Introduction

Deno is a JavaScript/TypeScript runtime with secure defaults and a great developer experience.

It's built on V8, Rust, and Tokio.

Feature Highlights

- Secure by default. No file, network, or environment access (unless explicitly enabled).
- Supports TypeScript out of the box.
- Ships a single executable (`deno`).
- Has built-in utilities like a dependency inspector (`deno info`) and a code formatter (`deno fmt`).
- Has a set of reviewed (audited) standard modules that are guaranteed to work with Deno.
- Scripts can be bundled into a single JavaScript file.

Philosophy

Deno aims to be a productive and secure scripting environment for the modern programmer.

Deno will always be distributed as a single executable. Given a URL to a Deno program, it is runnable with nothing more than the ~15 megabyte zipped executable. Deno explicitly takes on the role of both runtime and package manager. It uses a standard browser-compatible protocol for loading modules: URLs.

Among other things, Deno is a great replacement for utility scripts that

may have been historically written with bash or python.

Goals

- Only ship a single executable (`deno`).
- Provide Secure Defaults
  - Unless specifically allowed, scripts can't access files, the environment, or the network.
- Browser compatible: The subset of Deno programs which are written completely in JavaScript and do not use the global `Deno` namespace (or feature test for it), ought to also be able to be run in a modern web browser without change.
- Provide built-in tooling like unit testing, code formatting, and linting to improve developer experience.
- Does not leak V8 concepts into user land.
- Be able to serve HTTP efficiently

Comparison to Node.js

- Deno does not use `npm`
  - It uses modules referenced as URLs or file paths
- Deno does not use `package.json` in its module resolution algorithm.
- All async actions in Deno return a promise. Thus Deno provides different APIs than Node.
- Deno requires explicit permissions for file, network, and environment access.

- Deno always dies on uncaught errors.
- Uses "ES Modules" and does not support `require()`. Third party modules are imported via URLs:

```
import * as log from "https://deno.land/std/log/mod
  ↪ .ts";
```

Other key behaviors

- Remote code is fetched and cached on first execution, and never updated until the code is run with the `--reload` flag. (So, this will still work on an airplane.)
- Modules/files loaded from remote URLs are intended to be immutable and cacheable.

Logos

These Deno logos, like the Deno software, are distributed under the MIT license (public domain and free for use)

- [A hand drawn one by @ry](https://deno.land/images/deno_logo.p
- [An animated one by @hashrock](https://github.com/denolib/anim
  deno-logo/)
- [A high resolution SVG one by @kevinkassimo](https://github.com/
  res-deno-logo)
- [A pixelated animation one by @tanakaworld](https://deno.land/im

Getting Started

In this chapter we'll discuss:

- Installing Deno
- Setting up your environment
- Running a `Hello World` script
- Writing our own script
- Command line interface
- Understanding permissions
- Using Deno with TypeScript
- Using WebAssembly

Installation

Deno works on macOS, Linux, and Windows. Deno is a single binary executable. It has no external dependencies.

Download and install

deno_install provides convenience scripts to download and install the binary.

Using Shell (macOS and Linux):

```
curl -fsSL https://deno.land/x/install/install.sh | sh
```

Using PowerShell (Windows):

```
iwr https://deno.land/x/install/install.ps1 -useb | iex
```

Using Scoop (Windows):

```
scoop install deno
```

Using Chocolatey (Windows):

```
choco install deno
```

Using Homebrew (macOS):

```
brew install deno
```

Using Cargo (Windows, macOS, Linux):

```
cargo install deno
```

Deno binaries can also be installed manually, by downloading a zip file at github.com/denoland/deno/releases. These packages contain just a single executable file. You will have to set the executable bit on macOS and Linux.

Testing your installation

To test your installation, run `deno --version`. If this prints the Deno version to the console the installation was successful.

Use `deno help` to see help text documenting Deno's flags and usage. Get a detailed guide on the CLI here.

Updating

To update a previously installed version of Deno, you can run:

```
deno upgrade
```

This will fetch the latest release from github.com/denoland/deno/releases, unzip it, and replace your current executable with it.

You can also use this utility to install a specific version of Deno:

```
deno upgrade --version 1.0.1
```

Building from source

Information about how to build from source can be found in the `Contributing` chapter.

Setup your environment

To productively get going with Deno you should set up your environment. This means setting up shell autocomplete, environmental variables and your editor or IDE of choice.

Environmental variables

There are several env vars that control how Deno behaves:

`DENO_DIR` defaults to `$HOME/.cache/deno` but can be set to any path to control where generated and cached source code is written and read to.

`NO_COLOR` will turn off color output if set. See https://no-color.org/. User code can test if `NO_COLOR` was set without having `--allow-env` by using the boolean constant `Deno.noColor`.

Shell autocomplete

You can generate completion script for your shell using the `deno completions <shell>` command. The command outputs to stdout so you should redirect it to an appropriate file.

The supported shells are:

- zsh
- bash
- fish

- powershell
- elvish

Example:

```
deno completions bash > /usr/local/etc/bash_completion.d
  ↪ /deno.bash
source /usr/local/etc/bash_completion.d/deno.bash
```

Editors and IDEs

Because Deno requires the use of file extensions for module imports and allows http imports, and most editors and language servers do not natively support this at the moment, many editors will throw errors about being unable to find files or imports having unnecessary file extensions.

The community has developed extensions for some editors to solve these issues:

VS Code The beta version of vscode_deno is published on the Visual Studio Marketplace. Please report any issues.

JetBrains IDEs Support for JetBrains IDEs is available through the Deno plugin.

For more information on how to set-up your JetBrains IDE for Deno, read this comment on YouTrack.

Vim and NeoVim Vim works fairly well for Deno/TypeScript if you install CoC (intellisense engine and language server protocol).

After CoC is installed, from inside Vim, run `:CocInstall coc-tsserver` and `:CocInstall coc-deno`. To get autocompletion working for Deno type definitions run `:CocCommand deno.types`. Optionally restart the CoC server `:CocRestart`. From now on, things like `gd` (go to definition) and `gr` (goto/find references) should work.

Emacs Emacs works pretty well for a TypeScript project targeted to Deno by using a combination of tide which is the canonical way of using TypeScript within Emacs and typescript-deno-plugin which is what is used by the official VSCode extension for Deno.

To use it, first make sure that `tide` is setup for your instance of Emacs. Next, as instructed on the typescript-deno-plugin page, first `npm install --save-dev typescript-deno-plugin typescript` in your project (`npm init -y` as necessary), then add the following block to your `tsconfig.json` and you are off to the races!

```
{
  "compilerOptions": {
    "plugins": [
      {
        "name": "typescript-deno-plugin",
        "enable": true, // default is `true`
        "importmap": "import_map.json"
      }
    ]
  }
}
```

If you don't see your favorite IDE on this list, maybe you can develop an extension. Our community Discord group can give you some pointers on where to get started.

First steps

This page contains some examples to teach you about the fundamentals of Deno.

This document assumes that you have some prior knowledge of JavaScript, especially about `async/await`. If you have no prior knowledge of JavaScript, you might want to follow a guide on the basics of JavaScript before attempting to start with Deno.

Hello World

Deno is a runtime for JavaScript/TypeScript which tries to be web compatible and use modern features wherever possible.

Browser compatibility means a `Hello World` program in Deno is the same as the one you can run in the browser:

```
console.log("Welcome to Deno 🦕");
```

Try the program:

```
deno run https://deno.land/std/examples/welcome.ts
```

Making an HTTP request

Many programs use HTTP requests to fetch data from a webserver. Let's write a small program that fetches a file and prints its contents out to the terminal.

Just like in the browser you can use the web standard `fetch` API to make HTTP calls:

```
const url = Deno.args[0];
const res = await fetch(url);
```

```
const body = new Uint8Array(await res.arrayBuffer());
await Deno.stdout.write(body);
```

Let's walk through what this application does:

1. We get the first argument passed to the application, and store it in the `url` constant.
2. We make a request to the url specified, await the response, and store it in the `res` constant.
3. We parse the response body as an `ArrayBuffer`, await the response, and convert it into a `Uint8Array` to store in the `body` constant.
4. We write the contents of the `body` constant to `stdout`.

Try it out:

```
deno run https://deno.land/std/examples/curl.ts https://
  ↪ example.com
```

You will see this program returns an error regarding network access, so what did we do wrong? You might remember from the introduction that Deno is a runtime which is secure by default. This means you need to explicitly give programs the permission to do certain 'privileged' actions, such as access the network.

Try it out again with the correct permission flag:

```
deno run --allow-net=example.com https://deno.land/std/
  ↪ examples/curl.ts https://example.com
```

Reading a file

Deno also provides APIs which do not come from the web. These are all contained in the `Deno` global. You can find documentation for these APIs on doc.deno.land.

Filesystem APIs for example do not have a web standard form, so Deno provides its own API.

In this program each command-line argument is assumed to be a filename, the file is opened, and printed to stdout.

```
const filenames = Deno.args;
for (const filename of filenames) {
  const file = await Deno.open(filename);
  await Deno.copy(file, Deno.stdout);
  file.close();
}
```

The `copy()` function here actually makes no more than the necessary kernel→userspace→kernel copies. That is, the same memory from which data is read from the file, is written to stdout. This illustrates a general design goal for I/O streams in Deno.

Try the program:

```
deno run --allow-read https://deno.land/std/examples/cat
  ↪ .ts /etc/passwd
```

TCP server

This is an example of a server which accepts connections on port 8080, and returns to the client anything it sends.

```
const hostname = "0.0.0.0";
```

```
const port = 8080;
const listener = Deno.listen({ hostname, port });
console.log(`Listening on ${hostname}:${port}`);
for await (const conn of listener) {
  Deno.copy(conn, conn);
}
```

For security reasons, Deno does not allow programs to access the network without explicit permission. To allow accessing the network, use a command-line flag:

```
deno run --allow-net https://deno.land/std/examples/
  ↪ echo_server.ts
```

To test it, try sending data to it with netcat:

```
$ nc localhost 8080
hello world
hello world
```

Like the `cat.ts` example, the `copy()` function here also does not make unnecessary memory copies. It receives a packet from the kernel and sends it back, without further complexity.

More examples

You can find more examples, like an HTTP file server, in the `Examples` chapter.

Command line interface

Deno is a command line program. You should be familiar with some simple commands having followed the examples thus far and already understand the basics of shell usage.

There are multiple ways of viewing the main help text:

```
# Using the subcommand.
deno help

# Using the short flag -- outputs the same as above.
deno -h

# Using the long flag -- outputs more detailed help text
  ↪  where available.
deno --help
```

Deno's CLI is subcommand-based. The above commands should show you a list of those supported, such as `deno bundle`. To see subcommand-specific help for `bundle`, you can similarly run one of:

```
deno help bundle
deno bundle -h
deno bundle --help
```

Detailed guides to each subcommand can be found here.

Script source

Deno can grab the scripts from multiple sources, a filename, a url, and '-' to read the file from stdin. The later is useful for integration with other applications.

```
deno run main.ts
deno run https://mydomain.com/main.ts
cat main.ts | deno run -
```

Script arguments

Separately from the Deno runtime flags, you can pass user-space arguments to the script you are running by specifying them after the script name:

```
deno run main.ts a b -c --quiet
```

```
// main.ts
console.log(Deno.args); // [ "a", "b", "-c", "--quiet" ]
```

Note that anything passed after the script name will be passed as a script argument and not consumed as a Deno runtime flag. This leads to the following pitfall:

```
# Good. We grant net permission to net_client.ts.
deno run --allow-net net_client.ts

# Bad! --allow-net was passed to Deno.args, throws a net
  ↪  permission error.
deno run net_client.ts --allow-net
```

Some see it as unconventional that:

> a non-positional flag is parsed differently depending on its position.

However:

1. This is the most logical way of distinguishing between runtime flags and script arguments.
2. This is the most ergonomic way of distinguishing between runtime flags and script arguments.
3. This is, in fact, the same behaviour as that of any other popular runtime.

- Try `node -c index.js` and `node index.js -c`. The first will only do a syntax check on `index.js` as per Node's `-c` flag. The second will *execute* `index.js` with `-c` passed to `require ↪ ("process").argv`.

There exist logical groups of flags that are shared between related sub-commands. We discuss these below.

Integrity flags

Affect commands which can download resources to the cache: `deno ↪ cache`, `deno run` and `deno test`.

```
--lock <FILE>     Check the specified lock file
--lock-write      Write lock file. Use with --lock.
```

Find out more about these here.

Cache and compilation flags

Affect commands which can populate the cache: `deno cache`, `deno run` and `deno test`. As well as the flags above this includes those which affect module resolution, compilation configuration etc.

```
--config <FILE>                 Load tsconfig.json
  ↪ configuration file
--importmap <FILE>              UNSTABLE: Load import map
  ↪ file
--no-remote                     Do not resolve remote
  ↪ modules
--reload=<CACHE_BLACKLIST>      Reload source code cache (
  ↪ recompile TypeScript)
```

```
--unstable                    Enable unstable APIs
```

Runtime flags

Affect commands which execute user code: `deno run` and `deno test`. These include all of the above as well as the following.

Permission flags These are listed here.

Other runtime flags More flags which affect the execution environment.

```
--cached-only                 Require that remote
  ↪ dependencies are already cached
--inspect=<HOST:PORT>         activate inspector on host:
  ↪ port ...
--inspect-brk=<HOST:PORT>     activate inspector on host:
  ↪ port and break at ...
--seed <NUMBER>               Seed Math.random()
--v8-flags=<v8-flags>         Set V8 command line options
  ↪ . For help: ...
```

Permissions

Deno is secure by default. Therefore, unless you specifically enable it, a deno module has no file, network, or environment access for example. Access to security sensitive areas or functions requires the use of permissions to be granted to a deno process on the command line.

