**New Age University**

Web Development Course

E-Book

**Course Description**

This comprehensive 8-week course aims to equip students with the essential skills and knowledge to become a web developer. You'll delve into the core technologies that power the web, learn how to build user-friendly and interactive websites, and gain hands-on experience with industry-standard tools and frameworks. Through a combination of lectures, coding exercises and projects you'll be well-equipped to contribute to web development projects or launch your own web development career.

**Learning Objectives**

* Understand web development (front-end, back-end, full-stack).
* Develop proficiency in core technologies that power the web (HTML, CSS, JavaScript).
* Apply best practices for writing clean, maintainable, and accessible HTML and CSS code.
* Understand the benefits of using client-side frameworks like React or Vue.js for building complex web applications.
* Understand the basics of server-side development using technologies like PHP or Python (basics).
* Complete a Capstone Project that demonstrates the practical application of course concepts.
* Develop a portfolio showcasing your web development skills.
* Learn about additional web development technologies and career paths.
* Identify resources for continued learning and professional development.

**Target Audience**

This course is ideal for:

• Individuals with no prior coding experience who are interested in learning web development.

• Career changers seeking to transition into a web development role.

• Business owners and marketers who want to understand how websites are built and managed.

# Week 1: Introduction to Web Development

**Overview:** In Week 1, students will be introduced to the fundamental concepts of web development, including the distinctions between front-end and back-end development. The focus will be on understanding HTML (Hypertext Markup Language), setting up a development environment, and building a basic web page. This week lays the foundation for all subsequent learning in the course.

**Topics Covered:**

1. Introduction to Web Development
2. Understanding Front-End vs. Back-End Development
3. HTML Basics
4. Setting Up a Development Environment
5. Introduction CSS

### 1. ****Introduction to Web Development****

#### ****1.1. What is Web Development?****

Web development refers to the process of building and maintaining websites. It encompasses everything from creating simple static pages of plain text to complex web applications, e-commerce platforms, and social media networks. The field of web development is vast, and it involves several different technologies and disciplines.

Web development can be broadly categorized into three areas:

* **Front-End Development (Client-Side):** Focuses on the visual and interactive aspects of a website—the part that users see and interact with. It involves technologies like HTML, CSS, and JavaScript.
* **Back-End Development (Server-Side):** Deals with the server, database, and application logic. It involves technologies like Node.js, Python, PHP, Ruby, and databases like MySQL, MongoDB.
* **Full-Stack Development:** Involves both front-end and back-end development.

#### ****1.2. The Web Development Process****

* **Planning:** Defining the purpose, audience, and goals of the website.
* **Design:** Creating the visual layout, user interface (UI), and user experience (UX).
* **Development:** Writing code to build the front-end and back-end of the website.
* **Testing:** Ensuring the website functions correctly across different browsers and devices.
* **Deployment:** Publishing the website online.
* **Maintenance:** Regular updates and improvements.

### 2. ****Understanding Front-End vs. Back-End Development****

#### ****2.1. Front-End Development****

Front-end development, also known as client-side development, involves creating everything that users see and interact with in their web browsers. It includes:

* **HTML (Hypertext Markup Language):** The standard markup language for creating web pages. It structures the content on the web.
* **CSS (Cascading Style Sheets):** A language used to style the HTML content, including layouts, colors, and fonts.
* **JavaScript:** A scripting language used to create dynamic content that can change without reloading the page.

**Example:** When you visit a website and see a button that changes color when you hover over it, that’s front-end development at work.

#### ****2.2. Back-End Development****

Back-end development, also known as server-side development, is concerned with what happens behind the scenes. It involves:

* **Server:** A system that processes requests and delivers data to the client (browser).
* **Database:** Stores and retrieves data that is displayed on the website.
* **Application Logic:** The programming that processes inputs, interacts with the database, and returns outputs.

**Example:** When you log into a website and your personal data is retrieved and displayed, that’s back-end development at work.

### 3. ****HTML Basics****

#### ****3.1. What is HTML?****

HTML, or Hypertext Markup Language, is the backbone of any web page. It is used to structure content and define the elements that make up the page, such as headings, paragraphs, links, images, and more.

**Basic Structure of an HTML Document:**

html

Copy code

<!DOCTYPE html><html><head><title>My First Web Page</title></head><body><h1>Hello, World!</h1><p>This is my first web page using HTML.</p></body></html>

* **<!DOCTYPE html>:** This declaration defines the document type and version of HTML.
* **<html>:** The root element that wraps all content on the page.
* **<head>:** Contains meta-information about the HTML document (e.g., title, character set).
* **<title>:** Specifies the title of the web page, displayed on the browser tab.
* **<body>:** Contains the content of the web page, including text, images, and links.

#### ****3.2. Essential HTML Tags****

* **<h1> to <h6>:** Heading tags, with <h1> being the highest level and <h6> the lowest.
* **<p>:** Paragraph tag for blocks of text.
* **<a>:** Anchor tag for creating hyperlinks.
* **<img>:** Image tag to display pictures.
* **<div> and <span>:** Container tags for grouping elements.

**Example:**

html

Copy code

<h2>About Us</h2><p>We are a company dedicated to providing quality services.</p><a href="contact.html">Contact Us</a><img src="company-logo.png" alt="Company Logo">

### 4. ****Setting Up a Development Environment****

#### ****4.1. Choosing a Code Editor****

A code editor is a specialized text editor designed for writing and editing code. Some popular choices include:

* **Visual Studio Code (VS Code):** A free, open-source code editor with a rich ecosystem of extensions.
* **Sublime Text:** A fast, lightweight editor with powerful features.
* **Atom:** A free, open-source editor developed by GitHub.

**Example:** Installing Visual Studio Code:

1. Visit the [Visual Studio Code website](https://code.visualstudio.com/).
2. Download the version appropriate for your operating system.
3. Install the software following the on-screen instructions.

#### ****4.2. Setting Up Your First Project****

1. **Create a Project Folder:** Organize your project files in a dedicated folder.
2. **Create an HTML File:** Open your code editor and create a new file named index.html.
3. **Basic HTML Structure:** Write the basic HTML structure in index.html.
4. **Open in Browser:** Save the file and open it in your web browser to see the results.

**Example:**

html

Copy code

<!DOCTYPE html><html><head><title>My First Web Project</title></head><body><h1>Welcome to My Website</h1><p>This is a simple web page created during Week 1.</p></body></html>

### 5. ****Introduction to CSS****

#### ****5.1. What is CSS?****

CSS, or Cascading Style Sheets, is used to control the layout and appearance of HTML elements on a web page. It allows you to apply styles such as colors, fonts, spacing, and more.

**Example:**

css

Copy code

body {

background-color: #f0f0f0;

font-family: Arial, sans-serif;

}

h1 {

color: #333333;

}

p {

font-size: 16px;

line-height: 1.5;

}

This CSS code changes the background color of the page, sets a font style, and customizes the appearance of headings and paragraphs.

#### ****5.2. Adding CSS to Your Project****

There are three main ways to add CSS to your HTML:

1. **Inline CSS:** Directly within an HTML element using the style attribute.

html

Copy code

<pstyle="color: blue;">This text is blue.</p>

1. **Internal CSS:** Within a <style> tag in the <head> section of the HTML document.

html

Copy code

<head><style>body {

background-color: lightgray;

}

</style></head>

1. **External CSS:** In a separate .css file linked to the HTML document.

html

Copy code

<head><link rel="stylesheet" href="styles.css"></head>

### Conclusion of Week 1

By the end of Week 1, you should have a solid understanding of the basics of web development, including the role of HTML and CSS in building and styling web pages. You’ll have set up your development environment and created your first simple web page. These foundational skills will be crucial as you progress to more complex topics in the coming weeks.

This concludes the lecture for Week 1. As you move forward, remember to experiment with the code, explore additional HTML and CSS tags, and try building different types of web pages to strengthen your understanding.

# Week 2: Introduction to CSS and Responsive Web Design

**Overview:** In Week 2, students will dive deeper into CSS (Cascading Style Sheets) to enhance the look and feel of web pages. This week will cover styling techniques, layout control using CSS, and an introduction to responsive web design to ensure web pages look good on all devices. By the end of this week, students will be able to create visually appealing, well-structured, and responsive web pages.

**Topics Covered:**

1. Definitions and Key Roles
2. Data Types and Collection Methods
3. Data Quality and Cleaning
4. Introduction to Tools and Software
5. Real-World Applications

### 1. ****Introduction to CSS****

#### ****1.1. Understanding CSS****

CSS is a powerful tool that allows you to control the presentation of web pages. While HTML structures content, CSS is used to style that content, enabling you to control colors, fonts, layouts, spacing, and more.

