Deep Dive Into Modern Web Development

Full Stack open

Part 1 Intro to React

1. var, let and const - ES6 JavaScript Features

The var statement declares function-scoped or globally-scoped variables, optionally initialising each to a value.
The let declaration declares re-assignable, block-scoped local variables, optionally initialising each to a value.
Compared with var, let declarations have the following **differences**:

- let declarations are scoped to blocks as well as functions.
- let declarations can only be accessed after the place of declaration is reached (see temporal dead zone). For this reason, let declarations are commonly regarded as non-hoisted.
- let declarations do not create properties on globalThis when declared at the top level of a script.
- let declarations cannot be redeclared by any other declaration in the same scope.
- let begins declarations, not statements. That means you cannot use a lone let declaration as the body of a block (which makes sense, since there's no way to access the variable).

```
if (true) let a = 1; // SyntaxError: Lexical declaration cannot appear in a single-statement context
```

The const declaration declares block-scoped local variables. The value of a constant can't be changed through reassignment using the assignment operator, but if a constant is an object, its properties can be added, updated, or removed.
The const declaration is very **similar** to let:

- const declarations are scoped to blocks as well as functions.
- const declarations can only be accessed after the place of declaration is reached (see temporal dead zone). For this reason, const declarations are commonly regarded as non-hoisted.
- const declarations do not create properties on globalThis when declared at the top level of a script.
- const declarations cannot be redeclared by any other declaration in the same scope.

- const begins declarations, not statements. That means you cannot use a lone const declaration as the body of a block (which makes sense, since there's no way to access the variable).

```
if (true) const a = 1; // SyntaxError: Lexical declaration cannot appear in a
single-statement context
```

2. Map, filter, reduce

2.1. JavaScript Array map()

- map() creates a new array from calling a function for every array element.
- map() does not execute the function for empty elements.
- map() does not change the original array.

2.2. JavaScript Array filter()

- The filter() method creates a new array filled with (storing) elements that pass a test (condition) provided by a function.
- The filter() method does not execute the function for empty elements.
- The filter() method does not change the original array.

2.3. JavaScript Array reduce()

- The reduce() method executes a reducer function for array elements.
- The reduce() method returns a single value: the function's accumulated result.
- The reduce() method does not execute the function for empty array elements.
- The reduce() method does not change the original array.

Note: At the first callback, there is no return value from the previous callback.
Normally, array element 0 is used as initial value, and the iteration starts from array element 1.
If an initial value is supplied, this is used, and the iteration starts from array element 0.

```
const numbers = [15.5, 2.3, 1.1, 4.7];
console.log(numbers.reduce(getSum, 0)) // 0, 16, 18, 19, 14. Outputs 24
console.log(numbers.reduce(getSum)) // 15.5, 17.5, 18.5, 23.5. Outputs 23.5
function getSum(total, num) {
    console.log(total)
    return total + Math.round(num);
}
```

3. Destructuring

const t = [1, 2, 3, 4, 5]
const [first, second, …rest]

4. What is this?

In JavaScript, the ‘this’ keyword refers to an object.
Which object depends on how this is being invoked (used or called).
The this keyword refers to different objects depending on how it is used:

- In an object method, this refers to the object.
- Alone, this refers to the global object.
- In a function, this refers to the global object.
- In a function, in strict mode, this is undefined.
- In an event, this refers to the element that received the event.
- Methods like call(), apply(), and bind() can refer this to any object.

If you want to gain a better understanding of how this works in JavaScript, the Internet is full of material about the topic, e.g. the screencast series Understand JavaScript's this Keyword in Depth by egghead.io is highly recommended!

5. call, apply, bind

In JavaScript all functions are object methods.

