서버 사이드 렌더링(SSR)

개요

SSR이란?

Vue.js는 클라이언트 사이드 애플리케이션을 구축하기 위한 프레임워크입니다. 기본적으로 Vue 컴포넌트는 브라우저에서 DOM을 생성하고 조작합니다. 하지만 동일한 컴포넌트를 서버에서 HTML 문자열로 렌더링하고, 이를 브라우저로 직접 전송한 뒤, 클라이언트에서 정적 마크업을 완전히 상호작용 가능한 앱으로 "하이드레이트"하는 것도 가능합니다.

서버에서 렌더링된 Vue.js 앱은 "isomorphic" 또는 "universal" 앱이라고도 할 수 있습니다. 즉, 앱 코드의 대부분이 서버 **와** 클라이언트 모두에서 실행된다는 의미입니다.

왜 SSR을 사용할까?

클라이언트 사이드 싱글 페이지 애플리케이션(SPA)과 비교했을 때, SSR의 주요 장점은 다음과 같습니다:

더 빠른 콘텐츠 표시 시간: 느린 인터넷이나 느린 기기에서 더욱 두드러집니다. 서버에서 렌더링된 마크업은 모든 자바스크립트가 다운로드되고 실행될 때까지 기다릴 필요 없이 바로 표시되므로, 사용자는 더 빨리 완전히 렌더링된 페이지를 볼 수 있습니다. 또한, 초기 방문 시 데이터 페칭이 서버에서 이루어지므로, 클라이언트보다 데이터베이스에 더 빠르게 접근할 수 있습니다. 이는 일반적으로 Core Web Vitals 지표 개선, 더 나은 사용자 경험, 그리고 콘텐츠 표시 시간이 전환율과 직접적으로 연관된 애플리케이션에서 매우 중요할 수 있습니다.

통합된 사고 모델: 백엔드 템플릿 시스템과 프론트엔드 프레임워크를 오가며 개발하는 대신, 전체 앱 개발에 동일한 언어와 선언적, 컴포넌트 지향적 사고 모델을 사용할 수 있습니다.

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

> TIP

"동기적"이 핵심입니다. 만약 앱이 로딩 스피너도 시작한 뒤 Ajax로 콘텐츠를 가져온다면, 크롤러는 기다려주지 않습니다. 즉, SEO가 중요한 페이지에서 비동기적으로 콘텐츠를 가져온다면 SSR이 필요할 수 있습니다.

SSR을 사용할 때 고려해야 할 트레이드오프도 있습니다:

개발 제약. 브라우저 전용 코드는 특정 라이프사이클 훅 내에서만 사용할 수 있습니다. 일부 외부 라이브러리는 서버 렌더링 앱에서 실행되도록 별도의 처리가 필요할 수 있습니다.

더 복잡한 빌드 설정 및 배포 요구사항. 완전히 정적인 SPA는 어떤 정적 파일 서버에도 배포할 수 있지만, 서버 렌더링 앱은 Node.js 서버가 실행될 수 있는 환경이 필요합니다.

더 많은 서버 부하. Node.js에서 전체 앱을 렌더링하는 것은 단순히 정적 파일을 제공하는 것보다 CPU를 더 많이 사용하므로, 트래픽이 많을 것으로 예상된다면 이에 맞는 서버 부하와 캐싱 전략을 현명하게 준비해야 합니다.

앱에 SSR을 도입하기 전에, 가장 먼저 스스로에게 물어야 할 질문은 "정말로 SSR이 필요한가?"입니다. 이는 주로 앱에서 콘텐츠 표시 시간이 얼마나 중요한지에 달려 있습니다. 예를 들어, 초기 로딩에 몇 백 밀리초가 더 걸려도 상관없는 내부 대시보드를 만든다면 SSR은 과할 수 있습니다. 하지만 콘텐츠 표시 시간이 매우 중요한 경우, SSR은 최고의 초기 로딩 성능을 달성하는 데 도움이 될 수 있습니다.

SSR vs. SSG

정적 사이트 생성(SSG), 즉 프리렌더링은 빠른 웹사이트를 구축하기 위한 또 다른 인기 있는 기법입니다. 페이지를 서버에서 렌더링하는 데 필요한 데이터가 모든 사용자에게 동일하다면, 요청이 들어올 때마다 페이지를 렌더링하는 대신 빌드 과정에서 한 번만 렌더링할 수 있습니다. 프리렌더링된 페이지는 정적 HTML 파일로 생성되어 제공됩니다.

