서버 사이드 렌더링 (SSR)

개요

SSR 란?

Vue.js는 클라이언트측 앱을 빌드하기 위한 프레임워크입니다. 기본적으로 브라우저에서 DOM으로 출력되는 Vue 컴포넌트를 생성하고 조작합니다. 그러나 동일한 컴포넌트를 서버의 HTML 문자열로 렌더링하여 정적 마크업을 클라이언트 브라우저로 보내는 완전한 대화형 앱으로 "하이드레이트"(hydrate: 변환)하는 것도 가능합니다.

서버에서 렌더링된 Vue.js 앱은 앱 코드의 대부분이 **서버와 클라이언트 모두에서 실행된다는 점**에서 "동형"(isomorphic) 또는 "범용"(universal)으로 간주될 수도 있습니다.

왜 SSR 일까요?

클라이언트측 SPA(Single-Page Application)와 비교할 때 SSR의 장점은 주로 다음과 같습니다:

컨텐츠에 도달하는 시간 단축: 인터넷 속도가 느리거나 기기가 느린 경우 더 두드러집니다. 서버 렌더링 마크업은 모든 JavaScript가 다운로드 및 실행되어 표시될 때까지 기다릴 필요가 없으므로 사용자가 완전히 렌더링된 페이지를 더 빨리 볼 수 있습니다. 또한 데이터 가져오기는 초기 방문을 위해 서버 측에서 수행되므로 클라이언트보다 데이터베이스에 더 빠르게 연결할 수 있습니다. 이는 일반적으로 개선된 **Core Web Vitals** 측정항목과 더 나은 UX를 가져오며, 컨텐츠에 도달하는 시간이 전환율과 직접적으로 관련된 앱에 중요할 수 있습니다.

통합 유지보수 모델: 백엔드 템플릿 시스템과 프론트엔드 프레임워크 사이를 왔다 갔다 하는 대신, 전체 앱을 개발하기 위해 동일한 언어와 선언적 컴포넌트 지향의 유지보수 모델을 사용할 수 있습니다.

더 나은 SEO: 검색 엔진 크롤러는 완전히 렌더링된 페이지를 직접 볼 수 있습니다.

> TIP
>
> 현재 **Google**과 **Bing**은 동기식 **JavaScript** 앱을 잘 인덱싱할 수 있습니다. 동기가 핵심 단어입니다. 앱이 로딩 스피너로 시작한 다음 **Ajax**를 통해 컨텐츠를 가져오는 경우, 크롤러는 완료될 때까지 기다리지 않습니다. 즉, **SEO**가 중요한 페이지에서 비동기적으로 컨텐츠를 가져오는 경우, **SSR**이 필요할 수 있습니다.

SSR을 사용할 때 고려해야 할 몇 가지 단점도 있습니다:

개발 제약 사항. 브라우저별 코드는 특정 생명주기 훅 내에서만 사용할 수 있습니다. 일부 외부 라이브러리는 서버 렌더링 앱에서 실행할 수 있도록 특별한 처리가 필요할 수 있습니다.

더 복잡한 빌드 설정 및 배포 요구 사항. 모든 정적 파일 서버에 배포할 수 있는 완전 정적 SPA와 달리 서버 렌더링 앱에는 Node.js 서버를 실행할 수 있는 환경이 필요합니다.

더 많은 서버 측 부하. Node.js에서 전체 앱을 렌더링하는 것은 정적 파일을 제공하는 것보다 CPU를 더 많이 사용하므로, 트래픽이 많을 것으로 예상되는 경우, 해당 서버 로드에 대비하고 캐싱 전략을 현명하게 사용하세요.

앱에 SSR을 사용하기 전, 가장먼저 실제로 필요한지에 대해 고려해야 합니다. 주로 앱에서 컨텐츠에 도달하는 시간이 얼마나 중요한지에 따라 다릅니다. 예를 들어 초기 로드 시, 추가로 수백 밀리초가 중요하지 않은 내부 대시보드를 구축하는 경우 SSR은 과도합니다. 그러나 컨텐츠에 도달하는 시간이 절대적으로 중요한 경우, SSR은 가능한 최상의 초기 로드 성능을 달성하는 데 도움이 될 수 있습니다.