5.1. call

With the call() method, you can write a method that can be used on different objects.
With call(), an object can use a method belonging to another object.

```
const person = {
  fullName: function() {
    return this.firstName + " " + this.lastName;
  }
}

const person1 = {
  firstName:"John",
  lastName: "Doe"
}
```

```
// This will return "John Doe":
person.fullName.call(person1);
```

The call() method can accept arguments:

```
const person = {
 fullName: function(city, country) {
    return this.firstName + " " + this.lastName + "," + city + "," + country;
  }
}
const person1 = {
  firstName:"John",
  lastName: "Doe"
}
person.fullName.call(person1, "Oslo", "Norway")
// Outputs John Doe,Oslo,Norway
```

5.2. apply

The apply() method is similar to the call() method.
The difference is:
The call() method takes arguments separately.
The apply() method takes arguments as an array.

```
person.fullName.apply(person1, ["Oslo", "Norway"]);
```

Simulate a Max Method on Arrays

You can find the largest number (in a list of numbers) using the Math.max() method:

```
Math.max(1,2,3);  // Will return 3
Since JavaScript arrays do not have a max() method, you can apply the Math.max()
method instead.
Math.max.apply(null, [1,2,3]); // Will also return 3
```

The first argument (null) does not matter:

```
Math.max.apply(Math, [1,2,3]); // Will also return 3
Math.max.apply(" ", [1,2,3]); // Will also return 3
Math.max.apply(0, [1,2,3]); // Will also return 3
```

5.3. bind

5.3.1 Function Borrowing

With the bind() method, an object can borrow a method from another object.
The member object borrows the fullname method from the person object:

```
const person = {
  firstName:"John",
  lastName: "Doe",
  fullName: function() {
    return this.firstName + " " + this.lastName;
  }
}

const member = {
  firstName:"Hege",
  lastName: "Nilsen",
}

let fullName = person.fullName.bind(member);
fullName(); // Outputs: Hege Nilsen
```

5.3.2 Preserving this

```
person.display() // Outputs John Doe
```

Sometimes the bind() method has to be used to prevent losing this.
When a function is used as a callback, this is lost.

```
setTimeout(person.display, 3000); // Outputs undefined undefined
```

The bind() method solves this problem.

```
let display = person.display.bind(person); // bind person.display to person
setTimeout(display, 3000); // Outputs John Doe
```

6. JavaScript Closures

A closure gives you access to an outer function's scope from an inner function.
JavaScript variables can belong to the local or global scope.
Global variables can be made local (private) with closures.

6.1. Global Variables

A function can access all variables defined inside the function as well as outside the function.
In a web page, global variables belong to the page. Global variables can be used (and changed) by all other scripts in the page.
A local variable can only be used inside the function where it is defined. It is hidden from other functions and other scripting code.
Global and local variables with the same name are different variables. Modifying one, does not modify the other.

6.2. Variable Lifetime

Global variables live until the page is discarded, like when you navigate to another page or close the window.
Local variables have short lives. They are created when the function is invoked, and deleted when the function is finished.

```
1, myFunction();
 function myFunction() {
   let a = 5 // local variable
   console.log(a * a) // Outputs 25
 }
 console.log(a)  // Outputs null

2, let a = 4 ;
myFunction();
function myFunction() {
  let a = 5
  console.log(a * a)
}
console.log(a) // outputs 4
```

6.3. A Counter Dilemma

Suppose you want to use a variable for counting something, and you want this counter to be available to all functions.
You could use a global variable, and a function to increase the counter:

```
// Initiate counter
let counter = 0;

// Function to increment counter
function add() {
  counter += 1;
}
```

```
// Call add() 3 times
add();
add();
add();
// The counter should now be 3
```

Problem: Any code on the page can change the counter, without calling add(). The counter should be local to the add() function, to prevent other code from changing it:

```
// Initiate counter
let counter = 0;

// Function to increment counter
function add() {
  let counter = 0;
  counter += 1;
}

// Call add() 3 times
add();
add();
add();
//The counter should now be 3. But it is 0
```

It did not work because we display the global counter instead of the local counter. We can remove the global counter and access the local counter by letting the function return it:

```
// Function to increment counter
function add() {
  let counter = 0;
  counter += 1;
  return counter;
}

// Call add() 3 times
add();
add();
add();
//The counter should now be 3. But it is 1.
```

It did not work because we reset the local counter every time we call the function.

