👋 Hi there! I'm Shane

I'm a nerd and JavaScript enthusiast that joined Episource in January of 2020 Prior to coming aboard, I worked at Home Depot helping to build an internal code school for the company's 3000+ software developers as well as aspiring engineers. Currently, I'm based in Austin TX, with my wife Emily and Daughter Stella. When I'm not writing code, you can usually find me hanging out with my family, hiking, running, or attempting to read a good book.

I'm beyond excited to be a part of the work that Episource is doing. The EpiAnalytics (OCC) team is by-far the smartest and most talented group of developers I've ever worked with. I learn something new from my teammates every single day.

What is Deno?

According to the Deno site

> Deno is a simple, modern and secure runtime for JavaScript and TypeScript that uses V8 and is built in Rust.

So what does that mean to us?

It helps to understand a little bit of the history surronding Deno.

Inception

Deno was created by Ryan Dahl and introduced publicly during a talk titled "10 Things I Regret About Node" in June of 2018.

https://www.youtube.com/watch?v=M3BM9TB-8yA

Some of the highlights from that talk include:

- Initially abandoning promises in favor of callbacks
- `node_modules` & `package.json`
- magic found in:
  - module resolution via `index.js`

    - no file extensions on imports
- The GYP build system
    - V8 doesn't build with GYP anymore
    - JSON-like interface written in Python with several node wrappers for interoperability
- Security
    - *your linter shouldn't get complete access to your computer and network*
- `window vs global`

After leaving the Node project in 2012, Ryan spent a few years primarily working with Go. During that time he found Go had many positive qualities, and eventually he started kicking around the idea of a Go inpired runtime for JavaScript.

fun fact: Deno was originally written in Go until concerns began to arise over double garbage collection (JS & Go) and the team decided to make a switch to Rust

---

Is it pronounced den-o or dē-no?

In the talk linked above (and subsequent talks over the next months) Ryan always referred to it as den-o. The team eventually decided that since they already had a Dinosaur mascot dē-no would be a better fit.

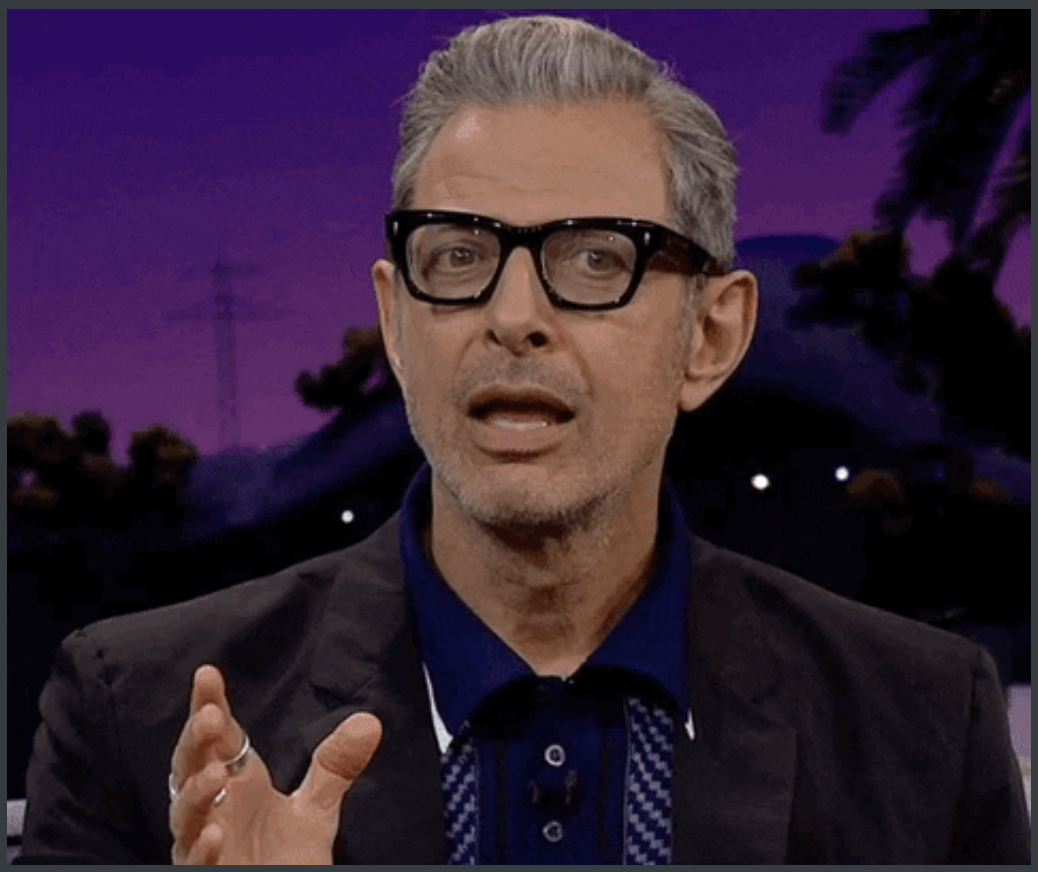
Why?

If you're saying to yourself: "Okay Shane, this is sounds neat. But why? Node seems to work fine"

You wouldn't be wrong...

It's imperative to understand that Node will be around for a very long time and Deno is not looking to replace Node.

Deno is simply an alternative based on a more modern approach to software development. After all, a lot has changed since Node was introduced in 2009...