* **Selectors:** Used to target specific HTML elements.
* **Properties:** Define what aspect of the element you want to style (e.g., color, margin).
* **Values:** Assign specific settings to the properties.

**Example:**

css

Copy code

h1 {

color: blue;

font-size: 24px;

}

In this example, h1 is the selector, color and font-size are properties, and blue and 24px are the corresponding values.

#### ****1.2. Types of CSS Selectors****

* **Element Selector:** Targets all instances of a specific HTML element.

css

Copy code

p {

color: green;

}

* **Class Selector:** Targets elements with a specific class attribute.

css

Copy code

.highlight {

background-color: yellow;

}

Applied in HTML: <p class="highlight">This text has a yellow background.</p>

* **ID Selector:** Targets a specific element with a unique ID attribute.

css

Copy code

#header {

font-size: 20px;

}

Applied in HTML: <div id="header">This is the header.</div>

* **Attribute Selector:** Targets elements with a specific attribute.

css

Copy code

input[type="text"] {

border: 1px solid #ccc;

}

* **Group Selector:** Targets multiple elements and applies the same style.

css

Copy code

h1, h2, h3 {

color: red;

}

### 2. ****Styling Text and Fonts****

#### ****2.1. Text Properties****

CSS allows for extensive control over text appearance, including font selection, size, weight, alignment, and more.

* **Font Family:** Specifies the font used for text.

css

Copy code

body {

font-family: 'Arial', sans-serif;

}

* **Font Size:** Sets the size of the font.

css

Copy code

p {

font-size: 16px;

}

* **Font Weight:** Controls the thickness of text.

css

Copy code

h1 {

font-weight: bold;

}

* **Text Alignment:** Aligns text horizontally within its container.

css

Copy code

.centered-text {

text-align: center;

}

* **Line Height:** Adjusts the spacing between lines of text.

css

Copy code

p {

line-height: 1.6;

}

#### ****2.2. Using Web Fonts****

CSS allows you to use custom fonts by importing them from online sources, such as Google Fonts.

css

Copy code

@import url('https://fonts.googleapis.com/css2?family=Roboto:wght@400;700&display=swap');

body {

font-family: 'Roboto', sans-serif;

}

This code imports the "Roboto" font and applies it to the entire page.

### 3. ****CSS Box Model****

#### ****3.1. Understanding the Box Model****

Every element on a web page is a rectangular box, and the CSS box model defines how these boxes are sized and spaced. The box model consists of:

* **Content:** The actual content of the element, such as text or images.
* **Padding:** The space between the content and the border.
* **Border:** The line surrounding the padding (optional).
* **Margin:** The space outside the border, separating the element from others.

**Example:**

css

Copy code

div {

width: 200px;

padding: 10px;

border: 2px solid black;

margin: 20px;

}

Here, the div element has a content area of 200px, 10px of padding, a 2px border, and 20px of margin.

#### ****3.2. Adjusting the Box Model****

* **Padding:**

css

Copy code

p {

padding: 15px;

}

* **Border:**

css

Copy code

img {

border: 5px solid #ccc;

}

* **Margin:**

css

Copy code

h1 {

margin-top: 20px;

margin-bottom: 10px;

}

### 4. ****CSS Layout Techniques****

#### ****4.1. Display Property****

The display property controls how an element is displayed on the web page.

* **Block:** The element takes up the full width available (e.g., <div>, <p>).

css

Copy code

div {

display: block;

}

* **Inline:** The element takes up only as much width as necessary (e.g., <span>, <a>).

css

Copy code

span {

display: inline;

}

* **Inline-Block:** Like inline, but allows setting width and height.

css

Copy code

img {

display: inline-block;

}

#### ****4.2. Positioning Elements****

CSS offers various ways to position elements on the page:

* **Static:** The default position; elements are placed according to the normal flow of the document.

css

Copy code

p {

position: static;

}

* **Relative:** The element is positioned relative to its normal position.

css

Copy code

.relative {

position: relative;

top: 10px;

left: 20px;

}

* **Absolute:** The element is positioned relative to its closest positioned ancestor.

css

Copy code

.absolute {

position: absolute;

top: 50px;

left: 100px;

}

* **Fixed:** The element is positioned relative to the browser window and does not move when scrolling.

css

Copy code

.fixed {

position: fixed;

top: 0;

right: 0;

}

### 5. ****Responsive Web Design****

#### ****5.1. What is Responsive Web Design?****

Responsive web design (RWD) is an approach that ensures web pages render well on a variety of devices and window or screen sizes. A responsive design adjusts its layout and content dynamically, providing an optimal viewing experience whether on a mobile phone, tablet, or desktop computer.

#### ****5.2. Media Queries****

Media queries are used to apply different styles depending on the device's screen size.

css

Copy code

@media (max-width: 768px) {

body {

background-color: lightblue;

}

.container {

width: 100%;

}

}

This media query changes the background color to light blue and makes the container take up 100% of the width when the screen is 768px wide or smaller.

#### ****5.3. Fluid Grids and Flexible Images****

* **Fluid Grids:** Layouts where elements have percentage-based widths instead of fixed pixel values.

css

Copy code

.column {

width: 50%;

}

* **Flexible Images:** Images that scale with the size of the screen.

css

Copy code

img {

max-width: 100%;

height: auto;

}

### 6. ****Hands-On Practice: Creating a Responsive Web Page****

#### ****6.1. Building a Responsive Layout****

Students will create a basic web page that includes a header, navigation menu, main content area, and footer. The layout should be fully responsive, adapting to different screen sizes.

* **Step 1:** Create a basic HTML structure.
* **Step 2:** Apply CSS to style the elements and create a layout using a grid or flexbox.
* **Step 3:** Implement media queries to adjust the layout for different screen sizes.
* **Step 4:** Test the web page on various devices or using browser developer tools to ensure responsiveness.

### Conclusion of Week 2

By the end of Week 2, students will have a strong understanding of CSS and its role in creating visually appealing and responsive web pages. They will be familiar with various CSS properties, the box model, layout techniques, and responsive design principles. The hands-on practice will provide the opportunity to apply these concepts and build a responsive web page that adapts to different devices. This knowledge will be crucial as students move on to more advanced topics in web development.

This concludes the lecture for Week 2. As you progress, continue experimenting with different CSS properties and layouts to refine your design skills and improve your ability to create professional-looking web pages.

# Week 3: Introduction to JavaScript

**Overview:** Week 3 introduces JavaScript, a powerful and essential programming language for creating dynamic and interactive web pages. JavaScript adds behaviour to websites, enabling features like form validation, animations, interactive maps, and much more. By the end of this week, students will have a foundational understanding of JavaScript syntax, basic programming concepts, and how to manipulate the Document Object Model (DOM) to create interactive web experiences.

**Topics Covered:**

1. Definitions and Key Roles
2. Data Types and Collection Methods
3. Data Quality and Cleaning
4. Introduction to Tools and Software
5. Real-World Applications

### 1. ****Introduction to JavaScript****

#### ****1.1. What is JavaScript?****

JavaScript is a high-level, interpreted programming language that is one of the core technologies of the web, alongside HTML and CSS. It enables you to create dynamically updated content, control multimedia, animate images, and much more.

* **JavaScript vs. Java:** Although their names are similar, JavaScript and Java are distinct programming languages with different uses.
* **Client-Side vs. Server-Side:** JavaScript is primarily used on the client side, but it can also be used on the server side with environments like Node.js.

**Example:**

javascript

Copy code

console.log("Hello, World!");

This simple JavaScript code prints "Hello, World!" to the browser’s console.

#### ****1.2. Embedding JavaScript in HTML****

JavaScript can be embedded directly in HTML files or linked as an external script.

* **Inline JavaScript:**

html

Copy code

<script>alert("Welcome to the website!");

</script>

* **External JavaScript:**

html

Copy code

<scriptsrc="script.js"></script>

This method is preferred for keeping HTML and JavaScript code separate.

### 2. ****JavaScript Syntax and Basics****

#### ****2.1. Variables and Data Types****

Variables store data that can be used and manipulated within your program. JavaScript supports various data types, including strings, numbers, booleans, arrays, and objects.

* **Declaring Variables:**

javascript

Copy code

let name = "John";

const pi = 3.14;

var age = 25;

* **Data Types:**

javascript

Copy code

let isStudent = true; // Booleanlet score = 85; // Numberlet firstName = "Jane"; // String

* **Choosing var, let, or const:**
  + let: For variables that may change.
  + const: For variables that should not change.
  + var: The old way of declaring variables (use let or const instead).

#### ****2.2. Operators****

Operators perform operations on variables and values.