A JavaScript inner function can solve this.

6.4. JavaScript Nested Functions

All functions have access to the global scope.
In fact, in JavaScript, all functions have access to the scope "above" them.
JavaScript supports nested functions. Nested functions have access to the scope "above" them.

```
function add() {
  let counter = 0;
  function plus() {counter += 1;}
  plus();
  return counter;
}
```

The inner function plus() has access to the counter variable in the parent function.
The counter dilemma could be solved, if we could reach the plus() function from the outside.
We also need to find a way to execute counter = 0 only once.
We need **closure**.

```
function add() {
  let counter = 0;
  function plus() {counter += 1;}
  plus();
  return counter;
}

const add = (function () {
  let counter = 0;
  return function () {counter += 1; return counter}
})();

add();
add();
add();
// the counter is now 3
```

Example Explained

The variable add is assigned to the return value of a self-invoking function.
The self-invoking function only runs once. It sets the counter to zero (0), and returns a function expression.

This way add becomes a function. The "wonderful" part is that it can access the counter in the parent scope.
This is called a JavaScript closure. It makes it possible for a function to have "private" variables.
The counter is protected by the scope of the anonymous function, and can only be changed using the add function.
A closure is a function having access to the parent scope, even after the parent function has closed.

7. React

Writing components with React is easy, and by combining components, even a more complex application can be kept fairly maintainable. Indeed, a core philosophy of React is composing applications from many specialised reusable components.
First letter of React component names must be capitalised. Note that the content of a React component (usually) needs to contain one root element.

8. Rule of hooks

Functions whose names start with use are called Hooks in React.

Only call Hooks at the top level

Don't call Hooks inside loops, conditions, nested functions, or `try`/`catch`/`finally` blocks.

- Call them at the top level in the body of a function component.
- ✅ Call them at the top level in the body of a custom Hook.

```
function Counter() {
  // ✅ Good: top-level in a function component
  const [count, setCount] = useState(0);
  // ...
}

function useWindowWidth() {
  // ✅ Good: top-level in a custom Hook
  const [width, setWidth] = useState(window.innerWidth);
  // ...
}
```

Note: Custom Hooks *may* call other Hooks (that's their whole purpose). This works because custom Hooks are also supposed to only be called while a function component is rendering.

Only call Hooks from React functions

Don't call Hooks from regular JavaScript functions. Instead, you can:

✅ Call Hooks from React function components.

✅ Call Hooks from custom Hooks.

9. An event handler is a function

Event handlers must always be a function or a reference to a function. The button will not work if the event handler is set to a variable of any other type.

10. React component

- React lets you create components, reusable UI elements for your app.
- In a React app, every piece of UI is a component.
- React components are regular JavaScript functions except:
    - Their names always begin with a capital letter.
    - They return JSX markup.

11. Do Not Define Components Within Components

- Reduces the reusability
- Have an impact on performance. Each time the parent component renders, the inner component is redefined, preventing React from optimising updates. Defining a component within another means that each render creates a new instance of the inner component. The inner component will be recreated on each render cycle of the parent component such that the application loses its previous state.

