Learning Outcome 3: Develop a Next.js Application

Next.js is a **React framework** that helps in **building modern web applications** with features like:

- Server-side rendering (**SSR**)
- Static site generation (**SSG**)
- API handling
- Built-in **routing system**
- Full **TypeScript support**

Next.js is a React framework that provides features like server-side rendering (SSR), static site generation (SSG), and API routes, making it ideal for building modern web applications.

Why Use Next.js?

- Pre-rendering for better performance and SEO
- Built-in routing system
- API routes for backend logic
- Full support for TypeScript
- Image optimization and caching

---

server-Side Rendering (SSR)

What is Server-Side Rendering (SSR)?

Server-Side Rendering (SSR) means that the page is **generated on the server every time a request is made**. Instead of sending a blank HTML file with JavaScript (like in React), Next.js sends a fully **pre-rendered page** from the server.

Key Benefits of SSR:

- Improves **SEO** (search engine optimization).
- Ensures **fresh data** because pages are built at request time.
- Better for **dynamic content** (e.g., personalized dashboards).

Example of SSR in Next.js

`pages/users.tsx`

```
export async function getServerSideProps() {
  const res = await fetch('https://jsonplaceholder.typicode.com/users');
  const users = await res.json();

  return { props: { users } };
}
```

```
export default function Users({ users }) {
  return (
    <div>
      <h1>Users List</h1>
      <ul>
        {users.map(user => (
          <li key={user.id}>{user.name}</li>
        ))}
      </ul>
    </div>
  );
}
```

How It Works:

- `getServerSideProps` runs **on the server at request time**.
- Fetches users **on every request**.
- The pre-rendered page is **sent to the client** with data included.

Visit: `http://localhost:3000/users`

The user list is fetched **in real-time** on every request.

Static Site Generation (SSG)

What is Static Site Generation (SSG)?

Static Site Generation (SSG) means that Next.js **generates the page at build time**, so it **does not need to be reloaded** from the server for every request.

Key Benefits of SSG:

Super **fast** because pages are pre-built.

Great for SEO since pages are pre-rendered.

Works well for **blogs, documentation, and marketing websites**.

Example of SSG in Next.js

`pages/blog.tsx`

```
export async function getStaticProps() {
  const res = await fetch('https://jsonplaceholder.typicode.com/posts');
  const posts = await res.json();

  return { props: { posts } };
}

export default function Blog({ posts }) {
  return (
    <div>
      <h1>Blog Posts</h1>
      <ul>
        {posts.map(post => (
          <li key={post.id}>{post.title}</li>
        ))}
      </ul>      v
    </div>
```

```
  );
}
```

How It Works:

- `getStaticProps` runs **only once at build time**.
- The data is fetched during the **build process** and saved in an HTML file.
- Pages load **instantly** because they don't need to be generated on request.

Visit: `http://localhost:3000/blog`
The blog posts load **instantly** since they were pre-built.

API Handling in Next.js

What is API Handling?

Next.js allows you to create **backend API routes** in the same project as your frontend.
Instead of using a separate backend (like Node.js or Django), you can create API routes **inside the Next.js project**.

Key Benefits of Next.js API Handling:
No need for a separate backend – Next.js provides **built-in API routes**.
Can handle **authentication, data fetching, and form submissions**.
Can be used with **SSR or SSG**.

Example of an API Route

`pages/api/hello.ts`

```
export default function handler(req, res) {
  res.status(200).json({ message: "Hello, Next.js API!" });
}
```

How It Works:

- This API is available at `http://localhost:3000/api/hello`.
- When accessed, it returns:

  ```
  { "message": "Hello, Next.js API!" }
  ```

- Can be used for **fetching or processing data**.

Example of Fetching Data from API in a Component

`pages/api-example.tsx`

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

export default function APIExample() {
  const [data, setData] = useState(null);

  useEffect(() => {
    fetch('/api/hello')
```

```
      .then(response => response.json())
      .then(data => setData(data));
  }, []);

  return <h1>{data ? data.message : 'Loading...'}</h1>;
}
```

Visit: `http://localhost:3000/api-example`

Fetches and displays: **"Hello, Next.js API!"**

Built-in Routing System

What is Next.js Routing?

Next.js provides **file-based routing**, meaning:

Every file inside the `pages/` folder becomes a route.

No need to manually set up `react-router-dom`.

Supports **dynamic routes** and **nested routes**.

Example of Basic Routing

`pages/about.tsx`

```
export default function About() {
  return <h1>About Page</h1>;
}
```

Visit: `http://localhost:3000/about`

The `about.tsx` **file automatically becomes the** `/about` **route**.

Dynamic Routes in Next.js

You can create **dynamic routes** with square brackets `[ ]`.

`pages/products/[id].tsx`

```
import { useRouter } from 'next/router';

export default function Product() {
  const router = useRouter();
  const { id } = router.query;

  return <h1>Product ID: {id}</h1>;
}
```

Visit: `http://localhost:3000/products/123`

Displays: **"Product ID: 123"**

Full TypeScript Support

What is TypeScript?

TypeScript is a strongly typed programming language built on JavaScript. It provides static type checking, improving code maintainability and reducing errors.

TypeScript is **a superset of JavaScript** that adds **type safety** to your code.

Helps **catch errors before running the code**.

Ensures **better maintainability**.

Fully supported in **Next.js**.

Example of TypeScript in Next.js

`components/User.tsx`

```
interface UserProps {
  name: string;
  age: number;
}

export default function User({ name, age }: UserProps) {
  return <h1>{name} is {age} years old.</h1>;
}
```

`pages/user-example.tsx`

