# Blazor와 프론트엔드 생태계의 조화

심윤섭 (닷넷데브)

# 발표 주제

+ Blazor를 사용하는데 JS가 필요한 이유

+ Blazor에서 JS파일을 가져오는 방법 소개

+ JS를 잘 사용하기 위한 도구 소개

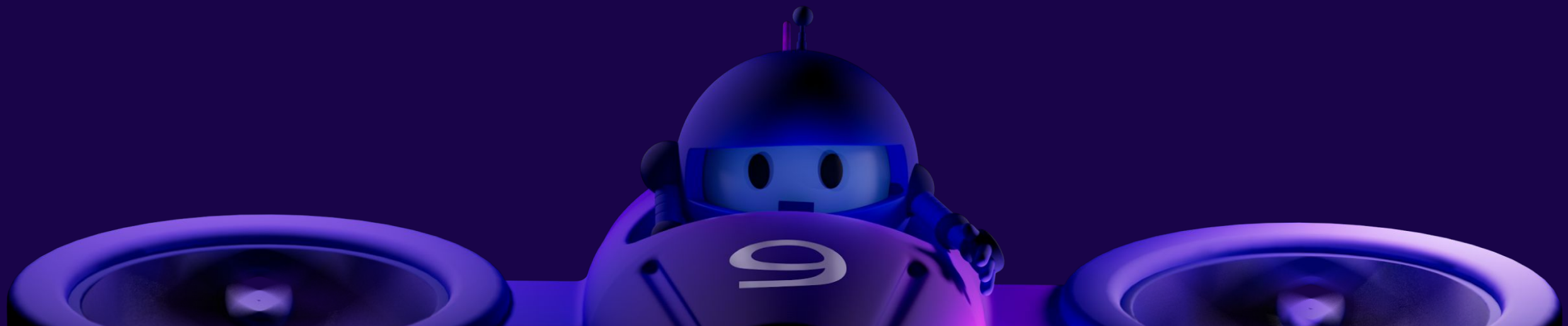
+ 적용해 본 예시 프로젝트 소개

- 닷넷 내부 코드 분석

- 라이브 코딩

# 01

## Blazor?



# Blazor는
# C#을 사용하여
# 대화형 UI를 만들 수 있는
# .NET 웹 프레임워크다.

라고 공식문서에 말합니다.

Blazor는
C#을 사용
대화형 U 을 수 있는
.NET 웹 프레임워크다.

라고 공식문서에 말합니다.

