

# Deno: A Next-generation JavaScript runtime

Yongwook Choi



JSConf Korea 2022

안녕하세요. 차세대 자바스크립트 런타임, Deno라는 주제로 발표하게 된 최용욱입니다.

## 소개

- 최용욱 (Yongwook Choi)
- DX Engineer @ Riiid
- [flow@hrmm.xyz](mailto:flow@hrmm.xyz)
- [@hyp3rflow \(GitHub\)](https://github.com/hyp3rflow)
- [@hrmm\\_flow \(twitter\)](https://twitter.com/hrmm_flow)



저는 Riiid에서 개발자 경험 엔지니어로 일하고 있고, pbkit이나 WRP 같은 사내에서 사용하는 여러 개발 툴링들을 Deno로 만들고 있습니다.  
또 여기 앞 슬라이드에 있던 친구가 Deno 캐릭터인데요, Deno가 귀여워서 Deno 생태계에 조금씩이나마 기여를 하고 있습니다.

## 목차

- 새로운 프로젝트 만들어보기
- Deno의 향상된 개발자 경험
- Deno - Node.js 호환성
  - Deno에서 Node.js 코드 실행하기 (Node.js → Deno)
  - Deno 코드를 Node.js에서 실행하기 (Deno → Node.js)
- 어떻게 사용하고 있나요 - pbkit 개발
- 그래서 프로덕션에서 쓸 수 있을까요?

처음으로는 함께 Node.js와 TypeScript를 사용하는 프로젝트를 만드는 상황을 살펴보면서, 어떤 문제점들이 있는지 알아보고 그 문제점들을 Deno가 어떻게 해결하는지 알려드리고자 합니다.

다음으로는 Deno의 향상된 개발자 경험에 대해 설명합니다. 저는 사실 비교할 수 없는 개발자 경험이라고 적고 싶었는데요, 그 정도로 Node.js에 비해서 Deno가 갖는 매우 큰 장점 중 하나입니다.

또 오늘은 Deno에 대해 설명드리지만, Node.js 생태계가 정말 크고 Node.js로 실행되는 레거시들도 엄청 많기 때문에 이 생태계를 잘 이용할 수 있어야 하겠죠. 그래서 Deno를 사용하면서도 어떻게 Node.js와의 호환성을 챙길 수 있는지에 대해 알아보도록 하겠습니다.

마지막으로는 저희 팀에서 Deno를 사용한 적용기를 조금 공유드리고자 합니다.

## 새로운 프로젝트 만들어보기

Node.js + TypeScript

Package manager



Linter / Formatter



ESLint



Testing Framework



그럼, 한번 TypeScript로 Node.js 프로젝트를 만드는 상황을 고려해봅시다.

일단 JavaScript 생태계의 많은 라이브러리를 사용하기 위해서 패키지 매니저가 필요할 것입니다.

npm, yarn, yarn berry, pnpm과 같은 패키지 매니저들 중에서 하나를 선택해야겠구요.

다음으로는 코드 퀄리티와 스타일을 맞추기 위해서 여러 툴링들을 사용하려고 합니다.

eslint, prettier와 같은 linter와 formatter를 추가하고, 각각 원하는 플러그인이나 컨피, 혹은 룰들을 추가하여 원하는 세팅을 맞췄습니다.

여기서 코드 퀄리티를 더 높이려면 테스팅 라이브러리도 추가해야겠죠.

jest나 mocha, 요즘에 나온 vitest와 같은 테스팅 라이브러리를 골랐습니다.