SSG는 SSR 앱과 동일한 성능 특성을 유지합니다: 뛰어난 콘텐츠 표시 성능을 제공합니다. 동시에, 출력물이 정적 HTML과 에셋이기 때문에 SSR 앱보다 더 저렴하고 쉽게 배포할 수 있습니다. 여기서 핵심은 **정적**이라는 점입니다: SSG는 빌드 시점에 알 수 있고, 요청마다 변하지 않는 정적 데이터를 제공하는 페이지에만 적용할 수 있습니다. 데이터가 변경될 때마다 새로 배포해야 합니다.

SSR을 도입하려는 이유가 마케팅 페이지 몇 개(예: `/`, `/about`, `/contact` 등)의 SEO를 개선하기 위함이라면, SSR 대신 SSG를 사용하는 것이 더 적합할 수 있습니다. SSG는 문서 사이트나 블로그와 같은 콘텐츠 기반 웹사이트에도 매우 적합합니다. 실제로, 지금 읽고 있는 이 웹사이트도 VitePress라는 Vue 기반 정적 사이트 생성기를 사용해 정적으로 생성되었습니다.

기본 튜토리얼

Vue SSR의 가장 기본적인 예제를 살펴보겠습니다.

1. 새 디렉터리를 만들고 `cd` 로 이동합니다.
2. `npm init -y` 를 실행합니다.
3. `package.json` 에 `"type": "module"` 을 추가하여 Node.js가 ES 모듈 모드로 실행되도록 합니다.
4. `npm install vue` 를 실행합니다.
5. `example.js` 파일을 생성합니다:

```js
// 이 코드는 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)
})
```

이제 다음을 실행합니다:

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

명령줄에 다음과 같이 출력되어야 합니다:

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

`renderToString()` 은 Vue 앱 인스턴스를 받아 앱의 렌더링된 HTML로 resolve되는 Promise를 반환합니다. Node.js Stream API나 Web Streams API를 사용해 스트림 렌더링도 가능합니다. 자세한 내용은 SSR API 레퍼런스를 참고하세요.

이제 Vue SSR 코드를 서버 요청 핸들러로 옮겨, 애플리케이션 마크업을 전체 페이지 HTML로 감쌀 수 있습니다. 다음 단계에서는 `express` 를 사용합니다:

`npm install express` 를 실행합니다.
다음과 같이 `server.js` 파일을 생성합니다:

```
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 접속하세요. 버튼이 있는 페이지가 정상적으로 동작하는 것을 볼 수 있습니다.

StackBlitz에서 직접 실행해보기

클라이언트 하이드레이션

버튼을 클릭해보면 숫자가 변하지 않는 것을 알 수 있습니다. 브라우저에서 Vue를 로드하지 않았기 때문에 HTML이 클라이언트에서 완전히 정적입니다.

클라이언트 사이드 앱을 상호작용 가능하게 만들려면, Vue가 **하이드레이션** 단계를 수행해야 합니다. 하이드레이션 과정에서 서버에서 실행된 것과 동일한 Vue 애플리케이션을 생성하고, 각 컴포넌트를 제어해야 할 DOM 노드와 매칭하며, DOM 이벤트 리스너를 연결합니다.

하이드레이션 모드로 앱을 마운트하려면 `createApp()` 대신 `createSSRApp()` 을 사용해야 합니다:

```
import { createSSRApp } from 'vue'

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

// 클라이언트에서 SSR 앱을 마운트하면
// HTML이 미리 렌더링되었다고 가정하고
// 새로운 DOM 노드를 마운트하는 대신 하이드레이션을 수행합니다.
app.mount('#app')
```

코드 구조

서버에서와 동일한 앱 구현을 재사용해야 한다는 점에 주목하세요. SSR 앱에서는 코드 구조에 대해 고민해야 합니다. 즉, 서버와 클라이언트에서 동일한 애플리케이션 코드를 어떻게 공유할 것인가?

여기서는 가장 기본적인 구조를 보여줍니다. 먼저, 앱 생성 로직을 `app.js` 라는 전용 파일로 분리해봅시다:

app.js

```
// (서버와 클라이언트에서 공유)
import { createSSRApp } from 'vue'

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

js

이 파일과 그 의존성들은 서버와 클라이언트에서 모두 공유됩니다. 우리는 이를 **유니버설 코드**라고 부릅니다. 유니버설 코드를 작성할 때 주의해야 할 점들이 있는데, 이는 아래에서 더 자세히 다룹니다.