```
import User from '../components/User';

export default function UserPage() {
  return <User name="Alice" age={25} />;
}
```

Visit: `http://localhost:3000/user-example`

Displays: **"Alice is 25 years old."**

Key TypeScript Features Used:

- `interface UserProps` → Defines expected properties.
- **Ensures that** `name` **is a string and** `age` **is a number**.
- **Prevents errors before running the code**.

Summary

Feature	Explanation	Example
SSR	Fetches data **at request time** on the server	Use `getServerSideProps()`
SSG	Pre-builds pages **at build time** for speed	Use `getStaticProps()`
API Handling	Creates **backend APIs** inside Next.js	Define API routes in

Feature	Explanation	Example
		`pages/api/`
Routing System	**File-based routing**, no setup needed	Create files in `pages/`
TypeScript Support	Ensures **type safety**	Use `interface` and strict typing

Setting Up Next.js with TypeScript

Step 1: Install Next.js with TypeScript

To start a Next.js project with TypeScript, run this command in your terminal:

```
npx create-next-app@latest my-next-app --typescript
cd my-next-app
npm run dev
```

What this does:

- `npx create-next-app@latest my-next-app --typescript` → Creates a Next.js project with TypeScript.
- `cd my-next-app` → Moves into your project folder.
- `npm run dev` → Starts the development server.

Your project is now running at `http://localhost:3000`

Step 2: Understanding `tsconfig.json` (TypeScript Configuration)

Next.js **automatically** detects TypeScript, but you can customize it in `tsconfig.json`:

`tsconfig.json`

```
{
  "compilerOptions": {
    "target": "es6",
    "module": "esnext",
    "strict": true,
    "jsx": "preserve",
    "baseUrl": "."
  }
}
```

What is `tsconfig.json`?

- It is a **configuration file** that controls how TypeScript compiles `.ts` files into `.js`.
- It allows customization of **target JavaScript version**, **strict type checking**, **module resolution**, etc.

What this does:

- `"target": "es6"` → Uses modern JavaScript (ES6).
- `"strict": true` → Ensures strict type checking.

- `"jsx": "preserve"` → Keeps JSX syntax for React.

Understanding TypeScript Basics in Next.js

2.1 Defining Types with Interfaces

TypeScript helps define **object structures** using **interfaces**.

What is an Interface of a Variable?

An **interface of a variable** refers to a TypeScript interface that defines the structure of an object **before assigning values**.

Example: Without an Interface (JavaScript)

```
const user = {
  name: "Alice",
  age: 25
};
```

The issue? This object **has no predefined structure**, so it can have any property.

Example: With an Interface (TypeScript)

```
interface User {
  name: string;
  age: number;
}

const user: User = {
  name: "Alice",
  age: 25
};
```

Now, `user` **must follow the** `User` **interface**, ensuring it always has `name` (string) and `age` (number).

`components/User.tsx`

```
interface User {
  id: number;
  name: string;
  email?: string; // Optional property
}

const user: User = { id: 1, name: "John Doe", email: "john@example.com" };

export default function UserProfile() {
  return <h1>User: {user.name}</h1>;
}
```

What this does:

- `interface User` → Defines an object structure.
- `id: number` → `id` must be a number.
- `email?: string` → `email` is optional (`?` means optional).

- **Renders a simple user profile** on the page.

2.2 Writing Functions in TypeScript

TypeScript ensures correct function inputs and outputs.

In TypeScript, functions allow you to define reusable blocks of code with **static typing**. This means you can specify **parameter types** and **return types**, making your code more predictable and less error-prone.

`utils/calculate.ts`

```
const add = (a: number, b: number): number => {
  return a + b;
};

export default add;
```

Explanation:

- `(a: number, b: number): number` → Takes two numbers and returns a number.

Now you can use this function in any component:

`components/Calculator.tsx`

```
import add from '../utils/calculate';

export default function Calculator() {
  return <h1>Sum: {add(5, 10)}</h1>;
}
```

This will display: **Sum: 15**

What is Data Handling?

Data Handling refers to the process of **collecting, processing, validating, and managing data** in a structured and efficient way. In **TypeScript and Next.js**, data handling is crucial for working with APIs, forms, databases, and state management.

Key Aspects of Data Handling

1.1 Data Collection

Data is collected from various sources like:

APIs (REST or GraphQL)
User input (Forms, Inputs, etc.)
Files or Databases

1.2 Data Validation

Ensuring data is correct before processing it.
Example: TypeScript Interface for Validation

```
interface User {
  name: string;
  age: number;
}

function validateUser(user: User): boolean {
  return typeof user.name === "string" && typeof user.age === "number";
}

console.log(validateUser({ name: "Alice", age: 25 })); // true
console.log(validateUser({ name: "Bob", age: "twenty" })); // false
```

Ensures data has the correct type before using it.

1.3 Form Handling

Capturing user input in forms and validating before submission.

Example: Handling Forms in Next.js with TypeScript

```
import { useState } from "react";

export default function FormExample() {
  const [name, setName] = useState<string>("");
  const [age, setAge] = useState<number | "">("");

  const handleSubmit = (e: React.FormEvent) => {
    e.preventDefault();
    if (name && age) {
      console.log(`User: ${name}, Age: ${age}`);
    } else {
      console.log("Please fill in all fields.");
    }
  };

  return (
    <form onSubmit={handleSubmit}>
      <input type="text" placeholder="Name" onChange={(e) =>
setName(e.target.value)} />
      <input type="number" placeholder="Age" onChange={(e) =>
setAge(Number(e.target.value))} />
      <button type="submit">Submit</button>
    </form>
  );
}
```

Uses `useState` to manage form data.

Validates that name and age are not empty before submission.

1.4 API Data Handling

Fetching data from an API and validating it.

Example: Fetching Data from an API in Next.js

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

interface Post {
  id: number;
  title: string;
  body: string;
}
```

```
export default function Posts() {
  const [posts, setPosts] = useState<Post[]>([]);

  useEffect(() => {
    fetch("https://jsonplaceholder.typicode.com/posts")
      .then((response) => response.json())
      .then((data: Post[]) => setPosts(data));
  }, []);

  return (
    <div>
      <h1>Posts</h1>
      {posts.map((post) => (
        <div key={post.id}>
          <h2>{post.title}</h2>
          <p>{post.body}</p>
        </div>
      ))}
    </div>
  );
}
```

Uses TypeScript to define the `Post` interface.

