```
nvim_lsp.tsserver.setup {
  on_attach = on_attach,
  root_dir = nvim_lsp.util.root_pattern("package.json"),
}
```

For Deno, the example above assumes a deno.json or deno.jsonc file exists at the root of the project.

2.2.1.4.1 coc.nvim

Once you have coc.nvim installed installed, you need to install the required plugin via :CocInstall coc-deno.

Once the plugin is installed, and you want to enable Deno for a workspace, run the command :CocCommand deno.initializeWorkspace and you should be able to utilize commands like gd (goto definition) and gr (go/find references).

2.2.1.4.2 ALE

ALE supports Deno via the Deno language server out of the box and in many uses cases doesn't require additional configuration. Once you have ALE installed you can perform the command :help ale-typescript-deno to get information on the configuration options available.

For more information on how to setup ALE (like key bindings) refer to the official documentation.

2.2.1.4.3 Vim-EasyComplete

Vim-EasyComplete supports Deno without any other configuration. Once you have vim-easycomplete installed, you need install deno via :InstallLspServer deno if you haven't installed deno. You can get more information from official documentation.

2.2.1.5 Emacs

2.2.1.5.1 lsp-mode

Emacs supports Deno via the Deno language server using lsp-mode. Once lsp-mode is installed it should support Deno, which can be configured to support various settings.

2.2.1.5.2 eglot

You can also use built-in Deno language server by using eglot.

An example configuration for Deno via eglot:

```
(add-to-list 'eglot-server-programs '((js-mode typescript-mode) . (eglot-deno "deno" "lsp")))

  (defclass eglot-deno (eglot-lsp-server) ()
    :documentation "A custom class for deno lsp.")

  (cl-defmethod eglot-initialization-options ((server eglot-deno))
    "Passes through required deno initialization options"
    (list :enable t
    :lint t))
```

2.2.1.6 Atom

The Atom editor supports integrating with the Deno language server via the atom-ide-deno package. atom-ide-deno requires that the Deno CLI be installed and the atom-ide-base package to be installed as well.

2.2.1.7 Sublime Text

Sublime Text supports connecting to the Deno language server via the LSP package. You may also want to install the TypeScript package to get full syntax highlighting.

Once you have the LSP package installed, you will want to add configuration to your .sublime-project configuration like the below:

```
{
  "settings": {
    "LSP": {
```

1 Introduction

Deno (/ˈdiːnoʊ/, pronounced dee-no) is a JavaScript, TypeScript, and WebAssembly runtime with secure defaults and a great developer experience.

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

1.1 Feature highlights

- Provides web platform functionality and adopts web platform standards. For example using ES modules, web workers, and support fetch().
- Secure by default. No file, network, or environment access unless explicitly enabled.
- Supports TypeScript out of the box.
- Ships a single executable (deno).
- Provides built-in development tooling like a code formatter (deno fmt), a linter (deno lint), a test runner (deno test), and a language server for your editor.
- Has a set of reviewed (audited) standard modules that are guaranteed to work with Deno.
- Can bundle scripts into a single JavaScript file or executable.

1.2 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 ~31 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.

1.3 Goals

- Ship as just a single executable (deno).
- Provide secure defaults.
  - Unless specifically allowed, scripts can't access files, the environment, or the network.
- Be 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 to improve developer experience.
  - E.g. unit testing, code formatting, and linting.
- Keep V8 concepts out of user land.
- Serve HTTP efficiently.

1.4 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.
- Deno uses "ES Modules" and does not support require(). Third party modules are imported via URLs:

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

1.5 Other key behaviors

- Fetch and cache remote code upon first execution, and never update it 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.

2 Getting Started

In this chapter we'll discuss:

- Installing Deno
- Setting Up Your Environment
- Running a Hello World Script
- Command Line Interface
- Configuration File
- Understanding Permissions
- Debugging Your Code

2.1 Installation

Deno works on macOS, Linux, and Windows. Deno is a single binary executable. It has no external dependencies.
On macOS, both M1 (arm64) and Intel (x64) executables are provided. On Linux and Windows, only x64 is supported.

2.1.1 Download and install

deno_install provides convenience scripts to download and install the binary.