## 새로운 프로젝트 만들어보기

Node.js + TypeScript

**tsc / babel**

**ts-node**

**Transpiler → Node.js**

**sucrase**

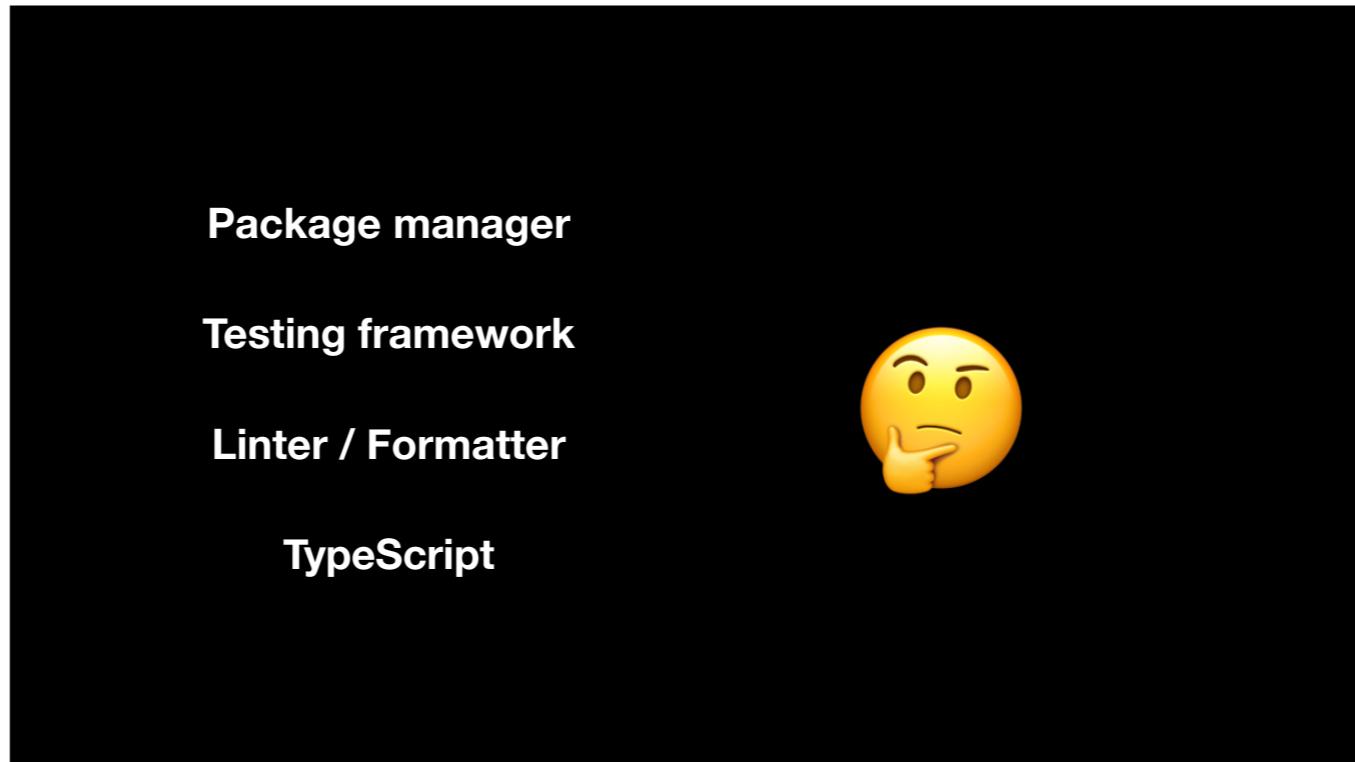
**swc**

여기서 끝이 아니죠.

TypeScript 코드는 Node.js에서 바로 실행할 수 없기 때문에 transpiler 설정들이 필요합니다.

TypeScript compiler를 이용하여 JavaScript 코드로 변환한 뒤에 이것을 Node.js에게 전달할 수도 있고,  
ts-node나 sucrase, swc 같은 transpiler를 TypeScript compiler처럼 사용하거나

register hook을 통해 Node.js가 TypeScript 모듈을 로드할 때 미리 이것을 JavaScript로 트랜스파일하여 넘겨줌으로써 TypeScript를 Node.js에서 돌릴 수 있게 됩니다.

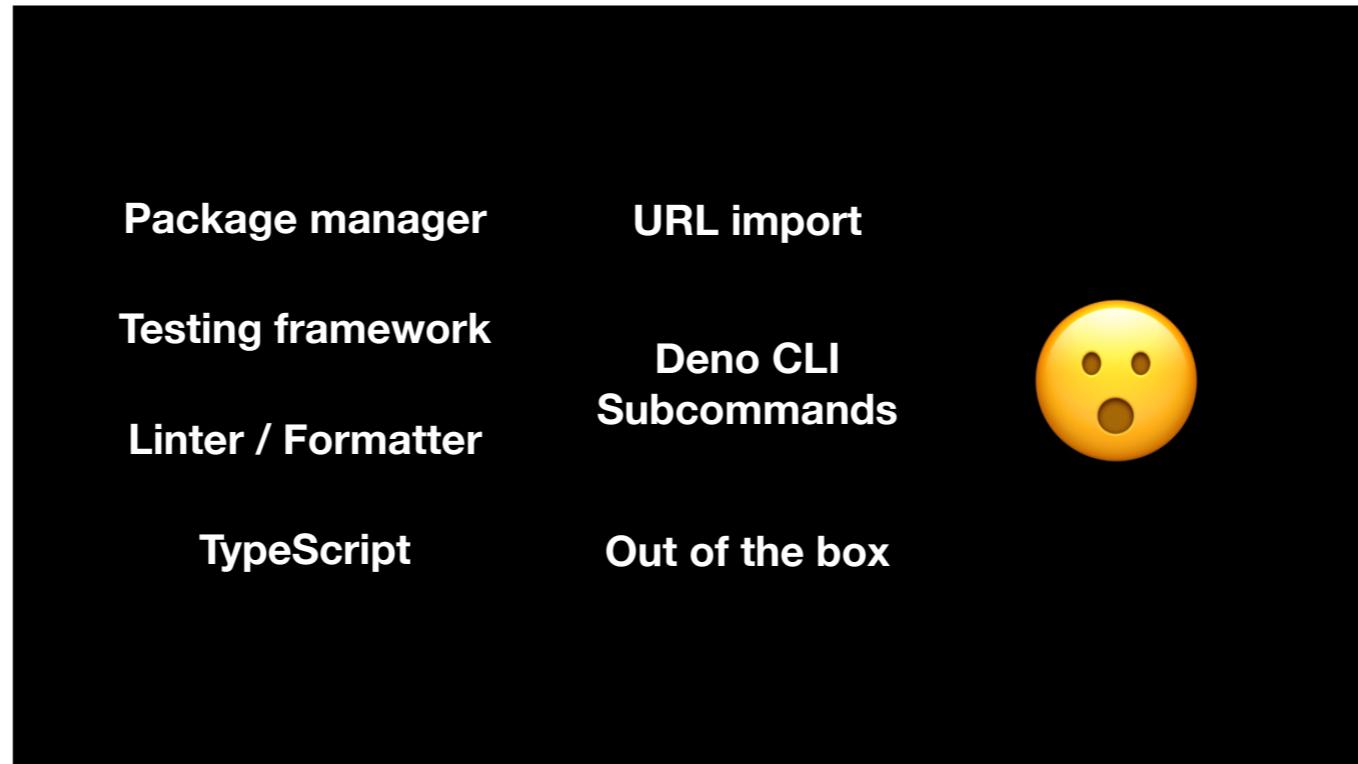


음.. 간단한 Node.js 프로젝트를 만드는데 고려해야 할 것들이 좀 많은 것 같아 보입니다.

더 빠르고 간단하게 프로젝트를 시작할 수는 없을까요? 이런 작업들을 하지 않고도 바로 모던한 스택의 코드를 작성할 수 있었으면 좋을 것 같은데 말이죠.



그래서 이런 문제를 해결해 줄 새로운 JavaScript 런타임, Deno를 소개해드리고자 합니다.



아까 있던 문제들은 Deno을 사용하면 쉽게 해결할 수 있습니다.

Deno는 패키지 매니저를 사용하지 않기 때문에, 패키지 매니저를 고를 필요도 없습니다. 외부 모듈들은 URL을 통해서 import하게 됩니다.