Fetches data using `useEffect` and updates state.

1.5 Error Handling & Exceptions

Handling errors to prevent crashes.

Example: Handling API Errors

```
async function fetchData() {
  try {
    const response = await fetch("https://jsonplaceholder.typicode.com/posts");
    if (!response.ok) throw new Error("Network response was not ok");
    const data = await response.json();
    return data;
  } catch (error) {
    console.error("Error fetching data:", error);
    return [];
  }
}
```

Uses `try...catch` to handle network errors.

Prevents app crashes by returning an empty array on failure.

Why is Data Handling Important?

Prevents errors: Validating data ensures type correctness.

Improves security: Proper handling prevents **XSS, SQL Injection, etc.**

Enhances user experience: Prevents crashes and incorrect data submissions.

Optimizes performance: Caching and error handling improve efficiency.

Environment Preparation

Before creating a Next.js project, ensure you have the necessary tools installed.

1.1 Install Node.js and npm/yarn

Next.js requires **Node.js** and **npm** (or **yarn**). You can check if Node.js is installed by running:

```
node -v
```

If Node.js is not installed, download it from nodejs.org and install it.

After installation, verify **npm** or **yarn**:

```
npm -v
```

or

```
yarn -v
```

Creating a Next.js Project

There are two main ways to create a Next.js project:

2.1 Using `create-next-app` (Recommended)

Run the following command to create a Next.js app with TypeScript:

```
npx create-next-app@latest my-next-app --typescript
```

or using yarn:

```
yarn create next-app my-next-app --typescript
```

This automatically sets up Next.js with TypeScript and all necessary configurations.

If you want a JavaScript project, remove `--typescript`:

```
npx create-next-app@latest my-next-app
```

2.2 Manually Setting Up Next.js

If you prefer manual setup, follow these steps:

Step 1: Create a New Directory and Initialize a Node.js Project

```
mkdir my-next-app
cd my-next-app
npm init -y
```

Step 2: Install Next.js, React, and ReactDOM

```
npm install next react react-dom
```

Step 3: Configure `package.json`

Modify `package.json` to include the Next.js scripts:

```
"scripts": {
  "dev": "next dev",
  "build": "next build",
  "start": "next start"
```

```
}
```

Initial Project Structure

A **Next.js** project typically includes the following structure:

```
my-next-app/
|— node_modules/            # Installed dependencies
|— public/                  # Static assets like images
|— pages/                   # Next.js pages (routes)
|    ├— index.tsx           # Home page
|    ├— about.tsx           # About page
|    ├— _app.tsx            # Custom App component
|    ├— _document.tsx       # Custom Document component
|— components/              # Reusable UI components
|— styles/                  # CSS files
|— next.config.js           # Next.js configuration
|— tsconfig.json            # TypeScript configuration
|— package.json             # Project metadata and scripts
|— .gitignore               # Files to ignore in Git
|— README.md                # Project documentation
```

Running the Next.js Application

Once setup is complete, start the development server:

`npm run dev`

Open **http://localhost:3000/** to see your Next.js app running.

1. Creating Pages and Components

Next.js uses a **file-based routing system**, where each `.tsx` or `.js` file inside the `pages/` directory automatically becomes a route.

Steps to Create Pages:

1. **Go to the `pages/` directory** and create a new file:

```
// pages/index.tsx
export default function HomePage() {
  return <h1>Welcome to My Next.js App!</h1>;
}
```

Running `npm run dev` will make this available at `http://localhost:3000/`.

2. **Create additional pages:**

```
// pages/about.tsx
export default function AboutPage() {
  return <h1>About Us</h1>;
}
```

Now, visiting `http://localhost:3000/about` will render this page.

2. Implementing Search Engine Optimization (SEO)

SEO is important for visibility. Next.js provides a built-in `<Head>` component to manage metadata.

Adding Metadata for SEO:

```
// pages/index.tsx
import Head from "next/head";

export default function HomePage() {
  return (
    <>
      <Head>
        <title>My Next.js App</title>
        <meta name="description" content="A beginner-friendly Next.js app" />
      </Head>
      <h1>Welcome to My Next.js App!</h1>
    </>
  );
}
```

This helps search engines properly index the page.

---

3. Styling in Next.js

There are multiple ways to style your Next.js app:

A) Using CSS Modules

```
// styles/Home.module.css
.title {
  color: blue;
  font-size: 24px;
}

// pages/index.tsx
import styles from "../styles/Home.module.css";

export default function HomePage() {
  return <h1 className={styles.title}>Welcome to My Next.js App!</h1>;
}
```

B) Using Tailwind CSS (Recommended)

1. **Install Tailwind CSS:**

   ```
   npm install -D tailwindcss postcss autoprefixer
   npx tailwindcss init -p
   ```

2. **Configure** `tailwind.config.js`**:**

```
module.exports = {
  content: ["./pages/**/*.{js,ts,jsx,tsx}", "./components/**/*.
{js,ts,jsx,tsx}"],
  theme: { extend: {} },
  plugins: [],
};
```

3. **Add Tailwind to global styles (`styles/globals.css`):**

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

4. **Use Tailwind classes in your components:**

```
export default function HomePage() {
  return <h1 className="text-3xl font-bold text-blue-500">Welcome!</h1>;
```

4. Caching Strategies for Performance

Caching improves speed and performance in Next.js.