1. Using Shell (macOS and Linux):

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

2. Using PowerShell (Windows):

```
irm https://deno.land/install.ps1 | iex
```

3. Using Scoop (Windows):

```
scoop install deno
```

4. Using Chocolatey (Windows):

```
choco install deno
```

5. Using Homebrew (macOS):

```
brew install deno
```

6. Using Nix (macOS and Linux):

```
nix-shell -p deno
```

7. Build and install from source using Cargo:

```
cargo install deno --locked
```

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.

2.1.2 Docker

For more information and instructions on the official Docker images: https://github.com/denoland/deno_docker

2.1.3 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.

2.1.4 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
```

2.1.5 Building from source

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

2.2 Set Up Your Environment

The Deno CLI contains a lot of the tools that are commonly needed for developing applications, including a full language server to help power your IDE of choice. Installing is all you need to do to make these tools available to you.
Outside using Deno with your favorite IDE, this section also documents shell completions and environment variables.

2.2.1 Using an editor/IDE

There is broad support for Deno in editors/IDEs. The following sections provide information about how to use Deno with editors. Most editors integrate directly into Deno using the Language Server Protocol and the language server that is integrated into the Deno CLI.
If you are trying to write or support a community integration to the Deno language server, there is some documentation located in the Deno CLI code repository, but also feel free to join the Discord community in the #dev-lsp channel.

2.2.1.1 Visual Studio Code

There is an official extension for Visual Studio Code called vscode_deno. When installed, it will connect to the language server built into the Deno CLI.
Because most people work in mixed environments, the extension does not enable a workspace as Deno enabled by default, and it requires that the `"deno.enable"` flag to be set. You can change the settings yourself, or you can choose `Deno: Initialize Workspace Configuration` from the command palette to enable your project.
More information can be found in the Using Visual Studio Code section of the manual.

2.2.1.2 JetBrains IDEs

You can get support for Deno in WebStorm and other JetBrains IDEs, including PhpStorm, IntelliJ IDEA Ultimate, and PyCharm Professional. For this, install the official Deno plugin from Preferences / Settings | Plugins - Marketplace.
Check out this blog post to learn more about how to get started with Deno.

2.2.1.3 Vim/Neovim via plugins

Deno is well-supported on both Vim and Neovim via coc.nvim, vim-easycomplete and ALE. coc.nvim offers plugins to integrate to the Deno language server while ALE supports it out of the box.

2.2.1.4 Neovim 0.6+ using the built-in language server

To use the Deno language server install nvim-lspconfig and follow the instructions to enable the supplied Deno configuration.
Note that if you also have `tsserver` as an LSP client, you may run into issues where both `tsserver` and `denols` are attached to your current buffer. To resolve this, make sure to set some unique `root_dir` for both `tsserver` and `denols`. Here is an example of such a configuration:

```
nvim_lsp.denols.setup {
  on_attach = on_attach,
  root_dir = nvim_lsp.util.root_pattern("deno.json", "deno.jsonc"),
}
```

- HTTP_PROXY - The proxy address to use for HTTP requests. See the Proxies section for more information.
- HTTPS_PROXY - The proxy address to use for HTTPS requests. See the Proxies section for more information.
- NO_COLOR - If set, this will cause the Deno CLI to not send ANSI color codes when writing to stdout and stderr. See the website https://no-color.org/ for more information on this de facto standard. The value of this flag can be accessed at runtime without permission to read the environment variables by checking the value of Deno.noColor.
- NO_PROXY - Indicates hosts which should bypass the proxy set in the other environment variables. See the Proxies section for more information.

2.3 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.

2.3.1 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@0.156.0/examples/welcome.ts
```

2.3.2 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@0.156.0/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@0.156.0/examples/curl.ts
https://example.com
```

2.3.3 Reading a file

```
      "deno": {
        "command": ["deno", "lsp"],
        "initializationOptions": {
          // "config": "", // Sets the path for the config file in your project
          "enable": true,
          // "importMap": "", // Sets the path for the import-map in your project
          "lint": true,
          "unstable": false
        },
        "enabled": true,
        "languages": [
          {
            "languageId": "javascript",
            "scopes": ["source.js"],
            "syntaxes": [
              "Packages/Babel/JavaScript (Babel).sublime-syntax",
              "Packages/JavaScript/JavaScript.sublime-syntax"
            ]
          },
          {
            "languageId": "javascriptreact",
            "scopes": ["source.jsx"],
            "syntaxes": [
              "Packages/Babel/JavaScript (Babel).sublime-syntax",
              "Packages/JavaScript/JavaScript.sublime-syntax"
            ]
          },
          {
            "languageId": "typescript",
            "scopes": ["source.ts"],
            "syntaxes": [
              "Packages/TypeScript-TmLanguage/TypeScript.tmLanguage",
              "Packages/TypeScript Syntax/TypeScript.tmLanguage"
            ]
          },
          {
            "languageId": "typescriptreact",
            "scopes": ["source.tsx"],
            "syntaxes": [
              "Packages/TypeScript-TmLanguage/TypeScriptReact.tmLanguage",
              "Packages/TypeScript Syntax/TypeScriptReact.tmLanguage"
            ]
          }
        ]
      }
    }
  }
```