In a recent interview with Deno core team member Kitson Kelly, he made the astute observation that:

> Deno is like a browser for modules

Deno does this by building on past experiences while embracing evolutions within the industry.

How does Deno work?

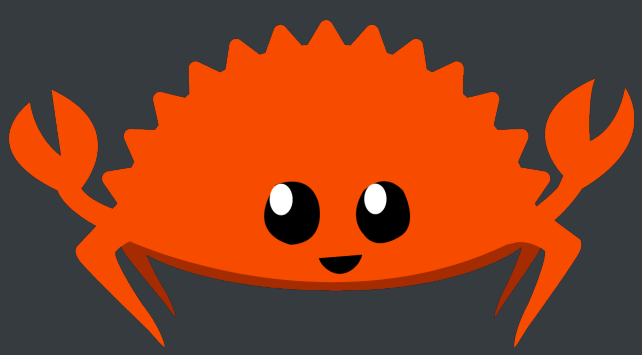
In many ways, you can imagine that Deno is built like modern web apps. It happens to have a "front-end" that most developers will interact with via TS & JS and a Rust backend. According to the core team, you won't need to *know* Rust unless you want to get involved with contributions to the project.

Deno's "backend"

Like Node, Deno makes use of the trusty Chrome V8 Engine - a battle tested tool for rendering JS

Rust & Tokio

As stated previously, Deno's internals are written in Rust.
Through the rusty_v8 crate (library) Deno is able to facilitate communication between Rust and V8.

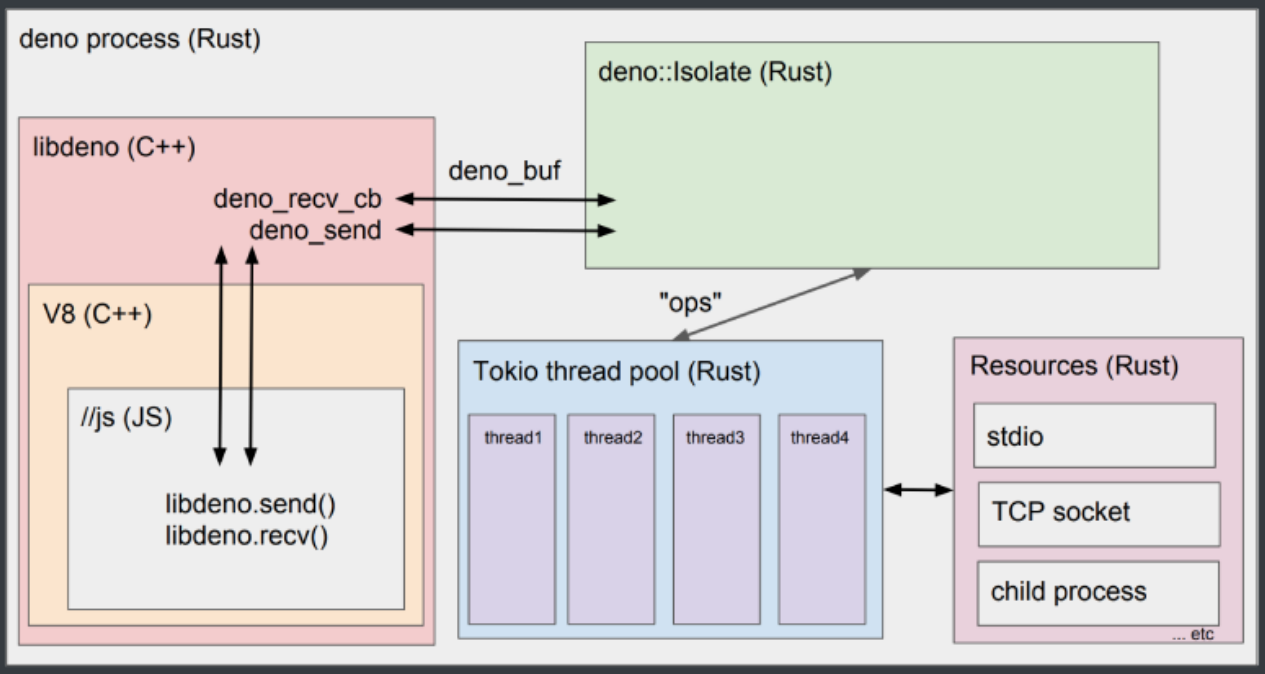


In addition, Tokio - is used to handle async tasks, returning a **future** (the rust version of a promise) when a task is complete.