For the following example, `mod.ts` has been granted read-only access to the file system. It cannot write to it, or perform any other security sensitive functions.

```
deno run --allow-read mod.ts
```

Permissions list

The following permissions are available:

- **-A, –allow-all** Allow all permissions. This disables all security.
- **–allow-env** Allow environment access for things like getting and setting of environment variables.
- **–allow-hrtime** Allow high resolution time measurement. High resolution time can be used in timing attacks and fingerprinting.
- **–allow-net=<allow-net>** Allow network access. You can specify an optional, comma separated list of domains to provide a whitelist of allowed domains.
- **–allow-plugin** Allow loading plugins. Please note that –allow-plugin is an unstable feature.
- **–allow-read=<allow-read>** Allow file system read access. You can specify an optional, comma separated list of directories or files to provide a whitelist of allowed file system access.
- **–allow-run** Allow running subprocesses. Be aware that subprocesses are not run in a sandbox and therefore do not have the same security restrictions as the deno process. Therefore, use with caution.
- **–allow-write=<allow-write>** Allow file system write access. You can specify an optional, comma separated list of directories or files to provide a whitelist of allowed file system access.

Permissions whitelist

Deno also allows you to control the granularity of some permissions with whitelists.

This example restricts file system access by whitelisting only the `/usr` directory, however the execution fails as the process was attempting to access a file in the `/etc` directory:

```
$ deno run --allow-read=/usr https://deno.land/std/
  ↪ examples/cat.ts /etc/passwd
error: Uncaught PermissionDenied: read access to "/etc/
  ↪ passwd", run again with the --allow-read flag
 $deno$/dispatch_json.ts:40:11
    at DenoError ($deno$/errors.ts:20:5)
    ...
```

Try it out again with the correct permissions by whitelisting `/etc` instead:

```
deno run --allow-read=/etc https://deno.land/std/
  ↪ examples/cat.ts /etc/passwd
```

`--allow-write` works the same as `--allow-read`.

Network access:

fetch.ts:

```
const result = await fetch("https://deno.land/");
```

This is an example on how to whitelist hosts/urls:

```
deno run --allow-net=github.com,deno.land fetch.ts
```

If `fetch.ts` tries to establish network connections to any other domain, the process will fail.

Allow net calls to any host/url:

```
deno run --allow-net fetch.ts
```

Using TypeScript

Deno supports both JavaScript and TypeScript as first class languages at runtime. This means it requires fully qualified module names, including the extension (or a server providing the correct media type). In addition, Deno has no "magical" module resolution. Instead, imported modules are specified as files (including extensions) or fully qualified URL imports. Typescript modules can be directly imported. E.g.

```
import { Response } from "https://deno.land/std@0.53.0/
  ↪ http/server.ts";
import { queue } from "./collections.ts";
```

Using external type definitions

The out of the box TypeScript compiler though relies on both extensionless modules and the Node.js module resolution logic to apply types to JavaScript modules.

In order to bridge this gap, Deno supports three ways of referencing type definition files without having to resort to "magic" resolution.

Compiler hint If you are importing a JavaScript module, and you know where the type definition for that module is located, you can specify the type definition at import. This takes the form of a compiler hint. Compiler hints inform Deno the location of `.d.ts` files and the JavaScript code that is imported that they relate to. The hint is `@deno ↪ -types` and when specified the value will be used in the compiler

instead of the JavaScript module. For example, if you had `foo.js`, but you know that alongside of it was `foo.d.ts` which was the types for the file, the code would look like this:

```
// @deno-types="./foo.d.ts"
import * as foo from "./foo.js";
```

The value follows the same resolution logic as importing a module, meaning the file needs to have an extension and is relative to the current module. Remote specifiers are also allowed.

The hint affects the next `import` statement (or `export ... from` statement) where the value of the `@deno-types` will be substituted at compile time instead of the specified module. Like in the above example, the Deno compiler will load `./foo.d.ts` instead of `./foo.js`. Deno will still load `./foo.js` when it runs the program.

Triple-slash reference directive in JavaScript files If you are hosting modules which you want to be consumed by Deno, and you want to inform Deno about the location of the type definitions, you can utilize a triple-slash directive in the actual code. For example, if you have a JavaScript module and you would like to provide Deno with the location of the type definition which happens to be alongside that file, your JavaScript module named `foo.js` might look like this:

```
/// <reference types="./foo.d.ts" />
export const foo = "foo";
```

Deno will see this, and the compiler will use `foo.d.ts` when type checking the file, though `foo.js` will be loaded at runtime. The resolution of the value of the directive follows the same resolution logic as importing a module, meaning the file needs to have an extension and is relative

to the current file. Remote specifiers are also allowed.

X-TypeScript-Types custom header If you are hosting modules which you want to be consumed by Deno, and you want to inform Deno the location of the type definitions, you can use a custom HTTP header of `X-TypeScript-Types` to inform Deno of the location of that file.

The header works in the same way as the triple-slash reference mentioned above, it just means that the content of the JavaScript file itself does not need to be modified, and the location of the type definitions can be determined by the server itself.

Not all type definitions are supported.

Deno will use the compiler hint to load the indicated `.d.ts` files, but some `.d.ts` files contain unsupported features. Specifically, some `.d.ts` files expect to be able to load or reference type definitions from other packages using the module resolution logic. For example a type reference directive to include `node`, expecting to resolve to some path like `./node_modules/@types/node/index.d.ts`. Since this depends on non-relative "magical" resolution, Deno cannot resolve this.

Why not use the triple-slash type reference in TypeScript files?

The TypeScript compiler supports triple-slash directives, including a type reference directive. If Deno used this, it would interfere with the behavior of the TypeScript compiler. Deno only looks for the directive in JavaScript (and JSX) files.

Custom TypeScript Compiler Options

In the Deno ecosystem, all strict flags are enabled in order to comply with TypeScript's ideal of being `strict` by default. However, in order to provide a way to support customization a configuration file such as `tsconfig.json` might be provided to Deno on program execution.

You need to explicitly tell Deno where to look for this configuration by setting the `-c` (or `--config`) argument when executing your application.

```
deno run -c tsconfig.json mod.ts
```

Following are the currently allowed settings and their default values in Deno:

```
{
  "compilerOptions": {
    "allowJs": false,
    "allowUmdGlobalAccess": false,
    "allowUnreachableCode": false,
    "allowUnusedLabels": false,
    "alwaysStrict": true,
    "assumeChangesOnlyAffectDirectDependencies": false,
    "checkJs": false,
    "disableSizeLimit": false,
    "generateCpuProfile": "profile.cpuprofile",
    "jsx": "react",
    "jsxFactory": "React.createElement",
    "lib": [],
    "noFallthroughCasesInSwitch": false,
    "noImplicitAny": true,
    "noImplicitReturns": true,
    "noImplicitThis": true,
    "noImplicitUseStrict": false,
    "noStrictGenericChecks": false,
```

```
    "noUnusedLocals": false,
    "noUnusedParameters": false,
    "preserveConstEnums": false,
    "removeComments": false,
    "resolveJsonModule": true,
    "strict": true,
    "strictBindCallApply": true,
    "strictFunctionTypes": true,
    "strictNullChecks": true,
    "strictPropertyInitialization": true,
    "suppressExcessPropertyErrors": false,
    "suppressImplicitAnyIndexErrors": false,
    "useDefineForClassFields": false
  }
}
```

For documentation on allowed values and use cases please visit the typescript docs.

Note: Any options not listed above are either not supported by Deno or are listed as deprecated/experimental in the TypeScript documentation.

WASM support

Deno can execute wasm binaries.

```
const wasmCode = new Uint8Array([
  0, 97, 115, 109, 1, 0, 0, 0, 1, 133, 128, 128, 128, 0,
    ↪  1, 96, 0, 1, 127,
  3, 130, 128, 128, 128, 0, 1, 0, 4, 132, 128, 128, 128,
    ↪  0, 1, 112, 0, 0,
  5, 131, 128, 128, 128, 0, 1, 0, 1, 6, 129, 128, 128,
    ↪ 128, 0, 0, 7, 145,
```

```
  128, 128, 128, 0, 2, 6, 109, 101, 109, 111, 114, 121,
    ↪ 2, 0, 4, 109, 97,
  105, 110, 0, 0, 10, 138, 128, 128, 128, 0, 1, 132,
    ↪ 128, 128, 128, 0, 0,
  65, 42, 11
]);
const wasmModule = new WebAssembly.Module(wasmCode);
const wasmInstance = new WebAssembly.Instance(wasmModule
  ↪ );
console.log(wasmInstance.exports.main().toString());
```

Runtime

Documentation for all runtime functions (Web APIs + `Deno` global) can be found on `doc.deno.land`.

Web APIs

For APIs where a web standard already exists, like `fetch` for HTTP requests, Deno uses these rather than inventing a new proprietary API.

The detailed documentation for implemented Web APIs can be found on doc.deno.land. Additionally, a full list of the Web APIs which Deno implements is also available in the repository.

The TypeScript definitions for the implemented web APIs can be found in the `lib.deno.shared_globals.d.ts` and `lib.deno.window.d.ts` files.

Definitions that are specific to workers can be found in the `lib.deno.`
↪ `worker.d.ts` file.

`Deno` global

All APIs that are not web standard are contained in the global `Deno` ↪ namespace. It has the APIs for reading from files, opening TCP sockets, and executing subprocesses, etc.

The TypeScript definitions for the Deno namespaces can be found in the `lib.deno.ns.d.ts` file.

The documentation for all of the Deno specific APIs can be found on doc.deno.land.

Stability

As of Deno 1.0.0, the `Deno` namespace APIs are stable. That means we will strive to make code working under 1.0.0 continue to work in future versions.

However, not all of Deno's features are ready for production yet. Features which are not ready, because they are still in draft phase, are locked behind the `--unstable` command line flag.

```
deno run --unstable mod_which_uses_unstable_stuff.ts
```

Passing this flag does a few things:

- It enables the use of unstable APIs during runtime.
- It adds the `lib.deno.unstable.d.ts` file to the list of TypeScript definitions that are used for type checking. This includes the output of `deno types`.

You should be aware that many unstable APIs have **not undergone a security review**, are likely to have **breaking API changes** in the future, and are **not ready for production**.

Standard modules

Deno's standard modules (https://deno.land/std/) are not yet stable. We currently version the standard modules differently from the CLI to reflect this. Note that unlike the `Deno` namespace, the use of the standard modules do not require the `--unstable` flag (unless the standard module itself makes use of an unstable Deno feature).

Program lifecycle

Deno supports browser compatible lifecycle events: `load` and `unload`. You can use these events to provide setup and cleanup code in your program.

Listeners for `load` events can be asynchronous and will be awaited. Listeners for `unload` events need to be synchronous. Both events cannot be cancelled.

Example:

main.ts

```
import "./imported.ts";

const handler = (e: Event): void => {
  console.log(`got ${e.type} event in event handler (
    ↪ main)`);
};

window.addEventListener("load", handler);

window.addEventListener("unload", handler);

window.onload = (e: Event): void => {
```

```
  console.log(`got ${e.type} event in onload function (
    ↪ main)`);
};

window.onunload = (e: Event): void => {
  console.log(`got ${e.type} event in onunload function
    ↪ (main)`);
};

console.log("log from main script");
```

imported.ts

```
const handler = (e: Event): void => {
  console.log(`got ${e.type} event in event handler (
    ↪ imported)`);
};

window.addEventListener("load", handler);
window.addEventListener("unload", handler);

window.onload = (e: Event): void => {
  console.log(`got ${e.type} event in onload function (
    ↪ imported)`);
};

window.onunload = (e: Event): void => {
  console.log(`got ${e.type} event in onunload function
    ↪ (imported)`);
};

console.log("log from imported script");
```

Note that you can use both `window.addEventListener` and `window.onload`

↪ /`window.onunload` to define handlers for events. There is a major difference between them, let's run the example:

```
$ deno run main.ts
log from imported script
log from main script
got load event in onload function (main)
got load event in event handler (imported)
got load event in event handler (main)
got unload event in onunload function (main)
got unload event in event handler (imported)
got unload event in event handler (main)
```

All listeners added using `window.addEventListener` were run, but `window` ↪ `.onload` and `window.onunload` defined in `main.ts` overrode handlers defined in `imported.ts`.

In other words, you can register multiple `window.addEventListener "` ↪ `load"` or `"unload"` events, but only the last loaded `window.onload` or `window.onunload` events will be executed.

Compiler API

> This is an unstable Deno feature. Learn more about unstable features.

Deno supports runtime access to the built-in TypeScript compiler. There are three methods in the `Deno` namespace that provide this access.

`Deno.compile()`

This works similar to `deno cache` in that it can fetch and cache the code, compile it, but not run it. It takes up to three arguments, the `rootName`, optionally `sources`, and optionally `options`. The `rootName` is the root module which will be used to generate the resulting program. This is like the module name you would pass on the command line in `deno run --reload example.ts`. The `sources` is a hash where the key is the fully qualified module name, and the value is the text source of the module. If `sources` is passed, Deno will resolve all the modules from within that hash and not attempt to resolve them outside of Deno. If `sources` are not provided, Deno will resolve modules as if the root module had been passed on the command line. Deno will also cache any of these resources. All resolved resources are treated as dynamic imports and require read or net permissions depending on if they're local or remote. The `options` argument is a set of options of type `Deno.CompilerOptions`, which is a subset of the TypeScript compiler options containing the ones supported by Deno.