클라이언트 엔트리에서는 유니버설 코드를 import하고, 앱을 생성한 뒤 마운트합니다:

client.js

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

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

js

서버에서도 요청 핸들러에서 동일한 앱 생성 로직을 사용합니다:

```js
// (불필요한 코드는 생략)
import { createApp } from './app.js'

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

또한, 브라우저에서 클라이언트 파일을 로드하려면 다음도 필요합니다:

1. `server.js` 에 `server.use(express.static('.'))` 를 추가하여 클라이언트 파일을 제공합니다.
2. HTML 셸에 `<script type="module" src="/client.js"></script>` 를 추가하여 클라이언트 엔트리를 로드합니다.
3. 브라우저에서 `import * from 'vue'` 와 같은 사용을 지원하려면 HTML 셸에 Import Map을 추가합니다.

완성된 예제를 StackBlitz에서 실행해보기. 이제 버튼이 상호작용 가능합니다!

상위 레벨 솔루션

예제에서 실제 운영 환경의 SSR 앱으로 발전시키려면 더 많은 작업이 필요합니다. 다음이 필요합니다:

Vue SFC 및 기타 빌드 단계 요구사항 지원. 실제로, 동일한 앱에 대해 클라이언트용 빌드와 서버용 빌드 두 가지를 조율해야 합니다.

> ⓘ TIP
>
> Vue 컴포넌트는 SSR에서 다르게 컴파일됩니다. 템플릿이 더 효율적인 렌더링 성능을 위해 Virtual DOM 렌더 함수 대신 문자열 연결로 컴파일됩니다.

서버 요청 핸들러에서 올바른 클라이언트 사이드 에셋 링크와 최적의 리소스 힌트로 HTML을 렌더링합니다. SSR과 SSG 모드를 전환하거나, 심지어 동일한 앱에서 둘을 혼합해야 할 수도 있습니다.

라우팅, 데이터 페칭, 상태 관리 스토어를 유니버설하게 관리합니다.

추상화해주는 상위 레벨의 의견이 반영된 솔루션을 사용하는 것을 강력히 권장합니다. 아래에서는 Vue 생태계에서 추천하는 SSR 솔루션 몇 가지를 소개합니다.

Nuxt

Nuxt는 Vue 생태계 위에 구축된 상위 레벨 프레임워크로, 유니버설 Vue 애플리케이션을 작성할 때 효율적인 개발 경험을 제공합니다. 게다가 정적 사이트 생성기로도 사용할 수 있습니다! 꼭 한 번 사용해보시길 추천합니다.

Quasar

Quasar는 하나의 코드베이스로 SPA, SSR, PWA, 모바일 앱, 데스크톱 앱, 브라우저 확장 프로그램까지 모두 타겟팅할 수 있는 완전한 Vue 기반 솔루션입니다. 빌드 설정을 처리할 뿐만 아니라, Material Design을 준수하는 UI 컴포넌트 전체 컬렉션도 제공합니다.

Vite SSR

Vite는 Vue 서버 사이드 렌더링 지원을 내장하고 있지만, 의도적으로 저수준입니다. Vite를 직접 사용하고 싶다면, 많은 어려운 세부사항을 추상화해주는 커뮤니티 플러그인인 vite-plugin-ssr을 참고하세요.

수동 설정으로 Vue + Vite SSR 프로젝트 예제도 여기에서 확인할 수 있습니다. 이는 SSR/빌드 도구에 익숙하고 상위 아키텍처를 완전히 제어하고 싶은 경우에만 추천합니다.

SSR 친화적인 코드 작성하기

빌드 설정이나 상위 프레임워크 선택과 관계없이, 모든 Vue SSR 애플리케이션에 적용되는 원칙이 있습니다.

서버에서의 반응성

SSR 중에는 각 요청 URL이 애플리케이션의 원하는 상태에 매핑됩니다. 사용자 상호작용이나 DOM 업데이트가 없으므로, 서버에서는 반응성이 불필요합니다. 기본적으로 SSR 중에는 성능 향상을 위해 반응성이 비활성화됩니다.