```
}
```

2.2.1.8 Nova

The Nova editor can integrate the Deno language server via the Deno extension.

2.2.1.9 GitHub Codespaces

GitHub Codespaces allows you to develop fully online or remotely on your local machine without needing to configure or install Deno. It is currently in early access.

If a project is a Deno enabled project and contains the .devcontainer configuration as part of the repository, opening the project in GitHub Codespaces should just "work". If you are starting a new project, or you want to add Deno support to an existing code space, it can be added by selecting the Codespaces: Add Development Container Configuration Files... from the command pallet and then selecting Show All Definitions... and then searching for the Deno definition.

Once selected, you will need to rebuild your container so that the Deno CLI is added to the container. After the container is rebuilt, the code space will support Deno.

2.2.1.10 Kakoune

Kakoune supports connecting to the Deno language server via the kak-lsp client. Once kak-lsp is installed an example of configuring it up to connect to the Deno language server is by adding the following to your kak-lsp.toml:

```
[language.typescript]
filetypes = ["typescript", "javascript"]
roots = [".git"]
command = "deno"
args = ["lsp"]
[language.typescript.settings.deno]
enable = true
lint = true
```

2.2.2 Shell completions

Built into the Deno CLI is support to generate shell completion information for the CLI itself. By using deno completions <shell>, the Deno CLI will output to stdout the completions. Current shells that are supported:

- bash
- elvish
- fish
- powershell
- zsh

2.2.2.1 bash example

Output the completions and add them to the environment:

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

2.2.2.2 PowerShell example

2.2.2.3 Output the completions:

```
> deno completions powershell >> $profile
> .$profile
```

This will create a Powershell profile at $HOME\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1, and it will be run whenever you launch the PowerShell.

2.2.2.4 zsh example

You should have a directory where the completions can be saved:

```
> mkdir ~/.zsh
```

Then output the completions:

```
> deno completions zsh > ~/.zsh/_deno
```

And ensure the completions get loaded in your ~/.zshrc:

```
fpath=(~/.zsh $fpath)
autoload -Uz compinit
compinit -u
```

If after reloading your shell and completions are still not loading, you may need to remove ~/.zcompdump/ to remove previously generated completions and then compinit to generate them again.

2.2.2.5 zsh example with ohmyzsh and antigen

ohmyzsh is a configuration framework for zsh and can make it easier to manage your shell configuration. antigen is a plugin manager for zsh.

Create the directory to store the completions and output the completions:

```
> mkdir ~/.oh-my-zsh/custom/plugins/deno
> deno completions zsh > ~/.oh-my-zsh/custom/plugins/deno/_deno
```

Then your .zshrc might look something like this:

```
source /path-to-antigen/antigen.zsh

# Load the oh-my-zsh's library.
antigen use oh-my-zsh

antigen bundle deno
```

2.2.2.6 fish example

Output the completions to a deno.fish file into the completions directory in the fish config folder:

```
> deno completions fish > ~/.config/fish/completions/deno.fish
```

2.2.3 Environment variables

There are several environment variables which can impact the behavior of Deno:

- DENO_AUTH_TOKENS - a list of authorization tokens which can be used to allow Deno to access remote private code. See the Private modules and repositories section for more details.
- DENO_TLS_CA_STORE - a list of certificate stores which will be used when establishing TLS connections. The available stores are mozilla and system. You can specify one, both or none. The order you specify the store determines the order in which certificate chains will be attempted to resolved. The default value is mozilla. The mozilla store will use the bundled Mozilla certs provided by webpki-roots. The system store will use your platforms native certificate store. The exact set of Mozilla certs will depend on the version of Deno you are using. If you specify no certificate stores, then no trust will be given to any TLS connection without also specifying DENO_CERT or --cert or specifying a specific certificate per TLS connection.
- DENO_CERT - load a certificate authority from a PEM encoded file. This "overrides" the --cert option. See the Proxies section for more information.
- DENO_DIR - this will set the directory where cached information from the CLI is stored. This includes items like cached remote modules, cached transpiled modules, language server cache information and persisted data from local storage. This defaults to the operating systems default cache location and then under the deno path.
- DENO_INSTALL_ROOT - When using deno install where the installed scripts are stored. This defaults to $HOME/.deno/bin.
- DENO_NO_PROMPT - Set to disable permission prompts on access (alternative to passing --no-prompt on invocation).
- DENO_WEBGPU_TRACE - The directory to use for WebGPU traces.

Subcommand	Type checking mode
deno cache	✕ None
deno check	🗀 Local
deno compile	🗀 Local
deno eval	✕ None
deno repl	✕ None
deno run	✕ None
deno test	🗀 Local

2.4.6.2 Permission flags

These are listed here.

2.4.6.3 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 ...
--location <HREF>	Value of 'globalThis.location' used by some web APIs
--prompt	Fallback to prompt if required permission wasn't passed
--seed <NUMBER>	Seed Math.random()
--v8-flags=<v8-flags>	Set V8 command line options. For help: ...

2.5 Configuration File

Deno supports configuration file that allows to customize built-in TypeScript compiler, formatter and linter.

The configuration file supports .json and .jsonc extensions. Since v1.18, Deno will automatically detect deno.json or deno.jsonc configuration file if it's in your current working directory (or parent directories). To manually tell Deno to use a specific configuration file pass --config path/to/file.json flag.

⚠ Starting with Deno v1.22 you can disable automatic detection of the configuration file, by passing --no-config.

Note that using a configuration file is not required now, and will not be required in the future. Deno still works best with the default options and no configuration file. All options specified in the configuration file can also be set using command line flags (for example --options-use-tabs for deno fmt). Using the configuration file should be considered an "as needed" feature, not something every user should be reaching to as the first thing when setting up a project.

2.5.1 Example

```
{
  "compilerOptions": {
    "allowJs": true,
    "lib": ["deno.window"],
    "strict": true
  },
  "importMap": "import_map.json",
```

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.

```
import { copy } from "https://deno.land/std@0.156.0/streams/conversion.ts";
const filenames = Deno.args;
for (const filename of filenames) {
  const file = await Deno.open(filename);
  await 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:

```
# macOS / Linux
deno run --allow-read https://deno.land/std@0.156.0/examples/cat.ts /etc/hosts

# Windows
deno run --allow-read https://deno.land/std@0.156.0/examples/cat.ts
"C:\Windows\System32\Drivers\etc\hosts"
```

2.3.4 TCP server

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

```
import { copy } from "https://deno.land/std@0.156.0/streams/conversion.ts";
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) {
  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@0.156.0/examples/echo_server.ts
```

To test it, try sending data to it with netcat (or telnet on Windows):

Note for Windows users: netcat is not available on Windows. Instead you can use the built-in telnet client. The telnet client is disabled in Windows by default. It is easy to enable however: just follow the instructions on Microsoft TechNet

```
# Note for Windows users: replace the `nc` below with `telnet`
$ 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.

2.3.5 More examples

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

2.4 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 subcommands supported, such as deno bundle. To see subcommand-specific help, for example for bundle, you can similarly run one of:

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

Detailed guides for each subcommand can be found here.

2.4.1 Script source

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

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

2.4.2 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 and ergonomic way of distinguishing between runtime flags and script arguments.
2. 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 subcommands. We discuss these below.

2.4.3 Watch mode

You can supply the --watch flag to deno run, deno test, deno bundle, and deno fmt to enable the built-in file watcher. The files that are watched depend on the subcommand used:

- for deno run, deno test, and deno bundle the entrypoint, and all local files the entrypoint(s) statically import(s) will be watched.
- for deno fmt all local files and directories specified as command line arguments (or the working directory if no specific files/directories is passed) are watched.

Whenever one of the watched files is changed on disk, the program will automatically be restarted / formatted / tested / bundled.