포매터와 린터, 테스트들도 별도의 설치 없이 Deno의 서브커맨드로 사용할 수 있습니다.

물론 TypeScript도 별도의 설정 없이 사용할 수 있습니다.

## 향상된 개발자 경험

- 풍부하고 잘 선별된 표준 라이브러리 사용 가능
- Promise 기반의 비동기 API 제공
- Web Platform API를 준수하고자 노력
- 보안을 위한 권한 관리
- ESM module 지원

이러한 장점들도 있지만, 이번에는 Deno의 향상된 개발자 경험에 대해 설명드리겠습니다.  
개발자 경험의 관점에서 제가 생각하기에 크다고 느끼는 몇 가지 장점들을 나열해보는데요.  
하나씩 소개를 드리도록 하겠습니다.

## 향상된 개발자 경험

풍부하고 잘 선별된 표준 라이브러리(Standard Library) 사용 가능

- Go에서 제공하는 표준 라이브러리 모듈들을 레버리지하여 작성
- async, bytes, io, streams, http 등 다양한 모듈들을 제공
  - async: debounce, deferred, delay ...
  - bytes: Uint8Array에 대한 도우미 함수 제공 (concat, equals, copy ...)
  - io: Buffer, BufReader, BufWriter
  - streams: Streams API에 대한 Buffer, conversion 제공 (copy, read, write ...)
  - http: HttpServer, Errors, Status, Cookie ...

Deno에는 다양한 상황에 사용할 수 있는 풍부하고 잘 선별된 표준 라이브러리가 있습니다.

대표적으로 async, bytes, io, streams, http 등 다양한 모듈이 있는데요, 하나씩 간단하게 소개를 드리면

async는 Promise를 가지고 만들 수 있는 다양한 유틸리티 함수를 포함하고 있습니다.

Debounce, deferred, delay 같은 자주 사용하는 비동기 함수를 더 이상 npm install을 통해 설치하거나 구현할 필요가 없습니다.

bytes는 Uint8Array를 다룰 수 있는 함수들을 제공하구요.

io는 Deno의 Reader와 Writer 인터페이스에 사용할 수 있는 Buffer를 제공합니다.

streams는 Web Streams API에 대한 버퍼와 유틸 함수를 제공합니다.

http는 HttpServer를 쉽게 만들 수 있도록 하고, HTTP 에러나 상태 코드, 또 쿠키와 관련된 함수도 제공합니다.

## 향상된 개발자 경험

풍부하고 잘 선별된 표준 라이브러리(Standard Library) 사용 가능

- 표준 라이브러리는 Deno API와 다른 버전을 사용할 수 있음



**Node.js v14**  
= API v14



**Deno v1.25**  
!= std v0.153

Node.js에서는 이러한 표준 라이브러리에 대응되는 빌트인 모듈이 있는데요, 이런 빌트인 모듈은 별도의 버저닝 없이 Node.js 런타임의 버저닝과 강결합되어 있어 버전을 쉽게 올리지 못하는 단점이 있습니다.

예를 들어 Node 12를 맞춰야 하고 16을 맞춰야 하고.. 이런 Node.js 버전을 쉽게 바꾸지 못하는 상황이 있는데, 그런 런타임 버전에 따라 빌트인 모듈의 버전도 고정되니 새로운 기능을 사용하기 힘듭니다.

## 향상된 개발자 경험

풍부하고 잘 선별된 표준 라이브러리(Standard Library) 사용 가능

- 표준 라이브러리는 Deno API와 다른 버전을 사용할 수 있음

```
import * from "https://deno.land/std@0.145.0/io/mod.ts";
```

↓ Updated!

```
import * from "https://deno.land/std@0.156.0/io/mod.ts";
```

Deno의 표준 라이브러리는 런타임과 버저닝을 따로 가져가고 있어 이런 상황에서 유연하게 런타임의 버전은 고정하면서도 표준 라이브러리의 버전을 조정할 수 있습니다. 다른 서드파티 라이브러리와 동일하게 배포되고, 동일한 방법으로 import 하기 때문에 불러오는 라이브러리의 버전을 올리는 것처럼 쉽게 업데이트할 수 있습니다.

## 향상된 개발자 경험

### Promise 기반의 비동기 API 제공

- Node.js → 비동기 API들이 callback으로 구현
- Promise 형태로 만들기 위해서는?
  - util에 있는 promisify를 이용
  - fs/promises 같은 promise 형식으로 제공되는 API 사용
- Deno → 모든 비동기 API들이 Promise를 반환

Deno는 Promise 기반의 비동기 API를 제공합니다.

Node.js에서 제공되는 여러 비동기 API들은 Promise가 등장하기 전에 작성되어 메인 로직은 callback 방식을 사용합니다.

그래서 Promise 형태로 사용하기 위해 util이라는 빌트인 모듈에 있는 promisify 함수를 사용하거나 promise를 반환하도록 재작공된 API를 사용할 수 밖에 없습니다.

Deno는 모든 비동기 API들이 Promise 기반으로 작성되었고, 따라서 callback 스타일의 API를 제공하지 않습니다.

Node.js lib/fs.js readFile	Deno runtime/40_read_file.js readFile
<pre> function readFile(path, options, callback) {   // ...   const context = new ReadfileContext(callback, options.encoding);   context.isUserFd = isFd(path); // File descriptor ownership    if (options.signal) {     context.signal = options.signal;   }   if (context.isUserFd) {     process.nextTick(function tick(context) {       ReflectApply(readFileAfterOpen, { context }, [null, path]);     }, context);     return;   }    if (checkAborted(options.signal, callback))     return;    const flagsNumber = stringToFlags(options.flag, 'options.flag');   path = getValidatedPath(path);    const req = new F5ReqCallback();   req.context = context;   req.oncomplete = readFileAfterOpen;   binding.open(pathModule.toNamespacedPath(path),                flagsNumber,                0666,                req); } </pre>	<pre> async function readFile(path, options) {   let cancelRid;   let abortHandler;   if (options?.signal) {     options.signal.throwIfAborted();     cancelRid = ops.op_cancel_handle();     abortHandler = () =&gt; core.tryClose(cancelRid);     options.signal[abortSignal.add](abortHandler);   }    try {     const read = await core.opAsync(       "op_readfile_async",       pathFromURL(path),       cancelRid,     );     return read;   } finally {     if (options?.signal) {       options.signal[abortSignal.remove](abortHandler);        // always throw the abort error when aborted       options.signal.throwIfAborted();     }   } } </pre>

Node.js와 Deno의 readFile 함수 구현을 각각 들고왔는데요.