12. Rendering Lists

Don't use an item's index in the array as its key.
https://react.dev/learn/rendering-lists#keeping-list-items-in-order-with-key

13. Updating Objects in State

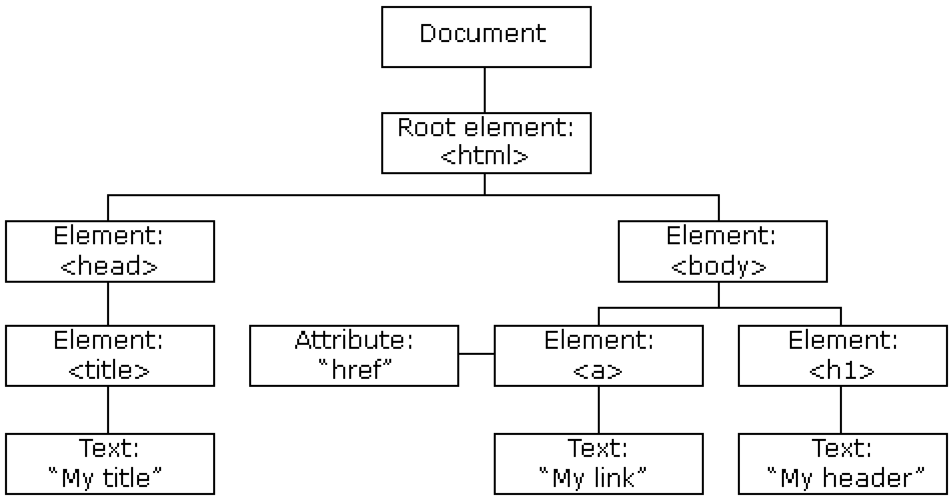
We must never mutate state directly in React! (Because of shallow comparison)

14. HTML DOM

The HTML DOM is a standard object model and programming interface for HTML. It defines:

- The HTML elements as objects
- The properties of all HTML elements
- The methods to access all HTML elements
- The events for all HTML elements

In other words: The HTML DOM is a standard for how to get, change, add, or delete HTML elements.
When a web page is loaded, the browser creates a Document Object Model of the page.
The HTML DOM model is constructed as a tree of Objects:



15. Virtual DOM

The virtual DOM (VDOM) is a programming concept where an ideal, or “virtual”, representation of a UI is kept in memory and synced with the “real” DOM by a library such as ReactDOM. This process is called reconciliation.

Part 2 Communicating with server

1. Promise

The Promise object represents the eventual completion (or failure) of an asynchronous operation and its resulting value.
A Promise is in one of these states:

- pending: initial state, neither fulfilled nor rejected.
- fulfilled: meaning that the operation was completed successfully.
- rejected: meaning that the operation failed.

Essentially, a promise is a returned object to which you attach callbacks, instead of passing callbacks into a function.
The then() function returns a new promise, different from the original

2. async/await

The **async** function declaration creates a binding of a new async function to a given name. The await keyword is permitted within the function body, enabling asynchronous, promise-based behaviour to be written in a cleaner style and avoiding the need to explicitly configure promise chains.
The **await** operator is used to wait for a Promise and get its fulfilment value. It can only be used inside an async function or at the top level of a module.

3. State hook and effect hook

State: A Component's Memory

Components need to “remember” things, for example, the current input value. In React, this kind of component-specific memory is called state.
The useState Hook provides those two things:

1. A state variable to retain the data between renders.
2. A state setter function to update the variable and trigger React to render the component again.

In React, useState, as well as any other function starting with “use”, is called a Hook.
State is local to a component instance on the screen. In other words, if you render the same component twice, each copy will have a completely isolated state!
Unlike props, state is fully private to the component declaring it.

Synchronising with Effects

Effects let a component connect to and synchronise with external systems. This includes dealing with network, browser DOM, animations, widgets written using a different UI library, and other non-React code.

- useEffect connects a component to an external system.

As such, effect hooks are precisely the right tool to use when fetching data from a server.

4. Axios

Promise-based function fetch to pull the data from the server. Fetch is a great tool. It is standardised and supported by all modern browsers (excluding IE).
The axios library can be used for communication between the browser and server. Axiod is a Promise-based HTTP client for the browser and node.js.

Features

- Make XMLHttpRequests from the browser
- Make http requests from node.js
- Supports the Promise API
- Intercept request and response
- Transform request and response data
- Cancel requests
- Automatic transforms for JSON data
- 🆕 Automatic data object serialization to `multipart/form-data` and `x-www-form-urlencoded` body encodings
- Client side support for protecting against XSRF

Part 3 Programming a server with NodeJS and Express

1. NodeJS

Node.js is an open-source and cross-platform JavaScript runtime environment. Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside of the browser.

