel plugins.

For example, Babel automatically compiling JSX calls into a JS Object that inlines right into the source code:

This input;

```
function Foo() {
  return <div />;
}
```

Babel will turn the code into this;

```
import { jsx as _jsx } from
"react/jsx-runtime";
function Foo() {
  return _jsx("div", ...);
}
```

Each React component must have a render() mandatorily.

It returns a single React element which is the representation of the native DOM component. If more than one HTML element needs to be rendered, then they must be grouped inside one enclosing tag such as <div>, <form>, <group>, etc. This function should be kept pure i.e., it must return the same result each time it is invoked.

Now, this is just the basic;

Automate React App

Question:

What automation tools can be used to identify issues in a React App?

Answer:

You can run automation tools that can be used to identify issues such as code quality, accessibility, coverage, testing, and more.

Figure 8: automation tools.

Static Analysis Tools

Question:

Give examples of static analysis tools you can use to check your code.

Answer:

Linting, formatting, testing, Husky pre-commit; such as ESLint can be used with plugins can analyze React projects at a component level. Static analysis tools run very quickly, here are few tools you can be using;

- Eslint + Prettier
- Jest + Jest-dom + Enzym + Sinon
- E2E Testing

Browser Tools

Question:

Give examples of browser tools you can use to automate Dev & CI (continuous integration) cycle.

Answer:
You can automate Dev & CI cycle with Jest testing, Puppeteer e2e testing, Google Lighthouse to automated accessibility, Github Actions, Codecov.io, Coveralls, Travis & DeepScan, etc. The possibilities are endless see here: https://medium.com/react-courses/set-an-ultimate-react-automated-dev-ci-cycle-with-husky-jest-puppeteer-github-actions-codecov-46b923c4f8e3

Writing comments in JSX

Question:

How can you add comments in JSX?

Answer:

Now that you understand toolchaining better and what's Babel is doing under the hood and its part during the toolchain process. To write code inside JSX "//" won't work;

```
function Foo() {
  return (
    <>
    // won't compile
    </>)
}
```

But this would work because the Babel plugin can handle this comment;

```
function Foo() {
  return (
    <>
    {/* Corrent way */}
    </>)
}
```

Everything is a component

Question:

What do you understand if I say: in React - everything is a component?

Answer:

Components are the bread and butter, building blocks of a well-designed React application's UI.

These components split up the entire UI into an as smallest independent and reusable stand-alone loosely coupled pieces as possible. Then React can render each of these components independent of each other without affecting the rest of the UI. The rendering can be done by SPA or SSR.

This architecture design allows service workers, lazy loading, code splitting and another optimization method to be deployed. Read my article about optimizing your code.

Functional and class components

Question:

What is the difference between functional and class components?

Answer:

Components let you split the UI into independent, reusable pieces, and think about each piece in isolation.

The traditional React Components is using classes
with React.PureComponent, React.Component or React.createClass().

These create stateful components (if we set a state), include constructor as well as hooks functions that allow us to “hook into” React state and lifecycle features.

If we don’t need the complete lifecycle methods and state, we can write Components with a pure function (these are my favorite), hence the term “pure functional Component”:

```
function Hello() {
  return <></>
}
```

This function that returns a React Element can be used wherever we see fit:

```
DOM.render(<div><Hello /><div>)
```

With that being said, we can still set the state, pass props, and useEffect to tap into the component lifecycle.

In React v17, the react team added PureComponent. When you don’t need shouldComponentUpdate lifecycle event it’s better to use, React.PureComponent instead of React.Component;

```
export default class
MyClassCounter extends
React.PureComponent<IMyClassCoun
terProps, IMyClassCounterState>
{
  constructor(props:
IMyClassCounterProps) {
    super(props)
    this.state = {
      count: 0,
    }
  }

  render() {
    return (
      <>
        <p>You clicked
MyClassCounter
{this.state.count} times</p>
        <button type="submit"
onClick={() => this.setState({
count: this.state.count + 1 })}>
          Click MyClassCounter
        </button>
      </>
    )
  }
}

interface IMyClassCounterProps {
  // TODO
}

interface IMyClassCounterState {
  count: number
}
```

Read my article regarding the type of component you can create here: https://medium.com/react-courses/react-component-types-functional-class-and-exotic-factory-components-for-javascript-1a098a49a831

Embed multiple child components into a parent

Question:

How can I embed multiple child components into a parent?