Node.js 쪽은 callback을 받아서 Context를 만들고, 이것을 내부 바인딩에 넘기는 식으로 콜백을 실행하고 있습니다.

이에 반해 Deno는 async function으로 readFile을 구성하고, await을 이용해 비동기 내부 API를 부르는 식으로 Promise 기반 로직을 구성하는 것을 알 수 있습니다.

**Node.js**  
**lib/internal/fs/promises.js readFile**

```

async function readFile(path, options) {
  ...options = getOptions(options, { flag: 'r' });
  ...const flag = options.flag || 'r';

  if (path instanceof FileHandle)
    ...return readFileHandle(path, options); → (Yellow arrow)

  ...checkAborted(options.signal);

  ...const fd = await open(path, flag, 0o666);
  ...return handleFdClose(readFileHandle(fd, options), fd.close());
}

async function readFileHandle(filehandle, options) {
  ...const signal = options.signal;
  ...checkAborted(signal);

  ...const statFields = await binding.fstat(filehandle.fd, false, kUsePromises);
  ...checkAborted(signal);

  let size;
  if ((statFields[1 & node_v] & S_IFMT) === S_IFREG) {
    size = statFields[8 & size];
  } else {
    size = 0;
  }

  if (size > kIoMaxlength)
    throw new ERR_FS_FILE_TOO_LARGE(size);

  let endOfFile = false;
  let totalRead = 0;
  const noSize = size === 0;
  const buffers = [];
  const fullBuffer = noSize ? undefined : Buffer.allocUnsafeSlow(size);
  do {
    ...checkAborted(signal);
    let buffer;
    let offset;
    let length;
    if (noSize) {
      buffer = fullBuffer;
      offset = 0;
      length = kReadFileUnknownBufferLength;
    } else {
      buffer = fullBuffer;
      offset = totalRead;
      length = MathMin(size - totalRead, kReadFileBufferLength);
    }

    const bytesRead = (await binding.read(filehandle.fd, buffer, offset,
      length, -1, kUsePromises)) || 0;
    totalRead += bytesRead;
    endOfFile = bytesRead === 0 || totalRead === size;
    if (noSize && bytesRead > 0) {
      const isBufferFull = bytesRead === kReadFileUnknownBufferLength;
      const chunkBuffer = isBufferFull
        ? ArrayPrototypePush(buffers, c => {
          async function handleFdClose(fileOpPromise, closeFunc) {
            return PromisePrototypeThen(
              fileOpPromise,
              (result) => PromisePrototypeThen(closeFunc(), () => result),
              (opError) => PromisePrototypeThen(
                closeFunc(),
                () => PromiseReject(opError),
                (closeError) => PromiseReject(aggregateTwoErrors(closeError, opError))
              );
            return options.encoding ? result : '';
          }
        })
        : buffers;
      if (isBufferFull) {
        result = totalRead === size ? buffers : chunkBuffer;
      } else {
        result = bytesRead === 1 ? buffers : chunkBuffer;
      }
    }
  } while (!endOfFile);
  ...return PromisePrototypeThen(
    fileOpPromise,
    (result) => PromisePrototypeThen(closeFunc(), () => result),
    (opError) => PromisePrototypeThen(
      closeFunc(),
      () => PromiseReject(opError),
      (closeError) => PromiseReject(aggregateTwoErrors(closeError, opError))
    );
  );
}

```



이렇게 구현 로직만 보더라도 Deno 코드가 더 깔끔한데,

이 fs의 readFile 함수를 Promise 형태로 감싸기 위해서 더 많은 양의 코드가 Node.js에서는 들어갑니다.

Deno에서는 이미 앞의 구현으로 Promise 형태를 반환하는 API를 만드는 것이 끝났습니다.

또, 그럴 일이 없다면 좋겠지만 만약 내부 구현을 뜯어볼 일이 있다면 두 런타임 중 Deno가 내부구현을 보기에는 더 편할 것입니다.



Deno의 표준 라이브러리는 더욱 깔끔한 내부 구현을 가지고 있습니다.

앞서 설명드린대로 Deno의 표준 라이브러리가 다른 서드파티 라이브러리와 제공되는 방식에서 크게 다른 점이 없기 때문에 당연하긴 하지만, Deno 코드를 작성할 때 주로 사용할 표준 라이브러리의 코드가 깔끔하다는 점은 디버깅이나 내부 동작을 이해하는데에 매우 큰 도움이 됩니다.

## 향상된 개발자 경험

### Web Platform API를 준수하고자 노력

- Deno에서는 비슷한 Web API가 있는 경우 바퀴의 재발명을 지양
- Fetch, Console, Streams, Web Storage, Web Worker 등 Web API 구현
  - 브라우저에서 사용하던 API를 Deno에서도 사용 가능

	Chrome	Edge	Firefox	Opera	Safari	Chrome Android	Firefox for Android	Opera Android	Safari on iOS	Samsung Internet	WebView Android	Deno!
localStorage	v 4	v 12	v 3.5	v 10.5	v 4	v 18	v 4	v 11	v 3.2	v 1.0	v 37	v 1.16

다음으로 Deno는 Web Platform API를 준수하고자 노력합니다.

Web Platform API는 크롬이나 파이어폭스 같은 브라우저에 내장되어 있는 Web API들을 의미합니다.