* **Arithmetic Operators:**

javascript

Copy code

let sum = 10 + 5; // 15let product = 10 \* 5; // 50

* **Assignment Operators:**

javascript

Copy code

let x = 10;

x += 5; // x is now 15

* **Comparison Operators:**

javascript

Copy code

let isEqual = (10 == "10"); // truelet isIdentical = (10 === "10"); // false

#### ****2.3. Control Structures****

Control structures allow you to control the flow of your program based on conditions and loops.

* **Conditional Statements:**

javascript

Copy code

let age = 18;

if (age >= 18) {

console.log("You are an adult.");

} else {

console.log("You are not an adult.");

}

* **Loops:**
  + **For Loop:**

javascript

Copy code

for (let i = 0; i < 5; i++) {

console.log("Number: " + i);

}

* + **While Loop:**

javascript

Copy code

let i = 0;

while (i < 5) {

console.log("Number: " + i);

i++;

}

### 3. ****Functions****

#### ****3.1. Defining and Calling Functions****

Functions are reusable blocks of code that perform a specific task.

* **Function Declaration:**

javascript

Copy code

functiongreet() {

console.log("Hello, World!");

}

greet(); // Calls the function

* **Function Parameters and Return Values:**

javascript

Copy code

functionadd(a, b) {

return a + b;

}

let result = add(5, 10); // 15

#### ****3.2. Arrow Functions****

Arrow functions provide a shorter syntax for writing functions.

* **Arrow Function Syntax:**

javascript

Copy code

constmultiply = (a, b) => a \* b;

console.log(multiply(2, 3)); // 6

### 4. ****The Document Object Model (DOM)****

#### ****4.1. What is the DOM?****

The Document Object Model (DOM) is a programming interface for web documents. It represents the structure of a document as a tree of objects that can be manipulated with JavaScript.

* **DOM Tree:** Every HTML element in a document becomes a node in the DOM tree.
* **Manipulating the DOM:** JavaScript can add, remove, or modify elements in the DOM.

#### ****4.2. Selecting Elements****

You can select and manipulate HTML elements using various methods.

* **getElementById:**

javascript

Copy code

let element = document.getElementById("header");

* **getElementsByClassName:**

javascript

Copy code

let elements = document.getElementsByClassName("item");

* **querySelector:**

javascript

Copy code

let element = document.querySelector(".menu");

#### ****4.3. Manipulating Elements****

Once selected, you can manipulate elements in various ways.

* **Changing Content:**

javascript

Copy code

let header = document.getElementById("header");

header.textContent = "Welcome!";

* **Changing Styles:**

javascript

Copy code

header.style.color = "blue";

header.style.fontSize = "24px";

* **Adding/Removing Elements:**

javascript

Copy code

let newElement = document.createElement("p");

newElement.textContent = "This is a new paragraph.";

document.body.appendChild(newElement);

### 5. ****Events in JavaScript****

#### ****5.1. What are Events?****

Events are actions or occurrences that happen in the browser (e.g., clicks, keypresses) that JavaScript can respond to.

* **Common Events:**
  + onclick: Triggered when an element is clicked.
  + onmouseover: Triggered when the mouse hovers over an element.
  + onkeyup: Triggered when a key is released.

#### ****5.2. Event Listeners****

You can listen for events and define a response when they occur.

* **Adding Event Listeners:**

javascript

Copy code

let button = document.getElementById("myButton");

button.addEventListener("click", function() {

alert("Button was clicked!");

});

* **Removing Event Listeners:**

javascript

Copy code

button.removeEventListener("click", handleClick);

### 6. ****Hands-On Practice: Building an Interactive Web Page****

#### ****6.1. Creating a Simple Web Page****

Students will build a web page that includes interactive features like buttons, text input fields, and dynamic content updates.

* **Step 1:** Create an HTML structure with headings, paragraphs, and buttons.
* **Step 2:** Write CSS to style the page.
* **Step 3:** Use JavaScript to add interactivity, such as:
  + Displaying alerts when buttons are clicked.
  + Changing text content based on user input.
  + Showing/hiding elements on the page.

### Conclusion of Week 3

By the end of Week 3, students will have a solid understanding of JavaScript fundamentals, including variables, functions, control structures, and DOM manipulation. They will also be familiar with handling events to create interactive web pages. This knowledge will be crucial as they progress to more advanced topics, such as using JavaScript with external libraries and frameworks.

This concludes the lecture for Week 3. Continue to practice JavaScript by building small projects and experimenting with the concepts covered this week. Mastery of JavaScript will provide a strong foundation for more complex web development tasks in the weeks to come.

# Week 4: Advanced JavaScript and Front-End Libraries

**Overview:** Week 4 builds on the foundational JavaScript knowledge acquired in Week 3, delving into advanced concepts that are crucial for modern web development. Students will learn about the latest JavaScript features introduced with ES6, explore asynchronous programming, and get hands-on experience with popular front-end libraries like jQuery and React. By the end of this week, students will be equipped to develop more complex and efficient web applications.

**Topics Covered:**

1. Definitions and Key Roles
2. Data Types and Collection Methods
3. Data Quality and Cleaning
4. Introduction to Tools and Software
5. Real-World Applications

### 1. ****Advanced JavaScript Concepts****

#### ****1.1. ES6 Features (ECMAScript 2015)****

ECMAScript 2015 (ES6) introduced several new features that modernized JavaScript, making it more powerful and easier to use.

* **let and const:**
  + let: Used for variables that can be reassigned.
  + const: Used for variables that are not meant to be reassigned.

javascript

Copy code

let age = 25;

const pi = 3.14;

* **Arrow Functions:**
  + Arrow functions provide a concise syntax for writing functions.

javascript

Copy code

constmultiply = (a, b) => a \* b;

* **Template Literals:**
  + Template literals allow for easier string formatting and multi-line strings.

javascript

Copy code

const name = "John";

const greeting = `Hello, ${name}! Welcome to ES6.`;

* **Destructuring Assignment:**
  + Destructuring allows for unpacking values from arrays or properties from objects into distinct variables.

javascript

Copy code

const person = { name: "Jane", age: 30 };

const { name, age } = person;

* **Default Parameters:**
  + Default parameters allow you to set default values for function parameters.

javascript

Copy code

functiongreet(name = "Guest") {

console.log(`Hello, ${name}!`);

}

* **Spread and Rest Operators:**
  + The spread operator (...) allows you to expand an array or object into individual elements.
  + The rest operator (...) allows you to group the remaining elements into an array.

javascript

Copy code

const numbers = [1, 2, 3];

console.log(...numbers); // Outputs: 1 2 3

#### ****1.2. Asynchronous JavaScript****

Asynchronous programming is essential for handling tasks that take time to complete, such as fetching data from a server. JavaScript provides several ways to handle asynchronous operations.

* **Callbacks:**
  + A callback is a function passed into another function as an argument to be executed later.

javascript

Copy code

functionfetchData(callback) {

setTimeout(() => {

callback("Data received!");

}, 2000);

}

fetchData((message) => {

console.log(message);

});

* **Promises:**
  + Promises provide a more elegant way to handle asynchronous operations by representing the eventual completion (or failure) of an asynchronous operation.

javascript

Copy code

const fetchData = newPromise((resolve, reject) => {

setTimeout(() => {

resolve("Data received!");

}, 2000);

});

fetchData.then((message) => {

console.log(message);

});

* **Async/Await:**
  + async and await keywords simplify the process of writing asynchronous code, making it look like synchronous code.

javascript

Copy code

asyncfunctionfetchData() {

const data = awaitnewPromise((resolve) => {

setTimeout(() => {

resolve("Data received!");

}, 2000);

});

console.log(data);

}

fetchData();

### 2. ****Introduction to jQuery****

#### ****2.1. What is jQuery?****

jQuery is a fast, small, and feature-rich JavaScript library. It simplifies tasks like DOM manipulation, event handling, and animation, making it easier to build dynamic and interactive web pages.

* **Why Use jQuery?**
  + Cross-browser compatibility.
  + Simplifies complex JavaScript tasks.
  + Extensive plugin ecosystem.

#### ****2.2. Basic jQuery Syntax****

jQuery uses a simple and intuitive syntax to perform operations on DOM elements.

* **Selecting Elements:**
  + jQuery selectors are similar to CSS selectors.

javascript

Copy code

$(document).ready(function() {

$("p").click(function() {

$(this).hide();

});

});

* **Event Handling:**
  + jQuery makes it easy to handle events such as clicks, hover, and more.

javascript

Copy code

$("button").click(function() {

alert("Button clicked!");

});

* **DOM Manipulation:**
  + jQuery simplifies DOM manipulation tasks like changing content, adding/removing elements, and more.

javascript

Copy code

$("#myDiv").text("Hello, jQuery!");

#### ****2.3. jQuery Animations****

jQuery provides methods to create animations and effects, enhancing the user experience on your web pages.