SSR vs. SSG

사전 렌더링이라고도 하는 **정적 사이트 생성**(SSG: Static-Site Generation)은 빠른 웹사이트 구축을 위한 또 다른 인기 있는 기술입니다. 페이지를 서버 렌더링하는 데 필요한 데이터가 모든 사용자에 대해 동일한 경우, 요청이 들어올 때마다 페이지를 렌더링하는 대신 미리 빌드 프로세스에서 한 번만 렌더링하면 됩니다. 미리 렌더링된 페이지가 생성되어 정적 HTML 파일로 제공됩니다.

SSG는 SSR 앱과 동일한 성능 특성을 유지합니다. 즉, 뛰어난 컨텐츠 구현 시간 성능을 제공합니다. 동시에 출력이 정적 HTML 및 자산이기 때문에 SSR 앱보다 저렴하고 배포하기 쉽습니다. 여기서 키워드는 **정적**입니다. SSG는 정적 데이터, 즉 빌드 시 결정되어 배포 간에 변경되

지 않는 데이터를 소비하는 페이지에만 적용할 수 있습니다. 하지만 데이터가 변경될 때마다 새로운 배포가 필요합니다.

소수의 마케팅 페이지(예: `/`, `/about`, `/contact` 등)의 SEO를 개선하기 위해 SSR만 고려하고 있다면, SSG를 대안으로 추천합니다. SSG는 문서 사이트나 블로그와 같은 컨텐츠 기반 웹사이트에도 적합합니다. 사실 지금 보고 있는 이 웹사이트는 Vue 기반 정적 사이트 생성기 **VitePress**를 사용하여 정적으로 생성된 것입니다.

기본 튜토리얼

앱 렌더링하기

Vue SSR이 작동하는 가장 기본적인 예를 살펴보겠습니다.

새 디렉터리를 만들고 그 안에 `cd` 를 넣습니다.
`npm init -y` 실행
Node.js가 **ES** 모듈 모드에서 실행되도록 `package.json` 에 `"type": "module"` 을 추가합니다.
`npm install vue` 실행
`example.js` 파일을 만듭니다.

```js
// 이것은 서버의 Node.js에서 실행됩니다.
import { createSSRApp } from 'vue'
// Vue의 서버 렌더링 API는 `vue/server-renderer`에 있습니다.
import { renderToString } from 'vue/server-renderer'

const app = createSSRApp({
  data: () => ({ count: 1 }),
  template: `<button @click="count++">{{ count }}</button>`
})

renderToString(app).then((html) => {
  console.log(html)
})
```

그런 다음 실행:

```sh
> node example.js
```

명령줄에 다음이 출력되야 합니다:

```
<button>1</button>
```

`renderToString()` 은 Vue 앱 인스턴스를 사용하여 앱의 렌더링된 HTML로 해결되는 Promise를 반환합니다. **Node.js Stream API** 또는 **Web Streams API**를 사용하여 스트리밍 렌더링도 가능합니다. 자세한 내용은 **SSR API**를 확인하세요.

그런 다음 앱이 전체 페이지 HTML로 마크업 된 Vue SSR 코드를 서버 리퀘스트 핸들러로 이동시킬 수 있습니다. 다음 단계에서는 `express` 를 사용할 것입니다:

`npm install express` 실행
아래와 같은 `server.js` 파일 생성:

```js
import express from 'express'
import { createSSRApp } from 'vue'
import { renderToString } from 'vue/server-renderer'

const server = express()

server.get('/', (req, res) => {
  const app = createSSRApp({
```

```
    data: () => ({ count: 1 }),
    template: `<button @click="count++">{{ count }}</button>`
  })

  renderToString(app).then((html) => {
    res.send(`
    <!DOCTYPE html>
    <html>
      <head>
        <title>Vue SSR 예제</title>
      </head>
      <body>
        <div id="app">${html}</div>
      </body>
    </html>
    `)
  })
})

server.listen(3000, () => {
  console.log('ready')
})
```

마지막으로 node server.js 를 실행하고 http://localhost:3000 에 접속합니다. 버튼으로 작동하는 페이지가 표시되어야 합니다.

