RQF LEVEL 5



SWDFA501

SOFTWARE DEVELOPMENT

Frontend Application Development With React.JS

TRAINEE'S MANUAL

October, 2024

FRONTEND APPLICATION DEVELOPMENT WITH REACT.JS

KOICA Korea International Cooperation Agency

TQUM TVET Quality Management Project

2024

AUTHOR'S NOTE PAGE (COPYRIGHT)

The competent development body of this manual is Rwanda TVET Board ©, reproduce with permission.

All rights reserved.

- This work has been produced initially with the Rwanda TVET Board with the support from KOICA throughTQUM Project
- This work has copyright, but permission is given to all the Administrative and Academic Staff of the RTB and TVET Schools to make copies by photocopying or other duplicating processes for use at their own workplaces.
- This permission does not extend to making of copies for use outside the immediate environment for which they are made, nor making copies for hire or resale to third parties.
- The views expressed in this version of the work do not necessarily represent the views of RTB. The competent body does not give warranty nor accept any liability
- RTB owns the copyright to the trainee and trainer's manuals. Training providers may reproduce these training manuals in part or in full for training purposes only. Acknowledgment of RTB copyright must be included on any reproductions. Any other use of the manuals must be referred to the RTB.

© Rwanda TVET Board

Copies available from:

- *HQs: Rwanda TVET Board-RTB*
- *Web: www.rtb.gov.rw*
- **KIGALI-RWANDA**

Original published version: October 2024

ACKNOWLEDGEMENTS

The publisher would like to thank the following for their assistance in the elaboration of this training manual:

The publisher would like to thank the following for their assistance in the elaboration of this training manual:

Rwanda TVET Board (RTB) extends its appreciation to all parties who contributed to the development of the trainer's and trainee's manuals for the TVET Certificate V in Software development, specifically for the module "SWDFA501: Frontend Application Development with React.JS."

We extend our gratitude to KOICA Rwanda for its contribution to the development of these training manuals and for its ongoing support of the TVET system in Rwanda

We extend our gratitude to the TQUM Project for its financial and technical support in the development of these training manuals.

We would also like to acknowledge the valuable contributions of all TVET trainers and industry practitioners in the development of this training manual.

The management of Rwanda TVET Board extends its appreciation to both its staff and the staff of the TQUM Project for their efforts in coordinating these activities.

This training manual was developed:

Under Rwanda TVET Board (RTB) Guiding Policies and Directives



Under Financial and Technical Support of



COORDINATION TEAM

RWAMASIRABO Aimable

MARIA Bernadette M. Ramos

MUTIJIMA Asher Emmanuel

Production Team

Authoring and Review

IRARORA Jonah

MUKESHIMANA Anastase

MINANI Gervais

Validation

NSENGIYUMVA Emmanuel

MUKARUGWIZA Annonciatha

KWIZERA Emmanuel

Conception, Adaptation and Editorial works

HATEGEKIMANA Olivier

GANZA Jean Francois Regis

HARELIMANA Wilson

NZABIRINDA Aimable

DUKUZIMANA Therese

NIYONKURU Sylvestre

MUKANGWIJE Yvonne

Formatting, Graphics, Illustrations, and infographics

YEONWOO Choe

SUA Lim

SAEM Lee

SOYEON Kim

WONYEONG Jeong

SHYAKA Emmanuel

Financial and Technical support

KOICA throughTQUM Project

TABLE OF CONTENT

AUTHOR'S NOTE PAGE (COPYRIGHT)---iii

ACKNOWLEDGEMENTS---iv

TABLE OF CONTENT ---vii

ACRONYMS---ix

INTRODUCTION ---1

MODULE CODE AND TITLE: SWDFA501 FRONTEND APPLICATION DEVELOPMENT WITH REACT.JS ---2

Learning Outcome 1: Develop React.Js Application ---3

Key Competencies For Learning Outcome 1: Develop React.Js Application---4

Indicative Content 1.1: Preparation of React.Js Environment ---7

Indicative Content 1.2: Applying React Basics ---21

Indicative Content 1.3: Application Of UI Navigation ---30

Indicative Content 1.4: Application Of React Hooks---38

Indicative Content 1.5: Implementation of Events Handling---56

Indicative Content 1.6: Implementation of Api Integration ---73

Learning Outcome 1 End Assessment---96

References---102

Learning Outcome 2: Apply Tailwind CSS Framework. ---103

Key Competencies For Learning Outcome 2: Apply Tailwind CSS Framework ---104

Indicative Content 2.1: Applying Tailwind Utility Classe---106

Indicative Content 2.2: Applying Responsive Design Principles ---133

Indicative Content 2.3: Customization of Tailwind Styles---150

Learning Outcome 2 End Assessment---156

References---159

Learning Outcome 3: Develop Next.JS Application ---160

Key Competencies for Learning Outcome 3: Develop Next.JS Application ---161

Indicative Content 3.1: Applying Typescript Basics---163

Indicative Content 3.2: Setup Nextjs Project---186

Indicative content 3.3: Implementing Rendering Techniques ---216

Indicative Content 3.4: Implementing Routing ---228

Indicative content 3.5: Creation of API ---238

Indicative Content 3.6: Securing The Application ---------- 242
Learning Outcome 3 End Assessment ---------- 259
References ---------- 262
Learning Outcome 4: Apply Progressive Web Application ---------- 263
Key Competencies For Learning Outcome 4: Apply Progressive Web Application ---------- 264
Indicative Content 4.1: Maintain Responsiveness ---------- 266
Indicative Content 4.2: Configuring Web Application Manifest ---------- 276
Indicative Content 4.3: Implementation of Service Workers ---------- 283
Learning Outcome 4 End Assessment ---------- 287
References ---------- 290
Learning Outcome 5: Publish the Application ---------- 291
Key Competencies For Learning Outcome 5: Publish The Application ---------- 292
Indicative Content 5.1: Configuration of Environment Variables ---------- 294
Indicative Content 5.2: Deploying React Application ---------- 297
Indicative Content 5.3: Setup Custom Domain ---------- 299
Learning Outcome 5 End Assessment ---------- 304
References ---------- 307

ACRONYMS

2FA: Two-Factor Authentication

API: Application Programming Interface

AWS:Amazon Web Services

CA: Certificate Authority

CDN:Content Delivery Network

CNAME: Canonical Name

CORS:Cross-Origin Resource Sharing

CSP:Content Security Policy

CSR:Client-Side Rendering

CSS: Cascading Style Sheet

DNS: Domain Name System

DOM:Document Object Model

Env: environment

FTP: File Transfer Protocol

HTML:HyperText Markup Language

HTTP: HyperText Transfer Protocol

HTTPS: HyperText Transfer Protocol Secure

IP: Internet Protocol

ISP: Internet Service Provider

ISR: Incremental Static Regeneration

JS: JavaScript

JSON: JavaScript Object Notation

JSX:JavaScript XML

NPM: Node Package Manager

NPX: Node Package eXecute

Nslookup: Name server lookup

PWA: Progressive Web application

RSCP: React Server Component Payload

RTB: Rwanda TVET Board

SEO: Search Engine Optimization

SQL:Structured Query Language

SSG:Static Site Generation

SSH: Secure Shell

SSL: Secure Socket Layer

SSR:Server-Side Rendering

TLD: Top Level Domain

TLS:Transport Layer Security

TQUM Project: TVET Quality Management Project

UI/UX:User Interface/User eXperience

URL: Uniform Resource Locator

VS: Visual Studio

WWW: World Wide Web

XSS:Cross-site scripting

INTRODUCTION

This trainee's manual includes all the knowledge and skills required in software development specifically for the module of **" Frontend Application Development with React.JS ".** Students enrolled in this module will engage in practical activities designed to develop and enhance their competencies. The development of this training manual followed the Competency-Based Training and Assessment (CBT/A) approach, offering ample practical opportunities that mirror real-life situations.

The trainee's manual is organized into Learning Outcomes, which is broken down into indicative content that includes both theoretical and practical activities. It provides detailed information on the key competencies required for each learning outcome, along with the objectives to be achieved.

As a trainee, you will start by addressing questions related to the activities, which are designed to foster critical thinking and guide you towards practical applications in the labor market. The manual also provides essential information, including learning hours, required materials, and key tasks to complete throughout the learning process.

All activities included in this training manual are designed to facilitate both individual and group work. After completing the activities, you will conduct a formative assessment, referred to as the end of learning outcome assessment. Ensure that you thoroughly review the key readings and the 'Points to Remember' section.

MODULE CODE AND TITLE: SWDFA501 FRONTEND APPLICATION DEVELOPMENT WITH REACT.JS

Learning Outcome 1: Develop React.Js Application
Learning Outcome 2: Apply Tailwind CSS Framework
Learning Outcome 3: Develop Nextjs Application
Learning Outcome 4: Apply Progressive Web Application
Learning Outcome 5: Publish the Application


NETFLIX
react
facebrook instagram
paypop
NETFLIX
react.js
ReactJs

Indicative contents

1.1 Preparation of React.Js Environment
1.2 Applying React Basics
1.3 Applying UI Navigation
1.4 Applying React Hooks
1.5 Implementation of Events Handling
1.6 Implementation of API Integration

Key Competencies For Learning Outcome 1: Develop React.Js Application

Knowledge	Skills	Attitudes
• Explain React.JS components, props, state, and lifecycle methods. • Explanation of Virtual DOM and its role in enhancing performance. • Identification of ReactJS uses and features. • Description of Lifecycle Methods • Description of UI navigation and ReactJS hooks • Description of ReactJS events and event handling • Description of ReactJS API integration, Security and Testing	• Installing NodeJS and Node Package Manager (NPM) • Installing React tools and libraries • Creating React Application • Exploring react project structure • Installing additional React tools and libraries • Applying React components • Creating UI navigation • Applying React hooks • Debouncing and Throttling Events • Passing Arguments to Event Handlers • Organizing API Calls	• Being creative/innovative in developing up-to-date ReactJS apps • Being Problem-Solving Oriented during development of ReactJS projects • Being Attentive to Detail during collection of ReactJS project requirements • Being a Continuous Learning while developing ReactJS applications • Being Adaptable to changes in ReactJS environment • Being Team work based during Developing robust ReactJS applications

	• Performing API Security and testing	



Duration: 40 hrs

Learning outcome 1 objectives:



By the end of the learning outcome, the trainees will be able to:

1. Describe clearly ReactJS common terminologies based on application requirements
2. Install properly NodeJS and NPM based on application requirements
3. Install properly required packages and libraries according to application requirements
4. Create properly ReactJS application based on user requirements
5. Explore clearly React project structure
6. Apply correctly React Basics based on application requirements
7. Apply correctly UI Navigation based on application navigation techniques
8. Apply Properly React hooks based on application navigation best practices
9. Implement correctly event handling techniques based on application navigation best practices
10. Implement correctly API integration based on Integration techniques

 **Resources**

Equipment	Tools	Materials
● Computer	● Text editor ● Browser ● Terminal	● Internet ● Electricity

Indicative Content 1.1: Preparation of React.Js Environment

Duration: 10 hrs

Theoretical Activity 1.1.1: Description of ReactJS environment

Tasks:

1: You are requested to answer the following questions

i. Explain the following terms:

a) ReactJS
b) Components
c) JSX
d) Props
e) State
f) Lifecycle methods
g) Hooks
h) Virtual Dom
i) Router
j) Redux

ii. Explain the uses of ReactJS

iii. What are the main Features of ReactJS

2: Write your findings on papers, flip chart or chalkboard.

3: Present the findings/answers to the whole class or trainer

4: For more clarification, read the key readings 1.1.1

Key readings 1.1.1.: Description of ReactJS environment

- **Description of key concepts**
  - ✓ **ReactJS**

ReactJS is a popular JavaScript library for building user interfaces or reusable UI Components, particularly single-page applications where you need a fast, interactive experience. To create a React JS application we need NodeJS runtime environment which will provide npm services to download and install ReactJs packages like Router, redux etc

  - ✓ **Component**

A component in React is an independent, reusable piece of UI. Components can be class-based or function-based.

Functional components are simpler, defined as JavaScript functions, and use hooks like useState and useEffect for managing state and lifecycle.

Class components are ES6 classes, requiring render() for JSX output and have lifecycle methods like componentDidMount.

Functional components are preferred in modern React for being more concise and easier to test, with hooks offering greater flexibility. Class components were traditionally used for state management before hooks, but are now less common due to the advantages of functional components

- ✓ **JSX (JavaScript XML)**

JSX is a syntax extension for JavaScript that looks similar to XML or HTML. It allows you to write HTML structures in the same file as JavaScript code.

- ✓ **Props**

Props (short for properties) are read-only attributes that are passed from a parent component to a child component. They allow data to be passed around your application. Props are immutable, meaning that once they are passed to a component, the component cannot modify them. Instead, they are used to render dynamic content based on the values passed.

- ✓ **State**

State is a built-in object that holds data or information about the component. Unlike props, state is managed within the component and can be changed over time, usually as a result of user actions.

- ✓ **Lifecycle Methods**

Lifecycle methods are special methods in React components that allow you to run code at particular times in the component's lifecycle, such as when the component mounts, updates, or unmounts.

- ✓ **Hooks**

Hooks are functions that let you use state and other React features in functional components. Examples include useState, useEffect, and useContext.

- ✓ **Virtual DOM**

The Virtual DOM is a lightweight copy of the actual DOM that React uses to optimize rendering. React updates the Virtual DOM, then efficiently updates the real DOM to match.

- ✓ **React Router**

React Router is a library for routing in React applications. It enables navigation among views of various components in a React application, allowing you to change the browser URL and keep the UI in sync with the URL.

- ✓ **Redux**

Redux is a predictable state container for JavaScript apps, often used with React to manage the application's state in a centralized store.

- **Uses of reactJS**
  - ✓ Building dynamic web applications
  - ✓ Creating reusable UI components
  - ✓ Developing single-page applications (SPAs)
- **Features of ReactJS**
  - ✓ Declarative syntax: React uses declarative programming, meaning that instead of manually telling the system how to change the state of the UI, you declare the desired state, and React will manage the transitions and updates automatically.
  - ✓ Component-based architecture: React allows developers to build the UI by breaking it into reusable, self-contained components. Each component manages its own state and logic, making development modular and maintainable.
  - ✓ Efficient rendering using the Virtual DOM: React utilizes a Virtual DOM to improve performance. Instead of updating the real DOM (which can be slow), React keeps a lightweight copy (the Virtual DOM) and only updates the real DOM when necessary.
  - ✓ Rich ecosystem with tools like React Router and Redux: A routing library that allows you to manage navigation and views in a React app. It helps with dynamic routing and renders components based on the current URL. A predictable state management library. It helps in managing the entire application's state in a single store, making it easier to debug and maintain applications with complex states.
- **Application of ReactJS**

As a **JavaScript framework**, React.js has a wide range of applications due to its flexibility, component-based architecture, and efficient rendering system. Here are key applications of React.js as a JS framework:

- ✓ **Single Page Applications (SPAs)**
  React is widely used to build **SPAs**, which load a single HTML page and dynamically update content based on user interaction without reloading the page. React's virtual DOM and component structure allow for efficient updating of the UI, making it perfect for SPAs.
  Examples: Gmail, Facebook, Instagram.
- ✓ **Dynamic and Interactive Web Interfaces**
  React is ideal for building **highly interactive and dynamic user interfaces** that require real-time updates or frequent UI changes. The declarative nature of

React allows developers to efficiently manage the state of an application and reflect changes in the UI without complex DOM manipulation.

Examples: Online marketplaces, social networking sites, news feeds.

- ✓ **Progressive Web Applications (PWAs)**

  React is used to build **PWAs** that offer a mobile-app-like experience within a browser. PWAs are responsive, work offline, and provide fast loading times. React's component-based architecture and efficient rendering system make it a suitable choice for creating PWAs.

  Examples: Twitter Lite, Pinterest.

- ✓ **Mobile Applications (React Native)**

  React.js powers **React Native**, a framework for building native mobile applications using the same codebase as a web app. React Native allows developers to create **cross-platform mobile apps** for iOS and Android by reusing components across both platforms.

  Examples: Facebook, Instagram, Airbnb.

- ✓ **Content Management Systems (CMS)**

  React can be used to create **content management systems** or integrate with existing CMS platforms. Its component structure allows for dynamic previews, content updates, and real-time interaction, making it an ideal solution for CMS-based applications.

  Examples: Netlify CMS, WordPress (with headless CMS).

- ✓ **E-Commerce Platforms**

  React is commonly used to build **e-commerce websites** that need fast page loads, dynamic product filtering, seamless navigation, and a smooth checkout experience. React's modularity allows developers to create reusable components like product cards, filters, and carts.

  Examples: Shopify, Walmart, Amazon.

- ✓ **Data-Driven Dashboards and Analytics**

  React is well-suited for building **data visualization and analytics dashboards**. React can efficiently manage frequent data updates and render complex visualizations using libraries like D3.js or Chart.js. Dashboards built with React offer interactive elements like filtering, sorting, and real-time updates.

  Examples: Financial dashboards, Google Analytics, health tracking platforms.

- ✓ **Enterprise Web Applications**

  React is widely used for **enterprise-level applications** that require scalable, maintainable, and high-performance web interfaces. Applications such as ERP (Enterprise Resource Planning), CRM (Customer Relationship

Management), and HR systems benefit from React's modular and reusable components.

Examples: Salesforce, SAP Fiori, Jira.

- ✓ **Real-Time Applications**

  React.js is excellent for building **real-time applications** like chat applications, collaboration tools, and streaming services. By leveraging technologies like WebSockets or Firebase, React can handle real-time data efficiently, making it ideal for such use cases.

  Examples: WhatsApp Web, Slack, Zoom.

- ✓ **Static Site Generation (SSG)**

  Using tools like **Gatsby.js** and **Next.js**, React can be used to generate **static websites** that are fast and optimized for SEO while still providing interactivity through React components. SSG is a popular choice for blogs, marketing websites, and documentation sites.

  Examples: Blogs, landing pages, documentation (e.g., React's own docs).

- ✓ **Server-Side Rendering (SSR)**

  Next.js, a React-based framework, provides features for **server-side rendering (SSR)**, which improves page load times and SEO by pre-rendering React components on the server. This approach is particularly beneficial for applications with a lot of dynamic content.

  Examples: E-commerce websites, content-heavy platforms.

- ✓ **Streaming Media Services**

  React is commonly used in **media streaming services**, where performance is critical due to the need for real-time media delivery. React's efficient DOM updates allow for smooth user interfaces even when handling large amounts of data.

  Examples: Netflix, Hulu, YouTube.

- ✓ **Real-Time Collaboration Platforms**

  React is ideal for **real-time collaboration platforms** where multiple users can interact with content simultaneously. This could include tools like project management platforms, document editors, or video conferencing applications.

  Examples: Google Docs, Trello, Figma.

- ✓ **Cross-Platform Desktop Applications (Electron + React)**

  React can be used in combination with **Electron** to create **cross-platform desktop applications** using web technologies. Electron provides a platform for running web apps as native desktop apps, and React enables building modular and dynamic user interfaces.

  Examples: Visual Studio Code, Slack Desktop, Discord.

- ✓ **Form-Based Applications**

  React is also suitable for building complex, interactive **form-based applications** that require real-time validation, conditional rendering, and smooth user interaction. These can range from survey forms to financial applications that require dynamic inputs.

  Examples: Tax filing apps, survey tools, insurance calculators.

- ✓ **Search Engines and Filtering Tools**
  - ✓ React is used in creating powerful **search engines and filtering tools** with instant feedback. Its ability to handle state management and dynamic updates makes it perfect for providing real-time filtering and search results.
  - ✓ **Examples**: E-commerce product searches, library search systems.
- ✓ **Interactive Educational Platforms**

  React can be used to create **educational platforms** with interactive features like quizzes, videos, progress tracking, and discussion boards. Its modular approach allows for the creation of reusable components for courses and lessons.

  Examples: Coursera, Udemy, Khan Academy.

- ✓ **Social Media Platforms**

  React.js is a go-to framework for building **social media platforms**, given its ability to handle large-scale, dynamic content, real-time updates (like posts, comments, and likes), and seamless user experience.

  Examples: Facebook, Instagram, Twitter.

- **Advantages of using react**

✓ **Faster debugging and rendering.**

ReactJS offers interaction for any UI layout and is simple to use. Additionally, it enables the rapid and quality assured development and rendering of applications, saving time for clients and React developers.

✓ **Reusability of components.**

React developers may leverage the reusable components that ReactJS offers to build new applications. Each of its web application's several components has its logic and control. These parts produce a short, reusable chunk of html code that you use anywhere you need it.

✓ **Readily available JavaScript libraries**

A robust JavaScript and HTML syntax mix streamlines the entire code development process for the intended project. The reason the majority of web React developers prefer ReactJS is that it provides a robust JavaScript library.

experiences for all browsers and search engines, making it easier for React developers to be found on different search engines.

✓ **Use of virtual Document Object Model or V-DOM.**

ReactJS leverages Virtual DOM, an API programming cross-platform interface that deals with web development to enhance performance. When creating an app with lots of user interaction and data exchanges, it's essential to test how the structure of your app will affect speed. Despite of fast client platforms and JavaScript engines, extensive DOM manipulation can affect the platforms operating speed and produce an uncomfortable user experience. React addresses this by employing a virtual DOM, representing the DOM of a web browser that lives in memory. As a result, we avoid writing to the DOM when creating React Components; instead make virtual components that respond and transform into DOM, resulting in more seamless and faster performing platforms.

- ✓ **Great for SEOs**

Search engine optimization(SEO) makes it more straightforward for search engines to identify relevant material for users. When a user searches, the search engine determines which page is the most pertinent to that particular search.

Conventional JavaScript frameworks need help to handle SEO. JavaScript-heavy apps have problems being read by search engines. ReactJS solves this issue by enabling search engine-optimized user experiences for all browsers and search engines, making it easier for react Developers to be found on different search engines.

- ✓ **Easy to learn and use.**

ReactJS trains the programmer how to traverse the framework and offer you new thought patterns that might help you in other fields of programming. it contains a wealth of information, video, and teaching materials and is simple to use and pick up.

- ✓ **Active React JS Community.**

ReactJS boast one of the most significant communities of react Developers and designers. React code is also open source, allowing anybody to download, edit, and distribute it with others to develop the framework.

- ✓ **Readily available tools for experimenting**

React enables you to view the virtual DOM's React component hierarchies. It consists of a browser plugin called React Developer Tools that are available for both Chrome and Firefox, using which any component's hierarchy may be tracked, allowing you to find both parent-child components.

- ✓ **Worldwide usage of ReactJS by prominent brands.**

For their applications, websites and internal projects, ReactJS has been selected by thousands of businesses worldwide. Companies like Walmart Labs, Tencent, Tesla, Bloomberg, Airbnb, Baidu Mobile, GoDaddy, Gyroscope, and many more developed mobile apps using Reactnative. Handreds of websites use React, including Coursera, Paypal, Netflix, Dailymotion, IMDb, Chrysler, BBC, American Express, Khan Academy,Lyft, New York Times, DropBox, Reddit, and others.



Practical Activity 1.1.2: Installing NodeJS and Node Package Manager



Task:

1 : You are requested to go to the computer lab to Install NodeJS and Node Package Manager .
2 : Read key readings 1.1.2
3 : Apply safety precautions.
4 : Install NodeJS and Node Package Manager.
5 : Present your work to the trainer
6 : Perform the task provided in application of learning 1.1



Key readings 1.1.2: Installing NodeJS and Node Package manager

There are steps to install

Step 1: Download Node.js

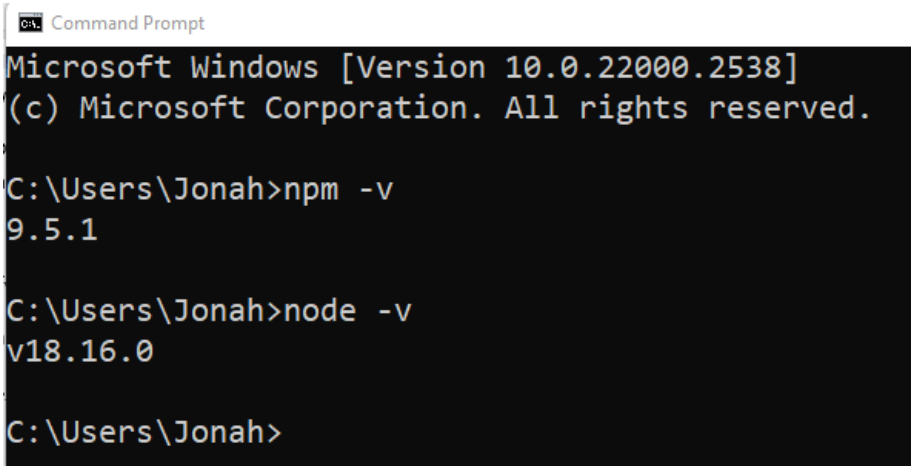
- Go to the official Node.js website: https://nodejs.org/
- Download the Windows Installer (.msi) file for the LTS (Long-Term Support) version.

Step 2: Install Node.js

- Run the downloaded installer.
- Follow the prompts in the Node.js Setup Wizard, making sure to check the box that says "Automatically install the necessary tools" (this installs NPM along with Node.js).
- Restart your system if required.

Step 3: Verify the installation

- Open Command Prompt.
- Check Node.js version by typing the command below in Command Prompt: node -v
- Check NPM version by typing the command below in Command Prompt: npm –v



Practical Activity 1.1.3: Creating ReactJS application

Task:

1 : You are requested to go to the computer lab to create a ReactJS application
2 : Read key Reading 1.1.3
3 : Apply safety precautions.
4 : Create a ReactJS application.
5 : Present your work to the trainer and whole class.
6 : Perform the task provided in application of learning 1.1.

Key readings 1.1.3: Creating ReactJS application

Steps to create of ReactJS application

Step 1. Install and create a new React project:

- Run npx create-react-app my-app Command

This will create a new directory named my-app with the basic structure of a React app.

```
npm install react react-dom react-scripts cra-template
Microsoft Windows [Version 10.0.22000.2538]
(c) Microsoft Corporation. All rights reserved.

E:\ReactApps>npx create-react-app my-app

Creating a new React app in E:\ReactApps\my-app.

Installing packages. This might take a couple of minutes.
Installing react, react-dom, and react-scripts with cra-template...

[                    ] / idealTree:my-app: sill idealTree buildDeps
```

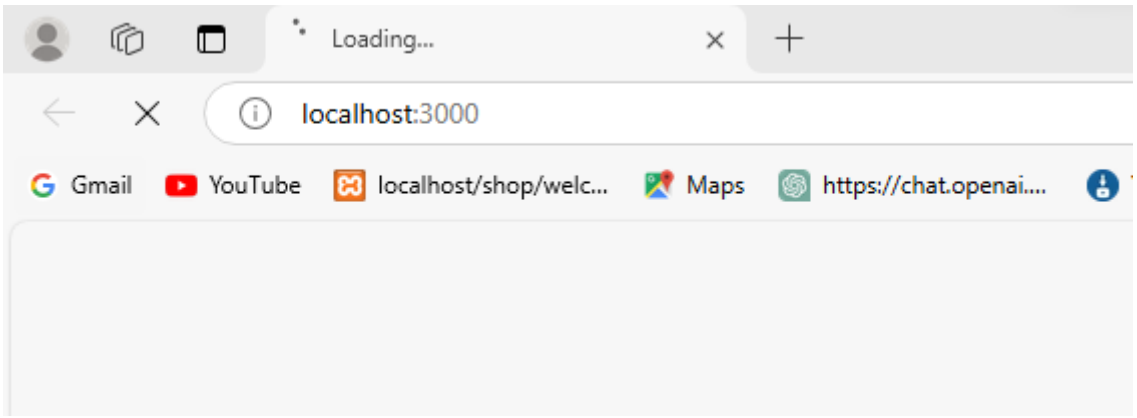
Step 2. Navigate into the project directory:

- Run Command: cd my-app

Step 3. Start the development server:

- Run Command: npm start

This will start the local development server, and you should be able to see your React app running at http://localhost:3000 in your browser.



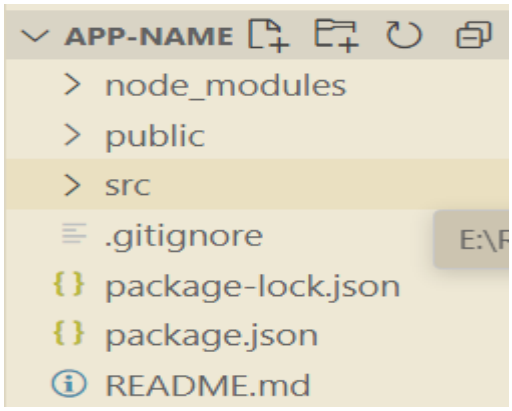
Practical Activity 1.1.4: Exploring of React project structure

Task:

1 : You are requested to go to the computer lab to open existing ReactJS application and explore its structure.
2 : Read key reading 1.1.4
3 : Apply safety precautions.
4 : Explore a ReactJS project structure.
5 : Present your work to the trainer.
6 : Perform the task provided in application of learning 1.1.

Key readings 1.1.4: Exploring of React project structure

The root directory of your React project contains several important files and folders as shown in the screen shot below.
The following are steps to explore the project structure
Step 1. Browse to the folder where your project is created
Step 2. Open Command prompt
Step 3. Type the command to open the project in your text editor
Step 4. View and analyse every file and folder from the project explorer in the text editor as shown below



- **Explanation of common folders and files**
  - ✓ node_modules/: Contains all the dependencies and packages installed via npm or yarn. You usually don't interact with this directly.

- ✓ public/: This folder holds static assets that are served directly by the server without any modification. Key files include:
  - index.html: The single HTML file that React injects into. This file is responsible for displaying the React app in the browser.
  - favicon.ico: The icon displayed on the browser tab.
  - manifest.json: Used for Progressive Web Apps (PWA). It provides metadata about the app.
  - robots.txt: Instructions for search engine bots.
- ✓ src/: The main source folder for all the React code. Most of your development happens here.
- ✓ package.json: Lists all dependencies, scripts, and metadata about the project.
- ✓ README.md: A markdown file that usually contains information about the project and instructions for setup.
- ✓ .gitignore: Specifies which files/folders to ignore in version control (like Git).



Practical Activity 1.1.5: Installing of additional React tools and libraries



Task:

1 : You are requested to go to the computer lab to install additional React tools and libraries
2 : Apply safety precautions.
3 : Read key reading 1.1.4
4 : Install additional React tools and libraries .
5 : Present your work to the trainer
6 : Perform the task provided in application of learning 1.1.



Key readings 1.1.5: Installing of additional React tools and libraries

Method 1: Installing a single package

- ✓ Install Redux

Run command: npm install redux react-redux

✓ Install React Router

Run command: npm install react-router-dom

✓ Install Axios

Run command: npm install axios

Method 2: Install all packages using single line of command

✓ npm install redux react-redux react-router-dom axios



Points to Remember

- **ReactJS** is a JavaScript library for building user interfaces, utilizing components to encapsulate UI elements.
- **JSX**, a syntax extension, allows HTML-like code within JavaScript.
- **Components** use **props** for data passing and **state** for internal data management.
- React's **lifecycle methods** manage component updates.
- H**ooks** like useState and useEffect provide functional components with state and side-effects handling.
- **Virtual DOM** improves performance by efficiently updating UI changes. React Router enables dynamic navigation.
- **Redux** manages application-wide state. ReactJS is widely used for its modular structure, reusability, and performance optimizations, making it ideal for complex UIs.

Steps to install and verify NodeJS and NPM

- Download Node.js
- Install Node.js
- Check Node.js version by using node -v
- Check NPM version by using npm –v

Use the create-react-app CLI tool to set up a new React project by running the following commands in the terminal

1. Open terminal inside a folder you want to create a project
2. Type the command *npx create-react-app my-app* in the terminal
3. After the project is created, type *cd my-app* to navigate to the project folder
4. After navigating to the project folder, type *npm start* to start your application server
5. The project will be initialized in a default browser on your computer with a domain name and port number

The root directory of your React project contains several important files and folders as shown below:

- node_modules
- public
- src
- package.json
- README.md
- .gitignore

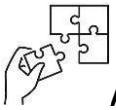
Methods to install the additional react packages

Method 1: Installing a single package

- Install Redux
  Command: npm install redux react-redux
- Install React Router
  Command: npm install react-router-dom
- Install Axios
  Command: npm install axios

Method 2: Install all packages using single line of command

- npm install redux react-redux react-router-dom axios



Application of learning 1.1

ABC company is an Online Car selling company, it needs a front-end application to manage its clients. You are requested to prepare React JS Environment and Create a front end application in ReactJS. Install additional react JS packages like Redux, react router and Axios.

Indicative Content 1.2: Applying React Basics

Duration: 5 hrs

Practical Activity 1.2.1: Creating React Class Components

Task:

1 : You are requested to go to the computer lab to create class components
2 : Read key reading 1.2.1
3 : Apply safety precautions.
4 : Create class components
5 : Present your work to the trainer and whole class.
6 : Perform the task provided in application of learning 1.2

Key readings 1.2.1: Creating React Class Components

Steps to Create a Class component in an existing ReactJS Project

Step 1. Create the Component File

Step 2. Inside your src folder, create a new file for your component. For example, you can call it App.js.

Step 3. Write the Class Component

Step 4. In the App.js file, define the class component as follows

Welcome | JS index.js | JS App.js ×

src > JS App.js > default

```
1  import { Component } from "react";
2
3  class App extends Component{
4      render(){
5          return (<h1>My Class Component</h1>)
6      }
7  }
8  export default App;
```

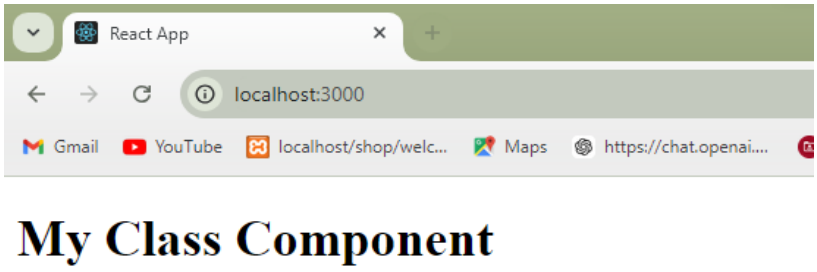
Step 5. Import and Use the Component

Step 6. In your src/index.js file (the entry point for your React app), ensure that your App

component is imported and used correctly

```
me    JS index.js  ×   JS App.js
index.js > ...
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
    <App />
);
```

Step 7. Run the Application





Practical Activity 1.2.2: Creating React Functional Components



Task:

1 : You are requested to go to the computer lab to create functional components
2 : Read key reading 1.2.1
3 : Apply safety precautions.
4 : Create functional components
5 : Present your work to the trainer and whole class.
6 : Perform the task provided in application of learning 1.2

Key readings 1.2.1: Creating React Functional Components

Steps to Create a functional component in an existing ReactJS Project

Step 1. Create the Component File

Step 2. Inside your src folder, create a new file for your component. For example, you can call it App.js.

Step 3. Write the functional Component

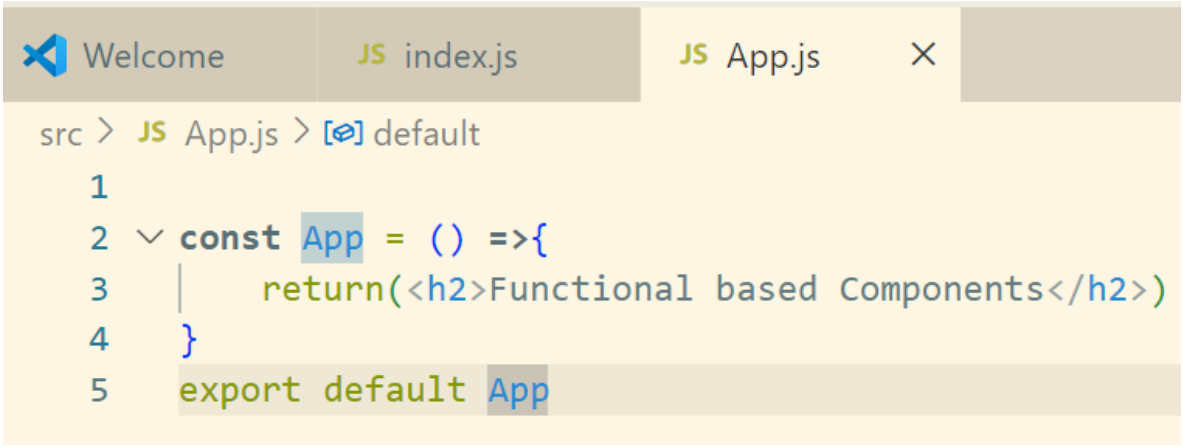
Step 4. In the App.js file, define a functional component as follows

Breakdown of Functional Component

Declaration: The function App is declared using the arrow function syntax (const App = () => { ... }).

Return Statement: The component returns JSX, which looks like HTML but is written in JavaScript.

Exporting the Component: The export default App; statement is used to make this component available for import in other parts of the app



Step 5. Import and Use the Component

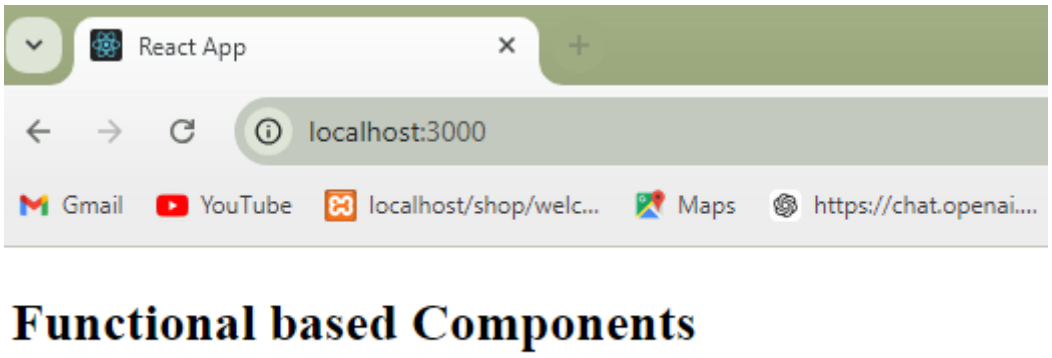
Step 6. In your src/index.js file (the entry point for your React app), ensure that your App component is imported and used correctly

me JS index.js JS App.js

index.js > ...

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
    <App />
);
```

Step 7. Run the Application





Practical Activity 1.2.3: Creating React Props



Task:

1 : You are requested to go to the computer lab to create Props
2 : Read key reading 1.2.1
3 : Apply safety precautions.
4 : Create Props
5 : Present your work to the trainer
6 : Perform the task provided in application of learning 1.2

Key readings 1.2.3: Creating Props

Step 1. Import Necessary Libraries:

Ensure you're importing React and ReactDOM correctly to use JSX syntax and render components.

```
import React from 'react';
import ReactDOM from 'reactdom/client';
```

Step 2. Create a Functional Component that Accepts Props

Let's take an example of a Farm.js component, the functional component Farm accepts props as an argument. Props are passed down from the parent component and can be accessed via props object. In the example below, we shall use the properties animal, crops, and location.

Welcome | index.js | Farm.js

src > Farm.js > Farm

```
1
2  const Farm = (props) =>{
3      return(<div>
4          <h2>Farm deals with:</h2>
5          Type of animals:{props.animal}<br/>
6          Crops:{props.crops}<br/>
7          Location:{props.location}
8
9      </div>)
10 }
11 export default Farm
```

Alternatively, you could destructure the props directly within the function signature to make it cleaner:

```
Welcome | JS index.js | JS Farm.js ●
src > JS Farm.js > [@] Farm
```

```jsx
const Farm = ({ animal, crops, location }) =>{
    return(<div>
      <h2>Farm deals with:</h2>
      Type of animals:{animal}<br/>
      Crops:{crops}<br/>
      Location:{location}

    </div>)
}
export default Farm
```

Step 3. Pass Props from the Parent Component:

Example: In index.js, you pass the props (animal, crops, location) from the parent component (Farm) when rendering it within ReactDOM.render.

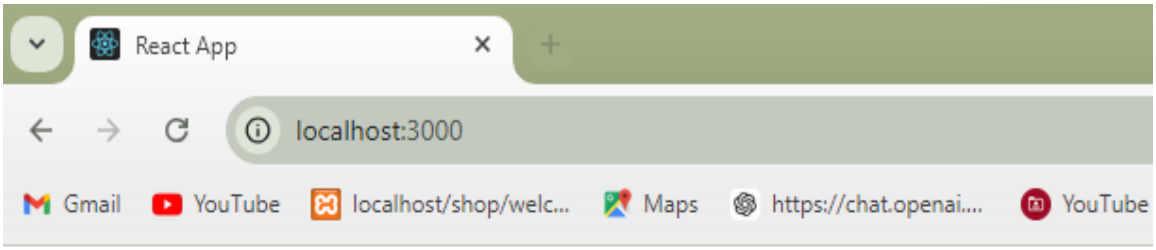
```
Welcome | JS index.js × | JS Farm.js ●
src > JS index.js > ...
```

```jsx
import React from 'react';
import ReactDOM from 'react-dom/client';
import Farm from './Farm';
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <Farm animal="cows" crops="Maize" location="Kigali"/>
);
```

In this example, we are passing the values "cows", "Maize", and "Kigali" as props, which will be passed down to the Farm component and accessed inside it.

Step 4. Render the Component to the DOM:

The Farm component will receive the props and display them as per the JSX template inside Farm.js. When this code runs, it will output the following content:



Farm deals with:

Type of animals:cows
Crops:Maize
Location:Kigali

Theoretical Activity 1.2.4: Description of Lifecycle methods

Tasks:

1 : You are requested to answer the following questions:

I. Describe the use of the componentDidMount method
II. Describe the use of the componentDidUpdate method
III. Describe the use of the componentDidUnmount method

2 : Write your findings on papers.

3 : Present the findings/answers to the whole class or trainer

4 : For more clarification, read the key readings 1.2.4.

Key readings 1.2.4: Description of Lifecycle methods

- **componentDidMount:**
  This method is called once, immediately after the component is added (mounted) to the DOM. It's commonly used for:
  - ✓ Fetching data from APIs.
  - ✓ Setting up subscriptions (e.g., WebSockets)
  - ✓ Initializing timers.
- **componentDidUpdate:**

This method is called after the component updates due to changes in state or props. You can perform side effects here based on the previous props or state, such as making API requests if certain data has changed.

- **componentWillUnmount:**
  This method is called just before the component is removed (unmounted) from the DOM. It's typically used for cleanup tasks like:
  - ✓ Clearing timers.
  - ✓ Canceling API requests.
  - ✓ Unsubscribing from services (e.g., WebSocket).

Note: In modern React, these lifecycle methods can be replaced using the useEffect hook in functional components.



Points to Remember

Steps to create a class component in ReactJS

1. In an existing ReactJS project, create a file in src folder and name it a name of your choice
2. Inside the file, on the top, import React and, if needed, other components or libraries
3. Create a new class that extends Component
4. Implement the Render Method. Every class component must have a render () method that returns the JSX to be rendered
5. Inside a render () method, input a return function that will return XML scripts
6. At the end of your file, export the component so it can be imported elsewhere.

Steps to create a functional based component in ReactJS

1. In an existing ReactJS project, create a file in the src folder and name it appropriately
2. Inside the file, create a function that represents the component
3. In the body of the functional component, return the JSX directly, which replaces the render () method in class components
4. At the end of your file, export the component so it can be imported elsewhere.

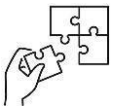
Steps to create Props in ReactJS

1. In an existing ReactJS project, create a two files in the src folder and name them appropriately

2. Inside the first file, create a functional based component, import and use a child component
3. In the second file, create a child component with a functions with props passed as a function argument
4. In the parent component, assign any number of attributes to the child component called in the return method.
5. In the child component, access attributes in the parent component from props object
6. Start the project and view the output.

Lifecycle methods

1. componentDidMount: Runs after a component mounts, used for initializing data (e.g., fetching data, setting up subscriptions).
2. componentDidUpdate: Invoked after component updates due to state/prop changes, allowing reactions to changes.
3. componentWillUnmount: Triggered before unmounting, used for clean-up like removing subscriptions or timers.



Application of learning 1.2.

ABC is a company located in Kigali City, Nyarugenge district. It Specializes in website development. The company wants to develop a web application using react.js. As a ReactJS developer you have been hired by the company and tasked to create a simple React app that displays a list of books. Use a **class component** to manage the book data and a **functional component** to display individual book details. Pass each book's title, author, and description as **props** from the class component to the functional component. Use **JSX** to render the book details in the UI. Ensure that the data is properly passed and displayed.

Indicative Content 1.3: Application Of UI Navigation

Duration: 5 hrs

Practical Activity 1.3.1: Applying Basic React navigation

Task:

1 : You are requested to go to the computer lab to apply basic react navigation
2 : Read key reading 1.3.1
3 : Apply safety precautions.
4 : Apply basic react navigation
5 : Present your work to the trainer
6 : Perform the task provided in application of learning 1.3

Key readings 1.3.1: Applying Basic React navigation

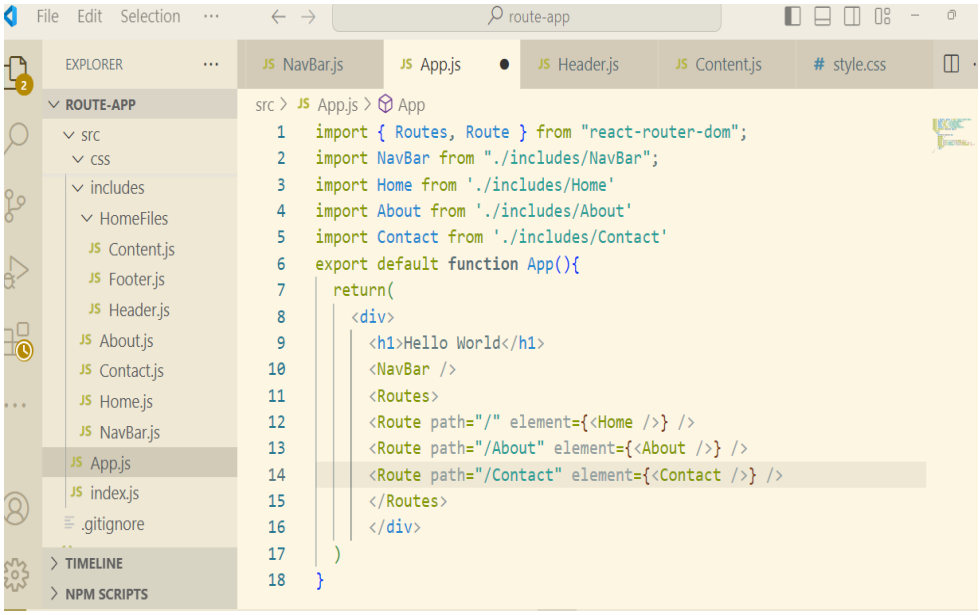
Step 1: Install React Route

1. Open your command prompt or terminal in your project directory.
2. Run the command to install React Router as: run npm install react-router-dom

Step 2: Configure Routes

1. First, ensure you have your React components, such as Home and About, created. These will be displayed based on the route.
2. Configure your routes in the main App.js or another component responsible for routing.

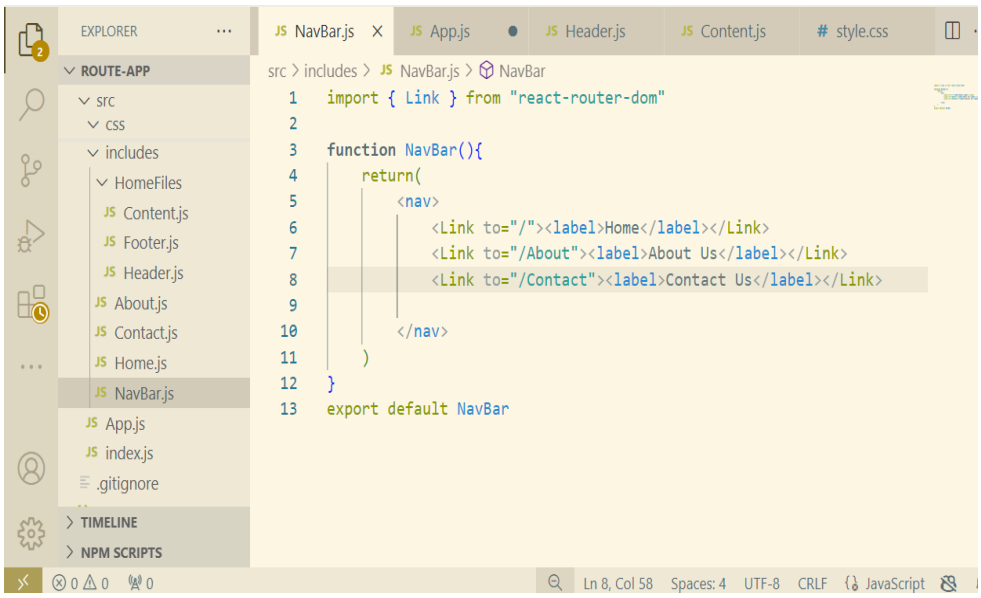
Sample code



Step 3: Apply basic React Navigation

1. To navigate between different routes, you can create a simple navigation bar using the Link component from react-router-dom
2. Replace Link component with traditional anchor tag (<a>) in React Router, allowing seamless navigation between pages without reloading the entire application.

Sample code





Practical Activity 1.3.2: Handling 404 Pages, Redirects and URL Parameters



Task:

1 : You are requested to go to the computer lab to Handle 404 Pages
2 : Read key reading 1.3.2
3 : Apply safety precautions.
4 : Handle 404 Pages, Redirects and URL Parameters
5 : Present your work to the trainer and whole class.
6 : Perform the task provided in application of learning 1.3



Key readings 1.3.2: Handling 404 Pages, Redirects and URL Parameters

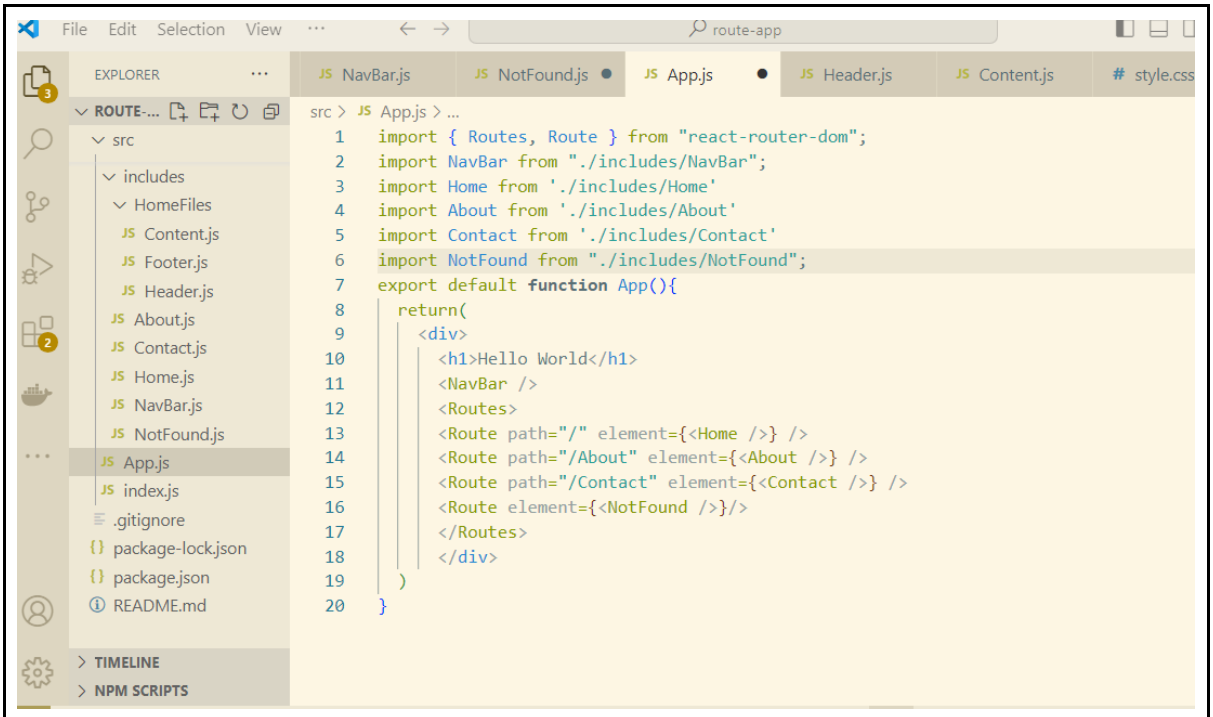
A 404 page is displayed when no routes match the requested URL. You can achieve this by using the <Route> component without a path and placing it at the end of your <Routes> block.

Steps to Handle 404 Pages:

Step 1. Ensure react-router-dom is installed.