Answer:

React child components can be embedded into a parent component. If you have been writing your entire code inside one large component, that is the standard, here is an example using class components;

```
Class SomeComponent extends
React.PureComponent{
  render(){
    return(
      <div>
        <h1>Hello</h1>
        <Header/>
      </div>
    )
  }
}
class Header extends
React.PureComponent{
  render() {
    return
    <h1>Header Component</h1>
  }
}
ReactDOM.render(
  <SomeComponent />,
document.getElementById('root')
)
```

Function Component with Hooks

Question:

Write a function component with hooks.

Answer:

If you need to use hooks, you can still use the function component. Using the functionality of the hook in React it is possible to use state without 'this', which simplifies component implementation and unit testing.

Example:

```
const SubmitButton = () => {
  const [myState, setMyState] =
useState(false);
return (
    <button onClick={() => {
        setMyState(true);
    }}>Submit</button>
  )
}
```

Component StrictMode

Question:

What is React component StrictMode?

Answer:

<StrictMode /> is a component helper included with React and provides additional visibility of potential issues that may exist in your components. If the application is running in development mode, any issues are logged to the development console, but these warnings are not shown if the application is running in production mode.

<StrictMode /> is great to use when in bug hunting mode to find problems such as deprecated lifecycle methods and legacy patterns, to ensure that all React components follow current best practices and side effect free. It’s also good to use when you unfamiliar with the codebase.

<StrictMode /> can be applied at any level of an application component hierarchy, which

allows it to be adopted incrementally within a codebase.

```
<React.StrictMode>
  <MyComponent />
</React.StrictMode>
```

Keep in mind, if you set linting correctly, that should do the same job as StrictMode and point out the same concerns so it may be redundancy depends on how you configure your project.

States vs props

Question:

What is the difference between states and props?

Answer:

Properties (aka props) and States are how we handle data in React. Props and States are the heart of React components.

Props are read-only components, which must be kept pure i.e. immutable. Props normally should be passed down from the parent to the child components. A child component is not meant to send a prop back to the parent component. React is meant to be a unidirectional data flow and is generally used to render the dynamically generated data.

States are the source of data and must be kept as simple as possible. States are the objects, which determine components rendering and behavior. They are mutable unlike the props and create dynamic and interactive components.

A mutable object can be changed after it's creation, and an immutable object can't.

```
class MyComponent extends
React.PureComponent {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }this.setState(prevState => {
  const newState =
prevState.count+1
  return ({
    ...prevState,
    count: newState
  });
})
```

Arrow functions

Question:

What are arrow functions in React (aka fat arrow)?

Answer:

Arrow functions (AKA ‘fat arrow‘ =>) are a brief syntax for writing the function expression. Use the fat arrow to bind the context of the components correctly since ES6 auto binding not available by default. Arrow functions are most useful while working with the higher-order functions.

```
// Fat Arrow Function examples//
JSX - Inline
<input name="firstName"
onChange={e => const newValue =
e.target.value } />// TS:<input
name="firstName" onChange= (e:
React.FormEvent<HTMLInputElement
>) => {
    const newValue =
e.currentTarget.value;
} />// Simple function
const turnStringToNumber: (str:
String) => Number =
  (str: String) => Number(str)//
Complex Callback and specifiy
result typesfunction
complexCallbackWithResultType(ca
llback: () => string): string {
  return callback();
}
```

Avoid binding

Question:

Why would you need to bind methods and how can you avoid that?