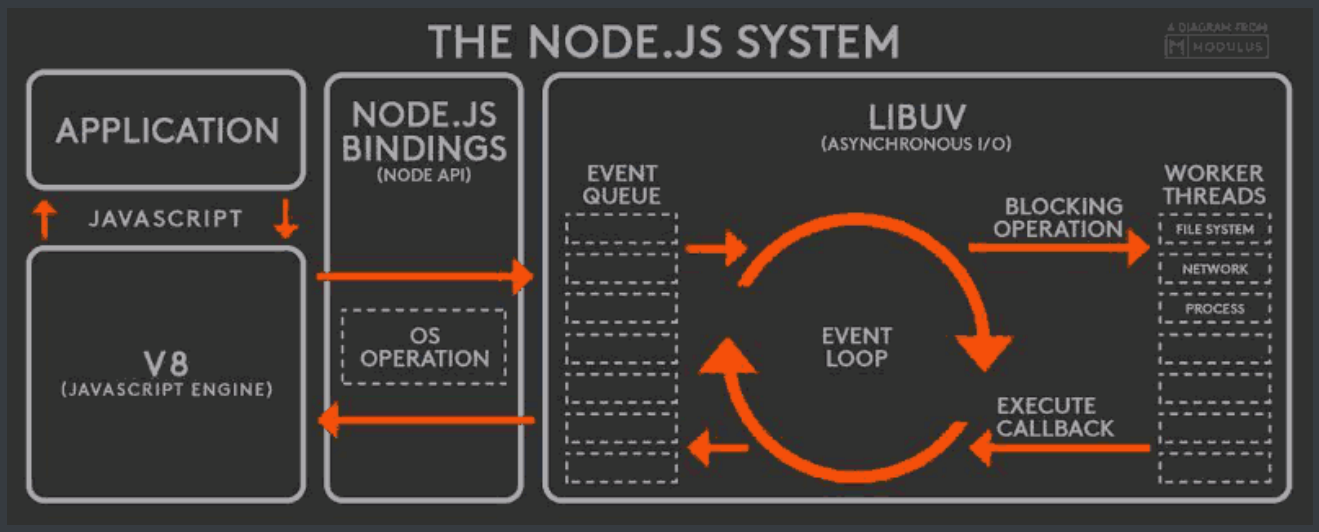
What is Tokio you ask?
Well, according to the docs

> Tokio is an event-driven, non-blocking I/O platform for writing asynchronous applications with the Rust programming language. At a high level, it provides a few major components:
>
> - Tools for working with asynchronous tasks, including synchronization primitives and channels and timeouts, delays, and intervals.
> - APIs for performing asynchronous I/O, including TCP and UDP sockets, filesystem operations, and process and signal management.
> - A runtime for executing asynchronous code, including a task scheduler, an I/O driver backed by the operating system's event queue...

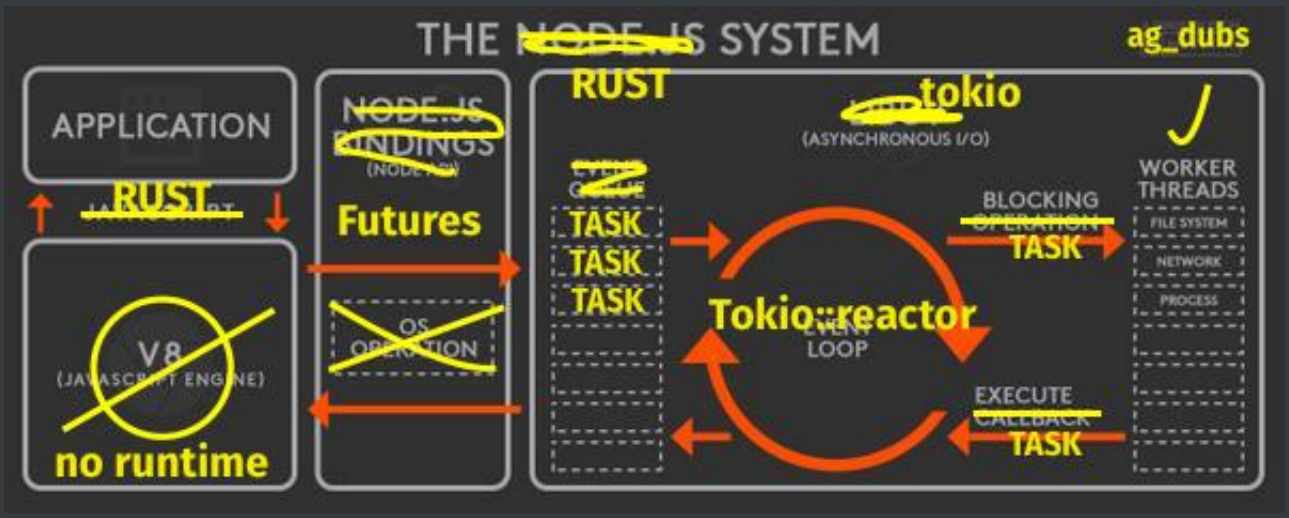
For our purpses, we can say that Tokio essentially takes the place of libuv for Deno



A comparison of the Node architecture with libuv



One more take from someone on the Rust core team:



What sets Deno apart

Aside from the internals and architecture, Deno presents a few key advantages. Let's take a look at several highlights.

TypeScript is a first class language

TS is compiled without any additional config.

This is a big deal! If you've ever tried setting up TypeScript for a project, you know it can be cumbersome to get up and running.

In addition to the TypeScript support, Deno ships type definitions for the runtime

JS => TS => Rust

Part of the vision Deno promotes is that your code should be able to evovle.

- prototype in JS
- move production code to TS
- convert intense computational tasks to Rust

Built in tooling

Deno doesn't stop with automatic TypeScript compliation. You also get:

- bundler (deno bundle)
- debugger (--inspect, --inspect-brk)
- dependency inspector (deno info)
- documentation generator (deno doc)
- formatter (deno fmt)
- test runner (deno test)
- linter (deno lint)

Let's take a look at several in action

---

deno bundle

Deno provides a bundle feature that outputs a single JS file that includes all of your dependencies.