- **Static Site Generation (SSG)** (Pre-render pages at build time)
- **Server-Side Rendering (SSR)** (Fetch data on every request)
- **Incremental Static Regeneration (ISR)** (Update static pages without rebuilding the whole site)
- **Client-Side Rendering (CSR)** (Load data in the browser)

Example of ISR in Next.js:

```
export async function getStaticProps() {
  return {
    props: { message: "Hello World" },
    revalidate: 10, // Rebuild page every 10 seconds
  };
}
```

Next Steps

After setting up the initial development, you can move on to:

- **Routing & Navigation**
- **API Routes**
- **Security & Authentication**
- **Deployment**

Creating Pages and Components

3.1 Creating a Basic Page

Every file inside `pages/` becomes a **route** automatically.

`pages/index.tsx`

```
export default function Home() {
  return <h1>Welcome to Next.js!</h1>;
}
```

Visit `http://localhost:3000` and you'll see:

Welcome to Next.js!

3.2 Creating a Reusable Component

`components/Button.tsx`

```
interface ButtonProps {
  text: string;
}

export default function Button({ text }: ButtonProps) {
  return <button>{text}</button>;
}
```

`pages/about.tsx`

```
import Button from '../components/Button';

export default function About() {
  return <div><Button text="Click Me!" /></div>;
}
```

Visit `http://localhost:3000/about`

You will see a **button labeled "Click Me!"**

Implementing SEO

Next.js has a built-in `Head` component for SEO (Search Engine Optimization).

`pages/contact.tsx`

```
import Head from 'next/head';

export default function Contact() {
  return (
    <>
      <Head>
        <title>Contact Us</title>
        <meta name="description" content="Get in touch with us." />
      </Head>
      <h1>Contact Page</h1>
    </>
  );
}
```

This improves SEO by adding metadata to the page.

Implementing Rendering Techniques

5.1 Static Site Generation (SSG)

`pages/blog.tsx`

```
export async function getStaticProps() {
  const res = await fetch('https://jsonplaceholder.typicode.com/posts');
  const posts = await res.json();
```

```
  return { props: { posts } };
}

export default function Blog({ posts }) {
  return <h1>Blog Posts: {posts.length}</h1>;
}
```

Pre-builds the page at **build time** for fast loading.

5.2 Server-Side Rendering (SSR)

`pages/users.tsx`

```
export async function getServerSideProps() {
  const res = await fetch('https://jsonplaceholder.typicode.com/users');
  const users = await res.json();

  return { props: { users } };
}
```

Fetches data on each request, ensuring fresh content.

Creating API Routes

6.1 Defining API Endpoints

`pages/api/hello.ts`

```
export default function handler(req, res) {
  res.status(200).json({ message: "Hello API" });
}
```

Visit `http://localhost:3000/api/hello`

You will see: `{ "message": "Hello API" }`

6.2 Using Dynamic API Routes

`pages/api/user/[id].ts`

```
export default function handler(req, res) {
  const { id } = req.query;
  res.status(200).json({ userId: id });
}
```

Visit `http://localhost:3000/api/user/5`

You will see: `{ "userId": "5" }`

Securing the Application

7.1 Client-Side Security

CORS – Prevents unauthorized API access.
Session Management – Secures user sessions.
Third-Party Auth (Auth0, Firebase, etc.) – Manages authentication.

7.2 Server-Side Security

HTTPS Enforcement – Ensures secure connections.
API Route Security – Protects endpoints with authentication.
Content Security Policy (CSP) – Prevents cross-site scripting attacks.

`next.config.js`

```
module.exports = {
  headers: async () => [
    {
      source: "/(.*)",
      headers: [{ key: "Content-Security-Policy", value: "default-src
'self'" }],
    },
  ],
};
```

Conclusion

Next.js simplifies React development with **built-in routing, API handling, and SEO optimization**.
TypeScript ensures **type safety** and better maintainability.
SSG, SSR, ISR, CSR improve performance and flexibility.
Security practices like **CORS, CSP, and authentication** keep apps secure.

Implementing Routing in Next.js

Routing in Next.js is built on a **file-based routing system**, making it simple to create dynamic and nested routes. This section covers key concepts, from basic routing to advanced features like dynamic and API routes.

3.1 Key Concepts in Routing

3.1.1 File-System Based Routing

- In Next.js, each file inside the `pages/` directory automatically becomes a route.
- Example:

```
// pages/index.tsx
export default function HomePage() {
  return <h1>Home Page</h1>;
```

```
}
```

- Accessible at: `http://localhost:3000/`

```
// pages/about.tsx
export default function AboutPage() {
  return <h1>About Us</h1>;
}
```

- Accessible at: `http://localhost:3000/about`

3.1.2 Dynamic Routes

- Use square brackets `[ ]` in filenames to create dynamic routes.

```
// pages/user/[id].tsx
import { useRouter } from "next/router";

export default function UserPage() {
  const router = useRouter();
  const { id } = router.query;
  return <h1>User ID: {id}</h1>;
}
```

- Visiting `http://localhost:3000/user/5` will show "**User ID: 5**".