```
deno run --watch main.ts
deno test --watch
deno fmt --watch
```

2.4.4 Integrity flags (lock files)

Affect commands which can download resources to the cache: deno cache, deno run, deno test, deno bundle, deno doc, and deno compile.

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

Find out more about these here.

2.4.5 Cache and compilation flags

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

--config <FILE>	Load configuration file
--import-map <FILE>	Load import map file
--no-remote	Do not resolve remote modules
--reload=<CACHE_BLOCKLIST>	Reload source code cache (recompile TypeScript)
--unstable	Enable unstable APIs

2.4.6 Runtime flags

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

2.4.6.1 Type checking flags

You can type-check your code (without executing it) using the command:

```
> deno check main.ts
```

You can also type-check your code before execution by using the --check argument to deno run:

```
> deno run --check main.ts
```

This flag affects deno run, deno eval, deno repl and deno cache. The following table describes the type-checking behavior of various subcommands. Here "Local" means that only errors from local code will induce type-errors, modules imported from https URLs (remote) may have type errors that are not reported. (To turn on type-checking for all modules, use --check=all.)

Subcommand	Type checking mode
deno bench	🗀 Local
deno bundle	🗀 Local

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.

⚠ If you use --inspect flag, the code will start executing immediately. If your program is short, you might not have enough time to connect the debugger before the program finishes execution. In such cases, try running with --inspect-brk flag instead, or add a timeout at the end of your code.

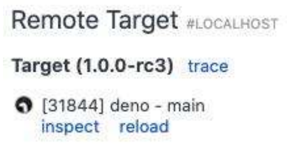
2.7.1 Chrome Devtools

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@0.156.0/http/file_server.ts
Debugger listening on ws://127.0.0.1:9229/ws/1e82c406-85a9-44ab-86b6-7341583480b1
Download https://deno.land/std@0.156.0/http/file_server.ts
Compile https://deno.land/std@0.156.0/http/file_server.ts
...
```

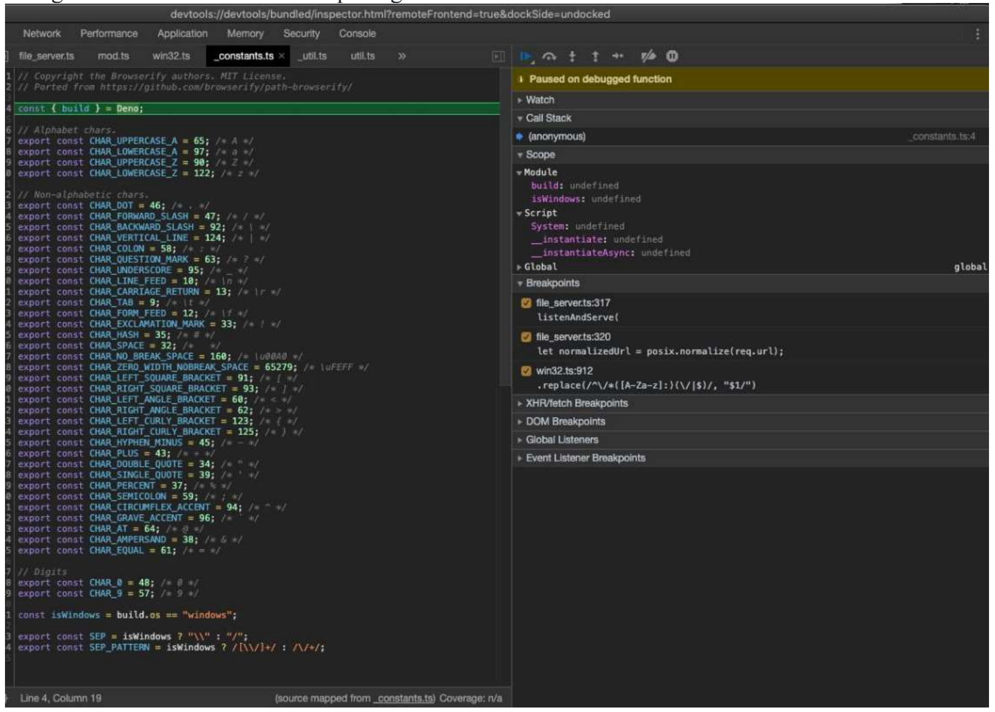
In a Chromium derived browser such as Google Chrome or Microsoft Edge, open chrome://inspect and click Inspect next to target:


Remote Target #LOCALHOST

Target (1.0.0-rc3) trace

[31844] deno - main
inspect reload

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



```
// Copyright the Browserify authors. MIT License.
// Ported from https://github.com/browserify/path-browserify/

const { build } = Deno;

// Alphabet chars.
export const CHAR_UPPERCASE_A = 65; /* A */
export const CHAR_LOWERCASE_A = 97; /* a */
export const CHAR_UPPERCASE_Z = 90; /* Z */
export const CHAR_LOWERCASE_Z = 122; /* z */

// Non-alphabetic chars.
export const CHAR_DOT = 46; /* . */
export const CHAR_FORWARD_SLASH = 47; /* / */
export const CHAR_BACKWARD_SLASH = 92; /* \ */
export const CHAR_VERTICAL_LINE = 124; /* | */
export const CHAR_COLON = 58; /* : */
export const CHAR_QUESTION_MARK = 63; /* ? */
export const CHAR_UNDERSCORE = 95; /* _ */
export const CHAR_LINE_FEED = 10; /* \n */
export const CHAR_CARRIAGE_RETURN = 13; /* \r */
export const CHAR_TAB = 9; /* \t */
export const CHAR_FORM_FEED = 12; /* \f */
export const CHAR_EXCLAMATION_MARK = 33; /* ! */
export const CHAR_HASH = 35; /* # */
export const CHAR_SPACE = 32; /*   */
export const CHAR_NO_BREAK_SPACE = 160; /* \u00A0 */
export const CHAR_ZERO_WIDTH_NOBREAK_SPACE = 65279; /* \uFEFF */
export const CHAR_LEFT_SQUARE_BRACKET = 91; /* [ */
export const CHAR_RIGHT_SQUARE_BRACKET = 93; /* ] */
export const CHAR_LEFT_ANGLE_BRACKET = 60; /* < */
export const CHAR_RIGHT_ANGLE_BRACKET = 62; /* > */
export const CHAR_LEFT_CURLY_BRACKET = 123; /* { */
export const CHAR_RIGHT_CURLY_BRACKET = 125; /* } */
export const CHAR_HYPHEN_MINUS = 45; /* - */
export const CHAR_PLUS = 43; /* + */
export const CHAR_DOUBLE_QUOTE = 34; /* " */
export const CHAR_SINGLE_QUOTE = 39; /* ' */
export const CHAR_PERCENT = 37; /* % */
export const CHAR_SEMICOLON = 59; /* ; */
export const CHAR_CIRCUMFLEX_ACCENT = 94; /* ^ */
export const CHAR_GRAVE_ACCENT = 96; /* ` */
export const CHAR_AT = 64; /* @ */
export const CHAR_AMPERSAND = 38; /* & */
export const CHAR_EQUAL = 61; /* = */

// Digits
export const CHAR_0 = 48; /* 0 */
export const CHAR_9 = 57; /* 9 */

const isWindows = build.os == "windows";

export const SEP = isWindows ? "\\" : "/";
export const SEP_PATTERN = isWindows ? /[\\/]+/ : /\/+/;
```

Line 4, Column 19 (source mapped from _constants.ts) Coverage: n/a

Paused on debugged function

Watch

Call Stack

(anonymous) _constants.ts:4

Scope

```
Module
  build: undefined
  isWindows: undefined
Script
  System: undefined
  __instantiate: undefined
  __instantiateAsync: undefined
Global                                   global
```

Breakpoints

✓ file_server.ts:317
`listenAndServe(`

✓ file_server.ts:320
`let normalizedUrl = posix.normalize(req.url);`

✓ win32.ts:912
`.replace(/^\/*([A-Za-z]:)(\/|$)/, "$1/")`

XHR/fetch Breakpoints

DOM Breakpoints

Global Listeners

Event Listener Breakpoints

```
  "lint": {
    "files": {
      "include": ["src/"],
      "exclude": ["src/testdata/"]
    },
    "rules": {
      "tags": ["recommended"],
      "include": ["ban-untagged-todo"],
      "exclude": ["no-unused-vars"]
    }
  },
  "fmt": {
    "files": {
      "include": ["src/"],
      "exclude": ["src/testdata/"]
    },
    "options": {
      "useTabs": true,
      "lineWidth": 80,
      "indentWidth": 4,
      "singleQuote": true,
      "proseWrap": "preserve"
    }
  },
  "test": {
    "files": {
      "include": ["src/"],
      "exclude": ["src/testdata/"]
    }
  }
}
```

2.5.2 JSON schema

A JSON schema file is available for editors to provide autocomplete. The file is versioned and available at: https://deno.land/x/deno/cli/schemas/config-file.v1.json

2.6 Permissions

Deno is secure by default. Therefore, unless you specifically enable it, a program run with Deno has no file, network, or environment access. Access to security sensitive functionality requires that permissions have been granted to an executing script through command line flags, or a runtime permission prompt.