```
npm install react-router-dom
```

Step 2. Set up the routing in your component as shown below



Redirects

You can use the <Redirect> component to automatically redirect users from an old path to a new one.

Steps to use Redirects:

In your routing component, add a route that uses <Redirect> to point from the old path to the new path.

```
import { Switch, Route, Redirect } from 'react-router-dom';
import Home from './Home';
import NewComponent from './NewComponent'; // Component for the new path
const App = () => (
<Switch>
<Route path="/" exact component={Home} />
{/ Redirect from /old-path to /new-path /}
<Route path="/old-path">
<Redirect to="/new-path" />
</Route>
<Route path="/new-path" component={NewComponent} />
<Route component={NotFound} />
</Switch>
);
export default App;
```

Steps to use URL Parameters

React Router allows you to capture URL parameters and use them inside components. This is useful for dynamic pages, such as a user profile.

Step 1: Define a route with a parameter in your App component.

Step 2: Access the parameter via match.params inside the target component.

```
import { Switch, Route } from 'react-router-dom';
import Home from './Home';
import User from './User'; // Component to handle the dynamic user id
const App = () => (
<Switch>
<Route path="/" exact component={Home} />
{/ Define a route with a URL parameter /}
<Route path="/user/:id" component={User} />
<Route component={NotFound} />
</Switch>
);
export default App;
```

Step 3: In the User component, extract the id parameter from match.params:

```
import React from 'react';
const User = ({ match }) => {
return <h1>User ID: {match.params.id}</h1>;
};
export default User;
```



Practical Activity 1.3.3: Applying Nested Routes



Task:

1 : You are requested to go to the computer lab to apply nested routes.
2 : Read key reading 1.3.3
3 : Apply safety precautions.
4 : Apply nested routes
5 : Present your work to the trainer
6 : Perform the task provided in application of learning 1.3



Key readings 1.3.3: Applying Nested Routes

To add nested routes, you'll structure your routes inside a parent component. The parent route will contain its own route but also define child routes.

Step 1: Define Parent Component

```
import { Outlet, Link } from 'react-router-dom';
function Dashboard() {
return (
<div>
<h1>Dashboard</h1>
<nav>
<ul>
<li><Link to="overview">Overview</Link></li>
<li><Link to="stats">Stats</Link></li>
</ul>
</nav>
{/ Outlet will render child routes here /}
<Outlet />
</div>
 );
}
export default Dashboard;
```

Step 2: Define Child Components

For the nested routes, you'll create child components:

```
function Overview() {
 return <h2>Overview Page</h2>;
}
function Stats() {
 return <h2>Stats Page</h2>;
}
```

Step 3: Add Nested Routes in App.js

Now, add the nested routes under the parent route in your Routes.

```
import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';
import Home from './components/Home';
import About from './components/About';
import Dashboard from './components/Dashboard';
import Overview from './components/Overview';
```

```
import Stats from './components/Stats';
function App() {
  return (
<Router>
<Routes>
<Route path="/" element={<Home />} />
<Route path="/about" element={<About />} />
{/ Parent Route with Nested Routes /}
<Route path="/dashboard" element={<Dashboard />}>
<Route path="overview" element={<Overview />} />
<Route path="stats" element={<Stats />} />
</Route>
</Routes>
</Router>
  );
}
export default App;
```

Step 4: Test the routes



Points to Remember

Steps used to apply UI navigation

- **Installation of React Router**
  1. Open your command prompt or terminal in your project directory.
  2. Run the command to install React Router:
- **Configuration of Routes**
- **Application of Basic React Navigation**

  Handling 404 pages

  1. Ensure react-router-dom is installed.
  2. Set up the routing in your component

  Redirects

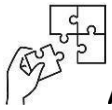
  1. In your routing component, add a route that uses <Redirect> to point from the old path to the new path.

  URL Parameters

  1. Define a route with a parameter in your App component.
  2. Access the parameter via match.params inside the target component.
  3. In the User component, extract the id parameter from match.params

  Steps to apply Nested Routing

1. Create a parent component with an Outlet for child routes.
2. Define child components and nested routes inside the parent route.
3. Test the navigation and routes



Application of learning 1.3.

ABC is an online business company Located in Kayonza district in Eastern Province-Rwanda. It needs a ReactJS developer to develop an ecommerce website. Assume you have been hired as a ReactJS developer for the company, you are tasked with creating an ecommerce application using React. Begin by installing React Router to manage UI navigation effectively. Configure various routes for product listings, user profiles, and shopping carts. Implement basic navigation to ensure users can easily access different sections. Handle 404 pages for non-existent routes and set up redirects for outdated URLs. Additionally, utilize URL parameters for filtering products and apply nested routing for displaying reviews within individual product pages.

Indicative Content 1.4: Application Of React Hooks

Duration: 5 hrs

Theoretical Activity 1.4.1: Description of Hooks

Tasks:

1 : You are requested to answer the following questions

i. What do you understand by the term:

a) Hooks
b) State Hooks
c) Effect Hooks
d) Context Hooks
e) Ref Hooks
f) Callback Hooks

ii. Explain different types of Hooks

iii. What are the factors considered while selecting the right Hook to be used?

iv. Why do we combine hooks?

2 : Write your findings on papers.

3 : Present the findings/answers to the whole class or trainer

4 : For more clarification, read the key readings 1.4.1

Key readings 1.4.1: Description of Hooks

- **Definition of Key Concepts**
  - ✓ **Hooks:** Hooks are special functions in React that let you use state and other React features in functional components without needing a class.
  - ✓ **State Hooks (useState):** State hooks allow functional components to store and manage state.
  - ✓ **Effect Hooks (useEffect):** Effect hooks let you perform side effects in functional components. These effects can be data fetching, subscriptions, or manually changing the DOM.
  - ✓ **Context Hooks (useContext):** Context hooks provide a way to pass data through the component tree without needing to pass props manually at every level.
  - ✓ **Ref Hooks (useRef):** Ref hooks allow you to persist values between renders without causing a re-render when the value changes.

- ✓ **Callback Hooks (useCallback):** Callback hooks memorize a function so that the function does not get recreated on every render. This is particularly useful for optimizing performance when passing functions as props to child components.

- **Types of Hooks**
  - ✓ useState: Used for state management.
  - ✓ useEffect: Used for performing side effects in components.
  - ✓ useContext: Used for consuming context values.
  - ✓ useReducer: An alternative to useState for more complex state logic, similar to Redux's reducer.
  - ✓ useCallback: Memorizes functions to prevent unnecessary re-creations on re-renders.
  - ✓ useMemo: Memorizes values to prevent expensive calculations on each render.
  - ✓ useRef: Accesses and persists values between renders without causing re-renders.
  - ✓ useLayoutEffect: Similar to useEffect, but it runs synchronously after all DOM mutations, which can affect layout.
  - ✓ useImperativeHandle: Customizes the instance value exposed to parent components when using ref.
- **Factors to Consider While Selecting the Right Hook**
  - ✓ State complexity: For simple state, useState is sufficient, but for complex state logic or state that depends on previous state, useReducer might be better.
  - ✓ Side effects: If the component needs to perform a side effect like fetching data or interacting with external APIs, useEffect is appropriate.
  - ✓ Optimization: When passing functions or values down as props, useCallback and useMemo help in optimizing performance by preventing unnecessary re-creations.
  - ✓ Context needs: If the component requires access to globally available data without prop drilling, useContext is the best choice.
  - ✓ DOM access: To directly interact with or store references to DOM elements or persist mutable values across renders, useRef is ideal.
- **Reasons for combining Hooks**
  - ✓ Modularity: Hooks encapsulate specific logic, allowing developers to compose different behaviours without code repetition.
  - ✓ Optimization: Combining useCallback or useMemo with other hooks can optimize rendering performance by preventing unnecessary re-renders or recalculations.

- ✓ Code organization: Combining multiple hooks can help organize state, side effects, and event handling logic in a way that's easier to maintain and understand.
- ✓ Reusability: Custom hooks can be created by combining multiple built-in hooks. This enables reusing common logic across different components.



Practical Activity 1.4.2: Implementation of Hooks



Task:

1 : You are requested to go to the computer lab to implement the following Hooks.
   a) State Hooks
   b) Effect Hooks
   c) Context Hooks
   d) Ref Hooks
   e) Call-back Hooks

2 : Read key reading 1.4.2

3 : Apply safety precautions.

4 : Implement Hooks

5 : Present your work to the trainer

6 : Perform the task provided in application of learning 1.4.2.



Key readings 1.4.2: Implementing Hooks

Step 1: Import the useState hook from React.

Step 2: Use useState to define a state variable and a function to update it.

Step 3: Use the state variable in your component's render logic.

Step 4: Call the state updater function to modify the state based on an event.

Welcome | index.js | App.js | Home.js | About.js

src > App.js > App

```
1   import {useState} from 'react'
2   function App(){
3     const [count, setCount]=useState(0)
4     return(
5       <>
6       <p>{count}</p>
7       <p><button onClick={()=>setCount(count+1)}>Add</button></p>
8       </>
9     )
10  }
11  export default App
```

Steps to implement Effect Hook

Step 1: Import the useEffect hook from React.
Step 2: Define the effect logic inside the useEffect callback.
Step 3: Optionally return a cleanup function to run when the component unmounts.
Step 4: Add dependencies in the second argument to control when the effect runs.

Welcome | index.js | App.js | Home.js | About.js

src > App.js > App

```
1   import { useEffect } from 'react';
2   const App = () => {
3     useEffect(() => {
4       const timerID = setInterval(() => console.log('Tick'), 1000);
5
6       return () => clearInterval(timerID); // Cleanup on unmount
7     }, []); // Empty dependency array ensures it runs only on mount
8
9     return <h1>Timer</h1>;
10  }
11  export default App
```

Steps to implement Ref Hook

Step 1: Import the useRef hook from React.
Step 2: Create a ref object using useRef.
Step 3: Attach the ref to a DOM element using the ref attribute.

Step 4: Use the current property of the ref to interact with the DOM element.

```
Welcome | index.js | App.js | Home.js | About.js
src > App.js > App
```

```jsx
import { useRef } from 'react';
const App = () => {
  const inputRef = useRef(null);

  const focusInput = () => {
    inputRef.current.focus();
  };
  return (
    <div>
      <input ref={inputRef} />
      <button onClick={focusInput}>Focus the input</button>
    </div>
  );
}
export default App
```

Steps to implement Callback Hook

Step 1: Import the useCallback hook from React.

Step 2: Define a function and wrap it with useCallback to memorize it.

Step 3: Specify dependencies to determine when the function should be re-created

```
Welcome | index.js | App.js | Home.js | About.js | Na
src > App.js > ...
```

```jsx
import { useCallback } from 'react';

const Button = ({ onClick }) => (
  <button onClick={onClick}>Click me</button>
);

const App = () => {
  const handleClick = useCallback(() => {
    console.log('Button clicked');
  }, []); // Empty dependency array ensures this function is memoized once

  return <Button onClick={handleClick} />;
};
export default App
```



Practical Activity 1.4.3 :Performing Performance Optimisation



Task:

1 : You are requested to Perform Performance Optimisation.
2 : Read key reading 1.4.3
3 : Apply safety precautions.
4 : Perform Performance Optimisation
5 : Present your work to the trainer
6 : Perform the task provided in application of learning 1.4



Key readings 1.4.3: Performing Performance Optimisation

- **Use useMemo**
  The useMemo hook is useful for optimizing expensive computations by memoizing their results until dependencies change.
  We identified the multiplication as an expensive computation.
  The useMemo hook caches the result, and recalculates only if count or multiplier changes
- **Steps to use useMemo**

Step 1. Identify expensive computations or derived data
Step 2. Wrap the computation inside a useMemo hook, passing the function that returns the computed value as the first argument.
Step 3. Provide a dependency array as the second argument, which will trigger re-computation when any of the dependencies change.
Step 4. Use the memorized value where necessary.

```jsx
import React, { useState, useMemo } from "react";

const Hooks = () => {
  const [count, setCount] = useState(0);
  const [multiplier, setMultiplier] = useState(2);

  // Expensive computation memoized with useMemo
  const expensiveResult = useMemo(() => {
    console.log("Performing expensive computation...");
    return count * multiplier * 1000;
  }, [count, multiplier]);

  return (
    <div>
      <h2>Expensive Computation Result: {expensiveResult}</h2>
      <button onClick={() => setCount(count + 1)}>Increment Count</button>
      <button onClick={() => setMultiplier(multiplier + 1)}>Change Multiplier</button>
    </div>
  );
};

export default Hooks;
```

- **Use useCallback Hook**

  The useCallback hook is useful to memorize functions, preventing them from being recreated on every render and passed down to child components.

  We wrapped the handleClick function in a useCallback hook so it isn't recreated on every render, avoiding unnecessary re-renders of the child component.

  ChildComponent is wrapped in React.memo to prevent re-rendering unless props change.

- **Steps to Use useCallback Hook**

Step 1. Identify callback functions or event handlers that are passed as props to child components or that are recreated on every render.

Step 2. Wrap the function definition inside a useCallback hook, passing the function as the first argument.

Step 3. Provide a dependency array as the second argument to control when the function gets recreated.

src > includes > JS Hooks.js > ...

```
1   import React, { useState, useCallback } from "react";
2   const ChildComponent = React.memo(({ handleClick }) => {
3     console.log("Child component re-rendered");
4     return <button onClick={handleClick}>Click Me</button>;
5   });
6   const ParentComponent = () => {
7     const [count, setCount] = useState(0);
8
9     // Memoize the callback with useCallback
10    const handleClick = useCallback(() => {
11      console.log("Button clicked");
12    }, []);
13
14    return (
15      <div>
16        <h2>Count: {count}</h2>
17        <button onClick={() => setCount(count + 1)}>Increment Count</button>
18        <ChildComponent handleClick={handleClick} />
19      </div>
20    );
21  };
22  export default ParentComponent;
23
```

- **Optimize rendering**

  Techniques like memoization of components, splitting complex components are used to avoid unnecessary state updates, and lazy loading.

  - ✓ **Memorize Components**: ComplexComponent is wrapped in React.memo to avoid unnecessary re-renders.
  - ✓ **Lazy Loading:** LazyLoadedComponent is loaded only when needed using lazy and Suspense, reducing the initial load time.
  - ✓ **Avoid Unnecessary State Updates:** The updateCount function directly increments the state, preventing extra logic that might trigger unwanted re-renders.

- **Steps to optimize rendering**

  Step 1. Memorize components

  Step 2. Split complex components

  Step 3. Embed Lazy loading method

```jsx
import React, { useState, lazy, Suspense } from "react";
// Lazy load heavy components
const LazyLoadedComponent = lazy(() => import("./LazyLoadedComponent"));
const ComplexComponent = React.memo(({ data }) => {
  console.log("Complex component re-rendered");
  return (
    <div>
      <h3>Complex Component</h3>
      <p>{data}</p>
    </div>
  );
});
const OptimizedComponent = () => {
  const [count, setCount] = useState(0);
  const [data, setData] = useState("Some data");
  const updateCount = () => setCount((prev) => prev + 1);
  return (
    <div>
      <h1>Optimized Rendering Example</h1>
      <button onClick={updateCount}>Increment Count</button>
      <Suspense fallback={<div>Loading...</div>}>
        <LazyLoadedComponent />
      </Suspense>
      <ComplexComponent data={data} />
    </div>
  );
};

export default OptimizedComponent;
```



Practical Activity 1.4.4 :Handling Complex State Logic



Task:

1 : You are requested to Handle Complex State Logic
2 : Read key reading 1.4.4
3 : Apply safety precautions.
4 : Implement complex State Logic
5 : Present your work to the trainer
6 : Perform the task provided in application of learning 1.4



Key readings 1.4.4: Handling Complex State Logic

Step 1. Import useReducer from React.

This hook manages complex state logic in React components.

import { useReducer } from 'react';

Step 2. Define the Initial State:

Create an initial state object.

const initialState = { count: 0 };

Step 3. Create the Reducer Function:

The reducer function accepts the current state and an action. Based on the action type, it returns a new state.

function reducer(state, action) {

switch (action.type) {

case 'increment':

return { count: state.count + 1 };

case 'decrement':

return { count: state.count - 1 };

default:

throw new Error();

}

}

Step 4. Use the useReducer Hook:

Inside the functional component (Hooks), invoke useReducer. It takes the reducer and the initialState, returning an array with the current state (state) and the dispatch function (dispatch).

const [state, dispatch] = useReducer(reducer, initialState);

Step 5. Handle Button Clicks:

Use dispatch to send actions to the reducer when the buttons are clicked.

Increment action:

<button onClick={() => dispatch({ type: 'increment' })}>+</button>

Decrement action:

<button onClick={() => dispatch({ type: 'decrement' })}>-</button>

The entire component will look as follows:

src > includes > JS Hooks.js > reducer

```
1   import { useReducer } from 'react';
2   const initialState = { count: 0 };
3   function reducer(state, action) {
4     switch (action.type) {
5       case 'increment':
6         return { count: state.count + 1 };
7       case 'decrement':
8         return { count: state.count - 1 };
9       default:
10        throw new Error();
11    }
12  }
13  const Hooks = () => {
14    const [state, dispatch] = useReducer(reducer, initialState);
15    return (
16      <div>
17        <p>Count: {state.count}</p>
18        <button onClick={() => dispatch({ type: 'increment' })}>+</button>
19        <button onClick={() => dispatch({ type: 'decrement' })}>-</button>
20      </div>
21    );
22  }
23  export default Hooks
```



Practical Activity 1.4.5 Managing Global State



Task:

1 : You are requested to go to the computer lab to manage global state
2 : Read key reading 1.4.5
3 : Apply safety precautions.
4 : Manage global state
5 : Present your work to the trainer
6 : Perform the task provided in application of learning 1.4



Key readings 1.4.5: Managing Global State

- **Context API**

The Context API is built into React and is a relatively simple solution for managing global state without additional libraries.

Steps to use Context API

Step 1. Create a Context:

Create a new context using React.createContext().

```
const GlobalStateContext = React.createContext();
```

Step 2. Provide the Context:

Wrap your app or components that need access to the global state with a Context.Provider.

```
const GlobalStateProvider = ({ children }) => {
const [state, setState] = useState(initialState);
return (
<GlobalStateContext.Provider value={{ state, setState }}>
{children}
</GlobalStateContext.Provider>
);
};
```

Step 3. Consume the Context:

Access the global state from any component using useContext.

```
const { state, setState } = useContext(GlobalStateContext);
```

The Context API works well for small to medium applications, but for complex state management, consider other options.

- **Redux**

Redux is a popular state management library that allows a single store to manage global state in a predictable way, using actions and reducers.

Steps to use Redux

Step 1. Install Redux:

Install the necessary packages.

```
npm install redux reactredux
```

Step 2. Create Reducers:

A reducer is a function that returns a new state based on the current state and the action dispatched.

```
const initialState = { count: 0 };
const counterReducer = (state = initialState, action) => {
```

```
switch (action.type) {
case 'INCREMENT':
return { count: state.count + 1 };
case 'DECREMENT':
return { count: state.count  1 };
default:
return state;
}
};
```

Step 3. Create a Redux Store:

Create a store using createStore.

```
import { createStore } from 'redux';
const store = createStore(counterReducer);
```

Step 4. Provide the Store:

Use Provider from reactredux to wrap your app and provide the Redux store.

```
import { Provider } from 'reactredux';
<Provider store={store}>
<App />
</Provider>;
```

Step 5. Connect Components:

Use the useSelector and useDispatch hooks to access and update the state from your components.

```
import { useSelector, useDispatch } from 'reactredux';
const Counter = () => {
const count = useSelector((state) => state.count);
const dispatch = useDispatch();
return (
<div>
<button onClick={() => dispatch({ type: 'DECREMENT' })}></button>
<span>{count}</span>
<button onClick={() => dispatch({ type: 'INCREMENT' })}>+</button>
</div>
);
};
```

Redux is suitable for larger applications where state mutations are complex, and you need predictability.

- **MobX**

MobX is a state management library that leverages observable state and reactions, making it more flexible and less boilerplate than Redux.

Steps to use MobX

Step 1. Install MobX:

Install MobX and its React bindings.

```
npm install mobx mobxreactlite
```

Step 2. Create Observable State:

Use makeAutoObservable to make your state observable.

```
import { makeAutoObservable } from 'mobx';
class CounterStore {
count = 0;
constructor() {
makeAutoObservable(this);
}
increment() {
this.count++;
}
decrement() {
this.count;
}
}
const counterStore = new CounterStore();
```

Step 3. Provide Store:

Use React's context to provide the MobX store to the components.

```
const StoreContext = React.createContext(counterStore);
const StoreProvider = ({ children }) => {
return (
<StoreContext.Provider value={counterStore}>
{children}
</StoreContext.Provider>
);
};
```

Step 4. Use Observer in Components:

Use the observer function to make components reactively update when the store changes.

```
import { observer } from 'mobxreactlite';
import { useContext } from 'react';
const Counter = observer(() => {
const store = useContext(StoreContext);
return (
<div>
<button onClick={() => store.decrement()}></button>
<span>{store.count}</span>
<button onClick={() => store.increment()}>+</button>
```

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

MobX allows more flexible and less restrictive state management, especially in applications that require more fluid, observable states.

- **Zustand**

Zustand is a lightweight state management library that is simpler and less boilerplateheavy than Redux or MobX.

Steps to use Zustand:

Step 1. Install Zustand:

Install the library.

```
npm install zustand
```

Step 2. Create a Store:

Create a Zustand store using create.

```
import create from 'zustand';
const useStore = create((set) => ({
count: 0,
increment: () => set((state) => ({ count: state.count + 1 })),
decrement: () => set((state) => ({ count: state.count  1 })),
}));
```

Step 3. Use the Store in Components:

Use the Zustand store in your React components with hooks.

```
const Counter = () => {
const { count, increment, decrement } = useStore();
return (
<div>
<button onClick={decrement}></button>
<span>{count}</span>
<button onClick={increment}>+</button>
</div>
);
};
```

Zustand is ideal for small to medium applications where you want simple and fast state management with minimal boilerplate.

- **Comparison:**
  - ✓ Context API: Best for small to medium apps with simple state.
  - ✓ Redux: Ideal for large, complex apps where state management needs to be predictable.
  - ✓ MobX: Great for apps with reactive state and complex workflows, offering flexibility.

- ✓ Zustand: A simple and lightweight solution for smaller apps with fewer state management requirements.



Points to Remember

Types of Hooks

1. State Hooks (useState)
2. Effect Hooks (useEffect)
3. Context Hooks (useContext)
4. Ref Hooks (useRef)
5. Callback Hooks (useCallback)

Additional Hooks:

1. useReducer
2. useMemo
3. useLayoutEffect
4. useImperativeHandle

Factors for Selecting the Right Hook

1. State Complexity
2. Side Effects
3. Optimization
4. Context Needs
5. DOM Access

Reasons for Combining Hooks

1. Modularity
2. Optimization
3. Code Organization
4. Reusability

Steps to implement State Hooks

1. Import the useState hook from React.
2. Use useState to define a state variable and a function to update it.
3. Use the state variable in your component's render logic.
4. Call the state updater function to modify the state based on an event.

Steps to implement Effect Hook

1. Import the useEffect hook from React.
2. Define the effect logic inside the useEffect callback.
3. Optionally return a cleanup function to run when the component unmounts.
4. Add dependencies in the second argument to control when the effect runs.

Steps to implement Ref Hook

1. Import the useRef hook from React.
2. Create a ref object using useRef.
3. Attach the ref to a DOM element using the ref attribute.
4. Use the current property of the ref to interact with the DOM element.

Steps to implement Callback Hook

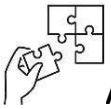
1. Import the useCallback hook from React.
2. Define a function and wrap it with useCallback to memorize it.
3. Specify dependencies to determine when the function should be re-created

To optimize performance, you have to use following hooks:

- useMemo
- useCallback
- useReducer

Global state:

- **Context API**
  - ✓ Create a Context
  - ✓ Provide the Context
  - ✓ Consume the Context
- **Redux**
  - ✓ Install Redux
  - ✓ Create Reducers
  - ✓ Create a Redux Store
  - ✓ Provide the Store
  - ✓ Connect Components
- **MobX**
  - ✓ Install MobX
  - ✓ Create Observable State
  - ✓ Provide Store
  - ✓ Use Observer in Components
- **Zustand:**
  - ✓ Install Zustand
  - ✓ Create a Store
  - ✓ Use the Store in Components



Application of learning 1.4.

ABC is a web Application development company located in Kimihurura – Kigali city, it's planning to create a customer's shopping site using React. As a React developer, the company has hired you to build a dashboard application using React. Utilize **State Hooks** to manage local component states and **Effect Hooks** for side effects like data fetching. To optimize performance, you have to strategically combine hooks and implement **Context API** for managing global state, this will allow seamless data access across components. As the app grows, they decide to integrate **Redux** for more complex state management, ensuring that components re-render efficiently and maintain synchronization with the global state. Throughout the process, employ **Ref Hooks** to access DOM elements directly and improve performance.

Indicative Content 1.5: Implementation of Events Handling

Duration: 10 hrs

Theoretical Activity 1.5.1: Description of ReactJS Events

Tasks:

1: You are requested to answer the following questions

i. What do you understand by Events?
ii. Explain different types of Events
iii. Describe Synthetic events
iv. Explain Event Bubbling
v. Explain the terms Debouncing and Throttling

2: Write your findings on papers, flip chart or chalkboard.

3: Present the findings/answers to the whole class or trainer

4: For more clarification, read the key readings 1.5.1

Key readings 1.5.1. Description of ReactJS Events

- **Description of key concepts**
  - ✓ **Events**

Events are actions that occur as a result of user interactions or browser actions, such as clicks, typing, submitting a form, or scrolling. React uses a system of event handling similar to DOM event handling but with a unified approach through Synthetic Events. These events help manage user inputs and trigger specific functions within a component.

✓ **Types of Events**

React supports many event types, similar to native HTML/DOM events, but they are implemented as synthetic events. Some common types include:

- Mouse Events:
  - onClick: Fires when an element is clicked.
  - onDoubleClick: Fires when an element is double clicked.
  - onMouseEnter: Fires when the mouse pointer enters an element.
  - onMouseLeave: Fires when the mouse pointer leaves an element.
- Keyboard Events:
  - onKeyDown: Fires when a key is pressed.
  - onKeyUp: Fires when a key is released.

- onKeyPress: Fires when a key is pressed, but this event is considered deprecated in favor of onKeyDown and onKeyUp.

- Form Events:
  - onChange: Fires when the value of an input element changes.
  - onSubmit: Fires when a form is submitted.
- Focus Events:
  - onFocus: Fires when an element receives focus.
  - onBlur: Fires when an element loses focus.
- Touch Events (for touch devices):
  - onTouchStart: Fires when a touch event begins.
  - onTouchMove: Fires when a touch event moves.
  - onTouchEnd: Fires when a touch event ends.
- Scroll Events:
  - onScroll: Fires when an element's scroll position changes.
- Clipboard Events:
  - onCopy: Fires when content is copied.
  - onPaste: Fires when content is pasted.
- Synthetic Events
  - Synthetic events are React's cross-browser wrapper around the native browser events. This system ensures that the events behave consistently across different browsers, simplifying event handling for developers. Synthetic events are normalized, meaning you can write event-handling code that works reliably across all browsers without worrying about discrepancies in how browsers implement events.

✓ **Event Bubbling**

Event bubbling refers to the process where an event triggered on an inner element propagates up through its parent elements. In React, when an event occurs on a child element, it bubbles up to its parent and can trigger any event handlers defined on the parent for that same event type.

✓ **Debouncing and Throttling**

Both debouncing and throttling are techniques used to control how often a function is executed in response to an event, improving performance when dealing with events like scrolling, typing, or resizing.

- Debouncing:

Debouncing ensures that a function is called only after a certain period of inactivity. This is useful for events that can fire in rapid succession, such as keystrokes or window resizing. The function will only execute after the user has stopped triggering the event for a specified delay.

- Throttling:

Throttling ensures that a function is executed at most once every specified interval, even if the event is triggered continuously. It limits the number of times the function is executed over time.
Example use case: Handling window scrolling events, where you want to limit how often a handler is called while the user is scrolling.



Practical Activity 1.5.2: Using Controlled Components



Task:

1 : You are requested to go to the computer lab to use Controlled Components
2 : Read key readings 1.5.2
3 : Apply safety precautions.
4 : Use Controlled Components
5 : Present your work to the trainer
6 : Perform the task provided in application of learning 1.5



Key readings 1.5.2: Using Controlled Components

Steps to use Controlled Components

Step 1. Create a functional or class component where you will manage your form state.

Step 2. Use the useState hook (for functional components) or this.state (for class components) to create state variables that will hold the values of your form inputs.

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

const Hooks = () => {
  // Step 2: Create state variables for form inputs
  const [formData, setFormData] = useState({
    username: '',
    email: '',
    password: '',
    agreeToTerms: false,
    gender: ''
  });
```

Step 3. Add input fields to your component and set their value prop to the corresponding state variable. This binds the input field's value to the state.

```jsx
  return (
    <form onSubmit={handleSubmit}>
      {/* Step 3: Bind state to input fields */}
```

Step 4. Implement an onChange event handler for each input element that updates the state. This ensures the component re-renders with the new value.

```jsx
// Step 4: Event handler to update state when input changes
const handleInputChange = (e) => {
  const { name, value, type, checked } = e.target;
  setFormData({
    ...formData,
    [name]: type === 'checkbox' ? checked : value // Handle checkbox
  });
};
```

Step 5. Create a function to handle form submission. This function can prevent the default form submission behaviour and access the state values.

```jsx
// Step 5: Form submission handler
const handleSubmit = (e) => {
  e.preventDefault(); // Prevent default form submission
  // You can now access formData values here for form submission or validation
  console.log('Form submitted with data:', formData);
};
```

Step 6. Use the state values in your component logic, such as for form validation, submission, or displaying the values elsewhere.

Step 7. You can repeat these steps for other input types like checkboxes, radio buttons, or selects.

Full Sample source code

```
import React, { useState } from 'react';
const Hooks = () => {
 // Step 2: Create state variables for form inputs
 const [formData, setFormData] = useState({
  username: '',
  email: '',
  password: '',
  agreeToTerms: false,
  gender: ''
 });
 // Step 4: Event handler to update state when input changes
 const handleInputChange = (e) => {
  const { name, value, type, checked } = e.target;
  setFormData({
   ...formData,
   [name]: type === 'checkbox' ? checked : value // Handle checkbox
  });
 };
 // Step 5: Form submission handler
 const handleSubmit = (e) => {
  e.preventDefault(); // Prevent default form submission
  // You can now access formData values here for form submission or validation
  console.log('Form submitted with data:', formData);
 };
 return (
  <form onSubmit={handleSubmit}>
   {/* Step 3: Bind state to input fields */}
   <div>
    <label>Username: </label>
    <input
     type="text"
     name="username"
     value={formData.username}
     onChange={handleInputChange}
    />
   </div>
```

```
<div>
  <label>Email: </label>
  <input
    type="email"
    name="email"
    value={formData.email}
    onChange={handleInputChange}
  />
</div>
<div>
  <label>Password: </label>
  <input
    type="password"
    name="password"
    value={formData.password}
    onChange={handleInputChange}
  />
</div>

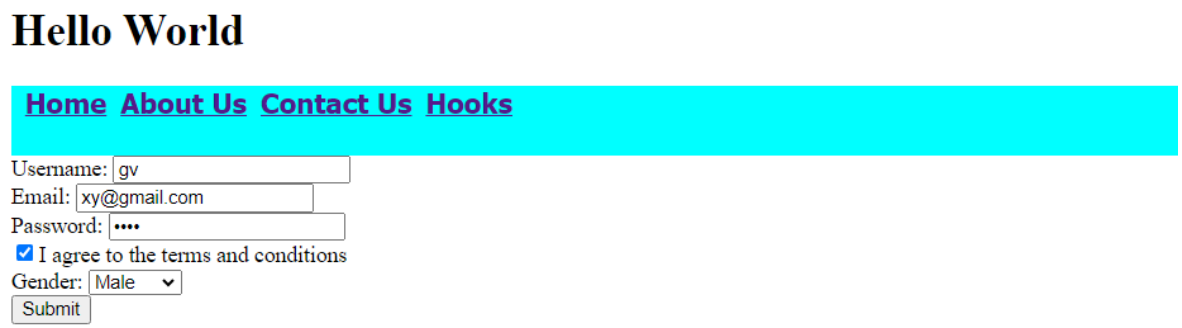
<div>
  <label>
    <input
      type="checkbox"
      name="agreeToTerms"
      checked={formData.agreeToTerms}
      onChange={handleInputChange}
    />
    I agree to the terms and conditions
  </label>
</div>

<div>
  <label>Gender: </label>
  <select
    name="gender"
    value={formData.gender}
    onChange={handleInputChange}
  >
    <option value="">Select</option>
    <option value="male">Male</option>
```

```
      <option value="female">Female</option>
     </select>
    </div>

    <button type="submit">Submit</button>
   </form>
  );
};
export default Hooks;
```

Output of the code:





Practical Activity 1.5.3: Passing Arguments to Event Handlers



Task:

1 : You are requested to go to the computer lab to pass arguments to events

2 : Read key readings 1.5.3

3 : Apply safety precautions.

4 : Pass arguments to events

5 : Present your work to the trainer

6 : Perform the task provided in application of learning 1.5



Key readings 1.5.3 : Passing Arguments to Event Handlers

Steps to Pass Arguments to Event Handlers

Step 1. Create a functional or class component where you will manage your form state.

Step 2. Use the useState hook (for functional components) or this.state (for class components) to create state variables that will hold the values of your form inputs.

src > includes > JS Hooks.js > Hooks

```
1   import React, { useState } from 'react';
2
3   const Hooks = () => {
4     // 1. Initialize state variables for each form field
5     const [name, setName] = useState('');
6     const [email, setEmail] = useState('');
7     const [password, setPassword] = useState('');
8     const [gender, setGender] = useState('');
9     const [isSubscribed, setIsSubscribed] = useState(false);
10
```

Step 3. Add input fields to your component and set their value prop to the corresponding state variable. This binds the input field's value to the state.

Step 4. Implement an onChange event handler for each input element that updates the state. This ensures the component re-renders with the new value.

```
// 2. Event handler for input changes
const handleInputChange = (event) => {
  const { name, value, type, checked } = event.target;

  // Update the corresponding state based on input type
  if (type === 'checkbox') {
    setIsSubscribed(checked);
  } else if (name === 'name') {
    setName(value);
  } else if (name === 'email') {
    setEmail(value);
  } else if (name === 'password') {
    setPassword(value);
  } else if (name === 'gender') {
    setGender(value);
  }
};
```

Step 5. Create a function to handle form submission. This function can prevent the default form submission behaviour and access the state values.

Step 6. Use the state values in your component logic, such as for form validation, submission, or displaying the values elsewhere.

Step 7. You can repeat these steps for other input types like checkboxes, radio buttons, or selects.

Sample code

```
import React, { useState } from 'react';
const Hooks = () => {
 // 1. Initialize state variables for each form field
 const [name, setName] = useState('');
 const [email, setEmail] = useState('');
 const [password, setPassword] = useState('');
 const [gender, setGender] = useState('');
 const [isSubscribed, setIsSubscribed] = useState(false);
 // 2. Event handler for input changes
 const handleInputChange = (event) => {
  const { name, value, type, checked } = event.target;

  // Update the corresponding state based on input type
  if (type === 'checkbox') {
   setIsSubscribed(checked);
  } else if (name === 'name') {
   setName(value);
  } else if (name === 'email') {
   setEmail(value);
  } else if (name === 'password') {
   setPassword(value);
  } else if (name === 'gender') {
   setGender(value);
  }
 };

 // 3. Form submit handler
 const handleSubmit = (event) => {
  event.preventDefault();

  // Access state values
  console.log('Form submitted with the following values:');
  console.log('Name:', name);
  console.log('Email:', email);
  console.log('Password:', password);
  console.log('Gender:', gender);
```

```
  console.log('Subscribed:', isSubscribed);

  // You can add form validation or submit logic here
};

return (
  <form onSubmit={handleSubmit}>
    <div>
      <label>Name:</label>
      <input
        type="text"
        name="name"
        value={name}
        onChange={handleInputChange}
      />
    </div>

    <div>
      <label>Email:</label>
      <input
        type="email"
        name="email"
        value={email}
        onChange={handleInputChange}
      />
    </div>

    <div>

      <label>Password:</label>

      <input

        type="password"

        name="password"

        value={password}

        onChange={handleInputChange}

      />

    </div>
```

```
    <div>
      <label>Gender:</label>
      <select name="gender" value={gender} onChange={handleInputChange}>
        <option value="">Select Gender</option>
        <option value="male">Male</option>
        <option value="female">Female</option>
      </select>
    </div>
    <div>
      <label>Subscribe to newsletter:</label>
      <input
        type="checkbox"
        name="isSubscribed"
        checked={isSubscribed}
        onChange={handleInputChange}
      />
    </div>
    <button type="submit">Submit</button>
  </form>
 );
};

export default Hooks;
```

Output of the code





Practical Activity 1.5.4: Using Custom Hooks for Event Listeners



Task:

1 : You are requested to go to the computer lab to use Custom Hooks for Event Listeners
2 : Read key readings 1.5.4
3 : Apply safety precautions.
4 : Use Custom Hooks for Event Listeners
5 : Present your work to the trainer
6 : Perform the task provided in application of learning 1.5



Key readings 1.5.4: Using Custom Hooks for Event Listeners

Steps to use Custom Hooks for Event Listeners

Step 1. Create the useEventListener Hook:

```
import { useEffect, useRef } from 'react';
function useEventListener(eventName, handler, element = window) {
  // Create a ref that stores the handler
  const savedHandler = useRef();
  // Update ref.current value if handler changes
  useEffect(() => {
    savedHandler.current = handler;
  }, [handler]);
  useEffect(() => {
    // Make sure the element supports addEventListener
```

```
  const isSupported = element && element.addEventListener;
  if (!isSupported) return;
  // Create an event listener that calls the saved handler
  const eventListener = (event) => savedHandler.current(event);
  // Add event listener
  element.addEventListener(eventName, eventListener);
  // Remove event listener on cleanup
  return () => {
    element.removeEventListener(eventName, eventListener);
  };
 }, [eventName, element]); // Rerun if eventName or element changes
}
```

Step 2. Use the useEventListener Hook in a Component:

```
import React, { useState, useRef } from 'react';
import useEventListener from './useEventListener'; // Import your custom hook
function ExampleComponent() {
 const [coords, setCoords] = useState({ x: 0, y: 0 });
 const buttonRef = useRef(null);
 // Handler for window mousemove event
 const handleMouseMove = (event) => {
  setCoords({ x: event.clientX, y: event.clientY });
 };
 // Attach mousemove event to the window
 useEventListener('mousemove', handleMouseMove);
 // Handle button click
 const handleButtonClick = () => {
  console.log('Button clicked!');
 };
 // Attach click event to the button
 useEventListener('click', handleButtonClick, buttonRef.current);
 return (
  <div>
   <p>Mouse position: {coords.x}, {coords.y}</p>
   <button ref={buttonRef}>Click me!</button>
  </div>
 );
}
export default ExampleComponent;
```

Note: Handling Dynamic Elements:

If the event listener is attached to dynamic or conditionally rendered elements, the hook should gracefully handle null or undefined values (which it does by checking isSupported). For instance, in the above case, buttonRef.current will initially be null until the button is mounted.
The hook will only add the event listener once the element exists, and will clean it up when the component is unmounted or the element is removed.



Practical Activity 1.5.5: Handling Events on Dynamic Lists



Task:

1 : You are requested to go to the computer lab to Handle Events on Dynamic Lists
2 : Read key readings 1.5.5
3 : Apply safety precautions.
4 : Handle Events on Dynamic Lists
5 : Present your work to the trainer
6 : Perform the task provided in application of learning 1.5



Key readings 1.5.5: Handling Events on Dynamic Lists

Steps to Handle Events on Dynamic Lists

Step 1. Set Up State for the List

Create a state variable to manage the dynamic list. This is usually done using the useState hook.

```
const [list, setList] = useState([]);
```

Step 2. Attach Event Handlers to List Items

Each list item should have an event handler attached (e.g., onClick, onMouseOver). You can pass a handler function to each item in the list.

```
list.map((item, index) => (
  <li key={index} onClick={() => handleItemClick(item)}>
    {item.name}
  </li>
));
```

Step 3. Handle Dynamic Updates (Add/Delete Items)

Use the setList function to add or delete items from the list dynamically.

```
const addItem = (newItem) => {
  setList((prevList) => [...prevList, newItem]);
```

```
};
const removeItem = (id) => {
  setList((prevList) => prevList.filter(item => item.id !== id));
};
```

Step 4. Pass Data to Event Handlers

Pass the appropriate data (such as the clicked item's data) into the event handler.

```
const handleItemClick = (item) => {
  console.log(Item clicked: ${item.name});
};
```

Step 5. Conditional Rendering for Dynamic Behavior

Depending on certain conditions, you might want to render different content or apply different styles.

```
{list.length > 0 ? (
  <ul>{/* Render list */}</ul>
) : (
  <p>No items available</p>
)}
```

Step 6. Update the State on Event Trigger

When an event is triggered (like clicking an item), ensure it updates the state appropriately.

```
const handleItemClick = (item) => {
  setSelectedItem(item);
};
```

Step 7. Use Proper key Props in List Rendering

When rendering lists, ensure each item has a unique key for optimal performance.

```
list.map((item) => (
  <li key={item.id}>{item.name}</li>
));
```

Step 8. Use useCallback or memo to Prevent Unnecessary Re-renders

If the functions passed to components are recreated unnecessarily, it can lead to performance issues. Use useCallback to memorize functions.

```
const handleItemClick = useCallback((item) => {
  // Handle click event
}, []);
```

Note: You can also use React.memo to optimize components that do not need to re-render on every state change.

```
const ListItem = React.memo(({ item, onClick }) => (
  <li onClick={() => onClick(item)}>{item.name}</li>
));
```



Points to Remember

- **Types of Events include** User Interface Events, Focus and Blur Events, Form Events, Mouse Events, Keyboard Events, Synthetic Events
- **Event Bubbling**
- **Debouncing and throttling**

Steps to use Controlled Components

- Create a functional or class component where you will manage your form state.
- Use the useState hook (for functional components) or this.state (for class components) to create state variables that will hold the values of your form inputs.
- Add input fields to your component and set their value prop to the corresponding state variable. This binds the input field's value to the state.
- Implement an onChange event handler for each input element that updates the state. This ensures the component re-renders with the new value.
- Create a function to handle form submission. This function can prevent the default form submission behaviour and access the state values.
- Use the state values in your component logic, such as for form validation, submission, or displaying the values elsewhere.
- You can repeat these steps for other input types like checkboxes, radio buttons, or selects.

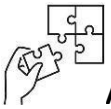
Steps to Pass Arguments to Event Handlers

- Create a functional or class component where you will manage your form state.
- Use the useState hook (for functional components) or this.state (for class components) to create state variables that will hold the values of your form inputs.
- Add input fields to your component and set their value prop to the corresponding state variable. This binds the input field's value to the state.
- Implement an onChange event handler for each input element that updates the state. This ensures the component re-renders with the new value.
- Create a function to handle form submission. This function can prevent the default form submission behaviour and access the state values.
- Use the state values in your component logic, such as for form validation, submission, or displaying the values elsewhere.
- You can repeat these steps for other input types like checkboxes, radio buttons, or selects.

Steps to use Custom Hooks for Event Listeners

- Set Up State for the List
- Attach Event Handlers to List Items

- Handle Dynamic Updates (Add/Delete Items)
- Pass Data to Event Handlers
- Conditional Rendering for Dynamic Behavior
- Update the State on Event Trigger
- Use Proper key Props in List Rendering
- Use useCallback or memo to Prevent Unnecessary Re-renders



Application of learning 1.5.

ABC company is a company located in Niboye- Sector, Kicukiro District in Kigali City, It has many employs who are assigned daily task, Assume you have been hired as their web developer to develop them a Todo List app in React that will help the employee manager to add, remove, and mark tasks as complete. Implement event handling for user interactions, such as clicking buttons or pressing keys. Use debouncing for search input to improve performance and throttling for scrolling events to manage resource usage. Utilize controlled components to maintain the state of inputs. Pass arguments to event handlers using arrow functions and the bind method. Create custom hooks for managing event listeners, and ensure event handling works seamlessly on a dynamically generated list of tasks

Indicative Content 1.6: Implementation of API Integration

Duration: 5 hrs

Theoretical Activity 1.6.1: Description of API Integration

Tasks:

1: You are requested to answer the following questions

i. What is an API?
ii. What are importance of API Integration?
iii. What are the common methods used for fetching data from APIs in React.js?

2: Write your findings on papers, flip chart or chalkboard.

3: Present the findings/answers to the whole class or trainer

4: For more clarification, read the key readings 1.6.1

Key readings 1.6.1. Description of API Integration

- **API in React.js**

API (Application Programming Interface) allows your application to interact with external services, servers, or databases. APIs enable the communication between the front-end (React) and the back-end systems. This is commonly used to fetch or send data from/to external sources like databases, other web services, or cloud platforms. For instance, if you're building a weather app in React, you would call a weather API to fetch data about the weather and display it to the user.

- **Importance of API**

✓ Dynamic Data: APIs allow React apps to retrieve dynamic data, such as user information, product details, or live weather, enabling the app to provide real-time updates without reloading.

✓ Scalability: React apps can connect to various external APIs to scale functionality, like integrating third-party services (e.g., payment gateways, social logins).

✓ Separation of Concerns: API integration ensures a separation between front-end (UI) and back-end (data/services). This makes the application architecture more modular and maintainable.

- ✓ Interactivity: By integrating APIs, react apps can make the user experience more interactive, such as updating data without refreshing the entire page (using AJAX or Fetch API).
- ✓ Reusability: API-based architectures allow you to reuse data and functionality across different platforms, such as web, mobile, or desktop applications.