Answer:

In JavaScript, the value of ‘this’ changes depending on the current context. Within React class component methods, developers normally expect ‘this’ to refer to the current instance of a component, so it is necessary to bind these methods to the instance. Normally this is done in the constructor—for example:

```
class MyComponent extends
React.PureComponent {
  constructor(props) {
    super(props)
    this.handler =
this.handler.bind(this)
  }handler() {
    // TODO
}render() {
    return (
      <button
onClick={this.handler} />
    )
  }
}
```

There are several common approaches used to avoid this binding: Arrow Function

```
// Inline Arrow
FunctiononClick={() => {
  // TODO
}}onClick={this.handler}// Arrow
Function Assigned to a Class
Fieldclass SubmitButton extends
React.Component {
  state = {
    isFormSubmitted: false
  }handleSubmit = () => {
    // TODO
  }render() {
    return (
      <button
onClick={this.handleSubmit}>Subm
it</button>
    )
  }
}
```

Component lifecycle

Question:

Explain the React component lifecycle.
What changed between React v16 to v17?

Answer:

Each component in React has
a lifecycle that you can monitor and manipulate during its three main phases;

- Mounting Phase: the component is created to start its uni-directional data flow journey and make its way to the DOM. constructor(), static getDerivedStateFromProps(), r ender(), componentDidMount()

- Updating Phase: the component is added to the DOM, updates can and re-render on a prop or state change. static getDerivedStateFromProps(), shouldComponentUpdate(), render(), getSnapshotBeforeUpdate(), componentDidUpdate()
- Unmounting Phase: the final phase of a component's life cycle. The component is destroyed and removed from the DOM.componentWillUnmount()

Note: in React v17, access to previous events such as componentDidMount was deprecated and you should be using hooks features to access the component lifecycle events. getDerivedStateFromProps, getSnapshotBeforeUpdate, and componentDidUpdate.

Tip The main difference is that getDerivedStateFromProps is invoked right before calling the render method. getSnapshotBeforeUpdate runs before the most recently rendered output are committed componentDidUpdate runs after.

You can write that code in React 16, however, in React version 17, React the

code will generate an error message if you do that. If you want to learn React 17, read my article, also check out React docs: https://reactjs.org/docs/react-component.html
For example;

You can still use 'componentWillReceiveProps' but you now need to call it:

'UNSAFE_componentWillReceiveProps' — This is to let you know that is unsafe. Refactor and use 'getDerivedStateFromProps' instead.

Some developers often forget that the render function itself is a component event lifecycle on its own. For instance, if we want to clone data from the props into my component, for instance, the render is a great place to make the data update for my state in case I am passing props since the data can be used for the render;

Once the rendering is complete for component and child component 'componentDidUpdate' will be called and you can set more operations.

```
// Read more about component
lifecycle in the official docs:
  //
https://reactjs.org/docs/react-
component.html  public
shouldComponentUpdate(nextProps:
```

```
ITemplateNameProps, nextState:
ITemplateNameState) {
    // invoked before rendering
when new props or state are
being received.
    return true // or prevent
rendering: false
  }  static
getDerivedStateFromProps:

React.GetDerivedStateFromProps<I
TemplateNameProps,
ITemplateNameState> =
(props:ITemplateNameProps,
state: ITemplateNameState) => {
    // invoked right before
calling the render method, both
on the initial mount and on
subsequent updates
    // return an object to
update the state, or null to
update nothing.
    return null
  }  public
getSnapshotBeforeUpdate(prevProp
s: ITemplateNameProps,
prevState: ITemplateNameState) {
    // invoked right before the
most recently rendered output is
committed
    // A snapshot value (or
null) should be returned.
    return null
  }
componentDidUpdate(prevProps:
ITemplateNameProps, prevState:
ITemplateNameState, snapshot:
ITemplateNameSnapshot) {
    // invoked immediately after
updating occurs. This method is
```

```
not called for the initial
render.
    // will not be invoked if
shouldComponentUpdate() returns
false.
  }render() {
  if (this.props.data !==
this.state.data) {
    this.setState({
      data:
JSON.parse(JSON.stringify(this.p
rops.data))
    })
  }
  return (<SomeSubComponent
data={this.state.data} />)
}
```

Action once on initialize class vs function component

Question:

How can I perform an action only once when the UI render in Class and Function components?