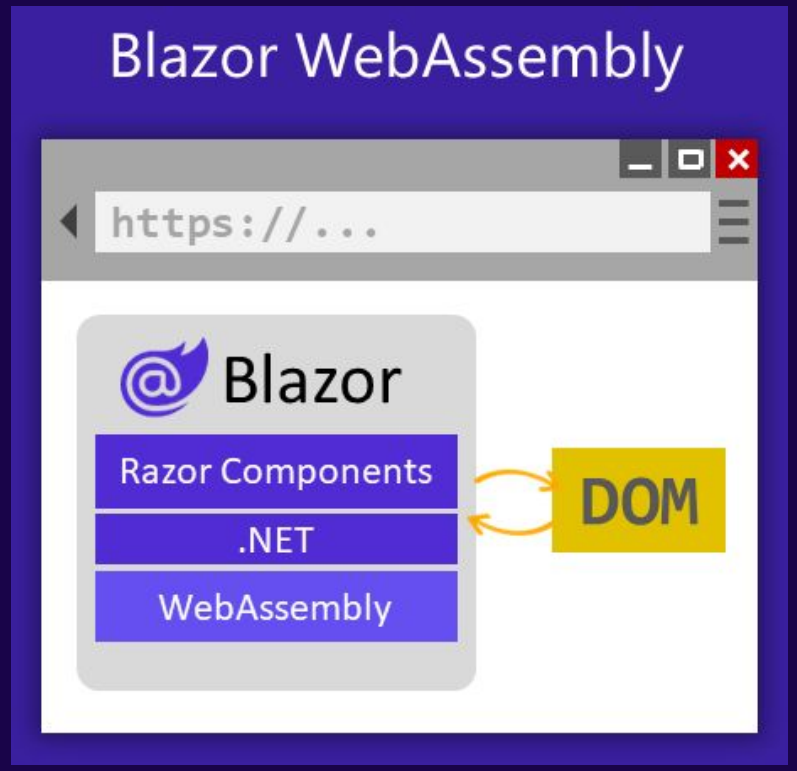
# Blazor의 호스팅 모델

## Blazor WASM

## Blazor Server

# Blazor의 호스팅 모델

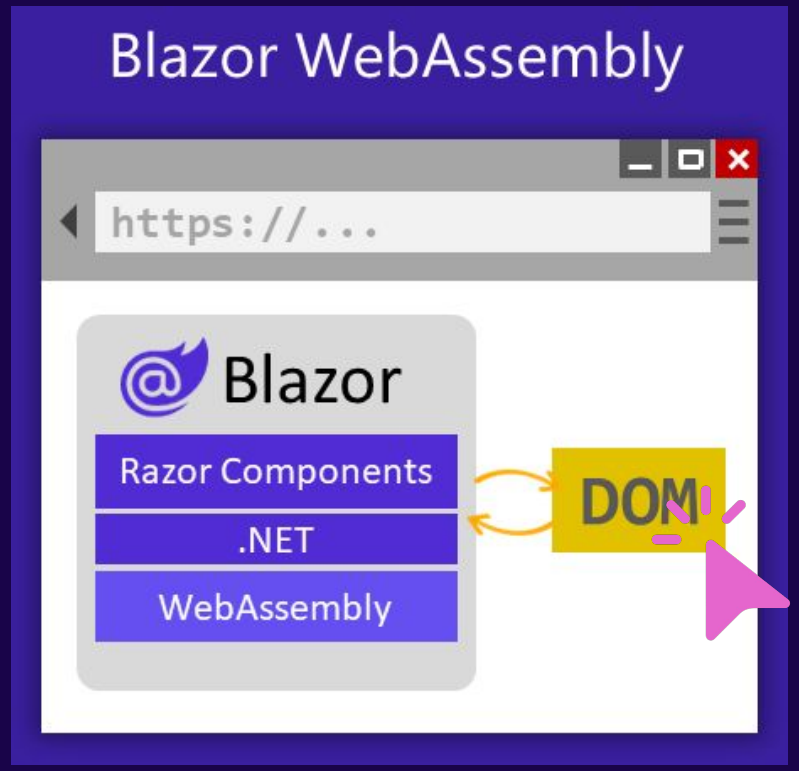
## Blazor WASM




## Blazor Server

# Blazor의 호스팅 모델

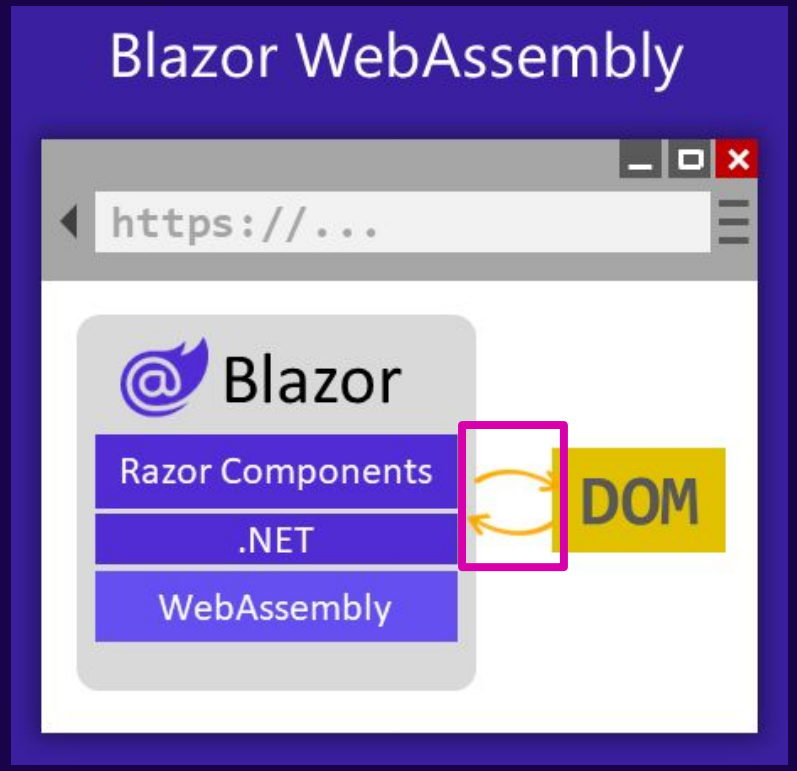
## Blazor WASM




## Blazor Server

# Blazor의 호스팅 모델

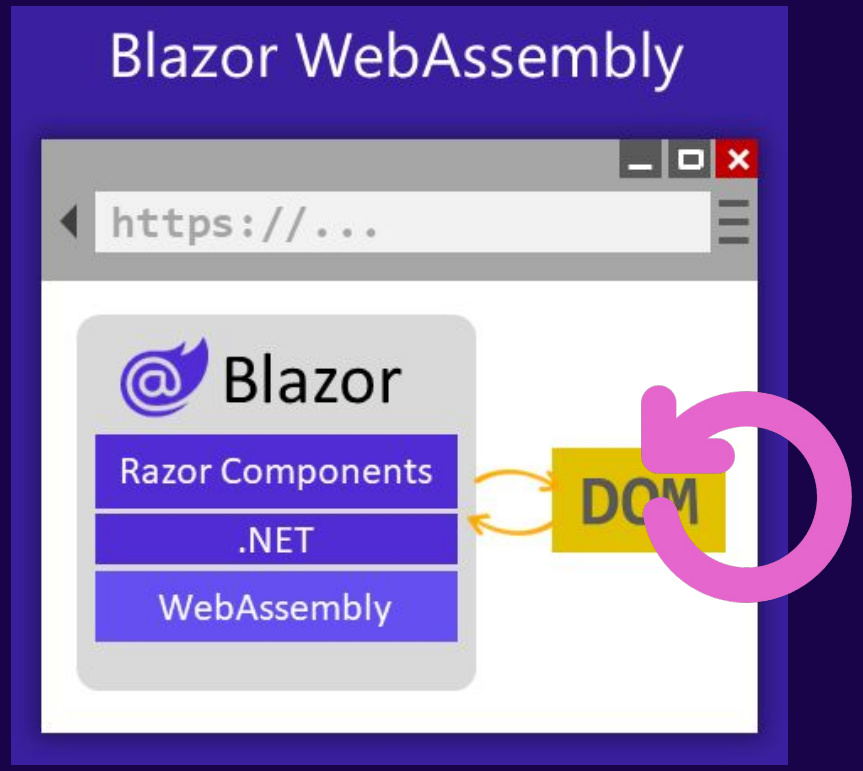
## Blazor WASM




## Blazor Server

# Blazor의 호스팅 모델

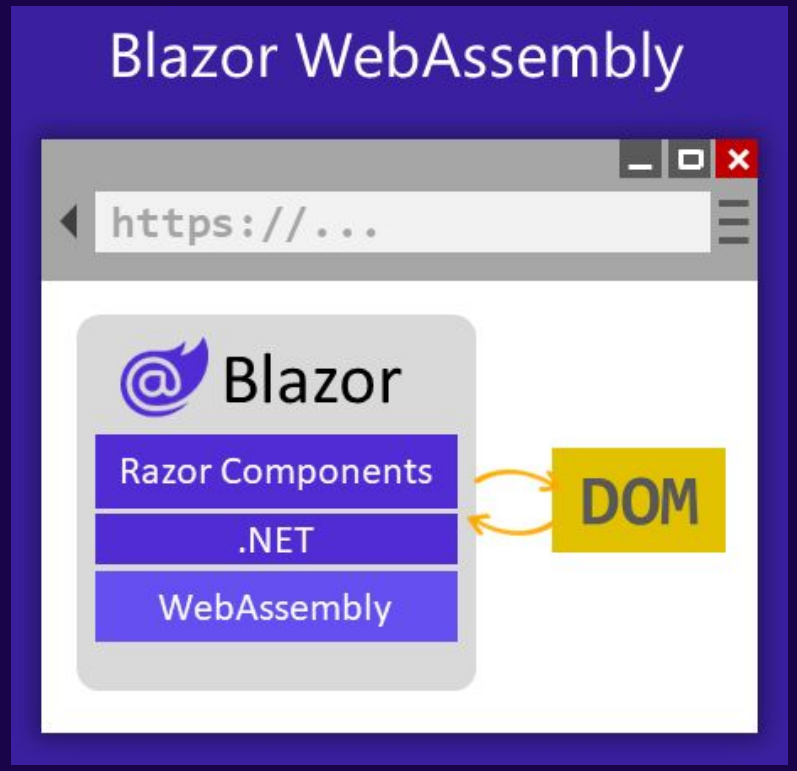
## Blazor WASM



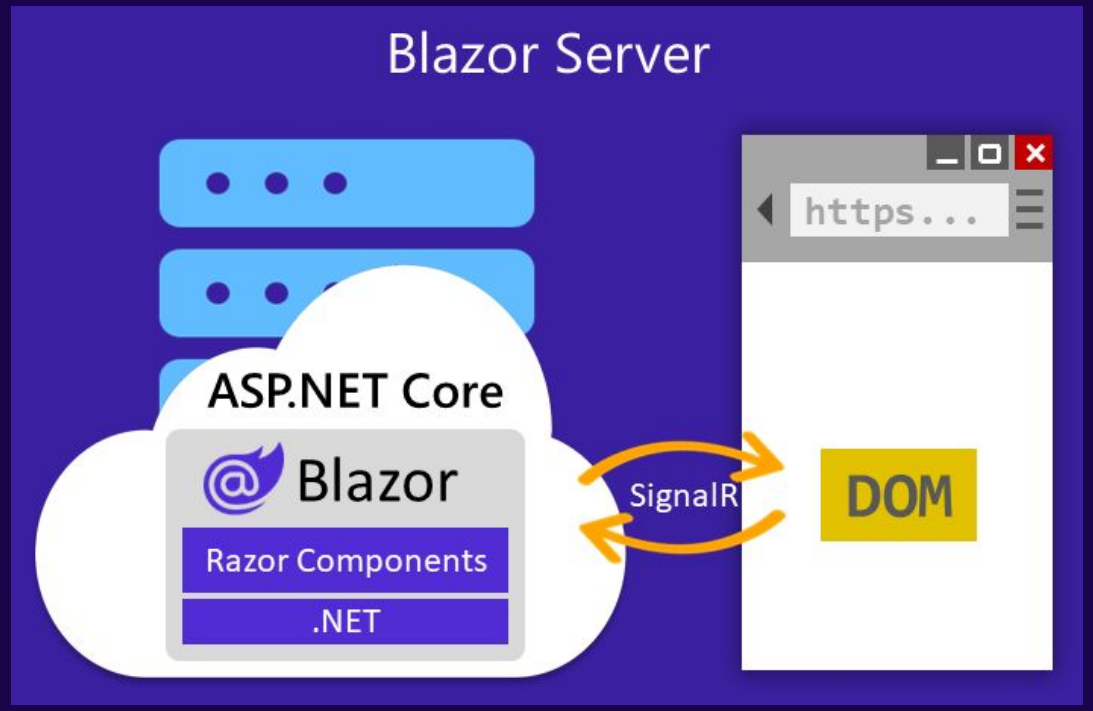