* **Basic Animation:**
  + animate(), fadeIn(), fadeOut(), slideUp(), slideDown(), etc.

javascript

Copy code

$("#myDiv").fadeOut();

* **Chaining Animations:**
  + jQuery allows chaining of animations and other methods for cleaner, more readable code.

javascript

Copy code

$("#myDiv").slideUp().slideDown().fadeOut();

### 3. ****Introduction to React****

#### ****3.1. What is React?****

React is a popular JavaScript library for building user interfaces, especially for single-page applications where you need a fast and interactive experience. React allows developers to create large web applications that can update and render efficiently in response to data changes.

* **Why Use React?**
  + Component-based architecture.
  + Virtual DOM for efficient updates.
  + Strong community support and ecosystem.

#### ****3.2. Components and JSX****

* **Components:**
  + React is all about components. Components are the building blocks of a React application. Each component corresponds to a part of the UI.

javascript

Copy code

functionWelcome() {

return<h1>Hello, World!</h1>;

}

* **JSX:**
  + JSX is a syntax extension for JavaScript that looks similar to HTML. It allows you to write HTML elements in JavaScript and place them in the DOM.

javascript

Copy code

const element = <h1>Hello, React!</h1>;

#### ****3.3. Creating Simple React Applications****

* **Rendering Components:**
  + Components can be rendered to the DOM using ReactDOM.render().

javascript

Copy code

ReactDOM.render(

<Welcome />,

document.getElementById('root')

);

* **Props and State:**
  + **Props:** Short for "properties," props allow you to pass data from parent components to child components.
  + **State:** State is a special object that holds data that may change over the lifecycle of a component.

javascript

Copy code

functionGreeting(props) {

return<h1>Hello, {props.name}!</h1>;

}

ReactDOM.render(

<Greetingname="John" />,

document.getElementById('root')

);

* **Handling Events:**
  + React allows you to handle events like clicks, form submissions, etc., in a declarative way.

javascript

Copy code

functionButton() {

functionhandleClick() {

alert('Button clicked!');

}

return (

<buttononClick={handleClick}>

Click me

</button>

);

}

### 4. ****Hands-On Practice: Building a Small Project Using React****

#### ****4.1. Setting Up the Project****

* **Install Node.js and npm:** Ensure you have Node.js and npm installed on your machine.
* **Create a React App:** Use create-react-app to quickly set up a new React project.

bash

Copy code

npx create-react-app my-react-app

cd my-react-app

npm start

#### ****4.2. Developing the Project****

Students will create a simple React application that allows users to input their name and see a personalized greeting. The project will involve:

* **Creating Components:** Break down the UI into reusable components.
* **Managing State:** Use React's state to manage user input and display dynamic content.
* **Handling Events:** Implement event handlers for user interactions.

#### ****4.3. Styling the Application****

* Use CSS to style the React components.
* Explore using a CSS framework like Bootstrap with React.

#### ****4.4. Deploying the Project****

* Deploy the React application using a platform like GitHub Pages, Vercel, or Netlify.

### Conclusion of Week 4

By the end of Week 4, students will have advanced their JavaScript skills, learned to work with asynchronous code, and gained hands-on experience with front-end libraries like jQuery and React. They will have also completed a small React project, preparing them for more complex full-stack development in the upcoming weeks.

This concludes the lecture for Week 4. Practice and exploration of these advanced JavaScript concepts and libraries will solidify your understanding and enable you to build more sophisticated web applications.

# Week 5: Introduction to Back-End Development

**Overview:** Week 5 introduces the fundamentals of back-end development, focusing on server-side concepts, setting up a basic server, and working with databases. Students will gain hands-on experience with Node.js and Express.js to create a functional server and learn how to manage and interact with databases. This week is crucial for understanding how to build the server-side logic that supports dynamic web applications.

**Topics Covered:**

1. Definitions and Key Roles
2. Data Types and Collection Methods
3. Data Quality and Cleaning
4. Introduction to Tools and Software
5. Real-World Applications

### 1. ****Overview of Back-End Development****

#### ****1.1. What is Back-End Development?****

Back-end development refers to the server-side of web development, where the server, database, and application logic reside. Unlike front-end development, which deals with the user interface and experience, back-end development handles data processing, storage, and server-side logic.

* **Key Components:**
  + **Server:** Hosts the web application and processes requests from clients.
  + **Database:** Stores and manages data that the application uses.
  + **Application Logic:** The code that defines how data is processed and handled.

#### ****1.2. Server-Side Concepts****

Understanding server-side concepts is essential for back-end development. This includes knowledge of how servers process requests, manage data, and communicate with front-end applications.

* **HTTP Requests and Responses:**
  + **Requests:** Made by clients (e.g., web browsers) to access resources on the server. Includes methods like GET, POST, PUT, DELETE.
  + **Responses:** Returned by the server with the requested data or status messages (e.g., 200 OK, 404 Not Found).
* **Middleware:**
  + Functions that process requests before they reach the final route handler. They can perform tasks like logging, authentication, or parsing request bodies.
* **Routing:**
  + Determines how server requests are handled based on the URL and HTTP method. Each route corresponds to a specific endpoint or resource.

### 2. ****Setting Up a Server****

#### ****2.1. Introduction to Node.js****

Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine. It enables developers to use JavaScript for server-side programming.

* **Installation:**
  + Download and install Node.js from [nodejs.org](https://nodejs.org/).
  + Verify the installation using node -v and npm -v in the terminal.
* **Basic Node.js Application:**

javascript

Copy code

// app.jsconst http = require('http');

const server = http.createServer((req, res) => {

res.statusCode = 200;

res.setHeader('Content-Type', 'text/plain');

res.end('Hello, World!\n');

});

server.listen(3000, '127.0.0.1', () => {

console.log('Server running at http://127.0.0.1:3000/');

});

* + **Explanation:**
    - http.createServer(): Creates a new HTTP server.
    - req: Represents the incoming request.
    - res: Represents the response that will be sent to the client.
    - server.listen(): Binds the server to a specific port and IP address.

#### ****2.2. Introduction to Express.js****

Express.js is a minimalist web framework for Node.js that simplifies the process of building web applications and APIs.

* **Installation:**
  + Install Express using npm.

bash

Copy code

npm install express

* **Creating a Basic Express Server:**

javascript

Copy code

// app.jsconst express = require('express');

const app = express();

app.get('/', (req, res) => {

res.send('Hello, Express!');

});

app.listen(3000, () => {

console.log('Server running on port 3000');

});

* + **Explanation:**
    - express(): Initializes an Express application.
    - app.get(): Defines a route for handling GET requests to the root URL.
    - app.listen(): Starts the server and listens on the specified port.

### 3. ****Working with Databases****

#### ****3.1. Introduction to Databases****

Databases are essential for storing and managing data used by web applications. They can be categorized into:

* **Relational Databases (SQL):**
  + Use Structured Query Language (SQL) for querying and managing data.
  + Examples: MySQL, PostgreSQL, SQLite.
* **NoSQL Databases:**
  + Designed for unstructured or semi-structured data.
  + Examples: MongoDB, CouchDB.

#### ****3.2. Connecting to a Database with Node.js****

This section focuses on connecting to MongoDB, a popular NoSQL database, using the mongoose library, which provides an elegant API for MongoDB.

* **Installation:**
  + Install mongoose and mongodb packages.

bash

Copy code

npm install mongoose

* **Basic MongoDB Connection with Mongoose:**

javascript

Copy code

const mongoose = require('mongoose');

mongoose.connect('mongodb://localhost/mydatabase', {

useNewUrlParser: true,

useUnifiedTopology: true

});

const db = mongoose.connection;

db.on('error', console.error.bind(console, 'connection error:'));

db.once('open', () => {

console.log('Connected to MongoDB');

});

* + **Explanation:**
    - mongoose.connect(): Connects to the MongoDB server.
    - db.on('error'): Handles connection errors.
    - db.once('open'): Confirms successful connection.
* **Creating a Schema and Model:**

javascript

Copy code

constSchema = mongoose.Schema;

const userSchema = newSchema({

name: String,

email: String,

age: Number

});

constUser = mongoose.model('User', userSchema);

* + **Explanation:**
    - Schema: Defines the structure of the documents in the collection.
    - mongoose.model(): Creates a model for the schema.
* **Performing CRUD Operations:**

javascript

Copy code

// Create a new userconst newUser = newUser({

name: 'Alice',

email: 'alice@example.com',

age: 25

});

newUser.save((err) => {

if (err) returnconsole.error(err);

console.log('User saved');

});

// Find usersUser.find({}, (err, users) => {

if (err) returnconsole.error(err);

console.log(users);

});

* + **Explanation:**
    - new User(): Creates a new document.
    - save(): Saves the document to the database.
    - find(): Retrieves documents from the database.