2. Express

Implementing our server code directly with Node's built-in http web server is possible. However, it is cumbersome, especially once the application grows in size.
Many libraries have been developed to ease server-side development with Node, by offering a more pleasing interface to work with the built-in http module. These libraries aim to provide a better abstraction for general use cases we usually require to build a backend server. By far the most popular library intended for this purpose is express.

3. REST

REST (representational state transfer) is a software architectural style that was created to guide the design and development of the architecture for the World Wide Web.
The term *representational state transfer* was introduced and defined in 2000 by computer scientist Roy Fielding in his doctoral dissertation. It means that a server will respond with the representation of a resource (today, it will most often be an HTML, XML or JSON document) and that resource will contain hypermedia links that can be followed to make the state of the system change. Any such request will in turn receive the representation of a resource, and so on.
https://www.geeksforgeeks.org/hateoas-and-why-its-needed-in-restful-api/

4. Routing

Routing refers to determining how an application responds to a client request to a particular endpoint, which is a URI (or path) and a specific HTTP request method (GET, POST, and so on).

5. Middleware

An Express application is essentially a series of middleware function calls.

Middleware functions are functions that have access to the request object (req), the response object (res), and the next middleware function in the application's request-response cycle. The next middleware function is commonly denoted by a variable named next.

6. Same origin policy

The same-origin policy is a critical security mechanism that restricts how a document or script loaded by one origin can interact with a resource from another origin.
Two URLs have the *same origin* if the protocol, port (if specified), and host are the same for both.

This mechanism bears a particular significance for modern web applications that extensively depend on HTTP cookies to maintain authenticated user sessions, as servers act based on the HTTP cookie information to reveal sensitive information or take state-changing actions.

The same-origin policy applies only to scripts. This means that resources such as images, CSS, and dynamically-loaded scripts can be accessed across origins via the corresponding HTML tags. Attacks take advantage of the fact that the same origin policy does not apply to HTML tags.

7. CORS (Cross-Origin Resource Sharing)

Cross-Origin Resource Sharing (CORS) is an HTTP-header based mechanism that allows a server to indicate any origins (domain, scheme, or port) other than its own from which a browser should permit loading resources.
(Cross-origin resource sharing (CORS) is a mechanism that allows restricted resources on a web page to be accessed from another domain outside the domain from which the first resource was served.)

8. CSRF

CSRF (Cross-Site Request Forgery) is an attack that impersonates a trusted user and sends a website unwanted commands.
For more details, visit Lecture 23: CSRF + Impersonation Attacks of CS 161 Computer Security, UC Berkeley.

9. Mongoose

Mongoose provides a straight-forward, schema-based solution to model your application data. It includes built-in type casting, validation, query building, business logic hooks and more, out of the box.
Mongoose could be described as an object document mapper (ODM), and saving

JavaScript objects as Mongo documents is straightforward with this library.

Part 4 Testing Express servers, user administration

1. HTTP response status codes

HTTP response status codes indicate whether a specific HTTP request has been successfully completed. Responses are grouped in five classes:

- Informational responses (100 – 199)
- Successful responses (200 – 299)
- Redirection messages (300 – 399)
- Client error responses (400 – 499)
- Server error responses (500 – 599)

2. Proxy

Configure custom proxy rules for the dev server.

3. express.Router

Use the express.Router class to create modular, mountable route handlers. A Router instance is a complete middleware and routing system.

Part 5 Testing React apps

1. props.children

When you nest content inside a JSX tag, the parent component will receive that content in a prop called children.

2. ref mechanism

When you want a component to “remember” some information, but you don’t want that information to trigger new renders, you can use a ref.