컴포넌트 라이프사이클 훅

출되지 않으며, 클라이언트에서만 실행됩니다.

`setup()` 또는 `<script setup>` 의 루트 스코프에서 정리(cleanup)가 필요한 부수 효과를 발생시키는 코드는 피해야 합니다. 예를 들어, `setInterval` 로 타이머를 설정하는 것이 부수 효과의 한 예입니다. 클라이언트 전용 코드에서는 타이머를 설정한 뒤 `onBeforeUnmount` 나 `onUnmounted` 에서 해제할 수 있습니다. 하지만 SSR 중에는 언마운트 훅이 절대 호출되지 않으므로, 타이머가 영원히 남아 있게 됩니다. 이를 피하려면 부수 효과 코드를 `onMounted` 로 옮기세요.

플랫폼 전용 API 접근

유니버설 코드는 플랫폼 전용 API에 접근할 수 있다고 가정해서는 안 됩니다. 예를 들어, 브라우저 전용 전역 객체인 `window` 나 `document` 를 직접 사용하면 Node.js에서 실행 시 오류가 발생합니다. 반대의 경우도 마찬가지입니다.

서버와 클라이언트 모두에서 공유하지만 플랫폼 API가 다른 작업의 경우, 플랫폼 전용 구현을 유니버설 API로 감싸거나, 이를 대신해주는 라이브러리를 사용하는 것이 좋습니다. 예를 들어, `node-fetch` 를 사용하면 서버와 클라이언트 모두에서 동일한 fetch API를 사용할 수 있습니다.

브라우저 전용 API의 경우, 일반적으로 `onMounted` 와 같은 클라이언트 전용 라이프사이클 훅 내부에서 지연 접근하는 방식이 일반적입니다.

서드파티 라이브러리가 유니버설 사용을 염두에 두고 작성되지 않았다면, 서버 렌더링 앱에 통합하는 것이 까다로울 수 있습니다. 일부 전역 객체를 모킹(mocking)하여 동작하게 만들 수도 있지만, 이는 해키(hacky)하며 다른 라이브러리의 환경 감지 코드에 영향을 줄 수 있습니다.

요청 간 상태 오염

상태 관리 챕터에서는 반응성 API를 사용한 간단한 상태 관리 패턴을 소개했습니다. SSR 환경에서는 이 패턴에 추가적인 조정이 필요합니다.

이 패턴은 자바스크립트 모듈의 루트 스코프에 공유 상태를 선언합니다. 이는 **싱글턴**이 됩니다. 즉, 애플리케이션 전체 생명주기 동안 반응형 객체 인스턴스가 하나만 존재합니다. 순수 클라이언트 사이드 Vue 애플리케이션에서는 각 브라우저 페이지 방문마다 모듈이 새로 초기화되므로 문제가 없습니다.

하지만 SSR 환경에서는 애플리케이션 모듈이 서버가 부팅될 때 한 번만 초기화되는 경우가 많습니다. 동일한 모듈 인스턴스가 여러 서버 요청에 재사용되며, 싱글턴 상태 객체도 마찬가지입니다. 만약 공유 싱글턴 상태를 사용자별 데이터로 변경하면, 다른 사용자의 요청에 그 데이터가 실수로 노출될 수 있습니다. 이를 **요청 간 상태 오염**이라고 합니다.

기술적으로는 각 요청마다 모든 자바스크립트 모듈을 다시 초기화할 수도 있지만, 이는 비용이 많이 들기 때문에 서버 성능에 큰 영향을 미칩니다.

턴스를 생성하는 것입니다. 그리고 컴포넌트에서 직접 import하는 대신, 앱 레벨 provide를 사용해 공유 상태를 제공하고, 필요한 컴포넌트에서 inject합니다:

app.js

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

// 각 요청마다 호출됨
export function createApp() {
  const app = createSSRApp(/* ... */)
  // 요청마다 새로운 store 인스턴스 생성
  const store = createStore(/* ... */)
  // 앱 레벨에서 store를 provide
  app.provide('store', store)
  // 하이드레이션을 위해 store도 노출
  return { app, store }
}
```