The method resolves with a tuple. The first argument contains any diagnostics (syntax or type errors) related to the code. The second argument is a map where the keys are the output filenames and the values are the content.

An example of providing sources:

```
const [diagnostics, emitMap] = await Deno.compile("/foo.ts", {
  "/foo.ts": `import * as bar from "./bar.ts";\nconsole.log(bar);\n`,
  "/bar.ts": `export const bar = "bar";\n`,
});
```

```
assert(diagnostics == null); // ensuring no diagnostics
  ↪ are returned
console.log(emitMap);
```

We would expect map to contain 4 "files", named `/foo.js.map`, `/foo.js`, `/bar.js.map`, and `/bar.js`.

When not supplying resources, you can use local or remote modules, just like you could do on the command line. So you could do something like this:

```
const [diagnostics, emitMap] = await Deno.compile(
  "https://deno.land/std/examples/welcome.ts"
);
```

In this case `emitMap` will contain a `console.log()` statement.

`Deno.bundle()`

This works a lot like `deno bundle` does on the command line. It is also like `Deno.compile()`, except instead of returning a map of files, it returns a single string, which is a self-contained JavaScript ES module which will include all of the code that was provided or resolved as well as exports of all the exports of the root module that was provided. It takes up to three arguments, the `rootName`, optionally `sources`, and optionally `options`. The `rootName` is the root module which will be used to generate the resulting program. This is like module name you would pass on the command line in `deno bundle example.ts`. The `sources` is a hash where the key is the fully qualified module name, and the value is the text source of the module. If `sources` is passed, Deno will resolve all the modules from within that hash and not attempt to resolve

them outside of Deno. If `sources` are not provided, Deno will resolve modules as if the root module had been passed on the command line. All resolved resources are treated as dynamic imports and require read or net permissions depending if they're local or remote. Deno will also cache any of these resources. The `options` argument is a set of options of type `Deno.CompilerOptions`, which is a subset of the TypeScript compiler options containing the ones supported by Deno.

An example of providing sources:

```
const [diagnostics, emit] = await Deno.bundle("/foo.ts",
  ↪  {
  "/foo.ts": `import * as bar from "./bar.ts";\nconsole.
    ↪ log(bar);\n`,
  "/bar.ts": `export const bar = "bar";\n`,
});

assert(diagnostics == null); // ensuring no diagnostics
  ↪ are returned
console.log(emit);
```

We would expect `emit` to be the text for an ES module, which would contain the output sources for both modules.

When not supplying resources, you can use local or remote modules, just like you could do on the command line. So you could do something like this:

```
const [diagnostics, emit] = await Deno.bundle(
  "https://deno.land/std/http/server.ts"
);
```

In this case `emit` will be a self contained JavaScript ES module with all of its dependencies resolved and exporting the same exports as the

source module.

`Deno.transpileOnly()`

This is based off of the TypeScript function `transpileModule()`. All this does is "erase" any types from the modules and emit JavaScript. There is no type checking and no resolution of dependencies. It accepts up to two arguments, the first is a hash where the key is the module name and the value is the content. The only purpose of the module name is when putting information into a source map, of what the source file name was. The second argument contains optional `options` of the type `Deno.CompilerOptions`. The function resolves with a map where the key is the source module name supplied, and the value is an object with a property of `source` and optionally `map`. The first is the output contents of the module. The `map` property is the source map. Source maps are provided by default, but can be turned off via the `options` argument.

An example:

```
const result = await Deno.transpileOnly({
  "/foo.ts": `enum Foo { Foo, Bar, Baz };\n`,
});

console.log(result["/foo.ts"].source);
console.log(result["/foo.ts"].map);
```

We would expect the `enum` would be rewritten to an IIFE which constructs the enumerable, and the map to be defined.

Referencing TypeScript library files

When you use `deno run`, or other Deno commands which type check TypeScript, that code is evaluated against custom libraries which describe the environment that Deno supports. By default, the compiler runtime APIs which type check TypeScript also use these libraries (`Deno.compile()` and `Deno.bundle()`).

But if you want to compile or bundle TypeScript for some other runtime, you may want to override the default libraries. To do this, the runtime APIs support the `lib` property in the compiler options. For example, if you had TypeScript code that is destined for the browser, you would want to use the TypeScript `"dom"` library:

```
const [errors, emitted] = await Deno.compile(
  "main.ts",
  {
    "main.ts": `document.getElementById("foo");\n`,
  },
  {
    lib: ["dom", "esnext"],
  }
);
```

For a list of all the libraries that TypeScript supports, see the `lib` compiler option documentation.

Don't forget to include the JavaScript library

Just like `tsc`, when you supply a `lib` compiler option, it overrides the default ones, which means that the basic JavaScript library won't be included and you should include the one that best represents your target runtime (e.g. `es5`, `es2015`, `es2016`, `es2017`, `es2018`, `es2019`, `es2020` or

`esnext`).

Including the `Deno` namespace In addition to the libraries that are provided by TypeScript, there are four libraries that are built into Deno that can be referenced:

- `deno.ns` - Provides the `Deno` namespace.
- `deno.shared_globals` - Provides global interfaces and variables which Deno supports at runtime that are then exposed by the final runtime library.
- `deno.window` - Exposes the global variables plus the Deno namespace that are available in the Deno main worker and is the default for the runtime compiler APIs.
- `deno.worker` - Exposes the global variables that are available in workers under Deno.

So to add the Deno namespace to a compilation, you would include the `deno.ns` lib in the array. For example:

```
const [errors, emitted] = await Deno.compile(
  "main.ts",
  {
    "main.ts": `document.getElementById("foo");\n`,
  },
  {
    lib: ["dom", "esnext", "deno.ns"],
  }
);
```

Note that the Deno namespace expects a runtime environment that is at least ES2018 or later. This means if you use a lib "lower" than ES2018 you will get errors logged as part of the compilation.

Using the triple slash reference You do not have to specify the `lib` in the compiler options. Deno also supports the triple-slash reference to a lib. which can be embedded in the contents of the file. For example, if you have a `main.ts` like:

```
/// <reference lib="dom" />

document.getElementById("foo");
```

It would compile without errors like this:

```
const [errors, emitted] = await Deno.compile("./main.ts
  ↪ ", undefined, {
  lib: ["esnext"],
});
```

Note that the `dom` library conflicts with some of the default globals that are defined in the default type library for Deno. To avoid this, you need to specify a `lib` option in the compiler options to the runtime compiler APIs.

Workers

Deno supports `Web Worker API`.

Workers can be used to run code on multiple threads. Each instance of `Worker` is run on a separate thread, dedicated only to that worker.

Currently Deno supports only `module` type workers; thus it's essential to pass the `type: "module"` option when creating a new worker.

Relative module specifiers are not supported at the moment. You can instead use the `URL` contructor and `import.meta.url` to easily create a specifier for some nearby script.

```
// Good
new Worker(new URL("worker.js", import.meta.url).href, {
  ↪  type: "module" });

// Bad
new Worker(new URL("worker.js", import.meta.url).href);
new Worker(new URL("worker.js", import.meta.url).href, {
  ↪  type: "classic" });
new Worker("./worker.js", { type: "module" });
```

Permissions

Creating a new `Worker` instance is similar to a dynamic import; therefore Deno requires appropriate permission for this action.

For workers using local modules; `--allow-read` permission is required:

main.ts

```
new Worker(new URL("worker.ts", import.meta.url).href, {
  ↪  type: "module" });
```

worker.ts

```
console.log("hello world");
self.close();
```

```
$ deno run main.ts
error: Uncaught PermissionDenied: read access to "./
  ↪ worker.ts", run again with the --allow-read flag

$ deno run --allow-read main.ts
hello world
```

For workers using remote modules; `--allow-net` permission is required:

main.ts

```
new Worker("https://example.com/worker.ts", { type: "
  ↪ module" });
```

worker.ts (at https://example.com/worker.ts)

```
console.log("hello world");
self.close();
```

```
$ deno run main.ts
error: Uncaught PermissionDenied: net access to "https
  ↪ ://example.com/worker.ts", run again with the --
  ↪ allow-net flag

$ deno run --allow-net main.ts
hello world
```

Using Deno in worker

> This is an unstable Deno feature. Learn more about unstable features.

By default the `Deno` namespace is not available in worker scope.

To add the `Deno` namespace pass `deno: true` option when creating new worker:

main.js

```
const worker = new Worker(new URL("worker.js", import.
  ↪ meta.url).href, {
  type: "module",
  deno: true,
});
worker.postMessage({ filename: "./log.txt" });
```

worker.js

```
self.onmessage = async (e) => {
  const { filename } = e.data;
  const text = await Deno.readTextFile(filename);
  console.log(text);
  self.close();
};
```

log.txt

```
hello world
```

```
$ deno run --allow-read --unstable main.js
hello world
```

When the `Deno` namespace is available in worker scope, the worker inherits its its parent process' permissions (the ones specified using `--allow-*` flags).

We intend to make permissions configurable for workers.

Linking to third party code

In the Getting Started section, we saw Deno could execute scripts from URLs. Like browser JavaScript, Deno can import libraries directly from URLs. This example uses a URL to import an assertion library:

test.ts

```
import { assertEquals } from "https://deno.land/std/
  ↪ testing/asserts.ts";

assertEquals("hello", "hello");
assertEquals("world", "world");
```

```
console.log("Asserted! ");
```

Try running this:

```
$ deno run test.ts
Compile file:///mnt/f9/Projects/github.com/denoland/deno
  ↪ /docs/test.ts
Download https://deno.land/std/testing/asserts.ts
Download https://deno.land/std/fmt/colors.ts
Download https://deno.land/std/testing/diff.ts
Asserted!
```

Note that we did not have to provide the `--allow-net` flag for this program, and yet it accessed the network. The runtime has special access to download imports and cache them to disk.

Deno caches remote imports in a special directory specified by the `DENO_DIR` environment variable. It defaults to the system's cache directory if `DENO_DIR` is not specified. 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`

FAQ

How do I import a specific version of a module?

Specify the version in the URL. For example, this URL fully specifies the code being run: `https://unpkg.com/liltest@0.0.5/dist/liltest.js` ↪ .

It seems unwieldy to import URLs everywhere.

> What if one of the URLs links to a subtly different version of a library?
>
> Isn't it error prone to maintain URLs everywhere in a large project?

The solution is to import and re-export your external libraries in a central `deps.ts` file (which serves the same purpose as Node's `package.` ↪ `json` file). For example, let's say you were using the above assertion library across a large project. Rather than importing `"https://deno.` ↪ `land/std/testing/asserts.ts"` everywhere, you could create a `deps.` ↪ `ts` file that exports the third-party code:

deps.ts

```
export {
  assert,
  assertEquals,
  assertStrContains,
} from "https://deno.land/std/testing/asserts.ts";
```

And throughout the same project, you can import from the `deps.ts` and avoid having many references to the same URL:

```
import { assertEquals, runTests, test } from "./deps.ts
  ↪ ";
```

This design circumvents a plethora of complexity spawned by package management software, centralized code repositories, and superfluous file formats.

How can I trust a URL that may change?

By using a lock file (with the `--lock` command line flag), you can ensure that the code pulled from a URL is the same as it was during initial development. You can learn more about this here.

But what if the host of the URL goes down? The source won't be available.

This, like the above, is a problem faced by *any* remote dependency system. Relying on external servers is convenient for development but brittle in production. Production software should always vendor its dependencies. In Node this is done by checking `node_modules` into source control. In Deno this is done by pointing `$DENO_DIR` to some project-local directory at runtime, and similarly checking that into source control:

```
# Download the dependencies.
DENO_DIR=./deno_dir deno cache src/deps.ts

# Make sure the variable is set for any command which
  ↪ invokes the cache.
DENO_DIR=./deno_dir deno test src

# Check the directory into source control.
git add -u deno_dir
```

```
git commit
```

Reloading modules

By default, a module in the cache will be reused without fetching or re-compiling it. Sometimes this is not desirable and you can force deno to refetch and recompile modules into the cache. You can invalidate your local `DENO_DIR` cache using the `--reload` flag of the `deno cache` sub-command. It's usage is described below:

To reload everything

```
deno cache --reload my_module.ts
```

To reload specific modules

Sometimes we want to upgrade only some modules. You can control it by passing an argument to a `--reload` flag.

To reload all v0.55.0 standard modules

```
deno cache --reload=https://deno.land/std@v0.55.0
  ↪ my_module.ts
```

To reload specific modules (in this example - colors and file system copy) use a comma to separate URLs

```
deno cache --reload=https://deno.land/std/fs/copy.ts,
  ↪ https://deno.land/std/fmt/colors.ts my_module.ts
```

Integrity checking & lock files

Introduction

Let's say your module depends on remote module `https://some.url/a.ts`. When you compile your module for the first time `a.ts` is retrieved, compiled and cached. It will remain this way until you run your module on a new machine (say in production) or reload the cache (through `deno cache --reload` for example). But what happens if the content in the remote url `https://some.url/a.ts` is changed? This could lead to your production module running with different dependency code than your local module. Deno's solution to avoid this is to use integrity checking and lock files.

Caching and lock files

Deno can store and check subresource integrity for modules using a small JSON file. Use the `--lock=lock.json` to enable and specify lock file checking. To update or create a lock use `--lock=lock.json --lock-write`. The `--lock=lock.json` tells Deno what the lock file to use is, while the `--lock-write` is used to output dependency hashes to the lock file (`--lock-write` must be used in conjunction with `--lock`).