StackBlitz에서 실행하기

> 참고
>
> StackBlitz 웹 사이트는 쿠키가 차단된 환경(예: 시크릿 모드)에서 원활하게 작동하지 않습니다.
> 위 예제 링크에서 버튼이 작동하지 않는 것은 의도된 것입니다.

클라이언트 하이드레이트

버튼을 클릭하면 숫자가 변경되지 않는 것을 알 수 있습니다. 브라우저에서 Vue를 로드하지 않기 때문에 HTML은 클라이언트에서 완전히 정적입니다.

클라이언트 측 앱을 대화형으로 만들기 위해 Vue는 **하이드레이트** 단계를 수행해야 합니다. 하이드레이트하는 동안 서버에서 실행된 것과 동일한 Vue 앱을 만들고 제어해야 하는 DOM 노드에 각 컴포넌트를 일치시키고 DOM 이벤트 핸들러를 연결합니다.

앱을 하이드레이트 모드로 마운트하려면 createApp() 대신 **createSSRApp()** 를 사용해야 합니다:

```
// 이것은 브라우저에서 실행됩니다
import { createSSRApp } from 'vue'

const app = createSSRApp({
  // ...서버와 동일한 앱
})

// 클라이언트에 SSR 앱을 탑재하면,
// HTML이 미리 렌더링되었다고 가정하고,
// 새 DOM 노드를 마운트하는 대신 하이드레이트를 수행합니다.
app.mount('#app')
```

js

코드 구조

서버에서와 동일한 앱 구현을 어떻게 재사용해야 하는지 주목하세요. 여기에서 SSR 앱의 코드 구조에 대해 생각해야 합니다. 서버와 클라이언트 간에 동일한 앱 코드를 공유하는 방법은 무엇입니까?

여기서 우리는 가장 기본적인 설정을 보여줄 것입니다. 먼저 앱 생성 로직을 전용 파일 app.js 로 분할해 보겠습니다:

```js
// app.js (서버와 클라이언트 간에 공유)
import { createSSRApp } from 'vue'

export function createApp() {
  return createSSRApp({
    data: () => ({ count: 1 }),
    template: `<button @click="count++">{{ count }}</button>`
  })
}
```

이 파일과 그 의존성은 서버와 클라이언트 간에 공유됩니다. 우리는 이를 **범용 코드**(universal code)라고 부릅니다. 범용 코드를 작성할 때 주의해야 할 몇 가지 사항이 있습니다. 아래에서 논의.

클라이언트 항목은 범용 코드를 가져오고 앱을 만들고 마운트를 수행합니다:

```js
// client.js
import { createApp } from './app.js'

createApp().mount('#app')
```

그리고 서버는 리퀘스트 핸들러에서 동일한 앱 생성 로직을 사용합니다:

```js
// server.js (관련 없는 코드 생략)
import { createApp } from './app.js'

server.get('/', (req, res) => {
  const app = createApp()
  renderToString(app).then(html => {
    // ...
  })
})
```

또한 브라우저에서 클라이언트 파일을 로드하려면 다음 작업도 수행해야 합니다:

server.js 에 server.use(express.static('.')) 를 추가하여 클라이언트 파일을 제공합니다.
HTML 셸에 <script type="module" src="/client.js"></script> 를 추가하여 클라이언트 항목을 로드합니다.
HTML 셸에 **Import Map**을 추가하여 브라우저에서 import * from 'vue' 와 같은 사용법을 지원합니다.

StackBlitz에서 완성된 예제 실행하기. 버튼은 이제 상호작용 합니다!

고수준 솔루션

예제에서 프로덕션 준비 SSR 앱으로 이동하려면 훨씬 더 많은 작업이 필요합니다. 다음을 수행해야 합니다:

Vue SFC 및 기타 빌드 과정 요구 사항을 지원합니다. 사실, 동일한 앱에 대해 두 개의 빌드를 조정해야 합니다. 하나는 클라이언트용이고 다른 하나는 서버용입니다.

> **TIP**
>
> Vue 컴포넌트는 SSR에 사용할 때 다르게 컴파일됩니다. 템플릿은 보다 효율적인 렌더링 성능을 위해 가상 DOM 렌더링 기능 대신 문자열 연결로 컴파일됩니다.

