Module 05

Purwadhika
Digital Technology School

AI Fullstack Software Development

Server-Side Rendering, SEO Optimization, and PWA Development

Job Connector Program

Outline

Purwadhika
Digital Technology School

- Server-Side Rendering
- Introduction to SEO Optimization
- Setup Progressive Web Apps (PWA)

Introduction to Server-Side Rendering (SSR)

Server Side Rendering (SSR) is a technique where the HTML of a web page is generated on the server instead of the client browser.

- The server pre-renders the React components into HTML.
- The client receives fully rendered HTML ➜ faster first paint.
- Then React hydrates the page on the client for interactivity.

CSR vs SSR vs SSG

Method	Render Location	When Rendered	Pros	Cons
CSR (Client Side Rendering)	Browser	After JS loads	Great for interactivity	Poor SEO, slow first render
SSR (Server Side Rendering)	Server	On each request	SEO-friendly, fast load	Higher server cost
SSG (Static Site Generation)	Build time	Before deployment	Super fast delivery	Content not dynamic

How SSR Works in React with Vite

- Client sends a request ➜ e.g. /about
- Server runs a render function using ReactDOMServer.renderToString()
- The HTML is returned to the browser
- React hydrates the HTML to make it interactive (hydration = attaching React event handlers to static HTML)

Setting Up React + Vite for SSR

Create Project

```
npm create vite@latest react-ssr-demo --template react-ts
cd react-ssr-demo
npm install
```

Create a file: **src/entry-server.tsx**

```
import React from 'react';
import ReactDOMServer from 'react-dom/server';
import App from './App';

export function render() {
  return ReactDOMServer.renderToString(<App />);
}
```

Creating the Express Server

Add a simple Node.js/Express server to handle SSR.

```
import express from 'express';
import fs from 'fs';
import path from 'path';
import { render } from './src/entry-server';

const app = express();

app.use('*', async (req, res) => {
  const html = render();
  const template = fs.readFileSync(
    path.resolve('index.html'), 'utf-8'
  );

  const finalHtml = template.replace(`<!--app-->`, html);
  res.send(finalHtml);
});

app.listen(5173, () => console.log('SSR server running on http://localhost:5173'));
```

Vite Configuration for SSR

Modify vite.config.ts to enable SSR mode.

```
import { defineConfig } from 'vite';
import react from '@vitejs/plugin-react';

export default defineConfig({
  plugins: [react()],
  build: {
    ssr: true, // enables SSR build
    rollupOptions: {
      input: './src/entry-server.tsx',
    },
  },
});
```

Hydrating on the Client

In **src/entry-client.tsx**:

```tsx
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';

ReactDOM.hydrateRoot(document.getElementById('root')!, <App />);
```

Key Difference from CSR:
`hydrateRoot()` is used instead of `createRoot()` —
because the HTML already exists and only needs reactivation, not re-rendering.

Introduction to SEO Optimization

Search Engine Optimization (SEO) is the process of improving your website visibility in search engines like Google, Bing, etc.

Why it matters:

- Increases organic traffic
- Improves brand credibility
- Enhances user trust and engagement

How Search Engines See React Apps

Problem with CSR:

React SPAs (Client-Side Rendering) send an empty HTML with a .

➜ Search crawlers may not execute JavaScript fully.

➜ Result: Poor indexing.

Solution:

✅ SSR (Server-Side Rendering)

✅ Pre-rendering (Static Generation)

✅ Metadata Injection (via react-helmet-async)

With Vite SSR, crawlers see the *fully rendered HTML* at first load.

Essential SEO Elements

Key on-page SEO factors in React apps:

Element	Description	Example
`<title>`	Page title displayed in search results	`<title>My Portfolio</title>`
`<meta>`	Describes page content	`<meta name="description" content="Fullstack Developer">`
`<h1>–<h6>`	Headings hierarchy	`<h1>About Me</h1>`
`<a href>`	Internal linking	`<a href="/projects">Projects</a>`
`alt` attributes	Image accessibility	`<img src="me.jpg" alt="Bagas photo" />`

Setting Up React Helmet Async

react-helmet-async is a library for managing meta tags and SEO attributes dynamically.

```
npm install react-helmet-async
```

Usage Example

```
import React from "react";
import { Helmet, HelmetProvider } from "react-helmet-async";

export default function App() {
  return (
    <HelmetProvider>
      <div>
        <Helmet>
          <title>React Vite SEO Example</title>
          <meta name="description" content="Demo app for SEO optimization
              with SSR and Vite." />
          <meta property="og:title" content="React Vite SEO Example" />
          <meta property="og:type" content="website" />
        </Helmet>
        <h1>Welcome to My SEO-Optimized React App</h1>
      </div>
    </HelmetProvider>
  );
}
```

SSR Integration with Helmet

To ensure metadata is included during SSR, integrate Helmet into the server render process.

```tsx
import React from 'react';
import ReactDOMServer from 'react-dom/server';
import { HelmetProvider } from 'react-helmet-async';
import App from './App';

export function render() {
  const helmetContext: any = {};
  const html = ReactDOMServer.renderToString(
    <HelmetProvider context={helmetContext}>
      <App />
    </HelmetProvider>
  );

  const { helmet } = helmetContext;
  return `
    <!DOCTYPE html>
    <html lang="en">
      <head>
        ${helmet.title.toString()}
        ${helmet.meta.toString()}
      </head>
      <body>
        <div id="root">${html}</div>
      </body>
    </html>`;
}
```

✅ Now, crawlers get full HTML + meta tags on first load!