## Blazor Server

# Blazor의 호스팅 모델

## Blazor WASM

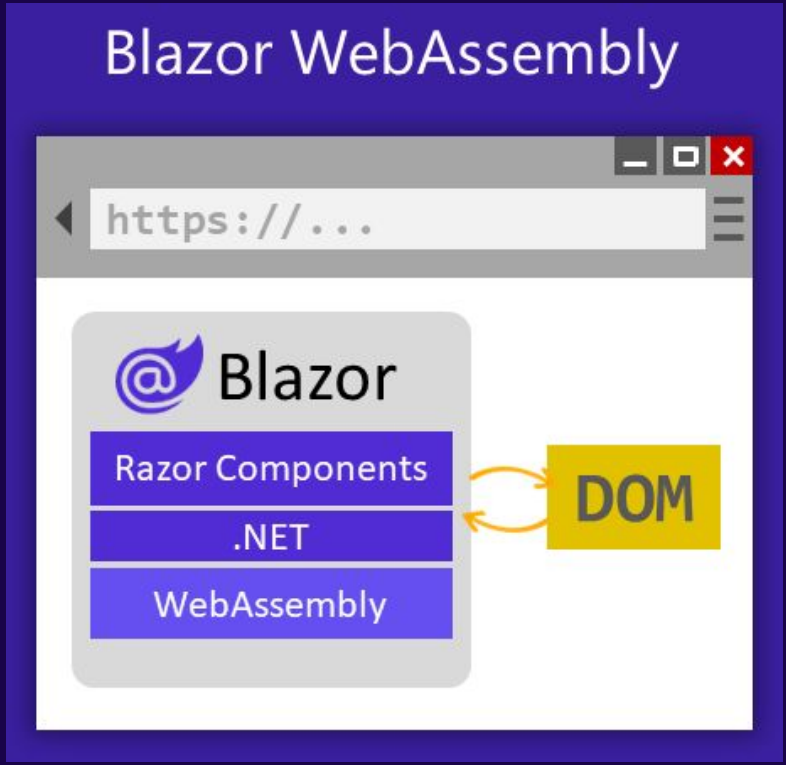



## Blazor Server

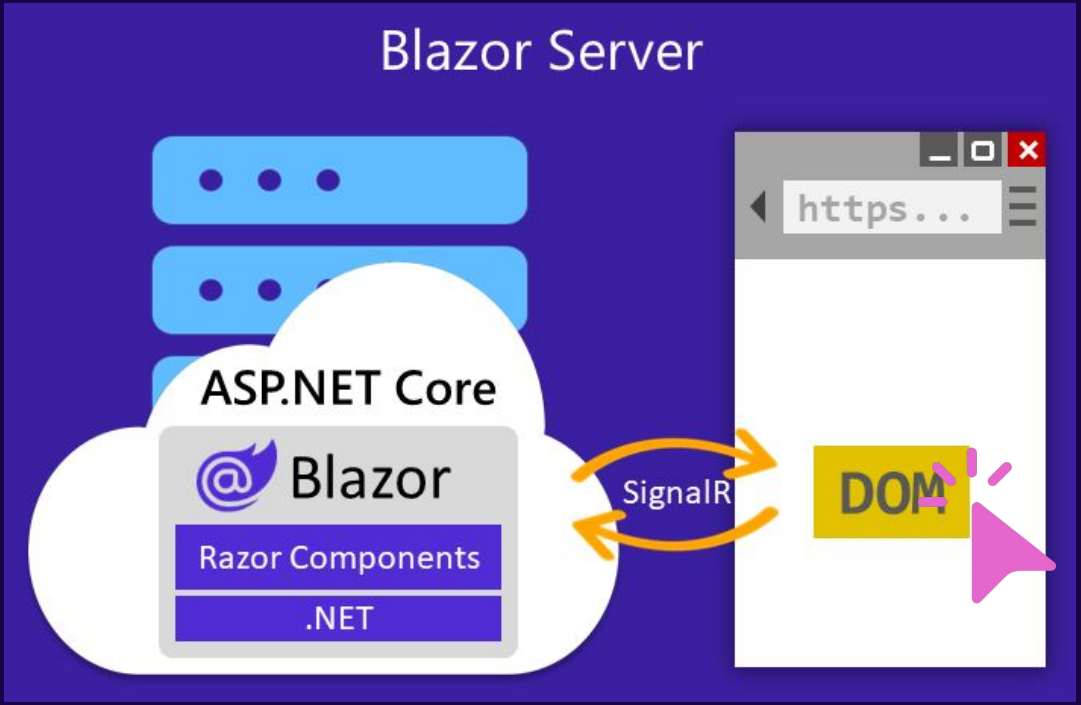


# Blazor의 호스팅 모델

## Blazor WASM

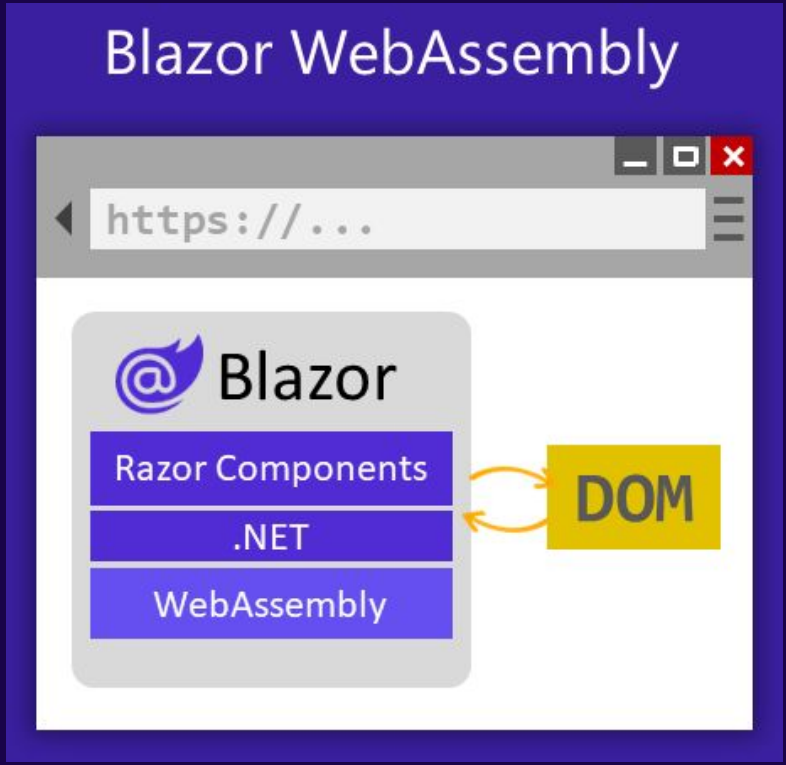



## Blazor Server

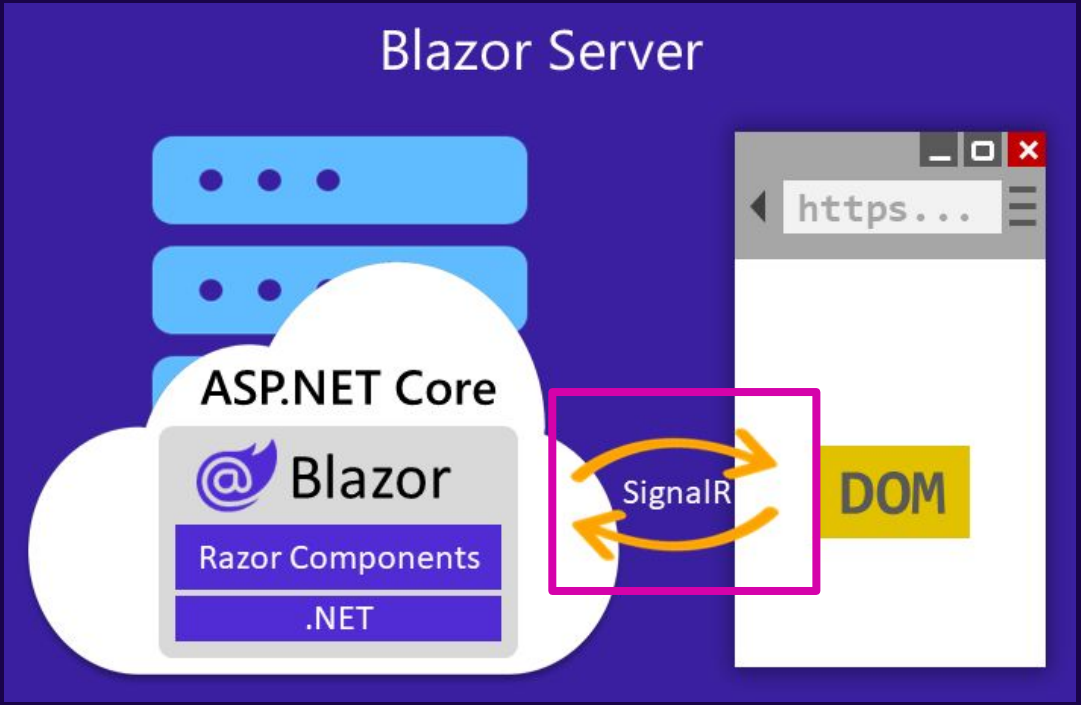


# Blazor의 호스팅 모델

## Blazor WASM

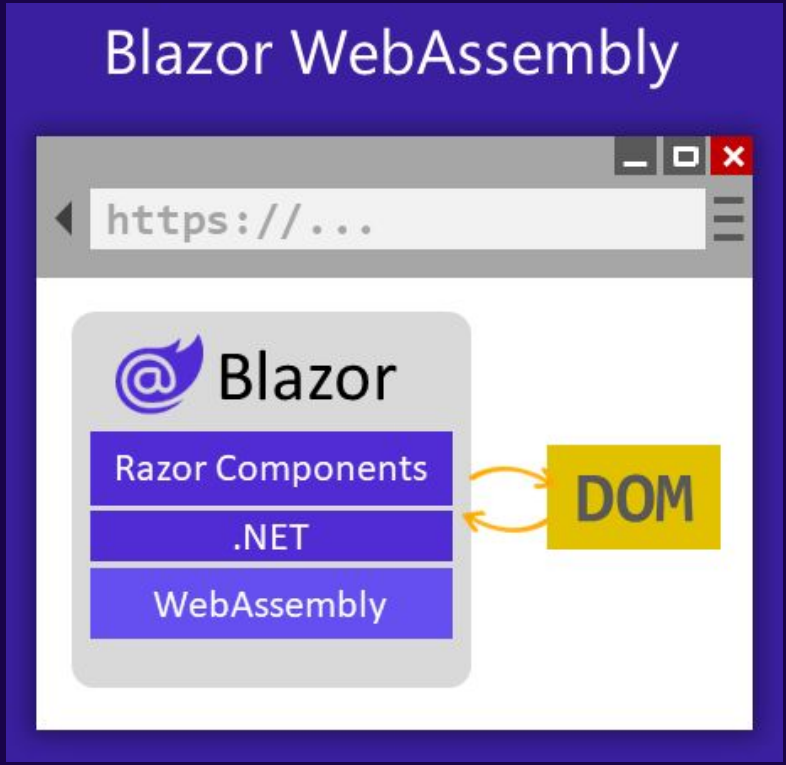



## Blazor Server

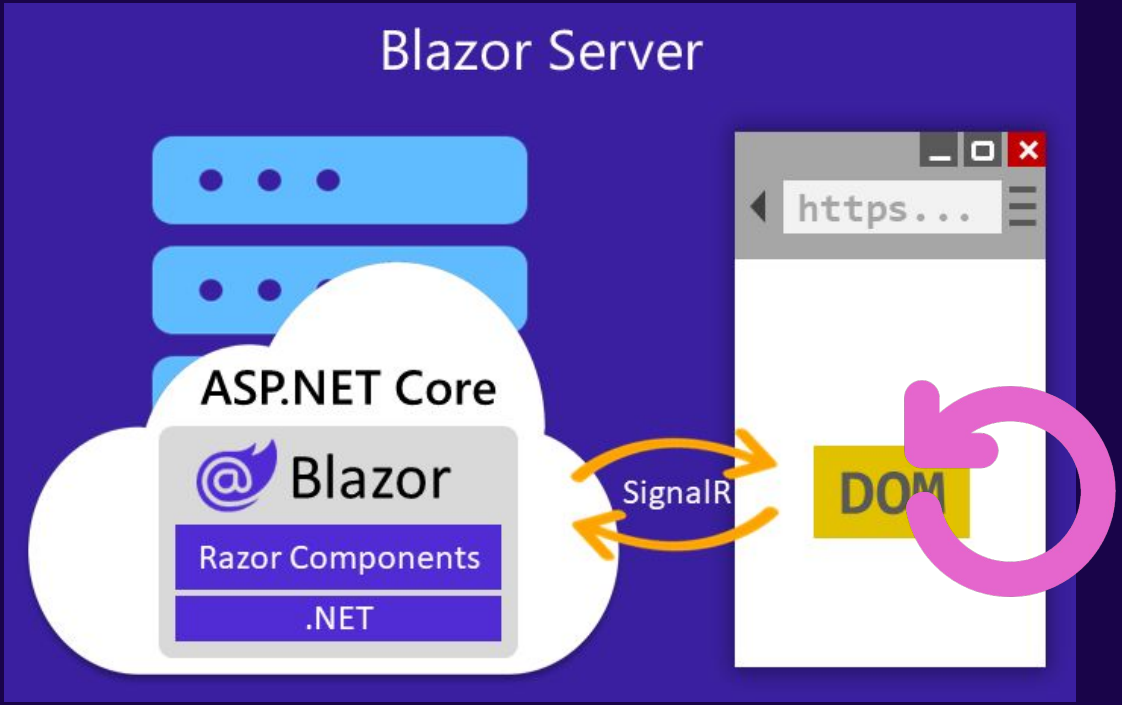


# Blazor의 호스팅 모델

## Blazor WASM

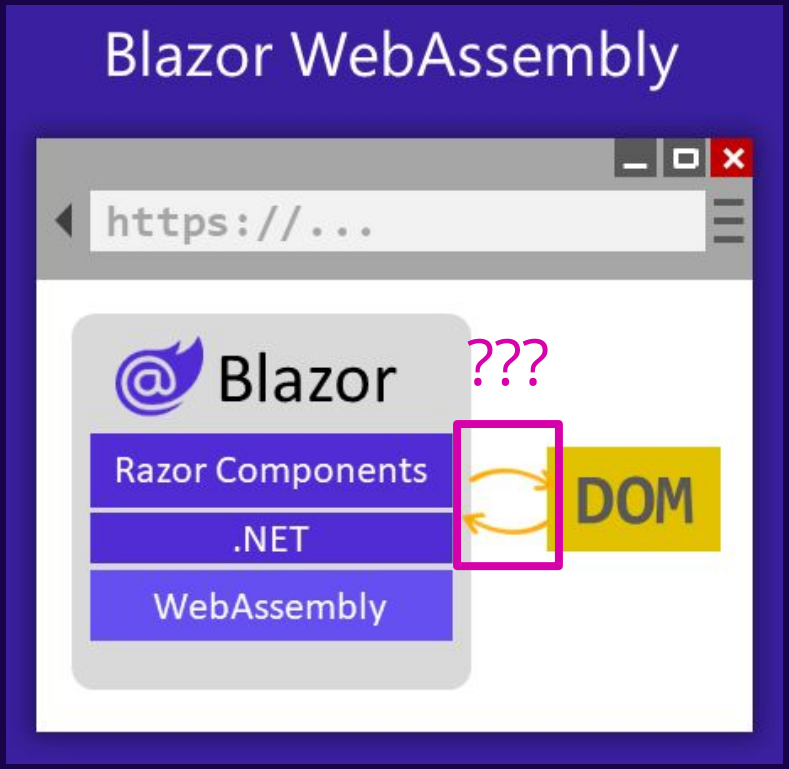



## Blazor Server

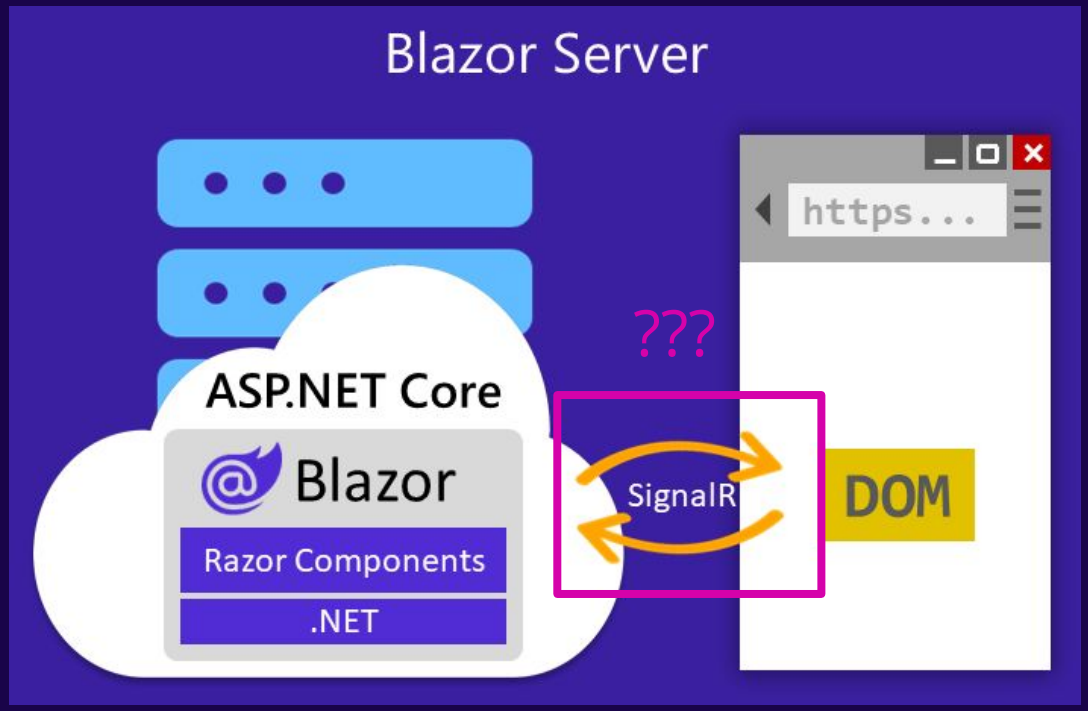