3. XSS

Cross-site scripting (XSS) is a security exploit which allows an attacker to inject malicious client-side code into a website. This code is executed by the victims and lets the attackers bypass access controls and impersonate users.

Part 6 Advanced state management

1. Redux Overview and Concepts
2. React Query

React Query is a server-state library, responsible for managing asynchronous operations between your server and client

3. Push, concat

The push() method of Array instances adds the specified elements to the end of an array and returns the new length of the array.
The concat() method of Array instances is used to merge two or more arrays. This method does not change the existing arrays, but instead returns a new array.

Part 7 React router, custom hooks, styling app with CSS and webpack

1. React Router

React has the React Router library which provides an excellent solution for managing navigation in a React application.
BrowserRouter is a Router that uses the HTML5 history API (pushState, replaceState and the popState event) to keep your UI in sync with the URL.
A <BrowserRouter> stores the current location in the browser's address bar using clean URLs and navigates using the browser's built-in history stack.

2. Built-in React DOM Hooks

The react-dom package contains Hooks that are only supported for web applications (which run in the browser DOM environment). These Hooks are not supported in non-browser environments like iOS, Android, or Windows applications. If you are looking for Hooks that are supported in web browsers and other environments see Built-in React Hooks.

3. Built-in React Hooks

Hooks let you use different React features from your components. You can either use the built-in Hooks or combine them to build your own.

4. Custom hooks

React offers the option to create custom hooks. According to React, the primary purpose of custom hooks is to facilitate the reuse of the logic used in components. Building your own Hooks lets you extract component logic into reusable functions. Custom hooks are not only a tool for reuse; they also provide a better way for dividing our code into smaller modular parts.

5. lifecycle methods

componentDidMount
componentDidUpdate
componentWillUnmount

6. Difference between Functional components and Class components

The biggest difference between Functional components and Class components is mainly that the state of a Class component is a single object, and that the state is updated using the method setState, while in Functional components the state can consist of multiple different variables, with all of them having their own update function.
A notable benefit of using Functional components is not having to deal with the self-referencing this reference of the Javascript class.

7. Model View Controller pattern

Model: The model represents the data and business logic of the application. It encapsulates the data and behaviour of the application's domain, such as user information, database interactions, and application rules. In a web application, the model interacts with the database to fetch or store data and performs operations on that data.
View: The view is responsible for presenting the user interface to the user. It renders the data from the model in a user-friendly format, typically as HTML, CSS, and JavaScript in web applications. Views are passive components that display information and respond to user input but do not perform any business logic or data manipulation.
Controller: The controller acts as an intermediary between the model and the view. It receives input from the user via the view, processes it, and interacts with the model to perform the necessary actions. The controller updates the model based on user input or other external events and then instructs the view to update accordingly. Controllers handle user interactions and application logic, coordinating the flow of data between the model and the view.

8. SQL injection

SQL injection is a code injection technique that might destroy your database.
SQL injection is one of the most common web hacking techniques.
SQL injection is the placement of malicious code in SQL statements, via web page input.
SQL Injection Based on ""="" is Always True
SQL Injection Based on Batched SQL Statements

```
let query = "SELECT * FROM Users WHERE name = '" + userName + "';"
Arto Hell-as'; DROP TABLE Users; --
SELECT * FROM Users WHERE name = 'Arto Hell-as'; DROP TABLE Users; --'
```

SQL injections are prevented using parameterized queries. With them, user input isn't mixed with the SQL query, but the database itself inserts the input values at placeholders in the query (usually ?).

```
execute("SELECT * FROM Users WHERE name = ?", [userName])
```

Injection attacks are also possible in NoSQL databases. However, mongoose prevents them by sanitizing the queries. More on the topic can be found e.g. here.

Part 13 Using relational databases

1. Sequelize

Sequelize is a so-called Object relational mapping (ORM) library that allows you to store JavaScript objects in a relational database without using the SQL language itself, similar to Mongoose that we used with MongoDB.