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

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

2.6.1 Permissions list

The following permissions are available:

- **--allow-env=<allow-env>** Allow environment access for things like getting and setting of environment variables. Since Deno 1.9, you can specify an optional, comma-separated list of environment variables to provide an allow-list of allowed 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 IP addresses or hostnames (optionally with ports) to provide an allow-list of allowed network addresses.
- **--allow-ffi** Allow loading of dynamic libraries. Be aware that dynamic libraries are not run in a sandbox and therefore do not have the same security restrictions as the Deno process. Therefore, use with caution. Please note that --allow-ffi 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 an allow-list of allowed file system access.
- **--allow-run=<allow-run>** Allow running subprocesses. Since Deno 1.9, You can specify an optional, comma-separated list of subprocesses to provide an allow-list of allowed 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 an allow-list of allowed file system access.
- **-A, --allow-all** Allow all permissions. This enables all security sensitive functions. Use with caution.

2.6.2 Configurable permissions

Some permissions allow you to grant access to a specific list of entities (files, servers, etc) rather than to everything.

2.6.2.1 File system access

This example restricts file system access by allowing read-only access to the /usr directory. In consequence the execution fails as the process was attempting to read a file in the /etc directory:

```
$ deno run --allow-read=/usr https://deno.land/std@0.156.0/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 allowing access to /etc instead:

```
deno run --allow-read=/etc https://deno.land/std@0.156.0/examples/cat.ts /etc/passwd
```

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

Note for Windows users: the /etc and /usr directories and the /etc/passwd file do not exist on Windows. If you want to run this example yourself,
replace /etc/passwd with C:\Windows\System32\Drivers\etc\hosts, and /usr with C:\Users.

2.6.2.2 Network access

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

This is an example of how to allow network access to specific hostnames or ip addresses, optionally locked to a specified port:

```
# Multiple hostnames, all ports allowed
deno run --allow-net=github.com,deno.land fetch.js

# A hostname at port 80:
deno run --allow-net=deno.land:80 fetch.js

# An ipv4 address on port 443
```

```
deno run --allow-net=1.1.1.1:443 fetch.js

# A ipv6 address, all ports allowed
deno run --allow-net=[2606:4700:4700::1111] fetch.js
```

If fetch.js tries to establish network connections to any hostname or IP not explicitly allowed, the relevant call will throw an exception.

Allow net calls to any hostname/ip:

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

2.6.2.3 Environment variables

```
// env.js
Deno.env.get("HOME");
```

This is an example of how to allow access to environment variables:

```
# Allow all environment variables
deno run --allow-env env.js

# Allow access to only the HOME env var
deno run --allow-env=HOME env.js
```

Note for Windows users: environment variables are case insensitive on Windows, so Deno also matches them case insensitively (on Windows only).

2.6.2.4 Subprocess permissions

Subprocesses are very powerful, and can be a little scary: they access system resources regardless of the permissions you granted to the Deno process that spawns them. The cat program on unix systems can be used to read files from disk. If you start this program through the Deno.run API it will be able to read files from disk even if the parent Deno process can not read the files directly. This is often referred to as privilege escalation.

Because of this, make sure you carefully consider if you want to grant a program --allow-run access: it essentially invalidates the Deno security sandbox. If you really need to spawn a specific executable, you can reduce the risk by limiting which programs a Deno process can start by passing specific executable names to the --allow-run flag.