A `lock.json` might look like this, storing a hash of the file against the dependency:

```
{
  "https://deno.land/std@v0.50.0/textproto/mod.ts":
    "3118
    d7a42c03c242c5a49c2ad91c8396110e14acca1324e7aaefd31a
    ",
  "https://deno.land/std@v0.50.0/io/util.ts": "
    ae133d310a0fdcf298cea7bc09a599c49acb616d34e148e263bc
```

```
    ↪ ",
  "https://deno.land/std@v0.50.0/async/delay.ts": "35957
    ↪ d585a6e3dd87706858fb1d6b551cb278271b03f52c5a2cb70e65
    ↪ ",
    ...
}
```

A typical workflow will look like this:

src/deps.ts

```
// Add a new dependency to "src/deps.ts", used somewhere
  ↪  else.
export { xyz } from "https://unpkg.com/xyz-lib@v0.9.0/
  ↪ lib.ts";
```

Then:

```
# Create/update the lock file "lock.json".
deno cache --lock=lock.json --lock-write src/deps.ts

# Include it when committing to source control.
git add -u lock.json
git commit -m "feat: Add support for xyz using xyz-lib"
git push
```

Collaborator on another machine – in a freshly cloned project tree:

```
# Download the project's dependencies into the machine's
  ↪  cache, integrity
# checking each resource.
deno cache --reload --lock=lock.json src/deps.ts

# Done! You can proceed safely.
deno test --allow-read src
```

Runtime verification

Like caching above, you can also use the `--lock=lock.json` option during use of the `deno run` sub command, validating the integrity of any locked modules during the run. Remember that this only validates against dependencies previously added to the `lock.json` file. New dependencies will be cached but not validated.

You can take this a step further as well by using the `--cached-only` flag to require that remote dependencies are already cached.

```
deno run --lock=lock.json --cached-only mod.ts
```

This will fail if there are any dependencies in the dependency tree for mod.ts which are not yet cached.

Proxies

Deno supports proxies for module downloads and the Web standard `fetch` API.

Proxy configuration is read from environmental variables: `HTTP_PROXY` and `HTTPS_PROXY`.

In case of Windows, if environment variables are not found Deno falls back to reading proxies from registry.

Import maps

> This is an unstable feature. Learn more about unstable features.

Deno supports import maps.

You can use import maps with the `--importmap=<FILE>` CLI flag.

Current limitations:

- single import map
- no fallback URLs
- Deno does not support `std:` namespace
- supports only `file:`, `http:` and `https:` schemes

Example:

import_map.json

```
{
   "imports": {
      "fmt/": "https://deno.land/std@0.55.0/fmt/"
   }
}
```

color.ts

```
import { red } from "fmt/colors.ts";

console.log(red("hello world"));
```

Then:

```
$ deno run --importmap=import_map.json --unstable color.
  ↪ ts
```

Standard library

Deno provides a set of standard modules that are audited by the core team and are guaranteed to work with Deno.

Standard library is available at: https://deno.land/std/

Versioning and stability

Standard library is not yet stable and therefore it is versioned differently than Deno. For latest release consult https://deno.land/std/ or https://deno.land/std/version.ts.

We strongly suggest to always use imports with pinned version of standard library to avoid unintended changes.

Troubleshooting

Some of the modules provided in standard library use unstable Deno APIs.

Trying to run such modules without `--unstable` CLI flag ends up with a lot of TypeScript errors suggesting that some APIs on `Deno` namespace do not exist:

```
// main.ts
import { copy } from "https://deno.land/std@0.50.0/fs/
  ↪ copy.ts";

copy("log.txt", "log-old.txt");
```

```
$ deno run --allow-read --allow-write main.ts
Compile file:///dev/deno/main.ts
Download https://deno.land/std@0.50.0/fs/copy.ts
Download https://deno.land/std@0.50.0/fs/ensure_dir.ts
Download https://deno.land/std@0.50.0/fs/_util.ts
error: TS2339 [ERROR]: Property 'utime' does not exist
  ↪ on type 'typeof Deno'.
    await Deno.utime(dest, statInfo.atime, statInfo.
      ↪ mtime);
                ~~~~~
```

```
    at https://deno.land/std@0.50.0/fs/copy.ts:90:16

TS2339 [ERROR]: Property 'utimeSync' does not exist on
  ↪ type 'typeof Deno'.
    Deno.utimeSync(dest, statInfo.atime, statInfo.mtime)
      ↪ ;
          ~~~~~~~~~
    at https://deno.land/std@0.50.0/fs/copy.ts:101:10
```

Solution to that problem requires adding `--unstable` flag:

```
deno run --allow-read --allow-write --unstable main.ts
```

To make sure that API producing error is unstable check `lib.deno.` ↪ `unstable.d.ts` declaration.

This problem should be fixed in the near future.

Testing

Deno has a built-in test runner that you can use for testing JavaScript or TypeScript code.

Writing tests

To define a test you need to call `Deno.test` with a name and function to be tested:

```
Deno.test("hello world", () => {
  const x = 1 + 2;
  if (x !== 3) {
    throw Error("x should be equal to 3");
  }
```

```
});
```

There are some useful assertion utilities at https://deno.land/std/testing to make testing easier:

```
import { assertEquals } from "https://deno.land/std/
  ↪ testing/asserts.ts";

Deno.test("hello world", () => {
  const x = 1 + 2;
  assertEquals(x, 3);
});
```

Async functions

You can also test asynchronous code by passing a test function that returns a promise. For this you can use the `async` keyword when defining a function:

```
import { delay } from "https://deno.land/std/async/delay
  ↪ .ts";

Deno.test("async hello world", async () => {
  const x = 1 + 2;

  // await some async task
  await delay(100);

  if (x !== 3) {
    throw Error("x should be equal to 3");
  }
});
```

Resource and async op sanitizers

Certain actions in Deno create resources in the resource table (learn more here). These resources should be closed after you are done using them.

For each test definition the test runner checks that all resources created in this test have been closed. This is to prevent resource 'leaks'. This is enabled by default for all tests, but can be disabled by setting the `sanitizeResources` boolean to false in the test definition.

The same is true for async operation like interacting with the filesystem. The test runner checks that each operation you start in the test is completed before the end of the test. This is enabled by default for all tests, but can be disabled by setting the `sanitizeOps` boolean to false in the test definition.

```
Deno.test({
  name: "leaky test",
  fn() {
    Deno.open("hello.txt");
  },
  sanitizeResources: false,
  sanitizeOps: false,
});
```

Ignoring tests

Sometimes you want to ignore tests based on some sort of condition (for example you only want a test to run on Windows). For this you can use the `ignore` boolean in the test definition. If it is set to true the test will be skipped.

```
Deno.test({
```

```
  name: "do macOS feature",
  ignore: Deno.build.os !== "darwin",
  fn() {
    doMacOSFeature();
  },
});
```

Running tests

To run the test, call `deno test` with the file that contains your test function:

```
deno test my_test.ts
```

You can also omit the file name, in which case all tests in the current directory (recursively) that match the glob `{*_,*.,}test.{js,mjs,ts,` ↪ `jsx,tsx}` will be run. If you pass a directory, all files in the directory that match this glob will be run.

Built-in tooling

Deno provides some built in tooling that is useful when working with JavaScript and TypeScript:

- 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`)

Debugger

Deno supports the V8 Inspector Protocol.

It's possible to debug Deno programs using Chrome Devtools or other clients that support the protocol (eg. VSCode).

To activate debugging capabilities run Deno with the `--inspect` or `--inspect-brk` flags.

The `--inspect` flag allows attaching the debugger at any point in time, while `--inspect-brk` will wait for the debugger to attach and will pause execution on the first line of code.

Chrome Devtools

Let's try debugging a program using Chrome Devtools. For this, we'll use file_server.ts from `std`, a static file server.

Use the `--inspect-brk` flag to break execution on the first line:

```
$ deno run --inspect-brk --allow-read --allow-net https
  ↪ ://deno.land/std@v0.50.0/http/file_server.ts
Debugger listening on ws://127.0.0.1:9229/ws/1e82c406-85
  ↪ a9-44ab-86b6-7341583480b1
Download https://deno.land/std@v0.50.0/http/file_server.
  ↪ ts
Compile https://deno.land/std@v0.50.0/http/file_server.
  ↪ ts
...
```

Open `chrome://inspect` and click `Inspect` next to target:

It might take a few seconds after opening the devtools to load all modules.

Figure 1: chrome://inspect

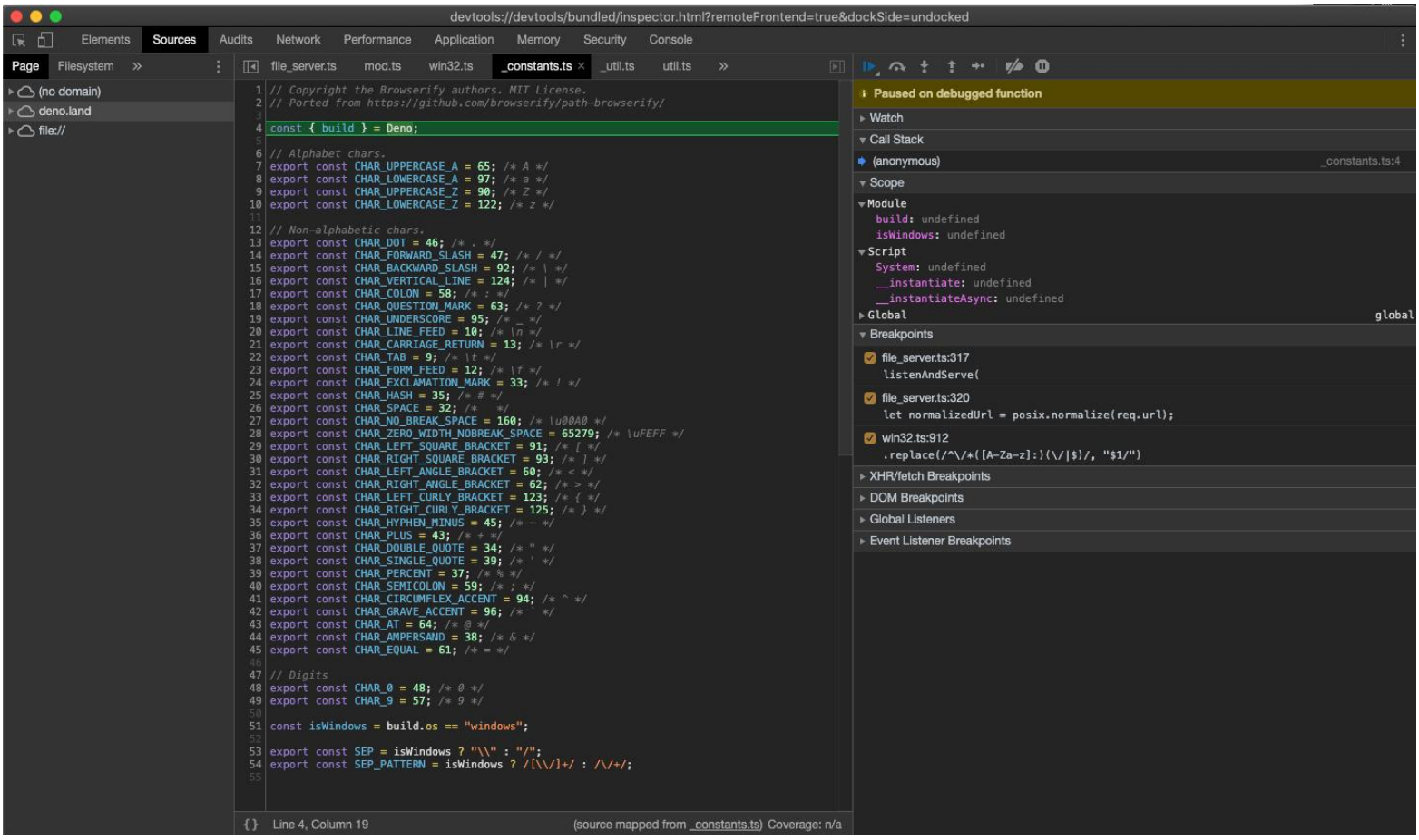

Figure 2: Devtools opened

You might notice that Devtools paused execution on the first line of `_constants.ts` instead of `file_server.ts`. This is expected behavior and is caused by the way ES modules are evaluated by V8 (`_constants` ↪ `.ts` is left-most, bottom-most dependency of `file_server.ts` so it is evaluated first).