브라우저 외부의 런타임에서 다양한 기능을 구현할 때 이미 브라우저 내에 동일한 기능이 탑재되어 있는 경우가 많습니다.

이런 경우 Deno는 바퀴의 재발명을 하지 않고, 브라우저와 호환되는 Web API를 구현함으로써 브라우저와의 상호호환성을 챙깁니다.

Fetch, console, streams, local storage, web worker 같은 Web API들이 Deno에 이미 구현이 되어 있습니다. Node.js에서도 Web API가 일부 구현되어 있지만 대부분의 기능들은 실험적인 상태여서 일부만 호환되거나 외부 라이브러리를 사용해야만 하는 상태입니다. 또 구현되어 있더라도 내부 모듈을 불러와야 사용할 수 있습니다.

Deno에서는 브라우저에서 사용하던 이런 API들을 바로 사용할 수 있어 브라우저 API 코드들을 주로 사용하시던 분들도 Deno 위에서는 쉽게 브라우저 밖 런타임에서 실행되는 코드를 작성할 수 있습니다.

이렇게 MDN에서 localStorage 같은 Web API를 검색해보면, 제일 아래에 있는 호환성 테이블에 Deno도 있는 것을 볼 수 있습니다.

## 향상된 개발자 경험

### 보안을 위한 권한 관리, ESM 지원

- 파일 시스템, 네트워크, 환경 변수 접근 등이 필요할 때 추가적인 권한 필요
  - Node.js → 권한 설정이 불가능하여 악의적인 코드가 마음대로 접근 가능
- CommonJS를 지원하지 않고 오직 ES Modules만을 지원
  - 오래되고 다양한 모듈 방식들은 코드/툴링의 난이도를 복잡하게 만듦

다음으로 저희가 사용하는 어떤 NPM 패키지가 다양한 이유로 악의적인 코드가 될 수 있는데, 이런 경우 Node.js에서는 악의적인 코드가 시스템을 조작하는 것에 대해서 대응하기 힘든 구조를 갖고 있습니다.

Deno에서는 파일 시스템이나 네트워크, 환경 변수 접근 등이 필요할 때 적합한 권한이 없는 경우 실행을 막을 수 있기에 이런 문제를 피할 수 있습니다.

또 제가 가장 좋아하는 것 중 하나는 모듈 시스템으로 오직 ES Modules만을 지원한다는 점입니다.

CommonJS와 AMD, ESM 같은 여러 모듈 시스템들이 처음 JavaScript를 배우는 데에 큰 진입장벽이 된다고 생각하는데,

Deno에서는 ESM만 지원하기 때문에 이러한 예전 모듈 시스템들을 생각하지 않고 코드를 작성하는 것이 가능합니다.

뿐만 아니라 이전의 표준 라이브러리도 ESM으로 작성되어 있어 자바스크립트의 숙련도와 관계없이 코드를 읽기 쉽다는 것도 장점입니다.

# Node.js 호환성

[Node.js 코드를 Deno에서 실행하기](#)

[Deno 코드를 Node.js에서 실행하기](#)

지금까지 Deno가 어떤 장점들을 가지고 있고, Node.js와 비교했을 때 왜 좋은지 설명드렸습니다.

그러나, 아무리 좋다고 하더라도 방대한 Node.js의 생태계를 이용할 수 있다면 더욱 좋겠죠?

이번 섹션에서는 서로서로의 코드를 반대 런타임에서 실행할 수 있는 방법을 소개합니다.

# Node.js 호환성

## 1. std/node: Deno에서 Node.js API 사용하기

- 표준 라이브러리의 std/node 모듈을 사용 가능
- 모든 기능이 지원되지는 않으나, 자주 사용하는 라이브러리 대응을 목표로 호환을 넓혀가고 있음
- <https://deno.land/std/node>

<input checked="" type="checkbox"/> assert	<input checked="" type="checkbox"/> path/posix
<input checked="" type="checkbox"/> assert/strict <i>partly</i>	<input checked="" type="checkbox"/> path/win32
<input checked="" type="checkbox"/> async_hooks <i>partly</i>	<input checked="" type="checkbox"/> perf_hooks
<input checked="" type="checkbox"/> buffer	<input checked="" type="checkbox"/> process <i>partly</i>
<input checked="" type="checkbox"/> child_process <i>partly</i>	<input checked="" type="checkbox"/> punycode
<input checked="" type="checkbox"/> cluster <i>partly</i>	<input checked="" type="checkbox"/> querystring
<input checked="" type="checkbox"/> console <i>partly</i>	<input checked="" type="checkbox"/> readline
<input checked="" type="checkbox"/> constants <i>partly</i>	<input checked="" type="checkbox"/> repl <i>partly</i>
<input checked="" type="checkbox"/> crypto <i>partly</i>	<input checked="" type="checkbox"/> stream
<input checked="" type="checkbox"/> dgram <i>partly</i>	<input checked="" type="checkbox"/> stream/promises
<input checked="" type="checkbox"/> diagnostics_channel	<input checked="" type="checkbox"/> stream/web <i>partly</i>
<input checked="" type="checkbox"/> dns <i>partly</i>	<input checked="" type="checkbox"/> string_decoder
<input checked="" type="checkbox"/> events	<input checked="" type="checkbox"/> sys
<input checked="" type="checkbox"/> fs <i>partly</i>	<input checked="" type="checkbox"/> timers
<input checked="" type="checkbox"/> fs/promises <i>partly</i>	<input checked="" type="checkbox"/> timers/promises
<input checked="" type="checkbox"/> http <i>partly</i>	<input checked="" type="checkbox"/> tls
<input checked="" type="checkbox"/> http2	<input checked="" type="checkbox"/> trace_events
<input checked="" type="checkbox"/> https <i>partly</i>	<input checked="" type="checkbox"/> tty <i>partly</i>
<input checked="" type="checkbox"/> inspector <i>partly</i>	<input checked="" type="checkbox"/> url
<input checked="" type="checkbox"/> module	<input checked="" type="checkbox"/> util <i>partly</i>
<input checked="" type="checkbox"/> net	<input checked="" type="checkbox"/> util/types <i>partly</i>
<input checked="" type="checkbox"/> os <i>partly</i>	<input checked="" type="checkbox"/> v8
<input checked="" type="checkbox"/> path	<input checked="" type="checkbox"/> vm <i>partly</i>
	<input checked="" type="checkbox"/> wasi
	<input checked="" type="checkbox"/> webcrypto
	<input checked="" type="checkbox"/> worker_threads
	<input checked="" type="checkbox"/> zlib

Node.js 코드를 Deno에서 실행하는 방법은 여러가지가 있습니다.