- **Methods used for fetching data from APIs**

  - ✓ Fetch API

The Fetch API is a built-in JavaScript feature used to make HTTP requests. It returns a promise, which you can resolve to get the response data.

- ✓ Axios

Axios is a popular third-party library for making HTTP requests, known for its simplicity and ease of use. Unlike Fetch, it automatically converts response data to JSON and has better support for handling errors.

- ✓ Async/Await:

Often used in combination with Fetch or Axios for cleaner and more readable asynchronous code.

- ✓ React Query:

A powerful library for managing server-side state and handling API requests with caching, background syncing, and many other features.



Practical Activity 1.6.2: Installing dependencies (Axios)



Task:

1 : You are requested to go to the computer lab to install dependencies
2 : Read key readings 1.6.2
3 : Apply safety precautions.
4 : Install dependencies
5 : Present your work to the trainer
6 : Perform the task provided in application of learning 1.6



Key readings 1.6.2: Installing dependencies (Axios)

Steps to Install dependencies

Step 1. Ensure Node.js and npm are Installed

Run the following commands to verify if node or npm are installed

node v

npm v

Step 2. Create a React Project

Use npx to create a new React project.

npx create-react-app my-api-project

This will generate a React project called my-api-project.

Step 3. Install Project Dependencies for API Integration

After your project has been created, navigate to the project directory:

cd my-api-project

Install necessary dependencies for making API requests, such as axios. Axios is a popular library for making HTTP requests from React.

npm install axios

Step 4. Add New Dependencies for API Integration

At this point, axios is installed, and you can use it to perform API integration. There might be other packages depending on your specific requirements, such as react-router if you're working with routes, but for now, we are focusing on basic API calls.

Step 5. Create a Functional Component for API Integration

create a simple functional component that fetches data from an API.

- Open the src folder.
- Inside the src folder, create a new file called ApiComponent.js.

```
import React, { useEffect, useState } from 'react';
import axios from 'axios';
const ApiComponent = () => {
  const [data, setData] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    // Replace with your API endpoint
    const fetchData = async () => {
      try {
        const response = await axios.get('https://jsonplaceholder.typicode.com/posts');
```

```
      setData(response.data);
      setLoading(false);
    } catch (error) {
      setError('Failed to fetch data');
      setLoading(false);
    }
  };
  fetchData();
 }, []);
 if (loading) return <p>Loading...</p>;
 if (error) return <p>{error}</p>;
 return (
  <div>
   <h1>API Data</h1>
   <ul>
    {data.map(item => (
     <li key={item.id}>{item.title}</li>
    ))}
   </ul>
  </div>
 );
};
export default ApiComponent;
```

In this example:

- useState is used to manage the state of the data, loading status, and errors.
- useEffect triggers the API call after the component is rendered.
- axios.get() is used to make a GET request to the API endpoint.

Step 6. Update App.js to Render the API Component

Open the src/App.js file and replace its content with the following to render ApiComponent:

```
import React from 'react';
import ApiComponent from './ApiComponent';
function App() {
 return (
  <div className="App">
   <ApiComponent />
  </div>
 );
}
export default App;
```

Step 7. Run the Project

Now you can run the project to test the API integration.

npm start



Practical Activity 1.6.3: Defining and Grouping API Calls



Task:

1 : You are requested to go to the computer lab to define and Group API Calls
2 : Read key readings 1.6.3
3 : Apply safety precautions.
4 : Define and Group API Calls
5 : Present your work to the trainer
6 : Perform the task provided in application of learning 1.6



Key readings 1.6.3 Steps to Defining and Grouping API Calls

Step 1. Install Axios (or use Fetch API) for API Calls

If using Axios, install it with the following command:

npm install axios

Step 2. Create a Separate API Utility File

Create a dedicated file (e.g., api.js) to handle all API interactions. This keeps the codebase organized and ensures API logic is reusable.

src/utils/api.js

Step 3. Define API Base URL and Common Configuration

Set a base URL for your API and create common configurations, like headers and timeouts, to avoid repeating this setup in every call.

```
import axios from 'axios';
const apiClient = axios.create({
 baseURL: 'https://api.example.com', // Replace with your API URL
 timeout: 10000,
headers: {
'Content-Type': 'application/json',
},
```

```
});
  export default apiClient;
```

Step 4. Define API Endpoints

Define functions for each API endpoint in the utility file. For example, you can create functions for GET, POST, PUT, DELETE requests, and map them to specific URLs.

```
export const getUsers = () => apiClient.get('/users');
  export const createUser = (data) => apiClient.post('/users', data);
  export const updateUser = (id, data) => apiClient.put(/users/${id}, data);
  export const deleteUser = (id) => apiClient.delete(/users/${id});
```

Step 5. Use API Calls in React Components

Import the functions from the utility file and use them inside your React components. Handle responses with async/await or .then()/.catch().

```
import React, { useEffect, useState } from 'react';
import { getUsers } from './utils/api';
const UsersList = () => {
const [users, setUsers] = useState([]);
useEffect(() => {
const fetchUsers = async () => {
try {
const response = await getUsers();
setUsers(response.data);
 } catch (error) {
console.error('Error fetching users:', error);
}
};
fetchUsers();
}, []);
return (
<ul>
{users.map(user => (
<li key={user.id}>{user.name}</li>
))}
</ul>
);
};
  export default UsersList;
```

Step 6. Perform Error Handling and Response Interception

Implement global error handling and intercept responses to manage authentication tokens or retry failed requests.

```
apiClient.interceptors.response.use(
```

```
(response) => response,
(error) => {
if (error.response && error.response.status === 401) {
// Handle unauthorized access (e.g., redirect to login)
}
return Promise.reject(error);
}
);
```

Step 7. Handle Caching and Optimizations

Implement caching strategies like memoizing API responses (e.g., using React Query or Redux). This minimizes unnecessary API calls and improves performance.

Example with React Query:

```
npm install react-query
import { useQuery } from 'react-query';
import { getUsers } from './utils/api';
const UsersList = () => {
const { data: users, isLoading, error } = useQuery('users', getUsers);
if (isLoading) return <div>Loading...</div>;
if (error) return <div>Error loading users</div>;
return (
<ul>
{users.map(user => (
<li key={user.id}>{user.name}</li>
))}
</ul>
);
};
    export default UsersList;
```



Practical Activity 1.6.4: Handling Data Fetching and Responses



Task:

1 : You are requested to go to the computer lab to Handle Data Fetching and Responses

2 : Read key readings 1.6.4

3 : Apply safety precautions.

4 : Handle Data Fetching and Responses

5 : Present your work to the trainer

6 : Perform the task provided in application of learning 1.6



Key readings 1.6.4: Handling Data Fetching and Responses

Steps to Handle Data Fetching and Responses

Step 1. Set Up State to Manage Data and Loading Status

```
import React, { useState, useEffect } from 'react';
const ExampleComponent = () => {
  const [data, setData] = useState(null);  // State to store fetched data
  const [loading, setLoading] = useState(true);  // State to manage loading status
  const [error, setError] = useState(null);  // State to manage errors
};
```

Step 2. Use useEffect to Fetch Data When Component Loads

Use the useEffect hook to perform side effects like fetching data when the component is mounted. You can make use of async/await for cleaner code and error handling.

```
useEffect(() => {
  const fetchData = async () => {
    try {
      setLoading(true);
```

```
    const response = await fetch('https://api.example.com/data');
    if (!response.ok) {
      throw new Error('Failed to fetch');
    }
    const result = await response.json();
    setData(result);
   } catch (err) {
    setError(err.message);
   } finally {
    setLoading(false);
   }
  };
  fetchData();
}, []);
```

Step 3. Render Based on State (Loading, Error, or Data)

In your component's render function, you'll conditionally render content based on the loading, error, and data states.

```
return (
  <div>
   {loading && <p>Loading...</p>}
   {error && <p>Error: {error}</p>}
   {data && <div>{JSON.stringify(data)}</div>}
  </div>
);
```

Step 4. Handle POST or Other Methods

For handling other HTTP methods like POST, you can create a function that sends data using fetch. You might need additional state to manage the post request status.

```
const postData = async (newData) => {
```

```
  try {
    const response = await fetch('https://api.example.com/data', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
      },
      body: JSON.stringify(newData),
    });
    if (!response.ok) {
      throw new Error('Failed to post data');
    }
    const result = await response.json();
    console.log('Post successful:', result);
  } catch (err) {
    console.error('Error posting data:', err.message);
  }
};
// Example of using postData when a button is clicked
return (
  <div>
    <button onClick={() => postData({ name: 'New Data' })}>Submit Data</button>
  </div>
);
```



Practical Activity 1.6.5: Error Handling



Task:

1 : You are requested to go to the computer lab to perform Error Handling

2 : Read key readings 1.6.5

3 : Apply safety precautions.

4 : Perform Error Handling

5 : Present your work to the trainer

6 : Perform the task provided in application of learning 1.6



Key readings 1.6.5: Error Handling

Steps to perform Error Handling

Step 1. Choose an API Request Method

You can choose between native fetch or third-party libraries like axios. Both are widely used for making HTTP requests in React apps.

Fetch: Built into modern browsers.

Axios: Offers more features like automatic JSON parsing, request/response interceptors, etc.

```
// Using Fetch
fetch('https://api.example.com/data')
  .then(response => response.json())
  .catch(error => console.error('Error:', error));
// Using Axios
import axios from 'axios';
axios.get('https://api.example.com/data')
  .then(response => console.log(response))
  .catch(error => console.error('Error:', error));
```

Step 2. Set Up State for Loading, Error, and Data

In your component, set up state to handle loading, error, and data responses.

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

const MyComponent = () => {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);
};
```

Step 3. Make the API Call with Error Handling

When making the API call, make sure to handle errors properly and update the state accordingly. You can do this inside a useEffect hook for fetching data when the component mounts.

```
useEffect(() => {
  const fetchData = async () => {
    setLoading(true);  // Start loading
    setError(null);    // Clear previous errors

    try {
      const response = await fetch('https://api.example.com/data');
          if (!response.ok) {  // Handle non-2xx status codes
       throw new Error(HTTP error! Status: ${response.status});
      }
      const result = await response.json();
      setData(result);  // Set fetched data
    } catch (err) {
      setError(err.message);  // Set error message
    } finally {
      setLoading(false);  // End loading
```

```
  }
 };
 fetchData();
}, []);
```

Step 4. Handle Errors Based on Status Code

You can also handle specific HTTP status codes differently.

```
try {
 const response = await fetch('https://api.example.com/data');
  if (response.status === 404) {
  throw new Error('Resource not found');
 } else if (response.status === 500) {
  throw new Error('Server error');
 }
 const result = await response.json();
 setData(result);
} catch (err) {
 setError(err.message);
}
```

For Axios:

```
axios.get('https://api.example.com/data')
 .then(response => setData(response.data))
 .catch(error => {
  if (error.response) {
   if (error.response.status === 404) {
    setError('Resource not found');
   } else if (error.response.status === 500) {
    setError('Server error');
```

```
    }
   } else {
    setError(error.message);
   }
  })
  .finally(() => setLoading(false));
```

Step 5. Show Loading, Error, and Data States in the UI

Display different states based on the values of loading, error, and data.

```
return (
  <div>
   {loading && <p>Loading...</p>}  {/ Show loading state /}
   {error && <p>Error: {error}</p>}  {/ Show error message /}
   {data && (
    <div>
     {/ Render data /}
     <p>Data: {JSON.stringify(data)}</p>
    </div>
   )}
  </div>
);
```

Example with Axios:

```
import React, { useState, useEffect } from 'react';
import axios from 'axios';
const MyComponent = () => {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);
```

```
 useEffect(() => {
   axios.get('https://api.example.com/data')
     .then(response => setData(response.data))
     .catch(error => setError(error.message))
     .finally(() => setLoading(false));
 }, []);
 return (
   <div>
     {loading && <p>Loading...</p>}
     {error && <p>Error: {error}</p>}
     {data && <div>{JSON.stringify(data)}</div>}
   </div>
 );
};
export default MyComponent;
```



Practical Activity 1.6.6: Handling Asynchronous and Concurrency



Task:

1 : You are requested to go to the computer lab to Handle Asynchronous and Concurrency

2 : Read key readings 1.6.6

3 : Apply safety precautions.

4 : Handle Asynchronous and Concurrency

5 : Present your work to the trainer

6 : Perform the task provided in application of learning 1.6



Key readings 1.6.6:Handling Asynchronous and Concurrency

Step 1. Select the Best Asynchronous Feature: Promises

React.js often handles asynchronous operations using promises or the async/await syntax. In this case, we'll focus on using promises with .then() for clarity and chainability.

Promises allow handling asynchronous tasks like API requests and give clear structure using .then(), .catch(), and .finally().

Step 2. Create or Fetch Asynchronous Data (API Call)

You can create an asynchronous API call using fetch() or libraries like Axios to get data. Here's an example of fetching data using fetch():

```
const fetchData = () => {
  return fetch('https://api.example.com/data')
    .then(response => response.json())
    .then(data => data)
    .catch(error => {
      console.error("Error fetching data: ", error);
    });
}
```

Step 3. Handle Async Operations in useEffect Hook

To fetch data when the component mounts, you can use the useEffect hook, which will run side effects such as API calls when a component renders.

```
import { useEffect, useState } from 'react';
const MyComponent = () => {
  const [data, setData] = useState([]);
  useEffect(() => {
    fetchData().then(result => {
      setData(result);
    });
```

```
  }, []);

    return (

   <div>

    {data.length > 0 ? (

      data.map(item => <div key={item.id}>{item.name}</div>)

    ) : (

      <p>Loading...</p>

    )}

   </div>

  );

};
```

Step 4. Understand the Flow

Component mounts → useEffect() runs → fetchData() is called → .then() processes the promise.

State is updated asynchronously when the promise is fulfilled, causing the component to re-render with the updated data.

Step 5. Use .then() Instead of async/await

Using .then() is preferred in certain situations for promise chaining. Here's how you can handle an API request using .then():

```
const fetchData = () => {

  return fetch('https://api.example.com/data')

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

    .then(data => {

      // Do something with the data

      return data;

    })

    .catch(error => console.error('Error fetching data:', error));

};
```

```
useEffect(() => {
  fetchData().then(result => {
    console.log("Data fetched:", result);
  });
}, []); // Run only once when the component mounts
```

Step 6. Fetch Data on User Interaction

You may want to fetch data on user interaction, such as clicking a button. This can be done by attaching an event handler to trigger the API call:

```
const [data, setData] = useState([]);
  const handleFetch = () => {
  fetchData().then(result => {
    setData(result);
  });
};
  return (
  <div>
    <button onClick={handleFetch}>Fetch Data</button>
    {data.length > 0 && data.map(item => <div key={item.id}>{item.name}</div>)}
  </div>
);
```



Practical Activity 1.6.7: Performing API Security and testing



Task:

1 : You are requested to go to the computer lab to Perform API Security and testing

2 : Read key readings 1.6.7

3 : Apply safety precautions.

4 : Perform API Security and testing

5 : Present your work to the trainer

6 : Perform the task provided in application of learning 1.6



Key readings 1.6.7: Performing API Security and testing

Step 1. Use HTTPS for Secure Communication

- Ensure all API requests are made over HTTPS to encrypt communication and protect sensitive data in transit.
- Update your API base URL to use `https://`.

Step 2. Authentication and Authorization

- Implement OAuth 2.0 or JWT (JSON Web Token) for secure authentication.
- Securely store tokens in memory or secure storage like HTTP-only cookies (preferred over localStorage for security).
- Use authorization headers to verify access permissions for API resources (e.g., `Authorization: Bearer <token>`).

Step 3. Input Validation and Sanitization

- On both the client (React.js) and server side, validate all input data (e.g., form data, URL params) to prevent SQL Injection or XSS (Cross-site scripting).
- Use libraries like validator.js or custom validation methods for sanitizing user inputs.

Step 4. Rate Limiting and Throttling

- Implement rate limiting on the server to prevent Denial of Service (DoS) attacks.
- Use middleware like express-rate-limit on the backend to limit requests from a single IP.

Step 5. CORS (Cross-Origin Resource Sharing)

- Configure your API's CORS policy to allow requests only from trusted domains.
- Avoid wildcard (``) domains for sensitive endpoints.
- In React, use the `proxy` setting or configure your server to handle preflight CORS requests appropriately.

Step 6. CSRF (Cross-Site Request Forgery) Protection

- For APIs that use cookies for session-based authentication, implement CSRF tokens to protect from cross-site attacks.
- Use libraries like csurf in your backend to generate and verify CSRF tokens.
- Store the token in HTTP headers for secure communication.

Step 7. Use Security Headers

- Configure security headers to harden your API against attacks:
- Content-Security-Policy (CSP) to prevent XSS.
- X-Frame-Options to prevent clickjacking.
- Strict-Transport-Security (HSTS) to enforce HTTPS.
- Libraries like helmet.js can help set these headers easily.

Step 8. Encryption of Sensitive Data

- Use AES or other encryption algorithms to encrypt sensitive data before storing or transmitting.
- For sensitive user data (like passwords), ensure encryption using bcrypt or PBKDF2.

Step 9. Logging and Monitoring

- Log all API requests and responses, particularly authentication events, data access, and errors.
- Implement monitoring tools like New Relic, Datadog, or ELK stack to detect anomalies and security breaches in real-time.

Step 10. Security Testing

- Integrate automated security testing tools, such as:
    - OWASP ZAP for vulnerability scanning.
    - Postman for API testing.
    - Burp Suite for penetration testing.
- Set up unit tests for authorization rules and data validation in your React components using testing frameworks like Jest and Enzyme.

Step 11. Penetration Testing

- Regularly conduct manual penetration testing using tools like Kali Linux and Metasploit.
- Test for common vulnerabilities, such as SQL injection, XSS, insecure direct object references (IDOR), and others from the OWASP Top 10.



Points to Remember

.

- APIs in React.js enable communication between the front-end and external services, fetching or sending data like user info or weather updates.
- Common methods for fetching API data include Fetch API, Axios, async/await for asynchronous operations, and React Query for advanced server-side state management with caching and syncing features.

Steps to install dependencies

- Ensure Node.js and npm are Installed
- Create a React Project
- Install Project Dependencies for API Integration
- Add New Dependencies for API Integration
- Create a Functional Component for API Integration
- Update App.js to Render the API Component
- Run the Project

Steps to define and group API calls

- Install Axios (or Fetch API) for API Calls
- Create a Separate API Utility File
- Define API Base URL and Common Configuration
- Define API Endpoints
- Use API Calls in React Components
- Perform Error Handling and Response Interception
- Handle Caching and Optimizations

Steps to Handle Data Fetching and Responses

- Set Up State to Manage Data and Loading Status
- Use useEffect to Fetch Data When Component Loads
- Render Based on State (Loading, Error, or Data)
- Handle POST or Other Methods

Steps to perform Error Handling

- Choose an API Request Method (Fetch or Axios)
- Set Up State for Loading, Error, and Data
- Make the API Call with Error Handling
- Handle Errors Based on Status Code
- Show Loading, Error, and Data States in the UI

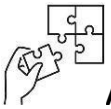
Steps to Handle Asynchronous and Concurrency

- Select the best asynchronous feature (promises) to handle operation
- In a ReactJS application, Create or Fetch Asynchronous Data (API Call
- Handle Async Operations in useEffect Hook
- Understand the Flow
- use. then () Instead of async/await
- Fetch Data on User Interaction

Steps to perform API Security and testing

- Use HTTPS for Secure Communication

- Authentication and Authorization
- Input Validation and Sanitization
- Rate Limiting and Throttling
- CORS (Cross-Origin Resource Sharing)
- CSRF Protection
- Use Security Headers
- Encryption of Sensitive Data
- Logging and Monitoring
- Security Testing
- Conduct regular penetration testing of your API to identify vulnerabilities.



Application of learning 1.6.

Rwanda Metrology Centre is a centre that provides real-time weather forecast. It has hired you as a software developer. The centre needs a software developer to work on integrating a weather API. You are requested to begin by planning the implementation, define the API's purpose and install dependencies like Axios. Organize their API calls by grouping related requests, implementing error handling, and managing asynchronous operations to optimize data fetching. Finally, ensure that the application is secure by testing for vulnerabilities and confirming that the API integration functions correctly, allow users to fetch real-time weather data while maintaining a seamless user experience.



Learning Outcome 1 End Assessment

Theoretical assessment

Question 1. Choose the correct alternative by circling the letter that corresponds to the answer

1. What is JSX in React?

   a) A JavaScript framework

   b) A JavaScript XML syntax extension

   c) A state management library

   d) A CSS library

2. Which of the following is a lifecycle method in React?

   a) render

   b) componentDidMount

   c) useEffect

   d) setState

3. Which hook is used to manage state in functional components?

   a) useState

   b) useEffect

   c) useContext

   d) useReducer

4. Which of the following is a valid way to pass props in React?

   a) Using state

   b) Through the constructor

   c) As an attribute in JSX

   d) Through the render method

5. What does the Virtual DOM do?

   a) Stores component states

   b) Provides routing in React

   c) Minimizes the number of DOM manipulations

d) Handles event listeners

6. What is React Router primarily used for?

a) Managing component state

b) Navigation between different views

c) Handling forms

d) Making API calls

7. Which method unmounts a React component?

a) componentWillUnmount

b) componentDidUpdate

c) render

d) shouldComponentUpdate

8. Which React hook would you use to run side effects?

a) useState

b) useEffect

c) useReducer

d) useMemo

9. What does Redux help manage in a React application?

a) UI components

b) Global state

c) Event handling

d) DOM updates

10. Which event type in React is responsible for handling input changes?

a) onClick

b) onChange

c) onSubmit

d) onMouseOver

11. What is a synthetic event in React?

a) A browser native event

b) A wrapper around browsernative events

c) A custom event created by the developer

d) A debugging tool

12. In React, which function is used to handle API calls?

a) useFetch

b) fetch

c) Axios

d) getRequest

13. The main purpose of the Context API in React is:

a) Data fetching

b) State management across components

c) Handling user inputs

d) Event handling

14. What method is used to prevent event bubbling in React?

a) stopBubbling

b) preventPropagation

c) stopPropagation

d) stopImmediatePropagation

15. How is a controlled component defined in React?

a) A component that controls event propagation

b) A component whose state is managed by itself

c) A component whose form data is handled by React state

d) A component that cannot rerender

16. What does the useEffect hook do?

a) Handles side effects in function components

b) Updates component state

c) Allows conditional rendering

d) Enables lifecycle methods in class components

17. What does the Redux store represent?

a) A place where React components are defined

b) The global state for a React application

c) A collection of React hooks

d) A virtual DOM replica

18. What is the main use of MobX in React applications?

a) Manage local state

b) Manage global state

c) API integrations

d) Routing

19. What is the purpose of componentDidUpdate in React?

a) To update the state

b) To perform actions after a component rerenders

c) To initialize a component

d) To unmount a component

20. How can you perform routing in React?

a) Using React hooks

b) Using Redux

c) Using React Router

d) Using event listeners

Question 2. From the questions below, identify the validity of the statement by indicating if the statement is TRUE or FALSE

1. JSX allows embedding HTML within JavaScript code

2. The useState hook is used for side effects in React components

3. The Virtual DOM helps improve the performance of React applications.

4. Props are mutable in React components.

5. The useEffect hook is equivalent to all three lifecycle methods in class components.

6. Redux is mainly used to manage local component state.

7. React Router helps in managing state transitions.

8. The Context API is a way to avoid passing props through multiple components

9. A synthetic event in React is a crossbrowser wrapper around native events

10. The componentWillUnmount method is used to clean up after a component is removed

11. React components rerender when their props or state changes

12. In JSX, class names are assigned using the class attribute

13. Lifecycle methods are only available in class components

14. The bind() method is used to bind event handlers in functional components

15. MobX is a state management tool similar to Redux

16. A controlled component uses internal state to manage form data

17. Context API and Redux both solve global state management problems

18. Nested routing is supported by React Router

19. Events in React follow the same event delegation as vanilla JavaScript

20. React Developer Tools is used to debug and inspect components in a browser

Question 3. From the questions below, complete the following sentences by inputting the best phrase or group of phrases from a list of phrases provided below.

navigation between different components or pages, JavaScript XML, state, JavaScript function that returns JSX, a component has been rendered to the DOM, side effects , hooks, class components, browser's native events, inmemory representation, global state, passing data through the component tree without props, parent components, child components, event.stopPropagation(), state management libraries, clean up resources before a component is removed from the DOM, class components, memorize callback functions, useParams, the useEffect hook or async/await

1. JSX stands for _____.

2. React Router is used for _____.

3. The useState hook is used to manage _____ in functional components.

4. The componentDidMount lifecycle method is called after _____.

5. A functional component in React is a _____.

6. React allows developers to manage state using both _____ and _____.

7. Synthetic events are a cross-browser wrapper around _____.

8. A Virtual DOM in React is an _____ of the real DOM.

9. The useEffect hook is primarily used for handling _____.

10. Redux helps in managing _____ in large React applications.

11. The React Context API is used for _____.

12. Props in React are passed from _____ to _____.

13. To prevent an event from bubbling up the DOM, you would use _____.

14. MobX and Redux are both _____.

15. The componentWillUnmount lifecycle method is used to _____.

16. The bind() method in React is used to bind event handlers to _____.

17. The useCallback hook in React is used to _____.

18. A controlled component's form data is handled by _____.

19. URL parameters in React Router are handled using the _____ method.

20. To handle asynchronous data fetching in React, you can use _____.

Practical assessment

ABC Company is an e-commerce company that is located in Rwamagana District- Eastern province – Rwanda. It has an online platform specializing in selling electronics. The platform is expanding its web application with a user-friendly React.js frontend to improve the shopping experience. They aim to enhance the performance, user interface, and functionality of the website, making it more dynamic, responsive, and interactive. You, as the React.js developer at ABC Company, have been assigned the task of building a Product Listing and Search Page that includes a search bar, filters, product listing, and product details. The page should provide a smooth user experience by incorporating various features using React.js

END



References

Banks, A., & Porcello, E. (2020). Learning React: A hands-on guide to building web applications using React and Redux (2nd ed.). O'Reilly Media.

Callahan, B. (2021). React Explained: Your step-by-step guide to React (1st ed.). CodeBrauer.

Cooper, C. (2021). React Design Patterns and Best Practices: Design, build, and deliver production-ready web applications using industry-standard practices (2nd ed.). Packt Publishing.

Grider, S. (2020). The complete React developer course (4th ed.). Udemy.

Haris, B. (2021). React Projects: Build advanced cross-platform projects with React and React Native to become a professional developer (1st ed.). Packt Publishing.

Hinnant, R. (2021). React by example: Building modern web applications using React (1st ed.). Manning Publications.

Lerner, R. (2022). Professional React: State management, performance optimization, and testing (1st ed.). Wiley.

Mackenzie, B., & Dodds, K. (2020). Epic React: Create interactive user interfaces with confidence (1st ed.). Kent C. Dodds Tech LLC.

Moore, S. (2020). React Cookbook: Recipes for mastering modern web development (2nd ed.). Packt Publishing.

Park, S., & Picinich, N. (2021). Fullstack React: The complete guide to React and friends (2nd ed.). Fullstack.io.

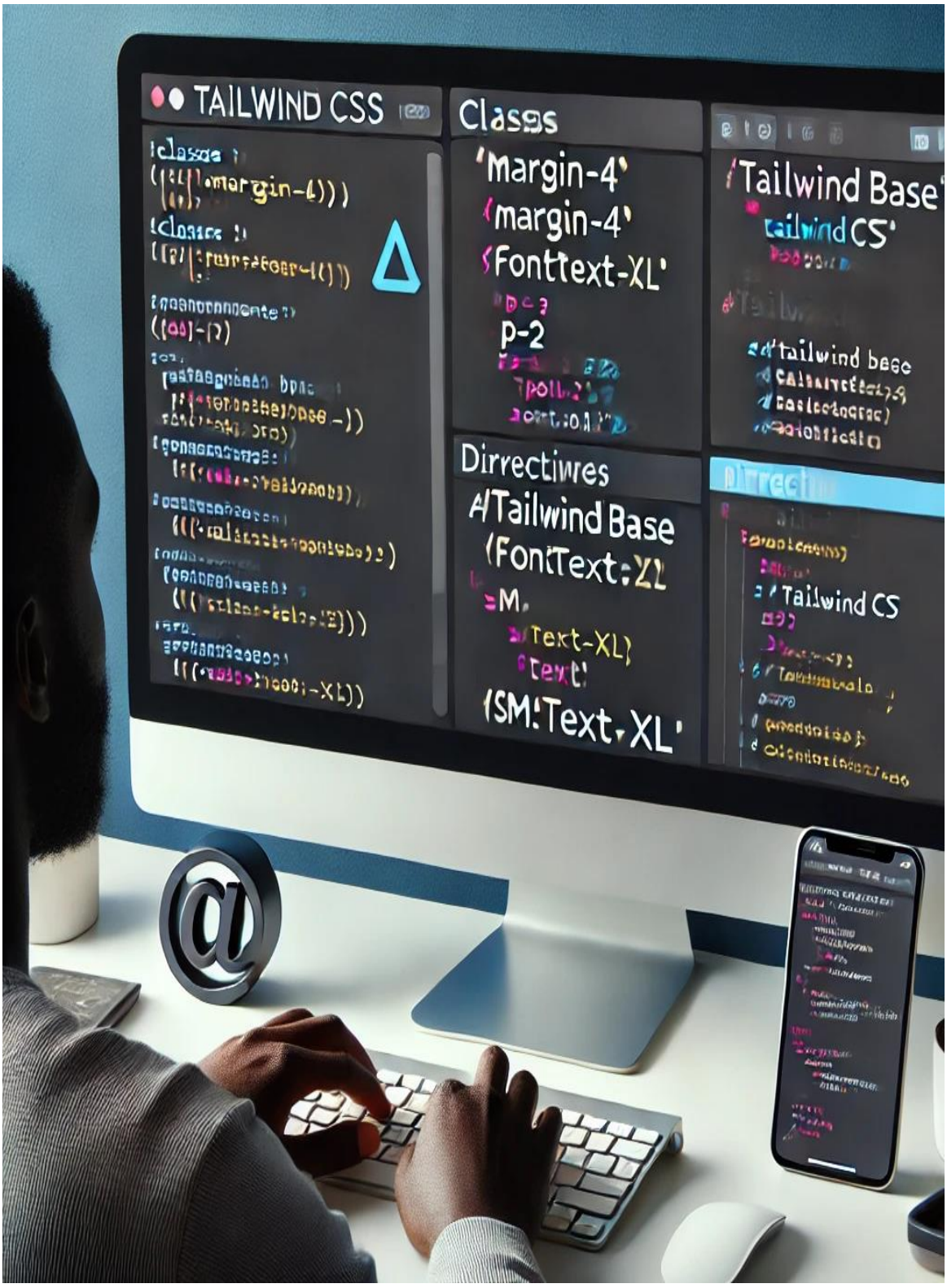
Sharma, A. (2020). React Hooks in Action: With Suspense, Concurrent mode, and more (1st ed.). Manning Publications.

Taube, C. (2021). React Router: Building Single-Page Applications with React (1st ed.). O'Reilly Media.

Wieruch, R. (2021). The Road to React: Your journey to master React.js in JavaScript (4th ed.). Leanpub.

Williams, A. (2021). React and React Native: A complete hands-on guide to modern web and mobile development (3rd ed.). Packt Publishing.

Zolotarev, A. (2020). React Quickly: Painless web apps with React, JSX, Redux, and GraphQL (2nd ed.). Manning Publications.


TAILWIND CSS
Classes
'margin-4'
'margin-4'
'Fonttext-XL'
p-2
Tailwind Base
Directiwres
Tailwind Base
Tailwind CS
Text-XL
SM:Text-XL

Indicative contents

2.1 Applying Tailwind Utility Classes
2.2 Applying Responsive Design Principles
2.3 Customization of Tailwind Styles

Key Competencies For Learning Outcome 2: Apply Tailwind CSS Framework

Knowledge	Skills	Attitudes
● Description of animation, transitions, flexbox, grid, states and classes. ● Description of media queries and breakpoints ● Description of typography and readability ● Identification of styles ● Description of responsive design principles	● Applying Tailwind CSS in React.JS ● Applying responsive design principles ● Customizing styles in Tailwind	● Being creative in developing up-to-date Tailwind CSS apps ● Being problem-solving oriented during development with Tailwind CSS ● Being updated on latest Tailwind CSS versions ● Being collaborative while developing of application with Tailwind CSS



Duration: 10 hrs

Learning outcome 2 objectives:



By the end of the learning outcome, the trainees will be able to:

1. Describe clearly animations and transitions classes used in Tailwind CSS.
2. Describe clearly flexbox and grid classes used in Tailwind CSS
3. Differentiate properly responsive design principles
4. Identify properly the styles used in Tailwind CSS.
5. Integrate correctly Tailwind CSS in React.JS application.
6. Customize properly Tailwind styles in React.JS application.



Resources

Equipment	**Tools**	**Materials**
● Computer	● VS Code ● Browser ● Terminal	● Internet ● Electricity



Advance Preparation:

Before delivering this learning outcome, you are recommended to:

- Have prepared computer lab with Internet connectivity
- Have visual studio code installed on all computers to be used.
- Have videos to be used as didactic material.
- Have sample React.JS project developed to be used for styling.

Indicative Content 2.1: Applying Tailwind Utility Classe

Duration: 4 hrs

Practical Activity 2.1.1: Integrating Tailwind CSS with React.JS

Task:

1: You are requested to go to the computer lab to integrate Tailwind CSS with ReactJS.
2: Read the key readings 2.1.1
3:Apply safety precautions.
4: Integrate Tailwind CSS with ReactJS.
5:Present your work to the trainer.
6: Perform the task provided in application of learning 2.1

Key readings 2.1.1: Integrating Tailwind CSS with React.JS

Step 1: Install Tailwind CSS

Install tailwindcss via npm by typing this command **npm install -D tailwindcss**

```
C:\Users\MGervais\Documents\reactapp3>npm install -D tailwindcss
```

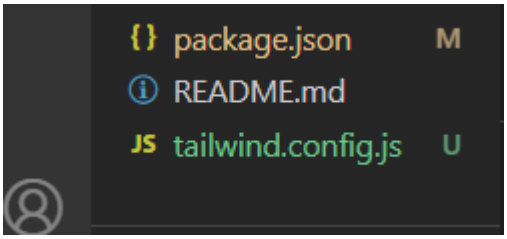
Step 2: Create Tailwind.config.js file

Create your **tailwind.config.js** file with this command **npx tailwindcss init** in prompt command.

```
C:\Users\MGervais\Documents\reactapp3>npx tailwindcss init

Created Tailwind CSS config file: tailwind.config.js

C:\Users\MGervais\Documents\reactapp3>
```



Step 3: Configure your template paths

Add the paths to all of your template files in your tailwind.config.js file.



Step 4: Include Tailwind in your CSS

In your React app, locate or create a src/index.css file, and include the Tailwind CSS directives to inject its base styles, components, and utilities.

Replace the contents of **src/index.css** with:

@tailwind base;

@tailwind components;

@tailwind utilities;

Step 5: Import the Tailwind CSS in your React application

You need to import the index.css file into your main **src/index.js** file to ensure Tailwind CSS is applied globally in your React app.

In src/index.js, add the following line: import './index.css';

Step 6:Finally, start your React application: npm start

Theoretical Activity 2.1.2: Description of utility-first fundamentals and states

Tasks:

1: You are requested to answer the following questions:

I. Define utility-first fundamentals.
II. Give at least 5 examples of utility-first classes that are used in Tailwind CSS.
III. What is the role of using the utility-first classes used in Tailwind CSS?
IV. Outline three (3) variant states that are used in Tailwind CSS.
V. What are the features of utility-first fundamentals in Tailwind CSS?
VI. Explain the use of:
    a) text-center
    b) p-4
    c) bg-blue-500

2: Write your findings on papers, flip chart or chalkboard.
3: Present the findings/answers to the whole class or trainer.
4: For more clarification, read the key readings 2.1.2.

Key readings 2.1.2: Description of utility-first fundamentals and states

- **Utility-First Fundamentals**

Utility-first fundamentals in Tailwind CSS is an approach where you build designs using pre-defined utility classes that are designed to be small, single-purpose, and composable. Instead of writing custom CSS for each component, you apply these utility classes directly in your HTML to style elements. This results in a faster development process and more maintainable code since you can focus on composing utilities rather than writing new styles for each element.

Utility classes are low-level, single-purpose classes that do one thing, such as setting a specific margin, padding, font size, color, etc.

Some features of utility-first fundamentals in Tailwind CSS are:

- ✓ **Single-purpose Classes**: Each utility class serves one specific function. For example:

  Padding (p-*): Controls padding for an element.
  - p-4: Adds padding of 1rem.
  - px-4: Adds padding of 1rem on the left and right (x-axis).

- py-4: Adds padding of 1rem on the top and bottom (y-axis).

Margin (m-*): Controls margin for an element.

- m-4: Adds margin of 1rem.
- mx-auto: Centers an element horizontally by setting left and right margins to auto.

Example: Content with margin and padding

Text Size:

- text-sm: Sets the font size to small.
- text-lg: Sets the font size to large.
- text-4xl: Sets the font size to extra-large (font-size: 2.25rem).

Font Weight:

- font-thin: Sets a thin font weight.
- font-bold: Sets a bold font weight.

Text Alignment:

- text-left, text-center, text-right: Aligns text accordingly.

Example: <h1 class="text-4xl font-bold text-center">Centered Heading</h1>

Text Color:

- text-gray-700: Sets text color to dark gray.
- text-blue-500: Sets text color to blue.

Example: This is gray text

Background Color (bg-*):

- bg-white: Sets the background color to white.
- bg-blue-500: Sets the background color to blue.

Example: This is a blue background

✓ **Composability classes**: They combine multiple utility classes to achieve complex designs. Instead of creating new CSS rules, you use a combination of classes.

- **Example:**

```
<div class="p-4 bg-blue-500 text-white rounded-lg">
This is a box with padding, background color, and rounded corners.
</div>
```

✓ **Responsiveness Built-in classes**: Tailwind makes it easy to create responsive designs by providing utilities for different screen sizes. You don't need to write custom media queries.

- **Example:**

```
<div class="p-4 sm:p-6 md:p-8 lg:p-10">
This element has different padding on small, medium, and large screens.
</div>
```

- **States in Tailwind CSS**

Tailwind CSS provides utilities for managing various states, like hover, focus, active, and more. These state utilities are applied by prefixing the utility class with the state modifier.

- ✓ **Hover State**: You can define styles that apply when an element is hovered using the hover: prefix.
  - **Example:**<button class="bg-gray-500 hover:bg-blue-500">Hover Me</button>
- ✓ **Focus State:** Styles can be applied when an element is focused with the focus: prefix.
  - **Example:** <input class="border-gray-300 focus:border-blue-500" />
- ✓ **Active State**: The active: prefix allows you to style elements when they are active (being clicked).
  - **Example:** <button class="bg-gray-500 active:bg-blue-500">Click Me</button>
- ✓ **Other States**:
  - disabled: for elements that are disabled.
  - group-hover: for applying hover styles to an element when its parent is hovered.
  - focus-within: for styling an element when any child element is focused.



Practical Activity 2.1.3: Applying Utility-First Fundamentals and states



Task:

1: You are requested to go to the computer lab to apply the utility-first fundamentals and states.
2: Read key readings 2.1.3
3: Apply safety precautions.
4: Apply utility-first fundamentals and states
5:Present your work to the trainer
6: Perform the task provided in application of learning 2.1



Key readings 2.1.3: Applying Utility-First Fundamentals and states

Step 1: Browse to the folder where your project is created

Open **File Explorer** (Windows) and navigate to the folder where your React project is located.

Step 2: Open the command prompt

In **Windows**, hold down Shift + Right-click inside the folder, then select "Open PowerShell window here" or "Open Command Prompt here."

Step 3: Type the command to open the project in your text editor

If you are using **VS Code** as your text editor, you can use the following command:**code .** This will open the current directory in VS Code.

Step 4: Access the React file you want to style

In VS Code, browse through the **src** folder and open the component file you want to style, typically something like **App.js, index.js**, or any component file.

Step 5: Use the utility-first fundamentals

If you have Tailwind CSS set up in your project, you can start applying Tailwind utility classes directly in your JSX code.

For example:

```
<div className="bg-blue-500 text-white p-6">
<h1 className="text-2xl font-bold">Hello World</h1>
</div>
```

Step 6: Apply the states

Tailwind makes it easy to style elements based on different states such as hover, focus, active, etc. For example:

```
<button className="bg-gray-500 hover:bg-blue-500 active:bg-blue-700 text-white font-bold py-2 px-4 rounded">
Click Me
</button>
```



Theoretical Activity 2.1.4: Description of animation and transitions



Tasks:

1: You are requested to answer the following questions

i. What do you understand by the term?
   a) Animation
   b) Transition

ii. Outline four utility classes that are used to animate object with Tailwind CSS.

iii. Enumerate the components of transitions in Tailwind CSS.

iv. Give the main possible classes of:
   a) Animation
   b) Transition delay
   c) Transition timing function
   d) Transition duration

e) Transition property

2: Write your findings on papers,flip chart or chalkboard

3: Present the findings/answers to the whole class or trainer

4: For more clarification, read the key readings 2.1.4.



Key readings 2.1.4: Description of animation and transitions

- **Animation**

Animation is the ability to apply motion effects to elements, either using pre-defined utility classes or custom animations. These effects can include transformations like spinning, pulsing, bouncing, or fading, which help create a more interactive and engaging user experience.

Animations in Tailwind are handled with utility classes that apply keyframe-based animations. Tailwind includes predefined animations like **bounce**, **spin**, and **ping**.

Common Animation Classes:

- ✓ **animate-none**: Disables animations.
- ✓ **animate-spin**: Rotates the element (spinning effect).
- ✓ **animate-ping**: Creates a pulsing effect, like a radar ping.
- ✓ **animate-pulse**: Causes an element to smoothly increase and decrease in opacity.
- ✓ **animate-bounce**: Bounces the element up and down.

Example: Spinning Loader

```
<div class="animate-spin h-8 w-8 border-4 border-blue-500 border-t-transparent rounded-full"></div>
```

Example: Bouncing Button

```
<button class="bg-green-500 text-white px-4 py-2 rounded-full animate-bounce">
  Bouncing Button
</button>
```

Here, the button will bounce up and down continuously because of the animate-bounce class.

- **Transitions**

Transition is the smooth change of CSS properties over a specified duration when an element's state changes, such as when it is hovered, focused, or clicked.

Transitions components in Tailwind CSS are: transition, transition –property, duration-*,ease-*,and delay-*.

Transition Property: utilities for controlling which CSS properties transition.

- ✓ **transition-none:**Disables all transitions for an element.
  Example: <button class="transition-none">No Transition</button>
- ✓ **transition-all**:Applies transitions to all properties of an element, such as background color, opacity, transform, shadow, etc.Useful when you want to apply transitions to multiple properties at once.
  Example: <button class="transition-all duration-300">Transition All</button>
- ✓ **Transition:** This is a shorthand class for transition-all. It enables a smooth transition for all properties but without specifying the details. It offers a quick way to apply transitions to all properties without specific customization.
  Example: <button class="transition">Default Transition</button>
- ✓ **transition-colors**:Applies transitions specifically to color-related properties, such as background-color, border-color, and text-color.
  Example: <button class="transition-colors duration-500 hover:bg-blue-500">Transition Colors</button>
- ✓ **transition-opacity**:Adds transitions to changes in opacity. Useful for fading effects where you want to control how an element appears or disappears.
  Example: Fade In
- ✓ **transition-shadow**: Adds transitions to changes in the box shadow. Useful for hover effects where you want to smoothly change the shadow of an element.
  Example: Shadow Transition
- ✓ **transition-transform**:Adds transitions to CSS transforms, such as translate, rotate, scale, or skew. Useful when you want smooth animations for transformations like rotating or scaling an element.

  Example: Scale on Hover

Use the **transition-*** utilities to specify which properties should transition when they change.

Example:

<button className="bg-blue-500 text-white font-bold py-2 px-4 rounded

transition-all duration-300 ease-in-out

hover:bg-blue-700 hover:scale-110">

Hover Me

</button>

Transition duration is the amount of time a transition takes to complete. It's used to control how quickly an element transitions from one state to another, such as when a hover effect changes the color, size, or position of an element. It is defined in milliseconds.

The common transition duration classes:

- duration-75: 75 milliseconds
- duration-100: 100 milliseconds
- duration-150: 150 milliseconds (default)
- duration-200: 200 milliseconds
- duration-300: 300 milliseconds
- duration-500: 500 milliseconds
- duration-700: 700 milliseconds
- duration-1000: 1000 milliseconds (1 second)

Use the **duration-*** utilities to control an element's transition-duration.

Transition timing function controls how intermediate states of a transition are calculated, defining the speed curve of the transition.

Tailwind provides several built-in timing function utilities, including:

- ease-linear: Transitions at a constant speed.
- ease-in: Starts slowly and speeds up.
- ease-out: Starts fast and slows down.
- ease-in-out: Starts slowly, speeds up, then slows down at the end.

Use the **ease-*** utilities to control an element's easing curve.

Example:

** **

<!-- Content that will transition -->

** **

Transition delay is the time before the transition effect starts when a state change occurs, like hover or focus. It controls how long the system waits before applying the transition effects like changing color, opacity, or transforming an element.

In Tailwind CSS, you can add transition delays using the delay-{time} utility. This defines how long (in milliseconds) a transition should wait before starting. For example, you can use delay-100, delay-200, etc., to apply delays of 100ms, 200ms, etc.

Example:

<button class="transition delay-150 duration-300 ease-in-out hover:bg-blue-500">

Hover me

</button>

Use the **delay-*** utilities to control an element's transition-delay.



Practical Activity 2.1.5: Applying animation and transitions



Task:

1: You are requested to go to the computer lab to apply the animation and transitions
2: Read the key readings 2.1.5
3:Apply safety precautions.
4:Apply the animation and transitions.
5: Present your work to the trainer
6: Perform the task provided in application of learning 2.1



Key readings 2.1.5: Utilizing animation and transitions

Step 1: Browse to the folder where your project is created

Open **File Explorer** or your terminal and navigate to the folder where your React project is stored.

Step 2: Open command prompt

- **Windows:** Press Windows + R, type cmd, and press Enter.
- **Mac/Linux:** Open the **Terminal** from your applications or use Ctrl + Alt + T.

Step 3: Type the command to open the project in your text editor

Assuming you're using **VS Code**, type the following command in the terminal:code .

This will open the current project folder in VS Code.

Step 3: Access the React component

In VS Code, browse the src folder and open any component file (like App.js or any other component file you wish to edit).

Step 4: Add animation

To add an animation, you can use **Tailwind CSS** classes (assuming Tailwind CSS is already installed in your project).

For example, let's add a bounce animation:

```
<div className="ml-5 animate-bounce"> I bounce! </div>
```

You can also customize animations like this:

```
<div className="animate-bounce duration-300 ease-in-out">
  Download
</div>
```

Step 5: Add transitions

Tailwind provides utilities for transitions. Here's an example of how to apply transitions to an element:

```
<button className="bg-blue-500 hover:bg-blue-700 text-white font-bold py-2 px-4
rounded transition duration-300 ease-in-out">
  Hover me
</button>
```

This button will smoothly transition to a darker blue when hovered.



Theoretical Activity 2.1.6: Description of flexbox and grid



Tasks:

1: You are requested to answer the following questions

i. What do you understand by the term?
   a) Flexbox
   b) Grid
ii. What are the possible classes of flex direction?
iii. What is flex-wrap-reverse?
iv. Differentiate the following:
   a) grow class from shrink class
   b) col-span from row-span
v. Give the main possible values of * :
   a) grid-flow-*
   b) auto-cols-*
   c) auto-rows-*

d) justify-*
e) justify-items-*
f) content-*
g) place-content-*

2: Write your findings on papers,flipchart or chalkboard.

3: Present the findings/answers to the whole class or trainer

4: For more clarification, read the key readings 2.1.6.



Key readings 2.1.6: Description of flexbox and grid

Tailwind CSS provides a powerful set of utility classes for implementing Flexbox and CSS Grid layouts, allowing for responsive and efficient design without writing custom CSS. Here's a breakdown of how both Flexbox and Grid can be utilized using Tailwind's utility classes:

- **Flexbox**

Flexbox is designed for one-dimensional layouts, where items are arranged in a row or column. Here are some key Tailwind utility classes for Flexbox:

✓ **Container setup**:

- flex: Makes the element a flex container, enabling the flex behavior for its children.
- inline-flex: Applies flex layout to the element but treats it as an inline element.

Example:

** ... **

✓ **Direction**

Controls the direction in which the flex items are laid out:

- flex-row: Lays out items in a horizontal row (default).
- flex-row-reverse: Reverses the order of items in a row.
- flex-col: Lays out items in a vertical column.
- flex-col-reverse: Reverses the order of items in a column.

**Example: ... **

✓ **Alignment**:

Controls how flex items are aligned within the container:

For Main Axis (Horizontal) Alignment:

- justify-start: Aligns items to the start of the container.
- justify-center: Centers items horizontally.

- justify-end: Aligns items to the end of the container.
- justify-between: Distributes items evenly with space between them.
- justify-around: Distributes items with equal space around them.
- justify-evenly: Distributes items with equal space between and around them.

  Example: ...

For Cross Axis (Vertical) Alignment:

- items-start: Aligns items to the top (start) of the container.
- items-center: Centers items vertically.
- items-end: Aligns items to the bottom (end) of the container.
- items-stretch: Stretches items to fill the container height.
- items-baseline: Aligns items based on their text baselines.

  Example: ...

✓ **Flex wrapping**

Controls how flex items wrap within the container:

- flex-wrap: Allows items to wrap onto multiple lines.
- flex-wrap-reverse: Wraps items in reverse order.
- flex-nowrap: Prevents items from wrapping (default).

  Example: ...

✓ **Flex Item Properties**

Controls how individual flex items grow, shrink, or maintain their sizes:

- flex-grow: Makes the flex item grow to fill available space.
- flex-shrink: Shrinks the item if necessary.
- flex-none: Prevents the item from growing or shrinking.
- flex-auto: Allows the item to grow and shrink as needed.

  Example:

```
<div className="flex">
 <div className="flex-grow">Level 3</div>
 <div className="flex-shrink">Level 4</div>
</div>
```

✓ **Gap**

Adds spacing between flex items:

- gap-x-{size}: Adds horizontal spacing between items.
- gap-y-{size}: Adds vertical spacing between items.
- gap-{size}: Adds both horizontal and vertical spacing.

  Example: ...

Example of full code:

```
function FlexExample() {
  return (
    <div className="flex flex-col items-center justify-center min-h-screen">
      <div className="flex gap-4">
        <div className="flex-grow p-4 bg-blue-500 text-white">Item 1</div>
        <div className="flex-shrink p-4 bg-green-500 text-white">Item 2</div>
        <div className="p-4 bg-red-500 text-white">Item 3</div>
      </div>
    </div>
  );
}
export default FlexExample;
```

Example:

```
<div class="flex flex-row gap-4">
    <div class="basis-1/4 bg-red-500 rounded">Level 3</div>
    <div class="basis-1/4 bg-green-500 rounded">Level 4</div>
    <div class="basis-1/2 bg-blue-500 rounded">Level 5</div>
  </div>
```

- **Grid**

Grid is more suited for two-dimensional layouts, allowing for both rows and columns. Tailwind provides a set of utility classes for Grid as well:

- ✓ **Basic Grid Structure**

Tailwind provides utility classes to define grids and control the number of columns and rows. To use the grid system, simply apply the grid class to a container element.