Answer:

The componentDidMount() lifecycle method is part of the Mounting Phase and can be overridden with class components. Any actions defined within a componentDidMount() lifecycle hook is called only once when the component is first mounted.

For function components (FC) it is possible to use componentDidMount, these components are called exotic components, however, that’s considered a bad habit and you should use the useEffect.

The useEffect() hook is more flexible than the lifecycle methods used for class components. It receives two parameters: the callback function executed and the optional second parameter it takes of an array containing any variables that are to be tracked.

The value passed as the second argument controls when the callback is executed:

- The second parameter is undefined, the callback is executed every time the component is rendered.
- The second parameter is an array of variables, then callback executed as part of the first render cycle (mounting phase), but it will also be executed again each time an item in the array is modified (updating phase).
- If the second parameter contains an empty array, the callback will be executed only once as part of the first render cycle (mounting phase).

```
// Class Component
class MyComponent extends
React.Component {
  componentDidMount() {
    // TODO
  }
  render() {
    return <></>
  }
}

// FC - Exotic Components -
AVOID
function MyComponent(props) {
  return {
    componentDidMount() {
      alert('wow')
    }
    render() {
      return <></>
    }
  };
}

// FC - Correct
const Homepage = () => {
  useEffect(() => {
    // TODO
  }, [])

  return <></>;
}
```

Update state class vs function component

Question:

How would you go about updating the state in class vs function component?

Answer:

- Function component pick a variable array with the usage of ‘useState’
- Class component — it’s tempting to use ‘this.setState({ var: var+ 1 })}’ but wrong. Use a callback in setState when referencing the previous state.

```
// Function
component[selectedIndex,
setSelectedIndex] = useState(0)
setSelectednIndex(nextItem)//
Class component// avoid
this.setState({ count:
this.state.count + 1 })//
correct
this.setState(prevState => {
  const newState =
prevState.count+1
  return ({
    ...prevState,
    count: newState
  });
})
```

Higher-Order Components (HOCs)

Question:

What is Higher-Order Components (HOCs)?

Answer:

Higher-Order Components (HOCs) are the coined term for a custom Component that accepts dynamically provided children. Using the HOCs pattern helps to reuse the component's logic. HOC are custom components that wrap another child component within it. The parent can accept any dynamically provided child component, but they don't modify or copy behavior from their child components. HOC keeps the components 'pure'.

The HOC because we pass in elements as this.props.children when we nest those elements inside the parent container component. Oh and don't forget about extending interfaces with PropsWithChildren when you need them in the prop interface.

HOC can be used for many tasks like:

- Code and logic reuse.
- Render hijacking — control a component output from another component.
- Props, State, and/or bootstrap abstraction and manipulation.

For example, let's make a component. The component has a child image tag, waits until the component scrolled into view, and then loads the image in the background (before rendering to the DOM). The HOC accepts children via props;

```
<HOC>
    <div>child</div>
    <Props/>
</HOC>Example;// parent
<MyComponent>
  <img src="someURL.jpeg"/>
</MyComponent>// child
class MyComponent extends
React.PureComponent {
  render() {
    const {children} =
this.props
  }
}
```

Modularize with lazy load

Question:

How can I modularize a component with lazy load?

Answer:

Modularize code by using the export and import properties. This allows us to write components separately in different files. We can use the lazy which uses the LazyExoticComponent to lazy load our module.

```
//ParentComponent.jsx
import ChildComponent from
```

```
'./childcomponent'class
ParentComponent extends
React.PureComponent {
  render() {
    return(
      <>
        <ChildComponent />
      </>)
  }
}const ReactQuestions = lazy(()
=>
import('./childcomponent'))<Chil
dComponent />
```

Mixins

Question:

What is React Mixins?