# Blazor의 호스팅 모델

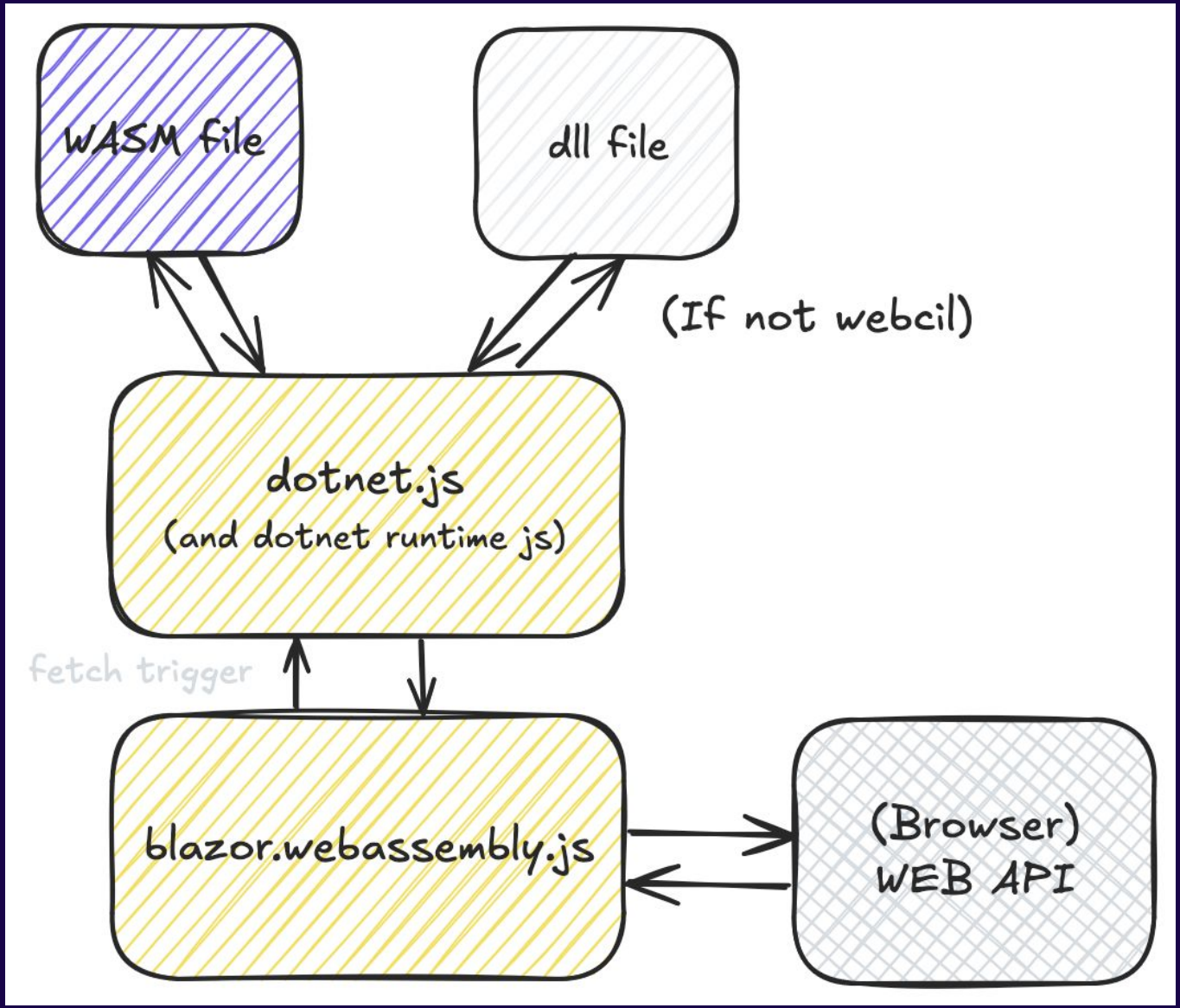
## Blazor WASM



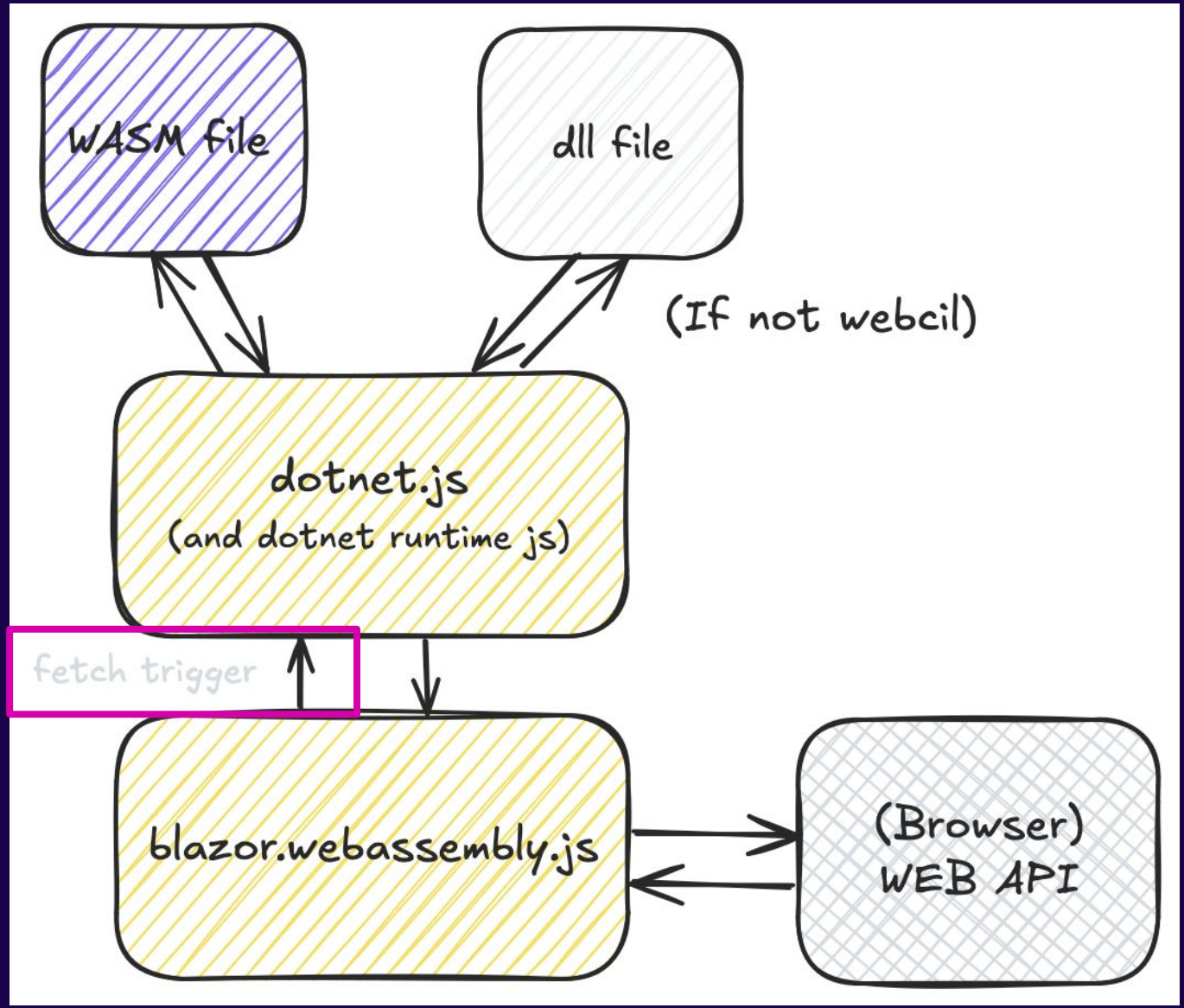

## Blazor Server



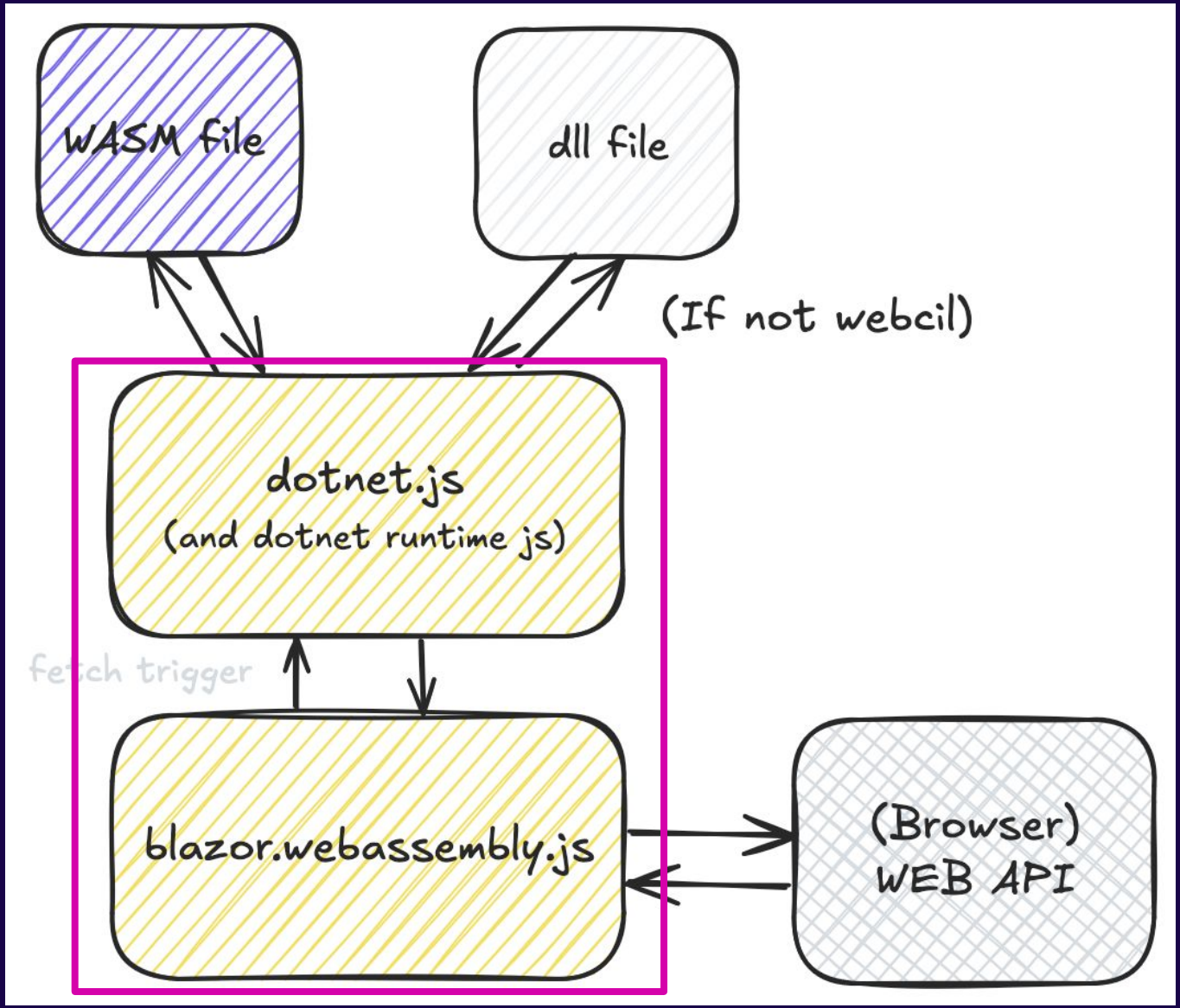
# Blazor WASM의 내부



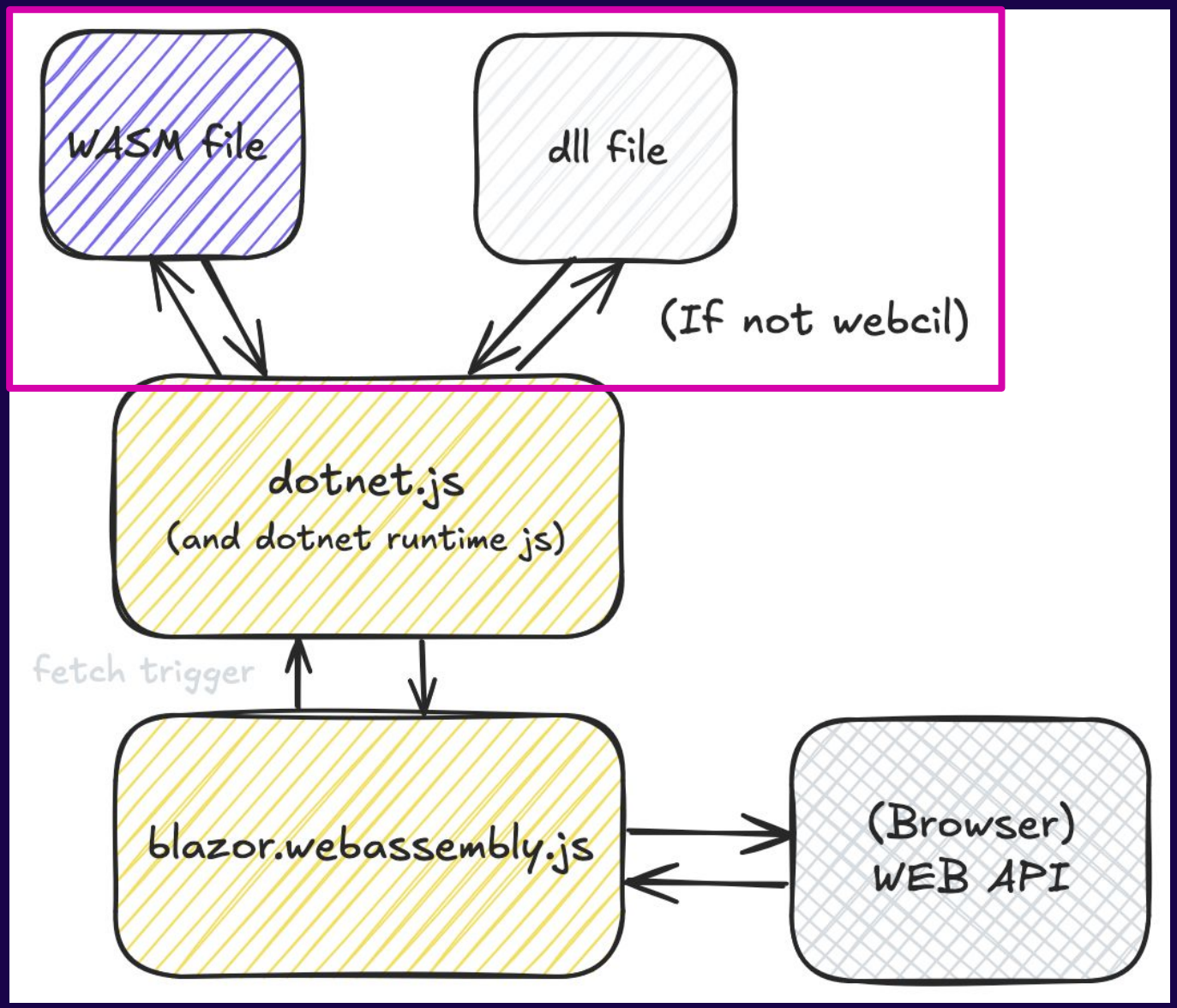
# Blazor WASM의 내부



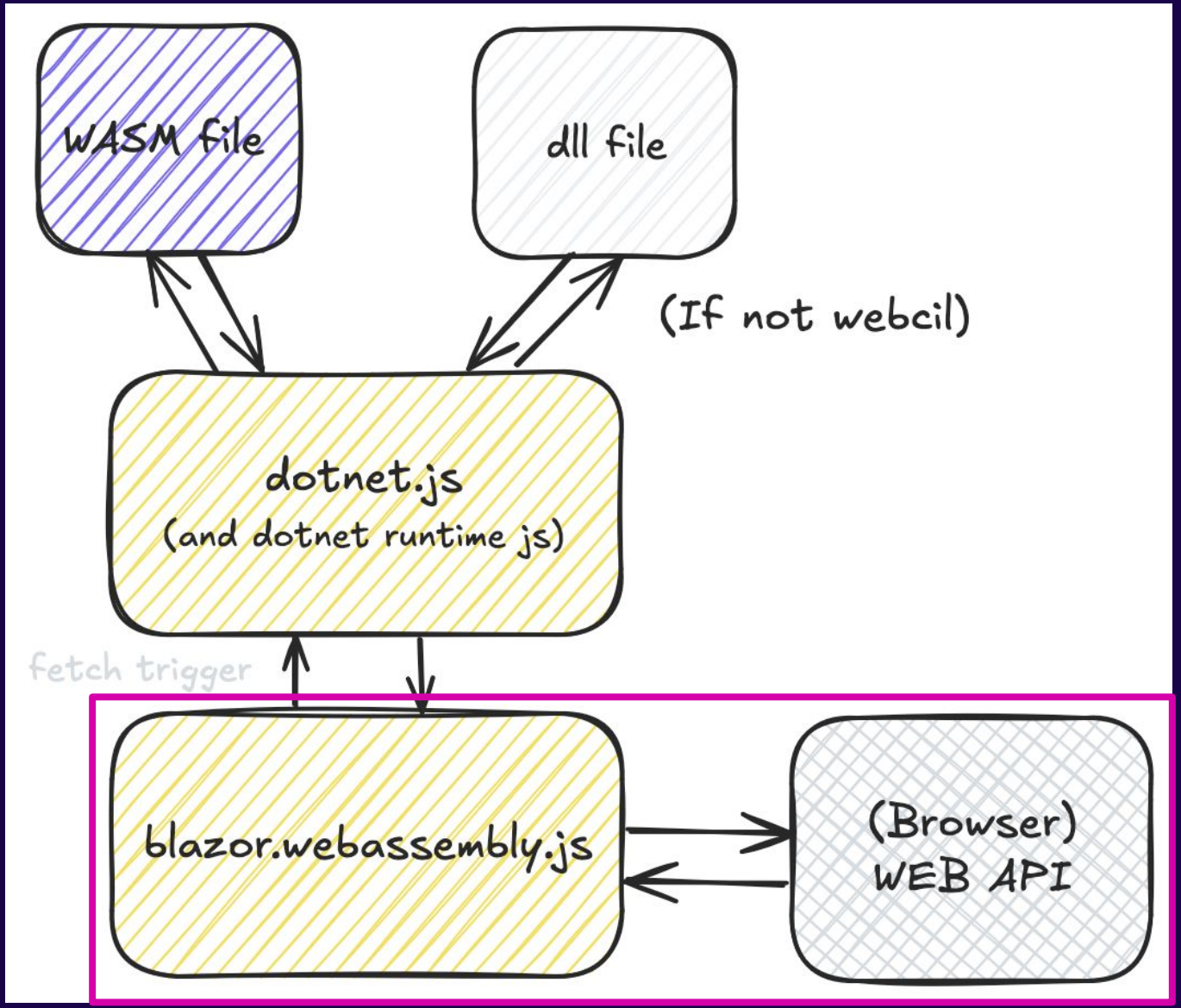
# Blazor WASM의 내부



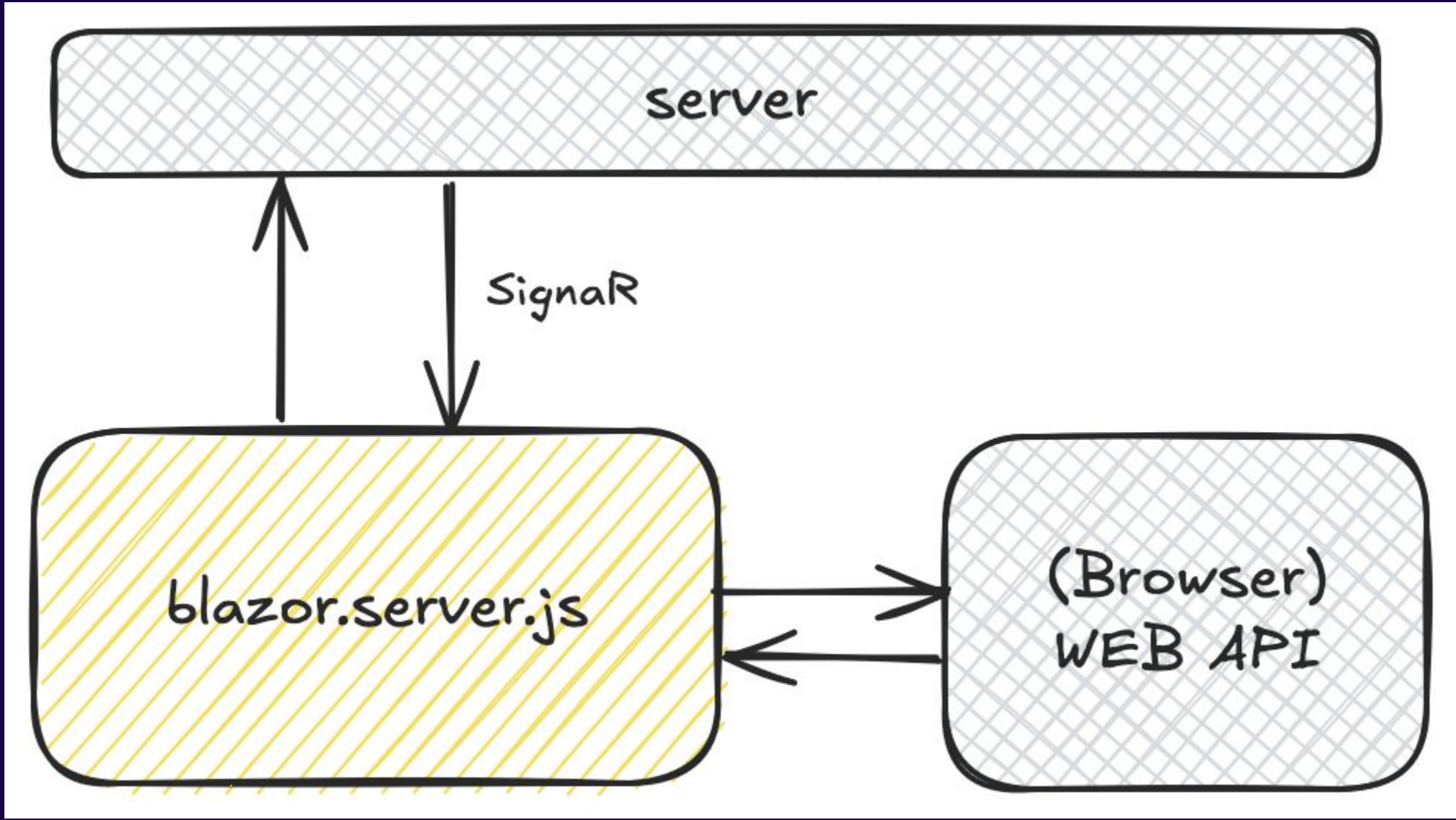
# Blazor WASM의 내부



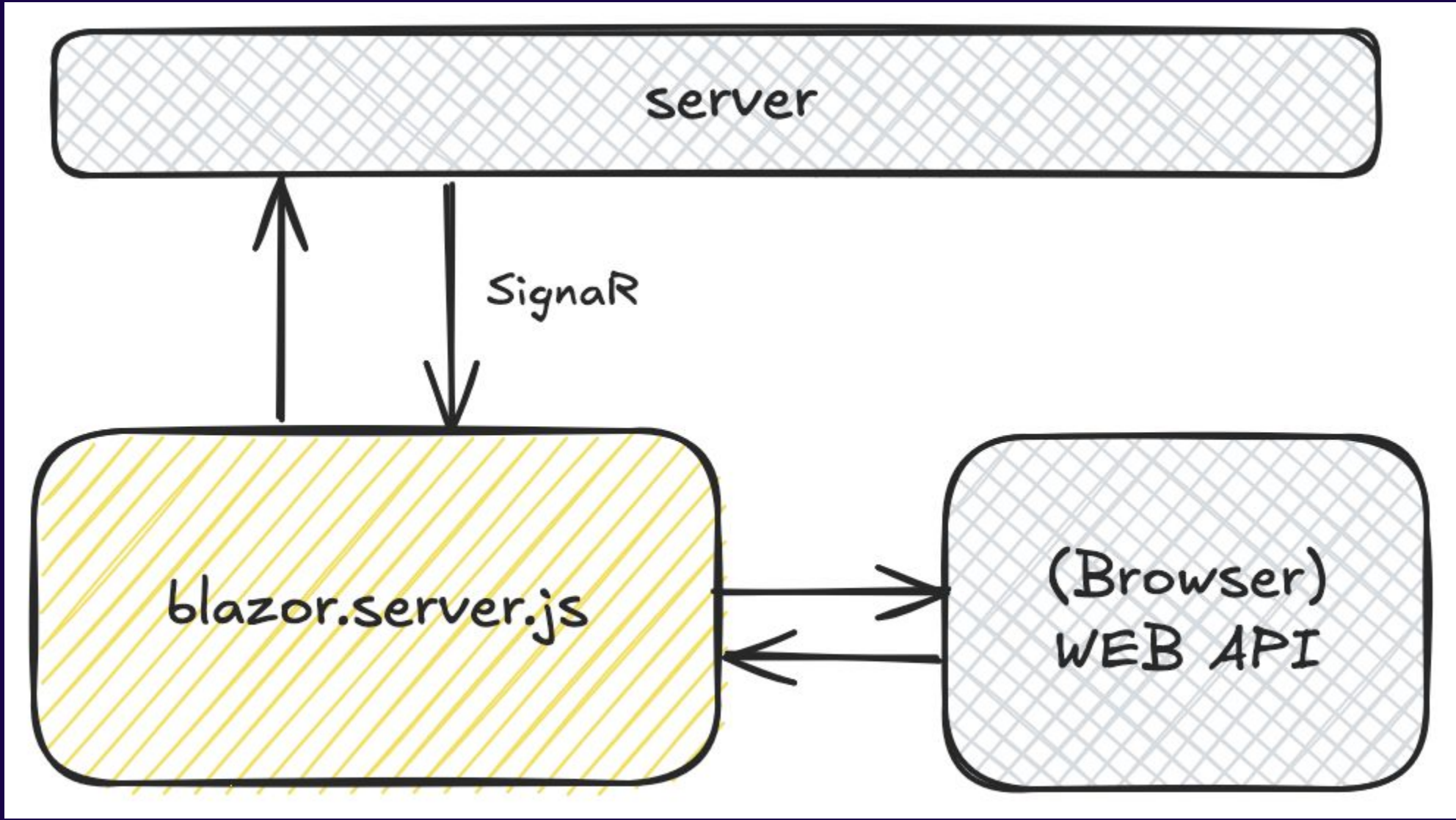
# Blazor WASM의 내부



# Blazor Server의 내부

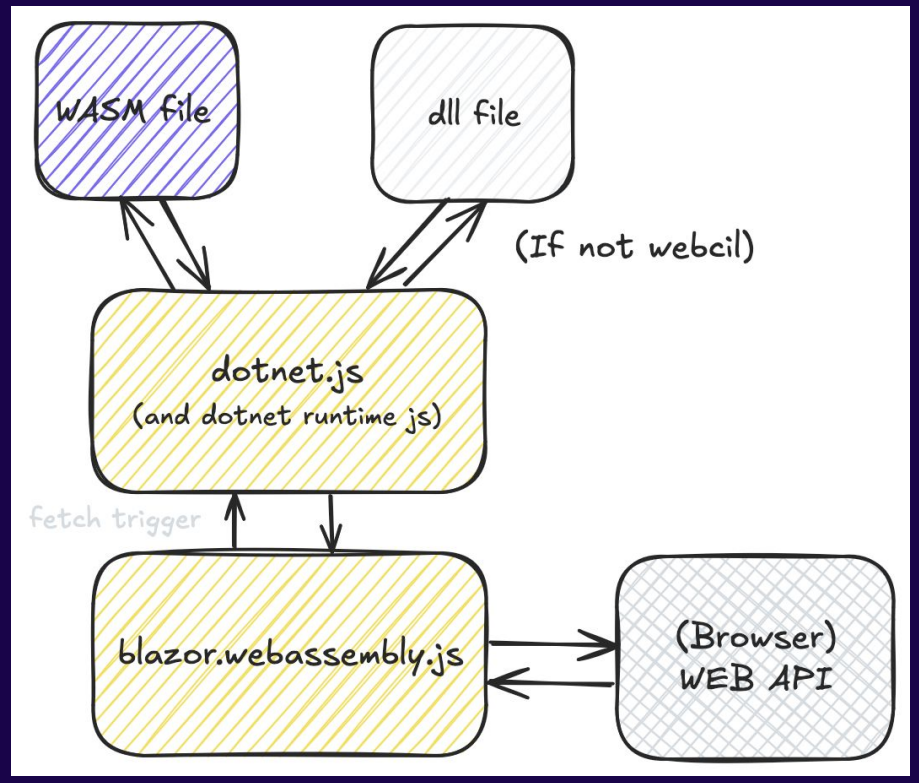