3.1.3 Nested Routes

- Simply create a folder structure inside `pages/`.

```
pages/
├── blog/
│   ├── index.tsx  // Accessible at /blog
│   ├── [postId].tsx // Accessible at /blog/postId
```

3.1.4 Link Component (Client-side Navigation)

- Use Next.js's `<Link>` component for fast navigation.

```
import Link from "next/link";

export default function HomePage() {
  return (
    <div>
      <h1>Home</h1>
      <Link href="/about">Go to About Page</Link>
    </div>
  );
}
```

- This provides **faster navigation without a full page reload**.

3.1.5 Programmatic Navigation

- Use `useRouter()` from `next/router` to navigate programmatically.

```
import { useRouter } from "next/router";

export default function Dashboard() {
  const router = useRouter();

  return (
    <button onClick={() => router.push("/profile")}>
      Go to Profile
    </button>
  );
}
```

  - Clicking the button will **navigate to** `/profile`.

3.1.6 API Routes

- Next.js allows you to create **backend API endpoints** inside the `pages/api/` directory.

```
// pages/api/hello.ts
export default function handler(req, res) {
  res.status(200).json({ message: "Hello from API!" });
}
```

  - Access it at `http://localhost:3000/api/hello`.

3.1.7 Catch-All Routes

- Use `[...params].tsx` to handle multiple dynamic segments.

```
// pages/docs/[...slug].tsx
import { useRouter } from "next/router";

export default function DocsPage() {
  const router = useRouter();
  const { slug } = router.query;

  return <h1>Viewing Docs: {slug?.join("/")}</h1>;
}
```

  - Visiting `/docs/javascript/advanced` will display: **"Viewing Docs: javascript/advanced"**.

3.2 API Creation in Next.js

3.2.1 Defining an API Endpoint

- API routes live in `pages/api/` and act as backend functions.

```
// pages/api/greet.ts
export default function handler(req, res) {
  res.status(200).json({ message: "Hello, Next.js API!" });
}
```

3.2.2 Handling Request Types

- Handle **GET, POST, PUT, DELETE** requests inside API routes.

```
// pages/api/user.ts
export default function handler(req, res) {
  if (req.method === "GET") {
    res.status(200).json({ message: "Fetching user data" });
  } else if (req.method === "POST") {
    res.status(201).json({ message: "Creating a new user" });
  } else {
    res.status(405).json({ message: "Method Not Allowed" });
  }
}
```

3.2.3 Using Dynamic API Routes

- Create dynamic API routes using square brackets `[ ]`.

```
// pages/api/user/[id].ts
export default function handler(req, res) {
  const { id } = req.query;
  res.status(200).json({ message: `Fetching user with ID: ${id}` });
}
```

    - Access it at `/api/user/123`.

3.2.4 Testing Your API

- Use **Postman**, **cURL**, or **your browser** to test API routes.
- Example request:

```
curl -X GET http://localhost:3000/api/user/5
```

3.3 Securing the Application

3.3.1 Performing Client-Side Security

Client-Side Rendering (CSR) Security

- Avoid exposing sensitive data in frontend code.

Cross-Origin Resource Sharing (CORS)

- Prevent unauthorized domains from accessing your API.

```
// Install CORS package
npm install cors
```

Session Management

- Use **JWT tokens** or **NextAuth.js** for authentication.

Third-Party Libraries (Auth0, Firebase)

- Secure authentication flows with **OAuth**.

3.3.2 Performing Server-Side Security

HTTPS Enforcement

- Always use HTTPS in production.

Server-Side Rendering (SSR) Security

- Avoid exposing sensitive data via `getServerSideProps()`.

API Routes Security

- Restrict access using middleware.

```
// Middleware example
export default function handler(req, res) {
  if (!req.headers.authorization) {
    return res.status(403).json({ message: "Unauthorized" });
  }
  res.status(200).json({ message: "Success" });
}
```

Content Security Policy (CSP)

- Prevent **XSS attacks** by defining CSP headers.

Authentication & Authorization

- Use **JWT tokens**, **OAuth**, or **NextAuth.js**.

3.3.3 Performing General Security Measures

Sanitize user input to prevent **SQL Injection & XSS attacks**.
Use environment variables (`.env.local`) to store API keys securely.
Enable logging and monitoring with **Sentry** or **Datadog**.

Next Steps

Now that you've set up routing, API creation, and security, the next steps include:

- **Fetching & Managing Data** (SSG, SSR, ISR, CSR)
- **Authentication with NextAuth.js**
- **Deployment to Vercel or AWS**

Learning Outcome 4: Apply Progressive Web Application

4.1 Maintain Responsiveness

To ensure a Progressive Web Application (PWA) delivers an optimal user experience across all devices, maintaining responsiveness is essential. This involves:

4.1.1 Leverage Progressive Enhancement

Progressive enhancement is a web design strategy that ensures basic content and functionality are accessible to all users, while advanced features are provided to those with modern browsers. It involves:

- Building a solid HTML foundation
- Enhancing with CSS for better styling
- Adding JavaScript for interactive and dynamic features
- Ensuring core functionalities work even if JavaScript fails

example on server-side rendering

```javascript
export async function getServerSideProps() {
  const data = await fetchData();
  return { props: { data } };
}
```

Part	Meaning
`export async function getServerSideProps()`	This is a **special function** that Next.js looks for. It tells Next.js to **run this function on the server** *every time someone visits the page*.
`await fetchData();`	It **fetches some data** — usually from an API, database, or another service — **while still on the server**.
`return { props: { data } };`	After fetching, it **returns an object**. The object must have a `props` key. The `props` will then be **passed into your page component** as props.