### 4. ****Hands-On Practice: Building a Simple CRUD Application****

#### ****4.1. Project Setup****

* **Initialize the Project:**
  + Create a new directory for your project and initialize it with npm.

bash

Copy code

mkdir my-backend-project

cd my-backend-project

npm init -y

* **Install Dependencies:**
  + Install Express and Mongoose.

bash

Copy code

npm install express mongoose

#### ****4.2. Creating the Server****

* **Setup Express Server:**

javascript

Copy code

// app.jsconst express = require('express');

const mongoose = require('mongoose');

const app = express();

app.use(express.json());

mongoose.connect('mongodb://localhost/mydatabase', {

useNewUrlParser: true,

useUnifiedTopology: true

});

const userSchema = new mongoose.Schema({

name: String,

email: String,

age: Number

});

constUser = mongoose.model('User', userSchema);

// Create user

app.post('/users', (req, res) => {

const newUser = newUser(req.body);

newUser.save((err, user) => {

if (err) return res.status(500).send(err);

res.status(201).send(user);

});

});

// Read users

app.get('/users', (req, res) => {

User.find({}, (err, users) => {

if (err) return res.status(500).send(err);

res.status(200).send(users);

});

});

// Update user

app.put('/users/:id', (req, res) => {

User.findByIdAndUpdate(req.params.id, req.body, { new: true }, (err, user) => {

if (err) return res.status(500).send(err);

res.status(200).send(user);

});

});

// Delete user

app.delete('/users/:id', (req, res) => {

User.findByIdAndDelete(req.params.id, (err) => {

if (err) return res.status(500).send(err);

res.status(204).send();

});

});

app.listen(3000, () => {

console.log('Server running on port 3000');

});

#### ****4.3. Testing the Application****

* **Using Postman:**
  + Test CRUD operations by sending HTTP requests to your API endpoints using Postman or any other API testing tool.
* **Verifying Functionality:**
  + Ensure that all routes are functioning correctly and that data is being stored, retrieved, updated, and deleted as expected.

### Conclusion of Week 5

By the end of Week 5, students will have a solid understanding of back-end development concepts, including server setup,

# Week 6: Advanced Back-End Development

**Overview:** Week 6 delves into more advanced back-end development topics, focusing on user authentication, RESTful API development, and security practices. Students will learn to implement user authentication, create RESTful APIs with Express.js, and apply essential error handling and security measures. This week is designed to build upon the basic server knowledge gained in Week 5 and prepare students for real-world back-end development challenges.

**Topics Covered:**

1. Definitions and Key Roles
2. Data Types and Collection Methods
3. Data Quality and Cleaning
4. Introduction to Tools and Software
5. Real-World Applications

### 1. ****User Authentication****

#### ****1.1. Implementing User Registration and Login Functionality****

User authentication is crucial for securing web applications. This involves creating user accounts, managing sessions, and verifying user credentials.

* **Setting Up Authentication Middleware:**
  + Middleware functions are used to handle authentication logic before requests reach the final route handler.

javascript

Copy code

const jwt = require('jsonwebtoken');

constauthenticateToken = (req, res, next) => {

const authHeader = req.headers['authorization'];

const token = authHeader && authHeader.split(' ')[1];

if (token == null) return res.sendStatus(401);

jwt.verify(token, process.env.ACCESS\_TOKEN\_SECRET, (err, user) => {

if (err) return res.sendStatus(403);

req.user = user;

next();

});

};

* + **Explanation:**
    - **jwt.verify():** Verifies the JWT token against the secret key. If valid, it attaches the decoded user information to the request object.
* **User Registration:**
  + **Hashing Passwords:**
    - Use bcrypt to hash passwords before storing them in the database.

javascript

Copy code

const bcrypt = require('bcrypt');

const saltRounds = 10;

app.post('/register', async (req, res) => {

try {

const hashedPassword = await bcrypt.hash(req.body.password, saltRounds);

const user = newUser({

name: req.body.name,

email: req.body.email,

password: hashedPassword

});

await user.save();

res.status(201).send('User registered');

} catch {

res.status(500).send('Error registering user');

}

});

* + **Explanation:**
    - **bcrypt.hash():** Hashes the user's password with a salt to ensure that passwords are not stored in plaintext.
* **User Login:**
  + **Generating JWT Tokens:**
    - Upon successful login, generate a JWT token for the user to authenticate future requests.

javascript

Copy code

app.post('/login', async (req, res) => {

const user = awaitUser.findOne({ email: req.body.email });

if (user == null) return res.status(400).send('Cannot find user');

try {

if (await bcrypt.compare(req.body.password, user.password)) {

const accessToken = jwt.sign({ name: user.name }, process.env.ACCESS\_TOKEN\_SECRET);

res.json({ accessToken });

} else {

res.status(400).send('Invalid credentials');

}

} catch {

res.status(500).send('Error logging in');

}

});

* + **Explanation:**
    - **bcrypt.compare():** Compares the provided password with the stored hashed password.
    - **jwt.sign():** Creates a JWT token with user information.

#### ****1.2. Managing Sessions and Cookies****

Sessions and cookies are used to maintain user state across multiple requests.

* **Creating Sessions:**
  + Use express-session to manage user sessions.

javascript

Copy code

const session = require('express-session');

app.use(session({

secret: 'your-secret-key',

resave: false,

saveUninitialized: true,

cookie: { secure: false }

}));

app.get('/session-test', (req, res) => {

if (req.session.views) {

req.session.views++;

res.send(`Views: ${req.session.views}`);

} else {

req.session.views = 1;

res.send('Welcome to the session test');

}

});

* + **Explanation:**
    - **express-session:** Middleware to handle sessions. It uses cookies to store the session ID on the client.
* **Using Cookies for Authentication:**
  + Store JWT tokens in cookies for persistent authentication.

javascript

Copy code

app.post('/login', async (req, res) => {

const user = awaitUser.findOne({ email: req.body.email });

if (user == null) return res.status(400).send('Cannot find user');

try {

if (await bcrypt.compare(req.body.password, user.password)) {

const accessToken = jwt.sign({ name: user.name }, process.env.ACCESS\_TOKEN\_SECRET);

res.cookie('token', accessToken, { httpOnly: true });

res.status(200).send('Logged in');

} else {

res.status(400).send('Invalid credentials');

}

} catch {

res.status(500).send('Error logging in');

}

});

* + **Explanation:**
    - **res.cookie():** Sets a cookie with the JWT token to manage user sessions.

### 2. ****RESTful API Development****

#### ****2.1. Creating RESTful APIs with Express.js****

RESTful APIs are a standard way to interact with server-side resources using HTTP methods.

* **Designing RESTful Endpoints:**
  + Use different HTTP methods to perform CRUD operations.

javascript

Copy code

// Create user

app.post('/api/users', (req, res) => {

const user = newUser(req.body);

user.save()

.then(() => res.status(201).send(user))

.catch((err) => res.status(500).send(err));

});

// Read all users

app.get('/api/users', (req, res) => {

User.find()

.then((users) => res.status(200).send(users))

.catch((err) => res.status(500).send(err));

});

// Read a single user

app.get('/api/users/:id', (req, res) => {

User.findById(req.params.id)

.then((user) => {

if (!user) return res.status(404).send('User not found');

res.status(200).send(user);

})

.catch((err) => res.status(500).send(err));

});

// Update user

app.put('/api/users/:id', (req, res) => {

User.findByIdAndUpdate(req.params.id, req.body, { new: true })

.then((user) => {

if (!user) return res.status(404).send('User not found');

res.status(200).send(user);

})

.catch((err) => res.status(500).send(err));

});

// Delete user

app.delete('/api/users/:id', (req, res) => {

User.findByIdAndDelete(req.params.id)

.then((user) => {

if (!user) return res.status(404).send('User not found');

res.status(204).send();

})

.catch((err) => res.status(500).send(err));

});

* + **Explanation:**
    - **app.post():** Handles the creation of new resources.
    - **app.get():** Retrieves resources.
    - **app.put():** Updates existing resources.
    - **app.delete():** Deletes resources.
* **Handling Different HTTP Methods:**
  + Understand how each method is used to perform specific operations and design APIs accordingly.

### 3. ****Error Handling and Security****

#### ****3.1. Implementing Error Handling in Node.js****

Effective error handling ensures that your application can gracefully handle unexpected situations.