# Blazor Server의 내부

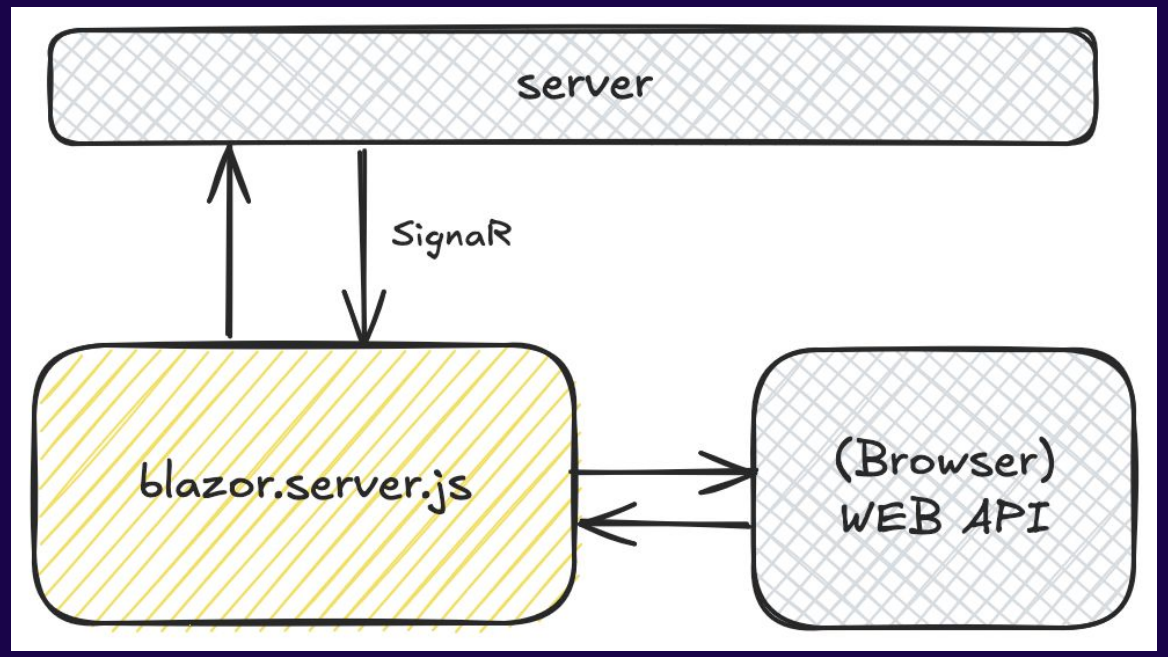


# Blazor의 호스팅 모델

## Blazor WASM




## Blazor Server



# WEB API는 정말 많다

## Web APIs

When writing code for the Web, there are a large number of Web APIs available. Below is a list of all the APIs and interfaces (object types) that you may be able to use while developing your Web app or site.

Web APIs are typically used with JavaScript, although this doesn't always have to be the case.

## Specifications

This is a list of all the APIs that are available.