Bundling files can be a tedious and time-consuming process. If you've ever rolled your own webpack config, you're well aware of the pain points.

To bundle, you simply provide a file/url and the output destination. A single file will be placed exactly where you specified.

```
deno bundle https://deno.land/std@0.66.0/examples/colors.ts colors.bundle.js
deno run colors.bundle.js
```

note: you can also import these bundles for use elsewhere

```
import { foo, bar } from "./lib.bundle.js";
```

This inclues the browser!

```
<script type="module" src="website.bundle.js"></script>
```

---

deno doc

With `deno doc` command we can print JSDoc comments to `stdout`

We can try this with the `http` module

```
deno doc https://deno.land/std/http/mod.ts
```

This will output lots of data, but if you look closely, you'll see that the info is exactly what you'd be looking for if you're building a server.

Take the `listenAndServe` function:

```
async function listenAndServe(addr: string | HTTPOptions, handler: (req:
ServerRequest) => void): Promise<void>
  Start an HTTP server with given options and request handler

      const body = "Hello World\n";
      const options = { port: 8000 };
      listenAndServe(options, (req) => {
        req.respond({ body });
      });

  @param options Server configuration
  @param handler Request handler

Defined in https://deno.land/std/http/server.ts:360:0
```

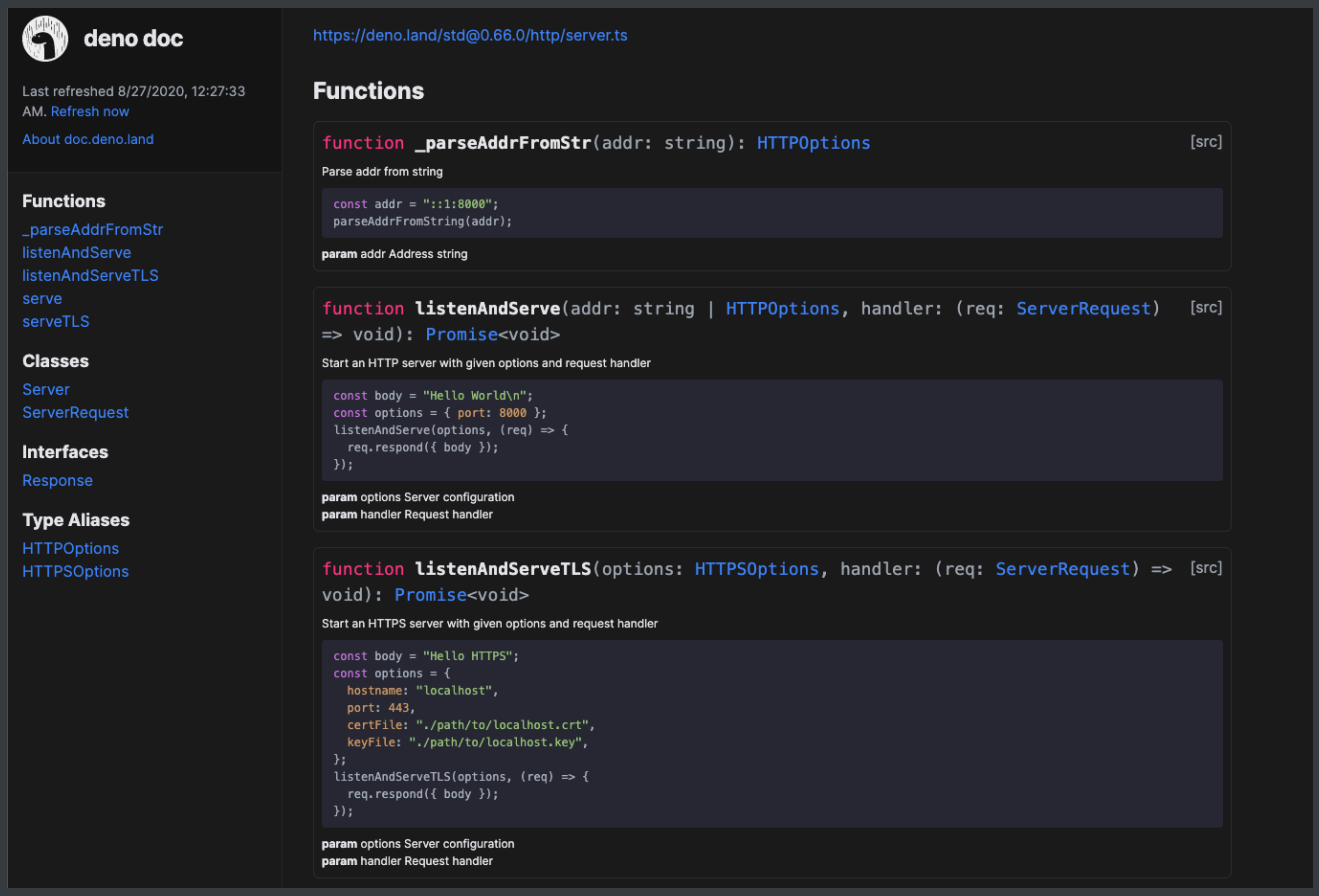
This even includes a link to the source code!

---

deno doc website

If we follow the source code link https://deno.land/std/http/server.ts:360:0 we'll be taken to the exact line number!

From there, we can review the code or click the `View Documentation` button to see the docs hosted in a nice format on the very pleasant `deno doc` website.