At this point all source code is available in the Devtools, so let's open up `file_server.ts` and add a breakpoint there; go to "Sources" pane and expand the tree:

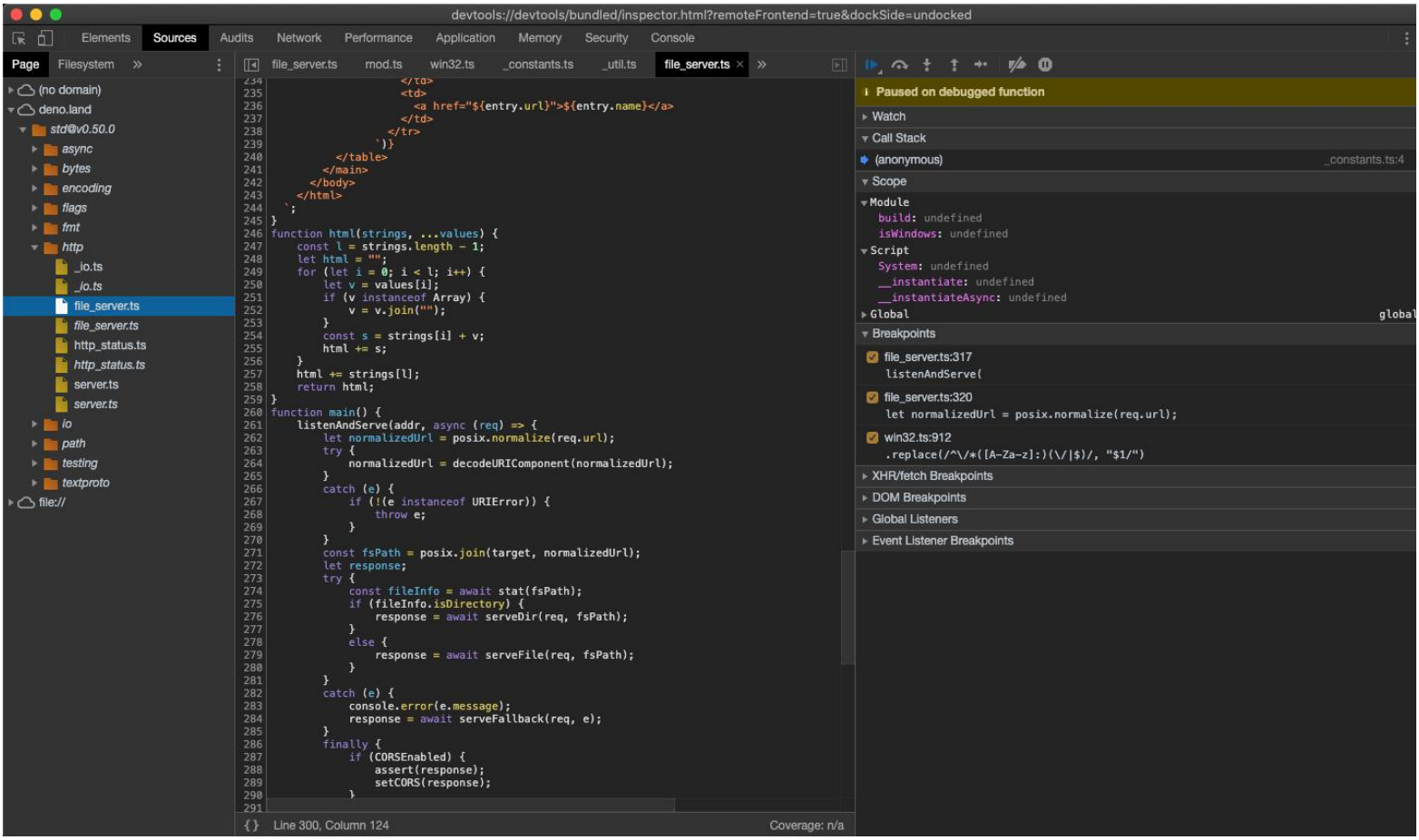

Figure 3: Open file_server.ts

Looking closely you'll find duplicate entries for each file; one written regularly and one in italics. The former is compiled source file (so in the case of `.ts` files it will be emitted JavaScript source), while the latter is a source map for the file.

Next, add a breakpoint in the `listenAndServe` method:

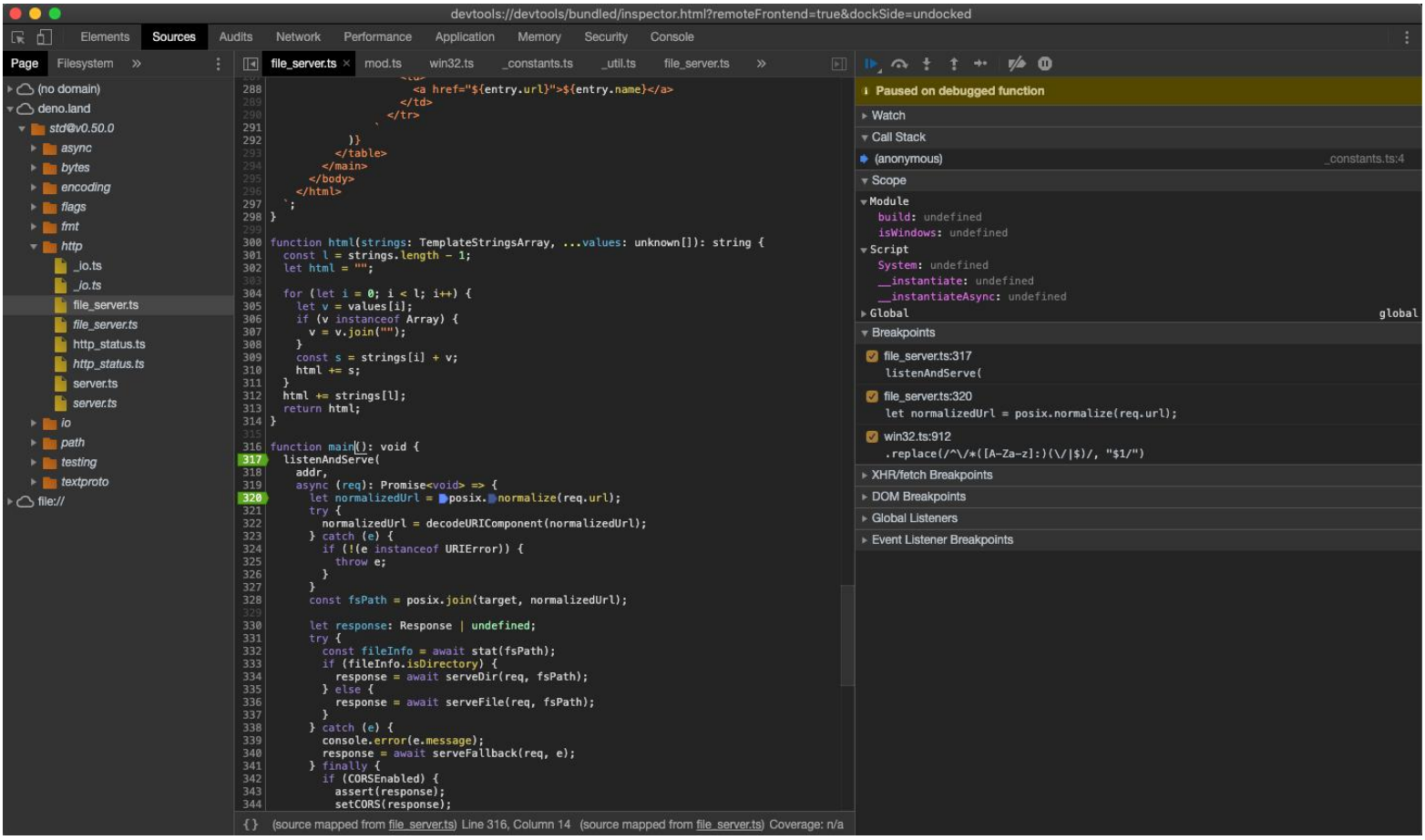

Figure 4: Break in file_server.ts

As soon as we've added the breakpoint Devtools automatically opened up the source map file, which allows us step through the actual source code that includes types.

Now that we have our breakpoints set, we can resume the execution of our script so that we might inspect an incoming request. Hit the Resume script execution button to do so. You might even need to hit it twice!

Once our script is running again, let's send a request and inspect it in Devtools:

```
$ curl http://0.0.0.0:4500/
```

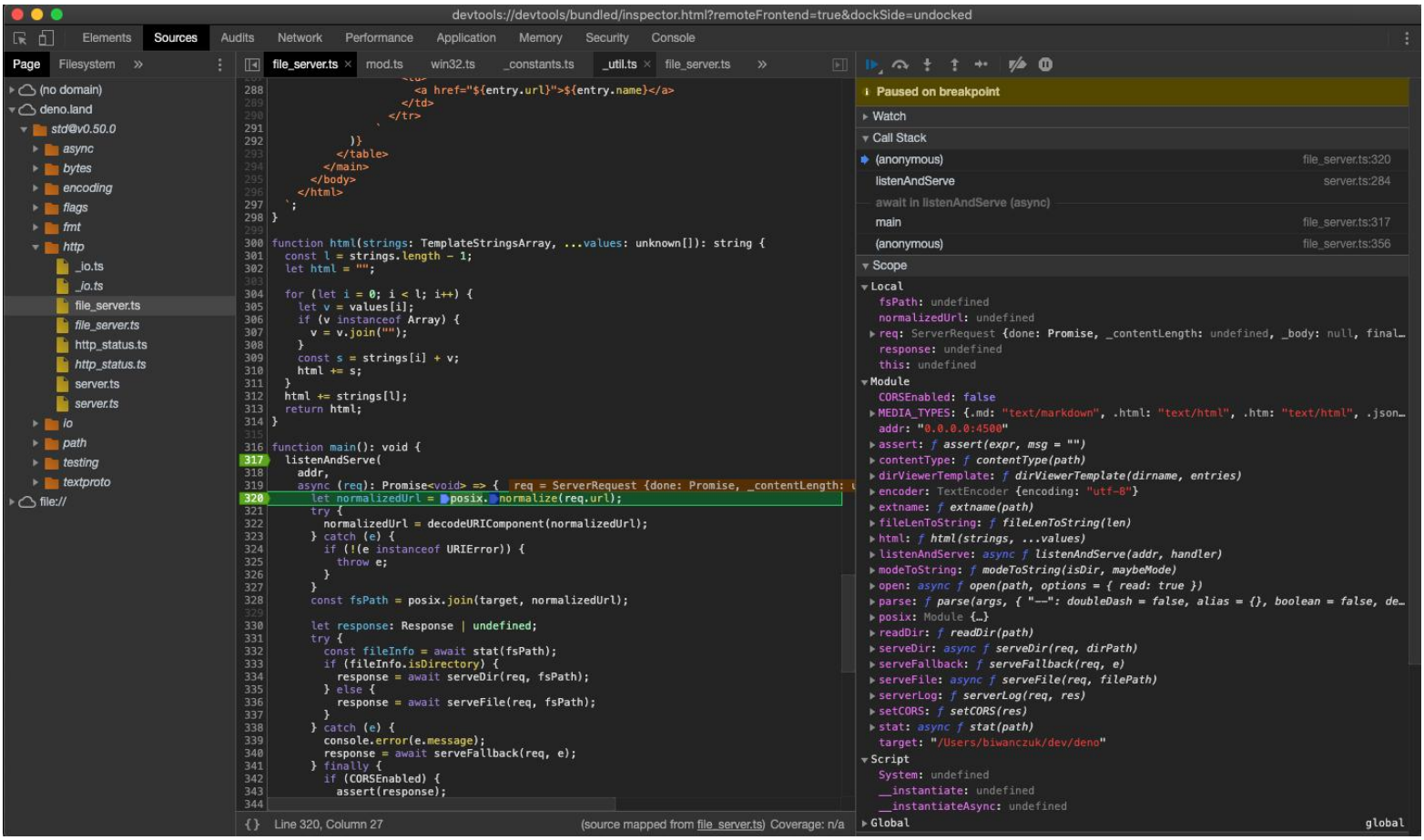

Figure 5: Break in request handling

At this point we can introspect the contents of the request and go step-by-step to debug the code.

VSCode

Deno can be debugged using VSCode.

Official support via the plugin is being worked on - https://github.com/de

We can still attach the debugger by manually providing a `launch.json` config:

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "Deno",
      "type": "node",
      "request": "launch",
      "cwd": "${workspaceFolder}",
      "runtimeExecutable": "deno",
      "runtimeArgs": ["run", "--inspect-brk", "-A", "${
        ↪ file}"],
      "port": 9229
    }
  ]
}
```

NOTE: This uses the file you have open as the entry point; replace `${file}` with a script name if you want a fixed entry point.

Let's try out debugging a local source file. Create `server.ts`:

```
import { serve } from "https://deno.land/std@v0.50.0/
  ↪ 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" });
}
```

Then we can set a breakpoint, and run the created configuration:

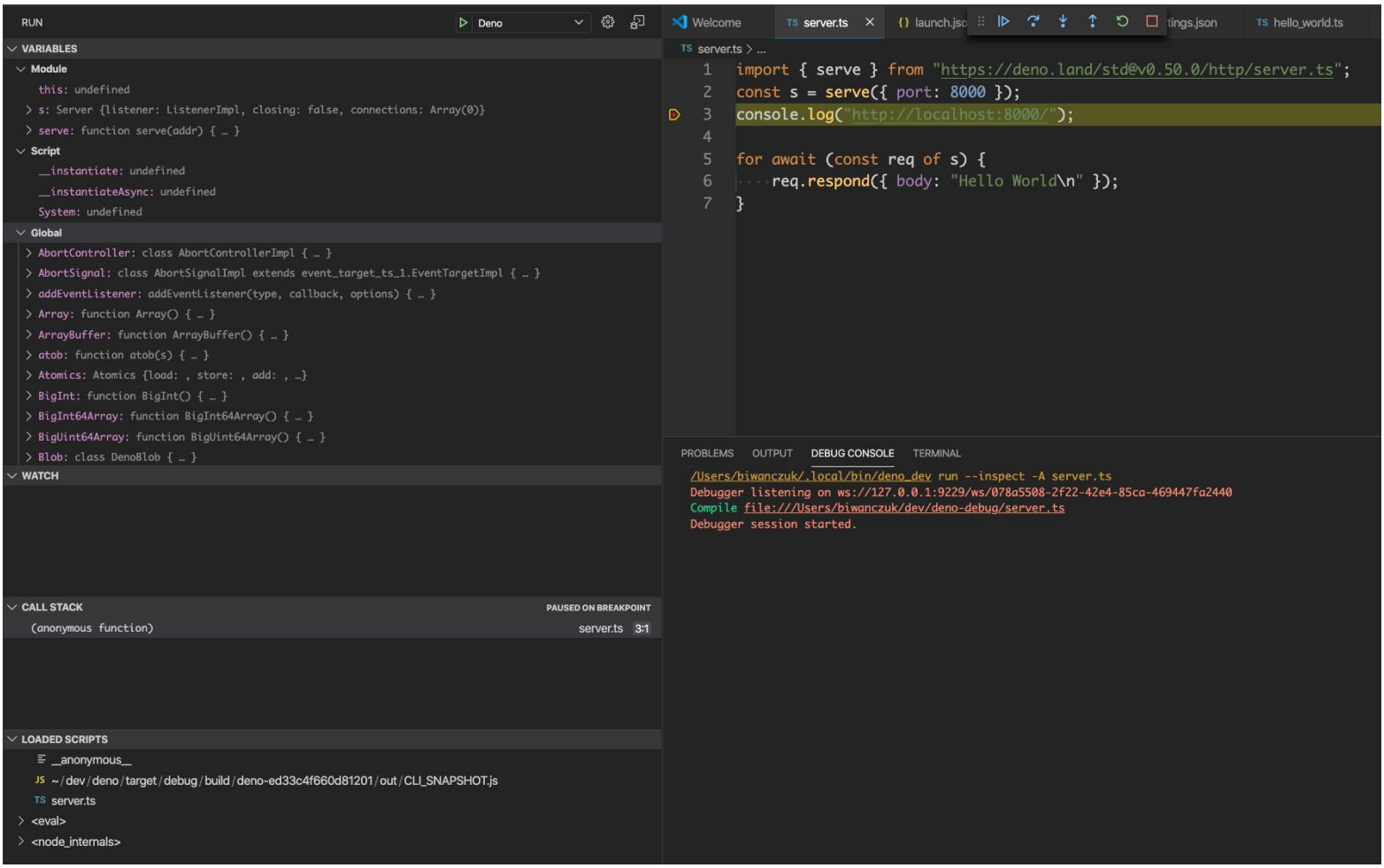

Figure 6: VSCode debugger

JetBrains IDEs

You can debug Deno using your JetBrains IDE by right-clicking the file you want to debug and selecting the `Debug 'Deno: <file name>'` option. This will create a run/debug configuration with no permission flags set.

To configure these flags edit the run/debug configuration and modify the `Arguments` field with the required flags.

Other

Any client that implements the Devtools protocol should be able to connect to a Deno process.

Limitations

Devtools support is still immature. There is some functionality that is known to be missing or buggy:

- autocomplete in Devtools' console causes the Deno process to exit
- profiling and memory dumps might not work correctly

Script installer

Deno provides `deno install` to easily install and distribute executable code.

`deno install [OPTIONS...] [URL] [SCRIPT_ARGS...]` will install the script available at `URL` under the name `EXE_NAME`.

This command creates a thin, executable shell script which invokes `deno` ↪ using the specified CLI flags and main module. It is placed in the installation root's `bin` directory.

Example:

```
$ deno install --allow-net --allow-read https://deno.
  ↪ land/std/http/file_server.ts
```

```
[1/1] Compiling https://deno.land/std/http/file_server.
  ↪ ts

 Successfully installed file_server.
/Users/deno/.deno/bin/file_server
```

To change the executable name, use `-n`/`--name`:

```
deno install --allow-net --allow-read -n serve https://
  ↪ deno.land/std/http/file_server.ts
```

The executable name is inferred by default:

- Attempt to take the file stem of the URL path. The above example would become 'file_server'.
- If the file stem is something generic like 'main', 'mod', 'index' or 'cli', and the path has no parent, take the file name of the parent path. Otherwise settle with the generic name.

To change the installation root, use `--root`:

```
deno install --allow-net --allow-read --root /usr/local
  ↪ https://deno.land/std/http/file_server.ts
```

The installation root is determined, in order of precedence:

- `--root` option
- `DENO_INSTALL_ROOT` environment variable
- `$HOME/.deno`

These must be added to the path manually if required.

```
echo 'export PATH="$HOME/.deno/bin:$PATH"' >> ~/.bashrc
```

You must specify permissions that will be used to run the script at installation time.

```
deno install --allow-net --allow-read https://deno.land/
  ↪ std/http/file_server.ts -p 8080
```

The above command creates an executable called `file_server` that runs with network and read permissions and binds to port 8080.

For good practice, use the `import.meta.main` idiom to specify the entry point in an executable script.

Example:

```
// https://example.com/awesome/cli.ts
async function myAwesomeCli(): Promise<void> {
  -- snip --
}

if (import.meta.main) {
  myAwesomeCli();
}
```

When you create an executable script make sure to let users know by adding an example installation command to your repository:

```
# Install using deno install

$ deno install -n awesome_cli https://example.com/
  ↪ awesome/cli.ts
```

Code formatter

Deno ships with a built in code formatter that auto-formats TypeScript and JavaScript code.

```
# format all JS/TS files in the current directory and
  ↪ subdirectories
```

```
deno fmt
# format specific files
deno fmt myfile1.ts myfile2.ts
# check if all the JS/TS files in the current directory
  ↪ and subdirectories are formatted
deno fmt --check
# format stdin and write to stdout
cat file.ts | deno fmt -
```

Ignore formatting code by preceding it with a `// deno-fmt-ignore` comment:

```
// deno-fmt-ignore
export const identity = [
    1, 0, 0,
    0, 1, 0,
    0, 0, 1,
];
```

Or ignore an entire file by adding a `// deno-fmt-ignore-file` comment at the top of the file.

Bundling

`deno bundle [URL]` will output a single JavaScript file, which includes all dependencies of the specified input. For example:

```
> deno bundle https://deno.land/std/examples/colors.ts
  ↪ colors.bundle.js
Bundling "colors.bundle.js"
Emitting bundle to "colors.bundle.js"
9.2 kB emitted.
```

If you omit the out file, the bundle will be sent to `stdout`.

The bundle can just be run as any other module in Deno would:

```
deno run colors.bundle.js
```

The output is a self contained ES Module, where any exports from the main module supplied on the command line will be available. For example, if the main module looked something like this:

```
export { foo } from "./foo.js";

export const bar = "bar";
```

It could be imported like this:

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

Bundles can also be loaded in the web browser. The bundle is a self-contained ES module, and so the attribute of `type` must be set to `"module"`. For example:

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

Or you could import it into another ES module to consume:

```
<script type="module">
  import * as website from "website.bundle.js";
</script>
```

Documentation Generator

`deno doc` followed by a list of one or more source files will print the JSDoc documentation for each of the module's **exported** members. Currently, only exports in the style `export <declaration>` and `export ... from ...` are are supported.

For example, given a file `add.ts` with the contents:

```
/**
 * Adds x and y.
 * @param {number} x
 * @param {number} y
 * @returns {number} Sum of x and y
 */
export function add(x: number, y: number): number {
  return x + y;
}
```

Running the Deno `doc` command, prints the function's JSDoc comment to `stdout`:

```
deno doc add.ts
function add(x: number, y: number): number
  Adds x and y. @param {number} x @param {number} y
    ↪ @returns {number} Sum of x and y
```

Use the `--json` flag to output the documentation in JSON format. This JSON format is consumed by the deno doc website and used to generate module documentation.

Dependency Inspector

`deno info [URL]` will inspect ES module and all of its dependencies.

```
deno info https://deno.land/std@0.52.0/http/file_server.
  ↪ ts
Download https://deno.land/std@0.52.0/http/file_server.
  ↪ ts
...
local: /Users/deno/Library/Caches/deno/deps/https/deno.
  ↪ land/5
  ↪ bd138988e9d20db1a436666628ffb3f7586934e0a2a9fe2a7b7bf4
```

```
  ↪
type: TypeScript
compiled: /Users/deno/Library/Caches/deno/gen/https/deno
  ↪ .land/std@0.52.0/http/file_server.ts.js
map: /Users/deno/Library/Caches/deno/gen/https/deno.land
  ↪ /std@0.52.0/http/file_server.ts.js.map
deps:
https://deno.land/std@0.52.0/http/file_server.ts
   https://deno.land/std@0.52.0/path/mod.ts
    https://deno.land/std@0.52.0/path/win32.ts
     https://deno.land/std@0.52.0/path/_constants.ts
     https://deno.land/std@0.52.0/path/_util.ts
      https://deno.land/std@0.52.0/path/_constants.ts
     https://deno.land/std@0.52.0/testing/asserts.ts
       https://deno.land/std@0.52.0/fmt/colors.ts
       https://deno.land/std@0.52.0/testing/diff.ts
    https://deno.land/std@0.52.0/path/posix.ts
     https://deno.land/std@0.52.0/path/_constants.ts
     https://deno.land/std@0.52.0/path/_util.ts
    https://deno.land/std@0.52.0/path/common.ts
     https://deno.land/std@0.52.0/path/separator.ts
    https://deno.land/std@0.52.0/path/separator.ts
    https://deno.land/std@0.52.0/path/interface.ts
    https://deno.land/std@0.52.0/path/glob.ts
      https://deno.land/std@0.52.0/path/separator.ts
      https://deno.land/std@0.52.0/path/_globrex.ts
      https://deno.land/std@0.52.0/path/mod.ts
      https://deno.land/std@0.52.0/testing/asserts.ts
   https://deno.land/std@0.52.0/http/server.ts
    https://deno.land/std@0.52.0/encoding/utf8.ts
    https://deno.land/std@0.52.0/io/bufio.ts
     https://deno.land/std@0.52.0/io/util.ts
      https://deno.land/std@0.52.0/path/mod.ts
```

```
    https://deno.land/std@0.52.0/encoding/utf8.ts
   https://deno.land/std@0.52.0/testing/asserts.ts
  https://deno.land/std@0.52.0/testing/asserts.ts
  https://deno.land/std@0.52.0/async/mod.ts
   https://deno.land/std@0.52.0/async/deferred.ts
   https://deno.land/std@0.52.0/async/delay.ts
   https://deno.land/std@0.52.0/async/
     ↪ mux_async_iterator.ts
     https://deno.land/std@0.52.0/async/deferred.ts
  https://deno.land/std@0.52.0/http/_io.ts
    https://deno.land/std@0.52.0/io/bufio.ts
    https://deno.land/std@0.52.0/textproto/mod.ts
     https://deno.land/std@0.52.0/io/util.ts
     https://deno.land/std@0.52.0/bytes/mod.ts
      https://deno.land/std@0.52.0/io/util.ts
     https://deno.land/std@0.52.0/encoding/utf8.ts
    https://deno.land/std@0.52.0/testing/asserts.ts
    https://deno.land/std@0.52.0/encoding/utf8.ts
    https://deno.land/std@0.52.0/http/server.ts
    https://deno.land/std@0.52.0/http/http_status.ts
 https://deno.land/std@0.52.0/flags/mod.ts
  https://deno.land/std@0.52.0/testing/asserts.ts
 https://deno.land/std@0.52.0/testing/asserts.ts
```

Dependency inspector works with any local or remote ES modules.

Cache location

`deno info` can be used to display information about cache location:

```
deno info
DENO_DIR location: "/Users/deno/Library/Caches/deno"
Remote modules cache: "/Users/deno/Library/Caches/deno/
  ↪ deps"
```

```
TypeScript compiler cache: "/Users/deno/Library/Caches/
  ↪ deno/gen"
```

Linter

Deno ships with a built in code linter for JavaScript and TypeScript.

Note: linter is a new feature and still unstable thus it requires `--unstable` flag

```
# lint all JS/TS files in the current directory and
  ↪ subdirectories
deno lint --unstable
# lint specific files
deno lint --unstable myfile1.ts myfile2.ts
```

Available rules

- ban-ts-comment
- ban-untagged-ignore
- constructor-super
- for-direction
- getter-return
- no-array-constructor
- no-async-promise-executor
- no-case-declarations
- no-class-assign
- no-compare-neg-zero
- no-cond-assign
- no-debugger
- no-delete-var

- no-dupe-args
- no-dupe-keys
- no-duplicate-case
- no-empty-character-class
- no-empty-interface
- no-empty-pattern
- no-empty
- no-ex-assign
- no-explicit-any
- no-func-assign
- no-misused-new
- no-namespace
- no-new-symbol
- no-obj-call
- no-octal
- no-prototype-builtins
- no-regex-spaces
- no-setter-return
- no-this-alias
- no-this-before-super
- no-unsafe-finally
- no-unsafe-negation
- no-with
- prefer-as-const
- prefer-namespace-keyword
- require-yield
- triple-slash-reference
- use-isnan
- valid-typeof

Ignore directives

Files To ignore whole file `// deno-lint-ignore-file` directive should placed at the top of the file.

```
// deno-lint-ignore-file

function foo(): any {
  // ...
}
```

Ignore directive must be placed before first stament or declaration:

```
// Copyright 2020 the Deno authors. All rights reserved.
  ↪  MIT license.

/**
 * Some JS doc
 **/

// deno-lint-ignore-file

import { bar } from "./bar.js";

function foo(): any {
  // ...
}
```

Diagnostics To ignore certain diagnostic `// deno-lint-ignore ↪ <codes...>` directive should be placed before offending line. Specifying ignored rule name is required.