```
<div className="grid">
  {/* Grid Items */}
</div>
```

- ✓ **Gap (Spacing)**

To add spacing between grid items, you can use the gap-* utility. For example, gap-4 will apply a uniform gap between all items.

```
<div className="grid grid-cols-2 gap-4">
  <div>Item 1</div>
  <div>Item 2</div>
</div>
```

- ✓ **Defining Columns**

You can control the number of columns in the grid using the grid-cols-* class. For example, grid-cols-3 will create three equal-width columns.

For example:

```
<div className="grid grid-cols-3 gap-4">
  <div>Item 1</div>
  <div>Item 2</div>
  <div>Item 3</div>
</div>
```

This creates a grid with 3 columns, and the gap-4 adds spacing between the grid items.

- ✓ **Defining Rows**

You can define rows using grid-rows-* classes. This sets the number of rows in the grid.

```
<div className="grid grid-rows-2 gap-4">
  <div>Item 1</div>
  <div>Item 2</div>
  <div>Item 3</div>
</div>
```

This creates a grid with 2 rows. If the content exceeds the defined rows, it will flow into new rows automatically.

- ✓ **Span Columns and Rows**:

You can make grid items span across multiple columns or rows using col-span-* or row-span-*.

```
<div className="grid grid-cols-3 gap-4">
  <div className="col-span-2">Item 1 (Spans 2 Columns)</div>
  <div>Item 2</div>
  <div>Item 3</div>
</div>
```

- ✓ **Auto Rows and Columns**

You can use auto-rows-{size} and auto-cols-{size} to control the size of automatically created rows and columns.

```
<div className="grid grid-flow-row auto-rows-min gap-4">
  <div>Item 1</div>
  <div>Item 2</div>
</div>
```

This ensures that each row takes the minimum space required by its content.

- ✓ **Alignment**

Tailwind provides alignment utilities such as justify-items-*, items-center, and place-items-* to control how grid items are aligned within their grid area.

```
<div className="grid grid-cols-3 items-center">
  <div>Item 1</div>
  <div>Item 2</div>
  <div>Item 3</div>
</div>
```

Example:

```
const GridExample = () => {
  return (
    <div className="grid grid-cols-2 gap-4 sm:grid-cols-3 lg:grid-cols-4">
      <div className="p-4 bg-blue-500">Item 1</div>
      <div className="p-4 bg-blue-500">Item 2</div>
      <div className="p-4 bg-blue-500">Item 3</div>
      <div className="p-4 bg-blue-500">Item 4</div>
    </div>
  );
};

export default GridExample;
```



Practical Activity 2.1.7: Utilizing flexbox and grid



Task:

1: You are requested to go to the computer lab to utilize the flexbox and grid.
2:Read the key readings 2.1.7
3: Apply safety precautions.
4: Utilize flexbox and grid
5:Present your work to the trainer
6: Perform the task provided in application of learning 2.1



Key readings 2.1.7: Utilizing flexbox and grid

Step 1: Browse to the folder where your project is created

Open File Explorer (Windows) or Finder (Mac) and navigate to the directory where your React project is located.

Step 2: Open Command Prompt

In Windows, hold Shift and right-click inside the folder, then select "Open PowerShell window here" or "Open Command Prompt here."

In Mac/Linux, open Terminal, type cd, drag the folder into the terminal window, and press Enter.

Step 3: Type the command to open the project in your text editor

If you are using VS Code, type:code .
This will open your project in VS Code.

Step 4: Access the React Component

In VS Code, go to the src folder and open the React component you want to work on, typically App.js or another component file like Header.js or Main.js.

Step 5: Create Flexbox Layout

Tailwind CSS provides utility classes for Flexbox. Here's how to apply them to your component:

```
function FlexComponent() {
  return (
    <div className="flex items-center justify-center h-screen">
      <div className="bg-blue-500 text-white font-bold p-4 m-2">Item 1</div>
      <div className="bg-green-500 text-white font-bold p-4 m-2">Item 2</div>
      <div className="bg-red-500 text-white font-bold p-4 m-2">Item 3</div>
    </div>
  );
}
export default FlexComponent;
```

Explanation:

flex: Enables Flexbox layout.
items-center: Vertically centers items.
justify-center: Horizontally centers items.
h-screen: Sets the height to full screen.
Each div inside the flex container is a flex item.

Step 6: Create the Grid Layout

Similarly, you can create a grid layout with Tailwind's grid utilities:

```
function GridComponent() {
  return (
    <div className="grid grid-cols-3 gap-4 p-4">
      <div className="bg-blue-500 text-white font-bold p-4">Grid Item 1</div>
      <div className="bg-green-500 text-white font-bold p-4">Grid Item 2</div>
      <div className="bg-red-500 text-white font-bold p-4">Grid Item 3</div>
      <div className="bg-yellow-500 text-white font-bold p-4">Grid Item 4</div>
      <div className="bg-purple-500 text-white font-bold p-4">Grid Item 5</div>
      <div className="bg-pink-500 text-white font-bold p-4">Grid Item 6</div>
    </div>
  );
}
export default GridComponent;
```

Explanation:

grid: Enables grid layout.
grid-cols-3: Creates a grid with 3 columns.
gap-4: Adds spacing between grid items.
Full Example:
If you want both Flexbox and Grid examples in the same file, you can do this:

```
function LayoutExamples() {
  return (
    <div>
      <h1 className="text-3xl font-bold mb-4">Flexbox Example</h1>
      <div className="flex items-center justify-center h-40 mb-10">
        <div className="bg-blue-500 text-white font-bold p-4 m-2">Item 1</div>
        <div className="bg-green-500 text-white font-bold p-4 m-2">Item 2</div>
        <div className="bg-red-500 text-white font-bold p-4 m-2">Item 3</div>
      </div>
      <h1 className="text-3xl font-bold mb-4">Grid Example</h1>
      <div className="grid grid-cols-3 gap-4 p-4">
        <div className="bg-blue-500 text-white font-bold p-4">Grid Item 1</div>
        <div className="bg-green-500 text-white font-bold p-4">Grid Item 2</div>
        <div className="bg-red-500 text-white font-bold p-4">Grid Item 3</div>
        <div className="bg-yellow-500 text-white font-bold p-4">Grid Item 4</div>
        <div className="bg-purple-500 text-white font-bold p-4">Grid Item 5</div>
        <div className="bg-pink-500 text-white font-bold p-4">Grid Item 6</div>
      </div>
    </div>
  );
}
export default LayoutExamples;
```



Practical Activity 2.1.8: Applying reusable and custom styles



Task:

1: You are requested to go to the computer lab to apply the reusable and custom styles.
2:Read the key readings 2.1.8
3: Apply safety precautions.
4: Apply the reusable and custom styles
5:Present your work to the trainer
6: Perform the task provided in application of learning 2.1



Key readings 2.1.8: Applying reusable and custom styles

Step 1: Browse to the folder where your project is created

Open **File Explorer** (Windows) or **Finder** (Mac) and navigate to the directory where your React project is located.

Step 2: Open command prompt

In **Windows**, hold Shift and right-click inside the folder, then select "Open PowerShell window here" or "Open Command Prompt here."

In **Mac/Linux**, open **Terminal**, type cd, drag the folder into the terminal window, and press Enter.

Step 3: Type the command to open the project in your text editor

If you are using **VS Code**, type the following command in the terminal:**code .**

This will open the project in VS Code.

Step 4: Access the React component

In VS Code, open the src folder, and navigate to the component you want to style. This could be App.js, Header.js, Footer.js, etc.

Step 5: Apply the Reusable styles

Tailwind CSS encourages reusable utility-first styles. You can apply these directly in your JSX files by using Tailwind utility classes.

Example of Reusable Styles in a component:

```
function ReusableStylesComponent() {
 return (
  <div className="p-4 max-w-md mx-auto bg-gray-200 rounded-lg shadow-md">
   <h1 className="text-xl font-bold text-center mb-4">Reusable Styles Example</h1>
   <p className="text-gray-700 text-base">
    This is an example of reusable styles applied using Tailwind CSS utility classes.
   </p>
  </div>
 );
```

```
}

export default ReusableStylesComponent;
```

Step 6: Apply the custom styles

Sometimes you may want to apply custom styles that Tailwind doesn't provide directly. You can extend Tailwind by modifying the tailwind.config.js file.

Open tailwind.config.js and extend the theme

```
module.exports = {
  theme: {
    extend: {
      colors: {
        primary: '#1E40AF', // Custom blue color
        secondary: '#F97316', // Custom orange color
      },
      spacing: {
        '72': '18rem',  // Custom spacing
      },
      fontFamily: {
        custom: ['Gabriola', 'serif'], // Custom font family
      },
    },
  },
};
```

Apply the Custom Styles in the Component

```
function CustomStylesComponent() {
  return (
    <div className="p-72 bg-primary text-white font-custom">
      <h1 className="text-2xl font-bold mb-4">Custom Styles Example</h1>
```

```
    <p className="text-secondary">
      This is an example of custom styles applied using Tailwind CSS extended configuration.
    </p>
   </div>
  );
}
export default CustomStylesComponent;
```

Explanation of custom styles:

- bg-primary: Applies the custom primary color (#1E40AF).
- text-secondary: Applies the custom secondary color (#F97316).
- p-72: Custom padding of 18rem (288px).
- font-custom: Uses the custom font family (Gabriola, serif).



Practical Activity 2.1.9: Using functions and directives



Task:

1: You are requested to go to the computer lab to use the functions and directives.

2:Read the key readings 2.1.9

3: Apply safety precautions.

4: Use the functions and directives

5:Present your work to the trainer

6: Perform the task provided in application of learning 2.1



Key readings 2.1.9: Using functions and directives

Step 1: Navigate to your project folder

Open the command prompt (or terminal).Use the cd (change directory) command to navigate to the folder where your project is created. For example:

cd path\to\your\project

Step 2: Open your project in a text editor

If you are using Visual Studio Code, you can open your project by typing:code .

Step 3: Access the tailwind.config.js in your React application

Make sure your tailwind.config.js is properly set up with your custom breakpoints and theme values:

```
module.exports = {
  theme: {
    screens: {
      sm: '640px',
      md: '768px',
      lg: '1024px',
      xl: '1280px',
    },
    extend: {
      spacing: {
        '4': '1rem',
        '6': '1.5rem',
        '8': '2rem',
      },
      colors: {
        blue: {
          500: '#3b82f6',
        },
      },
    },
  },
};
```

Step 4:Use Functions in Tailwind CSS

The **theme()** function allows you to access values defined in your Tailwind CSS configuration, such as colors, spacing, font sizes, and more.

Examples of using theme() function:

```
/* In your CSS file */
.example-class {
  padding: theme('spacing.4'); /* This will apply a padding of 1rem (16px) */
  color: theme('colors.blue.500'); /* This will apply a blue color */
```

```
}
```
In this example, .example-class will have padding of 1rem and a text color of blue-500.
```
/* Inside your Tailwind CSS file */
.card {
  padding: theme('spacing.6'); /* Apply padding */
  background-color: theme('colors.white'); /* Set background color */
}
@media (min-width: theme('screens.lg')) {
  .card {
    padding: theme('spacing.8'); /* Increase padding on large screens */
  }
}
```

The screen() function is used to apply media queries based on the breakpoints defined in your Tailwind configuration. It helps you define responsive behavior more easily by referencing Tailwind's predefined screen sizes.

Example of using screen() function:
```
/* Applying styles only for screens larger than `md` (medium) */
@media screen(`md`) {
  .example {
    font-size: 1.5rem;
  }
}
```
The screen() function converts the md (or any other breakpoint) into a corresponding media query based on your Tailwind configuration.

Step 5: Use Directives in Tailwind CSS

Directives in Tailwind allow you to include different layers of your styles:

- The main directives you'll use are @tailwind, @apply, and @layer. For example:

```
@tailwind base;      /* Base styles */
@tailwind components; /* Component styles */
@tailwind utilities;  /* Utility styles */
/* Custom component styles */
@layer components {
   .btn-custom {
      @apply bg-green-500 text-white rounded;
   }
}
```



Points to Remember

- Steps to integrate Tailwind CSS in ReactJS application:
  - ✓ Install Tailwind CSS via npm.
  - ✓ Initialize Tailwind config files using npx tailwindcss init .
  - ✓ Configure tailwind.config.js to scan your React components.
  - ✓ Add Tailwind directives (@tailwind base; @tailwind components; @tailwind utilities;) to your CSS file.
  - ✓ Import the CSS into src/index.js.
  - ✓ Run your React app, and start using Tailwind classes in your components with **npm start**
- **Utility-first fundamentals** in Tailwind CSS is an approach where you build designs using pre-defined utility classes that are designed to be small, single-purpose, and composable.
- **Single-purpose Classes:** Each utility class serves one specific function.
- **Composability classes**: The combine multiple utility classes to achieve complex designs.
- **Responsiveness Built-in classes**: Tailwind makes it easy to create responsive designs by providing utilities for different screen sizes.
- **Hover state** allows you to apply styles to an element when a user hovers over it with their cursor. Tailwind provides a hover: prefix.
- **Focus state** is the styles applied to an element when it receives focus with focus: prefix.
- **Active state** is the styles applied to an element when it is being interacted with, typically by a user clicking or pressing it with active: prefix.
- To apply the utility first fundamentals and states you set the followings:
  - ✓ Browse to the folder where your project is created
  - ✓ Open command prompt
  - ✓ Type the command to open the project in your text editor
  - ✓ Access the React the file you want to style
  - ✓ Use the utility-first fundamentals
  - ✓ Apply the states

Animation is the ability to apply motion effects to elements, either using pre-defined utility classes or custom animations.

- **Animation classes**: animate-spin, animate-ping, animate-pulse, and animate-bounce.
- **Transition utility**: This class enables transitions on an element. By default, it applies a transition to all properties that change.

- **Transition-property**: You can specify which properties should transition using utilities like transition-colors, transition-opacity, transition-transform, and more.
- **Transition duration:** This utility controls how long the transition lasts. It is defined in milliseconds, and Tailwind provides utilities such as duration-150, duration-300, duration-500, etc.
- **Transition time** function controls the transition's timing function, which dictates how the speed of the transition progresses over time. Tailwind provides utilities like ease-linear, ease-in, ease-out, and ease-in-out.
- **Transition delay** adds a delay before the transition starts.
- To apply the reusable and custom styles, you do the followings:

✓ **Open your project in VS Code** using the code . command.
✓ **Access your React component** in the src folder.
✓ **Apply reusable styles** by using Tailwind CSS utility classes in your JSX components.
✓ **Create custom styles** by extending Tailwind's theme in tailwind.config.js and apply them in your component.

- **Animation** is the ability to apply motion effects to elements, either using pre-defined utility classes or custom animations.
- **Animation classes**: animate-spin, animate-ping, animate-pulse, and animate-bounce.
- **Transition utility**: This class enables transitions on an element. By default, it applies a transition to all properties that change.
- **Transition-property**: You can specify which properties should transition using utilities like transition-colors, transition-opacity, transition-transform, and more.
- **Transition duration:** This utility controls how long the transition lasts. It is defined in milliseconds, and Tailwind provides utilities such as duration-150, duration-300, duration-500, etc.
- **Transition time** function controls the transition's timing function, which dictates how the speed of the transition progresses over time. Tailwind provides utilities like ease-linear, ease-in, ease-out, and ease-in-out.
- **Transition delay** adds a delay before the transition starts.

To apply animation and transitions you set the followings:

✓ Browse to the folder where your project is created
✓ Open command prompt
✓ Type the command to open the project in your text editor
✓ Access the react component
✓ Add animation
✓ Add transitions

- **Flexbox** simplifies the process of creating flexible and responsive layouts.

- **Flex Container**: A container element that uses Flexbox to lay out its children. You can create a flex container using the **flex** class.
- **Flex Direction**: Controls the direction in which flex items are placed within a flex container. Use **flex-row** for horizontal layout (default) and **flex-col** for vertical layout.
- Flex Wrap: Defines whether flex items should wrap onto multiple lines or stay on a single line. Use **flex-wrap** to allow wrapping and **flex-nowrap** to prevent it.
- **Justify Content**: Aligns flex items along the main axis of the container. Use classes like **justify-start, justify-center, justify-end, justify-between, justify-around, and justify-evenly.**
- **Align Items:** Aligns flex items along the cross axis (perpendicular to the main axis). Use classes like **items-start, items-center, items-end, items-baseline,** and **items-stretch**.
- **Align Self**: Allows individual flex items to override the align-items setting of their flex container. Use classes like **self-auto, self-start, self-center, self-end,** and **self-stretch**.
- **Flex Grow and Shrink:** Controls how flex items grow and shrink to fit the container. Use classes like **flex-grow** to allow an item to grow and **flex-shrink** to allow it to shrink.
- **Flex Basis:** Defines the initial size of a flex item before it grows or shrinks. Use classes like **basis-1/4**, **basis-1/2**, and **basis-full**.
- **Grid** is a layout system that allows you to create complex, responsive, and flexible grid-based designs using a set of utility classes.
- **Grid**: Applies CSS Grid display to an element, turning it into a grid container.
- **Grid-cols-***: Defines the number of columns in the grid.
- **Grid-rows-*:** Defines the number of rows in the grid.
- **Col-span-*:** Defines how many columns an item should span.
- **Row-span-*:** Defines how many rows an item should span.
- To utilize the flexbox and grid you do the followings:

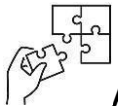
✓ Browse to the folder where your project is created
✓ Open command prompt
✓ Type the command to open the project in your text editor
✓ Access the react component
✓ Create flexbox
✓ Create the grid

- To apply the reusable and custom styles, you do the followings:

✓ Browse to the folder where your project is created
✓ Open command prompt
✓ Type the command to open the project in your text editor
✓ Access the react component

- ✓ Apply the reusable styles
- ✓ Apply the custom styles

- To use functions and directives, you do the followings:
- ✓ Navigate to your project folder
- ✓ Open your project in a text editor
- ✓ Use functions in Tailwind CSS
- ✓ Use directives in Tailwind CSS



Application of learning 2.1.

ABC Company is a software development company located in Nyanza district, Southern Province – Rwanda. Its customer has requested it to develop a React.JS application that will be faster for the users. Assume you have been hired as its front-end developer, you are requested to build a responsive dashboard application using ReactJS and Tailwind CSS as part of the web application required**.** The application will feature a header, a sidebar, and a main content area displaying data in both grid and list formats. You are required to implement interactivity with hover and focus states, use animations, transitions, custom styles, functions, directives, and customize Tailwind with utility classes to improve the attractiveness of the dashboard.

Indicative Content 2.2: Applying Responsive Design Principles

Duration: 3 hrs

Theoretical Activity 2.2.1: Description of responsive design principles

Tasks:

1: You are requested to answer the following questions

i. Explain the following principles:
   a) Mobile-first approach
   b) Flexible grid layouts
   c) Responsive images and media
   d) Media queries and breakpoints
   e) Typography and readability
   f) Interactive elements
   g) Testing and iteration
ii. What are the five default breakpoints inTailwind CSS?

2:Write your findings on papers,flipchart or chalkboard.

3: Present the findings/answers to the whole class or trainer.

4: For more clarification, read the key readings 2.2.1.

Key readings 2.1.2: Description of responsive design principles

- **Mobile-First Approach**

Mobile-First Approach is a strategy where you design your website or application starting with the smallest screen sizes (typically mobile devices) and progressively enhance the layout as the screen size increases. It ensures that the most essential content is prioritized for mobile users while adding more complex features for larger devices. Tailwind CSS makes it easy to implement this approach with its built-in responsive utility classes.

In Tailwind CSS, by default, styles apply to all screen sizes unless otherwise specified. This aligns well with the mobile-first approach since the base styles target mobile devices first.

Example:** Mobile-first text size and padding **

- **Flexible Grid Layouts**

Flexible Grid Layouts ensure that the layout adapts to different screen sizes and resolutions. Tailwind CSS offers a simple and powerful way to implement flexible grid layouts using its utility-first approach.

✓ **Grid Setup with grid-cols Utilities**

Tailwind provides grid utilities such as grid-cols-{n} to define the number of columns. You can easily set different column numbers for various screen sizes.

Example:

```
<div class="grid grid-cols-1 md:grid-cols-2 lg:grid-cols-3 gap-4">
  <div class="bg-blue-500">Item 1</div>
  <div class="bg-blue-500">Item 2</div>
  <div class="bg-blue-500">Item 3</div>
  <div class="bg-blue-500">Item 4</div>
</div>
```

✓ **Responsive Column Spans with col-span**

You can also define how many columns a grid item should span across on different screen sizes.

Example:

```
<div class="grid grid-cols-4 gap-4">
  <div class="col-span-4 md:col-span-2 bg-green-500">Item 1</div>
  <div class="col-span-4 md:col-span-2 bg-green-500">Item 2</div>
  <div class="col-span-2 bg-green-500">Item 3</div>
  <div class="col-span-2 bg-green-500">Item 4</div>
</div>
```

✓ **Flexible Grid Template Columns**

Tailwind's grid-cols-auto and grid-cols-{fraction} utilities allow auto-sized columns and fraction-based layouts.

Example:

```
<div class="grid grid-cols-1 sm:grid-cols-2 lg:grid-cols-3 xl:grid-cols-4 gap-6">
  <div class="bg-red-300">Auto 1</div>
  <div class="bg-red-300">Auto 2</div>
  <div class="bg-red-300">Auto 3</div>
  <div class="bg-red-300">Auto 4</div>
</div>
```

- ✓ **Auto-fill and Auto-fit Grids**

Using grid-cols-[auto-fit] or grid-cols-[auto-fill] can help in creating layouts where the grid automatically adjusts the number of items based on available space.

Example:

```
<div class="grid grid-cols-[repeat(auto-fit,_minmax(200px,_1fr))] gap-4">
  <div class="bg-yellow-300">Flexible 1</div>
  <div class="bg-yellow-300">Flexible 2</div>
  <div class="bg-yellow-300">Flexible 3</div>
  <div class="bg-yellow-300">Flexible 4</div>
</div>
```

- ✓ **Row Handling with grid-rows**

You can also control rows similarly to columns.

Example:

```
<div class="grid grid-cols-3 grid-rows-2 gap-4">
  <div class="row-span-2 bg-purple-400">Item 1</div>
  <div class="bg-purple-400">Item 2</div>
  <div class="bg-purple-400">Item 3</div>
  <div class="bg-purple-400">Item 4</div>
</div>
```

To achieve **Flexible Grid Layouts** in responsive designs with Tailwind CSS:

- Use grid-cols to define the number of columns.

- Combine with responsive utilities (e.g., md:grid-cols-2) for adaptability.
- Use gap to create spacing between grid items.
- Use auto-fit and auto-fill for layouts that adjust based on content and available space.

- **Responsive Images and Media**

In responsive design, ensuring that images and media adapt to different screen sizes is crucial. With Tailwind CSS, responsive images and media can be easily implemented by utilizing utility classes. Below are key principles and techniques to manage responsive images and media using Tailwind CSS.

✓ **Fluid Images**

To make images responsive, they need to scale according to the width of their container.

- **Use w-full**: This class makes the image scale to the full width of its container while maintaining its aspect ratio.

Example:

```
<img src="image.jpg" alt="Responsive image" class="w-full">
```

✓ **Object Fit for Images**

You can control how the image is resized and cropped using the object-fit utilities in Tailwind.

- **object-cover**: Ensures the image covers its container entirely, cropping any excess parts if needed.

Example:

```
<img src="image.jpg" alt="Cover image" class="w-full h-64 object-cover">
```

- **object-contain**: Ensures the entire image fits within its container, maintaining the aspect ratio but potentially leaving empty spaces.

Example:

```
<img src="image.jpg" alt="Contain image" class="w-full h-64 object-contain">
```

✓ **Responsive videos**

You can embed videos responsively using Tailwind CSS by applying the aspect-ratio utility to maintain aspect ratios.

Example:

```
<div class="w-full aspect-w-16 aspect-h-9">
 <iframe src="https://www.youtube.com/embed/videoid" frameborder="0" allowfullscreen class="w-full h-full"></iframe>
</div>
```

- ✓ **Lazy loading images**

For performance optimization, use the loading="lazy" attribute to defer the loading of images until they are about to enter the viewport.

Example:

```
<img src="image.jpg" alt="Lazy loaded image" class="w-full" loading="lazy">
```

- **Media Queries and Breakpoints**

Tailwind CSS comes with five default breakpoints that correspond to common device sizes:

Breakpoint	**Minimum width**	**CSS media query**	**used**
sm	640px	@media(min-width:640px){…}	Typically used for mobile devices
md	768px	@media(min-width:768px){…}	Tablets
lg	1024px	@media(min-width:1024px){…}	Laptops
xl	1280px	@media(min-width:1280px){…}	Desktops
2xl	1536px	@media(min-width:1536px){…}	Large desktops

You can apply these breakpoints using prefixes like sm:, md:, etc., before any utility class. For example:

```
<div class="text-base sm:text-lg md:text-xl lg:text-2xl xl:text-3xl">Responsive text
</div>
```

In this example:

- On screens smaller than 640px, the text size will be base.
- On screens 640px and larger (sm), the text size will be lg.
- On screens 768px and larger (md), the text size will be xl, and so on.

Tailwind automatically translates breakpoints into CSS media queries behind the scenes.

- **Typography and Readability**

Typography and readability are crucial components of responsive design, ensuring text remains legible and aesthetically pleasing across various devices and screen sizes. When using Tailwind CSS, you can control typography and readability using several built-in classes.

You can achieve good typography and readability in responsive designs with Tailwind CSS by using the following key aspects:

✓ **Font Size and Responsiveness**

Tailwind offers responsive font size classes that allow you to adjust text size based on screen size. For instance, you can use small font sizes on mobile devices and larger fonts on larger screens.

```
<p class="text-base md:text-lg lg:text-xl">
  This is a responsive text that adjusts based on the screen size.
</p>
```

✓ **Line Height (Leading)**

Adjusting line height enhances readability by preventing text from looking too cramped or too spread out.

```
<p class="leading-normal md:leading-relaxed lg:leading-loose">
  This text has responsive line height, improving readability across devices.
</p>
```

- leading-normal: Default line height.
- md:leading-relaxed: More relaxed line height on medium screens.
- lg:leading-loose: Loose line height for large screen

✓ **Font weight**

You can use font weight classes to ensure text is bold or thin enough to be readable on different devices without overwhelming the reader.

Example:

```
<p class="font-light md:font-medium lg:font-bold">
  This text changes its font weight based on screen size.
```

</p>

- ✓ **Responsive headings**

Use responsive text sizing for headings to ensure they scale appropriately on different devices, keeping a visual hierarchy.

Example:

```
<h1 class="text-2xl md:text-4xl lg:text-5xl font-bold">
  Responsive Heading Example
</h1>
```

- ✓ **Adjusting text alignment**

Text alignment can enhance readability depending on screen size. For example, centering text on mobile but left-aligning it on larger screens.

```
<p class="text-center md:text-left">
  This text is centered on mobile but left-aligned on larger screens.
</p>
```

- ✓ **Text color and contrast**

Ensure sufficient contrast between the text and background for readability, especially across various screen sizes or color modes (light/dark mode). Tailwind offers text color utilities.

Example:

```
<p class="text-gray-800 dark:text-white">
  This text adapts to dark mode for better readability.
</p>
```

- ✓ **Readability with padding and margins**

Adding sufficient spacing between text blocks improves readability by preventing the text from appearing cramped.

Example:

```
<p class="mb-4 md:mb-6 lg:mb-8">
  This text has more bottom margin on larger screens, improving spacing and readability.
</p>
```

- ✓ **Responsive letter spacing**

Adjusting letter spacing can help text be more legible, particularly on larger screens.

Example:

```
<p class="tracking-normal md:tracking-wide lg:tracking-wider">
  This text adjusts its letter spacing based on screen size.
</p>
```

- ✓ **Handling long text (Truncation & Wrapping)**

Tailwind allows you to manage long text, ensuring it wraps or truncates appropriately on smaller screens.

Example:

```
<p class="truncate md:whitespace-normal">
  This text will truncate on small screens but display normally on medium and larger screens.
</p>
```

Combining Classes for Readability

Here's an example that combines several typography classes to optimize readability:

```
<div class="p-4">
```

<h2 class="text-xl md:text-3xl lg:text-4xl font-semibold leading-tight">

Improving Typography for Responsive Design

</h2>

Typography plays a crucial role in ensuring a good user experience. With Tailwind CSS, you can easily adjust font sizes, line heights, and other typography aspects based on screen sizes, improving both readability and aesthetics.

Tailwind provides responsive typography utilities to ensure text remains legible across devices.

Example:

<h1 className="text-lg sm:text-xl lg:text-2xl font-bold">
Responsive Heading
</h1>

This example demonstrates responsive font sizes, line heights, and margins that adapt to different screens, ensuring optimal readability across all devices.

- **Interactive elements**

Interactive elements in responsive design are crucial for providing an engaging and user-friendly experience across different devices and screen sizes. When using **Tailwind CSS**, you can create responsive, interactive elements like buttons, forms, and navigation menus by applying utility classes.

✓ **Buttons and Links**

Tailwind provides utilities to style buttons and links that are responsive to user interactions like hover, focus, and active states.

- **Responsive Styles:** You can control the size and padding of buttons across breakpoints.

For example:<button class="bg-blue-500 text-white px-4 py-2 rounded md:px-6 md:py-3">Click Me</button>

- **Interactive States:**

<button class="bg-blue-500 hover:bg-blue-600 focus:ring focus:ring-blue-300">

Hover or Focus Me

</button>

✓ **Form Elements**

Forms are a key interactive element in many responsive designs, and Tailwind allows you to style inputs and text areas consistently.

- **Responsive Input Fields**

  Example:

  <input class="border p-2 w-full sm:w-1/2 md:w-1/3 lg:w-1/4 focus:outline-none focus:ring focus:ring-indigo-500" type="text" placeholder="Enter your name">

- **Focus and Active States:**

  Example:

  <input class="border border-gray-300 focus:border-blue-500 focus:ring focus:ring-blue-300" type="text" placeholder="Interactive Field">

✓ **Navigation menus**

Interactive menus can change their layout based on screen size, becoming more compact or expanding into full menus on larger screens.

- **Responsive Navigation**

  Example:

  <nav class="flex flex-col sm:flex-row sm:space-x-4">

  Home

  About

  Contact

```
</nav>
```

- ✓ **Modals and Dropdowns**

Modals and dropdowns are advanced interactive elements. Tailwind CSS helps with toggling their visibility using utilities like hidden or block and managing their layout for different screen sizes.

Example of responsive modal

```
<div class="hidden fixed inset-0 bg-gray-500 bg-opacity-75 transition-opacity" id="modal">

 <div class="bg-white p-6 rounded-lg shadow-lg max-w-sm mx-auto mt-10 sm:max-w-md md:max-w-lg">

  <h2 class="text-lg font-semibold">Responsive Modal</h2>

  <p class="mt-2">This modal adapts to screen size.</p>

 </div>

</div>
```

- ✓ **Hover, Focus, and Active States for interactivity**

Tailwind's hover:, focus:, and active: modifiers allow you to easily add interactivity to elements, making sure they respond to user inputs and screen sizes.

Example:

```
<button class="bg-indigo-500 text-white p-2 rounded hover:bg-indigo-600 focus:ring focus:ring-indigo-300 active:bg-indigo-700">

 Interactive Button

</button>
```

- ✓ **Touch-Friendly Elements**

Tailwind helps ensure touch-friendly design, which is crucial for mobile devices.

Example: <button class="p-4 md:p-6">Touch-Friendly Button</button>

- **Testing and Iteration**

Testing and Iteration in responsive design, especially when using **Tailwind CSS**, involves evaluating and refining your web application to ensure it adapts well to various screen sizes and devices.

✓ **Initial Testing**

- **Browser Developer Tools**: Use browser dev tools (Chrome, Firefox, Safari) to test responsiveness. Tailwind CSS makes it easy to see how classes like sm:, md:, lg:, and xl: work at different screen widths.
  Inspect the page using the **responsive mode** to check breakpoints and design behavior across devices like smartphones, tablets, and desktops.
- **Real Device Testing**: Testing on physical devices is crucial as emulators may not always reflect actual performance or display characteristics.
- **Viewport Variations**: Test for a wide range of viewports, from very small mobile screens (320px) to large desktop screens (1920px+).

✓ **User Interactions**

- Ensure interactive elements like buttons, navigation menus, and input fields are usable and accessible across all screen sizes. Tailwind allows easy styling adjustments with classes such as hover:, focus:, and active:.
- Touch Interaction: Test touch functionality for mobile devices (e.g., touch-friendly buttons, swipe actions).

✓ **Performance Optimization**

- Use tools like Lighthouse to assess the performance of your responsive design, focusing on metrics like loading speed, mobile-friendliness, and layout shifts.
- Tailwind's utility classes ensure you are not shipping unnecessary CSS, which enhances performance. You can also purge unused CSS using Tailwind's purge feature in production.

✓ **Iteration**

- **Analyze User Feedback**: If users encounter issues on specific devices or browsers, use this feedback to iterate and adjust breakpoints or tweak the UI.
- **A/B Testing**: Conduct A/B tests with different layout approaches to identify which version offers the best user experience.
- **Media Query Adjustments**: As new devices with varying screen sizes emerge, periodically review your media queries and adjust Tailwind's breakpoints if necessary.

✓ **Accessibility Testing**

- Ensure that the design is accessible for all users, including those with disabilities. Use Tailwind to implement accessibility-friendly classes like sr-only for screen readers and test with tools like **axe** or **Wave**.

Tools and Best Practices for Tailwind CSS:

- **PurgeCSS** to remove unused CSS in production, reducing file size.
- **PostCSS plugins** to enhance your CSS capabilities and optimize builds.
- **Tailwind's JIT mode** to build classes on demand, which speeds up development and iteration.



Practical Activity 2.2.2: Applying responsive design principles



Task:

1:You are requested to go to the computer lab to apply the responsive design principles.
2: Read the key readings 2.2.2
3: Apply safety precautions
4:Apply the responsive principles
5:Present your work to the trainer
4: Perform the task provided in application of learning 2.2



Key readings 2.2.2: Applying responsive design principles

Step 1:Browse to the Folder Where Your Project is Created

Open **File Explorer** (Windows) or **Finder** (Mac) and navigate to the directory where your React project is located.

Step 2: Open Command Prompt

In **Windows**, hold Shift and right-click inside the folder, then select "Open PowerShell window here" or "Open Command Prompt here."

In **Mac/Linux**, open **Terminal**, type cd, drag the folder into the terminal window, and press Enter.

Step 3: Type the Command to Open the Project in Your Text Editor

In the terminal, type the following command to open the project in **VS Code** (if you're using it):code .

Step 4: Access the React Component

Inside **VS Code**, open the src folder and navigate to the React component you want to make responsive, such as App.js or any other component.

Step 5: Apply Responsive Principles with Tailwind CSS

- **Mobile-First Approach**

Tailwind encourages designing for smaller screens first and then enhancing the layout for larger screens using breakpoints.

Example:

```
<div className="p-4 text-sm sm:text-base md:text-lg lg:text-xl">
  Mobile-First Design Example
</div>
```

- ✓ text-sm: Applied on mobile devices.
- ✓ sm:text-base: Larger text for small screens (640px and up).
- ✓ md:text-lg: Larger text for medium screens (768px and up).
- ✓ lg:text-xl: Larger text for large screens (1024px and up).

- **Flexible Grid Layouts**

Tailwind's grid system can be made responsive by using grid classes that adjust the number of columns based on screen size.

Example:

```
<div className="grid grid-cols-1 sm:grid-cols-2 md:grid-cols-3 lg:grid-cols-4 gap-4">
  <div className="bg-blue-500 p-4">Item 1</div>
  <div className="bg-green-500 p-4">Item 2</div>
  <div className="bg-red-500 p-4">Item 3</div>
  <div className="bg-yellow-500 p-4">Item 4</div>
</div>
```

- ✓ grid-cols-1: One column on small screens.
- ✓ sm:grid-cols-2: Two columns on small screens (640px and up).
- ✓ md:grid-cols-3: Three columns on medium screens (768px and up).
- ✓ lg:grid-cols-4: Four columns on large screens (1024px and up).

- **Responsive Images and Media**

To ensure images and media adapt to different screen sizes, Tailwind provides responsive utilities.

Example:

```
<img className="w-full h-auto md:w-1/2 lg:w-1/3" src="image.jpg" alt="Responsive Image" />
```

- ✓ w-full: Full width on small screens.
- ✓ md:w-1/2: Half width on medium screens (768px and up).
- ✓ lg:w-1/3: One-third width on large screens (1024px and up).

- **Media Queries and Breakpoints**

Tailwind has predefined media query breakpoints for responsive design:

- ✓ sm: 640px and up
- ✓ md: 768px and up
- ✓ lg: 1024px and up
- ✓ xl: 1280px and up

Example:

```
<div className="p-4 sm:p-8 md:p-12 lg:p-16">
  This padding increases with screen size.
</div>
```

- ✓ p-4: Padding on all sides for small screens.
- ✓ sm:p-8: Larger padding for small screens (640px and up).
- ✓ md:p-12: Even larger padding for medium screens (768px and up).
- ✓ lg:p-16: Largest padding for large screens (1024px and up).

- **Typography and Readability**

Ensure typography scales appropriately on different screen sizes by adjusting text size, line height, and letter spacing.

Example:

```
<h1 className="text-lg sm:text-2xl md:text-3xl lg:text-4xl font-bold">
  Responsive Typography Example
</h1>
```

- ✓ text-lg: Large text size on small screens.
- ✓ sm:text-2xl: Larger text size for small screens.
- ✓ md:text-3xl: Even larger text size for medium screens.
- ✓ lg:text-4xl: The largest text size for large screens.

- **Interactive Elements**

Make sure interactive elements, like buttons and links, respond to user actions such as hover and focus.

Example:

```
<button className="bg-blue-500 text-white p-4 rounded hover:bg-blue-700 focus:bg-blue-900">
  Interactive Button
</button>
```

- ✓ hover:bg-blue-700: Change background color on hover.
- ✓ focus:bg-blue-900: Change background color when focused.

- **Testing and Iteration**

After applying the responsive principles, it's crucial to test the design across multiple devices and screen sizes. You can use browser developer tools to emulate different screen sizes and make necessary adjustments.

- ✓ **Chrome DevTools**: Right-click on the page → Inspect → Toggle device toolbar.
- ✓ **Test on Actual Devices**: Ensure the app looks good on actual mobile, tablet, and desktop devices.
- ✓ **Iterate**: Refine the design based on test results.

Example putting it all together:

```
function ResponsiveComponent() {
  return (
<div className="p-4 sm:p-8 md:p-12 lg:p-16 bg-gray-100">
```

```
<h1 className="text-lg sm:text-2xl md:text-3xl lg:text-4xl font-bold">
Responsive Design Example
</h1>
    <div className="grid grid-cols-1 sm:grid-cols-2 md:grid-cols-3 lg:grid-cols-4 gap-4 mt-4">
<div className="bg-blue-500 text-white p-4">Item 1</div>
<div className="bg-green-500 text-white p-4">Item 2</div>
<div className="bg-red-500 text-white p-4">Item 3</div>
<div className="bg-yellow-500 text-white p-4">Item 4</div>
</div>
<img className="w-full h-auto mt-4 md:w-1/2 lg:w-1/3" src="image.jpg" alt="Responsive" />
    <button className="bg-blue-500 text-white p-4 rounded hover:bg-blue-700 focus:bg-blue-900 mt-4">
      Responsive Button
    </button>
   </div>
  );
}
export default ResponsiveComponent;
```



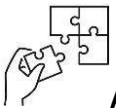
Points to Remember

- **Mobile-First Approach** is a strategy where you design your website or application starting with the smallest screen sizes (typically mobile devices) and progressively enhance the layout as the screen size increases.
- **Flexible Grid Layouts** ensure that the layout adapts to different screen sizes and resolutions.
- Responsive images and media by using **w-full, object-cover, object-contain** and **aspect-ratio** utiliy classes
- Five default breakpoints: **sm,md,lg,xl** and **2xl**.
- **Typography and readability** are crucial components of responsive design, ensuring text remains legible and aesthetically pleasing across various devices and screen sizes.
- **Interactive elements** in responsive design are crucial for providing an engaging and user-friendly experience across different devices and screen sizes.
- **Testing and Iteration** in responsive design, especially when using Tailwind CSS, involves evaluating and refining your web application to ensure it adapts well to various screen sizes and devices.

- To apply the responsive design principles with Tailwind CSS you do the followings:

✓ Browse to the folder where your project is created
✓ Open command prompt
✓ Type the command to open the project in your text editor
✓ Access the react component
✓ Apply the following responsive design principles with Tailwind CSS :

- Mobile-First Approach
- Flexible Grid Layouts
- Responsive Images and Media
- Media Queries and Breakpoints
- Typography and Readability
- Interactive Elements
- Testing and Iteration



Application of learning 2.2.

MXC is an online car selling company located in Niboye sector, Kicukiro District in Kigali city. It has an online car-selling platform that is not seemless and interactive. It has fixed images that only clearly browse on Laptop devices. As a ReactJS developer, you are requested to upgrade their website pages using ReactJS and Tailwind CSS, focusing on responsive design principles. The new ReactJS application should showcase articles, images, and interactive elements, ensuring a seamless experience across devices.

Indicative Content 2.3: Customization of Tailwind Styles

Duration: 3 hrs

Practical Activity 2.3.1: Customizing Tailwind styles

Task:

1: You are requested to go to the computer lab to customize the styles.
2: Read the key readings 2.3.1
3: Apply safety precautions
4:Customize the styles
5:Present your work to the trainer
6: Perform the task provided in application of learning 2.3

Key readings 2.3.1: Customizing Tailwind styles

Step 1: Browse to the folder where your project is created
Open **File Explorer** (Windows) or **Finder** (Mac) and navigate to the folder where your React project is located.
Step 2: Open command prompt
In **Windows**, hold Shift and right-click inside the folder, then select "Open PowerShell window here" or "Open Command Prompt here."
In **Mac/Linux**, open **Terminal**, type cd, drag the folder into the terminal window, and press Enter.
Step 3: Type the command to open the project in your text Editor
In the terminal, type the following command to open your project in **VS Code** (if you're using it):**code .**
Step 4: Access the React component
Inside **VS Code**, navigate to the src folder and open the React component you want to style, such as App.js, Header.js, etc.
Step 5: Customize the Default Theme
To customize Tailwind's default theme, modify the tailwind.config.js file. You can extend or override default styles, such as colors, spacing, fonts, and more.
Example:

```
module.exports = {
  theme: {
    extend: {
```

```
    colors: {
      primary: '#1D4ED8',  // Custom primary color
      secondary: '#64748B', // Custom secondary color
    },
    spacing: {
      72: '18rem',        // Custom spacing (18rem = 288px)
    },
  },
 },
}
```

Step 6: Customize the variants

In the **tailwind.config.js** file, you can also configure which variants (responsive, hover, focus, etc.) should be generated for each utility.

Example:

```
module.exports = {
 variants: {
  extend: {
    backgroundColor: ['hover', 'focus', 'active'],
    textColor: ['hover', 'focus', 'group-hover'],
  },
 },
};
```

This will allow you to apply different styles based on states like hover, focus, active, or group-hover.

Step 7: Customize fonts and typography

You can set custom fonts and typography by extending the fontFamily property in **tailwind.config.js**. You may import custom fonts from Google Fonts or locally.

Example:

```
module.exports = {
 theme: {
  extend: {
    fontFamily: {
      customFont: ['"Gabriola"', 'serif'],
    },
  },
 },
};
```

Then, you can apply the custom font in your React component like this:

```
<div className="font-customFont text-lg">
 Customized Typography with Tailwind CSS
```

```
</div>
```

Step 8: Customize colors

Extend or override colors for branding or theme purposes by defining your custom colors in the tailwind.config.js file.

Example:

```
module.exports = {
  theme: {
    extend: {
      colors: {
        customBlue: '#1DA1F2',  // Custom color for Twitter-like blue
      },
    },
  },
}
```

Use the custom color in your component:

```
<div className="bg-customBlue text-white p-4">
  Custom Color Example
</div>
```

Step 9: Customize the functionality with plugins

Tailwind CSS allows you to install and use plugins to add additional utilities or customize complex functionality.

Example: To install the **Typography plugin**:

```
npm install @tailwindcss/typography
```

Then, enable it in your tailwind.config.js file:

```
module.exports = {
  plugins: [
    require('@tailwindcss/typography'),
  ],
}
```

Now, use the plugin's utility classes in your component:

```
<div className="prose">
  <h1>Typography Plugin Example</h1>
  <p>This text is styled using the Tailwind Typography plugin.</p>
</div>
```

Step 10: Customize complex design with directives

Use **@apply** in Tailwind to combine multiple utility classes into reusable styles or apply complex styling directives.

Example: In your CSS file:

```
.btn-primary {
  @apply bg-blue-500 text-white font-bold py-2 px-4 rounded;
```

```
}
.grid-container {
  @apply grid grid-cols-2 gap-4 md:grid-cols-4;
}
```

In your component:

```
<button className="btn-primary">
  Primary Button
</button>
<div className="grid-container">
  <div>Grid Item 1</div>
  <div>Grid Item 2</div>
  <div>Grid Item 3</div>
  <div>Grid Item 4</div>
</div>
```

Step 11: Use conditional styles with JavaScript

You can conditionally apply Tailwind classes in React components based on state, props, or logic by using JavaScript.

Example:

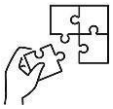
```
import { useState } from 'react';
function ConditionalStylesButton() {
  const [isActive, setIsActive] = useState(false);

  return (
    <button
      className={`p-4 rounded-lg ${
        isActive ? 'bg-customGreen text-white' : 'bg-gray-400 text-black'
      }`}
      onClick={() => setIsActive(!isActive)}
    >
      {isActive ? 'Active' : 'Inactive'}
    </button>
  );
}
```

In this example, the button's background and text color change based on the isActive state.

Points to Remember

- To customize the tailwind styles with Tailwind CSS you do the followings:
  - ✓ Browse to the folder where your project is created
  - ✓ Open command prompt
  - ✓ Type the command to open the project in your text editor
  - ✓ Access the react component
  - ✓ Customize the default theme
  - ✓ Customize the variants
  - ✓ Customize fonts and typography
  - ✓ Customize colors
  - ✓ Customize the functionality with plugins
  - ✓ Customize the complex design with directives
  - ✓ Use conditional styles with Javascript



Application of learning 2.3.

Web Tech is a web application development company located in Rwamagana District, Eastern Province – Rwanda. It receives customers with different requests pertaining to websites and other applications. Assume you are a front-end developer for the company. You are working on a customer's ReactJS project where you need to create a custom, responsive landing page for a website. You are using Tailwind CSS for styling. Assume one of the clients with some ReactJS knowledge has specific design requirements that go beyond the default Tailwind configuration indicated below.

1. **Extend the default theme**:
   - The primary color should be a custom shade of blue (#1A73E8), with secondary color as a custom shade of gray (#333333).
   - Spacing units should include 72px (18rem).
2. **Add custom variants**:
   - Add custom hover and focus variants for background color, so that the button background changes to green on hover and red on focus.
3. **Customize fonts and typography**:
   - Use a custom font Gabriola with a fallback of serif.
4. **Customize colors**:
   - Create a light and dark mode switch using custom color schemes. In light mode, the background should be white with dark text, and in dark mode, the background should be black with light text.

5. **Customize directives for complex designs**:
    - Create a custom directive for a responsive 3-column grid layout, adjusting to a 1-column layout on mobile.

Use the above customer specifications to come-up with a required final landing page as required by the customer.



Learning Outcome 2 End Assessment

Theoretical assessment

Q1. Read carefully the following statements about Tailwind CSS framework and answer by TRUE if the statement is correct and by FALSE if the statement is incorrect.

a) Tailwind CSS does not support grid-based layouts for creating flexible, responsive designs.
b) The mobile-first approach means designing for smaller screens first and progressively enhancing the layout for larger screens.
c) Tailwind CSS does not allow developers to customize typography or scale font sizes for responsive design.
d) Tailwind CSS offers utility classes to style interactive elements such as buttons, forms, and hover states, ensuring responsiveness.
e) Testing responsiveness and iterating on design are not necessary when using Tailwind CSS because it automatically makes everything responsive.
f) You can extend Tailwind's default theme by modifying the tailwind.config.js file.
g) Custom fonts can be added to Tailwind by specifying them in the theme's fontFamily section of the configuration file.
h) Tailwind CSS plugins can add additional utilities and variants for extended functionality.
i) You can apply conditional styles in Tailwind by integrating JavaScript logic using libraries like Alpine.js or React's state management.

Q2. Fill in the blank spaces with appropriate word(s). Select from the given choices in the box.

grid, colors, focus state flex-initial, tailwind.config.js, Hover state, Animation, transition, animate-bounce, delay-*, mobile-first, and media queries

a) …….. in Tailwind CSS is a feature that allows you to apply styles to an element when a user hovers over it with their cursor.
b) You can add plugins by installing them and then including them in the plugins array of ……
c) The …… approach is a design strategy where you prioritize mobile users by designing for smaller screens first and then progressively enhancing for larger screens.

d) Add the **……** utility to make an element bounce up and down, useful for things like “scroll down” indicators.
e) **……..** is the ability to apply motion effects to elements, either using pre-defined utility classes or custom animations.
f) You can customize the primary color by extending the ……. property in tailwind.config.js
g) Use the **……** utilities to control an element’s transition-delay.
h) TailwindCSS provides pre-configured …….. through its breakpoint utilities like sm, md, lg, xl, and 2xl.
i) Use **……** to allow a flex item to shrink but not grow, taking into account its initial size.
j) In TailwindCSS, you can implement flexible grid layouts using the …….. utility class and specifying the number of columns with grid-cols-{n}.

Q3.Match Column A and Column B

Column A		**Column B**
1.Utilities for controlling gutters between grid and flexbox items		**A.** Auto-cols-min
2. Utility for controlling the size of implicitly-created grid rows.		**B.** shrink-0
3. Utility for controlling the size of implicitly-created grid columns.		**C.** animate-pulse
4. Utility for specifying the rows in a grid layout.		**D.** Gap
5. It is used to prevent a flex item from shrinking.		**E.** grow-0
6. It is used to make an element gently fade in and out, useful for things like skeleton loaders.		**F.** Auto-rows-fr
7.It is used to add the margins to paragraph.		**G**.grid-rows-5
8.It is used to apply the line spacing to text.		**H.**md
9.Example of breakpoint		**I.**m-4
		J. leading-normal

10. It is used to apply the extra-large size to text		**K.** text-xl
		L. text-2xl
		M. p-4

Practical assessment

ABC Company is a software development company located in Nyanza district, Southern Province – Rwanda. The Company has developed a ReactJS application with unpleasant and not attractive design. It wants to improve its website development Techniques to modern ones. Assume it has hired you as a ReactJS expert and tasked you to develop a modern, responsive web application using React.JS and Tailwind CSS with a clean and consistent design across all devices. You are requested to integrate animations, transitions, handle various states like hover and focus, and follow a mobile-first approach. Additionally, the application requires custom styles to match the branding of the company, including custom fonts, colours, and layout directives.

END



References

Tailwind CSS. (n.d.). *Create React App*. https://tailwindcss.com/docs/guides/create-react-app

Tailwind CSS. (n.d.). *Plugins*. https://tailwindcss.com/docs/plugins

Tailwind CSS. (n.d.). *Installation*. Tailwind CSS. Retrieved October 14, 2024, from https://tailwindcss.com/docs/installation

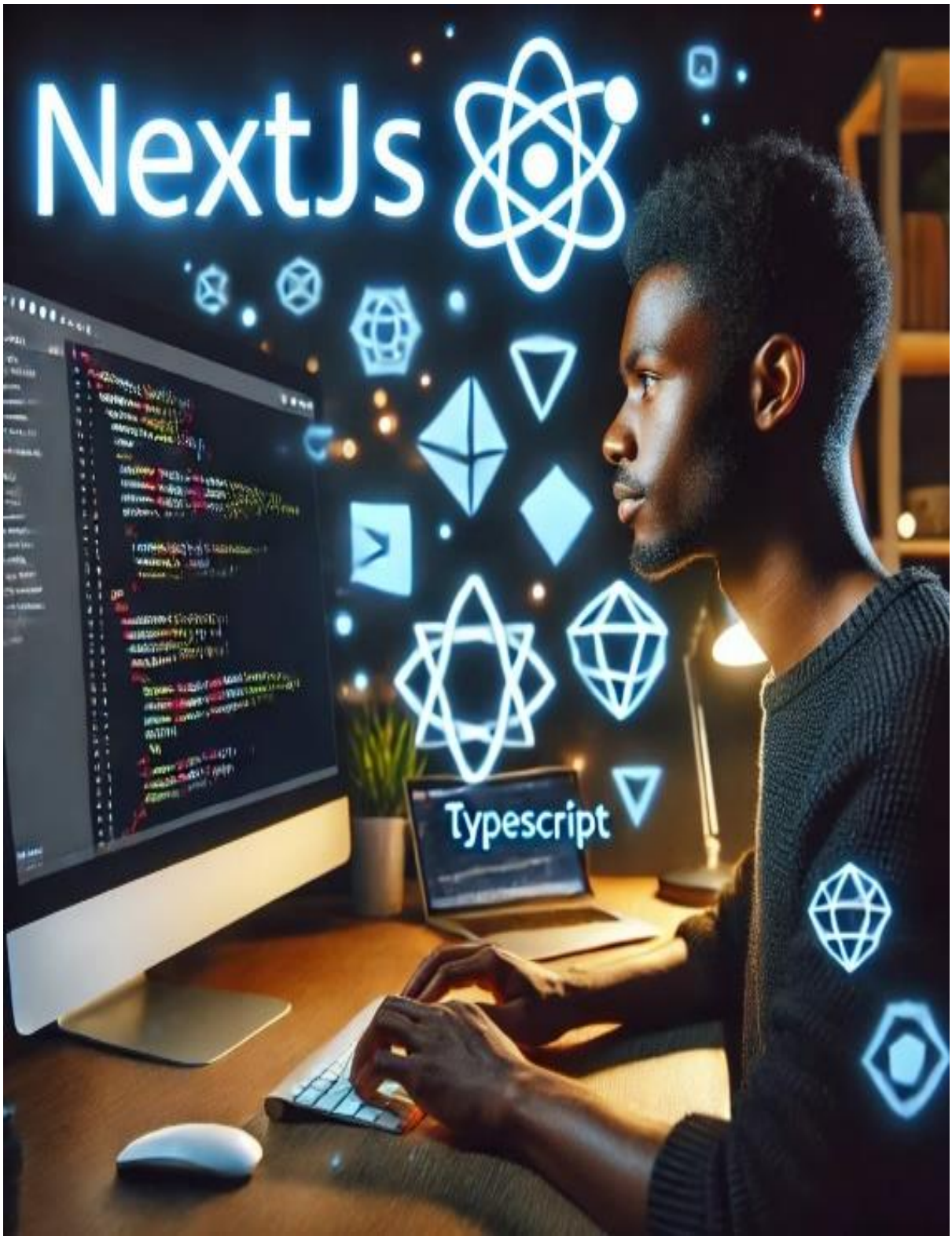
Tailwind CSS. (n.d.). *Grid template columns*. Tailwind CSS. https://tailwindcss.com/docs/grid-template-columns

Tailwind CSS. (n.d.). *Gap*. Tailwind CSS. https://tailwindcss.com/docs/gap

Tailwind CSS. (n.d.). *Functions and directives*. Tailwind CSS Documentation. https://v2.tailwindcss.com/docs/functions-and-directives#screen-1

Refine. (n.d.). *Tailwind CSS flex: A comprehensive guide*. Refine. https://refine.dev/blog/tailwind-flex/#icons

House, C. (2024, September 26). *CSS grid layout guide*. CSS-Tricks. https://css-tricks.com/snippets/css/complete-guide-grid/#aa-css-grid-properties


NextJs
Typescript

Indicative contents
3.1 Applying Typescript Basics **3.2 Setup Nextjs Project** **3.3 Implementing Rendering Techniques** **3.4 Implementing Routing** **3.5 Creation of API** **3.6 Securing the Application**

Key Competencies for Learning Outcome 3: Develop Next.JS Application

Knowledge	Skills	Attitudes
• Implementation of Rendering Techniques • Description of routing concepts • Definition of API Endpoint	• Setting up TypeScript environment • Implementing interface of variables • Performing functions and data Handling in TypeScript • Creating NextJS project • Initializing NextJS project Development • Implementing Rendering Techniques • Implementing routing • Creating API • Performing Client-Side Security in application • Performing Server-Side Security in application • Performing General Security Measures	• Being curious while developing Next.JS project • Being creative in developing up-to-date next.JS apps • Being Problem-Solving Oriented during development of Next.JS Application • Paying attention to Details during implementation of rendering techniques • Being updated on latest Next.JS versions • Being Adaptable to the Next.JS environment



Duration: 20 hrs

Learning outcome 3 objectives:



By the end of the learning outcome, the trainees will be able to:

1. Describe clearly Rendering Techniques and routing concepts based on user requirements
2. Prepare properly Typescript environment based on project requirements
3. Implement correctly interface of variables based on typescript standard
4. Perform correctly Handling functions and Data Handling based on TypeScript standard
5. Initialize properly NextJS project Development based on project created
6. Routing is properly implemented based on components.
7. API is properly created based on RESTful design principles.
8. Application is properly secured based on system requirements and security standards



Resources

Equipment	Tools	Materials
• Computer	• VS Code • Browser • Terminal • Postman	• Internet • Electricity

Indicative Content 3.1: Applying Typescript Basics

Duration: 5 hrs

Practical Activity 3.1.1: Applying TypeScript basics

Task:

1: You are requested to go to the computer lab to apply TypeScript basics.

2: Read key readings 3.1.1
3: Apply safety precautions.
4:Apply TypeScript basics.
5: Present your work to the trainer and whole class.
6: Perform the task provided in application of learning 3.1

Key readings 3.1.1: Applying TypeScript basics

- **TypeScript Installation and Environment Setup**

TypeScript is a well-liked programming language for creating complicated, large-scale applications. It supersets JavaScript and is an object-oriented, strongly typed language. Several capabilities offered in TypeScript are unavailable in JavaScript, including static typing, interfaces, classes, etc. You can follow the instructions in this article to set up a TypeScript development environment.

✓ **Steps to install TypeScript**

Install Node.js

The first step in setting up a TypeScript development environment is to install Node.js. Node.js is a JavaScript runtime that allows us to execute JavaScript code outside the browser. It is a crucial component of many modern web applications and is required for TypeScript development.

You need to download Node.js to install it. After downloading the installer, complete the installation by adhering to the instructions.

- **For Linux** – Install using terminal by the command 'sudo apt install nodejs'
- **For Windows** – Install the exe file by clicking it and running the executable file.
- **For Mac** – Download Node.js for macOS by clicking the "Macintosh Installer" option and run the downloaded Node.js .pkg Installer

Install TypeScript

Once you have installed Node.js, the next step is to install TypeScript. TypeScript can be installed using the Node Package Manager (npm). To install TypeScript, open a terminal or command prompt and run the following command: **'npm install -g typescript'**

With this command, TypeScript will be installed system-wide. The package is installed globally for use from any directory when the -g flag is used by npm.

✓ **Installing TypeScript using Visual Studio Code**

Visual Studio is a popular integrated development environment (IDE) that supports multiple programming languages, including TypeScript. To install TypeScript using the Visual Studio extension, follow the steps below:

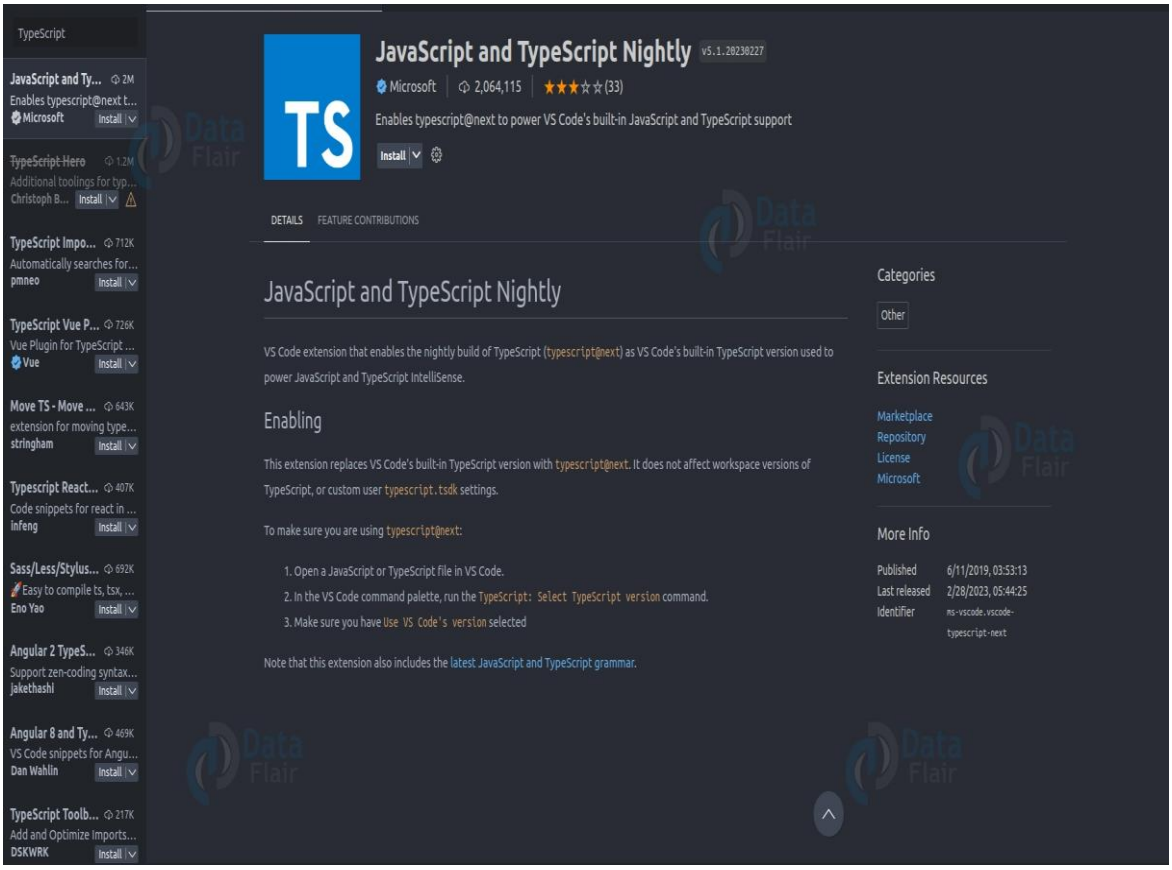
Step 1: Install Visual Studio

You can download Visual Studio from the official Visual Studio website if you still need to install it on your system. Make sure to download and install the version that supports TypeScript.

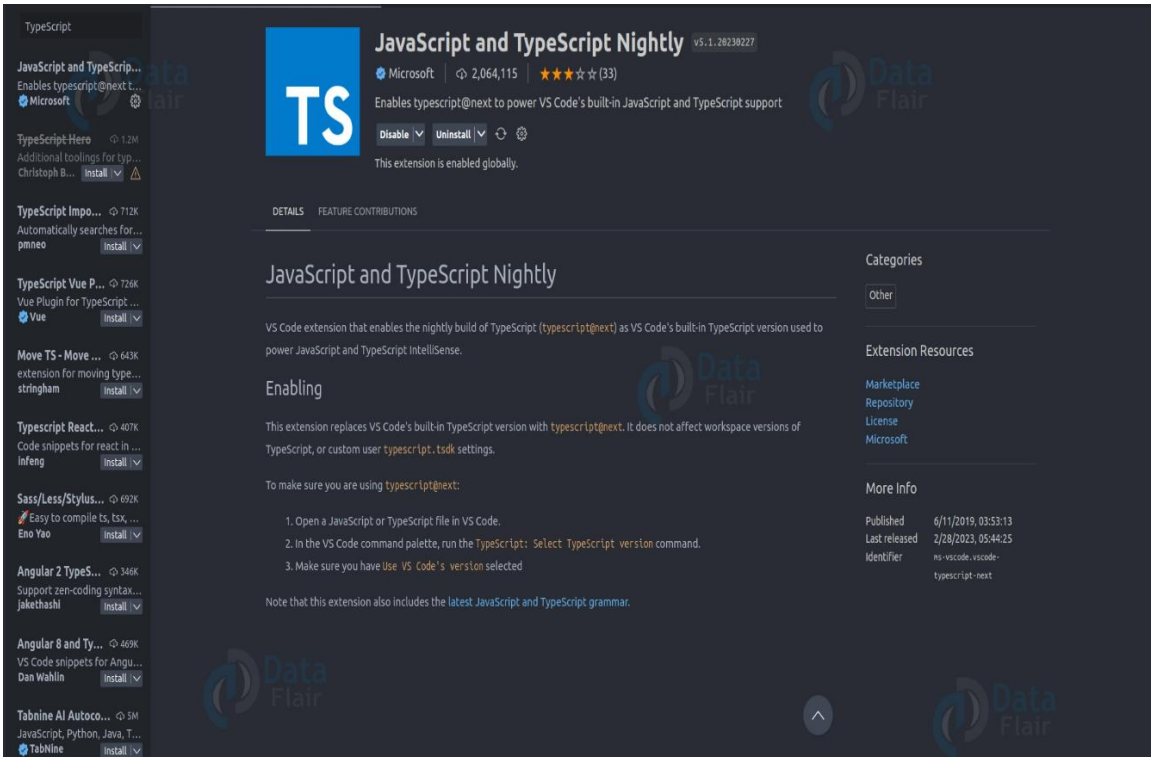
Step 2: Install TypeScript Extension

Once you have installed Visual Studio, you can install the TypeScript extension from the Visual Studio Marketplace. To do this, follow the steps below:

1. Open Visual Studio and click on the "Extensions" menu item from the top menu bar.
2. Click on "Manage Extensions."
3. In the "Extensions and Updates" dialog box, click "Online" from the left side menu.
4. Search for "TypeScript" in the search box and press enter.



5. From the search results, select the TypeScript extension and click on the “Download” button.
6. Once the download is complete, click on the “Install” button.
7. Follow the prompts to complete the installation process.



Step 3: Create a TypeScript project

After installing the TypeScript extension, you can create a new TypeScript project. To do this, follow the steps below:

a. Create a new folder called ‘DataFlair Project’

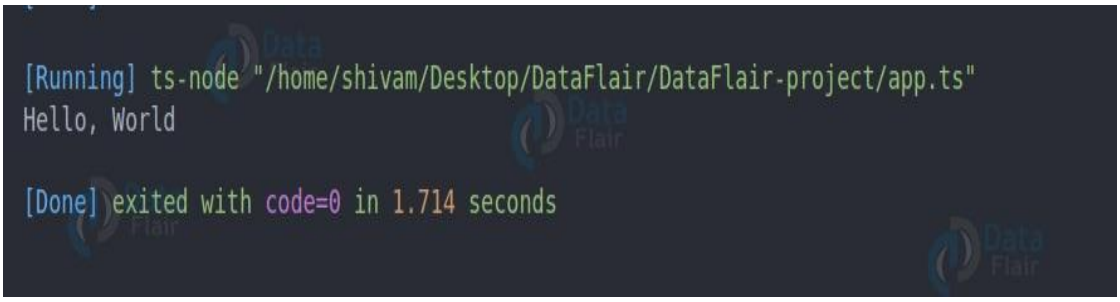
b. Create a new file called ‘app.ts’

c. Write the following code.

```
console.log('Hello, World');
```

d. Run the app using the play button on top right corner.

e. You will get the following output.



- ✓ **Creating a TypeScript Project**

We must build a new directory and start it as a Node.js project before creating a new TypeScript project.

To accomplish this, launch a terminal or command prompt and go to the directory where the project will be created. Run the subsequent commands after that:

'mkdir DataFlair-project'

'cd DataFlair-project'

'npm init -y'

```
→ DataFlair mkdir DataFlair-project
→ DataFlair cd DataFlair-project
→ DataFlair-project npm init -y
Wrote to /home/shivam/Desktop/DataFlair/DataFlair-project/package.json:

{
  "name": "dataflair-project",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}


→ DataFlair-project
```

The first command creates a new directory called DataFlair-project. The second command navigates into the new directory. The third command initializes the directory as a Node.js project and creates a package.json file.

The -y flag tells npm to use the default settings and not ask questions.

- ✓ **Configuring TypeScript**

We need to configure TypeScript for our project now that a new project has been created and TypeScript has been installed. To accomplish this, a tsconfig.json file must be created in the root of our project directory. The TypeScript configuration options are located in this file.

To create the tsconfig.json file, run the following command:

'tsc --init'



This command will create a new tsconfig.json file in the root of our project directory. The tsconfig.json file contains many configuration options, but we only need to change a few.

The first option we need to change is the target option. This option specifies the ECMAScript version that TypeScript will compile our code to. We will set this option to ES2018 to target the latest version of ECMAScript.

The second option we need to change is the outDir option. This option specifies the directory where the compiled TypeScript files will be placed. We will set this option to dist to create a new directory called dist in the root of our project directory.

Here is what our tsconfig.json file should look like:



- ✓ **Writing TypeScript Code**

Now that we have set up our TypeScript development environment, we can start writing TypeScript code. To create a new TypeScript file, create a new file with a .ts extension in the root of our project directory. For example, create a new file called index.ts.

In our index.ts file, we can write TypeScript code just like JavaScript code. However, we can take advantage of the additional features that TypeScript provides, such as static typing, interfaces, and classes.

For example, let's create a simple function that adds two numbers together and returns the result. We will define the types for the function parameters and return values using TypeScript's static typing:

function DataFlair_addNumbers(num1: number, num2: number): number {

return num1 + num2;

}

console.log(DataFlair_addNumbers(5, 10));

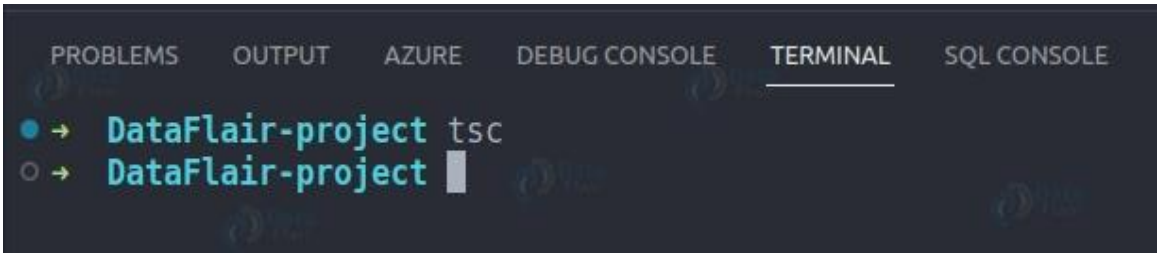
In this example, we have defined a function called DataFlair_addNumbers that takes two parameters of the type number and returns a value of the type number. We have also used TypeScript's static typing to ensure that the function is only called with the correct types of parameters.

✓ **Compiling TypeScript Code**

Once we have written our TypeScript code, we must compile it to JavaScript to be executed in a web browser or a Node.js environment. To compile our TypeScript code, we can use the tsc command.

To compile our TypeScript code, open a terminal or command prompt and navigate to the root of our project directory. Then, run the following command:

'tsc'



This command will compile all the TypeScript files in our project directory and place the compiled JavaScript files in the dist directory, as specified in our tsconfig.json file.

✓ **Running TypeScript Code**

Now that we have compiled our TypeScript code into JavaScript, we can run it in a web browser or a Node.js environment. To run our code in a web browser, we must create an HTML file referencing our compiled JavaScript file.

In the root of our project directory, create a new file called index.html, and add the following code:

```
<!DOCTYPE html>

<html lang="en">

<head>

<title>DataFlair TypeScript Example</title>

</head>

<body>

<script src="./index.ts"></script>

</body>

</html>
```

This HTML file references our compiled JavaScript file located in the dist directory.

We may use the node command to run our code in a Node.js environment. Open a terminal or command prompt, go to the root of our project directory, and run our index.js file from there. Run the subsequent command after that:

'node index.js'



This command will execute our index.js file in a Node.js environment.

- ✓ **Installing ts-node**

ts-node is a TypeScript execution engine that allows you to run TypeScript code directly without compiling it first. It is a popular tool among TypeScript developers, making it easy to write and test TypeScript code quickly without the extra compilation step.

Here are the steps to install ts-node:

1. Install Node.js:

ts-node requires Node.js to be installed on your computer. You can download and install the latest version of Node.js from the official website: https://nodejs.org/en/download/.

2. Install ts-node:

Once you have Node.js installed, you can install ts-node by running the following command in your terminal or command prompt:



This will install ts-node globally on your computer.

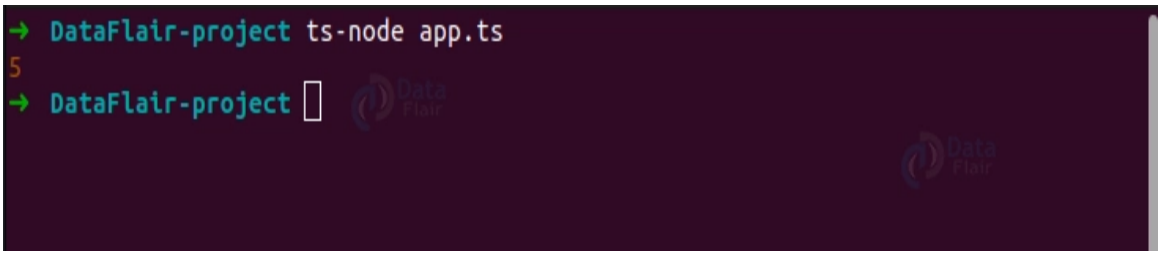
3. Verify installation:

To verify that ts-node is installed correctly, you can run the following command in your terminal or command prompt:



his should output the version number of ts-node that you have installed.

Once you have ts-node installed, you can use it to run TypeScript code directly from the command line. For example, if you have a file named app.ts that contains some TypeScript code, you can run it using ts-node like this:



This will execute the TypeScript code in app.ts without compiling it first.

Note that ts-node is intended for development and testing and is not recommended for use in production environments. Compiling your TypeScript code to JavaScript before deploying it to production is best.

Conclusion

we have walked you through the steps to set up a TypeScript development environment. We have installed Node.js, installed TypeScript using npm, created a new TypeScript project, configured TypeScript for our project, written TypeScript code, compiled our TypeScript code to JavaScript, and run our code in a web browser and a Node.js environment.

TypeScript is a powerful programming language that offers many features unavailable in JavaScript. Setting up a TypeScript development environment allows you to easily take advantage of these features and build large-scale, complex applications.

- **Implementing interface of variables**

Interfaces are a feature of TypeScript that allows us to define the structure or shape of an object and specify the properties and methods that an object has or should have. Their primary function is type checking and aiding developers in catching type-related errors during development.

Here, you can see a small example of how we can define an interface and apply it to an object.

```
interface Person {
        name: string;
        age: number;
        sex: "male" | "female";
}
const personOne: Person = {
        name: "Coner",
        age: 24,
        sex: "male",
}
console.log(personOne.name); // Coner
//  Property 'hobbies' does not exist on type 'Person'
console.log(personOne.hobbies); // undefined
```

As you can see in the above code block, we access a property that is defined in the interface with no issues by running console.log(personOne.name).
We also can see an example of us trying to access a property that doesn't exist in the interface by running console.log(personOne.hobbies), therefore throwing a type error.

- ✓ **Benefits of interfaces in TypeScript**
  - **Type checking**

The first benefit of interfaces is the most obvious one: they highlight any possible type errors and issues in our code to prevent us from accessing any properties that might not exist. This, in turn, helps us reduce runtime errors and prevent bugs from being created.

- **Contract definition**

Another benefit of interfaces is that they define and create clear contracts for the functions and code that consume them. They prevent us from consuming methods and properties that don't exist and help ensure we stay within the established structure defined for the object that the interface is describing.

- **Documentation and readability**

Because interfaces define the properties and methods that exist on an object as well as their types, they act as a form of documentation that enhances the code readability and helps developers reading the code understand how it works and how the code fits together.

- **Reusability**

Since interfaces can always be extended and reused in various places, they promote code reusability and help reduce duplication. By defining central, common interfaces that can

be reused and extended throughout an application, you can ensure consistency in your code and logic.

- **Code navigation and autocompletion**

IDEs that integrate with TypeScript can read the interfaces you define and offer autocompletion suggestions from them, as well as help with code navigation to make you a more productive and efficient developer.

- **Easier refactoring**

Finally, interfaces help make refactoring easier because you're able to update the implementation of a piece of code or logic, and as long as it adheres to the same interface, other code that depends on the changed logic shouldn't be impacted.

✓ **Using interfaces in TypeScript**

- **Function types**

In addition to defining the types of objects, we can also use interfaces to type functions, their return values, and their arguments. For example, we can do something like this.

```
interface Args {
  name: string;
  age: number;
}
interface Return {
  name: string;
  age: number;
  doubledAge: number
}
function ageDoubler({name, age}: Args): Return {
  return {
    name,
    age,
    doubledAge: age * 2,
  }
}
```

- **Classes**

TypeScript has native support for the class keyword that was implemented in ES2015. You can define a class as well as its fields and methods like this.

```
class Person {
  name: string = '';
  age: number = 0;
}
const me = new Person();
```

But, we can combine these class definitions with interfaces to make sure the class correctly implements all of the properties defined on the interface like so.

```
interface PersonInt {
  name: string;
  age: number;
}
// Class 'Person' incorrectly implements interface 'PersonInt'.
// Property 'age' is missing in type 'Person' but required in type 'PersonInt'
class Person implements PersonInt {
  name: string = '';
}
const me = new Person();
```

In this example, we have an interface called PersonInt and use the implements keyword to say the class Person will have all of the types defined in PersonInt. Because this isn't true and the age field is missing in the class, an error is thrown.

- **Optional properties**

When working with objects in TypeScript, it's quite common to have properties that might only be defined some of the time. In these instances, we can define optional properties like so.

```
interface Person {
        name: string;
        age: number;
        //  Note the ?: makes the property optional
        color?: string;
}
```

Now, when we consume the color property on an object typed with the Person interface, we'll have to account for the fact that it might not be present (it'll be undefined if not defined).

- **readonly properties**

In TypeScript, we can use the readonly keyword with interfaces to mark a property as readonly. This means that the target property can't be written to during type-checking although its behavior doesn't change during runtime.

```
interface Person {
  readonly name: string;
}
const person: Person = {
        name: 'Coner',
}
function updateName(person: Person) {
  // We can read from 'person.name'.
```

```
  console.log(`name has the value '${person.name}'.`); // "name has the value 'Coner'."
  // But we can't re-assign it.
  // Cannot assign to 'name' because it is a read-only property.
  person.name = "hello";
}
```

- **Index signatures**

There might be a time when you know the shape of your object, but you don't know the actual properties of it. Or, the properties might change, but the shape will remain consistent. In these situations, it's not practical or potentially possible to type every single property on the interface. To to get around this, we can use index signatures.

```
interface Index {
        [key: string]: boolean
}
```

What this interface says is if we index an object that is typed using the Index interface with a string, we'll have a boolean returned to us. You're not limited to just boolean types, either. It could also be another type or interface if you wish, which is great for times when you don't know all of the properties but know their shape.

Also, if you want to combine index signatures and normal interface definitions, you can do so. However, if you do this, the index signature needs to be updated to contain all of the potential return types.

```
interface Index {
        one: string;
        two: number;
        [key: string]: string | number | boolean
}
```

It's worth noting that while index signatures can make your life easier, where possible and feasible, you should always reach for actually typing properties on an object as that'll give you better type safety.

- **Extending interfaces**

Sometimes, you want to extend an existing interface and add new fields to it without changing the original one. This can be achieved by using the extends keyword. This allows you to take an existing interface and create a copy of it while also adding new fields to it. For example, we could do something like this.

```
interface Person {
  name: string;
  age: string;
}
interface PersonWithHobbies extends Person {
  hobbies: string[];
}
```

```
/*
 PersonWithHobbies would be:
interface PersonWithHobbies {
  name: string;
  age: string;
  hobbies: string[];
}
*/
```

In this example, we took the original Person interface and extended it with the hobbies property to create a new interface called PersonWithHobbies. So, at this point, we have two interfaces, Person and PersonWithHobbies, with them being identical apart from the latter having the hobbies property added to it.

If you want to, you can also combine multiple existing interfaces to create a new one without adding any new properties to it, which can be done like so.

```
interface Person {
  name: string;
  age: string;
}
interface Hobbies {
  hobbies: string[];
}
interface PersonWithHobbies extends Person, Hobbies {}
```

- **Discriminating unions**

Discriminating unions are a way we can define a new type from multiple interfaces and use a common property present on all of the interfaces (the “discriminator”) to distinguish between the types in our logic.

For example, we can define two interfaces for two different shapes (Circle and Square), and we can then use a property present on both of them (kind) to dictate which interface we are dealing with at that moment.

```
interface Circle {
  kind: "circle";
  radius: number;
}
interface Square {
  kind: "square";
  sideLength: number;
}
type Shape = Circle | Square;
function getArea(shape: Shape): number {
  if (shape.kind === "circle") {
```

```
  // We know it's a Circle interface here
  return Math.PI * shape.radius ** 2;
 }
 // We know it's a Square interface here
 return shape.sideLength ** 2;
}
const circle: Shape = { kind: "circle", radius: 5 };
const square: Shape = { kind: "square", sideLength: 4 };
console.log(getArea(circle)); // Output: 78.53981633974483
console.log(getArea(square)); // Output: 16
```

- **Generic interfaces**

Finally, we have Generic Interfaces, which allow us to combine the power of generics with interfaces to create interfaces that consume generics and assign the generic type to one of or multiple properties defined in the interface.

```
interface Item<T> {
 value: T;
}
const stringItem: Item<string> = { value: 'Item' };
const numberItem: Item<number> = { value: 22 };
```

- **Handling functions in TypeScript**
  - ✓ **Functions in Typescript**

In TypeScript, functions play a crucial role, and they allow you to define and use reusable blocks of code. TypeScript adds static typing to JavaScript, so you can specify the types of parameters and the return type of a function .

Here's a basic overview of functions in TypeScript :

- **Function Declaration:**

1. Function without explicit types:

```
function add(a, b) {
  return a + b;
}
let result = add(2, 3); // result is inferred as number
```

2. Function with explicit types:

```
function add(a: number, b: number): number {
  return a + b;
}
let result: number = add(2, 3); // specifying the types explicitly
```

- **Optional and Default Parameters:**

```
function greet(name: string, greeting?: string): string {
  if (greeting) {
    return `${greeting}, ${name}!`;
```

```
  } else {
    return `Hello, ${name}!`;
  }
}

let greetingMessage = greet("John"); // Hello, John!
let customGreeting = greet("Jane", "Good evening"); // Good evening, Jane!
```

Function with Default Parameter:

```
function greet(name: string, greeting: string = "Hello"): string {
  return `${greeting}, ${name}!`;
}

let greetingMessage = greet("John"); // Hello, John!
let customGreeting = greet("Jane", "Good evening"); // Good evening, Jane!
```

- **Rest Parameters:**

```
function sum(numbers: number[]): number {
  return numbers.reduce((total, num) => total + num, 0);
}

let result = sum([1, 2, 3, 4, 5]); // 15

// Using rest parameters
functionsum(...numbers:number[]):number {
  return numbers.reduce((total, num) => total + num, 0);
}

let result = sum(1, 2, 3, 4, 5); // 15
```

Let's go through the code block that demonstrates the use of rest parameters in TypeScript:

```
// Using rest parameters
function sum(...numbers: number[]): number {
  return numbers.reduce((total, num) => total + num, 0);
}

let result = sum(1, 2, 3, 4, 5); // 15
```

- **Function Declaration:**

  function sum(...numbers: number[]): number {: This declares a function named sum that accepts any number of parameters (represented by ...numbers) and all parameters are of type number. The colon : and number indicate that the function returns a value of type number.

- Rest Parameters:

...numbers: number[]: The ...numbers syntax is called rest parameters. It allows the function to accept any number of arguments as an array, and this array is named numbers. The type annotation : number[] specifies that all elements in the array must be of type number.

- Function Body:

return numbers.reduce((total, num) => total + num, 0);: The function body uses the reduce method to sum up all the numbers passed to the function. The initial value for the reduction is 0, and the arrow function (total, num) => total + num is used to accumulate the sum.

- Using the Function:

let result = sum(1, 2, 3, 4, 5);: This line demonstrates how to use the sum function. It calls the function with multiple numeric arguments, and the rest parameter ...numbers collects them into an array. The result is the sum of all the numbers, which is 15 in this case.

Additional Example: You can use the sum function with different numbers:

```
let result1 = sum(1, 2, 3);                    // 6
let result2 = sum(10, 20, 30, 40);             // 100
let result3 = sum(2, 4, 6, 8, 10);   // 30
```

The rest parameters provide a concise way to work with a variable number of arguments in a function, treating them as an array within the function body.

- **Function Types:**

You can also define types for functions:

```
type MathOperation = (a: number, b: number) => number;

let add: MathOperation = (a, b) => a + b;
let subtract: MathOperation = (a, b) => a - b;
```

These are just some basic examples, and TypeScript provides many more features for working with functions, such as function overloading, function types, and more.

- **Data Handling**
  - ✓ **API data validation**

API data validation in TypeScript can be achieved using various strategies to ensure that incoming data matches expected types and structures. Here are some common approaches:

- Using TypeScript Interfaces

You can define interfaces to describe the expected shape of your data. This is helpful for type-checking at compile time.

```
interface User {
  id: number;
```

```
  name: string;
  email: string;
}
// Example function to validate user data
function validateUser(user: any): user is User {
  return (
    typeof user.id === "number" &&
    typeof user.name === "string" &&
    typeof user.email === "string"
  );
}
// Usage
const incomingData: any = { id: 1, name: "John Doe", email: "john@example.com" };
if (validateUser(incomingData)) {
  console.log("Valid user:", incomingData);
} else {
  console.error("Invalid user data");
}
```

- Using zod for Runtime Validation

For runtime validation, libraries like zod provide a powerful way to define schemas and validate data.

1. **Install zod**:

```
npm install zod
```

2. **Create a Schema**:

```
import { z } from "zod";
const UserSchema = z.object({
  id: z.number(),
  name: z.string(),
  email: z.string().email(),
});
// Example of validation
const incomingData = { id: 1, name: "John Doe", email: "john@example.com" };
try {
  const user = UserSchema.parse(incomingData);
```

```
  console.log("Valid user:", user);
} catch (e) {
  console.error("Validation error:", e.errors);
}
```

- Using joi for Schema Validation

joi is another popular library for validating JavaScript objects.

1. **Install joi**:

```
npm install joi
```

2. **Create a Schema**:

```
import Joi from "joi";
const userSchema = Joi.object({
  id: Joi.number().required(),
  name: Joi.string().required(),
  email: Joi.string().email().required(),
});


// Example of validation
const incomingData = { id: 1, name: "John Doe", email: "john@example.com" };
const { error, value } = userSchema.validate(incomingData);
if (error) {
  console.error("Validation error:", error.details);
} else {
  console.log("Valid user:", value);
}
```

- Express Middleware for API Validation

If you're using Express, you can create middleware to validate incoming request bodies.

```
import express from "express";
import { z } from "zod";
const app = express();
app.use(express.json());
const UserSchema = z.object({
  id: z.number(),
```

```
  name: z.string(),
  email: z.string().email(),
});
app.post("/api/users", (req, res) => {
  try {
    const user = UserSchema.parse(req.body);
    // Handle valid user data
    res.status(201).json(user);
  } catch (e) {
    res.status(400).json({ errors: e.errors });
  }
});
app.listen(3000, () => {
  console.log("Server running on http://localhost:3000");
});
```

Conclusion

Choosing the right approach for API data validation in TypeScript depends on your needs. For static type checking, interfaces and type guards are useful. For runtime validation, libraries like zod and joi are more powerful and flexible. Integrating validation with frameworks like Express can help maintain the integrity of your API.

- ✓ **Form Validation**

Form validation ensures that the user input is correct before submission. TypeScript can help enforce type safety on the input values and provide better tooling and error checking.

```
interface FormInput {
  username: string;
  password: string;
}
function validateFormInput(input: FormInput): boolean {
```

```
  const usernameIsValid = input.username.length > 0;
  const passwordIsValid = input.password.length >= 8;
    return usernameIsValid && passwordIsValid;
}
// Example usage
const formInput: FormInput = { username: 'user', password: 'securePassword' };
if (validateFormInput(formInput)) {
  // Submit the form
} else {
  // Show validation error
}
```

3. Error Handling and Exceptions

Error handling in TypeScript can be done using try-catch blocks, allowing you to handle runtime errors gracefully. You can also define custom error classes for more specific error management.

```
class CustomError extends Error {
  constructor(message: string) {
    super(message);
    this.name = 'CustomError';
  }
}
function riskyOperation() {
  throw new CustomError('Something went wrong!');
}
try {
  riskyOperation();
} catch (error) {
  if (error instanceof CustomError) {
    console.error('Caught a custom error:', error.message);
  } else {
    console.error('An unexpected error occurred:', error);
  } }
```

Summary

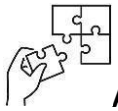
In TypeScript, you can enforce data structures and types for both API and form validation, helping to catch errors at compile time. Additionally, structured error handling using custom error classes and try-catch blocks allows for robust and maintainable code.



Points to Remember

Steps used to apply the TypeScript basics

- npm install -g typescript command to install Typescripts using terminal
- Steps to implement interfaces for variables in TypeScript
  1. Defining an Interface
  2. Implementing the defined Interface for Variables
  3. Optional Properties in Interfaces
  4. Read-Only Properties in Interfaces
  5. Implementing Methods in Interfaces
  6. Extending Interfaces
  7. Indexable Types in Interfaces
- Data Handling like API data validation, Form validation and, error handling and exceptions



Application of learning 3.1.

ABC Web Development Company is an IT company in Kigali City – Rwanda. Assume they have hired you as a full stack developer, you are tasked with creating a task management app using ReactJS and TypeScript. Install and configure TypeScript, ensuring the tsconfig.json file is set up. Implement interfaces to define task objects, then develop functions to add, update, and manage tasks. Make sure that API data validation is implemented to ensure data integrity, followed by form validation for user inputs. Finally, integrate additional packages, such as Formik for form handling or Axios for API requests, enhancing functionality and performance

Indicative Content 3.2: Setup Nextjs Project

Duration: 2 hrs

Practical Activity 3.2.1: Setup NextJS project

Task:

1: You are requested to go to the computer lab to setup NextJS project.
2: Read key reading 3.2.1
3: Apply safety precautions.
4: Setup NextJS project.
5: Present your work to the trainer and whole class.
6: Perform the task provided in application of learning 3.2

Key readings 3.2.1: Setup NextJS project

- **Installation**

✓ **Automatic Installation**

We recommend starting a new Next.js app using create-next-app, which sets up everything automatically for you. To create a project, run:
npx create-next-app@latest
On installation, you'll see the following prompts:
What is your project named?
my-appWould you like to use TypeScript? No / Yes
Would you like to use ESLint? No / Yes
Would you like to use Tailwind CSS? No / Yes
Would you like your code inside a `src/` directory? No / Yes
Would you like to use App Router? (recommended) No / Yes
Would you like to use Turbopack for `next dev`? No / Yes
Would you like to customize the import alias (`@/*` by default)? No / Yes
What import alias would you like configured? @/*
After the prompts, create-next-app will create a folder with your project name and install the required dependencies.

✓ **Manual Installation**

To manually create a new Next.js app, install the required packages:
Terminal
npm install next@latest react@latest react-dom@latest
Open your package.json file and add the following scripts:

package.json
{ "scripts": { "dev": "next dev", "build": "next build", "start": "next start", "lint": "next lint" }}
These scripts refer to the different stages of developing an application:

- dev: runs next dev to start Next.js in development mode.
- build: runs next build to build the application for production usage.
- start: runs next start to start a Next.js production server.
- lint: runs next lint to set up Next.js' built-in ESLint configuration.

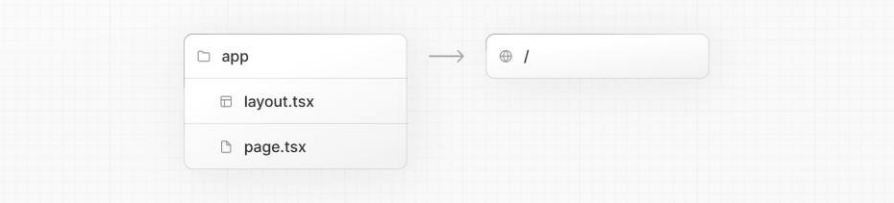
✓ **Creating directories**

Next.js uses file-system routing, which means the routes in your application are determined by how you structure your files.

The app directory

For new applications, we recommend using the App Router. This router allows you to use React's latest features and is an evolution of the Pages Router based on community feedback.

Create an app/ folder, then add a layout.tsx and page.tsx file. These will be rendered when the user visits the root of your application (/).



Create a root layout inside app/layout.tsx with the required <html> and <body> tags:
export default function RootLayout({ children,}: { children: React.ReactNode}) { return (<html lang="en"> <body>{children}</body> </html>)}
Finally, create a home page app/page.tsx with some initial content:
export default function Page() { return <h1>Hello, Next.js!</h1>}
Good to know: If you forget to create layout.tsx, Next.js will automatically create this file when running the development server with next dev.

The pages directory (optional)

If you prefer to use the Pages Router instead of the App Router, you can create a pages/ directory at the root of your project.
Then, add an index.tsx file inside your pages folder. This will be your home page (/):
export default function Page() { return <h1>Hello, Next.js!</h1>}
Next, add an _app.tsx file inside pages/ to define the global layout.
import type { AppProps } from 'next/app' export default function App({ Component, pageProps }: AppProps) { return <Component {...pageProps} />}
Finally, add a _document.tsx file inside pages/ to control the initial response from the server.
import { Html, Head, Main, NextScript } from 'next/document' export default function

Document() { return (<Html> <Head /> <body> <Main /> <NextScript /> </body> </Html>)}

Good to know: Although you can use both routers in the same project, routes in app will be prioritized over pages. We recommend using only one router in your new project to avoid confusion.

- **The public folder (optional)**

Create a public folder to store static assets such as images, fonts, etc. Files inside public directory can then be referenced by your code starting from the base URL (/).

Run the Development Server

1. Run npm run dev to start the development server.
2. Visit http://localhost:3000 to view your application.
3. Edit app/page.tsx (or pages/index.tsx) file and save it to see the updated result in your browser.

✓ **Project Creation**

Creating a Next.js project is straightforward. Here's a step-by-step guide to help you set it up:

Step 1: Install Node.js

Ensure you have Node.js installed. You can download it from nodejs.org. It's recommended to use the LTS version.

Step 2: Create a New Next.js Project

1. **Open your terminal.**
2. **Use the following command to create a new Next.js app**. You can replace my-next-app with your preferred project name.

npx create-next-app@latest my-next-app

This command uses npx to run the create-next-app package without installing it globally. It will prompt you to choose some configuration options.

3. **Change into the project directory**:

cd my-next-app

Step 3: Run the Development Server

Now, you can start the development server:

npm run dev

By default, your app will be available at http://localhost:3000. Open this URL in your web browser to see your Next.js app running.

Step 4: Project Structure

Here's a brief overview of the default project structure:

- **pages/**: Contains your application's routes. Each file represents a route.
- **public/**: Static assets like images can be placed here.
- **styles/**: CSS files can be found here.
- **package.json**: Lists your project dependencies and scripts.

Step 5: Adding TypeScript (Optional)

If you want to add TypeScript to your Next.js project:

1. **Install TypeScript and the necessary types**:

npm install --save-dev typescript @types/react @types/node

2. **Create a tsconfig.json file**:

Run the following command to create a default tsconfig.json:

npx tsc --init

3. **Rename your files**: Change your .js files in the pages directory to .tsx.
4. **Restart the development server**:

npm run dev

Next.js will automatically configure TypeScript for you, and you'll see any type errors in your code.

Step 6: Build for Production

When you're ready to deploy your app, you can build it for production:

npm run build

After building, you can start the production server with:

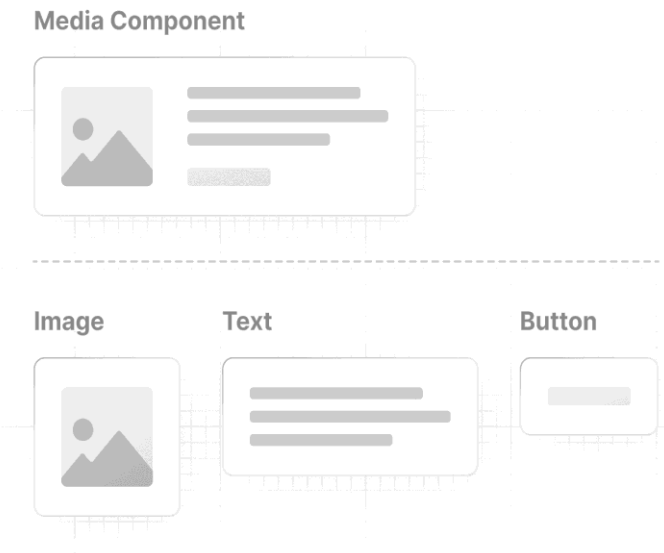
npm start

Conclusion

You now have a basic Next.js project set up! You can start adding pages, components, and styling as needed.

- **Components**

User interfaces can be broken down into smaller building blocks called **components**. Components allow you to build self-contained, reusable snippets of code. If you think of components as **LEGO bricks**, you can take these individual bricks and combine them together to form larger structures. If you need to update a piece of the UI, you can update the specific component or brick.



This modularity allows your code to be more maintainable as it grows because you can add, update, and delete components without touching the rest of our application.

The nice thing about React components is that they are just JavaScript. Let's see how you can write a React component, from a JavaScript perspective:

- ✓ **Creating components**

In React, components are **functions.** Inside your script tag, create a new function called header:

```
<script type="text/jsx">
const app = document.getElementById("app")
 function header() {
  }
const root = ReactDOM.createRoot(app);
 root.render(<h1>Develop. Preview. Ship.</h1>);
</script>
```

A component is a function that **returns UI elements**. Inside the return statement of the function, you can write JSX:

```
<script type="text/jsx">
 const app = document.getElementById("app")
 function header() {
    return (<h1>Develop. Preview. Ship.</h1>)
 }
 const root = ReactDOM.createRoot(app);
root.render(<h1>Develop. Preview. Ship.</h1>);
</script>
```

To render this component to the DOM, pass it as the first argument in the root.render() method:

```
<script type="text/jsx">
const app = document.getElementById("app")
 function header() {
   return (<h1>Develop. Preview. Ship.</h1>)
 }
const root = ReactDOM.createRoot(app);
root.render(header);
</script>
```

But, wait a second. If you try to run the code above in your browser, you'll get an error. To get this to work, there are two things you have to do:

First, React components should be capitalized to distinguish them from plain HTML and JavaScript:

```
function Header() {
return <h1>Develop. Preview. Ship.</h1>;
```

```
}
const root = ReactDOM.createRoot(app);
// Capitalize the React Component
root.render(Header);
```

Second, you use React components the same way you'd use regular HTML tags, with angle brackets <>:

```
function Header() {
 return <h1>Develop. Preview. Ship.</h1>;
}
const root = ReactDOM.createRoot(app);
root.render(<Header />);
```

If you try to run the code in your browser again, you'll see your changes.

- ✓ **Nesting components**

Applications usually include more content than a single component. You can **nest** React components inside each other like you would regular HTML elements.

In your example, create a new component called HomePage:

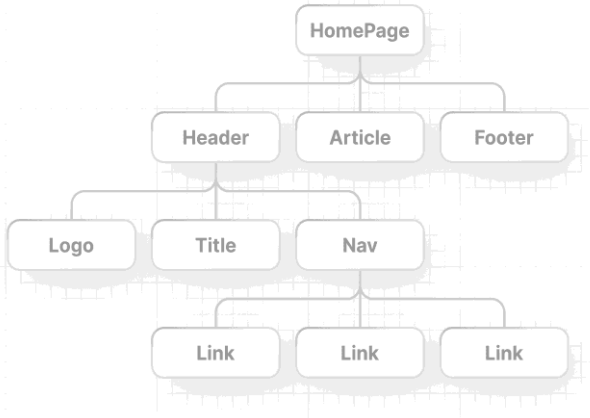
```
function Header() {
 return <h1>Develop. Preview. Ship.</h1>;
}
 function HomePage() {
 return <div></div>;
}
 const root = ReactDOM.createRoot(app);
root.render(<Header />);
```

Then nest the <Header> component inside the new <HomePage>component:

```
function Header() {
return <h1>Develop. Preview. Ship.</h1>;
}
function HomePage() {
return (    <div>     {/* Nesting the Header component */}
    <Header />
  </div>  );
}
const root = ReactDOM.createRoot(app);
root.render(<Header />);
```

- ✓ **Component trees**

You can keep nesting React components this way to form component trees.



For example, your top-level HomePage component could hold a Header, an Article, and a Footer Component. And each of those components could in turn have their own child components and so on. For example, the Header component could contain a Logo, Title and Navigation component.

This modular format allows you to reuse components in different places inside your app. In your project, since <HomePage> is now your top-level component, you can pass it to the root.render() method:

```
function Header() {
  return <h1>Develop. Preview. Ship.</h1>;
}
function HomePage() {
 return (    <div>     <Header />    </div>  );
}
const root = ReactDOM.createRoot(app);
root.render(<HomePage />);
```

- **SEO Optimization in NextJS**

Optimizing your Next.js application for search engines (SEO) is crucial for ensuring your site is discoverable and ranks well in search results. Next.js, being a React framework that enables server-side rendering, offers excellent opportunities for SEO optimization. This guide will walk you through the best practices and strategies to enhance your Next.js application's SEO.

- ✓ **Introduction to SEO in Next.js**

SEO optimization in Next.js involves configuring your application to be more accessible and indexable by search engines. This includes improving page loading times, ensuring content is correctly rendered server-side, and that your site is easily navigable by both users and search engine crawlers.

- ✓ **Initial Setup**

Before diving into the SEO strategies, ensure you have a Next.js project set up. If you're new to Next.js, you can create a new project by running:

npx create-next-app@lates

- ✓ **SEO Best Practices and Strategies**
  - **Server-Side Rendering (SSR) & Static Generation**

Next.js allows you to pre-render pages. This means the HTML is generated in advance, either at build time (Static Generation) or at request time (Server-Side Rendering), making the content immediately available to search engines.

- Use Static Generation for pages where the content doesn't change frequently. This is done using getStaticProps and getStaticPaths in Next.js.
- Use Server-Side Rendering for pages that need real-time data. Implement SSR with getServerSideProps in your page components.

- **Optimize Meta Tags**

Meta tags are crucial for SEO. They provide search engines with metadata about your webpage. Next.js's <Head> component allows you to dynamically set HTML tags in the header of your pages.

- Title Tag: Ensure each page has a unique and descriptive title.
- Meta Description: Provide a clear and concise description of the page's content.
- Open Graph and Twitter Cards: Implement social media meta tags to control how your content appears when shared.

Example:

```
import Head from 'next/head';
const HomePage = () => (
 <>
  <Head>
   <title>Your Page Title</title>
   <meta name="description" content="A short description of your page's content"/>
   <meta property="og:title" content="Your Page Title" />
   <meta property="og:description" content="A detailed description of your page's
content" />
   <meta property="og:image" content="https://example.com/thumbnail.jpg" />
   <meta name="twitter:card" content="summary_large_image" />
  </Head>
  {/* Page content */}
 </>
);
```

- **Implement Structured Data**

Structured data helps search engines understand the content and context of your pages better. Use JSON-LD format to add structured data within the <Head> component.

Example:

```
<Head>
  <script type="application/ld+json">
    {JSON.stringify({
      "@context": "http://schema.org",
      "@type": "Article",
      "headline": "Your Article Headline",
      "datePublished": "2021-01-01",
      "author": {
        "@type": "Person",
        "name": "Author Name"
      },
      // Additional structured data properties...
    })}
  </script>
</Head>
```

- **Optimize Images**

Use Next.js's built-in Image component to automatically optimize images for speed and performance. The Image component ensures images are lazy-loaded and formats like WebP are used when supported.

Example:

```
import Image from 'next/image';
const MyImage = () => (
  <Image
    src="/path/to/image.jpg"
    alt="Descriptive alt text"
    width={500}
    height={300}
  />
);
```

- **Improve Page Speed**

Page loading speed is a significant factor for SEO. Utilize Next.js features like automatic code splitting, and analyze your bundle size to remove unnecessary packages. Tools like Google's PageSpeed Insights can help identify areas for improvement.

- **Create a Sitemap and Robots.txt**

Sitemaps help search engines discover your pages. Use next-sitemap library to generate a sitemap automatically.

A robots.txt file tells search engine crawlers which pages or files they can or can't request from your site. It's important for controlling the traffic of web crawlers.

Conclusion

Optimizing your Next.js application for SEO requires a multifaceted approach, focusing on rendering strategies, meta tags, structured data, image optimization, page speed, and more. By implementing these best practices and strategies, you can significantly improve your site's visibility and ranking in search engine results.

- **Styling nextJS**

Next.js has become a powerful framework for building fast and scalable web applications. When it comes to styling your Next.js applications, you have various options at your disposal.

This blog will delve into the intricacies of Next.js styling, covering everything from CSS Modules to Tailwind, and ensuring you get the best out of your Next.js projects.

- ✓ **Setting Up Your Project**

Before diving into styling, you need to initialize your project.

1npx create-next-app@latest

This command will create a new Next.js project structure, setting up the basic configuration and file structure.

- ✓ **Installing Dependencies**

Next.js works seamlessly with many CSS frameworks and tools. To install the necessary packages, run the following npm commands:

1npm install --save-dev tailwindcss postcss autoprefixer

2npx tailwindcss init -p
This install process sets up Tailwind CSS, which is a utility-first CSS framework.
Adding Global CSS
For global styles, Next.js supports the use of a global CSS file. You can include a global css file by importing it into your **_app.js** or **_app.tsx** file.

```
// pages/_app.js
import '../styles/globals.css'
function MyApp({ Component, pageProps }) {
 return <Component {...pageProps} />
}
8export default MyApp
```

This setup ensures your global CSS file is applied across all pages.

- ✓ **CSS Modules**

For component-specific styling, CSS Modules offer a great way to encapsulate styles. CSS Modules allow you to scope CSS locally to the component, preventing naming conflicts.

Create a CSS module file and import it into your component:

```
/* styles/Home.module.css */
.container {
  padding: 0 2rem;
}
// pages/index.js
import styles from '../styles/Home.module.css'
export default function Home() {
return (
<div className={styles.container}>
<h1>Welcome to Next.js!</h1>
 </div>
) }
```

Using CSS Modules, the styles are scoped to the component avoiding any conflicts.

- ✓ **Tailwind CSS**

Integrating Tailwind CSS with Next.js is straightforward and enhances the styling process. Tailwind provides utility classes to style your components efficiently.

- **Configure Tailwind**

Ensure your tailwind config file is set up correctly:

```
 // tailwind.config.js
 module.exports = {
content: [
 './pages/**/*.{js,ts,jsx,tsx}',
 './components/**/*.{js,ts,jsx,tsx}',
 ],
 theme: {
extend: {},
},
plugins: [],
}
```

This configuration ensures Tailwind scans all your js, ts, jsx, tsx files for class names.

- **Using Tailwind**

You can now add Tailwind classes directly in your JSX:

```
// pages/index.js
export default function Home() {
 return (
 <div className="min-h-screen bg-gray-100 flex items-center justify-center">
   <h1 className="text-4xl font-bold">Hello, Next.js with Tailwind!</h1>
  </div>
)
```

}

The utility-first approach of Tailwind makes it easy to create responsive designs without writing custom CSS.

- ✓ **Handling CSS in JS**

Next.js supports various CSS-in-JS solutions, including styled-components and emotion. These libraries allow you to write CSS directly in your JavaScript or TypeScript files.

Styled-Components Example

First, install styled-components:

```
1npm install styled-components

2npm install --save-dev babel-plugin-styled-components
```

Next, configure Babel to use the styled-components plugin:

```
// .babelrc

{

  "presets": ["next/babel"],

  "plugins": [["styled-components", { "ssr": true }]]

}
```

You can now use styled-components in your Next.js components:

```
import styled from 'styled-components';

const Container = styled.div`

padding: 2rem;

background-color: #f0f0f0;

`;

export default function Home() {

return (

 <Container>

<h1>Styled Components in Next.js</h1>

</Container>

)  }
```

This approach allows you to create and manage component-level styles effectively.

Emotion Example

Emotion is another popular CSS-in-JS library that works well with Next.js. To get started, install the necessary packages:

```
1npm install @emotion/react @emotion/styled
```

You can now use Emotion to style your components:

```
/** @jsxImportSource @emotion/react */
import { css } from '@emotion/react';
import styled from '@emotion/styled';
const containerStyle = css`
padding: 2rem;
background-color: #f0f0f0;
const Heading = styled.h1`;
 color: #333;
`;
export default function Home() {
 return (
<div css={containerStyle}>
 <Heading>Emotion in Next.js</Heading>
</div>
)
}
```

- ✓ **Adding Custom Fonts**

Adding custom fonts in Next.js can enhance the look and feel of your application. You can include web fonts by updating your global CSS or using the **next/font** package.

- **Including Fonts in Global CSS**

To include custom fonts, update your global CSS:

```
/* styles/globals.css */
@import
url('https://fonts.googleapis.com/css2?family=Roboto:wght@400;700&display=swap');
body {
```

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

- **Using next/font**

Alternatively, you can use the **next/font** package for a more efficient way to load fonts:

```
npm install @next/font
```

Then use it in your _app.js:

```
import { Roboto } from '@next/font/google';
const roboto = Roboto({
weight: ['400', '700'],
subsets: ['latin'],
});
function MyApp({ Component, pageProps }) {
return (
 <main className={roboto.className}>
 <Component {...pageProps} />
</main>
);
 }
 export default MyApp;
```

This method ensures your fonts are optimized and loaded efficiently.

- ✓ **Additional Styling Options**
- ***Formatting with Prettier***

To ensure a consistent coding style, use prettier Next.js along with ESLint. Prettier automatically formats your code, making it easier to maintain a clean codebase.

- ***Using Sass for Enhanced Styling***

Next.js also supports Sass and Less for those who prefer these preprocessors. Additionally, you can manage fonts, global state, and themes efficiently within your Next.js applications.

To use Sass, install the necessary package:

```
npm install sass
```

You can now use .scss or .sass files in your project:

```
// styles/Home.module.scss

.container {

  padding: 2rem;

 background-color: #f0f0f0;

}
```

```
import styles from '../styles/Home.module.scss'

export default function Home() {

return (

<div className={styles.container}>

<h1>Sass in Next.js</h1>

</div>

)

}
```

Conclusion

Styling your Next.js application can be achieved through various methods, each offering unique advantages. Whether you choose CSS Modules, Tailwind, styled-components, emotion, or Sass, Next.js provides the flexibility to implement the best solution for your needs.

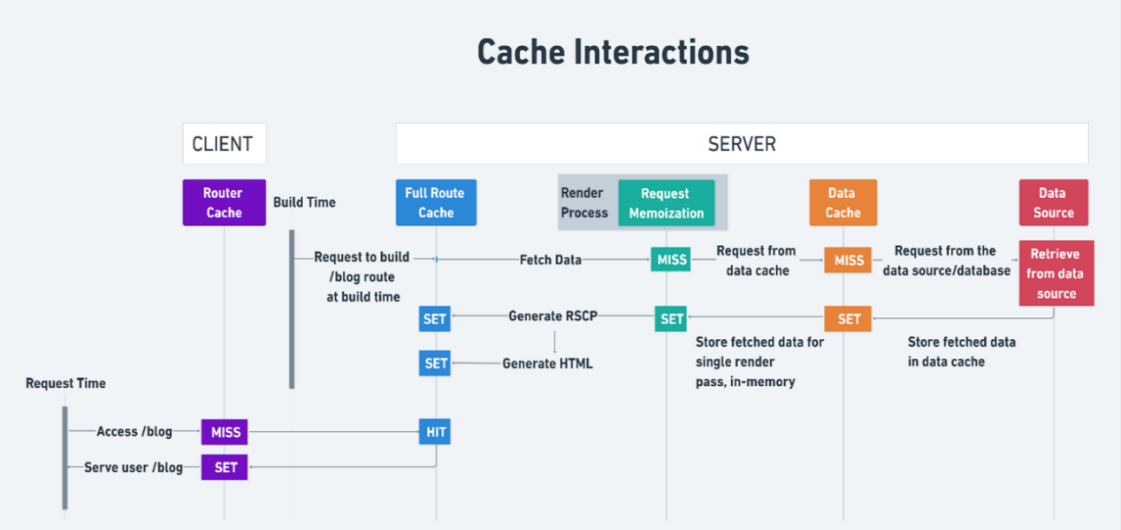
Understanding these styling techniques enhances your ability to build robust, scalable, and maintainable applications. With the knowledge gained, you can now confidently create beautiful and functional Next.js projects.

- **Caching Strategies**
  - ✓ **Caching Strategies in Next.js**

Next.js is a fantastic framework that simplifies the creation of complex server-rendered React applications. However, there's a significant challenge: its caching mechanism is highly intricate, often leading to bugs that are tough to diagnose and resolve.

Without a clear understanding of how caching works in Next.js, it can feel like a constant struggle, preventing you from fully benefiting from its powerful performance enhancements. This article aims to demystify Next.js's caching system, detailing every aspect to help you harness its potential without the frustration.

To start, here's an image showing how the different caches in Next.js interact. While it may seem overwhelming initially, by the end of this article, you will have a comprehensive understanding of each step in the process and how they work together.



In the image above, you likely noticed the terms "Build Time" and "Request Time." To prevent any confusion as we proceed, let's clarify these concepts.

Build time occurs when an application is constructed and deployed. Any content cached during this phase, primarily static content, becomes part of the build time cache. This cache is refreshed only when the application is rebuilt and redeployed.

Request time , on the other hand, refers to the moment when a user requests a page. The data cached during request time is typically dynamic, as it is fetched directly from the data source in response to user requests.

Next.js Caching Mechanisms

Grasping the intricacies of Next.js caching can initially seem overwhelming due to its four distinct caching mechanisms, each functioning at different stages of your application and interacting in seemingly complex ways.

Here are the four caching mechanisms in Next.js:

- Request Memoization
- Data Cache
- Full Route Cache
- Router Cache

In this article, I'll dive into each of these mechanisms, explaining their specific roles, where they are stored, their duration, and how to effectively manage them, including

methods for cache invalidation and opting out. By the end, you'll have a thorough understanding of how these mechanisms collaborate to enhance Next.js's performance.

Request Memoization

A common issue in React arises when the same information needs to be displayed in multiple places on a single page. The straightforward approach is to fetch the data wherever it's needed, but this isn't ideal because it results in multiple server requests for the same data. This is where Request Memoization becomes valuable.

Request Memoization is a feature in React that caches every fetch request made in a server component during the render cycle, which refers to the process of rendering all the components on a page. If a fetch request is made in one component and the same request is made in another component, the second request won't reach the server. Instead, it will retrieve the cached value from the first request.

```
export default async function getUserData(userId) {
  // Next.js automatically caches the 'fetch' function
  const response = await fetch('https://api.example.com/users/userId');
  return response.json();
}
export default async function Page({ params }) {
  const userData = await getUserData(params.id);
  return (
<>
<h1>{userData.name}</h1>
<UserDetails userId={params.id} />
</>
  );
}
async function UserDetails({ userId }) {
  const userData = await getUserData(userId);
  return <p>{userData.name}</p>;
}
```

In the code above, we have two components: Page and UserDetails. The initial call to the getUserData function in Page makes a standard fetch request, and the response from this request is stored in the Request Memoization cache. When getUserData is called again in the UserDetails component, it doesn't make a new fetch request. Instead, it uses the memorized value from the first request. This optimization significantly improves the performance of your application by reducing the number of server requests, and it also simplifies the component code since you don't need to manually optimize your fetch requests.

It's important to understand that this cache is entirely server-side, meaning it only caches fetch requests made from your server components. Additionally, the cache is cleared at the start of each request, so it is only valid for a single render cycle. This is intentional, as the cache's purpose is to eliminate duplicate fetch requests within a single render cycle.

Finally, note that this cache only stores fetch requests made with the GET method. To be memorized, a fetch request must also have the exact same parameters (URL and options).

- **Caching Non-fetch Requests**

React, by default, only caches fetch requests. However, there are scenarios where you might need to cache other types of requests, such as database queries. To achieve this, you can use React's cache function. Simply pass the function you want to cache to cache, and it will return a memorized version of that function.

```
import { cache } from "react";
import { queryDatabase } from "./databaseClient";


export const getUserData = cache(async (userId) => {
  // Perform a direct database query
  return queryDatabase("SELECT * FROM users WHERE id = ?", [userId]);
});
```

In the code above, the first time getUserData() is called, it directly queries the database since there is no cached result yet. However, the next time this function is called with the same userId, the data is retrieved from the cache. Similar to fetch memoization, this caching is only valid for the duration of a single render pass and operates identically to fetch memoization.

- **Revalidation**

Revalidation involves clearing the cache and updating it with fresh data. This is crucial

because a cache that is never updated will eventually become stale and outdated. Fortunately, with Request Memoization, we don't have to worry about revalidation since the cache is only valid for the duration of a single request.

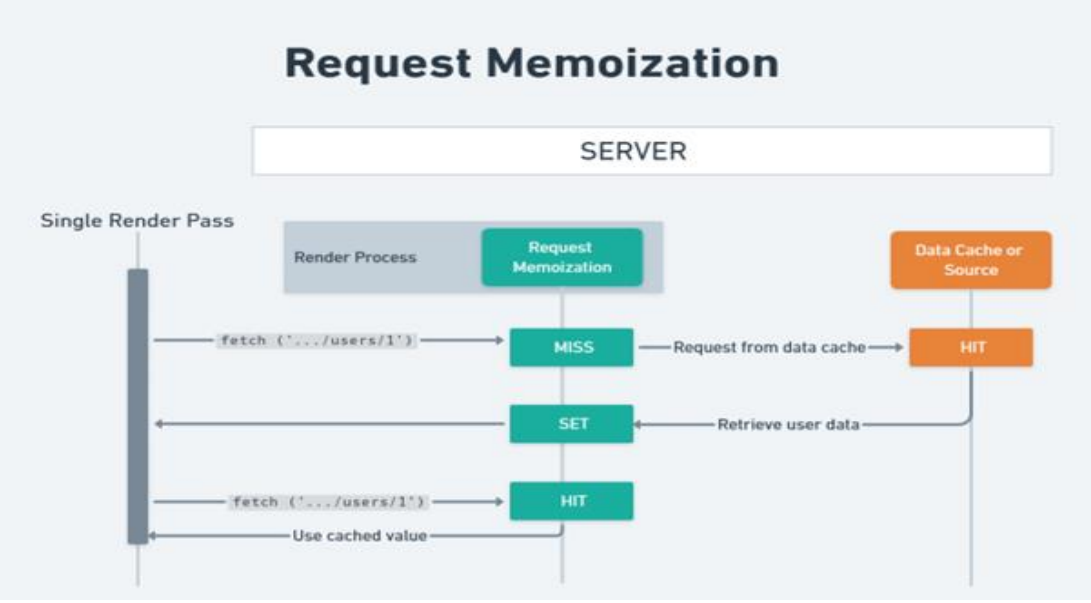
Opting Out

To opt out of this cache, you can pass an AbortController signal as a parameter to the fetch request.

```
async function getUserData(userId) {
  const { signal } = new AbortController();
  const response = await fetch('https://api.example.com/users/userId', {
    signal,
  });
  return response.json();
}
```

Avoiding caching with an AbortController signal instructs React not to store the fetch request in the Request Memoization cache. However, it's advisable to use this approach judiciously, as the cache can significantly enhance your application's performance.

The diagram below offers a visual overview of Request Memoization's functionality.



Data Cache

While Request Memoization excels at preventing duplicate fetch requests within a single render cycle, it falls short when it comes to caching data across requests or

users. This is where the Data Cache shines. It serves as the final cache that Next.js consults before actually fetching data from an API or database, and it persists across multiple requests and users.

Consider a scenario where we have a basic page that retrieves guide data for a specific city from an API.

```
import { cache } from "react";
export default async function Page({ params }) {
  const city = params.city;
  const guideData = await getGuideData(city);
  return (
<div>
<h1>{guideData.title}</h1>
<p>{guideData.content}</p>
    {/* Render the guide data */}
</div>
  );
}
export const getGuideData = cache(async (city) => {
  const response = await fetch('https://api.globetrotter.com/guides/city');
  return response.json();
});
```

- **Duration and Revalidation in the Data Cache**

The guide data, in this case, is unlikely to change frequently. Fetching it fresh every time someone needs it is inefficient. Instead, caching this data across all requests would ensure it loads instantly for future users. Fortunately, Next.js handles this automatically for us with the Data Cache.

By default, every fetch request in your server components is cached in the Data Cache, which is stored on the server. This cached data is then used for all future requests. For example, if you have 100 users all requesting the same data, Next.js will only make one fetch request to your API and then use that cached data for all 100 users. This significantly boosts performance.

Duration: Data in the Data Cache differs from the Request Memoization cache in

that it is never cleared unless explicitly instructed by Next.js. This data persists even across deployments, meaning that if you deploy a new version of your application, the Data Cache remains intact.

Revalidation: Since the Data Cache is never cleared by Next.js, we need a way to opt into revalidation, which is the process of removing data from the cache. Next.js offers two ways to achieve this: time-based revalidation and on-demand revalidation.

Time-based Revalidation: The simplest way to revalidate the Data Cache is to automatically clear the cache after a set period. This can be done in two ways.

```
const res = fetch('https://api.globetrotter.com/guides/city', {
  next: { revalidate: 3600 },
});
```

The first method involves using the next.revalidate option in your fetch request. This option specifies how many seconds Next.js should keep your data in the cache before considering it stale. In the example above, we are instructing Next.js to revalidate the cache every hour.

Another approach is to set a revalidation time using the revalidate segment in the configuration options.

```
export const revalidate = 3600;
export default async function Page({ params }) {
  const city = params.city;
  const res = await fetch('https://api.globetrotter.com/guides/city');
  const guideData = await res.json();
  return (
<div>
<h1>{guideData.title}</h1>
<p>{guideData.content}</p>
    {/* Render the guide data */}
</div>
  );
}
```

- **Understanding Time-based Revalidation**

When implementing time-based revalidation, all fetch requests for a page will be revalidated every hour, unless they have their own more specific revalidation time set.

It's crucial to understand how time-based revalidation handles stale data. Here's the process:

Initial Fetch: The first fetch request retrieves the data and stores it in the cache.

Cached Data Usage: Subsequent fetch requests made within the 1-hour revalidation time use the cached data without making additional fetch requests.

Revalidation After 1 Hour: After 1 hour, the first fetch request will still return the cached data, but it will also execute a fetch request to get the newly updated data and store it in the cache. From this point on, new fetch requests will use the newly cached data.

Stale-while-Revalidate: This pattern, where stale data is used while new data is fetched in the background, is called stale-while-revalidate and is the behaviour that Next.js employs.

- **On-demand Revalidation**

For data that is not updated regularly, on-demand revalidation can be used to revalidate the cache only when new data is available. This approach is suitable for scenarios where you want to invalidate the cache and fetch new data only when a new article is published, or a specific event occurs.

On-demand revalidation can be implemented in one of two ways.

```
import { revalidatePath } from "next/cache";

export async function publishArticle({ city }) {
  createArticle(city);
  revalidatePath('/guides/city');
}
```

For more specific revalidation, you can use the revalidateTag function, which clears the cache for all fetch requests with a specific tag

```
import { revalidateTag } from "next/cache"

export async function publishArticle({ city }) {
  createArticle(city);
  revalidateTag("city-guides");
}
```

Opting Out of the Data Cache

To opt out of the Data Cache, you can use the cache: "no-store" option in your fetch request, ensuring the request is not cached:

```
const res = fetch('https://api.globetrotter.com/guides/city', {
  cache: "no-store",
});
```

Alternatively, you can use the unstable_noStore function to opt out of the Data Cache for a specific scope:

```
import { unstable_noStore as noStore } from "next/cache";
function getGuide() {
  noStore();
  const res = fetch('https://api.globetrotter.com/guides/city');
}
```

For broader opt-out, you can use the dynamic segment config option at the top level of your file to force the page to be dynamic and opt out of the Data Cache entirely:

```
export const dynamic = "force-dynamic";
export const revalidate = 0;
```

This sets the entire page to be dynamic, ensuring nothing is cached.

✓ Caching Non-fetch Requests

Up to now, we've focused on caching fetch requests with the Data Cache, but we can do much more with it.

Let's revisit our example of city guides. In some cases, we may want to retrieve data directly from our database. To achieve this, we can utilize the cache function provided by Next.js. This function is similar to the React cache function but applies to the Data Cache instead of Request Memoization.

```
import { getGuides } from "./data";
import { cache as unstable_cache } from "next/cache";
const getCachedGuides = unstable_cache(city => getGuides(city), ["guides-cache-key"]);
export default async function Page({ params }) {
  const guides = await getCachedGuides(params.city);
```

```
  // ...
}
```

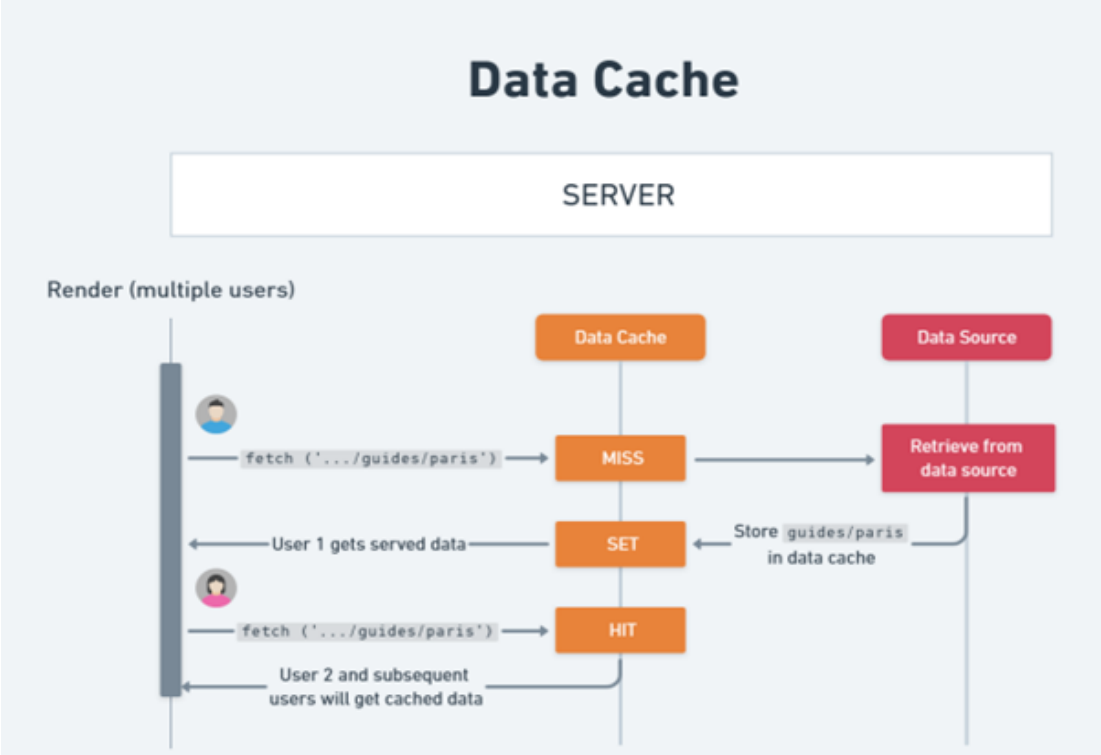
This feature is currently experimental, indicated by the prefix unstable_, but it is the sole method available for caching non-fetch requests in the Data Cache.

The cache function in the provided code snippet is concise but may be perplexing for those encountering it for the first time.

This function takes three parameters, but only two are mandatory. The first parameter is the function to be cached, which in this case is getGuides. The second parameter is the cache key, necessary for Next.js to distinguish between different caches. The key is an array of strings and must be unique for each cache. If two cache functions have the same key array, they are considered identical requests and stored in the same cache (similar to fetch requests with the same URL and params).

The third parameter is optional and allows you to specify options such as revalidation time and tags.

In this specific code, we are caching the results of the getGuides function and storing them in the cache with the key ["guides-cache-key"]. Consequently, if getCachedGuides is called twice with the same city, the second call will use the cached data instead of invoking getGuides again.



The third type of cache in Next.js is the Full Route Cache, which is relatively straightforward compared to the Data Cache. This cache is particularly useful because it allows Next.js to cache static pages at build time, eliminating the need to build those pages for each request.

In Next.js, the pages rendered to clients consist of HTML and a component called the React Server Component Payload (RSCP). This payload contains instructions for how the client components should interact with the rendered server components to render the page. The Full Route Cache stores both the HTML and RSCP for static pages at build time.

Now, let's illustrate this with an example.

```
import Link from "next/link"
 async function getBlogList() {
  const blogPosts = await fetch("https://api.example.com/posts")
  return await blogPosts.json()
 }
 export default async function Page() {
  const blogData = await getBlogList()
  return (
 <div>
 <h1>Blog Posts</h1>
 <ul>
       {blogData.map(post => (
 <li key={post.slug}>
 <Link href={'/blog/post.slug'}>
 <a>{post.title}</a>
 </Link>
 <p>{post.excerpt}</p>
 </li>
       ))}
 </ul>
 </div>
```

```
  )
 }
```

In the provided code, the Page component will be cached at build time because it does not contain any dynamic data. Specifically, its HTML and React Server Component Payload (RSCP) will be stored in the Full Route Cache, enabling faster serving when a user requests access. The only way this cached HTML/RSCP will be updated is by redeploying our application or manually invalidating the data cache that this page depends on.

Despite the fetch request in the code, the data fetched is cached by Next.js in the Data Cache. Therefore, this page is actually considered static. Dynamic data, on the other hand, refers to data that changes on every request to a page, such as dynamic URL parameters, cookies, headers, search parameters, etc.

Similar to the Data Cache, the Full Route Cache is stored on the server and persists across different requests and users. However, unlike the Data Cache, the Full Route Cache is cleared every time you redeploy your application.

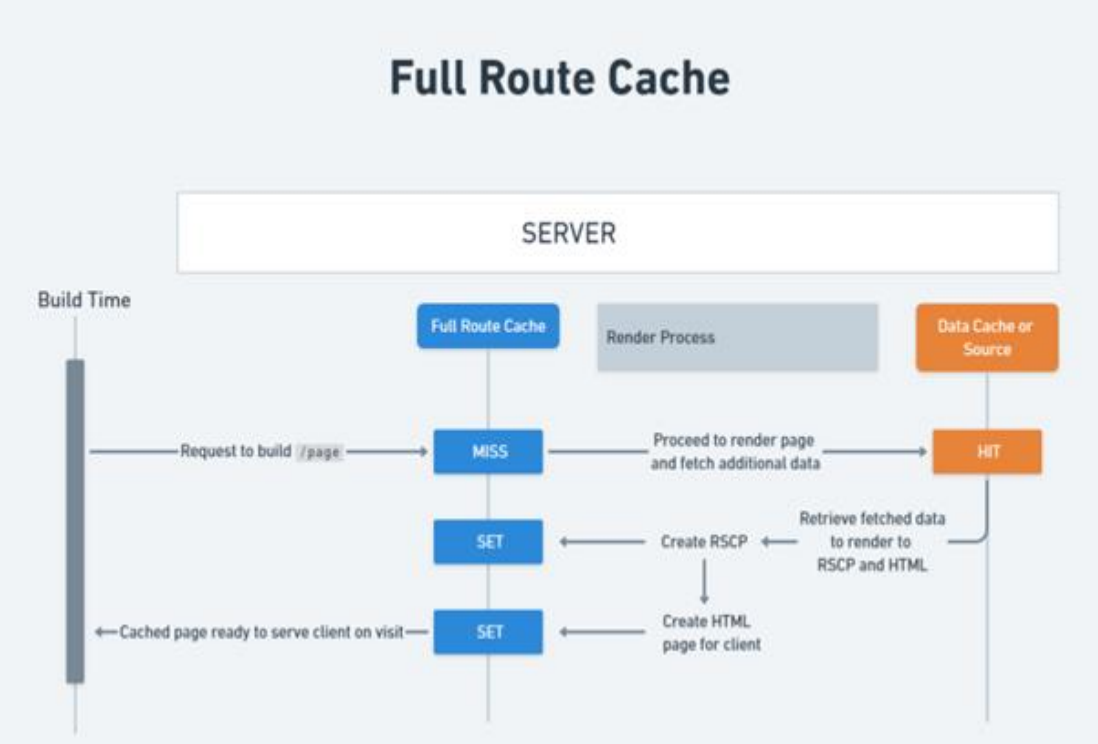
- ✓ **Opting Out of the Full Route Cache**

There are two ways to opt out of the Full Route Cache:

Opting Out of the Data Cache: If the data you are fetching for the page is not cached in the Data Cache, then the Full Route Cache will not be used.

Using Dynamic Data: Incorporating dynamic data in your page, such as headers, cookies, or searchParams dynamic functions, and dynamic URL parameters (e.g., id in /blog/[id]), will also prevent the Full Route Cache from being utilized.

The diagram below provides a step-by-step illustration of how the Full Route Cache functions.



Router Cache

The Router Cache is a unique cache that differs from the others in that it is stored on the client side instead of the server side. However, it can be a source of confusion and potential bugs if not properly understood. This cache stores routes that a user visits, so when they revisit those routes, the cached version is used instead of making a request to the server. While this can significantly improve page loading speeds, it can also lead to unexpected behavior, as outlined below.

Please note that this cache is only active in production builds. In development, all pages are rendered dynamically, so they are never stored in this cache.

```
export default async function Page() {
  const blogData = await getBlogList()
  return (
<div>
<h1>Blog Posts</h1>
<ul>
      {blogData.map(post => (
<li key={post.slug}>
```

```
<Link href={'/blog/post.slug'}>
<a>{post.title}</a>
</Link>
<p>{post.excerpt}</p>
</li>
      ))}
</ul>
</div>
 )
}
```

- **Router Cache**

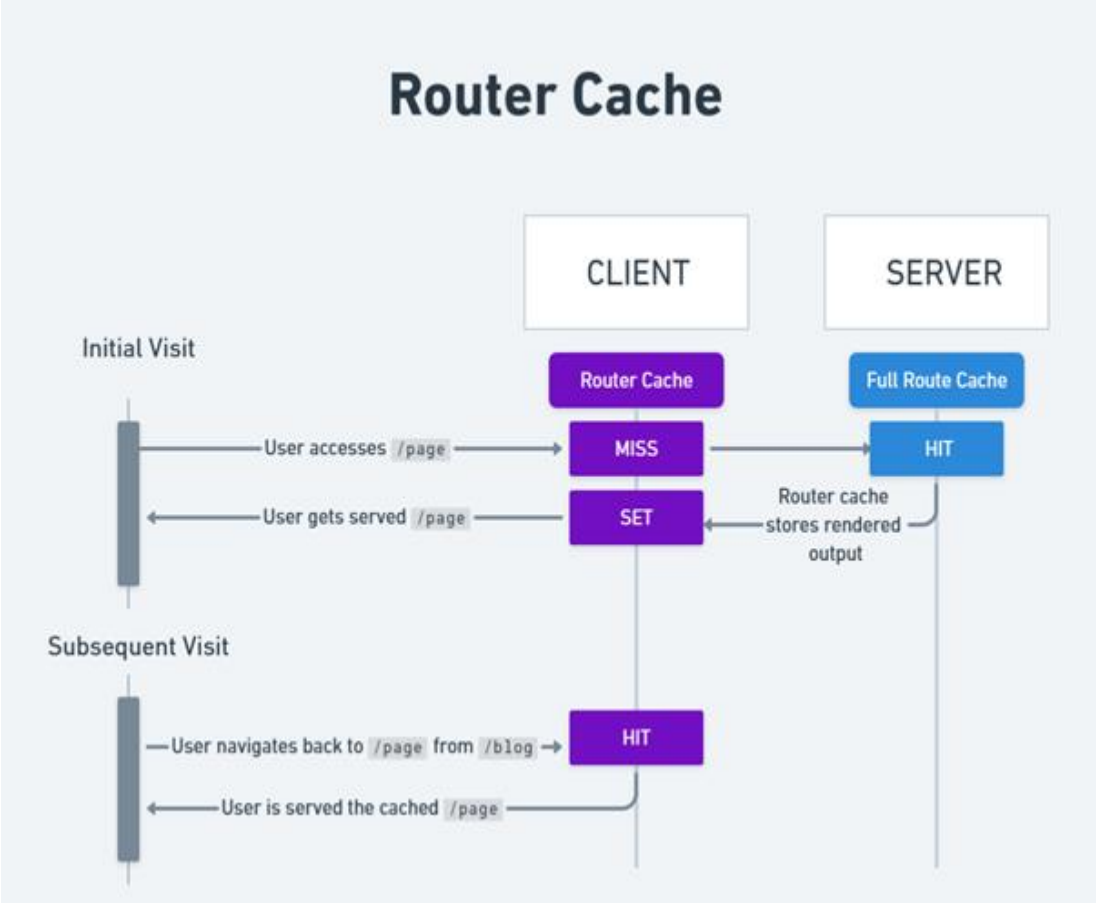
In the provided code snippet, when a user navigates to the page or any of the /blog/${post.slug} routes, the HTML/RSCP is stored in the Router Cache. Subsequently, if the user revisits a page they have previously viewed, the Router Cache is used to retrieve the cached HTML/RSCP instead of making a request to the server.

Duration: The Router Cache duration depends on the type of route. For static routes, the cache is stored for 5 minutes, while for dynamic routes, it is stored for only 30 seconds. If a user revisits a static route within 5 minutes or a dynamic route within 30 seconds, the cached version is used. Otherwise, a request to the server is made for the new HTML/RSCP. Additionally, the cache is only stored for the user's current session and is cleared if the user closes the tab or refreshes the page.

Revalidation: Revalidating the Router Cache can be done on demand, similar to the Data Cache, by using revalidatePath or revalidateTag. This action also revalidates the Data Cache.

Opting Out: There is no direct way to opt out of the Router Cache. However, with the various methods available for revalidation, opting out is not a significant concern.

The Router Cache provides a notable advantage in page loading speeds but can lead to unexpected behavior if not understood correctly.



Conclusion

Understanding the intricacies of Next.js caching, with its various mechanisms and interactions, can be challenging. However, this article aimed to shed light on how these caches operate and interact. While the official documentation suggests that knowledge of caching is not essential for productive use of Next.js, understanding its behavior can greatly assist in configuring settings that best suit your application.

By grasping the workings of Request Memoization, Data Cache, Full Route Cache, and Router Cache, you gain a deeper insight into optimizing your Next.js application for improved performance. Each cache plays a crucial role, and knowing how to leverage them effectively can enhance the user experience and streamline your development process.

Points to Remember

Steps used to prepare a Next.js project environment

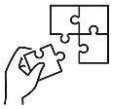
1. Install Node.js and npm

2. Set Up a New Next.js Project
3. Navigate to Your Project Directory
4. Run the Development Server
5. Explore the Project Structure
6. Install Additional Packages
7. Using TypeScript (Optional)
8. Environment Variables (Optional)
9. Configure ESLint (Optional)
10. Build and Production

Steps used to create NextJS Project

1. Install Node.js (if not already installed)
2. Create a New Next.js Project
3. Navigate to the Project Directory
4. Run the Development Server
5. Explore Your Next.js App

- Initialize NextJS project Development
    1. Creating Pages and components
    2. Implementing search engine optimization (SEO)
    3. Styling
    4. Caching Strategies



Application of learning 3.2.

ABC Web Development Company is a software development company located in Kacyiru- Kigali City. Assume you have been hired as Next.js developer and tasked with creating a portfolio website for a client by preparing the Next.js environment, installing Node.js and the Next.js framework. You have to also create a new Next.js project using the command line. After project creation, initialize the development environment with essential dependencies and configuration settings. Finally, apply CSS for styling the website, ensuring a responsive design that enhances user experience across various devices.

Indicative content 3.3: Implementing Rendering Techniques

Duration: 2 hrs

Theoretical Activity 3.3.1: Description of Rendering Techniques

Tasks:

1. You are requested to answer the following questions related to Rendering Techniques

   i. What is Static Site Generation (SSG), and how does it work?
   ii. How does SSG handle dynamic content updates?
   iii. What is Server-Side Rendering (SSR), and how does it differ from SSG?
   iv. What are the advantages of SSR for user experience and SEO?
   v. What are the performance implications of using ISR?
   vi. How does CSR impact initial load time and user experience?

2. Provide the answer for the asked questions and write them on papers.
3. Present the findings/answers to the whole class
4. For more clarification, read the key readings 3.3.1. In addition, ask questions where necessary.

Key readings 3.3.1.: Description of Rendering Techniques

- **Rendering techniques**

The way web pages are rendered has evolved significantly. This article delves into four prominent rendering techniques, showcasing how each is implemented using examples in the NextJS framework.

Rendering — Converting the framework level code to a browser-ready HTML representation for our UI. This can happen on the client-side(browser) or server side (pre-rendered).

Build time — Also known as **compile time** or **ahead of time**, refers to the process of generating static HTML, CSS, and JavaScript files for our application which gets served directly by a web server or content delivery network (CDN).

Run time — Occurs when a **user requests** and interacts with our application in their web browser.

- ✓ **Client Side Rendering (CSR)**

With CSR, the empty template is generated at build time and the rest of the content(fetched inside useEffect) during the request at run time. Also, this is the only technique, out of four, where rendering happens on the client side.

Initially, the server sends an empty template to the client, which then populates the content using JavaScript shifting the rendering process to the browser. Client Side Rendering (CSR) is a common approach used in React applications — those built without NextJS, where usually the content of a page is fetched inside useEffect hook and then rendered .

Here's what happens with CSR rendering step-by-step:

1. User enters a URL or clicks a link to navigate to a CSR-powered page.
2. The client (browser) sends a request to the server.
3. The server responds by sending an empty template (HTML, JavaScript, CSS).
4. The client's browser loads the JavaScript bundle and renders the empty template.
5. The JavaScript in the bundle fetches additional data from APIs using client-side JavaScript (e.g., fetch or AJAX).
6. The fetched data is used to populate the content of the page.
7. The client's browser renders the final content on the page.

Let's say we are building an e-commerce website with a product listing that needs to be dynamically loaded on the client side. We could do something like this for that;

```
import { useState, useEffect } from 'react';
const ProductsPage = () => {
 const [products, setProducts] = useState([]);
 useEffect(() => {
  // Fetch products data from an API endpoint
  // and this happens in client-side at run time
  fetch('/api/products')
   .then((response) => response.json())
   .then((data) => setProducts(data));
 }, []);
 return (
  <div>
   <h1>Products</h1>
   <ul>
    {products.map((product) => (
     <li key={product.id}>{product.name}</li>
    ))}
   </ul>
  </div>
```

```
  );
};
export default ProductsPage;
```

Pros:

- Enables dynamic and interactive user experiences.
- Once initial assets are loaded, subsequent interactions are quick.
- Well-suited for apps with complex UI and user interactions.

Cons:

- Requires downloading and executing JavaScript before rendering.
- Search engines may have difficulty indexing JavaScript-rendered content.
- May lead to "blank" pages during initial load.

✓ **Server Side Rendering (SSR)**

With SSR, generating HTML as well as fetching of external data, later required to populate the HTML template, is done in the server at run time.

Server Side Rendering dynamically generates HTML content on the server for each user request. This technique ensures up-to-date content and personalization. In NextJS, getServerSideProps fetches data on the server side, delivering a well-rounded balance between dynamic content and performance.

1. User enters a URL or clicks a link to navigate to an SSR-powered page.
2. The client sends a request to the server.
3. The server dynamically generates the HTML content for the requested page.
4. Server-side code (e.g., getServerSideProps) fetches data from external data sources.
5. The HTML content, along with the fetched data, is sent as a response to the client.
6. The client's browser receives the complete HTML and renders the page.

Consider a social media platform where users have personalized profiles. With SSR, you can ensure that each user's profile is dynamically rendered on the server side.

```
const ProfilePage = ({ user }) => {
  return (
    <div>
      <h1>{user.username}</h1>
      <p>{user.bio}</p>
    </div>
  );
};
export async function getServerSideProps({ params }) {
  // Fetch user data based on the username
  // and this happens on server side at run time
  const user = await fetchUserDataByUsername(params.username);
```

```
  return {
    props: {
      user,
    },
  };
}
export default ProfilePage;
```

Pros:

- Pre-renders content on the server, leading to faster load times.
- Search engines can index fully-rendered content.
- Supports dynamic content while providing good performance.

Cons:

- Requires server-side processing for each user request.
- May result in higher time-to-interact for users.
- More intricate implementation compared to SSG.

✓ **Static Site Generation (SSG)**

With SSG, everything required for UI is generated at build time, when the application is deployed.

Static Site Generation, the foundational rendering method, involves pre-generating static HTML pages during the build time. With NextJS, getStaticProps fetches data at build time resulting in fast and efficient rendering.

Rendering process in SSG occurs in following steps:

1. Developer initiates the build process using NextJS commands.
2. During the build, NextJS pre-generates static HTML pages for all specified routes.
3. These static HTML pages include the content fetched from data sources (e.g., APIs, databases) using the getStaticProps function.
4. When a user requests a page, the pre-rendered static HTML page is served by the server.
5. The user's browser receives the HTML and renders the page immediately.

To put this in use case, imagine you are building a blog website where you regularly publish articles. With SSG, you can pre-generate static HTML pages for each article at build time.