### A

- Attribution Reporting API
- Audio Output Devices API

### B

- Background Fetch API
- Background Synchronization API
- Background Tasks API
- Badging API
- Barcode Detection API
- Battery Status API
- Beacon API
- Web Bluetooth API
- Broadcast Channel API

### C

- CSS Custom Highlight API
- CSS Font Loading API
- CSS Painting API
- CSS Properties and Values API
- CSS Typed Object Model API
- CSS Object Model (CSSOM)
- Canvas API
- Channel Messaging API
- Clipboard API
- Compression Streams API
- Compute Pressure API
- Console API
- Contact Picker API
- Content Index API
- Cookie Store API
- Credential Management API

### D

- Document Object Model (DOM)
- Device Memory API
- Device orientation events
- Document Picture-in-Picture API

### E

- EditContext API
- Encoding API
- Encrypted Media Extensions API
- EyeDropper API

### F

- Federated Credential Management (FedCM) API
- Fenced Frame API
- Fetch API
- File API
- File System API
- File and Directory Entries API
- Force Touch events
- Fullscreen API

### G

- Gamepad API
- Geolocation API
- Geometry interfaces

### H

- The HTML DOM API
- HTML Drag and Drop API
- History API
- Houdini APIs

### I

- Idle Detection API
- MediaStream Image Capture API
- IndexedDB API
- Ink API
- InputDeviceCapabilities API
- Insertable Streams for MediaStreamTrack API
- Intersection Observer API
- Invoker Commands API

### K

- Keyboard API

### L

- Launch Handler API
- Local Font Access API

### M

- Media Capabilities API
- Media Capture and Streams API (Media Stream)
- Media Session API
- Media Source API
- MediaStream Recording API

### N

- Navigation API
- Network Information API

### P

- Page Visibility API
- Payment Handler API
- Payment Request API
- Performance APIs
- Web Periodic Background Synchronization API
- Permissions API
- Picture-in-Picture API
- Pointer events
- Pointer Lock API
- Popover API
- Presentation API
- Prioritized Task Scheduling API
- Push API

……

# 결국 JS가 필요합니다.

**Blazor에서 JS를 불러오는 법, JS 생태계 통합법을 알아보죠.**

# 02

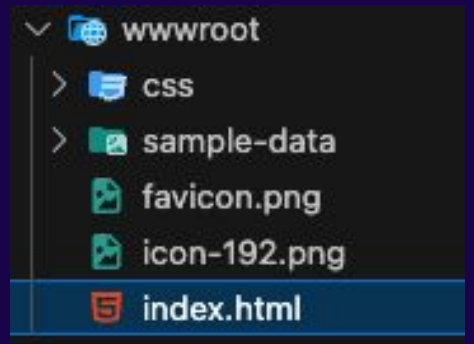
## Blazor에서 JS를 불러오기

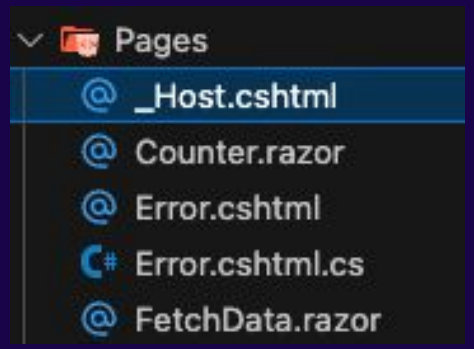
# Blazor에서 JS를 불러오기

<script>태그로 가져오기

IJSRuntime을 통해 가져오기

# <script> 태그로 가져오기




```html
<html>
<body>
  <script src="_framework/blazor.{ENVIRONMENT}.js"></script>
  <script src="{PATH}/target.js"></script>
</body>
</html>
```

# <script> 태그로 가져오기

```
@inject IJSRuntime JS

@code {
  protected override async Task OnInitializedAsync()
  {
    await JS.InvokeAsync(
      "triggerEvent",
      parameter
    );
  }
}
```

# <script> 태그로 가져오기

+ 설정하기 간편하다

+ 직관적으로 만들 수 있다

# <script> 태그로 가져오기

+ 설정하기 간편하다

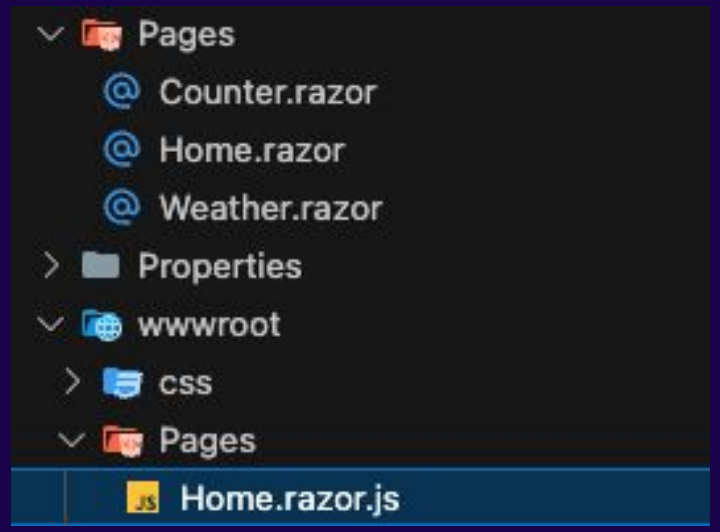
+ 직관적으로 만들 수 있다

- 파일 크기가 커지면 분리가 힘들다

- .NET 호출시 네임스페이스 이름 관리가 힘들다

- 네임스페이스(window) 오염을 관리해야 한다

# IJSRuntime을 통해 가져오기




```
@inject IJSRuntime JS

@code {
  protected override async Task OnInitializedAsync()
  {
    jsUtils = await JS.InvokeAsync<IJSObjectReference>(
      "import",
      "{PATH}/{COMPONENT}.razor.js"
    );
  }
}
```

# IJSRuntime을 통해 가져오기

## 구성 요소와 함께 배치된 외부 JavaScript 파일(`.js`)에서 스크립트 로드

구성 요소에 대한 JS JavaScript(Razor) 파일의 배치는 앱에서 스크립트를 구성하는 편리한 방법입니다.

Razor앱의 Blazor 구성 요소는 확장 프로그램을 JS 사용하여 파일을 정렬 `.razor.js` 하고 프로젝트의 파일 경로를 사용하여 공개적으로 주소를 지정합니다.

`{PATH}/{COMPONENT}.razor.js`

- `{PATH}` 자리 표시자는 구성 요소의 경로입니다.
- `{COMPONENT}` 자리 표시자가 구성 요소입니다.

앱이 게시되면 프레임워크는 자동으로 스크립트를 웹 루트로 이동합니다. 스크립트는 자리 표시자가 있는 위치로 이동 `bin/Release/{TARGET FRAMEWORK MONIKER}/publish/wwwroot/{PATH}/{COMPONENT}.razor.js` 됩니다.

- `{TARGET FRAMEWORK MONIKER}` 는 TFM(대상 프레임워크 모니커)입니다.
- `{PATH}` 는 구성 요소의 경로입니다.
- `{COMPONENT}` 는 구성 요소 이름입니다.

https://learn.microsoft.com/ko-kr/aspnet/core/blazor/javascript-interoperability/location-of-javascript#load-a-script-from-an-external-javascript-file-js-collocated-with-a-component

# IJSRuntime을 통해 가져오기

```
@inject IJSRuntime JS

@code {
  protected override async Task OnInitializedAsync()
  {
    // ...
    await jsUtils.InvokeAsync("triggerEvent", parameter);
  }
}
```

# IJSRuntime을 통해 가져오기

+ 필요한 JS파일만 가져올 수 있다

+ 네임스페이스/함수 이름 관리가 쉽다

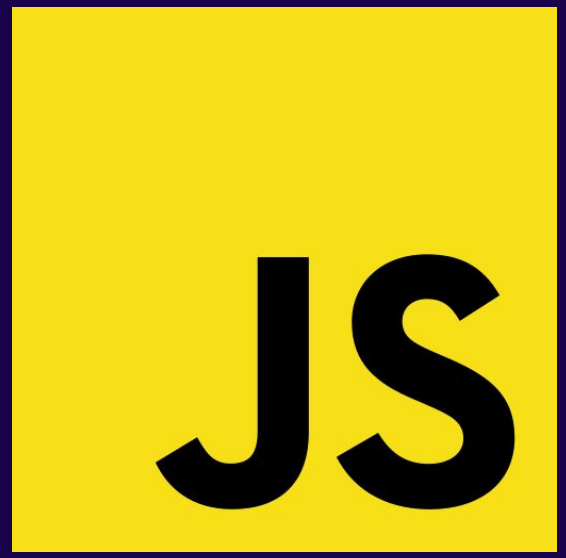
- 설정하기 어렵다

- 직관적이지 않다

# Blazor에서 JS를 불러오기

<script>태그로 가져오기

IJSRuntime을 통해 가져오기



# 03

## JS 생태계 입문하기

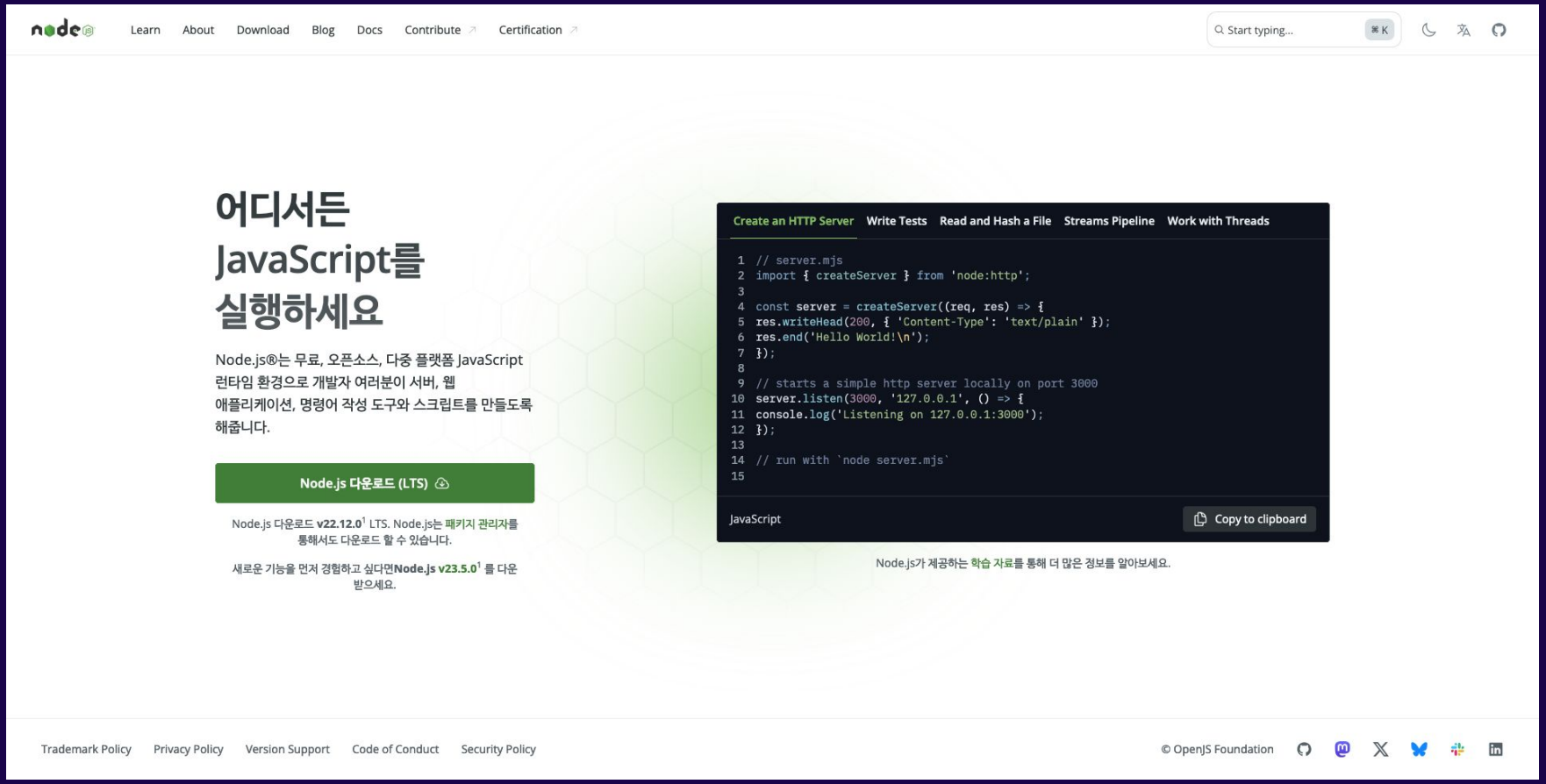
등장하는 코드는 나중에 공유될 발표자료를 참고하세요!