`--json`

By passing along the `--json` flag, we can generate JSON output that can be consumed by the deno doc website, should you choose to host your docs there.

Lint & Format

When working with Go, I quickly came to appreciate `gofmt` and `golint` as they *just worked*

Deno aims to do the same with their respective `deno fmt` and `deno lint` commands. This is great as it saves considerable time with setup and unnecessary maintenance of prettier and eslint configs.

While it's not a game changer, these features are very nice to have.

note: at the time of this writing `deno lint` must me run with the unstable command

---

Standard Library

Another inspiration borrowed from Go is the concept of a standard library that does not have any dependencies outside of the standard library.

According to the docs, Deno's standard library is a loose port of Go's standard library.
The docs go on to say:

> When in doubt, simply port Go's source code, documentation, and tests. There are many times when the nature of JavaScript, TypeScript, or Deno itself justifies diverging from Go, but if possible we want to leverage the energy that went into building Go.

We caught a glipmse of the standard library when working with `http` earlier.

The standard lib also includes modules for:

- uuid
- path
- fs
- datetime
- Encoding
- etc

You can consider these modules to be safe and use for any project as each module is officialy reviewed by the Deno team.

To utilize the standard library we can import the file or URL directly:

```
import { v4 } from 'https://deno.land/std/uuid/mod.ts';

const uuid = v4.generate();
const isValid = v4.validate(uuid);
console.log(`your UUID is: ${uuid} 🤫don't tell anyone!`);
```

Running this file should output something similar to: `your UUID is: 4130beb7-104e-4293-85dd-96726e75e1ea 🤫 don't tell anyone!`

Going back to `http` we can use the standard library to spin up a simple server

```
import { serve } from "https://deno.land/std/http/server.ts";
const server = serve({ port: 8000 });
console.log("http://localhost:8000/");
for await (const req of server) {
  req.respond({ body: "Hello World\n" });
}
```

note: to run this file, you'll need to enable network permissions (more on security later) with the `--allow-net` flag:

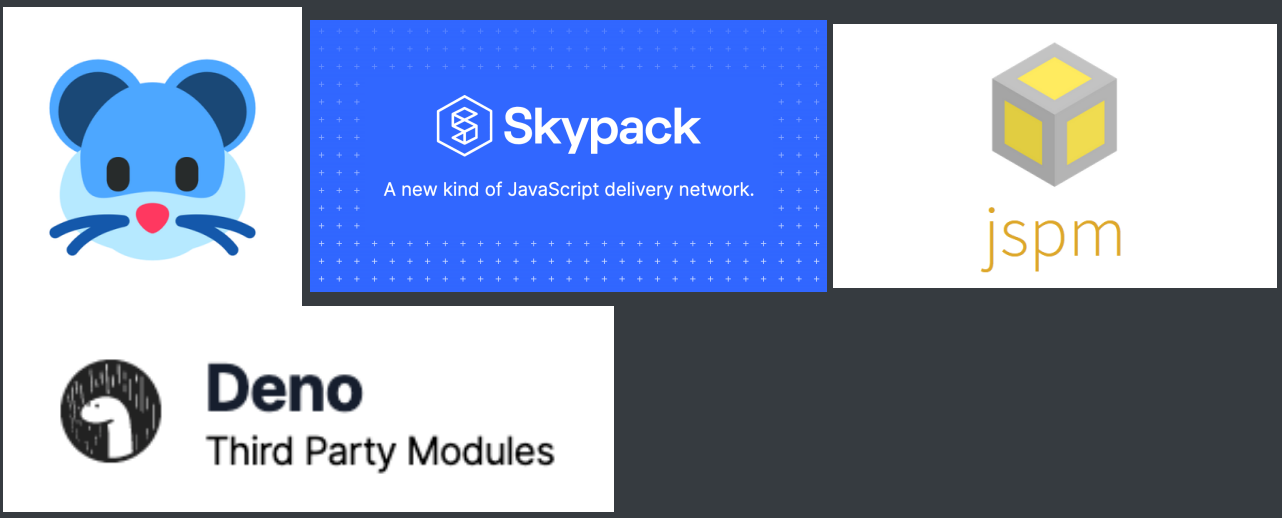
```
$ deno run --allow-net simple-server.ts
```

Decentralized Package Management

One of the more controversial aspects of Deno is the conscious choice to exclude a centralized package management system like npm.

Also, similar to Go, you can simply import packages via URL or file paths. These packages can be hosted in places like:

- deno.land/x
- GitHub
- CDN's such as Pika/Snowpack or jspm
- Private http servers



Module === File === URL

The approach here is so simple it can be strangely difficult to grasp. To make use of a package we can simply `import` the file or url

for example:

```
import { camelCase } from 'https://deno.land/x/lodash@4.17.15-es/lodash.js';

console.log(camelCase('__FOO_BAR__'));
```

note: file extensions are required

ES Modules

You'll see above that `-es` is appeneded to the URL above: `lodash@version-es`

Because Deno is *like a browser for modules*, the modern ES Module syntax is used by deafult. Meaning that any package using ES2015 imports should work in your project. In the case of lodash, we need to ensure the appropriate package is being used.