```
import { useRouter } from 'next/router';
const ArticlePage = ({ article }) => {
  return (
    <div>
      <h1>{article.title}</h1>
      <p>{article.content}</p>
    </div>
  );
```

```
};
export async function getStaticPaths() {
  const articleSlugs = await fetchArticleSlugs();
  const paths = articleSlugs.map((slug) => ({ params: { slug } }));
  return {
    paths,
    fallback: false, // Generate 404 for unmatched paths
  };
}
// If our page utilizes getStaticProps, the page will be
// generated during the build time including the content
// fetched inside the function
export async function getStaticProps({ params }) {
  const article = await fetchArticleBySlug(params.slug);
  return {
    props: {
      article,
    },
  };
}
export default ArticlePage;
```

Pros:

- Pre-generated static HTML leads to quick load times.
- Search engines easily index static pages with content.
- Hosting static files requires less server processing.
- Well-suited for blogs, documentation, and news sites.

Cons:

- Less suitable for highly dynamic or real-time content.
- May require rebuilding the site to update content.

✓ **Incremental Site Regeneration (ISR)**

With ISR, page generation happens at build time.However, there will be regeneration of the page at run time too depending on the upadated page content andrevalidate value. Incremental Site Regeneration extends the benefits of Static Site Generation. It pre-generates static pages while allowing periodic re-validation and regeneration. This technique is ideal for scenarios where content needs to be relatively fresh without sacrificing performance. NextJS supports ISR through the revalidate option.

1. Developer initiates the build process using NextJScommands.
2. During the build, NextJS pre-generates static HTML pages for specified routes.

3. The pre-generated pages include content fetched from data sources using the getStaticProps function.
4. When a user requests a page, the pre-rendered static HTML page is served by the server.
5. In the background, NextJS periodically (based on the specified revalidation interval) revalidates the data using the getStaticProps function.
6. If new data is available, NextJS regenerates the page with fresh content.
7. The regenerated page is saved and served for subsequent requests.

Suppose you are building a news website where articles are updated frequently. With ISR, you can ensure that the articles are revalidated and regenerated at specified intervals.

```
const NewsArticlePage = ({ article }) => {
  return (
    <div>
      <h1>{article.title}</h1>
      <p>{article.content}</p>
    </div>
  );
};
export async function getStaticPaths() {
  const articleSlugs = await fetchNewsArticleSlugs();
  const paths = articleSlugs.map((slug) => ({ params: { slug } }));
  return {
    paths,
    fallback: true, // Generate static page on the first request
  };
}
export async function getStaticProps({ params }) {
  // Fetch news article data based on the slug at build time
  const article = await fetchNewsArticleBySlug(params.slug);
  return {
    props: {
      article,
    },
    revalidate: 3600, // Revalidate and regenerate page every 1 hour
                      // which happens at run time
  };
}
export default NewsArticlePage;
```

Pros:

- Combines benefits of SSG and real-time updates.
- Allows periodic updates while maintaining good performance.
- Search engines can index pre-rendered content.

Cons:

- First load may still experience the limitations of SSG.
- Requires setting up revalidation intervals and handling stale content.
- Limited to scheduled revalidation intervals.

Conclusion

Understanding web page rendering techniques is crucial for creating great web experiences. NextJS empowers developers with the flexibility to choose the technique that aligns with their project's needs. Whether it's the static efficiency of SSG, the interactive nature of CSR, the dynamic capabilities of SSR, or the balanced approach of ISR, these techniques enable developers to create web applications that excel in performance, content delivery, and user engagement.



Practical Activity 3.3.2: Implementing Rendering Techniques



Task:

1: You are requested to go to the computer lab to Implement Rendering Techniques in NextJS Framework.
2: Read key reading 3.3.2
3: Apply safety precautions.
4: Implement rendering techniques.
5: Present your work to the trainer and whole class.
6: Perform the task provided in application of learning 3.3



Key readings 3.3.2:Implementing rendering techniques

To implement the various **React rendering techniques**, you'll need to work with different approaches based on your application's requirements, such as whether it's a static site, dynamic, or needs fast interaction. Below are implementations of the common rendering techniques:

- ✓ **Client-Side Rendering (CSR)**

This is the default way React applications work. Everything is rendered in the client's browser after JavaScript is loaded.

Steps:

1. Use **Create React App** to set up your project.
2. Components are rendered directly on the browser.

```
npx create-react-app my-csr-app

cd my-csr-app

npm start
```

Example: Rendering in the Client

```
// App.js

import React from 'react';

function App() {

  return (

    <div>

      <h1>Welcome to Client-Side Rendering</h1>

      <p>This content is rendered in the browser.</p>

    </div>

  );

}

export default App;

// index.js

import React from 'react';

import ReactDOM from 'react-dom';

import App from './App';

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

- ✓ **Server-Side Rendering (SSR) with Next.js**

Server-Side Rendering (SSR) pre-renders the app on the server and sends HTML to the browser.

Steps:

1. Install **Next.js** for SSR capabilities.

```
npx create-next-app@latest my-ssr-app

cd my-ssr-app

npm run dev
```

Example: Implementing SSR in Next.js

```
// pages/index.js

import React from 'react';

const HomePage = ({ data }) => {
  return (
    <div>
      <h1>Server-Side Rendered Page</h1>
      <p>Data from the server: {data}</p>
    </div>
  );
};

// This function runs on the server for every request

export async function getServerSideProps() {
  const data = 'This data was fetched from the server at request time';
  return {
    props: { data }, // will be passed to the page component as props
  };
}

export default HomePage;
```

- **getServerSideProps** is a special function in Next.js that allows data fetching on the server during runtime, ensuring SEO-friendliness and faster initial load times.

✓ **Static Site Generation (SSG) with Next.js**

Static Site Generation (SSG) pre-renders pages at **build time**, and the HTML is served as static files.

Steps:

1. Use **Next.js** again, but with **Static Site Generation**.

Example: Implementing SSG in Next.js

```
// pages/index.js

import React from 'react';

const HomePage = ({ data }) => {

  return (

    <div>

      <h1>Static Site Generated Page</h1>

      <p>Static data: {data}</p>

    </div>

  );

};

// This function runs at build time

export async function getStaticProps() {

  const data = 'This static content was generated at build time';

  return {

    props: { data }, // will be passed to the page component as props

  };

}

export default HomePage;
```

- **getStaticProps** is used to fetch data at build time and pre-render the page as static HTML.

✓ **Incremental Static Regeneration (ISR) with Next.js**

ISR allows pages to be updated after build time, serving static pages but regenerating them based on a defined time interval.

Steps:

1. Use **Next.js** with **ISR** to serve static content but update the page after a specified time interval.

Example: Implementing ISR in Next.js

```
// pages/index.js

import React from 'react';

const HomePage = ({ data }) => {
  return (
    <div>
      <h1>Incremental Static Regeneration Page</h1>
      <p>Regenerated data: {data}</p>
    </div>
  );
};


// This function runs at build time and updates the page every 10 seconds
export async function getStaticProps() {
  const data = 'This content was regenerated!';
  return {
    props: { data },
    revalidate: 10, // Revalidate the page every 10 seconds
  };
}
export default HomePage;
```

- The revalidate option defines the interval for regenerating the page, allowing for dynamic updates while still serving static content.

Points to Remember

Types of rendering techniques

1. Static Site Generation (SSG)

2. Server-Side Rendering (SSR)

3. Incremental Static Regeneration (ISR)

4. Client-Side Rendering (CSR)

Steps used to implement Static Site Generation (SSG)

Create a Next.js Project:

Use getStaticProps:

Steps used to implement Server-Side Rendering(SSR)

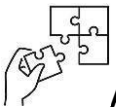
1. Create a Next.js Project (if not already done)
2. Use getServerSideProps

Steps used to implement Incremental Static Regeneration (ISR)

1. Create a Next.js Project (if not already done)
2. Use getStaticProps with Revalidation:

Steps used to implement Client-Side Rendering (CSR)

1. Create a Next.js Project (if not already done)
2. Fetch Data on the Client-Side



Application of learning 3.3.

ABC is a business company located in Kimihurura – Kigali City. It is planning to create a real-time Next.js e-commerce website. Assume you have been hired as a Next.js developer. You are tasked with building a product listing page for an e-commerce site using Next.js. The page should display a dynamic list of products fetched from an API, allowing users to filter by category and sort by price. Additionally, when users scroll down, more products should load seamlessly (infinite scrolling). You are requested to implement efficient rendering techniques, like server-side rendering (SSR) and static site generation (SSG), to handle the dynamic UI changes and optimize performance, ensuring the app renders smoothly without unnecessary re-renders or slow loading times

Indicative Content 3.4: Implementing Routing

Duration: 4 hrs

Theoretical Activity 3.4.1: Description of Routing

Tasks:

1. You are requested to answer the following questions related to nextJS routing

I. Describe the following concepts:
   a. File-system Based Routing
   b. Dynamic Routes
   c. Nested Routes
   d. Link Component
   e. Programmatic Navigation
   f. API Routes
   g. Catch-all Routes

II. What is the difference between using the <Link> component and a regular <a> tag?

III. What methods can you use for programmatic navigation in Next.js?

IV. What are some use cases for using dynamic routes in a web application?

2. Provide the answer for the asked questions and write them on papers.

3. Present the findings/answers to the whole class

4. For more clarification, read the key readings 3.4.1. In addition, ask questions where necessary.

Key readings 3.4.1.: Description of Routing

- **Description of key concepts**

✓ **File-System Based Routing**

File-System Based Routing is a method where the file structure of your application determines the routing of your pages. In frameworks like Next.js, pages are created by adding files to the pages directory. The file names and directory structure directly correspond to the route paths.

How it works:

- Each file in the pages directory automatically becomes a route.
- For example, pages/about.js becomes the /about route, and pages/blog/[id].js creates a dynamic route.

Example:

```
/pages
  ├── index.js      // Routes to '/'
  ├── about.js      // Routes to '/about'
  └── blog
     └── [id].js    // Routes to '/blog/:id'
```

- ✓ **Dynamic Routes**

Dynamic Routes allow you to create routes with variable segments, which can be useful for pages that depend on dynamic data like user IDs or post slugs.

How it works:

- You create dynamic routes using file names with brackets in the pages directory.
- Inside the file, you can access the dynamic segments using useRouter or getServerSideProps/getStaticProps.

Example:

```
/pages
  └── blog
     └── [id].js
```

Accessing Dynamic Segments:

```
// pages/blog/[id].js
import { useRouter } from 'next/router';
const BlogPost = () => {
  const router = useRouter();
  const { id } = router.query; // Access the dynamic segment
  return <div>Blog Post ID: {id}</div>;
};
export default BlogPost;
```

- ✓ **Nested Routes**

Nested Routes involve creating routes within routes, allowing for a hierarchical structure in your application.

How it works:

- Create a directory structure inside the pages folder to represent the nested routes.
- Each nested directory and file corresponds to a route segment.

Example:

```
/pages
  ├── index.js
  └── dashboard
     ├── index.js      // Routes to '/dashboard'
     └── settings.js   // Routes to '/dashboard/settings'
```

- ✓ **Link Component**

The **Link Component** provided by frameworks like Next.js helps in navigating between pages without causing a full page reload, which improves performance and user experience.

How it works:

- Use the Link component to wrap the text or elements you want to turn into links.
- The href prop specifies the path to navigate to.

Example:

```
// pages/index.js
import Link from 'next/link';
const HomePage = () => (
  <div>
    <h1>Home Page</h1>
    <Link href="/about">
      <a>Go to About Page</a>
    </Link>
  </div>
);
export default HomePage;
```

- ✓ **Programmatic Navigation**

Programmatic Navigation allows you to navigate to different routes programmatically in response to events or conditions, such as form submissions or button clicks.

How it works:

- Use router methods provided by useRouter (e.g., push, replace) to navigate programmatically.

Example:

```
import { useRouter } from 'next/router';
const NavigateButton = () => {
  const router = useRouter();
  const handleClick = () => {
    router.push('/about'); // Navigate to the /about page
  };
  return <button onClick={handleClick}>Go to About Page</button>;
};
export default NavigateButton;
```

- ✓ **API Routes**

API Routes allow you to build API endpoints directly within your Next.js application. These endpoints can be used to handle HTTP requests and respond with JSON data.

How it works:

- Create files inside the pages/api directory.
- Each file exports a function that handles HTTP requests.

Example:

```
/pages
  └── api
     └── hello.js
```

Example API Route:

```
// pages/api/hello.js
export default function handler(req, res) {
  res.status(200).json({ message: 'Hello World' });
}
```

- ✓ **Catch-All Routes**

Catch-All Routes allow you to match all routes under a certain path, including nested paths. They are useful for handling various dynamic routes or to create a fallback for unmatched routes.

How it works:Use brackets with three dots to create catch-all routes (e.g., [...slug].js).

Example:

```
/pages
  └── blog
     └── [...slug].js
```

Example Catch-All Route:

```
// pages/blog/[...slug].js
import { useRouter } from 'next/router';
const Blog = () => {
  const router = useRouter();
  const { slug } = router.query; // slug is an array of all matched segments
  return <div>Blog Slug: {JSON.stringify(slug)}</div>;
};
export default Blog;
```

Summary:

- **File-System Based Routing**: Routes are determined by the file structure in the pages directory.
- **Dynamic Routes**: Create routes with dynamic segments using bracket notation.
- **Nested Routes**: Create hierarchical routes by nesting directories in the pages directory.
- **Link Component**: Use the Link component for client-side navigation.
- **Programmatic Navigation**: Use router.push or router.replace for navigation in response to events.
- **API Routes**: Build API endpoints inside the pages/api directory.
- **Catch-All Routes**: Match all routes under a certain path using [...param].js.

These techniques collectively allow for flexible and powerful routing in modern React applications.



Practical Activity 3.4.2: Implementing routing



Task:

1: You are requested to go to the computer lab to implement Next.JS routing .

2. Read key reading 3.4.2

3: Apply safety precautions.

4: Implement routing.
5: Present your work to the trainer and whole class.
6: Perform the task provided in application learning 3.4



Key readings 3.4.2: Implementing routing

Here's how to implement routes in a React application using Next.js, which supports various routing techniques including **Linking Components**, **Programmatic Navigation**, **Dynamic Routes**, and **Query Parameters**.

- ✓ **Linking Components**

Linking Components is used for navigating between pages in a React application without full page reloads. In Next.js, you use the Link component from the next/link package to achieve this.

Steps:

1. **Create Your Pages**: Define the pages you want to link to.

```
// pages/index.js

import Link from 'next/link';

const HomePage = () => (

  <div>

    <h1>Home Page</h1>

    <Link href="/about">

      <a>Go to About Page</a>
```

```
  </Link>
 </div>
);
export default HomePage;
// pages/about.js
const AboutPage = () => (
 <div>
  <h1>About Page</h1>
  <Link href="/">
   <a>Back to Home Page</a>
  </Link>
 </div>
);
export default AboutPage;
```

2. **Run the Application**: Navigate between pages using the Link component.

✓ **Programmatic Navigation**

Programmatic Navigation allows you to navigate to different routes in response to events or conditions using JavaScript.

Steps:

1. **Create Your Pages**: Define the pages you want to navigate to.

```
// pages/index.js
import { useRouter } from 'next/router';
const HomePage = () => {
 const router = useRouter();
 const handleNavigate = () => {
  router.push('/about'); // Navigate to the About page
 };
```

```
  return (
    <div>
      <h1>Home Page</h1>
      <button onClick={handleNavigate}>Go to About Page</button>
    </div>
  );
};
export default HomePage;
```

2. **Run the Application**: Click the button to navigate programmatically.

✓ **Dynamic Routes**

Dynamic Routes are used to handle routes with variable segments, like user profiles or blog posts.

Steps:

1. **Create a Dynamic Route File**: Use square brackets to define a dynamic route.

```
// pages/blog/[id].js
import { useRouter } from 'next/router';
const BlogPost = () => {
  const router = useRouter();
  const { id } = router.query; // Get the dynamic segment
  return <div>Blog Post ID: {id}</div>;
};
export default BlogPost;
```

2. **Navigate to a Dynamic Route**: Use the Link component or programmatic navigation to visit the dynamic route.

```
// pages/index.js
import Link from 'next/link';
const HomePage = () => (
  <div>
```

```
    <h1>Home Page</h1>
    <Link href="/blog/123">
      <a>Go to Blog Post with ID 123</a>
    </Link>
  </div>
);
export default HomePage;
```

- ✓ **Query Parameters**

Query Parameters are used to pass data in the URL, such as search terms or filters.

Steps:

1. **Create a Page that Reads Query Parameters**:

```
// pages/search.js
import { useRouter } from 'next/router';
const SearchPage = () => {
  const router = useRouter();
  const { query } = router.query; // Read the query parameters
  return (
    <div>
      <h1>Search Results</h1>
      <p>Query: {query}</p>
    </div>
  );
};
export default SearchPage;
```

2. **Navigate with Query Parameters**:

```
// pages/index.js
import Link from 'next/link';
```

```
const HomePage = () => (
  <div>
    <h1>Home Page</h1>
    <Link href="/search?query=Next.js">
      <a>Search for Next.js</a>
    </Link>
  </div>
);
export default HomePage;
```

Summary of Implementations

1. **Linking Components**:
    - Use Link component for client-side navigation.
2. **Programmatic Navigation**:
    - Use router.push() or router.replace() for navigation based on events or conditions.
3. **Dynamic Routes**:
    - Define dynamic routes using square brackets in file names and access dynamic segments via router.query.
4. **Query Parameters**:
    - Access query parameters via router.query and construct URLs with query strings using Link.

These techniques collectively help you manage and navigate routes efficiently in a React application with Next.js.



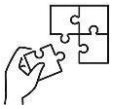
Points to Remember

Key concepts on routing

a) File-System Based Routing
b) Dynamic Routes
c) Nested Routes
d) Link Component
e) Programmatic Navigation
f) API Routes
g) Catch-all Routes
h) Linking Components
i) Programmatic Navigation
j) Dynamic Routes
k) Query Parameters

Steps to perform NextJS project routing

a) Linking Components
b) Programmatic Navigation
c) Dynamic Routes
d) Query Parameters



Application of learning 3.4.

ABC is a business company located in Kimihurura – Kigali City. It is planning to create a real-time Next.js e-commerce website. It is planning to create an e commerce website with real time loading links enabled by routing. Assume you have been hired to create their application using Next.js to explore key routing features. Start by setting up the project and creating a structured file hierarchy, including a home page, a dynamic route for individual posts, and a posts listing page populated with sample data. Implement a PostLink component to allow navigation to individual posts, and display all posts on the home page. Create a dynamic route that shows the post content and incorporates programmatic navigation for returning to the home page. Finally, enhance the user experience by adding a filtering feature that uses query parameters to dynamically filter posts based on user input, showcasing the core capabilities of Next.js routing.

Indicative content 3.5: Creation of API

Duration: 3 hrs

Practical Activity 3.5.1: Creating API

Task:

1: You are requested to go to the computer lab to create API.

2: Read key reading 3.5.1

3: Apply safety precautions.

4: Create API.

5: Present your work to the trainer and whole class.

6: Perform the task provided in application of learning 3.5

Key readings 3.5.1: Creating API

- **Steps to create API**

✓ **Define the API Endpoint**

An API endpoint is a URL where your API can be accessed by clients. To define an API endpoint:

- **Choose a Framework:** Depending on the technology stack you're using, the way to define an endpoint will vary. For example, in Node.js with Express, you define endpoints in a server file.

```
const express = require('express');

const app = express();

// Define a GET endpoint

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

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

});

app.listen(3000, () => {

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

});
```

- **Set the URL Path:** Decide on the path for your endpoint (e.g., /api/users, /api/products).
- **Specify HTTP Method:** Determine what HTTP methods (GET, POST, PUT, DELETE, etc.) your endpoint will handle.

✓ **Handling Request Types**

Handling request types involves processing different HTTP methods that a client can use to interact with your API.

- **GET:** Retrieve data from the server.

```
app.get('/api/items', (req, res) => {
  // Retrieve and return items
});
```

- **POST:** Send data to the server to create a new resource.

```
app.post('/api/items', (req, res) => {
  // Create a new item
});
```

- **PUT:** Update an existing resource on the server.

```
app.put('/api/items/:id', (req, res) => {
  // Update item with id
});
```

- **DELETE:** Remove a resource from the server.

```
app.delete('/api/items/:id', (req, res) => {
  // Delete item with id
});
```

✓ **Using Dynamic API Routes**

Dynamic routes allow you to create endpoints that can handle variable data in the URL.

- **In Express.js:**

```
app.get('/api/items/:id', (req, res) => {
  const itemId = req.params.id;
```

```
  // Handle request for item with the given id
});
```

Here, :id is a placeholder for dynamic data. You can access this data using req.params.id.

- **In Next.js (for example):**

```
// pages/api/items/[id].js
export default function handler(req, res) {
  const { id } = req.query;
  // Handle request for item with the given id
}
```

Here, [id] is a dynamic segment in the file name that corresponds to the URL parameter.

- ✓ **Testing Your API**

Testing is crucial to ensure that your API behaves as expected. Here are some ways to test:

- **Unit Tests:** Use frameworks like Mocha, Jest, or Supertest in Node.js to write automated tests.

```
const request = require('supertest');
const app = require('../app'); // Your Express app
describe('GET /api/items', () => {
  it('should return items', async () => {
    const res = await request(app).get('/api/items');
    expect(res.status).toBe(200);
    expect(res.body).toHaveProperty('items');
  });
});
```

- **Manual Testing:** Use tools like Postman or cURL to send requests to your API and inspect the responses.
    - **Postman:** Create requests for each endpoint and method, and check the responses.

- **cURL:** Use command-line requests to test your API.

curl -X GET http://localhost:3000/api/items

- **Integration Tests:** Test how different parts of your application work together. This often involves more complex setups and can be done using the same tools as unit tests.

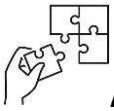
By following these steps, you should be able to define, handle, and test your API endpoints effectively. If you need more specific examples or help with a different technology stack, let me know!



Points to Remember

Steps used to create API

1. Define the API Endpoint
2. Handling Request Types
3. Using Dynamic API Routes
4. Testing your API



Application of learning 3.5.

HTD Company is a software development company located in Kicukiro District – Kigali City. It wants to hire a full stack developer to integrate their front-end application to other external applications in order to share data. Assume you are hired as their developer. You are tasked to develop a simple task management application using Next.js. Create an API that defines a central endpoint at pages/api/tasks.js, which handles various request types such as GET for retrieving tasks, POST for creating new tasks, PUT for updating existing ones, and DELETE for removing tasks. To enhance functionality, implement dynamic API routes like pages/api/tasks/[id].js, allowing users to interact with specific tasks based on their unique identifiers. Finally, test your API using tools like Postman or curl to ensure each operation—retrieving, creating, updating, and deleting tasks—works correctly and returns the expected responses.

Indicative Content 3.6: Securing The Application

Duration: 4 hrs

Practical Activity 3.6.1: Securing the application

Task:

1: You are requested to go to the computer lab to Perform Client-Side Security, Server-Side Security, and General Security Measures.
2. Read key reading 3.6.1
3: Apply safety precautions.
4: Secure the application
5: Present your work to the trainer and whole class.
6: Perform the task provided in application of learning 3.6

Key readings 3.6.1: Securing the application

- **Performing Client-Side Security**

✓ **Client-Side Rendering (CSR) Security**

In CSR, you need to ensure that data fetched from APIs and other resources is handled securely.

- **Avoid Exposing Sensitive Data:** Ensure that sensitive data is not exposed in the client-side code. Never include secrets or tokens directly in your client-side code.
- **Use HTTPS:** Always serve your application over HTTPS to protect data in transit. Ensure that any APIs or resources fetched from the client are also served over HTTPS.
- **Sanitize and Validate Data:** Always validate and sanitize user inputs on the client side before sending them to your API. Use libraries like dompurify to clean any HTML content.

```
npm install dompurify

import DOMPurify from 'dompurify';

// Sanitize HTML content

const cleanHTML = DOMPurify.sanitize(userInputHTML);
```

- **Implement Security Headers:** Use Next.js middleware or custom servers to add security headers to your application.

```
// Example with Next.js custom server using Helmet (Express.js)
const helmet = require('helmet');
const express = require('express');
const next = require('next');
const app = next({ dev: process.env.NODE_ENV !== 'production' });
const handle = app.getRequestHandler();
app.prepare().then(() => {
 const server = express();
 // Use Helmet to set security headers
 server.use(helmet());
 server.all('*', (req, res) => handle(req, res));
 server.listen(3000, (err) => {
  if (err) throw err;
  console.log('> Ready on http://localhost:3000');
 });
});
```

✓ **Cross-Origin Resource Sharing (CORS)**

CORS is a mechanism that allows you to specify who can access your resources.

- **Server-Side CORS Configuration:** Configure CORS on your server to allow or restrict access to your resources.

```
If you're using Next.js API routes, configure CORS using a library like cors.
npm install cors
// pages/api/_middleware.js
import Cors from 'cors';
// Initialize CORS middleware
const cors = Cors({
```

```
  methods: ['GET', 'POST', 'PUT', 'DELETE'],
  origin: 'https://your-allowed-origin.com',
});
// Helper method to wait for a middleware to execute before continuing
function runMiddleware(req, res, fn) {
  return new Promise((resolve, reject) => {
    fn(req, res, (result) => {
      if (result instanceof Error) {
        return reject(result);
      }
      return resolve(result);
    });
  });
}
export async function middleware(req, res) {
  await runMiddleware(req, res, cors);
  // Rest of your API handler logic
}
```

- **Client-Side CORS Issues:** Ensure that you handle CORS-related issues correctly. The browser enforces CORS, so if you're making requests from the client, make sure the server supports the necessary CORS headers.

✓ **Session Management**

Managing sessions securely is crucial for protecting user data and authentication.

- **Use HTTP-Only Cookies:** Store session tokens in HTTP-only cookies to prevent JavaScript access, mitigating XSS risks.

```
import Cookie from 'js-cookie';
// Set a cookie
Cookie.set('sessionToken', 'your-session-token', { secure: true, sameSite: 'Strict', httpOnly:
```

true });

// Retrieve a cookie

const token = Cookie.get('sessionToken');

- **Implement Session Expiry:** Ensure sessions expire after a reasonable period. Use secure methods for token invalidation and renewal.
- **Use Secure Storage:** For sensitive data, avoid using localStorage or sessionStorage. Prefer HTTP-only cookies or secure storage solutions.

✓ **Third-Party Libraries (Auth0)**

Auth0 is a popular service for authentication and authorization. Integrate Auth0 for secure authentication.

- **Set Up Auth0 in Next.js:**

Install the necessary packages:

npm install @auth0/nextjs-auth0

Configure Auth0:

// pages/api/auth/[...auth0].js

import { handleAuth } from '@auth0/nextjs-auth0';

export default handleAuth();

Create a login button in your components:

// components/LoginButton.js

import { useUser } from '@auth0/nextjs-auth0';

export default function LoginButton() {

```
 const { user } = useUser();
 return user ? (
  <button onClick={() => window.location.href = '/api/auth/logout'}>
   Logout
  </button>
 ) : (
  <button onClick={() => window.location.href = '/api/auth/login'}>
```

```
    Login
  </button>
 );
}
```

Protect pages using Auth0:

```
// pages/protected.js
import { withPageAuthRequired } from '@auth0/nextjs-auth0';
function ProtectedPage() {
 return <div>This page is protected and requires authentication.</div>;
}
export default withPageAuthRequired(ProtectedPage);
```

Update your .env.local with Auth0 credentials:

```
AUTH0_SECRET=your-secret
AUTH0_BASE_URL=http://localhost:3000
AUTH0_ISSUER_BASE_URL=https://your-auth0-domain.auth0.com/
AUTH0_CLIENT_ID=your-client-id
AUTH0_CLIENT_SECRET=your-client-secret
```

By implementing these client-side security measures in Next.js, you can help ensure that your application is robust against common security threats. If you need further assistance with any specific aspect, feel free to ask!

- ✓ **Performing Server-Side Security**

- HTTPS Enforcement

Enforcing HTTPS is critical for securing data in transit between clients and your server.

Configure HTTPS on Your Hosting Provider: Most modern hosting providers like Vercel, Netlify, and AWS offer automatic HTTPS with SSL/TLS certificates. Ensure this is enabled for your domain.

- **Force HTTPS in Custom Servers**

If you are using a custom server (e.g., with Express), you can redirect HTTP traffic to HTTPS:

```
// server.js (Custom server example)
import express from 'express';
import next from 'next';
const dev = process.env.NODE_ENV !== 'production';
const app = next({ dev });
const handle = app.getRequestHandler();
app.prepare().then(() => {
  const server = express();
  // Redirect HTTP to HTTPS
  server.use((req, res, next) => {
    if (req.headers['x-forwarded-proto'] !== 'https') {
      return res.redirect('https://' + req.headers.host + req.url);
    }
    next();
  });

  server.all('*', (req, res) => handle(req, res));
  server.listen(3000, (err) => {
    if (err) throw err;
    console.log('> Ready on http://localhost:3000');
  });
});
```

- ✓ **Server-Side Rendering (SSR) Security**

Server-side rendering requires careful handling of data to avoid exposing sensitive information.

- **Avoid Data Leakage**

Ensure that sensitive data is not included in the HTML sent to the client.

```
// pages/index.js
export async function getServerSideProps(context) {
  const data = await fetchDataFromAPI(); // Secure data fetching


  // Ensure no sensitive data is exposed
  return {
    props: {
      data: data.publicInfo, // Only send non-sensitive data
    },
  };
}
```

- **Use Environment Variables**

Store sensitive configuration values in environment variables and access them securely in your server-side code.

```
// .env.local
DATABASE_URL=your-database-url
// Example usage
const databaseUrl = process.env.DATABASE_URL;
```

✓ **API Routes Security**

Securing API routes is crucial for protecting your application's backend.

- **Authentication and Authorization**

```
// pages/api/secure-data.js
import jwt from 'jsonwebtoken';
const secret = process.env.JWT_SECRET;
export default function handler(req, res) {
  const token = req.headers.authorization?.split(' ')[1];
```

```
  if (!token) {
    return res.status(401).json({ message: 'Unauthorized' });
  }
  try {
    jwt.verify(token, secret);
    res.status(200).json({ data: 'Secure data' });
  } catch (error) {
    res.status(401).json({ message: 'Invalid token' });
  }
}
```

- **Input Validation and Rate Limiting**

Validate inputs and implement rate limiting to protect against abuse.

```
npm install express-rate-limit
// pages/api/secure-data.js
import rateLimit from 'express-rate-limit';
const apiLimiter = rateLimit({
  windowMs: 15 * 60 * 1000,
  max: 100,
  message: 'Too many requests, please try again later.',
});

export default function handler(req, res) {
  apiLimiter(req, res, () => {
    // Handle API request
    res.status(200).json({ message: 'Request successful' });
  });
}
```

✓ **Content Security Policy (CSP)**

CSP helps prevent XSS attacks by specifying which sources of content are allowed.

Set CSP Headers

Use Next.js middleware or a custom server setup to add CSP headers.

```
// Example with Next.js custom server using Helmet (Express.js)
import helmet from 'helmet';
import express from 'express';
import next from 'next';
const app = next({ dev: process.env.NODE_ENV !== 'production' });
const handle = app.getRequestHandler();
app.prepare().then(() => {
  const server = express();
  server.use(helmet({
    contentSecurityPolicy: {
      directives: {
        defaultSrc: ["'self'"],
        scriptSrc: ["'self'", "https://trusted-scripts.example.com"],
        styleSrc: ["'self'", "https://trusted-styles.example.com"],
      },
    },
  }));
  server.all('*', (req, res) => handle(req, res));
  server.listen(3000, (err) => {
    if (err) throw err;
    console.log('> Ready on http://localhost:3000');
  });
});
```

✓ **Authentication**

Implement robust authentication mechanisms to protect your application.

- **Use Auth0 or Other OAuth Providers**

Auth0 can be integrated for secure authentication:

npm install @auth0/nextjs-auth0

- **Configure Auth0**

```
// pages/api/auth/[...auth0].js

import { handleAuth } from '@auth0/nextjs-auth0';

export default handleAuth();
```

- **Protect Pages**

```
// pages/protected.js

import { withPageAuthRequired } from '@auth0/nextjs-auth0';

function ProtectedPage() {
  return <div>This page is protected and requires authentication.</div>;
}

export default withPageAuthRequired(ProtectedPage);
```

- **Configure Auth0 in .env.local**

```
AUTH0_SECRET=your-secret

AUTH0_BASE_URL=http://localhost:3000

AUTH0_ISSUER_BASE_URL=https://your-auth0-domain.auth0.com/

AUTH0_CLIENT_ID=your-client-id

AUTH0_CLIENT_SECRET=your-client-secret
```

By implementing these security measures, you can help protect your Next.js application from various threats and ensure a safer experience for your users. If you need more details or have additional questions, feel free to ask!

✓ **Performing General Security Measures**

Implementing general security measures is crucial for ensuring the overall security and integrity of your Next.js application. These measures encompass various aspects, including

best practices for code, server configurations, data handling, and user interactions. Here's a detailed guide to general security measures you should consider:

- **Code Security Practices**

  Input Validation and Sanitization: Ensure that all user inputs are validated and sanitized both client-side and server-side to prevent injection attacks (e.g., SQL injection, XSS).

```
npm install validator dompurify

// Example for input validation using validator

import validator from 'validator';

if (!validator.isEmail(userInput.email)) {

  throw new Error('Invalid email');

}

// Example for input sanitization using dompurify

import DOMPurify from 'dompurify';

const sanitizedInput = DOMPurify.sanitize(userInput.html);
```

  Use Secure Dependencies: Regularly audit your dependencies for vulnerabilities using tools like npm audit or yarn audit.

```
npm audit
```

  Follow Best Practices for Secure Coding:

  - Avoid hardcoding secrets or sensitive information in your code.
  - Use parameterized queries to prevent SQL injection.
  - Regularly review and update your code to follow security best practices.

- **Environment and Configuration Security**

  Use Environment Variables: Store sensitive information such as API keys and database credentials in environment variables rather than hardcoding them.

```
DATABASE_URL=your-database-url

API_KEY=your-api-key
```

  Secure Your .env Files: Ensure .env files are not included in version control by adding them to .gitignore.

.env

Limit Access to Environment Variables: Restrict access to environment variables and configuration files to authorized personnel only.

- **Network Security**
- **Implement HTTPS:** Ensure that your application and API endpoints are served over HTTPS to encrypt data in transit.
- **Secure Your APIs:**
    - **Rate Limiting:** Protect APIs from abuse by implementing rate limiting.

```
npm install express-rate-limit

import rateLimit from 'express-rate-limit';

const apiLimiter = rateLimit({
  windowMs: 15 * 60 * 1000,
  max: 100,
  message: 'Too many requests, please try again later.',
});

server.use('/api/', apiLimiter);
```

    - **CORS:** Configure Cross-Origin Resource Sharing (CORS) to restrict which domains can access your APIs.

```
npm install cors

import Cors from 'cors';

const cors = Cors({
  methods: ['GET', 'POST', 'PUT', 'DELETE'],
  origin: 'https://your-allowed-origin.com',
});

export default async function handler(req, res) {
  await runMiddleware(req, res, cors);
  // Your API logic here
}
```

- **User Authentication and Authorization**

  Implement Strong Authentication: Use robust authentication mechanisms like OAuth or Auth0 to manage user access.

```
npm install @auth0/nextjs-auth0

// pages/api/auth/[...auth0].js

import { handleAuth } from '@auth0/nextjs-auth0';

export default handleAuth();
```

  Authorization: Ensure that users have the correct permissions to access specific resources or perform certain actions.

```
// pages/protected.js

import { withPageAuthRequired } from '@auth0/nextjs-auth0';

function ProtectedPage() {
  return <div>This page is protected and requires authentication.</div>;
}

export default withPageAuthRequired(ProtectedPage);
```

- **Data Security**

  Encrypt Sensitive Data: Use encryption to protect sensitive data both at rest and in transit.

```
npm install bcrypt

import bcrypt from 'bcrypt';

// Hash a password

const saltRounds = 10;

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

// Verify a password

const isMatch = await bcrypt.compare('password', hashedPassword);
```

- **Secure Data Storage:** Ensure that databases and storage systems are securely configured and access is restricted.

- **Monitoring and Logging**

Implement Logging: Use logging to monitor your application and detect potential security issues.

```
npm install winston

import winston from 'winston';

const logger = winston.createLogger({
  level: 'info',
  format: winston.format.json(),
  transports: [
    new winston.transports.File({ filename: 'error.log', level: 'error' }),
    new winston.transports.File({ filename: 'combined.log' }),
  ],
});

logger.info('Server started');
```

Monitor Security Events: Regularly review logs and set up alerts for suspicious activity.

- **Regular Security Audits**

  Conduct Penetration Testing: Periodically test your application for vulnerabilities using penetration testing tools or services.

  Review and Update Security Policies: Regularly review and update your security policies and practices to address new threats and vulnerabilities.

- **User Security Awareness**

  Educate Users: Provide guidance to users on best practices for maintaining their account security, such as using strong passwords and enabling two-factor authentication (2FA).

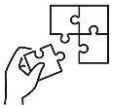
By implementing these general security measures, you can significantly enhance the security posture of your Next.js application and protect against a wide range of threats. If you have any specific questions or need further details, feel free to ask!.



Points to Remember

Steps used to secure Next.JS application

- Performing Client-Side Security
- Performing Server-Side Security
- Performing General Security Measures



Application of learning 3.6.

XYZ Developers is a web application company located in Kigali city. It has developed a banking application for XMY bank in Next.JS and requires a competent developer to manage security in the application before it is deployed. Assume it has hired you to you to secure their application. You are requested to implement a comprehensive strategy that encompasses both client-side and server-side measures. Start by focusing on validating user inputs, manage sessions with authentication libraries, and configure access controls to limit unauthorized API access. Ensure secure data transmission, protect sensitive information during rendering, and enforce checks for accessing API routes. Establish policies to prevent various types of attacks and incorporate general security measures such as regular audits, secure headers, and keeping third-party libraries updated.



Points to Remember

Steps used to apply the TypeScript basics

1. npm install -g typescript command to install Typescripts using terminal
2. Steps to implement interfaces for variables in TypeScript
3. Defining an Interface
4. Implementing the defined Interface for Variables
5. Optional Properties in Interfaces
6. Read-Only Properties in Interfaces
7. Implementing Methods in Interfaces
8. Extending Interfaces
9. Indexable Types in Interfaces
10. Data Handling like API data validation, Form validation and, error handling and exceptions

Steps used to prepare a Next.js project environment

1. Install Node.js and npm
2. Set Up a New Next.js Project
3. Navigate to Your Project Directory
4. Run the Development Server
5. Explore the Project Structure
6. Install Additional Packages
7. Using TypeScript (Optional)
8. Environment Variables (Optional)
9. Configure ESLint (Optional)
10. Build and Production

Steps used to create NextJS Project

1. Install Node.js (if not already installed)
2. Create a New Next.js Project
3. Navigate to the Project Directory
4. Run the Development Server
5. Explore Your Next.js App

- Initialize NextJS project Development
    1. Creating Pages and components
    2. Implementing search engine optimization (SEO)
    3. Styling
    4. Caching Strategies

Steps used to implement Static Site Generation (SSG)

1. Create a Next.js Project:
2. Use getStaticProps:

Steps used to implement Server-Side Rendering(SSR)

1. Create a Next.js Project (if not already done)
2. Use getServerSideProps

Steps used to implement Incremental Static Regeneration (ISR)

1. Create a Next.js Project (if not already done)
2. Use getStaticProps with Revalidation:

Steps used to implement Client-Side Rendering (CSR)

1. Create a Next.js Project (if not already done)
2. Fetch Data on the Client-Side

Steps used to implement Next.JS Routing

1. File-System Based Routing
2. Dynamic Routes
3. Nested Routes
4. Link Component
5. Programmatic Navigation
6. API Routes
7. Catch-all Routes
8. Linking Components
9. Programmatic Navigation
10. Dynamic Routes
11. Query Parameters

Steps used to create API

1. Define the API Endpoint
2. Handling Request Types
3. Using Dynamic API Routes
4. Testing your API

Steps used to secure Next.JS application

1. Performing Client-Side Security
2. Performing Server-Side Security
3. Performing General Security Measures



Learning Outcome 3 End Assessment

Theoretical assessment

Q1. Read the Following statement and answer by true if correct or false otherwise

TypeScript is a statically typed superset of JavaScript that requires a separate compiler, enhancing type checking and error detection. While it lacks configuration options for module resolution, it offers features like interfaces and enums for better code organization. In frameworks like Next.js, nested routes allow for a structured routing system, but API routes do not support dynamic URL segments. Client-side security measures are essential even with server-side rendering (SSR). The Next.js Link component facilitates navigation within the app rather than to external URLs. Additionally, implementing a Content Security Policy (CSP) is effective in mitigating cross-site scripting (XSS) attacks, enhancing security.

a. TypeScript requires a separate compiler that is different from JavaScript.
b. TypeScript does not support configuration options for module resolution
c. Nested routes allow you to create child routes within a parent route.
d. API routes in Next.js cannot handle dynamic segments in the URL.
e. Performing client-side security measures is unnecessary if you are using server-side rendering (SSR).
f. The Link component in Next.js can only navigate to external URLs.
g. Implementing a Content Security Policy (CSP) can help mitigate cross-site scripting (XSS) attacks.

Q2.Read carefully the following questions related to Next.Js application and circle the most correct alternative alphabetic letter.

1. Which component is used for client-side navigation in Next.js?
A) Router
B) Link
C) Navigate
D) Route

2. Dynamic routes in Next.js can be created by using:
A) [] brackets in the filename
B) () brackets in the filename
C) # symbols in the filename
D) None of the above

3. Which of the following is NOT a request type handled by an API?
A) GET
B) POST
C) PATCH
D) EXECUTE

4. To secure your Next.js application against cross-origin attacks, you should implement:
A) CORS
B) HTTPS
C) CSP
D) All of the above

Q3. Read the following sentences and fill the gap with the missing word.

a) The purpose of _____ is to manage user sessions and authenticate requests on the server side.
b) Using _____ in Next.js allows you to pass parameters directly in the URL and retrieve them in your component.
c) Programmatic navigation can be achieved using the _____ method provided by the Next.js router.
d) To create a catch-all route, you would use the syntax _____ in the filename.
e) Testing your API can be done using tools like _____ or Postman to ensure the endpoints are functioning correctly.

Q4. Match each concept on the left with the correct description or related item on the right.

Concepts	**Descriptions**
1. Preparation of Environment	A. The process of defining the layout and visual appearance of your application using CSS or a CSS-in-JS library.
2. Project Creation	B. The configuration of tools and dependencies, such as Node.js and npm, required to run a Next.js application.
3. Creating Pages and Components	C. A method to enhance web performance by storing and retrieving data from a cache rather than fetching it each time.
4. Implementing Search Engine Optimization (SEO)	D. The use of meta tags and structured data to improve visibility and ranking in search engines.
5. Styling	E. The fundamental building blocks of a Next.js application, typically stored in the pages directory.

6. Caching Strategies	F. The command npx create-next-app used to initiate a new Next.js project.
7. API Integration	
8. Testing and Quality Assurance	

Practical assessment

XYZ Developers is a web application company located in Kigali city. It needs to hire a Next.JS developer for the company. Assume you are hired as their developer and tasked with preparing a travel booking platform using Next.js and TypeScript. You are requested to start by setting up the Next.js project with TypeScript support. Ensure safety throughout your application. Implement various rendering techniques, such as server-side rendering and static site generation, to optimize performance and improve user experience. Create a robust API to handle user requests for searching and booking accommodations, activities, and transportation. Finally, leverage Next.js routing to establish a seamless navigation system between key pages, including Home, Search Results, and Booking Details, ensuring a smooth user journey throughout the platform.

END



References

Morrow, T. (2020). *Understanding Client-Side Rendering vs. Server-Side Rendering*. Smashing Magazine. https://www.smashingmagazine.com/2020/06/client-side-rendering-vs-server-side-rendering/

Tailwind CSS. (n.d.). *Using Next.js with Tailwind CSS*. https://tailwindcss.com/docs/guides/nextjs

ts-node. (n.d.). ts-node Documentation. https://typestrong.org/ts-node/

TypeScript. (n.d.). Interfaces in TypeScript. https://www.typescriptlang.org/docs/handbook/2/objects.html#interfaces

TypeScript. (n.d.). tsconfig.json Reference. https://www.typescriptlang.org/tsconfig

TypeScript. (n.d.). TypeScript Handbook. https://www.typescriptlang.org/docs/handbook/intro.html

Vercel. (n.d.). Configuring TypeScript in Next.js. https://nextjs.org/docs/app/building-your-application/configuring-typescript

Vercel. (n.d.). Create a Next.js app. https://nextjs.org/docs/app/building-your-application/create-next-app

Vercel. (n.d.). Data fetching in Next.js. https://nextjs.org/docs/app/building-your-application/data-fetching

Vercel. (n.d.). File-system routing in Next.js. https://nextjs.org/docs/app/building-your-application/routing/file-system-routing

Vercel. (n.d.). Next.js documentation. https://nextjs.org/docs

Vercel. (n.d.). Next.js project structure. https://nextjs.org/docs/app/building-your-application/project-structure

Vercel. (n.d.). Rendering methods in Next.js. https://nextjs.org/docs/app/building-your-application/rendering

Vercel. (n.d.). Running a development server in Next.js. https://nextjs.org/docs/app/building-your-application/development

Vercel. (n.d.). Setting up ESLint in Next.js. https://nextjs.org/docs/app/building-your-application/configuring-linting

Vercel. (n.d.). Static files in Next.js. https://nextjs.org/docs/app/building-your-application/static-file-serving

Vercel. (n.d.). Using the App Router. https://nextjs.org/docs/app/building-your-application/routing/app-router

Williams, C. (2021). *Static Site Generation with Next.js*. CSS-Tricks. https://css-tricks.com/static-site-generation-with-next-js/


Service Worker
Cache

Indicative contents

4.1 Maintain Responsiveness
4.2 Configuring Web Application Manifest
4.3 Implementation of Service Workers

Key Competencies For Learning Outcome 4: Apply Progressive Web Application

Knowledge	Skills	Attitudes
● Description of mobile first design ● Identification of performance optimization techniques ● Description of manifest file ● Description of service workers	● Maintaining the responsiveness ● Configuring web application manifest ● Implementing the service workers	● Being creative in developing progressive web application ● Being problem-solving oriented during development of progressive web application ● Being updated on latest ReactJS versions ● Being collaborative while developing progressive web application



Duration: 20 hrs

Learning outcome 4 objectives:



By the end of the learning outcome, the trainees will be able to:

1. Describe clearly mobile first design in web application development.
2. Identify correctly the performance optimization techniques that are used in web application development.
3. Describe clearly the manifest file and service workers in development of web application.
4. Maintain correctly the responsiveness of web application in development.
5. Configure correctly web application manifest in the development.
6. Implement correctly the service workers in development of progressive web application.



Resources

Equipment	Tools	Materials
● Computer	● VS Code ● Browser ● Terminal	● Internet ● Electricity



Advance Preparation:

Before delivering this learning outcome, you are recommended to:

- Have prepared computer lab with Internet connectivity
- Have visual studio code installed on all computers to be used.
- Have videos to be used as didactic material.
- Have sample React.JS project developed to be used.

Indicative Content 4.1: Maintain Responsiveness

Duration: 8 hrs

Theoretical Activity 4.1.1: Description of web application responsiveness

Tasks:

1: You are requested to answer the following questions

i. What do you understand by:
   a) Leverage progressive enhancement
   b) Prioritizing mobile-first design
ii. What are the performance optimization techniques of PWA?

2:Write your findings on papers,flipchart or chalkboard
3:Present the findings/answers to the whole class or trainer.
4: For more clarification, read the key readings 4.1.1.

Key readings 4.1.1.: Description of web application responsiveness

- **Leverage progressive enhancement**

Leverage progressive enhancement is a design strategy where you build the most basic, core functionality first, ensuring it works across all devices and browsers, and then progressively add more advanced or enhanced features for browsers and devices that can support them.

✓ **Basic functionality first**

Ensure that your application works on all devices with basic functionality. This means that even if advanced features or styles are not supported, users can still access the core content and functionality.

Example:

```
// Basic component with TailwindCSS classes

const Button = () => (

 <button className="px-4 py-2 bg-blue-500 text-white rounded">

  Click Me

 </button>
```

);

In this example, the button will look basic but functional on all devices.

- ✓ **Mobile-First Approach**

Start by designing for the smallest screen size and then progressively enhance the design for larger screens. TailwindCSS's utility classes make it easy to handle this with responsive modifiers.

Example:

```
// Responsive component using TailwindCSS

const Card = () => (

  <div className="p-4 bg-white shadow-md rounded sm:p-6 md:p-8 lg:p-10">

    <h1 className="text-lg md:text-xl lg:text-2xl font-bold">Card Title</h1>

    <p className="text-sm md:text-base lg:text-lg">

      This is a responsive card component.

    </p>

  </div>

);
```

In this example, the padding and font sizes change based on screen size, providing a better experience on larger screens.

- ✓ **Progressive enhancement with advanced features**

Enhance the user experience on devices that support advanced features. This could include adding animations, advanced interactions, or additional styling that isn't necessary but adds value.