```
// deno-lint-ignore no-explicit-any
function foo(): any {
  // ...
```

```
}

// deno-lint-ignore no-explicit-any explicit-function-
  ↪ return-type
function bar(a: any) {
  // ...
}
```

To provide some compatibility with ESLint `deno lint` also supports `// ↪ eslint-ignore-next-line` directive. Just like in `// deno-lint-ignore` it's required to specify ignored rule name is required.

```
// eslint-ignore-next-line no-empty
while (true) {}

// eslint-ignore-next-line @typescript-eslint/no-
  ↪ explicit-any
function bar(a: any) {
  // ...
}
```

Embedding Deno

Deno consists of multiple parts, one of which is `deno_core`. This is a rust crate that can be used to embed a JavaScript runtime into your rust application. Deno is built on top of `deno_core`.

The Deno crate is hosted on crates.io.

You can view the API on docs.rs.

Contributing

- Read the style guide.
- Please don't make the benchmarks worse.
- Ask for help in the community chat room.
- If you are going to work on an issue, mention so in the issue comments *before* you start working on the issue.
- Please be professional in the forums. We follow Rust's code of conduct (CoC) Have a problem? Email ry@tinyclouds.org.

Development

Instructions on how to build from source can be found here.

Submitting a Pull Request

Before submitting, please make sure the following is done:

1. That there is a related issue and it is referenced in the PR text.
2. There are tests that cover the changes.
3. Ensure `cargo test` passes.
4. Format your code with `tools/format.py`
5. Make sure `./tools/lint.py` passes.

Changes to `third_party`

`deno_third_party` contains most of the external code that Deno depends on, so that we know exactly what we are executing at any given time.

It is carefully maintained with a mixture of manual labor and private scripts. It's likely you will need help from @ry or @piscisaureus to make changes.

Adding Ops (aka bindings)

We are very concerned about making mistakes when adding new APIs. When adding an Op to Deno, the counterpart interfaces on other platforms should be researched. Please list how this functionality is done in Go, Node, Rust, and Python.

As an example, see how `Deno.rename()` was proposed and added in PR #671.

Releases

Summary of the changes from previous releases can be found here.

Documenting APIs

It is important to document public APIs and we want to do that inline with the code. This helps ensure that code and documentation are tightly coupled together.

Utilize JSDoc

All publicly exposed APIs and types, both via the `deno` module as well as the global/`window` namespace should have JSDoc documentation. This documentation is parsed and available to the TypeScript compiler, and therefore easy to provide further downstream. JSDoc blocks come just

prior to the statement they apply to and are denoted by a leading `/**` before terminating with a `*/`. For example:

```
/** A simple JSDoc comment */
export const FOO = "foo";
```

Find more at https://jsdoc.app/

Building from source

Below are instructions on how to build Deno from source. If you just want to use Deno you can download a prebuilt executable (more information in the `Getting Started` chapter).

Cloning the Repository

Clone on Linux or Mac:

```
git clone --recurse-submodules https://github.com/
  ↪ denoland/deno.git
```

Extra steps for Windows users:

1. Enable "Developer Mode" (otherwise symlinks would require administrator privileges).
2. Make sure you are using git version 2.19.2.windows.1 or newer.
3. Set `core.symlinks=true` before the checkout:

   ```
   git config --global core.symlinks true
   git clone --recurse-submodules https://github.com/
     ↪ denoland/deno.git
   ```

Prerequisites

You will need to install Rust. Make sure to fetch the latest stable release as Deno does not support nightly builds. Check that you have the required tools:

```
rustc -V
cargo -V
```

The easiest way to build Deno is by using a precompiled version of V8:

```
cargo build -vv
```

However if you want to build Deno and V8 from source code:

```
V8_FROM_SOURCE=1 cargo build -vv
```

When building V8 from source, there are more dependencies:

Python 2. Ensure that a suffix-less `python/python.exe` exists in your `PATH` and it refers to Python 2, not 3.

For Linux users glib-2.0 development files must also be installed. (On Ubuntu, run `apt install libglib2.0-dev`.)

Mac users must have XCode installed.

For Windows users:

1. Get VS Community 2019 with "Desktop development with C++" toolkit and make sure to select the following required tools listed below along with all C++ tools.
    - Visual C++ tools for CMake
    - Windows 10 SDK (10.0.17763.0)
    - Testing tools core features - Build Tools
    - Visual C++ ATL for x86 and x64

- Visual C++ MFC for x86 and x64
- C++/CLI support
- VC++ 2015.3 v14.00 (v140) toolset for desktop

2. Enable "Debugging Tools for Windows". Go to "Control Panel" → "Programs" → "Programs and Features" → Select "Windows Software Development Kit - Windows 10" → "Change" → "Change" → Check "Debugging Tools For Windows" → "Change" -> "Finish". Or use: Debugging Tools for Windows (Notice: it will download the files, you should install `X64 Debuggers And Tools-x64_en-us.msi` file manually.)

See rusty_v8's README for more details about the V8 build.

Building

Build with Cargo:

```
# Build:
cargo build -vv

# Build errors?  Ensure you have latest master and try
  ↪ building again, or if that doesn't work try:
cargo clean && cargo build -vv

# Run:
./target/debug/deno run cli/tests/002_hello.ts
```

Testing and Tools

Tests

Test `deno`:

```
# Run the whole suite:
cargo test

# Only test cli/js/:
cargo test js_unit_tests
```

Test `std/`:

```
cargo test std_tests
```

Lint and format

Lint the code:

```
./tools/lint.py
```

Format the code:

```
./tools/format.py
```

Profiling

To start profiling,

```
# Make sure we're only building release.
# Build deno and V8's d8.
ninja -C target/release d8

# Start the program we want to benchmark with --prof
./target/release/deno run tests/http_bench.ts --allow-
  ↪ net --v8-flags=--prof &

# Exercise it.
third_party/wrk/linux/wrk http://localhost:4500/
kill `pgrep deno`
```

V8 will write a file in the current directory that looks like this: `isolate-0x7fad98242400-v8.log`. To examine this file:

```
D8_PATH=target/release/ ./third_party/v8/tools/linux-tick-processor
isolate-0x7fad98242400-v8.log > prof.log
# on macOS, use ./third_party/v8/tools/mac-tick-processor instead
```

`prof.log` will contain information about tick distribution of different calls.

To view the log with Web UI, generate JSON file of the log:

```
D8_PATH=target/release/ ./third_party/v8/tools/linux-tick-processor
isolate-0x7fad98242400-v8.log --preprocess > prof.json
```

Open `third_party/v8/tools/profview/index.html` in your browser, and select `prof.json` to view the distribution graphically.

Useful V8 flags during profiling:

- –prof
- –log-internal-timer-events
- –log-timer-events
- –track-gc
- –log-source-code
- –track-gc-object-stats

To learn more about `d8` and profiling, check out the following links:

- https://v8.dev/docs/d8
- https://v8.dev/docs/profile

Debugging with LLDB

We can use LLDB to debug Deno.

```
$ lldb -- target/debug/deno run tests/worker.js
> run
> bt
> up
> up
> l
```

To debug Rust code, we can use `rust-lldb`. It should come with `rustc` and is a wrapper around LLDB.

```
$ rust-lldb -- ./target/debug/deno run --allow-net tests
  ↪ /http_bench.ts
# On macOS, you might get warnings like
# `ImportError: cannot import name _remove_dead_weakref`
# In that case, use system python by setting PATH, e.g.
# PATH=/System/Library/Frameworks/Python.framework/
  ↪ Versions/2.7/bin:$PATH
(lldb) command script import "/Users/kevinqian/.rustup/
  ↪ toolchains/1.36.0-x86_64-apple-darwin/lib/rustlib/
  ↪ etc/lldb_rust_formatters.py"
(lldb) type summary add --no-value --python-function
  ↪ lldb_rust_formatters.print_val -x ".*" --category
  ↪ Rust
(lldb) type category enable Rust
(lldb) target create "../deno/target/debug/deno"
Current executable set to '../deno/target/debug/deno' (
  ↪ x86_64).
(lldb) settings set -- target.run-args  "tests/
  ↪ http_bench.ts" "--allow-net"
(lldb) b op_start
(lldb) r
```

V8 flags

V8 has many many internal command-line flags.

```
# list available v8 flags
$ deno --v8-flags=--help

#  example for applying multiple flags
$ deno --v8-flags=--expose-gc,--use-strict
```

Particularly useful ones:

```
--async-stack-trace
```

Continuous Benchmarks

See our benchmarks over here

The benchmark chart supposes https://github.com/denoland/benchmark_data/gh-pages/data.json has the type `BenchmarkData[]` where `BenchmarkData` is defined like the below:

```
interface ExecTimeData {
  mean: number;
  stddev: number;
  user: number;
  system: number;
  min: number;
  max: number;
}

interface BenchmarkData {
  created_at: string;
  sha1: string;
  benchmark: {
```

```
    [key: string]: ExecTimeData;
  };
  binarySizeData: {
    [key: string]: number;
  };
  threadCountData: {
    [key: string]: number;
  };
  syscallCountData: {
    [key: string]: number;
  };
}
```

Deno Style Guide

Table of Contents

Copyright Headers

Most modules in the repository should have the following copyright header:

```
// Copyright 2018-2020 the Deno authors. All rights
  ↪ reserved. MIT license.
```

If the code originates elsewhere, ensure that the file has the proper copyright headers. We only allow MIT, BSD, and Apache licensed code.

Use underscores, not dashes in filenames.

Example: Use `file_server.ts` instead of `file-server.ts`.

Add tests for new features.

Each module should contain or be accompanied by tests for its public functionality.

TODO Comments

TODO comments should usually include an issue or the author's github username in parentheses. Example:

```
// TODO(ry): Add tests.
// TODO(#123): Support Windows.
// FIXME(#349): Sometimes panics.
```

Meta-programming is discouraged. Including the use of Proxy.

Be explicit even when it means more code.

There are some situations where it may make sense to use such techniques, but in the vast majority of cases it does not.

Rust

Follow Rust conventions and be consistent with existing code.

Typescript

The TypeScript portions of the codebase include `cli/js` for the built-ins and the standard library `std`.

Use TypeScript instead of JavaScript.

Use the term "module" instead of "library" or "package".

For clarity and consistency avoid the terms "library" and "package". Instead use "module" to refer to a single JS or TS file and also to refer to a directory of TS/JS code.

Do not use the filename `index.ts/index.js`.

Deno does not treat "index.js" or "index.ts" in a special way. By using these filenames, it suggests that they can be left out of the module specifier when they cannot. This is confusing.

If a directory of code needs a default entry point, use the filename `mod` ↪ `.ts`. The filename `mod.ts` follows Rust's convention, is shorter than `index.ts`, and doesn't come with any preconceived notions about how it might work.

Exported functions: max 2 args, put the rest into an options object.

When designing function interfaces, stick to the following rules.

1. A function that is part of the public API takes 0-2 required arguments, plus (if necessary) an options object (so max 3 total).
2. Optional parameters should generally go into the options object.

   An optional parameter that's not in an options object might be acceptable if there is only one, and it seems inconceivable that we would add more optional parameters in the future.

3. The ‘options’ argument is the only argument that is a regular ‘Object’.

   Other arguments can be objects, but they must be distinguishable from a ‘plain’ Object runtime, by having either:

   - a distinguishing prototype (e.g. `Array`, `Map`, `Date`, `class MyThing`)
   - a well-known symbol property (e.g. an iterable with `Symbol.iterator`).

   This allows the API to evolve in a backwards compatible way, even when the position of the options object changes.

```
// BAD: optional parameters not part of options object. (#2)
export function resolve(
  hostname: string,
  family?: "ipv4" | "ipv6",
  timeout?: number
): IPAddress[] {}

// GOOD.
export interface ResolveOptions {
  family?: "ipv4" | "ipv6";
  timeout?: number;
}
export function resolve(
  hostname: string,
  options: ResolveOptions = {}
): IPAddress[] {}
```

```
export interface Environment {
  [key: string]: string;
}
```

```
// BAD: `env` could be a regular Object and is therefore
  ↪  indistinguishable
// from an options object. (#3)
export function runShellWithEnv(cmdline: string, env:
  ↪ Environment): string {}

// GOOD.
export interface RunShellOptions {
  env: Environment;
}
export function runShellWithEnv(
  cmdline: string,
  options: RunShellOptions
): string {}
```

```
// BAD: more than 3 arguments (#1), multiple optional
  ↪ parameters (#2).
export function renameSync(
  oldname: string,
  newname: string,
  replaceExisting?: boolean,
  followLinks?: boolean
) {}

// GOOD.
interface RenameOptions {
  replaceExisting?: boolean;
  followLinks?: boolean;
}
export function renameSync(
  oldname: string,
  newname: string,
  options: RenameOptions = {}
```

```
) {}
```

```
// BAD: too many arguments. (#1)
export function pwrite(
  fd: number,
  buffer: TypedArray,
  offset: number,
  length: number,
  position: number
) {}

// BETTER.
export interface PWrite {
  fd: number;
  buffer: TypedArray;
  offset: number;
  length: number;
  position: number;
}
export function pwrite(options: PWrite) {}
```

Minimize dependencies; do not make circular imports.

Although `cli/js` and `std` have no external dependencies, we must still be careful to keep internal dependencies simple and manageable. In particular, be careful not to introduce circular imports.

If a filename starts with an underscore: `_foo.ts`, do not link to it.

Sometimes there may be situations where an internal module is necessary but its API is not meant to be stable or linked to. In this case

prefix it with an underscore. By convention, only files in its own directory should import it.

Use JSDoc for exported symbols.

We strive for complete documentation. Every exported symbol ideally should have a documentation line.

If possible, use a single line for the JS Doc. Example:

```
/** foo does bar. */
export function foo() {
  // ...
}
```

It is important that documentation is easily human readable, but there is also a need to provide additional styling information to ensure generated documentation is more rich text. Therefore JSDoc should generally follow markdown markup to enrich the text.

While markdown supports HTML tags, it is forbidden in JSDoc blocks.

Code string literals should be braced with the back-tick (‘) instead of quotes. For example:

```
/** Import something from the `deno` module. */
```

Do not document function arguments unless they are non-obvious of their intent (though if they are non-obvious intent, the API should be considered anyways). Therefore `@param` should generally not be used. If `@param` is used, it should not include the `type` as TypeScript is already strongly typed.