4.1.2 Prioritize Mobile-First Design

Mobile-first design ensures that web applications are designed for mobile users first, then enhanced for larger screens. This approach includes:

- Designing layouts for smaller screens before scaling up
- Using responsive units like percentages, `em`, and `rem`
- Implementing flexible grid systems (e.g., CSS Grid, Flexbox)
- Testing designs on different screen sizes

javascript Copy Edit

```javascript
function EnhancedForm() {
  const handleSubmit = (e) => {
    e.preventDefault();
    // AJAX submission logic
  };

  return (
    <form action="/submit" method="POST" onSubmit={handleSubmit}>
      <input name="name" required />
      <button type="submit">Submit</button>
    </form>
  );
}
```

Part	Meaning
`function EnhancedForm()`	This defines a **React functional component** called `EnhancedForm`.
`const handleSubmit = (e) => {}`	This is a **handler function** that will be called **when the form is submitted**.
`e.preventDefault();`	**Stops the browser's default behavior** of refreshing or navigating away when submitting a form. This is important because you want to handle the form submission with **JavaScript (AJAX)** instead.
`// AJAX submission logic`	Here, you would normally **send the form data to the server manually** using `fetch()` or `axios`. (It's left as a comment in your code.)
`<form action="/submit" method="POST" onSubmit={handleSubmit}>`	This is the actual **form HTML**. It says: "If JS is active, use `handleSubmit`". Otherwise, if JS is broken, **the form will submit normally** to `/submit`.
`<input name="name" required />`	A simple input field for the user to type their name. It's marked **required**, so the browser will **validate** it before submission.
`<button type="submit">Submit</button>`	A **submit button** that triggers the form's `onSubmit` event.

Big Picture:

- **If JavaScript is working**:
    - Clicking "Submit" will trigger `handleSubmit()`.

- `preventDefault()` will stop the browser from reloading.
- Then you can manually send the form data via AJAX (e.g., `fetch('/submit', { method: 'POST', body: ... })`).

- **If JavaScript is disabled or broken**:
  - The browser will still **submit the form normally** to `/submit` using the traditional POST method.

This is Progressive Enhancement!
The form works without JS and becomes fancier when JS is available.

4.1.3 Utilize Performance Optimization Techniques

Performance optimization is critical for enhancing user experience and reducing load times. Key techniques include:

- **Minimizing HTTP requests** by combining CSS and JavaScript files
- **Using lazy loading** for images and content
- **Optimizing images** by compressing and using next-gen formats like WebP
- **Enabling Gzip or Brotli compression** for faster data transfer
- **Reducing render-blocking resources** with asynchronous loading (`async` and `defer`)

1. Code Splitting

- **Break big JS bundles into smaller chunks**.
- Only **load what is needed** on a particular page.

Use:

```
javascript
CopyEdit
import dynamic from 'next/dynamic';

const HeavyComponent = dynamic(() => import('./HeavyComponent'));
```

`What it means:`

`dynamic()` is a **function from Next.js** that allows **dynamic importing** of a component.

Instead of importing the component at the top (like `import HeavyComponent from './HeavyComponent'`), it **only loads** the component **when it's actually needed**.

Why would you do this?

Because if the `HeavyComponent` is **big** (maybe lots of JS, animations, charts, etc.), **you don't want to load it immediately** when the page loads.
Instead, **you delay loading it** until it's really needed, making your page **faster to load initially**.

In short:

- *Without* `dynamic()`: the component is loaded immediately when the page loads (even if not used yet).
- *With* `dynamic()`: it loads later *only when the component is needed*.

Without dynamic import:

```
import HeavyComponent from './HeavyComponent';

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

This means HeavyComponent is **always bundled** and **always loaded immediately**.

With dynamic import:

```
import dynamic from 'next/dynamic';

const HeavyComponent = dynamic(() => import('./HeavyComponent'));

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

Now **the initial page load is faster** because `HeavyComponent` is **loaded separately**.

2. Lazy Loading Components / Images

- **Don't load everything at once**.
- Load **components and images only when they are needed** (like when they come into view).

Example for images:

```
html
```

```
CopyEdit
<img src="small-placeholder.jpg" data-src="actual-big-image.jpg"
loading="lazy" />
```

For components: (use `React.lazy` or `dynamic()` in Next.js)

---

3. Memoization

- **Prevent unnecessary re-renders** of components.

Use:

```
javascript
CopyEdit
import { memo } from 'react';

const MyComponent = memo(function MyComponent({ data }) {
  // Only re-renders if props change
});
```

For functions inside components:

```
javascript
CopyEdit
const handleClick = useCallback(() => {
  console.log('Clicked');
}, []);
```

For heavy calculations:

```
javascript
CopyEdit
const expensiveValue = useMemo(() => computeHeavyCalculation(data), [data]);
```

---

4. Server-Side Rendering (SSR) or Static Site Generation (SSG)

- **Pre-render pages** on the server whenever possible.
- Tools: **Next.js**, **Remix**.

SSR = always fresh, good for dynamic content.
SSG = super fast, good for blogs or mostly static pages.

---

5. Optimize Context and State Management

- **Don't put huge amounts of state into React Context** unless absolutely necessary.
- Prefer **lightweight state** in components, **lift up only what's needed**, and **use libraries like Zustand, Jotai, Recoil** if state gets complex.

Example:

- Use small, **local component state** when possible.

- Only **globalize** when you **must share** between many parts.

---

6. Use Keyed Lists Properly

- Always give lists **proper unique `key` props**.
- This helps React efficiently re-render lists without breaking things.

Good:

```
javascript
CopyEdit
{items.map(item => <li key={item.id}>{item.name}</li>)}
```

Bad:

```
javascript
CopyEdit
{items.map((item, index) => <li key={index}>{item.name}</li>)}
```

4.2 Configuring Web Application Manifest

A web application manifest is a JSON file that provides metadata about the PWA, enabling it to function like a native app.