Example:

```
// Adding animations using TailwindCSS for supported browsers

const AnimatedButton = () => (

  <button className="px-4 py-2 bg-blue-500 text-white rounded transition-transform transform hover:scale-105">

    Hover Me

  </button>
```

);

In this example, the transition-transform and transform classes add a hover effect that enhances the button's interactivity on modern browsers while keeping it functional without animations on older ones.

- **Prioritize Mobile-First Design**

Prioritizing mobile-first design in a Progressive Web Application (PWA) means designing and developing your application primarily for mobile devices before scaling up to larger screens like tablets and desktops. This approach is crucial for ensuring a good user experience, especially given the prevalence of mobile device usage. The steps you can use to prioritize mobile-first design in a PWA are:

✓ **Design for small screens first**

Start your design process by focusing on mobile screens, which are typically smaller and have more constraints compared to larger screens. This ensures that your core content and functionality are optimized for mobile users.

Example:

- **Layout:** Use single-column layouts that work well on small screens.

```
<div className="grid grid-cols-1 sm:grid-cols-2 lg:grid-cols-3 gap-4">
  <div className="bg-gray-200 p-4">Grid Item 1</div>
  <div className="bg-gray-200 p-4">Grid Item 2</div>
  <div className="bg-gray-200 p-4">Grid Item 3</div>
</div>
```

- **Navigation:** Implement a mobile-friendly navigation menu, such as a hamburger menu or bottom navigation bar.

✓ **Use Responsive Units and Breakpoints**

TailwindCSS offers responsive units and breakpoints to handle different screen sizes. Start with styles for small screens and then use media queries or responsive utility classes to adjust for larger screens.

Example:

```
// Responsive example using TailwindCSS
const ResponsiveCard = () => (
```

```
<div className="p-4 bg-white shadow-md rounded sm:w-1/2 md:w-1/3 lg:w-1/4">
  <h1 className="text-lg md:text-xl lg:text-2xl font-bold">Card Title</h1>
  <p className="text-sm md:text-base lg:text-lg">
    This card adjusts its width and font size based on screen size.
  </p>
</div>
);
```

- ✓ **Consider performance and loading times**

Mobile devices often have slower network speeds and less processing power. Optimize your PWA for performance by minimizing resource sizes, lazy-loading content, and using efficient caching strategies.

Example:

- Image optimization: Use responsive images with srcset or sizes attributes.

```
<img src={`${process.env.PUBLIC_URL}/images/tailwind.png`}
      srcSet={`${process.env.PUBLIC_URL}/images/tailwind.png  2x`}
      class="w-full h-auto h-auto sm:w-1/2 lg:w-1/3 object-cover" alt="Description"
/>
```

- Lazy loading: Implement lazy loading for off-screen images and components.

```
<img src="../images/tailwind.png" alt="Tailwind image" loading="lazy"
  className="w-full h-auto"/>
```

- ✓ **Responsive Typography**

Make sure your text scales well across devices by using Tailwind's responsive typography utilities.

```
<p className="text-base sm:text-lg md:text-xl lg:text-2xl">
 Responsive text content for different screen sizes.
</p>
```

- **Utilize performance optimization techniques**

Optimizing performance is crucial to ensure fast load times and smooth interactions in PWAs.

Some key performance optimization techniques:

- ✓ **Code Splitting and Lazy Loading**

Code splitting helps break down your application into smaller chunks, which can be loaded on demand. React's **React.lazy** and **Suspense** are useful for this purpose.

Example:

```
// Lazy loading a component
import React, { Suspense, lazy } from 'react';
const LazyComponent = lazy(() => import('./LazyComponent'));
const App = () => (
  <div>
    <Suspense fallback={<div>Loading...</div>}>
      <LazyComponent />
    </Suspense>
  </div>
);
```

- ✓ **Use TailwindCSS Purge**

TailwindCSS Purge is a feature designed to remove unused CSS from your production build, significantly reducing the size of your final CSS file. Tailwind generates thousands of utility classes, which can lead to large CSS files if not optimized. Purge scans your HTML, JavaScript, and other templates for classes you're actually using, and then removes any unused ones.

TailwindCSS generates a lot of utility classes, but not all of them are used in your application. Use Tailwind's purge feature to remove unused CSS classes and reduce the CSS file size.

In the tailwind.config.js file, define the content key (previously called purge in older versions) to specify which files should be scanned for TailwindCSS classes.

Example:

```
module.exports = {
  content: [
    './src/**/*.{html,js,jsx,ts,tsx}',  './public/index.html',
  ],
  theme: {
    extend: {},
  },
  plugins: [],
}
```

- The content array tells Tailwind where to look for classes. You should include the paths for your HTML, JSX, Blade templates, or any other file types where you use Tailwind classes.
- Use the * wildcard to select all files of certain types and their subdirectories.

- ✓ **Improve initial load time**

Initial load time is the time it takes for a web page or application to fully load and become usable when a user first accesses it. This includes downloading and rendering the necessary resources like HTML, CSS, JavaScript, images, fonts, and other assets. Initial load time is a critical factor in user experience, as longer load times can lead to higher bounce rates and reduced user satisfaction.

Optimize the initial load time by using techniques like server-side rendering (SSR) with Next.js or static site generation (SSG) where applicable.

Example:

Server-Side Rendering: Use frameworks like Next.js for SSR to deliver pre-rendered HTML to the client.

```
// Example of Next.js server-side rendering

export async function getServerSideProps() {

  const res = await fetch('https://api.example.com/data');

  const data = await res.json();

  return { props: { data } };
}
const Page = ({ data }) => (
  <div>
    <h1>{data.title}</h1>
  </div>
);
```

- ✓ **Minimize and compress assets**

Minimizing and compressing assets in a React.js application is to reducing the size of your CSS, JavaScript, HTML, and other assets (like images) to improve load times, performance, and overall user experience.

Example:

- **JavaScript and CSS Minification:** Use tools like **Terser** for JavaScript and **cssnano** for CSS.
- **Image Optimization:** Use image formats like WebP, and tools like ImageOptim or Squoosh for compression.

- ✓ **Efficient CSS with TailwindCSS**

Efficient CSS with TailwindCSS refers to using TailwindCSS to create fast, scalable, and maintainable styles by optimizing how CSS is written and applied in web projects.

Tailwind's utility-first approach, where you use pre-defined classes instead of writing custom CSS rules for each component, helps streamline development, reduce file size, and improve overall performance.

Example:

Utility-First Approach: Use Tailwind's utility classes for styling, which avoids the need for custom CSS rules that could lead to larger stylesheets.

- ✓ **Analyze and optimize performance**

Use performance analysis tools to identify bottlenecks and optimize accordingly.

Example:

- **Chrome DevTools:** Use the Performance tab to analyze rendering and network performance.
- **Lighthouse:** Use Lighthouse for auditing performance, accessibility, and best practices.



Practical Activity 4.1.2: Maintaining web application responsiveness



Task:

1:You are requested to go to the computer lab to maintain the web application responsiveness of web application.
2: Read the key readings 4.1.2
3:Apply safety precautions
4:Maintain web application responsiveness
5: Present your work to the trainer
6: Perform the task provided in application of learning 4.1



Key readings 4.1.2: Maintaining web application responsiveness

Step 1: Use Tailwind's utility classes for styling

- **Responsive Design**: Tailwind CSS provides responsive utility classes that allow you to apply different styles at various breakpoints. For example, you can use

classes like sm:bg-red-500, md:bg-blue-500, and lg:bg-green-500 to change background colors based on the screen size.

- **Flexibility and Customization**: Instead of writing custom CSS, you can compose styles directly in your HTML using utility classes. This approach promotes a clean and maintainable codebase, as styles are applied inline.
- **Grid and Flex Utilities**: Utilize Tailwind's grid and flex utilities (grid, flex, flex-row, flex-col, etc.) to create responsive layouts that adapt to different screen sizes effortlessly.

Step 2: Use TailwindCSS Purge

- **Removing Unused CSS**: Tailwind CSS can generate a large CSS file due to its extensive utility classes. By enabling the Purge option in your Tailwind configuration file, you can remove unused styles from your final build, significantly reducing file size.
- **Configuration**: Set up PurgeCSS in the tailwind.config.js file to specify which files to scan for class names. For example:

```
module.exports = {
  purge: ['./src/**/*.{js,jsx,ts,tsx}', './public/index.html'],
  // other configurations...
};
```

Improved Performance: By purging unused styles, your application loads faster and uses less bandwidth, contributing to a better user experience, especially on mobile devices.

Step 3: Optimizing images

- **Use Responsive Images**: Use the srcset attribute in the <img> tag to serve different image sizes based on device resolution and viewport size. This ensures users only download images suitable for their screen.
- **Image Formats**: Choose appropriate formats (e.g., WebP for modern browsers) to reduce file sizes without sacrificing quality.
- **Lazy Loading**: Implement lazy loading for images using the loading="lazy" attribute, which helps in loading images only when they are about to enter the viewport, improving initial load times.

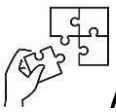
Step 4: Make sure your text scales well across devices

- **Responsive Font Sizes**: Use Tailwind's responsive font size utilities (e.g., text-sm, text-base, text-lg, text-xl) to ensure that text scales appropriately on different screen sizes. You can also use responsive modifiers like text-sm md:text-base lg:text-lg to adjust text size.



Points to Remember

- **Leverage progressive enhancement** is a design strategy where you build the most basic, core functionality first, ensuring it works across all devices and browsers.
- **Prioritizing mobile-first design** in a Progressive Web Application (PWA) means designing and developing your application primarily for mobile devices before scaling up to larger screens like tablets and desktops.
- **Code splitting** helps break down your application into smaller chunks, which can be loaded on demand.
- **TailwindCSS Purge** is a feature designed to remove unused CSS from your production build, significantly reducing the size of your final CSS file.
- **Initial load time** is the time it takes for a web page or application to fully load and become usable when a user first accesses it.
- **Minimizing and compressing assets** is to reducing the size of your CSS, JavaScript, HTML, and other assets (like images) to improve load times, performance, and overall user experience.
- **Efficient CSS with TailwindCSS** refers to using TailwindCSS to create fast, scalable, and maintainable styles by optimizing how CSS is written and applied in web projects.
- To maintain the web application responsiveness, you do the followings:
    - ✓ Use Tailwind's utility classes for styling.
    - ✓ Use TailwindCSS Purge
    - ✓ Optimizing the image
    - ✓ Make sure your text scales well across devices.



Application of learning 4.1.

ABC is a company located in Kabuga- Kigali city. It sells It equipment online. It has an online website which the customers are not happy about its displays on all gadgets. It requires to make it user friendly to all internet accessing gadget. Assume it has hired you to solve the problem, you are tasked with developing a Progressive Web Application (PWA) for the company that caters to both desktop and mobile users. The company expects the PWA to deliver an optimal and responsive experience across all devices, including those with varying network speeds and hardware capabilities. The PWA should be performant, fast-loading, and fully functional on older browsers as well as the latest ones.

Indicative Content 4.2: Configuring Web Application Manifest

Duration: 5 hrs

Theoretical Activity 4.2.1: Description of web application manifest

Tasks:

1: You are requested to answer the following questions

i. What is manifest file?
ii. What are the key functions of manifest.json?
iii. Outline the common properties of manifest.json?
iv. Explain the steps used for referencing the manifest in your HTML.
v. Explain how can you test React application in development environment with Light house.

2: Write your findings on papers,flipchart or chalkboard
3:Present the findings/answers to the whole class or trainer
4: For more clarification, read the key readings 4.2.1.

Key readings 4.2.1.: Description of web application manifest

- **Creating and configuring the Manifest File**

Manifest file is a JSON file that provides important metadata about the web application, helping to control how the app appears to the user and how it behaves when installed on a device.

The manifest.json file is used to define how the Progressive Web App (PWA) should behave when installed on a user's device, including icons, theme color, and more.

The key functions of manifest.json are:

- ✓ It allows users to install the web app on their home screen, like a native app.
- ✓ It defines how the app looks when launched (e.g., full screen, standalone), what icons are used, and what splash screen appears.
- ✓ It controls behaviors like the starting URL, orientation, and display mode.

The web application manifest provides essential metadata for your PWA, allowing it to be installed and used like a native app.

The steps to be followed:

Step 1. Create a manifest.json file in the **public** directory of your React app.

```
{
  "name": "My PWA",
```

```
  "short_name": "PWA",
  "description": "A progressive web app built with React and TailwindCSS.",
  "start_url": "/",
  "display": "standalone",
  "background_color": "#ffffff",
  "theme_color": "#007bff",
  "icons": [
   {
    "src": "/icons/icon-192x192.png",
    "sizes": "192x192",
    "type": "image/png"
   },
   {
    "src": "/icons/icon-512x512.png",
    "sizes": "512x512",
    "type": "image/png"
   }
  ]
}
```

- **Common properties in manifest.json:**
  - ✓ **name**: The full name of the application (used in the install prompt or app listings).
  - ✓ **short_name**: A short version of the app name, often used when space is limited (e.g., on a mobile home screen).
  - ✓ **start_url**: The URL that is loaded when the app is launched from the home screen or desktop.
  - ✓ **display**: Controls how the app is displayed, for example:
    - "fullscreen": Uses the entire screen.
    - "standalone": Runs without browser UI, like a native app.
    - "minimal-ui": Displays with a minimal browser UI.
  - ✓ **background_color**: The color shown on the splash screen while the app is loading.
  - ✓ **theme_color**: The color of the browser's toolbar or the status bar when the app is launched.
  - ✓ **icons**: Specifies a set of icons for the app in various sizes (used for the app icon on a device's home screen or desktop).
  - ✓ **orientation**: Defines the default orientation for the app (e.g., portrait or landscape).

Step 2: Include icons in your **/public/icons/** directory. The manifest file refers to these icons, so make sure to include the correct sizes, like 192x192 and 512x512 pixels. You can use tools like PWABuilder to generate icons.

- **Referencing the manifest in your HTML**

You need to link the manifest file in your **index.html** file located in the **public** directory.
The steps to be followed:
Step 1: Open **public/index.html** and add the following **<link>** tag inside the **<head>** tag to reference the manifest.
<link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
The **%PUBLIC_URL%** is a special placeholder in React that will be replaced with the correct URL path during the build process.
Step 2: Add a meta tag for the theme color.
<meta name="theme-color" content="#007bff" />

- **Testing and Validation**

Once you've set up the manifest file and referenced it correctly, you'll want to test and validate the PWA setup.
The steps to be followed:

✓ **Testing in development:**
Ensure you are running your app with a secure context (i.e., https). You can test your PWA locally using Lighthouse in Chrome DevTools:
- Open your React app in Chrome.
- Open DevTools by right-clicking and selecting "Inspect."
- Go to the "Lighthouse" tab.
- Check "Progressive Web App" and click "Generate report."
- Lighthouse will run tests and provide a PWA score, helping you identify areas for improvement.

✓ **Testing in production:**
- Build your React app for production using: **npm run build**
- Serve the build using a simple server like **serve**: **npx serve -s build**
- Access the app at **http://localhost:5000** and test PWA functionality by checking for the "Add to Home Screen" prompt.

✓ **Validation:**
- Use online tools such as Web App Manifest Validator to check if your manifest is properly set up.
- Check that the app can be installed and opened in standalone mode on a mobile device.



Practical Activity 4.2.2: Configuring web application manifest



Task:

1: You are requested to go to the computer lab to configure web application manifest file

2: Read the key readings 4.2.2

3: Apply safety precautions

4:Configure the web application manifest file

5: Present your work to the trainer.

6:Perform the task provided in application of learning 4.2



Key readings 4.2.2: Configuring web application manifest

Step1: Create a manifest.json File in the Public Folder

In your project's root directory, navigate to the public folder (if it doesn't exist, create it).

Create a new file named manifest.json.

Step 2:Add configuration elements

Open the manifest.json file and include the following elements:

```
{
  "name": "Your App Name",
  "short_name": "App",
  "description": "A brief description of your app.",
  "start_url": "/index.html",
  "display": "standalone",
  "background_color": "#ffffff",
  "theme_color": "#ffffff",
```

```
 "orientation": "portrait",
 "icons": [
  {
   "src": "/icons/icon-192x192.png",
   "sizes": "192x192",
   "type": "image/png"
  },
  {
   "src": "/icons/icon-512x512.png",
   "sizes": "512x512",
   "type": "image/png"
  }
 ]
}
```

Step 3: Place your app icons in the /public/icons folder

- Create an icons folder inside the public folder if it doesn't already exist.
- Add your app icons (e.g., icon-192x192.png and icon-512x512.png) to the /public/icons folder.

Step 4: Include the manifest in index.html

In your index.html file, add a link to the manifest:

```
<link rel="manifest" href="/manifest.json">
```

Step 5: Configure index.html for web app installation

To allow users to install your web app, ensure you have the following meta tags in your index.html file:

```
<meta name="theme-color" content="#ffffff">
<meta name="apple-mobile-web-app-capable" content="yes">
<meta name="apple-mobile-web-app-status-bar-style" content="default">
<meta name="apple-mobile-web-app-title" content="Your App Name">
```

Step 6: Testing the manifest

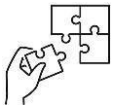
- Use Chrome DevTools (F12) and go to the "Application" tab.
- Check the "Manifest" section to ensure that all details are correct and that the icons are loading properly.
- Use the "Lighthouse" tool in Chrome DevTools to run an audit on your PWA. It will check for PWA compliance and give suggestions for improvement.



Points to Remember

- **Manifest file** is a JSON file that provides important metadata about the web applicati on, helping to control how the app appears to the user and how it behaves when inst alled on a device.
- **name**: The full name of the application (used in the install prompt or app listings).
- **short_name**: A short version of the app name, often used when space is limited (e.g., on a mobile home screen).
- **start_url**: The URL that is loaded when the app is launched from the home screen or desktop.
- **display**: Controls how the app is displayed.
- **background_color**: The color shown on the splash screen while the app is loading.
- **theme_color**: The color of the browser's toolbar or the status bar when the app is lau nched.
- **icons**: Specifies a set of icons for the app in various sizes (used for the app icon on a d evice's home screen or desktop).
- **orientation**: Defines the default orientation for the app (e.g., portrait or landscape).
- Build your React app for production using: **npm run build**
- Serve the build using a simple server like **serve**: **npx serve -s build**
- To configure web application manifest, you do the followings:
  - ✓ Create a manifest.json file in public folder
  - ✓ Add configuration elements such as name, short_name, description, start_url, display, background_color, theme_color, orientation, icons
  - ✓ Place your app icons in the /public/icons/folder.

- ✓ Include the manifest in index.html
- ✓ Configure index.html for web app installation
- ✓ Testing the Manifest



Application of learning 4.2.

ABC is a University located in Kabuga- Kigali city. It has an eLearning website but students complain of typing its long domain name. Sometimes they forget it due to its length. The university has decided to implement PWA in their website. Assume you have been hired to solve the problem. You are tasked to develop a PWA for the university. The platform should offer various courses that users can access on both desktop and mobile devices. To enhance the user experience, you need to create and configure the web application manifest file, ensuring it meets the PWA requirements.

Indicative Content 4.3: Implementation of Service Workers

Duration: 7 hrs

Practical Activity 4.3.1: Implementing of service workers

Task:

1: You are requested to go to the computer lab to implement the service worker in React.JS application.
2: Read the key readings 4.3.1
3:Apply safety precautions
4: Implement service worker
5:Present your work to the trainer
6: Perform the task provided in application of learning 4.3

Key readings 4.3.1: Implementing of service workers

Service workers are a key component of PWAs, running in the background and acting as a proxy between your web application and the network. They enable features like offline functionality, caching, background sync, and push notifications.

The key features of Service Workers are:

- **Offline caching:** Store assets (HTML, CSS, JavaScript, images) so the application can load even without an internet connection.
- **Background synchronization:** Handle tasks such as syncing data with the server in the background.
- **Performance improvement:** Reduce network requests by serving cached resources.
- **Push notifications:** Enable push messages even when the app is not active.

Step 1: Create a service worker file

Create a service worker file in the public directory of your React application.

- Name the file service-worker.js.
- The structure of the public directory will look like this:

```
my-app/
├── public/
|   ├── index.html
|   ├── service-worker.js  // Your service worker file
├── src/
|   ├── index.js
```

```
|  └── ...
```

Step 2: Register the Service Worker

Register the service worker in the index.js file of your React application.

- Modify the index.js file to include the registration logic:

```
// src/index.js
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';
import './index.css';
const root = document.getElementById('root');
ReactDOM.render(<App />, root);
// Register service worker
if ('serviceWorker' in navigator) {
  window.addEventListener('load', () => {
    navigator.serviceWorker
      .register('/service-worker.js')
      .then(registration => {
        console.log('Service Worker registered with scope:', registration.scope);
      })
      .catch(error => {
        console.error('Service Worker registration failed:', error);
      });
  });
}
```

Step 3: Install the Service Worker

Install the service worker by using the file you created.

Inside service-worker.js, add the following code to set up a basic service worker:

```
// public/service-worker.js
self.addEventListener('install', (event) => {
  console.log('Service Worker installing...');
});
```

Step 4: Implement cache strategies

Implement cache strategies in the service worker file.

Here's an example of a cache strategy using the Cache API:

```
const CACHE_NAME = 'my-app-cache-v1';
const urlsToCache = [
  '/',
  '/index.html',
  '/static/js/bundle.js',
  '/static/css/main.css',
```

```
  // Add other assets as needed
];
self.addEventListener('install', (event) => {
  event.waitUntil(
    caches.open(CACHE_NAME)
      .then((cache) => {
        console.log('Opened cache');
        return cache.addAll(urlsToCache);
      })
  );
});
self.addEventListener('fetch', (event) => {
  event.respondWith(
    caches.match(event.request)
      .then((response) => {
        return response || fetch(event.request);
      })
  );
});
```

Step 5: Activate the Service Worker

Add an activate event listener in your service-worker.js to clean up old caches:

```
self.addEventListener('activate', (event) => {
  const cacheWhitelist = [CACHE_NAME];
  event.waitUntil(
    caches.keys().then((cacheNames) => {
      return Promise.all(
        cacheNames.map((cacheName) => {
          if (cacheWhitelist.indexOf(cacheName) === -1) {
            return caches.delete(cacheName);
          }
        })
      );
    })
  );
});
```

Step 6: Update the Service Worker

Update the service worker whenever changes occur.

To manage updates, add logic to handle the update event in your service-worker.js:

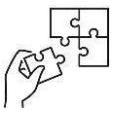
```
self.addEventListener('fetch', (event) => {
  event.respondWith(
```

```
    caches.match(event.request)
      .then((response) => {
        if (response) {
          // Update the cache with the latest response
          fetch(event.request).then((networkResponse) => {
            caches.open(CACHE_NAME).then((cache) => {
              cache.put(event.request, networkResponse.clone());
            });
          });
          return response;
        }
        return fetch(event.request);
      })
  );
});
```



Points to Remember

- To implement service worker, you do the followings:
  - ✓ Create a service worker file in public directory of React Application.
  - ✓ Register the service worker in the index.js of React Application.
  - ✓ Install service worker by using service worker file created.
  - ✓ Implement cache strategies in service worker file created.
  - ✓ Activate the service worker
  - ✓ Update the service worker.



Application of learning 4.3.

X-buy has a retail e-commerce web application, is experiencing performance challenges. Users often complain about slow loading times, especially in areas with poor network connectivity. Assume you are hired as a front-end developer to solve the above problem. You are tasked to improve user experience by implementing service workers to enable caching of critical assets and provide offline functionality.



Learning Outcome 4 End Assessment

Theoretical assessment

Q1. Read carefully the following statements about Progressive Web Application and answer by TRUE if the statement is correct and by FALSE if the statement is incorrect.

a) Leveraging Progressive Enhancement means starting with a basic version of the application and adding features for more capable browsers.
b) Prioritizing Mobile-First Design involves designing the application primarily for desktop users and then adapting it for mobile devices.
c) Utilizing Performance Optimization Techniques can include minimizing image sizes, optimizing code, and reducing the number of HTTP requests to enhance the loading speed of a React application.
d) Progressive Enhancement ignores the capabilities of modern browsers and focuses solely on older browsers to ensure compatibility.
e) A Mobile-First Design approach typically results in a better user experience on all devices by ensuring that mobile users have a fully functional and optimized interface.
f) Performance optimization techniques have no impact on the responsiveness of a React application.
g) The web application manifest is a JSON file that provides metadata about the web application.
h) The manifest file can specify the application's name, icons, and start URL.
i) The display property in the manifest controls how the app appears on the user's home screen.
j) Testing and validation of the manifest file can be done using the browser's Developer Tools.
k) The manifest file should be included in the public directory of a React application to be served correctly.
l) A service worker must be registered before it can be installed.
m) A service worker will continue to function even if the user closes the browser.
n) Service workers can only cache static assets; they cannot handle dynamic content.

Q2. Fill in the blank spaces with appropriate word(s). Select from the given choices in the box.

<link>, index.js, Lazy loading, index.css, Lighthouse, caches.delete(), display, install and <a>

a) A service worker can be registered in a React application by using the navigator.serviceWorker.register method, typically placed in the …… file of the application.
b) You can reference the manifest file by adding a …….. tag in the index.html file inside the public folder.
c) The …… event is triggered when the service worker is being installed. This is where caching of static assets and other initial setup can be performed.
d) The …… method is used to remove specific cache entries.
e) You can use tools like the …… tool in Chrome DevTools or online manifest validators to ensure the manifest file follows best practices and is PWA-compliant.
f) The …… property is set to standalone or fullscreen to make it appear like a native app.
g) …… is a performance optimization technique used in web development, where certain resources (like images, videos, scripts, or components) are loaded only when they are needed rather than during the initial page load.

3.Match the column A and Column B

Description(column A)		**Key term(Column B)**
1.When a new service worker is detected, the existing one is replaced.		**A.** Service workers
2.They act as a proxy between the network and the application.		**B**.Prioritize Mobile-First Design
3.Start design with mobile layouts and scale up to larger screens.		**C**.CSS Media Queries
4.Serve appropriate image sizes based on device resolution.		**D**.Lazy loading
5.Use to apply different styles based on device characteristics.		**E**.Responsive images
6.Utilize tools like Lighthouse or Manifest Validator to check your manifest.		**F**.Name
7.Load resources only when they are needed to improve performance		**G**.JSON
8.Property specifies the name of the application in the manifest		**H**.Validating the manifest file with tools

9.Primary format used to define the manifest file		**I**.Updating Service Worker

Practical assessment

MXN breads is a Bakery Company found in Rwamagana district- Eastern Province –Rwanda. It needs to have an ecommerce website that will enable its customers to buy online. Assume it has hired you as a full stack developer and tasked you to develop a Progressive Web Application (PWA) for a local bakery using React. The bakery wants to provide users with an engaging and fast experience, accessible on both desktop and mobile devices. Your tasks include ensuring that the app leverages progressive enhancement, prioritizes mobile-first design, and utilizes performance optimization techniques. Additionally, you need to configure the manifest file, implement a caching strategy, and manage the service worker lifecycle effectively.

END



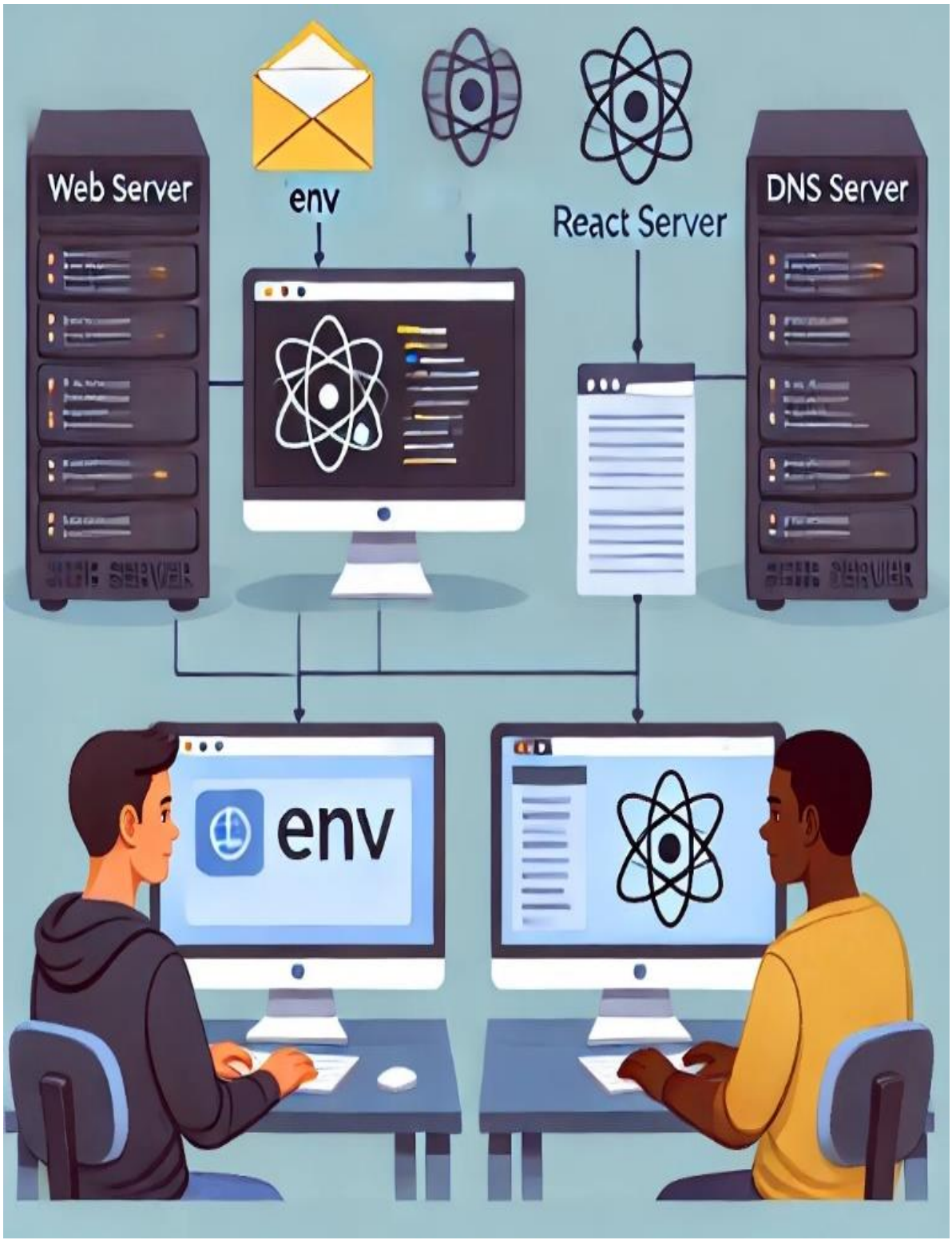
References

DhiWise. (n.d.). *The ultimate guide to achieving React mobile responsiveness*. DhiWise. https://www.dhiwise.com/post/the-ultimate-guide-to-achieving-react-mobile-responsiveness

Google Developers. (n.d.). *Add a web app manifest*. web.dev. Retrieved October 11, 2024, from https://web.dev/articles/add-manifest

MDN Web Docs. (n.d.). *Progressive web apps (PWAs)*. Mozilla. Retrieved October 11, 2024, from https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps

Uploadcare. (n.d.). *Service workers: A complete guide with examples*. Uploadcare Blog. https://uploadcare.com/blog/service-workers-tutorial/


Web Server
env
React Server
DNS Server
env

Indicative contents

5.1 Configuration of Environment Variables
5.2 Deploying React Application
5.3 Setup Custom Domain

Key Competencies For Learning Outcome 5: Publish The Application

Knowledge	Skills	Attitudes
● Identification of variables ● Description of application files of React application ● Description of DNS	● Configuring the environment variables ● Deploying React application ● Setting up the custom domain	● Being problem-solving oriented during deployment of web application ● Being collaborative while publishing the application ● Being adaptable on different hosting platforms



Duration: 20 hrs

Learning outcome 5 objectives:



By the end of the learning outcome, the trainees will be able to:

1. Identify correctly variables that are used in configuring the environment.
2. Describe clearly the application files that are used in deploying a React application.
3. Describe clearly DNS in publishing a React application
4. Configure correctly the environment variables in publishing the React application.
5. Deploy correctly React application.
6. Setup correctly custom domain in publishing the React application.



Resources

Equipment	Tools	Materials
● Computer ● DNS server	● Browser ● Terminal	● Internet ● Electricity

Indicative Content 5.1: Configuration of Environment Variables

Duration: 5 hrs

Practical Activity 5.1.1: Configuring environment variables

Task:

1: You are requested to go to the computer lab to configure the environment variables.

2: Read the key readings 5.1.1

3: Apply safety precautions

4:Configure the environment variables

5: Present your work to the trainer

6:Perform the task provided in application of learning 5.1

Key readings 5.1.1: Configuring environment variables

Step 1: Create .env files

- **Location**: Place the .env file at the root of your React project (same level as package.json).
- **File Naming**: You can create multiple environment files, such as:
  - ✓ .env (for development)
  - ✓ .env.production (for production)
  - ✓ .env.development (for specific development settings)

Step 2: Define variables

Define your variables in the .env file using the format:

REACT_APP_VARIABLE_NAME=value

Example:

REACT_APP_API_URL=https://api.example.com

REACT_APP_API_KEY=your_api_key

Prefix Requirement: Note that in React, all environment variables must start with REACT_APP_ to be accessible in your application.

Step 3: Access the variables in the code

Using process.env: Access the defined environment variables in your React components or JavaScript files like this:

```
const apiUrl = process.env.REACT_APP_API_URL;
const apiKey = process.env.REACT_APP_API_KEY;
```

```
console.log(apiUrl); // Outputs: https://api.example.com
Usage in Components: You can use these variables directly in your components
function App() {
  return (
    <div>
      <h1>API URL: {process.env.REACT_APP_API_URL}</h1>
    </div>
  );
}
```

Step 4: Setup the storage environment

- **Development Setup**: For local development, ensure your .env file is set up with all necessary variables. Use a .gitignore file to exclude it from version control.
- **Production Setup**: On deployment platforms (like Vercel), you can usually set environment variables directly in the platform's settings. This way, they don't need to be stored in your codebase.

Step 5: Use platform-specific options

- **Vercel**:
  - ✓ Go to your project settings and navigate to the "Environment Variables" section.
  - ✓ Add variables there without needing to prefix them.
- **Netlify**:
  - ✓ In your site settings, find "Build & Deploy" → "Environment" → "Environment Variables."
  - ✓ Add your variables, ensuring they are prefixed with REACT_APP_.
- **Heroku**:
  - ✓ Use the Heroku CLI to set environment variables

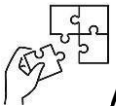
```
heroku config:set REACT_APP_API_URL=https://api.example.com
```

This command can also be done through the Heroku dashboard under the "Settings" tab.



Points to Remember

- To configure the environment variables, you do the followings:
  - ✓ Create .env files
  - ✓ Define variables
  - ✓ Access the variables in the code.
  - ✓ Setup the storage environment.
  - ✓ Use platform-specific option



Application of learning 5.1.

ABC is a e commerce company located in Kigali city. It has an isolated website which does not interact with payment websites. The company wants a full stack developer to integrate it with other application. Assume you have been hired and tasked with setting up environment variables for a ReactJS application that needs to interact with several backend services and third-party platforms. The project involves connecting to an FTP server to upload files and making API requests to a backend server. The following environment variables are required:

- Backend Host URL
- FTP Host URL
- FTP Username
- FTP Password

Additionally, you need to configure storage options depending on whether the app is running in a development, production, or testing environment. You also need to ensure platform-specific configurations are handled correctly.

Indicative Content 5.2: Deploying React Application

Duration: 8 hrs

Practical Activity 5.2.1: Deploying React application

Task:

1: You are requested to go to the computer lab to deploy React application.
2: Read the key readings 5.2.1
3: Apply safety precautions
4: Deploy React application
5: Perform the task provided in application of learning 5.2

Key readings 5.2.1: Deploying React application

Step 1: Create build folder

To prepare your React application for deployment, you first need to create a production build. This process compiles your source code into static files.

- **Command**: Run the following command in your terminal:npm run build
- **Output**: This command generates a build folder containing optimized production-ready files (HTML, CSS, and JavaScript). The files are minified and bundled for performance.

Step 2: Configure the deployment platform

Depending on the platform you choose (e.g., Vercel, Netlify, GitHub Pages), you'll need to follow specific steps to configure your deployment.

- **Choose a Deployment Platform**: Select a platform that fits your needs.
- **Link Repository**: For platforms like Vercel or Netlify, link your GitHub (or other version control) repository.
- **Set Build Command**: Most platforms will automatically detect the build command (npm run build), but ensure it's set correctly.
- **Set Publish Directory**: Specify the directory where the production build files are located, usually build/.

Step 3: Migrate the necessary files

After configuring your platform, you need to migrate the build folder contents to the hosting server.

- **Automatic Deployment**: For platforms like Vercel or Netlify, the migration happens automatically once you link your repository and trigger a build.
- **Manual Upload**: If you're deploying to a server (e.g., using FTP or SSH), manually upload the contents of the build folder to the server's designated directory (often

public_html or similar).

Step 4: Test the React application deployed

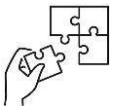
Once your application is deployed, it's crucial to test its functionality.

- **Access the Deployed URL**: Visit the URL provided by your deployment platform to see your application live.
- **Functionality Testing**: Check that all routes, links, and components function as expected. Test various features to ensure everything works correctly.
- **Performance Monitoring**: Use tools like Google Lighthouse to analyze performance, accessibility, and SEO.



Points to Remember

- To deploy React application, you do the followings:
  - ✓ Create build folder.
  - ✓ Configure the deployment platform
  - ✓ Migrate the necessary files
  - ✓ Test the React application deployed.



Application of learning 5.2.

ABC company is a company located in Kigali City. It hired a React.js developer to develop a customer feedback system. The company has the application on its local machines and needed to deploy it for users to access. Assume you are hired by the company and tasked to deploy a React application that has been developed and is ready for production. The deployment will be done using Vercel as the hosting platform. You will be responsible for building the project, configuring the deployment platform, migrating the files, and testing the deployed application to ensure it is running smoothly.

Indicative Content 5.3: Setup Custom Domain

Duration: 7 hrs

Theoretical Activity 5.3.1: Description of DNS

Tasks:

1: You are requested to answer the following questions

i. Write in full words DNS.
ii. What is the role of DNS?
iii. Give five examples of domain names.
iv. Explain the hierarchy and structure of DNS.
v. Explain the name resolution process performed by DNS.

2: Write your findings on papers,flipchart or chalkboard

3: Present the findings/answers to the whole class or trainer

4: For more clarification, read the key readings 5.3.1

Key readings 5.3.1: Description of DNS

- **DNS**

DNS (Domain Name System) translates human-readable domain names (e.g., www.example.com) into IP addresses (e.g., 192.0.2.1) that computers use to identify each other on the network. It is an essential component of the internet's infrastructure.

✓ **Translation of Domain Names to IP Addresses**

The Translation of Domain Names to IP Addresses is a fundamental function of DNS. Human-readable domain names (like www.example.com) need to be converted into numerical IP addresses (like 192.0.2.1) that computers use to identify each other on the network.

How It Works:

- When a user types a domain name into a web browser, the browser needs to find the corresponding IP address to connect to the server hosting the website.
- This translation allows users to interact with websites using easy-to-remember names instead of complex numerical addresses.

Example: If you enter www.example.com in your browser, a DNS query will be made to find the associated IP address, allowing the browser to access the site.

✓ **Hierarchy and Structure**

The **Hierarchy and Structure** of DNS is organized in a tree-like format, which allows for efficient management and resolution of domain names.

Components:

- **Root Level**: The top of the hierarchy, represented by a dot (.). It includes the root DNS servers.
- **Top-Level Domains (TLDs)**: Directly below the root, these are the domains like .com, .org, .net, and country-code TLDs like .uk, .de.
- **Second-Level Domains**: These are directly to the left of the TLD. For example, in example.com, example is the second-level domain.
- **Subdomains**: Further divisions of second-level domains. For instance, blog.example.com is a subdomain of example.com.

Purpose: This hierarchical structure allows DNS to efficiently handle queries and distribute the load across multiple servers, making the system scalable and resilient.

✓ **Name Resolution Process**

The **Name Resolution Process** describes how a domain name is resolved into an IP address through a series of steps involving various DNS components.

Steps:

1. **DNS Query Initiation**: When a user enters a domain name in the browser, the operating system checks its local cache for a stored IP address.
2. **Recursive Query**: If the address is not cached, the query is sent to a recursive DNS resolver (often provided by the ISP). The resolver is responsible for finding the IP address.
3. **Root Name Server**: The resolver queries one of the root name servers, which returns the IP address of the TLD server for the requested domain (e.g., for .com).
4. **TLD Name Server**: The resolver then queries the TLD name server, which responds with the IP address of the authoritative name server for the specific domain.
5. **Authoritative Name Server**: Finally, the resolver queries the authoritative name server for the domain, which provides the IP address associated with the requested domain name.
6. **Response to User**: The resolver caches the IP address for future requests and sends the IP address back to the user's browser, allowing it to connect to the server.

Example Flow:

1. User types www.example.com.
2. OS checks local cache—no entry found.
3. Recursive resolver queried.
4. Resolver contacts a root server, then a .com TLD server, and finally the authoritative server for example.com.
5. IP address returned to the browser, which connects to the server.



Practical Activity 5.3.2: Setting up custom domain



Task:

1: You are requested to go to the computer lab to set up the custom domain
2: Read the key readings 5.3.2
3: Apply safety precautions
4: Set up the custom domain
5: Present your work to the trainer
6:Perform the task provided in application of learning 5.3



Key readings 5.3.3: Setting up custom domain

Step1: Link the Custom Domain to the React.js Application

- **Choose a Hosting Provider:** Select a hosting platform (e.g., Vercel, Netlify, GitHub Pages) that supports custom domains.
- **Access Domain Settings:** Go to the dashboard of your hosting provider and find the option to add a custom domain.
- **Add Custom Domain:** Enter your custom domain (e.g., www.yourdomain.com) in the appropriate field. The hosting provider will often provide you with specific instructions or an automatic setup option.

Step 2: Configure DNS Settings

- **Access Domain Registrar:** Log in to your domain registrar (e.g., GoDaddy, Namecheap).
- **Find DNS Management:** Navigate to the DNS management section for your domain.
- **Add DNS Records:**

✓ **A Record:** If your hosting provider gives you an IP address, create an A record pointing to that IP.

✓ **CNAME Record:** If your hosting provider provides a CNAME (e.g., yourapp.vercel.app), create a CNAME record pointing your custom domain to this URL.

✓ **Subdomain Setup:** If using a subdomain (e.g., app.yourdomain.com), configure the subdomain in your hosting settings and set the corresponding DNS record.

Step 3: Configure the SSL

- **Automatic SSL Setup:** Most hosting providers offer automatic SSL configuration for custom domains. After linking the domain, check if the SSL certificate is automatically generated.
- **Manual SSL Setup (if needed):** If your provider does not automatically configure

SSL:

- ✓ Obtain an SSL certificate from a certificate authority (CA) or use a service like Let's Encrypt.
- ✓ Follow the hosting provider's instructions to upload and configure the SSL certificate.

Step 4: Perform Testing and Verification

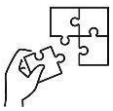
- **Check DNS Propagation:** Use online tools like "What's My DNS" to verify that the DNS changes have propagated worldwide.
- **Access Your Domain:** Open a web browser and navigate to your custom domain (e.g., www.yourdomain.com) to check if the application loads correctly.
- **Verify SSL Configuration:** Look for the padlock icon in the browser's address bar to confirm that SSL is properly configured.
- **Testing Functionality:** Ensure all links, resources, and functionalities in the application work as expected under the custom domain.



Points to Remember

- **DNS** (Domain Name System) translates human-readable domain names into IP addresses that computers use to identify each other on the network.
- **Root Level:** The top of the hierarchy, represented by the dot (.) at the end of the domain (often implied).
- **TLD (Top-Level Domain):** The first part after the root, such as .com, .org, .net, etc.
- **Second-Level Domain:** The actual domain name chosen by the user, such as example in example.com.
- **Subdomains:** Optional parts of the domain that come before the second-level domain, such as www in www.example.com.

- **To set up custom domain, you do the followings:**
  - ✓ Link the custom domain to the React.js application
  - ✓ Configure DNS settings
  - ✓ Configure the SSL
  - ✓ Perform testing and verification



Application of learning 5.3.

ABC company is a company located in Kigali City. It hired a React.js developer to develop a customer feedback system. The company has the application on its local machines and needed to deploy it for users to access. Assume you are hired by the company and tasked to configure it for a custom domain with DNS and SSL settings. Your client, ABC Technologies, has purchased the domain www.abctech.com. You are using Vercel to host the React app.

Your task includes the following:

1. Set up the custom domain www.abctech.com for the React application.
2. Configure the DNS settings correctly for the domain.
3. Enable SSL to ensure secure communication (HTTPS).
4. Verify that the domain is correctly configured, secured, and that the application is accessible via https://www.abctech.com.



Learning Outcome 5 End Assessment

Theoretical assessment

Q1. Read carefully the following statements about publish the application and answer by TRUE if the statement is correct and by FALSE if the statement is incorrect.

a) Environment variables in a React application can be used to store sensitive information such as backend host, FTP host, FTP username, and FTP password.
b) Changing the value of an environment variable requires restarting the development server for the changes to take effect.
c) Vercel supports automatic deployments from a Git repository, allowing for continuous integration.
d) When migrating application files to a deployment platform, it's unnecessary to include configuration files like package.json.
e) React applications can use platform-specific options for environment variable configurations.
f) Testing the deployed application is important to ensure that it works as expected in the production environment.
g) You can directly access environment variables in React without prefixing them with REACT_APP_.
h) It is best practice to keep all environment variables in a single .env file for security purposes.
i) After configuring DNS settings, it is unnecessary to wait for DNS propagation to take effect.
j) You can create separate .env files for different environments (e.g., .env.development, .env.production) in a React application.
k) Migrating application files involves copying only the src folder from your React application.
l) After deploying a React application, you should avoid running any performance audits to ensure the app functions correctly.
m) Verifying SSL settings can be done by accessing the website and checking if the URL starts with "http://" instead of "https://".

Q2. Fill in the blank spaces with appropriate word(s). Select from the given choices in the box.

dot(.), CNAME record, .env, nslookup, REACT_APP_,A Record, npm run build, comma(,) and IP address

a) The command …… is used to create a production-ready build of a React application.
b) You can set environment variables in ReactJS by creating a …… file in the root of your project.
c) Add an …… to point your domain to the IP address of the server hosting your React application.
d) Variables should be prefixed with …… for React to recognize them
e) Use the command ……to check if your domain resolves to the correct IP address.
f) The …… is returned to the browser, which uses it to connect to the server.
g) The top of the DNS hierarchy represented by a ……. symbol.

Q3.Match term in middle column and its corresponding meaning in right column and then write the answer in the left column.

Answer	**Key term**	**Description**
1……..	1. DNS	A. It is an extension of HTTP that adds an extra layer of security.
2……..	2. SSL	B. It is a type of DNS (Domain Name System) record that maps an alias name to a true or canonical domain name.
3……..	3. HTTPS	C. It is the last part of a domain name, appearing after the final dot. It helps indicate the nature or origin of the domain.
4……	4. FTP host	D. The top of the hierarchy, represented by the dot (.) at the end of the domain.
5……	5.CNAME Record	E. It is a system that translates human-readable domain names into machine-readable IP addresses that computers use to identify each other on the network.

6…….	6.Top-Level Domain	F. It the server or system that provides the File Transfer Protocol service, allowing users to upload, download, and manage files over the internet.
7……	7.Root Level	G. It is a company that provides individuals and organizations access to the internet.
8……	8. IP address	H. It is a standard security technology used to establish an encrypted link between a server and a client, typically a web server (website) and a browser.
9……	9. ISP	I. It is the unique identifier for the account.
10…….	10. Domain registrar	J. It is a type of DNS (Domain Name System) record that maps a domain name to its corresponding IPv4 address.
		K. It is a company or organization that manages the reservation of Internet domain names.
		L. It is a unique identifier assigned to devices (such as computers, smartphones, servers) connected to a network that uses the Internet Protocol for communication.

Practical assessment

ABC company is a company located in Kigali City. It hired a React.js developer to develop a customer feedback system. The company wants to hire a ReactJS developer to deploy the application on Vercel. The client has provided a custom domain, www.abc.com, and they want the application to be accessible via this domain. Assume you are hired as a ReactJS developer and tasked to make sure application to have SSL (HTTPS) enabled for secure access. Your tasks are to:

1. Configure the DNS settings to point the custom domain to the Vercel deployment.
2. Set up SSL to ensure that the site can be accessed via HTTPS.
3. Test and verify that the domain is properly configured and secured.

END



References

Appwrk. (n.d.). *How to set up the .env file in ReactJS*. Retrieved October 14, 2024, from https://appwrk.com/reactjs-environment-variables#h-how-to-set-up-the-env-file-reactjs

Create React App. (n.d.). *Deployment*. https://create-react-app.dev/docs/deployment/

Vercel. (n.d.). *Deploying a React application with Vercel*. Vercel. https://vercel.com/guides/deploying-react-with-vercel

Vercel. (n.d.). *Add a domain*. Vercel. https://vercel.com/docs/projects/domains/add-a-domain

RTB | RWANDA TVET BOARD

October, 2024