* **Basic Error Handling:**

javascript

Copy code

app.use((err, req, res, next) => {

console.error(err.stack);

res.status(500).send('Something broke!');

});

* + **Explanation:**
    - **Error-handling middleware:** Catches errors and sends an appropriate response.
* **Handling Specific Errors:**
  + Implement custom error handling for different types of errors (e.g., 404 Not Found, 400 Bad Request).

javascript

Copy code

app.use((req, res, next) => {

res.status(404).send('Page not found');

});

* + **Explanation:**
    - **404 Error:** Handles requests to non-existent routes.

#### ****3.2. Basic Security Practices for Web Applications****

Security is a vital aspect of back-end development to protect your application and its data.

* **Preventing SQL Injection:**
  + Use parameterized queries or ORM methods to avoid SQL injection attacks.
* **Implementing HTTPS:**
  + Use HTTPS to encrypt data transmitted between the server and clients.
* **Securing HTTP Headers:**
  + Use helmet to set various HTTP headers to improve security.

javascript

Copy code

const helmet = require('helmet');

app.use(helmet());

* + **Explanation:**
    - **helmet():** Adds security-related HTTP headers to your Express application.
* **Protecting Against Cross-Site Scripting (XSS) and Cross-Site Request Forgery (CSRF):**
  + **XSS Protection:** Sanitize user inputs and use libraries like xss-clean.
  + **CSRF Protection:** Use libraries like csurf to protect against CSRF attacks.

javascript

Copy code

const csurf = require('csurf');

app

#### ****3.2. Basic Security Practices for Web Applications**** (Continued)

* **Protecting Against Cross-Site Scripting (XSS) and Cross-Site Request Forgery (CSRF):**
  + **XSS Protection:**
    - **Sanitize User Inputs:** Use libraries such as xss-clean to remove any harmful scripts from user inputs.

javascript

Copy code

const xss = require('xss-clean');

app.use(xss());

* + - **Explanation:**
      * **xss-clean():** Middleware that helps prevent XSS attacks by sanitizing user inputs.
  + **CSRF Protection:**
    - **Using csurf Middleware:** Implement CSRF protection by requiring a CSRF token for state-changing requests.

javascript

Copy code

const csrf = require('csurf');

const csrfProtection = csrf({ cookie: true });

app.use(csrfProtection);

app.get('/form', (req, res) => {

res.render('form', { csrfToken: req.csrfToken() });

});

* + - **Explanation:**
      * **csrf():** Middleware that provides CSRF protection by generating and validating tokens.

### ****4. Testing and Debugging****

#### ****4.1. Unit Testing****

Unit testing ensures that individual components of your application work as expected.

* **Setting Up Testing Frameworks:**
  + **Mocha and Chai:** Use Mocha as the test framework and Chai for assertions.

javascript

Copy code

const chai = require('chai');

const expect = chai.expect;

describe('User Authentication', function() {

it('should register a new user', asyncfunction() {

const res = awaitrequest(app).post('/register')

.send({ name: 'John Doe', email: 'john@example.com', password: 'password123' });

expect(res.status).to.equal(201);

});

});

* + **Explanation:**
    - **request(app):** Simulates HTTP requests to the Express app for testing.
    - **expect():** Provides assertions to verify test outcomes.

#### ****4.2. Integration Testing****

Integration tests ensure that different parts of your application work together correctly.

* **Testing API Endpoints:**
  + Test that your RESTful API endpoints function correctly and handle edge cases.

javascript

Copy code

describe('API Endpoints', function() {

it('should return all users', asyncfunction() {

const res = awaitrequest(app).get('/api/users');

expect(res.status).to.equal(200);

expect(res.body).to.be.an('array');

});

});

* + **Explanation:**
    - **Testing with request(app).get()**: Simulates GET requests to the API endpoint and checks the response.

#### ****4.3. Debugging Techniques****

Effective debugging helps identify and fix issues in your application.

* **Using console.log():**
  + Log important information to the console for debugging.

javascript

Copy code

app.post('/login', async (req, res) => {

console.log('Login attempt:', req.body);

// Further login logic...

});

* + **Explanation:**
    - **console.log():** Useful for tracing execution flow and checking variable values.
* **Using Debugging Tools:**
  + **Node.js Inspector:** Use the built-in Node.js inspector for advanced debugging.

bash

Copy code

node --inspect server.js

* + **Explanation:**
    - **--inspect:** Starts the Node.js application with debugging enabled.

### ****5. Deployment and Environment Configuration****

#### ****5.1. Environment Variables****

Environment variables store sensitive information and configuration settings.

* **Using dotenv to Manage Environment Variables:**

javascript

Copy code

require('dotenv').config();

console.log(process.env.DATABASE\_URL);

* + **Explanation:**
    - **dotenv:** Loads environment variables from a .env file into process.env.
* **Creating a .env File:**
  + Store sensitive information like API keys and database URLs in a .env file.

makefile

Copy code

DATABASE\_URL=mongodb://localhost/mydatabase

ACCESS\_TOKEN\_SECRET=mysecrettoken

#### ****5.2. Deployment Strategies****

* **Deploying to Cloud Services:**
  + **Heroku, AWS, or Vercel:** Platforms for deploying Node.js applications.
  + **Heroku Deployment Example:**
    - Set up Heroku CLI and deploy using Git.

bash

Copy code

heroku create myapp

git push heroku main

* + - **Explanation:**
      * **heroku create:** Creates a new Heroku app.
      * **git push heroku main:** Deploys the application to Heroku.
* **Setting Up CI/CD Pipelines:**
  + Use CI/CD tools like GitHub Actions or Travis CI to automate testing and deployment.

yaml

Copy code

# .github/workflows/deploy.ymlname:Deployon:push:branches:-mainjobs:deploy:runs-on:ubuntu-lateststeps:-uses:actions/checkout@v2-name:DeploytoHerokurun:|

git remote add heroku https://git.heroku.com/myapp.git

git push heroku main

env:HEROKU\_API\_KEY:${{secrets.HEROKU\_API\_KEY}}

* + **Explanation:**
    - **GitHub Actions:** Automates deployment workflows based on repository changes.

### ****6. Practical Exercises and Projects****

#### ****6.1. Building a User Management System****

* **Project Overview:**
  + Create a user management system with features like registration, login, and profile management.
* **Project Requirements:**
  + Implement user authentication.
  + Create RESTful endpoints for user operations.
  + Add basic security features.
* **Project Tasks:**
  + Develop user registration and login functionality.
  + Build RESTful API endpoints for managing user data.
  + Secure the application with JWT tokens and HTTPS.

#### ****6.2. API Integration Project****

* **Project Overview:**
  + Integrate third-party APIs into your application, such as weather or social media APIs.
* **Project Requirements:**
  + Fetch data from an external API.
  + Handle API responses and errors.
  + Display data in your application.
* **Project Tasks:**
  + Set up API integration using tools like Axios.
  + Handle API responses and display data to users.

### ****7. Review and Q&A****

* **Review Key Concepts:**
  + Recap user authentication, RESTful API development, and security practices.
  + Discuss best practices for error handling and testing.
* **Q&A Session:**
  + Address any questions or concerns from students.
  + Provide additional resources for further learning.

### ****8. Resources and Further Reading****

* **Books:**
  + "Node.js Design Patterns" by Mario Casciaro
  + "Web Application Security" by Andrew Hoffman
* **Online Resources:**
  + [MDN Web Docs - HTTP](https://developer.mozilla.org/en-US/docs/Web/HTTP)
  + [Express.js Documentation](https://expressjs.com/)
  + JWT Authentication Guide
* **Tools and Libraries:**
  + **bcrypt:**[bcrypt GitHub](https://github.com/kelektiv/node.bcrypt.js)
  + **jsonwebtoken:**[jsonwebtoken GitHub](https://github.com/auth0/node-jsonwebtoken)
  + **xss-clean:**[xss-clean GitHub](https://github.com/jsonmaur/xss-clean)
  + **csurf:**[csurf GitHub](https://github.com/expressjs/csurf)
  + **dotenv:**[dotenv GitHub](https://github.com/motdotla/dotenv)
  + **Mocha & Chai:**[Mocha](https://mochajs.org/) | [Chai](https://www.chaijs.com/)

By the end of Week 6, students should have a comprehensive understanding of advanced back-end development concepts, including user authentication, RESTful API creation, error handling, and security practices. They will also have practical experience through projects that demonstrate their ability to implement these concepts effectively.

# ****Week 7: Full-Stack Development****

**Overview:** Week 7 focuses on integrating the front-end and back-end skills you've learned, deploying applications, and managing code with version control. This week provides a comprehensive overview of full-stack development, emphasizing how different components work together in a real-world application. By the end of this week, you'll have a solid foundation in connecting various parts of a web application and deploying it to a live environment.