# NodeJS 설치하기

개발환경 권장

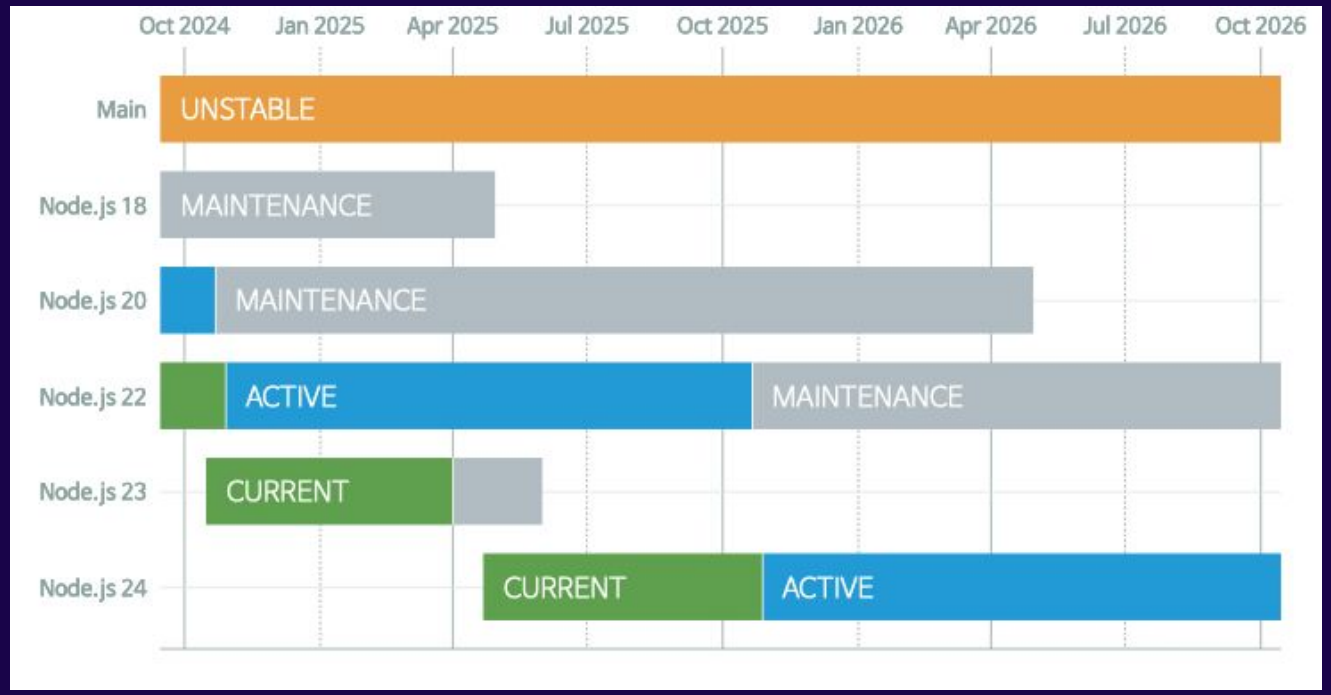
# NodeJS 설치하기

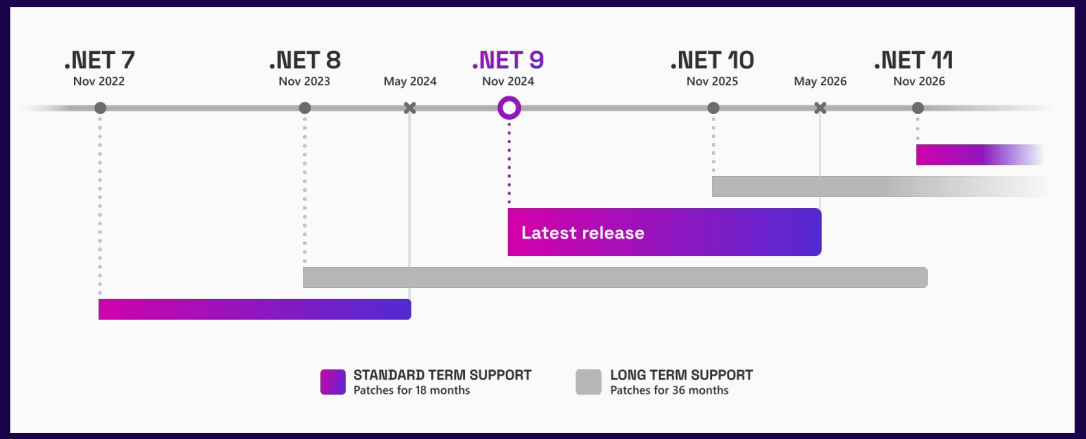
개발환경 권장



# NodeJS 설치하기

개발환경 권장



# NodeJS 설치하기

개발환경 권장

nvm Public

+ 많이 사용한다

- Unix기반만 사용 가능하다

volta Public

+ Unix/Windows 모두 사용 가능하다

- 비교적 신생 도구

? 패키지 관리자도 관리한다(추후 소개할 corepack 사용 불가)

fnm Public

+ Unix/Windows 모두 사용 가능하다

- 비교적 신생 도구

# 패키지 관리자 선택하기

프로덕션 선택

# 패키지 관리자 선택하기

프로덕션 선택

npm

# 패키지 관리자 선택하기

프로덕션 선택

yarn berry(v1 아님)

+ 성능이 대부분의 경우 가장 좋다

- Plugin'n'Play(PnP)의 오류 해결이 쉽지 않다

? Zero install의 장단점이 극명하다(disable 가능)

pnpm

+ 성능이 npm에 비해 대부분의 경우 좋다

+ npm에서 교체하기 쉽다

- npm보다 제약사항이 생긴다(ex. 패키지 내 패키지 사용 어려움)

# 패키지 관리자 선택하기

프로덕션 선택

## corepack

OpenJS Slack #nodejs-corepack

Corepack is a zero-runtime-dependency Node.js script that acts as a bridge between Node.js projects and the package managers they are intended to be used with during development. In practical terms, **Corepack lets you use Yarn, npm, and pnpm without having to install them.**

```bash
#!/bin/bash

# corepack 활성화
corepack enable

# 패키지 매니저 버전 고정
corepack use pnpm@lateset

# 사용
pnpm install
```

```bash
#!/bin/bash

# corepack 활성화
corepack enable

# yarn 초기화
yarn init -2

# 사용
yarn install
```

# 번들러 사용하기

프로덕션 권장

```html
<html>
<body>
  <script src="_framework/blazor.{ENVIRONMENT}.js"></script>
  <script src="{PATH}/target.js"></script>
</body>
</html>
```

```razor
@inject IJSRuntime JS

@code {
  protected override async Task OnInitializedAsync()
  {
    jsUtils = await JS.InvokeAsync<IJSObjectReference>(
      "import",
      "{PATH}/{COMPONENT}.razor.js"
    );
  }
}
```

# 번들러 사용하기

프로덕션 권장

tsup

npm v8.3.5 downloads 4M/month

Bundle your TypeScript library with no config, powered by esbuild.

tsup(권장)

+ 설정이 간편하고 빠르다

- 비교적 신생 라이브러리이며, ESBuild 기반으로 만들어져 디버깅이 어려울 수 있다

webpack

+ 오래되고 많이 써서 자료가 매우 많다

- 느리고 설정이 복잡하다

esbuild

+ 빠르다

- 설정이 복잡하다.

- 디버깅이 어려울 수 있다

Vite

? 라이브러리 모드가 지원이 부족하여 권장하지 않습니다

# 번들러 사용하기

프로덕션 권장

```
#!/bin/sh

# JS 파일을 모두 import하기 위해서 설치
pnpm add -D tinyglobby

# 사용하려는 번들러 라이브러리 설치
pnpm add -D tsup
```

# 번들러 사용하기

프로덕션 권장

```
// {최상단}/tsup.config.mjs
import path from 'node:path';
import { defineConfig } from 'tsup';
import { globSync } from 'tinyglobby';

export default {
  entry: globSync(
    [
      './javascript/**/*.js',
      './javascript/**/*.ts', // TypeScript
    ],
    { cwd: import.meta.dirname }
  ),
  outDir: path.resolve(
    import.meta.dirname,
    './wwwroot/js'
  ),
  format: ['esm'],
  platform: 'browser',
}
```

(중요 부분)

# 번들러 사용하기

프로덕션 권장

package.json

```
{
  // ...
  "type": "module",
  "script": {
    "build": "tsup"
  }
}
```

```
#!/bin/sh

npm run build

# pnpm일 경우
pnpm run build
```

# 번들러 사용하기

프로덕션 권장

.gitignore

```
# 빌드파일 제거
./wwwroot/js/**/*.js
```

# MSBuild에 통합하기

프로덕션 권장

# MSBuild에 통합하기

프로덕션 권장

Project.csproj

```
<Project Sdk="Microsoft.NET.Sdk.BlazorWebAssembly">
  <!-- 생략 -->
  <!-- pnpm을 통해 패키지 설치하는 과정 -->
  <Target Name="NpmInstall" Inputs="package.json" Outputs="node_modules/.install-stamp">
    <Exec
      Command="pnpm i"
      Condition="$(RestorePackagesWithLockFile) != 'true'"
      ConsoleToMsBuild="true" />

    <Touch Files="node_modules/.install-stamp" AlwaysCreate="true" />
    <PropertyGroup>
      <RestorePackagesWithLockFile>true</RestorePackagesWithLockFile>
    </PropertyGroup>
  </Target>
  <!-- 빌드 전에 번들러를 통해 빌드 -->
  <Target Name="PreBuild" DependsOnTargets="NpmInstall" BeforeTargets="BeforeBuild">
    <Exec Command="pnpm build" />
  </Target>
</Project>
```

Blazor WebAssembly + Webpack + Sass + TypeScript (.NET Conf 2022) 변형

# MSBuild에 통합하기

프로덕션 권장

Project.csproj

```
<Project Sdk="Microsoft.NET.Sdk.BlazorWebAssembly">
  <!-- 생략 -->
  <!-- pnpm을 통해 패키지 설치하는 과정 -->
  <Target Name="NpmInstall" Inputs="package.json" Outputs="node_modules/.install-stamp">
    <Exec
      Command="pnpm i"
      Condition="$(RestorePackagesWithLockFile) != 'true'"
      ConsoleToMsBuild="true" />

    <Touch Files="node_modules/.install-stamp" AlwaysCreate="true" />
    <PropertyGroup>
      <RestorePackagesWithLockFile>true</RestorePackagesWithLockFile>
    </PropertyGroup>
  </Target>
  <!-- 빌드 전에 번들러를 통해 빌드 -->
  <Target Name="PreBuild" DependsOnTargets="NpmInstall" BeforeTargets="BeforeBuild">
    <Exec Command="pnpm build" />
  </Target>
</Project>
```

Blazor WebAssembly + Webpack + Sass + TypeScript (.NET Conf 2022) 변형

# TypeScript

개발환경 선택

```
function startLottie(element) {
  element.
}
export {
  startLottie
};
```

abc element
abc startLottie

# TypeScript

개발환경 선택

```
You, 3개월 전 | 1 author (You)

function startLottie(element: HTMLDivElement) {
  element.        You, 3개월 전 • Uncommitted changes
}
                  hasPointerCapture
                  hidden
export { s        hidePopover
                  id
                  inert
                  innerHTML
                  innerText
                  inputMode
                  insertAdjacentElement
                  insertAdjacentHTML
                  insertAdjacentText
                  insertBefore
```

# TypeScript 개발환경 선택

TypeScript 다운로드 문서 핸드북 커뮤니티 플레이그라운드 도구 검색

- Get Started
  - TS for the New Programmer
  - TypeScript for JS Programmers
  - TS for Java/C# Programmers
  - TS for Functional Programmers
  - TypeScript Tooling in 5 minutes
- Handbook
- Reference
- Modules Reference
- Tutorials
- What's New
- Declaration Files
- JavaScript
- Project Configuration

## TypeScript for Java/C# Programmers

TypeScript는 C#, Java와 같이 정적 타이핑을 사용하는 언어에 익숙한 프로그래머들에게 인기 있는 선택입니다.

TypeScript의 타입 시스템은 더 나은 코드 완성, 오류의 조기 발견, 프로그램 부분 간의 더 명확한 통신과 같이 정적 타이핑이 가지는 많은 이점을 제공합니다. TypeScript는 이러한 개발자에게 친숙한 기능을 많이 제공하지만, JavaScript(또한 TypeScript도 마찬가지로)가 기존의 객체 지향 프로그래밍(OOP) 언어와 어떤 차이가 있는지 다시 살펴볼 필요가 있습니다. 이러한 차이점을 이해하는 것은 더 나은 JavaScript 코드를 작성하는데 도움을 줄 것이고, C#/Java에서 TypeScript로 바로 입문한 프로그래머가 겪는 흔한 함정을 피할 수 있을 것입니다.

### JavaScript 함께 배우기 (Co-learning JavaScript)

만약 JavaScript에 이미 익숙하지만 주로 Java또는 C#을 사용하는 프로그래머라면, 이 소개 페이지는 흔히 접할 수 있는 오해와 함정에 대한 설명에 도움을 줄 수 있습니다. TypeScript 모델이 유형화하는 방법 중 일부는 Java나 C#과 상당히 다르며, TypeScript를 학습하는 데에 있어 이 부분을 염두에 두는 것이 중요합니다.

만약 JavaScript를 처음 접하는 Java나 C# 프로그래머라면, JavaScript의 런타임 동작을 이해하기 위해 우선적으로 타입을 *제외한* JavaScript의 일부분을 배우는 것이 좋습니다. TypeScript는 코드를 *실행하는* 방식을 바꾸지 않기 때문에, 실제로 무언가 동작하는 코드를 작성하기 위해서는 여전히 JavaScript가 어떻게 작동하는지 배워야 합니다!

TypeScript가 JavaScript와 동일한 *런타임*을 사용하므로, 특정한 런타임 동작(문자열을 숫자로 변환하기, 경고 표시, 디스크에 파일 쓰기 등)을 구현하려는 리소스는 항상 TypeScript 프로그램에 똑같이 잘 적용된다는 점을 기억하는 것은 매우 중요합니다. TypeScript에 특정된 리소스에만 제한을 두지 마십시오!

### 클래스 다시 생각하기 (Rethinking the Class)

C#과 Java는 *의무적 OOP* 언어라고 부릅니다. 이러한 언어에서 *클래스*는 코드 구성의 기본 단위일 뿐만 아니라 런타임 시 모든 데이터 *그리고* 동작의 기본적인 컨테이너입니다. 기능과 데이터를 전부 클래스에 담도록 강제하는 것은 일부 문제에 대해선 좋은 도메인 모델이 될 수 있지만, 모든 도메인이 이러한 방식으로 표현될 *필요*는 없습니다.