What is a Web Application Manifest?

It’s a simple **JSON file** (`manifest.json`) that tells the browser about your **web app's settings** — like:

- The app’s **name**
- **Icons** to use
- **Theme colors**
- How it should behave when "installed" on a device (like a mobile home screen)

It is **essential** if you want your site to behave like a **PWA** (Progressive Web App) — installable, full-screen, offline-capable, etc.

4.2.1 Creating and Configuring the Manifest File

The manifest file (`manifest.json`) includes:

- **Name and short name** for display on the home screen
- **Icons** in multiple sizes for different devices
- **Theme and background colors** for consistent UI styling
- **Start URL** to define the first page when launched from a shortcut
- **Display mode** (`standalone`, `fullscreen`, `minimal-ui`, `browser`)

How to Configure a Web App Manifest

---

1. Create a `manifest.json` file

Create a file called `manifest.json` in your `public/` folder (in Next.js, React, etc.).

Example `manifest.json` file:

```
json
CopyEdit

{

  "name": "My PWA",

  "short_name": "PWA",

  "icons": [

    {

      "src": "icon.png",

      "sizes": "192x192",

      "type": "image/png"

    }

  ],

  "theme_color": "#000000",

  "background_color": "#ffffff",

  "display": "standalone",

  "start_url": "/index.html"

}
```

4.2.2 Referencing the Manifest in Your HTML

In your `index.html` (for React) or `_document.js` (for Next.js), add:

```
<link rel="manifest" href="/manifest.json" />
```

This tells the browser: "Hey, here's the app manifest for this site."

Example for Next.js in `_app.js`:

```
import Head from 'next/head';

function MyApp({ Component, pageProps }) {
  return (
    <>
      <Head>
        <link rel="manifest" href="/manifest.json" />
      </Head>
      <Component {...pageProps} />
    </>
  );
}

export default MyApp;
```

What this code is doing:

Part	Meaning
`import Head from 'next/head';`	Imports Next.js's special `<Head>` component, which lets you customize the HTML `<head>` section (like adding meta tags, links, titles, etc.).
`function MyApp({ Component, pageProps })`	Defines the **custom App** in Next.js. This wraps **every page** in your app with some shared layout or logic.
`<Head>` inside `return`	Adds a `<link>` tag to include your `manifest.json` file (which helps make your app a PWA).
`<Component {...pageProps} />`	Renders whatever page the user requested (like `Home`, `About`, etc.), passing along the right props.
`export default MyApp;`	Exports the `MyApp` component so Next.js can use it to wrap all your pages.

4.2.3 Testing and Validation

To ensure the manifest file is correctly configured:

You can **test your manifest** by:

- Opening Chrome
- Visiting your site
- Opening **DevTools → Application Tab → Manifest Section**

You should see all your settings there!

- Use **Chrome DevTools** (`Application > Manifest`) to inspect the manifest
-

4.3 Implementation of Service Workers

A service worker is a JavaScript file that runs in the background, handling caching, push notifications, and offline functionality.

4.3.1 Describe Service Workers

Service Workers in React (and in general web applications) are a type of web worker that run in the background, separate from the main browser thread.

Service workers act as a proxy between the web application and the network. They:

- Enable offline access by caching resources
- Improve performance by loading assets from cache
- Enable background sync and push notifications

4.3.2 Registration and Installation

To register a service worker, add the following in your `app.js`:

```
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('/service-worker.js')
    .then(reg => console.log('Service Worker Registered', reg))
    .catch(err => console.log('Service Worker Registration Failed', err));
}
```

`explanation (line by line)`

`if ('serviceWorker' in navigator)`

- Checks if the **browser supports Service Workers**.
- Prevents errors in browsers that don’t support this feature.

`navigator.serviceWorker.register('/service-worker.js')`

- **Registers** the service worker located at `/service-worker.js`.
- This file must be in the **public root directory**, not `src`, because service workers require a **scope** that matches their location.

`.then(reg => console.log('Service Worker Registered', reg))`

- Runs if registration is successful.
- Logs the registration object, which contains info about the service worker.

`.catch(err => console.log('Service Worker Registration Failed', err))`

- Runs if registration fails (e.g., file not found, wrong scope).
- Logs the error.

4.3.3 Caching Strategy Implementation

Caching strategies determine how service workers handle cached content:

- **Cache-first:** Loads resources from the cache first, then updates if needed
- **Network-first:** Tries fetching from the network before using cached data
- **Stale-while-revalidate:** Uses cache while updating in the background

Strategy	Used For	Behavior
Cache First	Static assets	Loads from cache if available; otherwise fetches
Network First	HTML pages	Tries network first; falls back to cache or offline page
Stale While Revalidate	API, dynamic content	Loads cache instantly, then updates from network

Example of caching static assets in `service-worker.js`:

```
self.addEventListener('install', event => {
  event.waitUntil(
    caches.open('v1').then(cache => {
      return cache.addAll([
        '/',
        '/index.html',
        '/styles.css',
        '/script.js'
      ]);
    })
  );
});
```

4.3.4 Updating Service Worker

Updating a Service Worker properly is critical to ensure users get the latest version of your app without being stuck with old cached data.

To update a service worker, listen for the `activate` event and remove old caches:

```
self.addEventListener('activate', event => {
  event.waitUntil(
    caches.keys().then(keys => {
      return Promise.all(
        keys.filter(key => key !== 'v1').map(key => caches.delete(key))
      );
    })
  );
});
```

Forcing service worker updates:

```
navigator.serviceWorker.getRegistrations().then(registrations => {
  for (let registration of registrations) {
    registration.update();
  }
});
```

Learning Outcome 5: Publish the Application

Publishing a web application involves configuring environment variables, deploying the application to a hosting platform, and setting up a custom domain with DNS and SSL settings.