**Topics Covered:**

1. Definitions and Key Roles
2. Data Types and Collection Methods
3. Data Quality and Cleaning
4. Introduction to Tools and Software
5. Real-World Applications

#### ****1. Integrating Front-End and Back-End****

**1.1. Connecting a React Front-End with a Node.js Back-End**

Integrating a React front-end with a Node.js back-end involves setting up communication between these two layers of your application. Here's how to achieve this:

* **Understanding the Basics of API Integration:**
  + **API Endpoints:** Your Node.js back-end exposes API endpoints that the React front-end can call to retrieve or send data. These endpoints are typically defined in routes in your Express application.

javascript

Copy code

// Node.js (Express) - server.js

app.get('/api/users', async (req, res) => {

const users = awaitUser.find();

res.json(users);

});

* + - **Explanation:**
      * **/api/users:** An endpoint that returns a list of users in JSON format.
      * **res.json(users):** Sends the user data as a JSON response to the front-end.
  + **Making API Requests in React:**
    - Use fetch() or libraries like Axios to make HTTP requests from your React components.

javascript

Copy code

// React - UserList.jsimportReact, { useState, useEffect } from'react';

import axios from'axios';

functionUserList() {

const [users, setUsers] = useState([]);

useEffect(() => {

axios.get('/api/users')

.then(response => {

setUsers(response.data);

})

.catch(error => {

console.error('There was an error fetching the users!', error);

});

}, []);

return (

<ul>

{users.map(user => (

<likey={user.id}>{user.name}</li>

))}

</ul>

);

}

exportdefaultUserList;

* + - **Explanation:**
      * **useEffect():** Fetches data from the API when the component mounts.
      * **axios.get():** Sends a GET request to the back-end and updates the component state with the retrieved data.

**1.2. Fetching Data from APIs and Displaying it in the UI**

* **Handling API Responses:**
  + **Parsing JSON Responses:**
    - Use JavaScript’s .json() method to parse JSON responses from the API.

javascript

Copy code

fetch('/api/data')

.then(response => response.json())

.then(data =>console.log(data));

* + **Error Handling:**
    - Implement error handling to manage scenarios where API requests fail.

javascript

Copy code

fetch('/api/data')

.then(response => {

if (!response.ok) {

thrownewError('Network response was not ok');

}

return response.json();

})

.then(data =>console.log(data))

.catch(error =>console.error('There was a problem with the fetch operation:', error));

* **Dynamic UI Updates:**
  + **Updating React State:**
    - Use useState to manage state in your React components and dynamically update the UI based on API data.

javascript

Copy code

const [data, setData] = useState([]);

useEffect(() => {

fetch('/api/data')

.then(response => response.json())

.then(data =>setData(data));

}, []);

* + - **Explanation:**
      * **setData(data):** Updates the component state with the data fetched from the API, triggering a re-render.

#### ****2. Deployment****

**2.1. Introduction to Web Hosting and Cloud Services**

* **Web Hosting Basics:**
  + **What is Web Hosting?**
    - Web hosting services provide the infrastructure and technologies required to store, serve, and manage your website on the internet.
    - **Types of Web Hosting:**
      * **Shared Hosting:** Multiple websites share the same server resources.
      * **VPS Hosting:** Virtual Private Server with dedicated resources.
      * **Dedicated Hosting:** Entire server dedicated to a single website.
      * **Cloud Hosting:** Scalable hosting using cloud infrastructure.
* **Choosing a Hosting Provider:**
  + Popular choices include Heroku, AWS (Amazon Web Services), and Vercel.

**2.2. Deploying a Full-Stack Application to a Platform Like Heroku**

* **Deploying to Heroku:**
  + **Setting Up Heroku CLI:**
    - **Install the Heroku CLI:** Follow the installation guide from Heroku's documentation.

bash

Copy code

curl https://cli-assets.heroku.com/install.sh | sh

* + **Creating a New Heroku App:**
    - **Log in to Heroku:**

bash

Copy code

heroku login

* + - **Create a New App:**

bash

Copy code

heroku create myapp

* + - **Explanation:**
      * **heroku create:** Creates a new Heroku app with a unique name.
  + **Deploying Your Code:**
    - **Prepare Your Application:**
      * Ensure that your package.json includes scripts for starting the application.

json

Copy code

"scripts":{"start":"node server.js"}

* + - **Deploy Code with Git:**

bash

Copy code

git add .

git commit -m "Deploy to Heroku"

git push heroku main

* + - **Explanation:**
      * **git push heroku main:** Pushes your code to the Heroku remote repository for deployment.
* **Managing and Scaling Your Application:**
  + **View Logs:**

bash

Copy code

heroku logs --tail

* + **Scale Dynos:**

bash

Copy code

heroku ps:scale web=1

* + - **Explanation:**
      * **heroku ps:scale:** Adjusts the number of web dynos running your application.

**2.3. Using Continuous Integration/Continuous Deployment (CI/CD) Tools**

* **Introduction to CI/CD:**
  + CI/CD automates the process of testing, building, and deploying your application.
  + **Setting Up CI/CD with GitHub Actions:**
    - **Create a GitHub Actions Workflow:**

yaml

Copy code

# .github/workflows/deploy.ymlname:Deployon:push:branches:-mainjobs:deploy:runs-on:ubuntu-lateststeps:-uses:actions/checkout@v2-name:SetupNode.jsuses:actions/setup-node@v2with:node-version:'14'-run:npminstall-run:npmtest-name:DeploytoHerokurun:gitpushherokumainenv:HEROKU\_API\_KEY:${{secrets.HEROKU\_API\_KEY}}

* + - **Explanation:**
      * **GitHub Actions Workflow:** Automates the process of deploying your app to Heroku upon code changes.

#### ****3. Version Control with Git****

**3.1. Basics of Git and GitHub**

* **Introduction to Git:**
  + Git is a distributed version control system that tracks changes in your source code.
  + **Basic Git Commands:**
    - **Initialize a Git Repository:**

bash

Copy code

git init

* + - **Add Files to Staging:**

bash

Copy code

git add .

* + - **Commit Changes:**

bash

Copy code

git commit -m "Initial commit"

* + - **Push to Remote Repository:**

bash

Copy code

git push origin main

* **Introduction to GitHub:**
  + GitHub is a platform for hosting and managing Git repositories.
  + **Creating a Repository on GitHub:**
    - **Sign in to GitHub** and create a new repository from the GitHub dashboard.
  + **Connecting Your Local Repository to GitHub:**

bash

Copy code

git remote add origin https://github.com/username/repository.git

git push -u origin main

* + - **Explanation:**
      * **git remote add origin:** Links your local repository to GitHub.

**3.2. Collaborative Development with Git**

* **Branching and Merging:**
  + **Create a Branch:**

bash

Copy code

git checkout -b feature-branch

* + **Merge Branches:**

bash

Copy code

git checkout main

git merge feature-branch

* + **Resolve Conflicts:**
    - When merging, if conflicts arise, Git will highlight them. Edit the conflicting files to resolve the conflicts and commit the changes.
* **Pull Requests:**
  + **Creating a Pull Request:**
    - **Push your branch** to GitHub and open a pull request from the GitHub interface.
  + **Review and Merge:**
    - **Review changes** and **merge** the pull request once approved.

#### ****4. Practical Exercises and Projects****

**4.1. Full-Stack Application Integration Project**

**Objective:** Build a full-stack application that integrates the front-end with the back-end, demonstrating your ability to connect React components with Node.js services and deploy the entire application.