처음으로 작은 코드 조각을 실행하는 방법부터 시작해보면,

기존에 Node.js에서 실행되는 작은 코드 조각들은 Deno 표준 라이브러리 내에 있는 node 호환 모듈로 교체하는 식으로 호환성을 챙길 수 있습니다.

모든 기능이 지원되지는 않으나, 오른쪽에 보이는 것처럼 대부분의 모듈들이 호환되고 있고, 유명한 라이브러리들을 위주로 대응함으로써 호환을 넓혀가고 있습니다.

## Node.js 호환성

### 2. Node compatibility mode: Node.js 코드를 Deno로 실행시키기

- Node compatibility mode를 이용해 Deno 위에서 실행 가능
  - CommonJS 코드 실행 가능 (\_dirname, \_filename도 처리)
  - Node.js global(Buffer, process, ..) 사용 가능
  - Node.js API를 std/node polyfill로 변경
  - deno run --compat 플래그로 사용 가능

```
● ● ●  
const payload = "filename: " + __filename + "\n";  
process.stdout.write(new TextEncoder().encode(payload));
```



```
● ● ●  
Object.defineProperty(globalThis, "process", {  
  value: processModule,  
  enumerable: false,  
  writable: true,  
  configurable: true,  
});  
  
(function (require, __filename, ...) {  
  (function (require, __filename, ...) {  
    const payload = "filename: " + __filename + "\n";  
    process.stdout.write(new TextEncoder().encode(payload));  
  }).call(this, require, __filename, ...);  
}) // -> call this function with computed values
```

작은 코드 조각 뿐만 아니라 Node.js를 위해 작성된 코드 전체도 쉽게 돌릴 수 있습니다. Deno에는 Node compatibility 모드가 있어서, 다양한 전역 객체들과 CommonJS를 사용할 수 있게 해줍니다.

또, 따로 Node.js API들을 Deno 표준 라이브러리의 node 호환 모듈로 변경할 필요 없이 polyfill을 채워줍니다.

Deno run의 compat 플래그를 사용해서 전체적인 실행 모드를 호환성 모드로 변경하여 사용할 수 있습니다.

CommonJS 코드는 아래와 같이 주위에 wrapper를 만들어서 전체 코드를 평가하는 식으로, 내부적으로 진행됩니다. 많은 것들이 생략되어 있지만 filename과 같은 특수한 변수나 require 같은 함수를 채워주는 것을 볼 수 있습니다. 예시에는 process 같은 전역 모듈도 제공해주는 것을 볼 수 있습니다.

## Node.js 호환성

### 3. Deno-friendly CDN: NPM 모듈을 Deno에서 import하기

- Deno 친화적인 CDN에서 Deno와 호환되는 NPM 라이브러리를 제공
  - esm.sh / skypack
  - CJS → ESM 변환
  - TypeScript를 위한 typing 제공 (X-Typescript-Types header)
  - Compat mode처럼 std/node polyfill도 주입
  - import from "<https://esm.sh/react>"

때로는 큰 NPM 모듈을 Deno 코드에서 import해서 사용하고 싶은 경우가 있을텐데요, 이런 경우에는 Deno 친화적인 CDN들을 이용하면 번거로운 작업 없이도 Node.js 모듈을 쉽게 사용할 수 있습니다.

Deno 친화적인 CDN은 esm.sh나 skypack 등이 있는데, 이러한 CDN에서 모듈을 받는 경우, Node compatibility mode가 해주던 것들을 자동으로 처리된 상태의 모듈을 사용할 수 있게 됩니다.

CommonJS 기반의 코드를 ESM으로 자동으로 바꿔주며, TypeScript를 위한 typing도 제공하고, 각종 Node.js 빌트인 모듈을 std/node 폴리필로 대체하여 주입해줍니다. 예를 들어 다음과 같이 import하면, 자동으로 Deno에서도 사용가능하게 바꿔진 모듈을 받게 됩니다.

## Node.js 호환성

### 4. NPM Specifier: 더 편하게 NPM 모듈 사용하기

- 이전의 Node Compatibility mode가 삭제되고 새롭게 추가된 호환성 방법
- import \* from "npm:express@4"
- deno run으로도 npm에 있는 Node.js 코드 실행 가능

```
~/.deno (0.426s)
deno run --unstable -A npm:cowsay hello
< hello >
-----
 \  ^__^
  (oo)\----)
  (__)\       )\/\
    ||----w |
    ||     ||
```

앞서 Node compatibility mode와 Deno 친화적인 CDN을 사용해서 NPM 모듈 호환성을 챙기는 법을 소개해드렸는데요.

이번 Deno 1.25.2부터는 Node compatibility mode가 삭제되고 NPM specifier를 사용하도록 변경되었습니다.

NPM specifier를 사용하면 더 이상 Node compat 모드처럼 특수한 플래그를 이용해 전체적인 실행 환경을 바꿀 필요도 없고,

Deno 친화적인 CDN에 의존하지 않아도 NPM 모듈을 사용할 수 있습니다.

사용하는 방법도 간단합니다. Deno 친화적인 CDN이 import 경로 앞에 주소를 달아야했던 것처럼, NPM prefix를 붙여주기만 하면 됩니다.