Answer:

A mixin is a class that contains methods for use by other classes without having to be the parent class of those other classes.

In React Mixins are considered harmful and basically dead — see https://reactjs.org/blog/2016/07/13/mixins-considered-harmful.html;

> *“Let’s make it clear that mixins are not technically deprecated. If you use them React.createClass(), you may keep using them. We only say that they didn’t work well for us, and so we won’t recommend using them in the future”*

Why use Mixins?

Question:

When should I use React Mixins?

Answer:

The main use case is using mixins to have more control to merge several lifecycle methods — React can intelligently merge lifecycle methods, so each method will be called to form the composition initial state.

However, mixins components are often fragile and too easy to break, confusing to new developers, sometimes even to the devs who wrote them in the first place. That's why other techniques such as HOC are preferred.

However, it's good to know that option exists and in rare use cases maybe even useful.

```
const createReactClass =
require('create-react-class');

createReactClass({
  mixins: [PureRenderMixin],

  render: function() {
    return <div
className={this.props.className}
>foo</div>;
  }
});
```

Keys

Question:

What’s the importance of keys in React?

Answer:

Keys are used for identifying unique VDOM UI Elements with their corresponding data.

> *Keys help React identify which items have changed, are added, or are removed. Keys should be given to the elements inside the array to give the elements a stable identity.*
>
> — https://reactjs.org/docs/lists-and-keys.html

Keys and refs are added as an attribute to a React.createElement() call. They help React optimize rendering by recycling all the existing elements in the DOM.

These keys MUST be a unique number or string, using which React just reorders the elements instead of re-rendering them. Doing so will lead to an increase in the application’s performance.

```
// keys must be unique, don't
use array's index.
// Here are couple of
examples;{books.map((book:
bookObject) => (
  <BookListItem book={book}
key={book.id} />
))}// react-uui - is a great
library to generateconst uuid =
require('react-uuid') // yarn
add react-uuid
{cities.map((city, index) => (
  <circle
    key={`myId-${uuid()}`}
  />
)}
```

Refs

Question:

What are React Refs and when should I use them?

Answer:

The ref provide access to the DOM Element represented by the React Element.

Example of usages: integrating with 3rd party DOM libraries (example d3). Manage focus, select text, and/or media playback. Trigger animations.
Here's how;

- Refs can be either a string or a function. Using a string will tell React to automatically store the DOM Element as this.refs[refValue].
- Assign and store the ref

- Dynamically uses of refs — assign and store DOM nodes as variables in your code.

```
// Assign and store the
refexport default class
Component extends
React.PureComponent {

  ref: RefObject<HTMLDivElement>

  constructor(props:
ITemplateNameProps) {
    super(props)
    this.state = {
      // TODO
    }
    this.ref = React.createRef()
  }

  componentDidMount(){
    d3.select(this.ref.current)
      .append('p')
      .text('Hello World')
  }

  render() {
    return <div ref={this.ref}
/>
  }
}// Dynamically uses of
refs<input type="text" ref={node
=> this.myTextInput = node} />
```

Avoid Prop drilling

Question:

What is prop drilling and should I avoid them?

Answer:

In React everything is a component, data passed top-down (parent to child) via props. Let’s say you need a prop ina child of a child of a child of a parent component, what you do? You can pass that prop from one component to another. That technique deeply of nested component using a data provided by another component that is much higher in the hierarchy is called prop drilling.

The main disadvantage is a prop drilling is that components that should not otherwise be aware of the data — become unnecessarily complicated and cumbersome, harder to maintain because now we have to add that in our tests (if we can tests) as well as try to figure out the parent component that provided the data.