#### 자유로운 함수와 데이터 (Free Functions and Data)

JavaScript에서 함수는 어디에나 있을 수 있고, 데이터를 미리 정의된 'class'나 'struct'에 속하지 않고 자유롭게 전달할 수 있습니다. 이러한 유연성은 매우 강력합니다. OOP 계층과 상관 없이 데이터를 처리하는 "자유로운" (클래스와 연관되지 않은) 함수는 프로그램을 JavaScript로

On this page


JavaScript 함께 배우기 (Co-le...
클래스 다시 생각하기 (Rethinki...
자유로운 함수와 데이터 (Free ...
정적 클래스 (Static Classes)
TypeScript의 OOP (OOP in T...
타입 다시 생각하기 (Rethinking...
이름으로 구체화된 타입 시스템 ...
집합으로서의 타입 (Types as ...
삭제된 구조적 타입 (Erased S...
구조적 타입화의 결과 (Conseq...
반영 (Reflection)


Is this page helpful?

Yes No

https://www.typescriptlang.org/ko/docs/handbook/typescript-in-5-minutes-oop.html

# TypeScript

개발환경 선택

```
# npm일 경우
npm i -D typescript @types/node

# pnpm일 경우
pnpm add -D typescript @types/node

tsc --init
```

```
// {최상단}/tsup.config.mjs
import path from 'node:path';
import { defineConfig } from 'tsup';
import { globSync } from 'tinyglobby';

export default {
  entry: globSync(
    [
      './javascript/**/*.js',
      './javascript/**/*.ts', // TypeScript
    ],
    { cwd: import.meta.dirname }
  ),
  outDir: path.resolve(
    import.meta.dirname,
    './wwwroot/js'
  ),
  format: ['esm'],
  platform: 'browser',
}
```

# TypeScript

개발환경 선택

```
# npm일 경우
npm i -D typescript @types/node

# pnpm일 경우
pnpm add -D typescript @types/node

tsc --init
```

```
// {최상단}/tsup.config.mjs
import path from 'node:path';
import { defineConfig } from 'tsup';
import { globSync } from 'tinyglobby';

export default {
  entry: globSync(
    [
      './javascript/**/*.js',
      './javascript/**/*.ts', // TypeScript
    ],
    { cwd: import.meta.dirname }
  ),
  outDir: path.resolve(
    import.meta.dirname,
    './wwwroot/js'
  ),
  format: ['esm'],
  platform: 'browser',
}
```

# TypeScript (번들러가 없는 경우)

개발환경 선택

Learn / Visual Studio / JavaScript 및 TypeScript /

## TypeScript 코드 컴파일(ASP.NET Core)

아티클 • 2024. 12. 12. • 기여자 8명

피드백

TypeScript NuGet 패키지를 사용하여 ASP.NET Core 프로젝트에 TypeScript 지원을 추가합니다. Visual Studio 2019부터 TypeScript SDK 대신 NuGet 패키지를 사용하는 것이 좋습니다. TypeScript NuGet 패키지는 다양한 플랫폼 및 환경에서 더 큰 이식성을 제공합니다.

ASP.NET Core 프로젝트의 경우 NuGet 패키지의 일반적인 용도 중 하나는 .NET Core CLI를 사용하여 TypeScript를 컴파일하는 것입니다. .NET 시나리오에서 NuGet 패키지는 기본 옵션이며 `dotnet build` 및 `dotnet publish` 같은 .NET Core CLI 명령을 사용하여 TypeScript 컴파일을 사용하도록 설정하는 유일한 방법입니다. 또한 ASP.NET Core 및 TypeScript와 MSBuild 통합 경우 NuGet 패키지를 선택합니다.

**중요**

JSPS(JavaScript Project System)또는 *.esproj* 프로젝트를 기반으로 하는 프로젝트의 경우 NuGet 대신 npm 패키지 사용하여 TypeScript 지원을 추가합니다.

https://learn.microsoft.com/ko-kr/visualstudio/javascript/compile-typescript-code-nuget

# TypeScript 개발환경 선택

## 정적 .NET 메서드 호출

JavaScript(JS)에서 정적 .NET 메서드를 호출하려면 JS 함수를 사용합니다.

- `DotNet.invokeMethodAsync` (*권장*): 서버 쪽 및 클라이언트 쪽 구성 요소 모두에 대한 비동기입니다.
- `DotNet.invokeMethod`: 클라이언트 쪽 구성 요소에 대해서만 동기식입니다.

메서드를 포함하는 어셈블리의 이름, 정적 .NET 메서드 식별자 및 인수를 전달합니다.

다음 예제에서

- `{ASSEMBLY NAME}` 자리 표시자는 앱의 어셈블리 이름입니다.
- `{.NET METHOD ID}` 자리 표시자는 .NET 메서드 식별자입니다.
- `{ARGUMENTS}` 자리 표시자는 메서드에 전달할 선택적 쉼표로 구분된 인수이며, 각 인수는 JSON 직렬화 가능해야 합니다.

JavaScript 복사

```javascript
DotNet.invokeMethodAsync('{ASSEMBLY NAME}', '{.NET METHOD ID}', {ARGUMENTS});
```

# TypeScript 개발환경 선택

```
# npm일 경우
npm i -D @microsoft/dotnet-js-interop

# pnpm일 경우
pnpm i -D @microsoft/dotnet-js-interop
```

# TypeScript 개발환경 선택

{ROOT}/global.d.ts

```ts
import type { DotNet } from '@microsoft/dotnet-js-interop';

declare global {
  interface Window {
    DotNet: typeof DotNet;
  }
}
```

{ROOT}/global.d.ts | tsconfig.json

```json
{
  "compilerOptions": {
    // 생략
    "types": [
      "./global.d.ts"
    ]
  }
}
```

# TypeScript

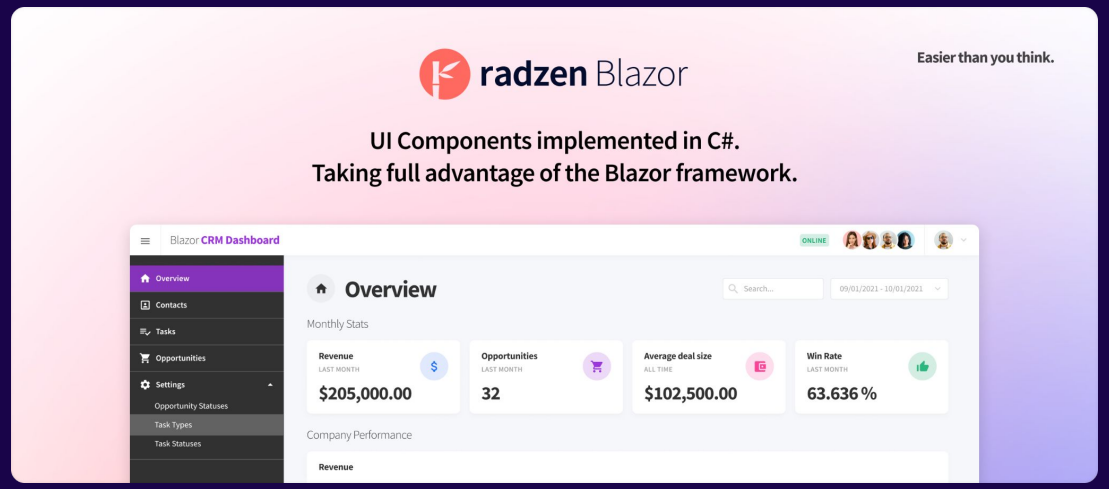
개발환경 선택

```
window.DotNet.        You, 15초 전 • Uncommitted changes
              DotNetObject
              JSCallResultType
              attachDispatcher
              attachReviver
              createJSObjectReference
              createJSStreamReference
              disposeJSObjectReference
              disposeJSObjectReferenceById
              findJSFunction
              invokeMethod
              invokeMethodAsync
```

# 웹 컴포넌트

프로덕션 선택

MudBlazor




Blaz<>rise

# 웹 컴포넌트

프로덕션 선택

## 웹 컴포넌트

웹 컴포넌트는 그 기능을 나머지 코드로부터 캡슐화하여 재사용 가능한 커스텀 엘리먼트를 생성하고 웹 앱에서 활용할 수 있도록 해주는 다양한 기술들의 모음입니다.

## 개념 및 사용

개발자로서 우리 모두는 가능한 한 코드를 재사용하는 것이 좋은 생각이라는 것을 알고 있습니다. 이는 전통적으로 커스텀 마크업 구조에선 쉽지 않았습니다. 커스텀 UI 컨트롤을 렌더링하기위해 작성해야하는 복잡한 HTML (및 관련된 스타일과 스크립트)을 생각해보십시오. 여러번 사용할 때 조심하지 않으면 페이지가 엉망이 될 수 있습니다.

웹 컴포넌트는 다음 문제들을 해결하는 것을 목표로 합니다 — 세 가지 주요 기술들로 구성되며, 재사용을 원하는 어느곳이든 코드 충돌에 대한 걱정이 없는 캡슐화된 기능을 갖춘 다용도의 커스텀 엘리먼트를 생성하기 위해 함께 사용될 수 있습니다.

# 웹 컴포넌트

프로덕션 선택

Spectrum Web Components

Red Hat Design system

Shoelace 2.19.1

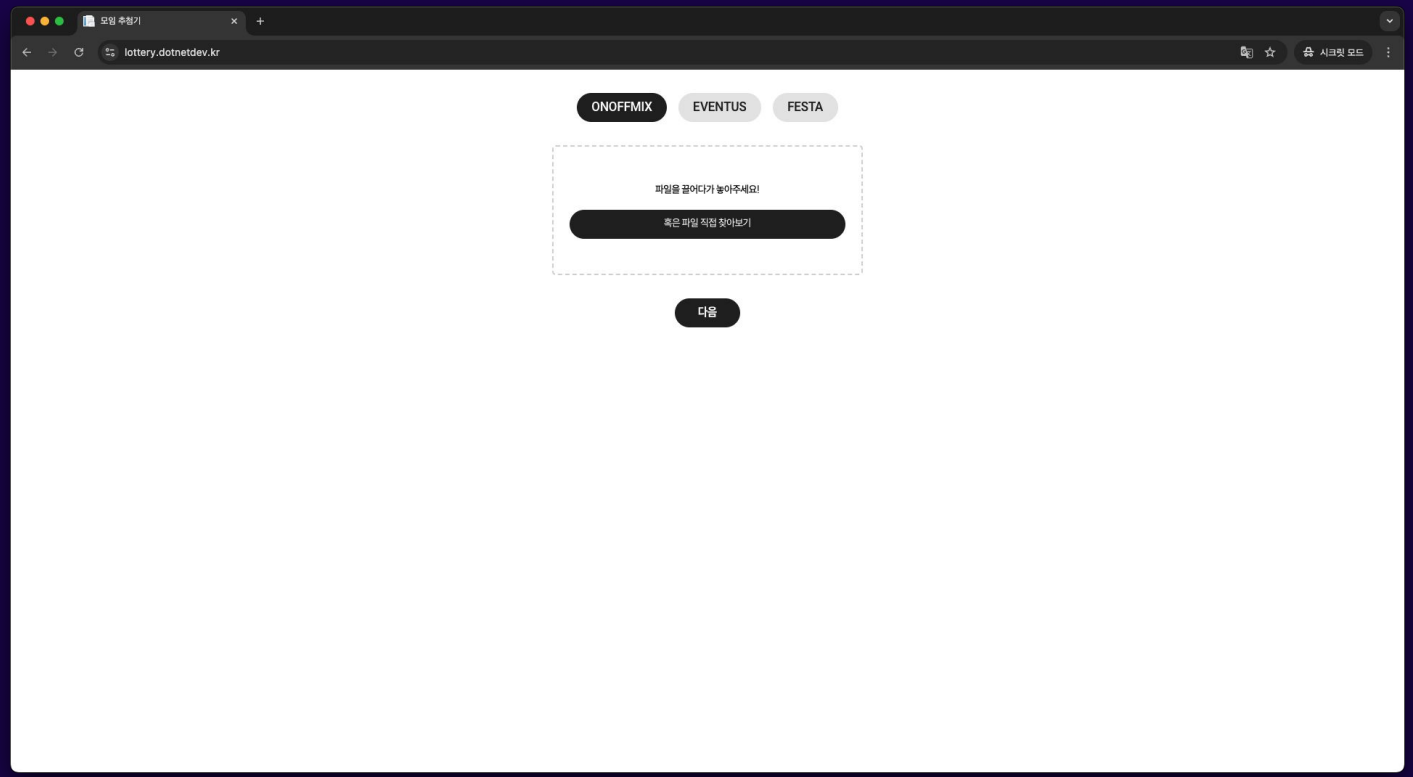
FLEXIBLE, PERFORMANT, & INTUITIVE

FAST

FAST is dedicated to providing support for native Web Components and modern Web Standards, designed to help you efficiently tackle some of the most common challenges in website and application design and development.

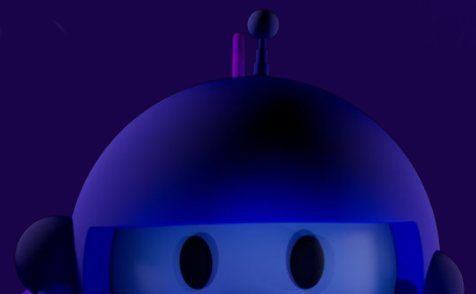
For an in-depth explanation of FAST, see our docs introduction.

# 마지막으로..




DotNetDevLottery

# 감사합니다!

닷넷데브 닷넷데브

in/pedogunu