**Step-by-Step Instructions:**

1. **Project Setup:**
   * **Create a New Repository:**
     + Initialize a new Git repository for your project.

bash

Copy code

git init

* + - Create a new repository on GitHub and link it to your local repository.

bash

Copy code

git remote add origin https://github.com/username/full-stack-app.git

1. **Back-End Setup:**
   * **Set Up Node.js and Express:**
     + Create a new directory for the back-end and navigate into it.

bash

Copy code

mkdir backend

cd backend

* + - Initialize a new Node.js project.

bash

Copy code

npm init -y

* + - Install Express and other dependencies.

bash

Copy code

npm install express mongoose cors

* + - Create a basic Express server (server.js):

javascript

Copy code

// backend/server.jsconst express = require('express');

const mongoose = require('mongoose');

const cors = require('cors');

const app = express();

constPORT = process.env.PORT || 5000;

app.use(cors());

app.use(express.json());

// Connect to MongoDB

mongoose.connect('mongodb://localhost:27017/fullstackapp', {

useNewUrlParser: true,

useUnifiedTopology: true,

});

app.get('/api/items', (req, res) => {

// Fetch items from the database

res.json([{ id: 1, name: 'Item 1' }]);

});

app.listen(PORT, () =>console.log(`Server running on port ${PORT}`));

1. **Front-End Setup:**
   * **Create a New React App:**
     + Navigate back to the root directory and create a React app.

bash

Copy code

npx create-react-app frontend

cd frontend

* + - Install Axios for making API requests.

bash

Copy code

npm install axios

* + - Create a React component to fetch and display data (App.js):

javascript

Copy code

// frontend/src/App.jsimportReact, { useState, useEffect } from'react';

import axios from'axios';

functionApp() {

const [items, setItems] = useState([]);

useEffect(() => {

axios.get('http://localhost:5000/api/items')

.then(response =>setItems(response.data))

.catch(error =>console.error('There was an error!', error));

}, []);

return (

<divclassName="App"><h1>Items List</h1><ul>

{items.map(item => (

<likey={item.id}>{item.name}</li>

))}

</ul></div>

);

}

exportdefaultApp;

1. **Integrating Front-End and Back-End:**
   * **Proxy Setup:**
     + Configure the React app to proxy requests to the Express server. Add a proxy entry to frontend/package.json:

json

Copy code

"proxy":"http://localhost:5000"

* + **Run Both Servers:**
    - Open two terminal windows. In the first, run the back-end server:

bash

Copy code

cd backend

node server.js

* + - In the second, run the React development server:

bash

Copy code

cd frontend

npm start

1. **Deployment:**
   * **Deploy to Heroku:**
     + Prepare your app for deployment by creating a Procfile in the root directory.

bash

Copy code

web: node backend/server.js

* + - Commit and push your changes to GitHub:

bash

Copy code

git add .

git commit -m "Setup full-stack app"

git push origin main

* + - Deploy to Heroku:

bash

Copy code

heroku create

git push heroku main

* + - Set up build and start scripts in the package.json files as needed for Heroku deployment.

1. **Testing and Debugging:**
   * **Test Endpoints:**
     + Ensure that your API endpoints are working correctly by using Postman or similar tools to verify responses.
   * **Debug Errors:**
     + Use browser developer tools and server logs to debug any issues that arise during development.

#### ****5. Review and Next Steps****

**5.1. Review Key Concepts:**

* **Integration:**
  + How front-end and back-end components interact through APIs.
  + Using React for dynamic front-end development and Express for server-side logic.
* **Deployment:**
  + Deploying a full-stack application to a cloud platform like Heroku.
  + Managing application environments and scaling as needed.

**5.2. Career Advice and Further Learning Resources:**

* **Building a Portfolio:**
  + Showcase your projects on GitHub or a personal website.
  + Highlight your full-stack skills through practical projects and detailed documentation.
* **Continuous Learning:**
  + Explore advanced topics like serverless architecture, microservices, and GraphQL.
  + Stay updated with the latest technologies and best practices in web development.
* **Networking:**
  + Join developer communities and forums to connect with other professionals and stay informed about industry trends.

This comprehensive approach for Week 7 ensures that novices not only understand how to integrate various technologies but also gain practical experience through hands-on projects and deployment.

# ****Week 8: Capstone Project and Review****

**Overview:** This final week focuses on synthesizing all the skills and knowledge acquired throughout the course by working on a comprehensive capstone project. The aim is to integrate both front-end and back-end technologies to create a fully functional web application. The week concludes with a project presentation, review, and guidance on next steps for career development.

**Topics Covered:**

1. Definitions and Key Roles
2. Data Types and Collection Methods
3. Data Quality and Cleaning
4. Introduction to Tools and Software
5. Real-World Applications

### ****1. Capstone Project Development****

**1.1. Project Overview:**

* **Objective:** Develop a fully functional web application that incorporates front-end and back-end technologies learned during the course. The project should demonstrate the integration of React for the front-end and Node.js/Express for the back-end, with a focus on real-world application and deployment.
* **Project Ideas:**
  + **Task Management App:** A web application that allows users to create, update, and manage tasks.
  + **E-Commerce Site:** An online store with product listings, a shopping cart, and user authentication.
  + **Blog Platform:** A platform where users can create, edit, and comment on blog posts.

**1.2. Steps for Development:**

1. **Planning and Design:**
   * **Define Requirements:**
     + Create a list of features and functionalities your application will have.
     + Outline the user flow and wireframes for the application.
   * **Set Up a Project Repository:**
     + Initialize a Git repository for your project.

bash

Copy code

git init

* + - Create a new repository on GitHub and push your initial setup.

bash

Copy code

git remote add origin https://github.com/username/capstone-project.git

git push -u origin master

1. **Back-End Development:**
   * **Create the Server:**
     + Set up an Express server and define your routes and controllers.
     + Connect to a database (e.g., MongoDB) and define schemas and models.

javascript

Copy code

// server.jsconst express = require('express');

const mongoose = require('mongoose');

const cors = require('cors');

const app = express();

constPORT = process.env.PORT || 5000;

app.use(cors());

app.use(express.json());

mongoose.connect('mongodb://localhost:27017/capstone', {

useNewUrlParser: true,

useUnifiedTopology: true,

});

const itemSchema = new mongoose.Schema({

name: String,

description: String,

completed: Boolean,

});

constItem = mongoose.model('Item', itemSchema);

app.get('/api/items', async (req, res) => {

const items = awaitItem.find();

res.json(items);

});

app.post('/api/items', async (req, res) => {

const newItem = newItem(req.body);

await newItem.save();

res.json(newItem);

});

app.listen(PORT, () =>console.log(`Server running on port ${PORT}`));

* + **Implement Authentication (if needed):**
    - Add routes and middleware for user authentication and authorization.
    - Utilize libraries like jsonwebtoken for token-based authentication.

1. **Front-End Development:**
   * **Create React Components:**
     + Develop React components that interact with the back-end API.
     + Implement state management and data fetching.

javascript

Copy code

// App.jsimportReact, { useState, useEffect } from'react';

import axios from'axios';

functionApp() {

const [items, setItems] = useState([]);

useEffect(() => {

axios.get('/api/items')

.then(response =>setItems(response.data))

.catch(error =>console.error('There was an error!', error));

}, []);

return (

<divclassName="App"><h1>Item List</h1><ul>

{items.map(item => (

<likey={item.\_id}>{item.name}: {item.description}</li>

))}

</ul></div>

);

}

exportdefaultApp;

* + **Style the Application:**
    - Use CSS or CSS frameworks like Bootstrap to style your application and ensure it is responsive.

1. **Deployment:**
   * **Prepare for Deployment:**
     + Ensure your project is configured correctly for deployment.
     + Create build scripts and set environment variables.
   * **Deploy to Heroku or Similar Platform:**
     + Follow deployment steps similar to those from Week 7.

bash

Copy code

heroku create

git push heroku master

* + **Verify Deployment:**
    - Check the deployed application for any issues and ensure all functionalities are working as expected.

### ****2. Project Presentation and Review****

**2.1. Presentation Preparation:**

* **Create a Presentation:**
  + Prepare a slide deck that outlines the project goals, features, technologies used, and a live demo or screenshots of the application.
  + Include a brief overview of the development process and any challenges faced.
* **Prepare for Q&A:**
  + Anticipate questions about your project’s design, implementation, and future improvements.

**2.2. Conducting the Presentation:**

* **Present Your Project:**
  + Walk through the features and functionalities of your application.
  + Demonstrate how the front-end interacts with the back-end.
  + Discuss any unique or challenging aspects of the project.
* **Receive Feedback:**
  + Collect feedback from peers and instructors.
  + Take notes on areas for improvement and suggestions for future development.

### ****3. Final Review and Next Steps****

**3.1. Review Key Concepts:**

* **Recap:**
  + Review the core concepts and technologies covered in the course.
  + Reflect on how these concepts were applied in your capstone project.
* **Skills Assessment:**
  + Evaluate your proficiency in both front-end and back-end development.
  + Identify areas where you may need further practice or study.

**3.2. Career Advice and Further Learning:**

* **Building a Portfolio:**
  + Include your capstone project in your portfolio.
  + Showcase other projects and skills relevant to web development.
* **Exploring Career Opportunities:**
  + Research job roles such as Full-Stack Developer, Front-End Developer, Back-End Developer, and other related positions.
  + Network with professionals and join web development communities.
* **Continuing Education:**
  + Explore advanced topics such as cloud computing, DevOps, or data science.
  + Consider pursuing additional certifications or online courses to enhance your skills.

This comprehensive approach to Week 8 ensures that learners not only complete a significant project but also reflect on their learning journey, receive valuable feedback, and are guided toward further development opportunities.