Deno run 명령을 통해서도 NPM 모듈을 실행할 수도 있습니다.

## Node.js 호환성

Future: Implement Node API #13633

- 성능 상 혹은 이식성의 이점을 위해 Node.js Addon을 사용
- 저수준의 Addon 코드에서 JavaScript 값을 조작할 수 있는 Node API 제공
- NPM에 있는 sqlite나 이미지 처리 라이브러리 sharp

지금까지 알아본 호환성 내용은 코드 레벨에서의 호환성과 관련된 것이였습니다.

Node.js 생태계에는 Node.js의 API를 사용하는 코드들도 있지만, Node.js 애드온도 존재합니다.

저수준 언어로 작성된 애드온들은 Node API를 이용해서 JavaScript 값을 조작하기 때문에, Node API에 대한 호환을 제공해야 다른 런타임에서도 애드온을 불러와서 실행할 수 있습니다.

이러한 Node API에 대한 호환성을 챙기는 PR이 현재 진행중이여서, 이 PR이 적용된다면 NPM에 있는 애드온을 사용하는 sqlite나 이미지 처리 라이브러리인 sharp 같은 친구들도 Deno 위에서 사용할 수 있게 될 예정입니다.

## Node.js 호환성

### 1. Deno에서 작성된 코드를 Node.js에서 쓰고 싶은 경우

- dnt (Deno - Node transform) - <https://github.com/denoland/dnt>
- Deno 코드를 Node.js 코드로 변환해주는 Deno 라이브러리
- dnt를 통해 NPM 패키지를 만든 후 Node.js에서 사용 가능

지금까지는 Node.js 코드를 Deno에서 실행하는 법이였는데요, 반대로 Deno에서 작성된 코드를 Node.js에서 실행할 수 있게 하는 법도 있습니다.

dnt라고 불리는 Deno to node 변환 라이브러리가 있어서, dnt를 이용해서 작은 변환 코드만 작성해주면 손쉽게 Deno 코드를 Node.js 코드로 바꿀 수 있습니다.

```
// ex. scripts/build_npm.ts
import { build, emptyDir } from "https://deno.land/x/dnt/mod.ts";

await emptyDir("./npm"); ←

await build({
  entryPoints: ["./mod.ts"], ←
  outDir: "./npm",
  shims: {
    // see JS docs for overview and more options
    deno: true,
  },
  package: { ←
    // package.json properties
    name: "your-package",
    version: Deno.args[0],
    description: "Your package.",
    license: "MIT",
    repository: {
      type: "git",
      url: "git+https://github.com/username/repo.git",
    },
    bugs: {
      url: "https://github.com/username/repo/issues",
    },
  },
});

// post build steps
Deno.copyFileSync("LICENSE", "npm/LICENSE");
Deno.copyFileSync("README.md", "npm/README.md");
```

## Node.js 호환성

### 1. Deno에서 작성된 코드를 Node.js에서 쓰고 싶은 경우

- 모듈 식별자를 재작성
- Deno namespace의 shim 주입
- skypack/esm.sh → package.json
- NPM package 지정 가능

dnt는 import 식별자를 재작성하고, Deno 네임스페이스에 있는 여러 자원들을 Node.js에서도 실행될 수 있게 shim을 주입해줍니다.

왼쪽에 있는 코드가 dnt 코드인데요,

현재 디렉토리에 npm이라는 디렉토리를 만들고, 이 디렉토리를 타겟으로 패키지를 구성하는 build 스크립트를 작성합니다.

엔트리포인트를 기준으로 모듈들을 resolve하면서 node.js 코드로 변경해주고, 아래처럼 package.json에 들어갈 내용을 채워넣을 수 있습니다.

또한 특정 의존성을 특정 npm 모듈로 지정하는 기능도 가능하고, 만약 원래 코드에 node용 코드를 이미 만든 경우 deno 코드 대신 node 코드를 사용하게 만들 수도 있습니다.

## Node.js 호환성

### 2. Deno에서 작성된 코드를 Node.js에서 쓰고싶은 경우

- Deno API에 대한 의존성이 없다면 dnt를 사용하지 않아도 됨
- esbuild + esbuild\_deno\_loader
  - Deno의 import 의존성을 해결할 수 있도록 해주는 플러그인

만약 Deno API에 대한 의존성이 없다면, 즉 Deno 표준 라이브러리나 외부 라이브러리는 사용하는데, 그 라이브러리들도 Deno API는 사용하지 않는 경우에 대해서는 굳이 dnt를 사용하지 않아도 됩니다. esbuild와 esbuild\_deno\_loader라는 플러그인을 이용해서 쉽게 Deno 코드를 esbuild를 이용해 번들링할 수 있습니다.

## Node.js 호환성

### 2. Deno에서 작성된 코드를 Node.js에서 쓰고싶은 경우

- [github.com/esbuild/deno-esbuild](https://github.com/esbuild/deno-esbuild)
- [github.com/lucacasonato/esbuild\\_deno\\_loader](https://github.com/lucacasonato/esbuild_deno_loader)
- 간단하게 esbuild를 이용해 번들링하는 Deno 코드 완성!

```
import * as esbuild from "https://deno.land/x/esbuild@v0.14.51/mod.js";
import { denoPlugin } from "https://deno.land/x/esbuild_deno_loader@0.5.2/mod.ts";

await esbuild.build({
  ...plugins: [denoPlugin()],
  ...entryPoints: ["https://deno.land/std@0.150.0/hash/sha1.ts"],
  ...outfile: "./dist/sha1.esm.js",
  ...bundle: true,
  ...format: "esm",
});
esbuild.stop();
```

esbuild는 별도의 Deno Wrapper도 있어서, 이렇게 Deno로 esbuild를 이용해 번들링하는 코드를 dnt처럼 쉽게 작성할 수 있습니다.

아래의 예제는 Deno의 표준 라이브러리 중 SHA1 코드를 esbuild를 통해 번들링하는 코드입니다.