To avoid prop drilling, a common approach is to use React context. This allows a Provider the component that supplies data to be defined, and allows nested components to consume context data via either a Consumer component or a useContext hook.

> *“Context is designed to share data that can be considered “global” for a tree of React components, such as the current authenticated user,*

theme, or preferred language. For example, in the code below we manually thread through a “theme” prop in order to style the Button component”

— https://reactjs.org/docs/context.html

While context can be used directly for sharing global state, it is also possible to use context indirectly via a state management module. If you dealing with a data being retrieved or maintained by state management such as Redux or even better React own solution Recoil, you can pass that data directly to the grandchild components.

```
// Parent component const
MyContext =
React.createContext('Hello');cla
ss ParentComponent extends
React.Component {
  render() {
    return (
      <MyContext.Provider
value="world">
        <Toolbar />
      </MyContext.Provider>
    );
  }
}

// A child component - don't
need the props;function
ChildComponent() {
  return (
    <div>
      <GrandChildComponent />
    </div>
  );
}// Grand child componentclass
GrandChildComponent extends
React.Component {
static contextType = MyContext;
  render() {
    return <>{this.context}</>
  }
}
```

Recoil example;

```
const GrandChildComponent = ()
=> {
  const data: dataObject[] =
useRecoilValue(getData) as
dataObject[]
}
```

Testing Shallow vs Mount vs Render

Question:

What is difference between unit testing Shallow vs Mount vs Render?

Answer:

Enzyme includes three rendering methods. Mount, Shallow and Render.

- Mount — https://enzymejs.github.io/enzyme/docs/api/mount.html

- Shallow — https://enzymejs.github.io/enzyme/docs/api/shallow.html
- Render — https://enzymejs.github.io/enzyme/docs/api/render.html

Mounting

- Full DOM rendering including child components.
- Ideal for use cases where you have components that need to interact with DOM API, or use React lifecycle methods.
- Allows access to both props directly passed into the root component (including default props) and props passed into child components.

Shallow

- Renders only the single component, not children. This is useful to isolate the component for pure unit testing. It protects against changes or bugs in a child component.
- shallow components access to lifecycle methods by default

- Cannot access props passed into the root component, but can access props passed into child components, and can test the effect of props passed into the root component.
- When we call shallow(<Calculator />), we are testing what Calculator renders - not the element we passed into shallow

Render

- Renders to static HTML, including children.
- It does not have access to React lifecycle methods.
- Uses fewer resources than the other APIs, but have less functionality.

Check my Jest + Enzyme article here: https://medium.com/javascript-in-plain-english/are-you-not-testing-your-react-app-instantly-test-with-jest-enzyme-a-reactjs-2020-tutorial-e9ce0182d66d

```
import React from 'react'
import { shallow } from 'enzyme'
import MyComponent from
'./MyComponent'

describe('<MyComponent />', ()
=> {
 let component

 beforeEach(() => {
  component =
shallow(<MyComponent/>)
 })

 test('It should mount', () => {

expect(component.length).toBe(1)
 })
})
```

Styling a React application

Question:

What are my options when it comes to styling a react App?

Answer:

CSS Classes

React class names can be specified for a component like class names are specified for a DOM element in HTML.

```
// CSS Classes
<div className="top-right">
```

Inline CSS

Styling React elements using inline CSS allows styles to be completely scoped to an element using a well-understood, standard approach. However, certain styling features are not available with inline styles.

```
// Inline CSS
<div style={{ width: '100%' }}>
```

Pre-processors

Cascading Style Sheets (CSS) preprocessor lets you generate CSS from the preprocessor's own unique syntax. The dynamic preprocessor style sheet language compiled into CSS. Examples: Sass/SCSS, PostCSS, LESS, Stylus.

Read here: https://medium.com/react-courses/ready-to-integrate-or-switch-css-preprocessors-on-react-project-sass-scss-vs-postcss-vs-less-vs-58bf26c379ab

CSS-in-JS Modules

CSS-in-JS modules are a popular option for styling React applications because they integrate closely with React components.