서버 리퀘스트 핸들러에서 올바른 클라이언트 측 애셋 링크와 최적의 리소스 힌트를 사용하여 HTML을 렌더링합니다. SSR과 SSG 모드 사이를 전환하거나 동일한 앱에서 둘 다를 혼합해야 할 수도 있습니다.

보편적인 방식으로 라우팅, 데이터 가져오기 및 상태 관리 저장소를 관리합니다.

완전한 구현은 매우 복잡하며 작업하기로 선택한 빌드 도구 체인에 따라 다릅니다. 따라서 복잡성을 추상화하는 더 높은 수준의 독창적인 솔루션을 사용하는 것이 좋습니다. 아래에서는 Vue 생태계에서 권장되는 몇 가지 SSR 솔루션을 소개합니다.

Nuxt

Nuxt는 범용 Vue 앱을 작성하기 위한 간소화된 개발 경험을 제공하는 Vue 생태계 위에 구축된 상위 수준 프레임워크입니다. 더 좋은 점은 정적 사이트 생성기로도 사용할 수 있다는 것입니다! 시도해 볼 것을 적극 권장합니다.

Quasar

Quasar는 하나의 코드베이스를 사용하여 SPA, SSR, PWA, 모바일 앱, 데스크톱 앱 및 브라우저 확장을 모두 타겟팅할 수 있는 완전한 Vue 기반 솔루션입니다. 빌드 설정을 처리할 뿐만 아니라 Material Design 호환 UI 컴포넌트의 전체 컬렉션을 제공합니다.

Vite SSR

Vite는 내장된 **Vue** 서버 사이드 렌더링 지원을 제공하지만 의도적으로 저수준입니다. Vite를 직접 사용하고 싶다면 **vite-plugin-ssr**을 확인하십시오. 커뮤니티 플러그인은 많은 어려운 세부 사항을 추상화해 줍니다.

여기에서 수동 설정을 사용하는 Vue + Vite SSR 프로젝트의 예를 찾을 수도 있습니다. 이 프로젝트를 베이스로 삼을 수 있습니다. 이것은 SSR/빌드 도구에 대한 경험이 있고 더 높은 수준의 아키텍처를 완전히 제어하려는 경우에만 권장됩니다.

SSR 친화적인 코드 작성

빌드 설정 또는 고수준 프레임워크 선택에 관계없이 모든 Vue SSR 앱에 적용되는 몇 가지 원칙이 있습니다.

서버에서 반응형

SSR 동안 각 리퀘스트 URL은 앱이 원하는 상태로 매핑됩니다. 사용자 상호 작용 및 DOM 업데이트가 없으므로 서버에서 반응형이 필요하지 않습니다. 기본적으로 반응성은 더 나은 성능을 위해 SSR 동안 비활성화됩니다.

컴포넌트 생명 주기 훅

동적 업데이트가 없기 때문에 `onMounted` 또는 `onUpdated` 과 같은 생명 주기 훅은 SSR 중에 호출되지 않고 클라이언트에서만 실행됩니다.

`setup()` 또는 `<script setup>` 의 루트 범위 에서 정리가 필요한 사이드 이펙트을 생성하는 코드를 피해야 합니다. 이러한 사이드 이펙트의 예는 `setInterval` 로 타이머를 설정하는 것입니다. 클라이언트 측 전용 코드에서 타이머를 설정한 다음 `onBeforeUnmount` 또는 `onUnmounted` 에서 해제할 수 있습니다. 그러나 마운트 해제 훅은 SSR 중에 호출되지 않기 때문에 타이머는 영원히 유지됩니다. 이를 피하려면 사이드 이펙트 코드를 `onMounted` 로 이동하십시오.

플랫폼별 API에 대한 접근

범용 코드는 플랫폼별 API에 대한 접근을 가정할 수 없으므로, 코드가 `window` 또는 `document` 와 같은 브라우저 전용 전역을 직접 사용하는 경우, Node.js에서 실행할 때 에러가 발생하고, 그 반대의 경우도 마찬가지입니다.

서버와 클라이언트 간에 공유되지만 플랫폼 API가 다른 작업의 경우, 플랫폼별 구현을 범용 API 내에 래핑하거나 이를 수행하는 라이브러리를 사용하는 것이 좋습니다. 예를 들어, **node-fetch** 를 사용하여 서버와 클라이언트 모두에서 동일한 가져오기 API를 사용할 수 있습니다.