```
/**
 * Function with non obvious param.
```

```
 * @param foo Description of non obvious parameter.
 */
```

Vertical spacing should be minimized whenever possible. Therefore single line comments should be written as:

```
/** This is a good single line JSDoc. */
```

And not

```
/**
 * This is a bad single line JSDoc.
 */
```

Code examples should not utilise the triple-back tick (“‘) notation or tags. They should just be marked by indentation, which requires a break before the block and 6 additional spaces for each line of the example. This is 4 more than the first column of the comment. For example:

```
/** A straight forward comment and an example:
 *
 *        import { foo } from "deno";
 *        foo("bar");
 */
```

Code examples should not contain additional comments. It is already inside a comment. If it needs further comments it is not a good example.

Each module should come with a test module.

Every module with public functionality `foo.ts` should come with a test module `foo_test.ts`. A test for a `cli/js` module should go in `cli/js/` ↪ `tests` due to their different contexts, otherwise it should just be a sibling to the tested module.

Unit Tests should be explicit.

For a better understanding of the tests, function should be correctly named as its prompted throughout the test command. Like:

```
test myTestFunction ... ok
```

Example of test:

```
import { assertEquals } from "https://deno.land/std@v0
  ↪ .11/testing/asserts.ts";
import { foo } from "./mod.ts";

Deno.test("myTestFunction" function() {
  assertEquals(foo(), { bar: "bar" });
});
```

Top level functions should not use arrow syntax.

Top level functions should use the `function` keyword. Arrow syntax should be limited to closures.

Bad

```
export const foo = (): string => {
  return "bar";
};
```

Good

```
export function foo(): string {
  return "bar";
}
```

`std`

Do not depend on external code. `https://deno.land/std/` is intended to be baseline functionality that all Deno programs can rely on. We want to guarantee to users that this code does not include potentially unreviewed third party code.

Document and maintain browser compatiblity. If a module is browser compatible, include the following in the JSDoc at the top of the module:

```
/** This module is browser compatible. */
```

Maintain browser compatibility for such a module by either not using the global `Deno` namespace or feature-testing for it. Make sure any new dependencies are also browser compatible.

Internal details

Deno and Linux analogy

Linux	Deno
Processes	Web Workers
Syscalls	Ops
File descriptors (fd)	Resource ids (rid)
Scheduler	Tokio
Userland: libc++ / glib / boost	https://deno.land/std/
/proc/$$/stat	Deno.metrics()
man pages	deno types

Resources Resources (AKA `rid`) are Deno's version of file descriptors. They are integer values used to refer to open files, sockets, and other concepts. For testing it would be good to be able to query the system for how many open resources there are.

```
const { resources, close } = Deno;
console.log(resources());
// { 0: "stdin", 1: "stdout", 2: "stderr" }
close(0);
console.log(resources());
// { 1: "stdout", 2: "stderr" }
```

Metrics Metrics is Deno's internal counter for various statistics.

```
> console.table(Deno.metrics())

     (index)        Values

  opsDispatched       9
   opsCompleted       9
 bytesSentControl    504
  bytesSentData       0
  bytesReceived      856
```

Schematic diagram

Examples

In this chapter you can find some example programs that you can use to learn more about the runtime.

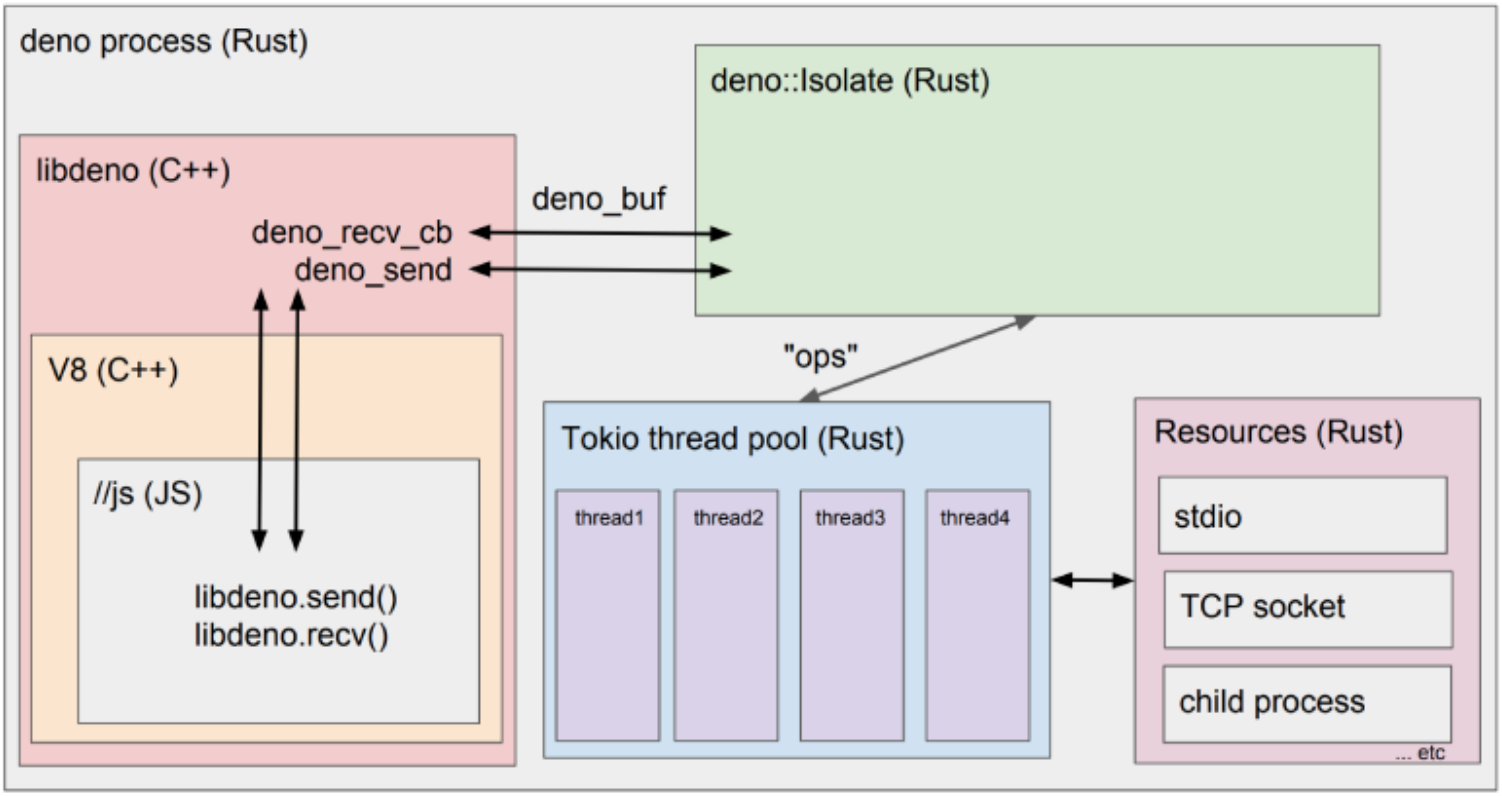

Figure 7: architectural schematic

An implementation of the unix "cat" program

In this program each command-line argument is assumed to be a filename, the file is opened, and printed to stdout.

```
for (let i = 0; i < Deno.args.length; i++) {
  const filename = Deno.args[i];
  const file = await Deno.open(filename);
  await Deno.copy(file, Deno.stdout);
  file.close();
}
```

The `copy()` function here actually makes no more than the necessary kernel -> userspace -> kernel copies. That is, the same memory from which data is read from the file, is written to stdout. This illustrates a general design goal for I/O streams in Deno.

Try the program:

```
deno run --allow-read https://deno.land/std/examples/cat
  ↪ .ts /etc/passwd
```

File server

This one serves a local directory in HTTP.

```
deno install --allow-net --allow-read https://deno.land/
  ↪ std/http/file_server.ts
```

Run it:

```
$ file_server .
Downloading https://deno.land/std/http/file_server.ts...
[...]
HTTP server listening on http://0.0.0.0:4500/
```

And if you ever want to upgrade to the latest published version:

```
file_server --reload
```

TCP echo server

This is an example of a server which accepts connections on port 8080, and returns to the client anything it sends.

```
const listener = Deno.listen({ port: 8080 });
console.log("listening on 0.0.0.0:8080");
for await (const conn of listener) {
  Deno.copy(conn, conn);
}
```

When this program is started, it throws PermissionDenied error.

```
$ deno run https://deno.land/std/examples/echo_server.ts
error: Uncaught PermissionDenied: network access to
  ↪ "0.0.0.0:8080", run again with the --allow-net flag
 $deno$/dispatch_json.ts:40:11
    at DenoError ($deno$/errors.ts:20:5)
    ...
```

For security reasons, Deno does not allow programs to access the network without explicit permission. To allow accessing the network, use a command-line flag:

```
deno run --allow-net https://deno.land/std/examples/
  ↪ echo_server.ts
```

To test it, try sending data to it with netcat:

```
$ nc localhost 8080
hello world
hello world
```

Like the `cat.ts` example, the `copy()` function here also does not make unnecessary memory copies. It receives a packet from the kernel and sends back, without further complexity.

Run subprocess

API Reference

Example:

```
// create subprocess
const p = Deno.run({
  cmd: ["echo", "hello"],
});
```

```
// await its completion
await p.status();
```

Run it:

```
$ deno run --allow-run ./subprocess_simple.ts
hello
```

Here a function is assigned to `window.onload`. This function is called after the main script is loaded. This is the same as onload of the browsers, and it can be used as the main entrypoint.

By default when you use `Deno.run()` subprocess inherits `stdin`, `stdout` and `stderr` of parent process. If you want to communicate with started subprocess you can use `"piped"` option.

```
const fileNames = Deno.args;

const p = Deno.run({
  cmd: [
    "deno",
    "run",
    "--allow-read",
    "https://deno.land/std/examples/cat.ts",
    ...fileNames,
  ],
  stdout: "piped",
  stderr: "piped",
});

const { code } = await p.status();

if (code === 0) {
  const rawOutput = await p.output();
  await Deno.stdout.write(rawOutput);
```

```
} else {
  const rawError = await p.stderrOutput();
  const errorString = new TextDecoder().decode(rawError)
    ↪ ;
  console.log(errorString);
}

Deno.exit(code);
```

When you run it:

```
$ deno run --allow-run ./subprocess.ts <somefile>
[file content]

$ deno run --allow-run ./subprocess.ts non_existent_file
  ↪ .md

Uncaught NotFound: No such file or directory (os error
  ↪ 2)
    at DenoError (deno/js/errors.ts:22:5)
    at maybeError (deno/js/errors.ts:41:12)
    at handleAsyncMsgFromRust (deno/js/dispatch.ts
      ↪ :27:17)
```

Inspecting and revoking permissions

> This program makes use of an unstable Deno feature.
> Learn more about unstable features.

Sometimes a program may want to revoke previously granted permissions. When a program, at a later stage, needs those permissions, it will fail.

```
// lookup a permission
```

```
const status = await Deno.permissions.query({ name: "
  ↪ write" });
if (status.state !== "granted") {
  throw new Error("need write permission");
}

const log = await Deno.open("request.log", { write: true
  ↪ , append: true });

// revoke some permissions
await Deno.permissions.revoke({ name: "read" });
await Deno.permissions.revoke({ name: "write" });

// use the log file
const encoder = new TextEncoder();
await log.write(encoder.encode("hello\n"));

// this will fail.
await Deno.remove("request.log");
```

Handle OS Signals

> This program makes use of an unstable Deno feature. Learn more about unstable features.

API Reference

You can use `Deno.signal()` function for handling OS signals.

```
for await (const _ of Deno.signal(Deno.Signal.SIGINT)) {
  console.log("interrupted!");
}
```

`Deno.signal()` also works as a promise.

```
await Deno.signal(Deno.Signal.SIGINT);
console.log("interrupted!");
```

If you want to stop watching the signal, you can use `dispose()` method of the signal object.

```
const sig = Deno.signal(Deno.Signal.SIGINT);
setTimeout(() => {
  sig.dispose();
}, 5000);

for await (const _ of sig) {
  console.log("interrupted");
}
```

The above for-await loop exits after 5 seconds when sig.dispose() is called.

File system events

To poll for file system events:

```
const watcher = Deno.watchFs("/");
for await (const event of watcher) {
  console.log(">>>> event", event);
  // { kind: "create", paths: [ "/foo.txt" ] }
}
```

Note that the exact ordering of the events can vary between operating systems. This feature uses different syscalls depending on the platform:

- Linux: inotify
- macOS: FSEvents
- Windows: ReadDirectoryChangesW

Testing if current file is the main program

To test if the current script has been executed as the main input to the program check `import.meta.main`.

```
if (import.meta.main) {
  console.log("main");
}
```