For example, you can change style based on React props at runtime. Also, by default, most of these systems scope all styles to the respective component being styled, popular examples: Styled Components, Emotion, and Styled-jsx.

```
// SCSS Example// SCSS
source$font-stack:    Helvetica,
sans-serif;
```

```
$primary-color: #333;
body {
  font: 100% $font-stack;
  color: $primary-color;
}
// CSS output;
body {
  font: 100% Helvetica, sans-
serif;
  color: #333;
}
```

CSS framework

CSS Frameworks: use the main ones; Bootstrap or MaterialUI.

Material-UI — is popular and has a styled import similar to Styled components but also support using Styled components. We can leverage the advantages of both Material-UI and styled-components together.

Material-UI has plenty examples of implementations, here’s an example of lists: https://material-ui.com/components/lists/

Bootstrap — is one of the most popular and widely used CSS framework. React Bootstrap is a set of React components that implementation of Bootstrap framework.

```
// BootstrapIf you need
bootstrap. Just add;
$ yarn add bootstrap// add to
index.tsx;
```

```
import
'bootstrap/dist/css/bootstrap.cs
s'
```

Stateful vs stateless components

Question:

What's the difference between stateful vs stateless components?

Answer:

The difference between stateless and stateful components is that stateful component has a state object, and a stateless component doesn't.

When you don't need state it's better to keep your component stateless as much as possible, examples of when to use stateless components;

- props data is all you need
- none-interactive element
- reusable code for different use cases

```
function WelcomeUser(props) {
    return <h1>Hi
{props.userName}</h1>;
}
const element = <WelcomeUser
userName="John" />;

ReactDOM.render(
    element,

document.getElementById('app')
);
```

Controlled vs uncontrolled components

Question:

What is the difference between controlled vs uncontrolled components?

Answer:

Controlled component does not maintain their own state vs uncontrolled Components.

Controlled component

https://reactjs.org/docs/lifting-state-up.html

In controlled component the data is controlled by some parent component. They take the data via props and send a callback to notify of a change. Holds no knowledge of past, current, and possible future changes in state

Uncontrolled components

https://reactjs.org/docs/uncontrolled-components.html

Data is controlled by the DOM. Refs are used to getting their current values from the DOM, and the component should hold no knowledge of past, current, and possible future changes in state.

```
// Controlled
componenthandleChange(e) {

this.props.onChange(e.target.val
ue);
    // ...
}// Uncontrolled components
  handler(event) {

console.log('this.input.current.
value)
  }
```

Event & synthetic events in React

Question:

Part I: What is difference between DOM events vs React events? Part II: What are synthetic events in React?

Answer:

In React, events are the triggered reactions to specific actions like mouse hover, mouse click, keypress, etc. Handling events in React is similar to handling events in DOM elements. But there are some syntactical differences.

> *SyntheticEvent, a cross-browser wrapper around the browser's*

native event. It has the same interface as the browser's native event,
including stopPropagation() and pr eventDefault(), except the events work identically across all browsers.

— https://reactjs.org/docs/events.html

- Events are named using a camel case instead of just using the lowercase. For example: React.KeyboardEven tand React.MouseEvent.
- Events are passed as functions instead of strings.

If you follow the code, you can see that React Events extends UIEvents which extends SyntheticEvent.

React uses its own event system. That's why we can't use typically MouseEvent from standard DOM. We need to use them, otherwise, we get an error or just we won't be able to access methods. Generally speaking, most events are mapped with the same name.

Luckily, React typings give you the proper equivalent of each event you might be familiar with from standard DOM.