Pinia와 같은 상태 관리 라이브러리는 이를 염두에 두고 설계되었습니다. 자세한 내용은 Pinia의 SSR 가이드를 참고하세요.

하이드레이션 불일치

프리렌더된 HTML의 DOM 구조가 클라이언트 사이드 앱의 예상 출력과 일치하지 않으면 하이드레이션 불일치 오류가 발생합니다. 하이드레이션 불일치는 주로 다음과 같은 원인으로 발생합니다:

1. 템플릿에 잘못된 HTML 중첩 구조가 포함되어 있고, 브라우저의 기본 HTML 파싱 동작에 의해 렌더링된 HTML이 "수정"된 경우. 예를 들어, `<div>` 는 `<p>` 안에 올 수 없다는 점이 흔한 함정입니다:

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

   서버 렌더링된 HTML에서 이런 구조가 나오면, 브라우저는 `<div>` 를 만나는 순간 첫 번째 `<p>` 를 종료하고 다음과 같은 DOM 구조로 파싱합니다:

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

와 클라이언트에서 각각 한 번씩 실행되므로, 무작위 값이 두 번의 실행에서 동일하다는 보장이 없습니다. 무작위 값으로 인한 불일치를 피하는 방법은 두 가지입니다:

1. `v-if` + `onMounted` 를 사용해 무작위 값에 의존하는 부분을 클라이언트에서만 렌더링합니다. 프레임워크에 따라 이를 쉽게 해주는 내장 기능이 있을 수 있습니다. 예를 들어 VitePress의 `<ClientOnly>` 컴포넌트가 있습니다.

2. 시드를 지원하는 난수 생성 라이브러리를 사용하고, 서버 실행과 클라이언트 실행이 동일한 시드를 사용하도록 보장합니다(예: 시드를 직렬화된 상태에 포함하고 클라이언트에서 가져오기).

3. 서버와 클라이언트가 서로 다른 시간대에 있는 경우. 때로는 타임스탬프를 사용자의 로컬 시간으로 변환하고 싶을 수 있습니다. 하지만 서버 실행 시점의 시간대와 클라이언트 실행 시점의 시간대가 항상 같지 않으며, 서버 실행 시점에 사용자의 시간대를 신뢰성 있게 알 수 없습니다. 이런 경우, 로컬 시간 변환도 클라이언트 전용 작업으로 처리해야 합니다.

Vue가 하이드레이션 불일치를 감지하면, 자동으로 복구를 시도하고 프리렌더된 DOM을 클라이언트 사이드 상태와 일치하도록 조정합니다. 이 과정에서 잘못된 노드가 폐기되고 새로운 노드가 마운트되므로 렌더링 성능이 일부 저하될 수 있지만, 대부분의 경우 앱은 정상적으로 동작합니다. 그럼에도 불구하고, 개발 중에는 하이드레이션 불일치를 제거하는 것이 가장 좋습니다.

하이드레이션 불일치 억제 3.5+

Vue 3.5+에서는 `data-allow-mismatch` 속성을 사용해 불가피한 하이드레이션 불일치를 선택적으로 억제할 수 있습니다.

커스텀 디렉티브

대부분의 커스텀 디렉티브는 직접 DOM을 조작하므로, SSR 중에는 무시됩니다. 하지만 커스텀 디렉티브가 어떻게 렌더링되어야 하는지(즉, 렌더링된 엘리먼트에 어떤 속성을 추가해야 하는지)를 지정하고 싶다면, `getSSRProps` 디렉티브 훅을 사용할 수 있습니다:

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

텔레포트

텔레포트는 SSR 중에 특별한 처리가 필요합니다. 렌더링된 앱에 텔레포트가 포함되어 있으면, 텔레포트된 콘텐츠는 렌더링된 문자열에 포함되지 않습니다. 더 쉬운 해결책은 마운트 시점에 텔레포트를 조건부로 렌더링하는 것입니다.

텔레포트된 콘텐츠의 하이드레이션이 필요한 경우, ssr context 객체의 `teleports` 속성으로 노출됩니다:

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

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

최종 페이지 HTML에서 텔레포트 마크업을 올바른 위치에 삽입해야 하며, 이는 메인 앱 마크업을 삽입하는 것과 유사합니다.

> ⓘ TIP
>
> 텔레포트와 SSR을 함께 사용할 때는 `body` 를 타겟팅하지 마세요. 일반적으로 `<body>` 에는 다른 서버 렌더링 콘텐츠가 포함되어 있어, 텔레포트가 하이드레이션의 올바른 시작 위치를 결정할 수 없습니다.
>
> 대신, 오직 텔레포트된 콘텐츠만 포함하는 전용 컨테이너(예: `<div id="teleported"></div>`)를 사용하는 것이 좋습니다.

GitHub에서 이 페이지 편집

‹ Previous
테스트

Next ›
프로덕션 배포