5.1 Configuration of Environment Variables

Environment variables store sensitive information such as API keys, database credentials, and platform-specific settings. Proper configuration ensures security, scalability, and seamless deployment.

5.1.1 Set Variables (Backend Host, FTP Host, FTP User, FTP Password, etc.)

- **Backend Host**: Defines the backend API URL used by the frontend
- **FTP Host**: Specifies the server location for file storage
- **FTP User & FTP Password**: Used for authentication when accessing the file server
- **Other Secrets**: API keys, database credentials, third-party service keys

Example `.env` file:

```
REACT_APP_API_URL=https://api.myapp.com
FTP_HOST=ftp.myserver.com
FTP_USER=myftpuser
FTP_PASS=myftppassword
```

Environment variables should **never be hardcoded** in the source code.

5.1.2 Setup Storage Environment

Storage configurations depend on the platform:

- **Cloud Storage (AWS S3, Firebase Storage, Azure Blob Storage)**
- **Local Storage (for development only)**
- **Database Storage (PostgreSQL, MySQL, MongoDB, etc.)**

Example storage configuration in `.env`:

```
STORAGE_BUCKET=myapp-bucket
STORAGE_REGION=us-east-1
```

5.1.3 Platform-Specific Options

Different hosting platforms (Vercel, Netlify, AWS, Heroku) may require additional settings. Examples:

- **Vercel**: Environment variables are set in the Vercel dashboard
- **Netlify**: Uses `netlify.toml` for build settings
- **Heroku**: Uses `heroku config:set` to manage environment variables

5.1.4 Separate `.env` Files

For security and better organization, different environments should have separate `.env` files:

- `.env.development` – Local development settings
- `.env.staging` – Staging/testing settings
- `.env.production` – Production settings

Example usage in a React app (`package.json`):

```
"scripts": {
  "start": "react-scripts start",
  "build": "react-scripts build --env=production"
}
```

5.2 Deploying React Application

Deployment involves building the React application, selecting a hosting platform, migrating the files, and testing the live version.

5.2.1 Run Build Script in React Application

To prepare the React app for deployment, generate optimized production files:

```
npm run build
```

This creates a `build/` folder containing minified static files (`index.html`, `.js`, `.css`).

5.2.2 Configure Deployment Platform (Vercel, Netlify, etc.)

Using Vercel:

1. Install Vercel CLI:

   ```
   npm install -g vercel
   ```

2. Deploy using CLI:

   ```
   vercel
   ```

3. Follow prompts to configure the deployment

Using Netlify:

1. Install Netlify CLI:

   ```
   npm install -g netlify-cli
   ```

2. Deploy using CLI:

   ```
   netlify deploy --prod
   ```

3. Alternatively, connect GitHub repo to Netlify for automatic deployments

Using GitHub Pages:

1. Install the `gh-pages` package:

   ```
   npm install gh-pages --save-dev
   ```

2. Add deployment script to `package.json`:

```
"homepage": "https://username.github.io/myapp",
"scripts": {
  "predeploy": "npm run build",
  "deploy": "gh-pages -d build"
}
```

3. Deploy:

```
npm run deploy
```

5.2.3 Migrate the Application Files

After building the React app, migrate the files to the hosting platform:

- **Vercel** & **Netlify**: Automatically handled when linked to a GitHub repository
- **FTP/SFTP**: Manually upload the `build/` folder to a server
- **Heroku**: Push the build to Heroku using `git push heroku main`

5.2.4 Test the Deployed Application

- Open the deployed URL and verify that the app loads correctly
- Check for **broken links, missing images, or styling issues**
- Use **Chrome DevTools** (`F12 > Console`) to identify any errors
- Test different devices and browsers

5.3 Setup Custom Domain

After deployment, configure a custom domain for branding and professionalism.

5.3.1 Description of DNS

The **Domain Name System (DNS)** translates human-readable domain names (e.g., `myapp.com`) into IP addresses (`192.168.1.1`).

Translation of Domain Names to IP Addresses

- When a user enters `myapp.com`, a DNS lookup retrieves the corresponding IP
- The browser then connects to the server hosting the website

Hierarchy and Structure of DNS

1. **Root DNS Servers (`.`)** – Top-level servers
2. **Top-Level Domain (TLD) Servers (`.com, .org`)** – Manage domain extensions
3. **Authoritative DNS Servers** – Store records for specific domains

Name Resolution Process:

1. User enters `myapp.com`
2. Browser queries the DNS resolver
3. Resolver contacts the authoritative DNS server

4. IP address is returned and the website loads

5.3.2 Configure DNS and SSL Settings

Step 1: Purchase a Domain

- Register a domain via **Namecheap, GoDaddy, Google Domains**

Step 2: Set Up DNS Records

- **A Record**: Points the domain to an IP address
- **CNAME Record**: Maps subdomains (`www.myapp.com`) to another domain
- **TXT Record**: Used for verification (e.g., Google Search Console)

Example **A Record** setup:

Type	Name	Value (IP Address)
A	@	192.168.1.100
CNAME	www	myapp.com

Step 3: Configure SSL for HTTPS

- **Let's Encrypt (Free SSL)**:

```
certbot --nginx -d myapp.com -d www.myapp.com
```

- **Vercel/Netlify:** Automatic SSL setup
- **Cloudflare:** Provides free SSL

5.3.3 Testing and Verification

- Use `nslookup myapp.com` to verify DNS resolution
- Check SSL certificate validity using **https://www.sslshopper.com**
- Test HTTPS redirection in the browser

END