We can either useReact.MouseEvent or import the MouseEvent typing from the React module. We can either useReact.MouseEvent or import the MouseEvent typing from the React module:

```
const onClickHandler = (evt:
React.MouseEvent) => {
  // TODO
  evt.preventDefault()
}<Button type="submit"
onClick={onClickHandler}>
```

Event argument contains a set of specific properties to an event. Each event type contains has its own properties and behavior, that can be accessed via its event handler only.

```
class Display extends
React.Component({
  handler(evt: React.MouseEvent)
{
    // TODO
  },
  render() {
    return (
      <div
onClick={this.handler}>Click</di
v>
    )
  }
})
```

Forms

Question:

What is unique about React forms vs HTML forms?

Answer:

React forms are similar to HTML forms. But in React, the state is contained in the state property of the component and updated via setState() only.

Thus, the form elements don’t hold state and can’t directly update their state. Submission is handled by a method we need to set. This method we set has full access to the data that is entered by the user into the form.

> *In React, mutable state is typically kept in the state property of components, and only updated with setState()*
>
> — https://reactjs.org/docs/forms.html

```
render() {
  return (<form
onSubmit={this.handler}>
      <label>
        Name:
        <input type="text"
value={this.state.value}
onChange={this.handler} />
      </label>
      <input type="submit"
value="Submit" />
    </form>);
}
```

Optimize slow App

Question:

What would do if your React App is slow?

Answer:

One of the most common issues in React applications is when components re-re-re-render. There are two tools provided by React that are helpful in these situations:

- React.memo()- this prevents unnecessary re-rendering of function components
- PureComponent-this prevents unnecessary re-rendering of class components

Beside that we can optimize our React App using the following;

- PureComponent and React.memo() — gain performance.
- Lazy loading — break your JS bundles and serve once needed avoiding wait time to see your content.
- Prerender — almost static HTML.
- Precache — have your App work offline.
- Code Splitting — split JS Bundle even more.

- Tree shaking — dead code removal configure 'package.json' file.
- Reduce Media size — reduce the size of the media resources with image sprite.
- Prefetching — set loading hierarchy.
- Clean unused side effects event handlers — avoid memory leaks.

Check my article: https://medium.com/react-courses/optimize-react-app-best-optimzing-techniques-i-wish-i-knew-before-i-wrote-my-first-line-of-code-2b4651f45a48

React vs React Native

Question:

What's the difference between React and React Native?

Answer:

React is a framework for building applications using JavaScript. React Native is an entire platform to build native, cross-platform mobile apps, and React.

React browser code is rendered through VDOM, React Native uses mobile Native API's to render components.

Let's Recap

In this article, part I of II, I will be covering the following:

- React vs Angular
- React advantages
- React limitations
- DOM vs VDOM.
- JSX
- ES6 vs ES5
- Toolchain
- Automate React App
- Writing comments in JSX
- Everything is a component
- Functional and class components
- Embed multiple child components into the parent
- Function Component with Hooks
- Component StrictMode
- States vs props

- Arrow functions
- Avoid binding
- Component lifecycle
- Action once on init class vs function component
- Update state class vs function component
- Higher-Order Components (HOCs)
- Modularize with lazy load
- Mixins
- Keys
- Refs
- Avoid Prop drilling
- Testing Shallow vs Mount vs Render
- Styling a React application
- Stateful vs stateless components
- Controlled vs uncontrolled components
- Event & synthetic events in React
- Forms
- Optimize slow App

- React vs React Native

Summary

In this book I covered questions and answers that touches the most important aspect of both programming in JS/TS as well as top concepts of ReactJS.

The book was classified into two main categories with over 60 questions and aswers:

- JavaScript / TypeScript Questions
- React Core

Development is a journey that require constant attention, learning and improving and I hope you not only found this book helpful preparing to your next interview and help you achieve your goals but als you draw an inspiration and learned a thing or two you did not know about React and JS/TS.

As a bonus, I have created for you a flip book that can help you review all the questions and answers in this book: https://elielrom.com/ReactQuestions.

Finally, I would like to invite you to checkout my courses on Udemy: https://www.udemy.com/user/eli-elad-elrom/

Thank you again for purchasing this book and keep learning.