Verify SEO Optimization

1. Use Google Lighthouse (Built into Chrome DevTools)

- Open your app in Chrome
- Press F12 → **Lighthouse tab**
- Choose SEO category → click Analyze page load
- Check score and recommendations for:
    - Page titles and meta descriptions
    - Proper **<h1>** structure
    - Links with descriptive text
    - Viewport and mobile-friendly layout

2. Use “View Page Source”

- Right-click → **View Page Source**
- Ensure server-rendered HTML contains:
    - **<title>** and **<meta>** tags
    - Readable text (not empty)

Introduction to Progressive Web Apps (PWA)

A Progressive Web App (PWA) is a web application that uses modern web capabilities to deliver an app-like experience to users.

Core Features:

- Works offline
- Installable on devices (Add to Home Screen)
- Fast & reliable, even on slow networks
- Responsive across devices

Goal: Combine the reach of the web with the user experience of native apps.

Adding PWA Support in React + Vite

1. Install plugin

```
npm install vite-plugin-pwa --save-dev
```

2. Update vite.config.ts:

```
import { defineConfig } from 'vite';
import react from '@vitejs/plugin-react';
import { VitePWA } from 'vite-plugin-pwa';

export default defineConfig({
  plugins: [
    react(),
    VitePWA({
      registerType: 'autoUpdate',
      manifest: {
        name: 'React Vite PWA Demo',
        short_name: 'VitePWA',
        start_url: '/',
        display: 'standalone',
        background_color: '#ffffff',
        theme_color: '#0ea5e9',
        icons: [
          {
            src: '/pwa-192x192.png',
            sizes: '192x192',
            type: 'image/png'
          },
          {
            src: '/pwa-512x512.png',
            sizes: '512x512',
            type: 'image/png'
          }
        ]
      }
    })
  ]
});
```

Creating the Web Manifest File

vite-plugin-pwa automatically generates *manifest.webmanifest*, but you can define it manually for more control.

```
{
  "name": "React Vite PWA Demo",
  "short_name": "VitePWA",
  "start_url": "/",
  "display": "standalone",
  "theme_color": "#0ea5e9",
  "background_color": "#ffffff",
  "icons": [
    { "src": "/pwa-192x192.png", "sizes": "192x192", "type": "image/png" },
    { "src": "/pwa-512x512.png", "sizes": "512x512", "type": "image/png" }
  ]
}
```

Purpose: Defines how your app appears on the user's home screen and splash screen.

Understanding the Service Worker

The Service Worker runs in the background and acts as a network proxy.

Main responsibilities:

- Cache static assets (HTML, JS, CSS, images)
- Serve cached content when offline
- Listen for updates and push notifications

Example auto-generated by vite-plugin-pwa:

```
self.addEventListener('fetch', (event) => {
  event.respondWith(
    caches.match(event.request).then((res) => {
      return res || fetch(event.request);
    })
  );
});
```

Registering the Service Worker in React

Add the following in your main.tsx or main.jsx:

```tsx
import React from "react";
import ReactDOM from "react-dom/client";
import App from "./App";
import { registerSW } from "virtual:pwa-register";

const updateSW = registerSW({
  onNeedRefresh() {
    if (confirm("New content available. Reload?")) {
      updateSW(true);
    }
  },
});

ReactDOM.createRoot(document.getElementById("root")!).render(<App />);
```

✅ This ensures the Service Worker is registered

✅ Prompts users when a new version is available

Testing your PWA

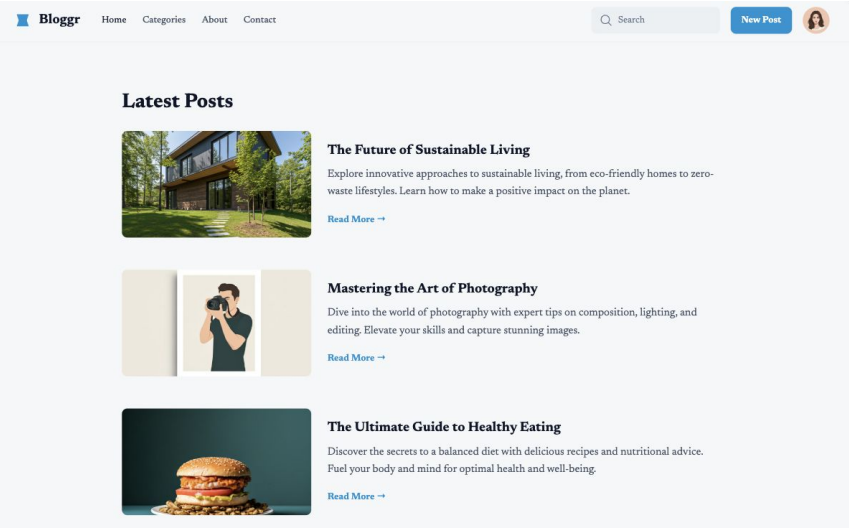
- **Build and Preview**

```
npm run build
npm run preview
```

- **Open Chrome DevTools → Application tab**
  - Check “Service Workers”
  - Check “Manifest” details
  - Verify Add to Home Screen prompt
- **Test Offline**
  - Go to DevTools → Network → Offline
  - Reload → your app should still load from cache

Exercise

- Develop a Blog App that uses **Server Side Rendering (SSR)** to improve page load speed, applies **SEO Optimization** to make articles easily discoverable by search engines, and integrates **Progressive Web App (PWA)** features so users can read posts offline and install the app on their devices.



Thank you