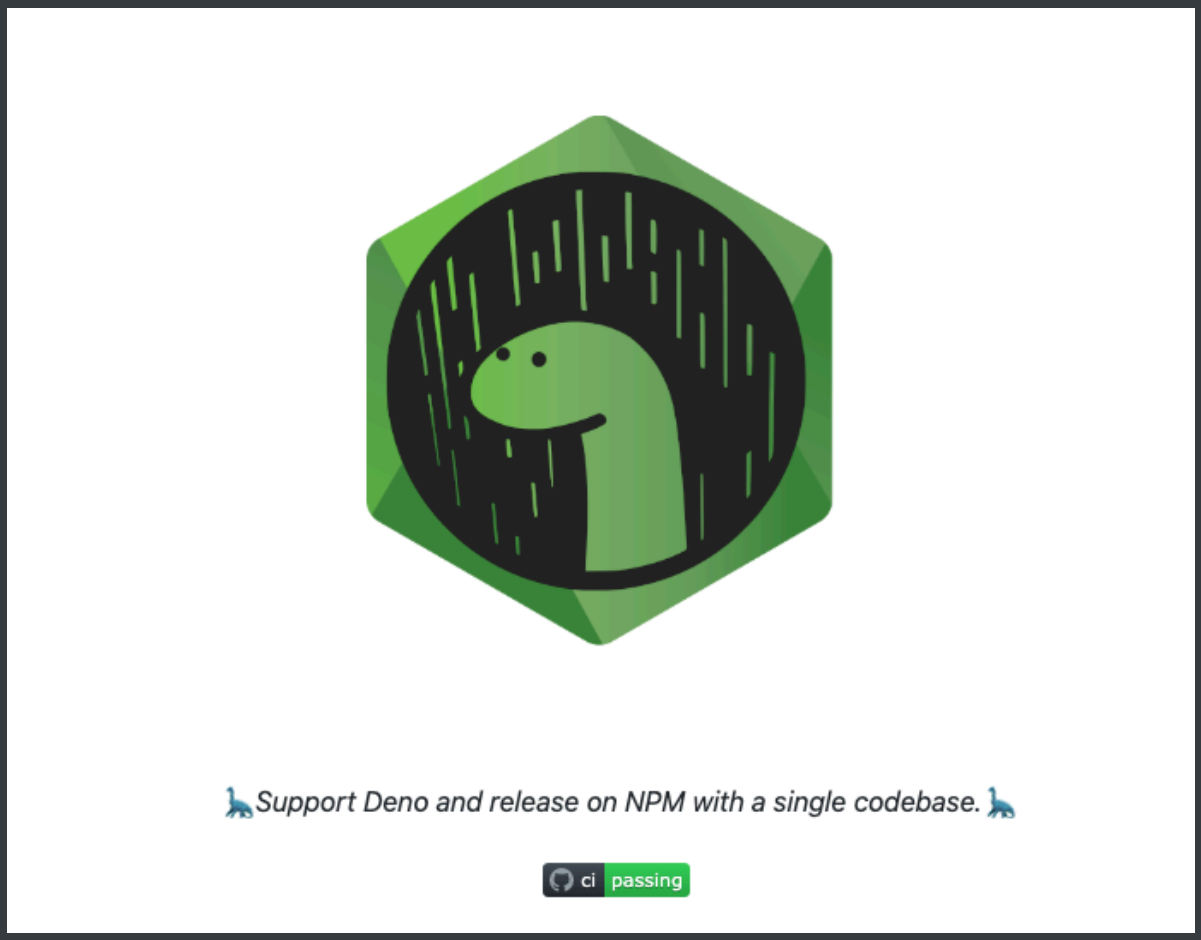
This also means no more common.js `require` statements 🎉

It's also worth nothing this can lead to compatibility issues for libraries only using common.js, which is where services like Pika/Skypack come in.

```
import _ from 'https://dev.jspm.io/lodash@4.17.15';

console.log(_.camelCase('__FOO_BAR__'));
```

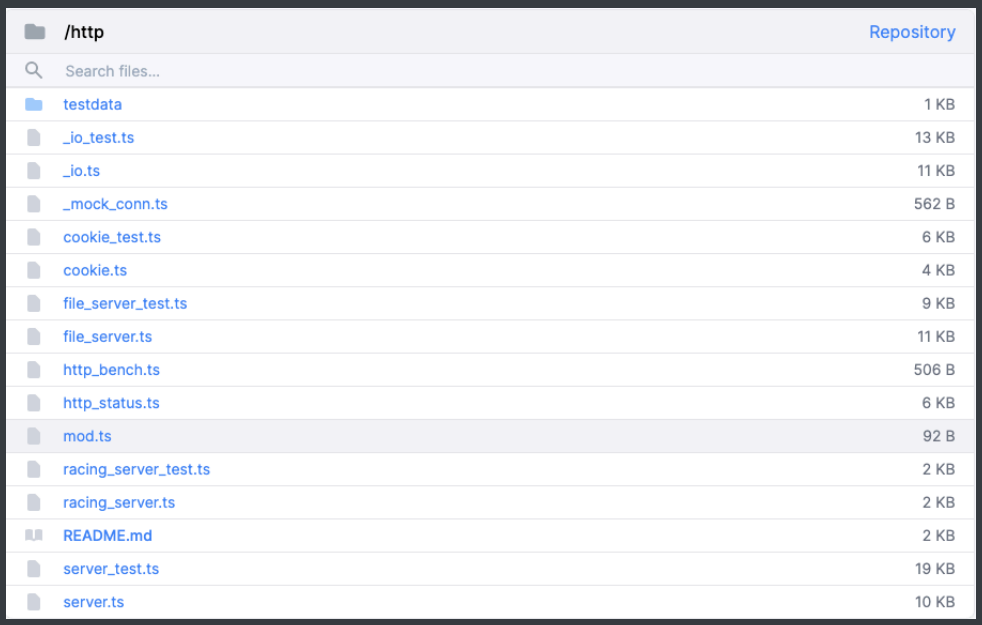
In addition, there are interesting projects like Denoify popping up that aim to assist library maintainers in supporting Node/npm & Deno moving forward.