# 어떻게 사용하고 있나요

pbkit 라이브러리 개발

pbkit Core

pbkit Codegen

Protobuf  
Language  
Server

마지막으로 이러한 Node 호환성을 이용해서 Node.js 생태계 위에서도 동작하는 개발 툴링을 Deno로 만들었던 경험을 공유해드리고자 합니다.

Pbkit은 Protocol buffers와 관련된 개발 툴링들의 모음인데요. pbkit은 Node.js 런타임에 제공되어야 함에도 Deno를 사용해 개발자 경험을 잃지 않고도 Node.js 툴링을 만들 수 있었습니다.

Pbkit 메인 레포지토리의 모든 코드가 Deno로 작성되어 있지만, Node.js 호환성을 고려해야하는 세 부분이 있습니다.

Pbkit core, pbkit codegen, 그리고 Protobuf 언어서버입니다. 하나하나의 조건이 다 다르기 때문에 서로 다른 방법으로 코드 변환을 진행할 수 밖에 없었는데요. 하나씩 소개를 드리고자 합니다.

# 어떻게 사용하고 있나요

pbkit 라이브러리 개발

NPM / Deno 의존성 없음

브라우저에서 동작해야 함

**pbkit Core**

**import** 문의 확장자만 제거

pbkit core는 NPM 의존성이 없고, 의도적으로 Deno에 대한 의존성도 없게끔 작성되었습니다.

이 코드를 브라우저 위에서 실행 가능하게끔 만들어야 하는데, 특정 벤더에 대한 의존성이 있으면 변환이 힘들기 때문입니다.

이렇게 의존성이 없는 pbkit core는, 간단하게 import 문의 파일 확장자만 제거해서 브라우저에서 실행 가능하도록 만들 수 있습니다.

물론 TypeScript를 JavaScript로 변환하는 과정은 core 코드를 사용하는 사용자의 워크플로우에서 진행해야 합니다.

## 어떻게 사용하고 있나요

pbkit 라이브러리 개발

Deno API 의존성 없음

번들링 될 필요가 있음

**pbkit Codegen**

**esbuild**

**+ esbuild\_deno\_loader**

또 코드젠 부분은 하나의 파일로 번들링할 필요가 있었는데, 이런 경우에는 esbuild에 esbuild\_deno\_loader를 이용하면 Deno 코드를 의존성 resolution 문제 없이 번들링할 수 있습니다.

# 어떻게 사용하고 있나요

pbkit 라이브러리 개발

Protobuf  
Language  
Server

NPM 의존성: `vscode-jsonrpc`

Node.js (VSCode)에서 실행

`dnt (deno-node transform)` 사용

Protocol buffers 언어 서버는 `vscode-jsonrpc`라는 npm 패키지를 이용해 만들어지면서, vscode 위에서 동작하기 때문에 Node.js 런타임에서 실행될 필요가 있습니다. 그래서 Deno 의존성이 있는 LSP는 `dnt`를 이용해 NPM 모듈로 만들어서 vscode plugin에서 설치해 사용할 수 있도록 했습니다.

## 어떻게 사용하고 있나요

pbkit 라이브러리 개발

**pbkit Core**

확장자 제거

**pbkit Codegen**

esbuild

**Protobuf  
Language  
Server**

dnt

이렇게 다양한 상황에서 Node.js와의 호환성을 챙기는 프로젝트에서도 큰 이슈없이 Deno를 사용할 수 있습니다.

## 그래서 프로덕션에서 쓸 수 있을까요?

- 개발 툴링 같이 NPM 의존성이 적은 분야라면 GOOD
- 의존하는 NPM 모듈이 호환되는지 여부 확인 기여해주세요
- Node.js 호환성을 제공해야 하는 입장에서는 큰 문제 없음

그래서 마지막으로, 프로덕션에서 쓸 수 있을까요?

저희 팀에서는 Deno를 이용해 개발 툴링을 만들면서 NPM 패키지의 도움이 꼭 필요한 상황이 적어서 Deno를 쉽게 도입할 수 있었는데요.

이렇게 기존 생태계와의 의존성이 적은 부분을 개발하신다면 Deno를 프로덕션에 도입하는데에 무리가 없다고 생각됩니다.

만약 특정 NPM 모듈에 대한 의존성이 깊은 경우, 그 모듈이 앞서 소개드린 호환 방법들을 통해 작동되는지 체크하는게 우선일 것 같습니다.

그럼에도 호환성이 해결되지 않는 경우, 물론 직접 기여해주시면 좋겠지만, Deno의 노드 호환성 관련 이슈에 리스팅해주시는 것만으로도 메인테이너분들께서 호환되도록 작업을 진행해주실 것입니다.

이렇게 Node.js의 생태계를 사용할 때는 고려해야 할 점이 있지만, 반대로 Deno에서 Node.js에서도 호환되게끔 유지하는 것은 크게 어렵지 않습니다. 일단 코드 자체가 저희가 작성하다보니 pbkit의 예제처럼 의존성을 분리한다던지 하는 전략을 세울 수 있고, 상황에 맞게 dnt나 esbuild를 돌려가며 호환되는 코드를 쉽게 얻을 수 있습니다.

제 개인적으로는, 복잡한 세팅 없이 사용할 수 있고, 생태계에 퍼진 레거시 설계 철학들을 고려하지 않아도 되고, 표준 라이브러리의 많은 도움을 받을 수 있는 Deno로 개발을 진행하면서 JavaScript 코드를 작성하는 것이 다시 한번 재밌어지는 순간을 경험할 수 있었던 것 같습니다. 꼭 프로덕션이 아니여도 좋습니다. 작은 CLI 툴링이나 블로그에서부터 시작하는 것도 좋습니다. 어떤 형태로든 Deno로 코드를 작성해보고, 저와 같은 가슴뛰는 경험을 느껴보셨으면 좋겠습니다.

감사합니다



감사합니당