브라우저 전용 API의 경우, 일반적인 접근 방식은 한 탬포 느리게 `onMounted` 와 같은 클라이언트 전용 생명 주기 훅 내에서 접근하는 것입니다.

타사 라이브러리가 보편적 사용을 염두에 두고 작성되지 않은 경우, 서버에서 렌더링된 앱에 통합하기가 까다로울 수 있습니다. 일부 전역을 모의하여 작동하도록 할 수 있지만 임시방편적이며, 다른 라이브러리의 환경 감지 코드를 방해할 수 있습니다.

교차 요청 상태 오염(Cross-Request State Pollution)

상태 관리 문서에서 반응성 **API**를 통한 간단한 상태 관리 패턴을 소개했습니다. SSR 컨텍스트에서 이 패턴은 몇 가지 추가 조정이 필요합니다.

패턴은 JavaScript 모듈의 루트 범위에서 공유 상태를 선언합니다. 이것은 그것들을 **싱글톤**으로 만듭니다. 즉, 앱의 전체 생명 주기 동안 반응형 객체의 인스턴스가 하나만 있습니다. 이것은 우리 앱의 모듈이 각 브라우저 페이지 방문에 대해 새로 초기화되기 때문에 순수한 클라이언트 측 Vue 앱에서 예상대로 작동합니다.

그러나 SSR 컨텍스트에서 앱 모듈은 일반적으로 서버가 부팅될 때 서버에서 한 번만 초기화됩니다. 동일한 모듈 인스턴스가 여러 서버 요청에서 재사용되고 싱글톤 상태 객체도 재사용됩니다. 공유 싱글톤 상태를 한 사용자와 관련된 데이터로 변경하면 실수로 다른 사용자의 요청으로 유출될 수 있습니다. 우리는 이것을 **교차 요청 상태 오염**이라고 부릅니다.

브라우저에서 하는 것처럼 각 요청에 대해 모든 JavaScript 모듈을 기술적으로 다시 초기화할 수 있습니다. 그러나 JavaScript 모듈을 초기화하는 것은 비용이 많이 들 수 있으므로 서버 성능에 상당한 영향을 미칠 수 있습니다.

권장되는 솔루션은 각 요청에 대해 라우터 및 전역 저장소를 포함한 전체 앱의 새 인스턴스를 만드는 것입니다. 그런 다음 컴포넌트에서 직접 가져오는 대신 앱 수준 **provide**를 사용하여 공유 상태를 제공하고, 이를 필요로 하는 컴포넌트에 주입합니다.

```js
// app.js (서버와 클라이언트 간에 공유)
import { createSSRApp } from 'vue'
import { createStore } from './store.js'

export function createApp() {
  const app = createSSRApp(/* ... */)
  // 리퀘스트 마다 store의 새 인스턴스 생성
  const store = createStore(/* ... */)
  // 앱 수준에서 store를 provide
  app.provide('store', store)
  // also expose store for hydration purposes
  // 또한 하이드레이션(hydration)을 위해 store를 내보냄.
  return { app, store }
}
```

Pinia와 같은 상태 관리 라이브러리는 이를 염두에 두고 설계되었습니다. 자세한 내용은 **Pinia**의 **SSR** 가이드를 참고하세요.

하이드레이션 불일치(Hydration Mismatch)

미리 렌더링된 HTML의 DOM 구조가 클라이언트 측 앱의 예상 출력과 일치하지 않으면 하이드레이션 불일치 오류가 발생합니다. 하이드레이션 불일치는 다음과 같은 원인으로 인해 가장 일반적으로 발생합니다:

템플릿에 잘못된 HTML 중첩 구조가 포함되어 있고 브라우저의 기본 HTML 구문 분석 동작에 의해 렌더링된 HTML이 "수정"된 경우입니다. 예를 들어, `<div>` 를 `<p>` 안에 넣을 수 없다 는 것이 일반적인 문제입니다.

```html
<p><div>hi</div></p>
```

이것을 서버에서 렌더링하여 HTML로 생성하면, 브라우저는 `<div>` 를 만났을 때 첫 번째 `<p>` 를 종료하고 다음 DOM 구조로 구문 분석합니다.

```html
<p></p>
<div>hi</div>
<p></p>
```