####

`mod.ts` & `deps.ts`

It's also worth noting that a convention has been established around the idea of a `mod.ts` serving as the entry point to your module.



Take `deno.land/std@0.66.0/http/mod.ts` as an example:

```
export * from "./cookie.ts";
export * from "./http_status.ts";
export * from "./server.ts";
```

the src can be found @ https://deno.land/std@0.66.0/http/mod.ts where you'll notice this file is extremely clean. Only containing `exports`

Another best practice that is being established within the Deno world is the idea of a `deps.ts` file.

This singular file serves as a central place for all of your project's dependencies.

As an example, we can look at Denon - a Deno based implentation of nodemon

```
// provide better logging, see src/log.ts
export * as log from "https://deno.land/std@0.63.0/log/mod.ts";
export { LogRecord } from "https://deno.land/std@0.63.0/log/logger.ts";
export {
  LogLevels,
  LevelName as LogLevelName,
} from "https://deno.land/std@0.63.0/log/levels.ts";
export { BaseHandler } from "https://deno.land/std@0.63.0/log/handlers.ts";
```

```
// ...

// configuration parsing and writing (JSON)
export { readJson } from "https://deno.land/std@0.63.0/fs/read_json.ts";
export { writeJson } from "https://deno.land/std@0.63.0/fs/write_json.ts";

// event control
export { deferred, delay } from "https://deno.land/std@0.63.0/async/mod.ts";

// permission management
export { grant } from "https://deno.land/std@0.63.0/permissions/mod.ts";

// // autocomplete
// export * as omelette from
"https://raw.githubusercontent.com/eliassjogreen/omelette/patch-
1/deno/omelette.ts";

// did you mean
export { default as levenshtein } from
"https://deno.land/x/levenshtein@v1.0.1/mod.ts";
```

_note: this file has been truncated for clarity. The full file can be seen @ https://deno.land/x/denon@2.3.2/deps.ts

---

package.json

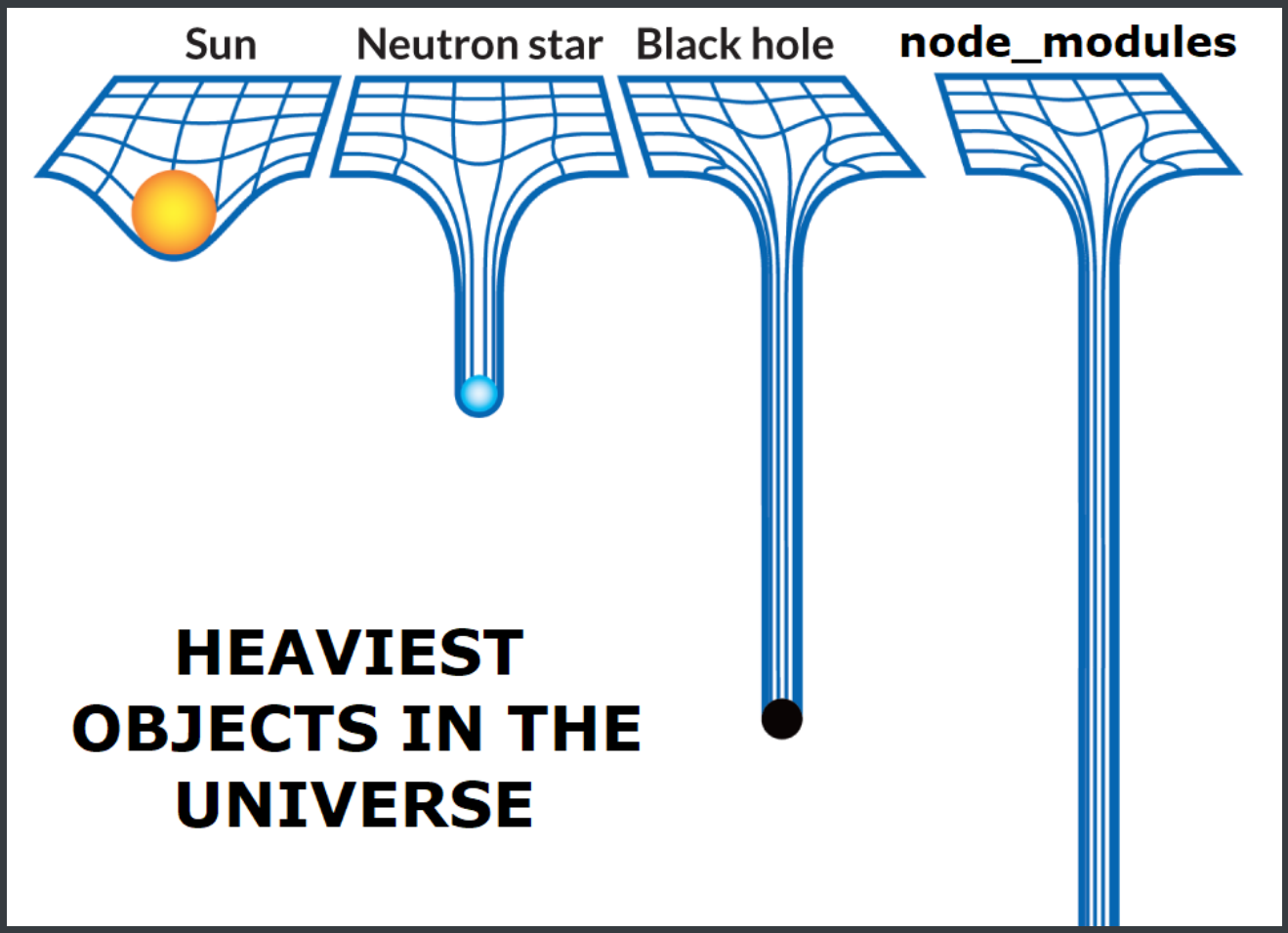
Speaking of configurations, you may have noticed that `package.json` has yet to come up. This is because deno does not **need** a `package.json` file.

I'll give you a minute

Node Modules & Caching

It's hard to remember a time before `package.json` was the default, but it hasn't always been this way. In reality, `package.json` was introduced primarily as a way to manage `node_modules` and over time evolved into what it is today.

While we're on the subject of `node_modules` it's now time to breath a sigh of relief as I'm happy to inform you that Deno does not require `node_modules` to use depenencies.



DENO_DIR

Instead, according to the Docs:

> Deno caches remote imports in a special directory specified by the `DENO_DIR` environment variable.

The docs go on to say:

> The next time you run the program, no downloads will be made. If the program hasn't changed, it won't be recompiled either. The default directory is:
>
> - On Linux/Redox: `$XDG_CACHE_HOME/deno` or `$HOME/.cache/deno`
> - On Windows: `%LOCALAPPDATA%/deno` (`%LOCALAPPDATA%` = `FOLDERID_LocalAppData`)
> - On macOS: `$HOME/Library/Caches/deno`

> - If something fails, it falls back to `$HOME/.deno`

This is another concept leveraged from Go and one that I've found to be a very pleasant experience overall.

Compatibility with web standards

Deno embraces the idea of "Always bet on the web" by implementing modern web standards where possible.

This is great as it removes the need for workarounds such as `node_fetch`

```
const response = await fetch('https://jsonplaceholder.typicode.com/todos/1');
const todo = await response.json();
console.log(todo);
```

While there is not full feature parity, Deno supports many of the common API's and standards we use today, including:

- Event Listeners
- Fetch
- Clear & Set Interval
- Blob
- File
- Form Data
- Headers
- Request & Response
- Console
- URL Search Params
- Window
- Workers

Top Level Await

You may have looked at the `fetch` example above and thought it was odd to use `await` outside of a function.

Deno makes this possible because asynchrounous events *always* return promises. And while you can write this code with promise chains, a lot of time and attention has gone into the ergonomic benefits provided by `async/await`

same example w/o await

```
    const todo = fetch('https://jsonplaceholder.typicode.com/todos/1')
      .then(response => response.json())
      .then(json => console.log(json))

    console.log(todo)
```

Security First Approach

Deno takes a security first approach to building applications. Much like a browser, programs being run from the web should not have access to your file system unless *you allow it*. So, out of the box, Deno requires you to grant permissions for things like:

- network access
- system read/write
- Environment values
- Running subroutines

You'll find that Deno takes the Principle of least privilege approach, and promotes the concept that a suspect should be given only those privleges needed for it to complete the specified task.

In other words, Deno takes an opt-in approach where you grant permissions via command line flags.

For example, take the example we used earlier with the http module from the standard library (snippet below). If we remove the `--allow-net` flag you'll get the following message

```
error: Uncaught PermissionDenied: network access to "0.0.0.0:8000", run again with the --allow-net flag
    at unwrapResponse (rt/10_dispatch_json.js:24:13)
    at sendSync (rt/10_dispatch_json.js:51:12)
    at opListen (rt/30_net.js:33:12)
    at Object.listen (rt/30_net.js:204:17)
    at serve (server.ts:287:25)
    at simple-server.ts:2:16
```

in case you need the code

```
import { serve } from "https://deno.land/std/http/server.ts";
const server = serve({ port: 8000 });
console.log("http://localhost:8000/");
for await (const req of server) {
  req.respond({ body: "Hello World\n" });
}
```

A full list of permissions can be found with: `$ deno run --h`

tip: Check out the Drake package for how to setup `Make-like` tasks for enabling specific permissions by default

---

Summary

We've come to the end. And while Deno is very young, it's clear to see that the creators are standing on the shoulders of giants and building a robust runtime for modern software development.

Is it ready for use today?
Honestly, I think that depends on the task you're trying to accomplish. If you need lots of third party libraries that rely on common.js, probably not.

Should we ditch our existing Node projects?
No! Even the core team of Deno clearly states that Node will be around for many years to come. Deno is simply a new library built from the lessons learned along the way.

In many ways, Deno mirrors the progression of our own careers. And, imo deserves consideration for future projects where security, type safety, and performance come into play.

Thanks 👋