렌더링 중에 사용되는 데이터에는 무작위로 생성된 값이 포함됩니다. 동일한 애플리케이션이 서버에서 한 번, 클라이언트에서 한 번 두 번 실행되므로 두 실행 간에 무작위 값이 동일하다고 보장할 수 없습니다. 무작위 값으로 인한 불일치를 방지하는 방법에는 두 가지가 있습니다:

`v-if` + `onMounted` 를 사용하여 무작위 값에 의존하는 부분을 클라이언트에서만 렌더링합니다. 프레임워크에 이 작업을 더 쉽게 할 수 있는 기능이 내장되어 있을 수도 있습니다(예: VitePress의 `<ClientOnly>` 컴포넌트).

시드 생성을 지원하는 난수 생성기 라이브러리를 사용하고, 서버 실행과 클라이언트 실행이 동일한 시드를 사용하도록 보장합니다 (예: 시드를 직렬화된 상태로 포함하고 클라이언트에서 검색하는 방식).

서버와 클라이언트가 서로 다른 시간대에 있는 경우. 타임스탬프를 사용자의 현지 시간으로 변환해야 하는 경우가 있습니다. 그러나 서버 실행 중 시간대와 클라이언트 실행 중 시간대가 항상 같은 것은 아니며, 서버 실행 중 사용자의 시간대를 안정적으로 알 수 없을 수도 있습니다. 이러한 경우 현지 시간 변환은 클라이언트 전용 작업으로 수행해야 합니다.

Vue에서 하이드레이션 불일치가 발생하면 클라이언트 측 상태와 일치하도록 사전 렌더링된 DOM을 자동으로 복구하고 조정하려고 시도합니다. 이로 인해 잘못된 노드가 삭제되고 새 노드가 마운트되어 렌더링 성능이 약간 저하될 수 있지만 대부분의 경우 앱은 예상대로 계속 작동합니다. 그렇긴 하지만 개발 중에 하이드레이션 불일치를 제거하는 것이 가장 좋습니다.

커스텀 디렉티브

대부분의 커스텀 디렉티브는 직접적인 DOM 조작을 포함하므로 SSR 동안 무시됩니다. 그러나 커스텀 디렉티브를 렌더링하는 방법(즉, 렌더링된 엘리먼트에 추가해야 하는 속성)을 지정하려면 `getSSRProps` 디렉티브 훅을 사용할 수 있습니다:

```js
const myDirective = {
  mounted(el, binding) {
    // 클라이언트 측 구현:
    // DOM을 직접 업데이트
    el.id = binding.value
  },
  getSSRProps(binding) {
    // 서버 측 구현:
    // 렌더링할 props를 반환
    // getSSRProps는 디렉티브 바인딩만 받습니다.
    return {
      id: binding.value
    }
  }
}
```

텔레포트

텔레포트는 SSR 동안 특별한 처리가 필요합니다. 렌더링된 앱에 텔레포트가 포함된 경우 텔레포트된 컨텐츠는 렌더링된 문자열의 일부가 아닙니다. 더 쉬운 솔루션은 마운트 시 텔레포트를 조건부로 렌더링하는 것입니다.

텔레포트된 컨텐츠를 하이드레이트 해야 하는 경우, ssr 컨텍스트 객체의 `teleports` 속성에서 노출됩니다:

```js
const ctx = {}
const html = await renderToString(app, ctx)

console.log(ctx.teleports) // { '#teleported': '텔레포트된 컨텐츠' }
```

메인 앱 마크업을 삽입하는 방법과 유사하게 최종 페이지 HTML의 올바른 위치에 텔레포트 마크업을 삽입해야 합니다.

TIP

텔레포트와 SSR을 함께 사용할 때 `body` 를 타겟팅하지 마십시오. 일반적으로 `<body>` 에는 다른 서버 렌더링 컨텐츠가 포함되므로, Teleports가 하이드레이션을 위한 올바른 시작 위치를 결정할 수 없습니다.

그러니 텔레포트된 컨텐츠만 포함하는 전용 컨테이너(예: `<div id="teleported"></div>`) 방식을 추구하십시오.