```
// run.js
const proc = Deno.run({ cmd: ["whoami"] });
# Allow only spawning a `whoami` subprocess:
deno run --allow-run=whoami run.js

# Allow running any subprocess:
deno run --allow-run run.js
```

You can only limit the executables that are allowed; if permission is granted to execute it then any parameters can be passed. For example if you pass --allow-run=cat then the user can use cat to read any file.

2.6.3 Conference

Permission flags were explained by Ryan Dahl in his 2020 talk about the Deno security model at Speakeasy JS: https://www.youtube.com/watch?v=r5F6dekUmdE#t=34m57

2.7 Debugging Your Code

Deno supports the V8 Inspector Protocol used by Chrome, Edge and Node.js. This makes it possible to debug Deno programs using Chrome DevTools or other clients that support the protocol (for example VSCode).

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

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).

3.2 Program Lifecycle

Deno supports browser compatible lifecycle events: load, beforeunload 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, this event cannot be canceled.

Listeners for beforeunload need to be synchronous and can be cancelled to keep the program running.

Listeners for unload events need to be synchronous and cannot be cancelled.

Example:

main.ts

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

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

globalThis.addEventListener("load", handler);

globalThis.addEventListener("beforeunload", handler);

globalThis.addEventListener("unload", handler);

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

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

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

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

imported.ts

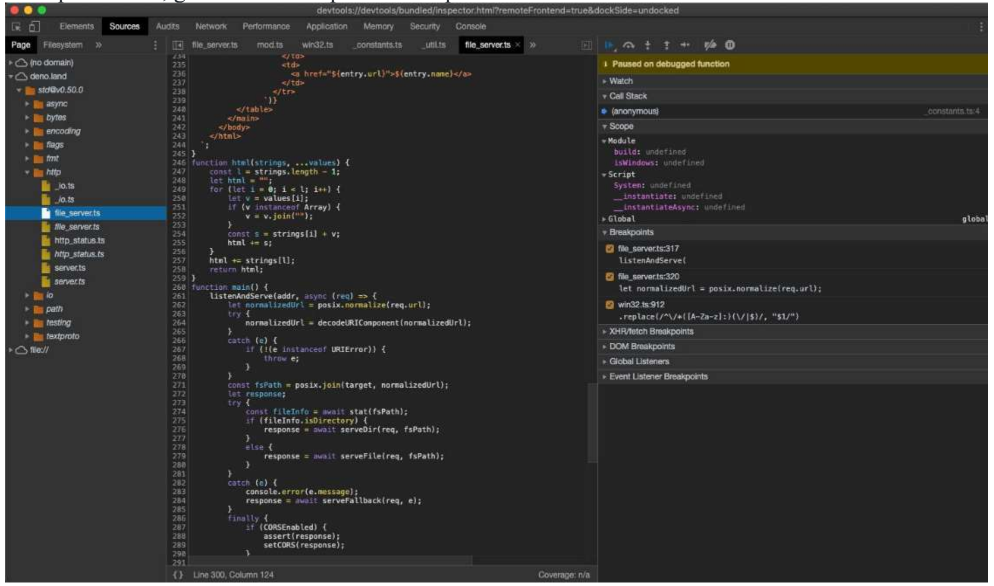
```
const handler = (e: Event): void => {
  console.log(`got ${e.type} event in event handler (imported)`);
};

globalThis.addEventListener("load", handler);
globalThis.addEventListener("beforeunload", handler);
globalThis.addEventListener("unload", handler);

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

You might notice that DevTools pauses execution on the first line of _constants.ts instead of file_server.ts. This is expected behavior caused by the way ES modules are evaluated in JavaScript (_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:



```
                </td>
                <td>
                  <a href="${entry.url}">${entry.name}</a>
                </td>
              </tr>
            `)}
          </table>
        </main>
      </body>
    </html>
  `;
}
function html(strings, ...values) {
    const l = strings.length - 1;
    let html = "";
    for (let i = 0; i < l; i++) {
        let v = values[i];
        if (v instanceof Array) {
            v = v.join("");
        }
        const s = strings[i] + v;
        html += s;
    }
    html += strings[l];
    return html;
}
function main() {
    listenAndServe(addr, async (req) => {
        let normalizedUrl = posix.normalize(req.url);
        try {
            normalizedUrl = decodeURIComponent(normalizedUrl);
        }
        catch (e) {
            if (!(e instanceof URIError)) {
                throw e;
            }
        }
        const fsPath = posix.join(target, normalizedUrl);
        let response;
        try {
            const fileInfo = await stat(fsPath);
            if (fileInfo.isDirectory) {
                response = await serveDir(req, fsPath);
            }
            else {
                response = await serveFile(req, fsPath);
            }
        }
        catch (e) {
            console.error(e.message);
            response = await serveFallback(req, e);
        }
        finally {
            if (CORSEnabled) {
                assert(response);
                setCORS(response);
            }
```

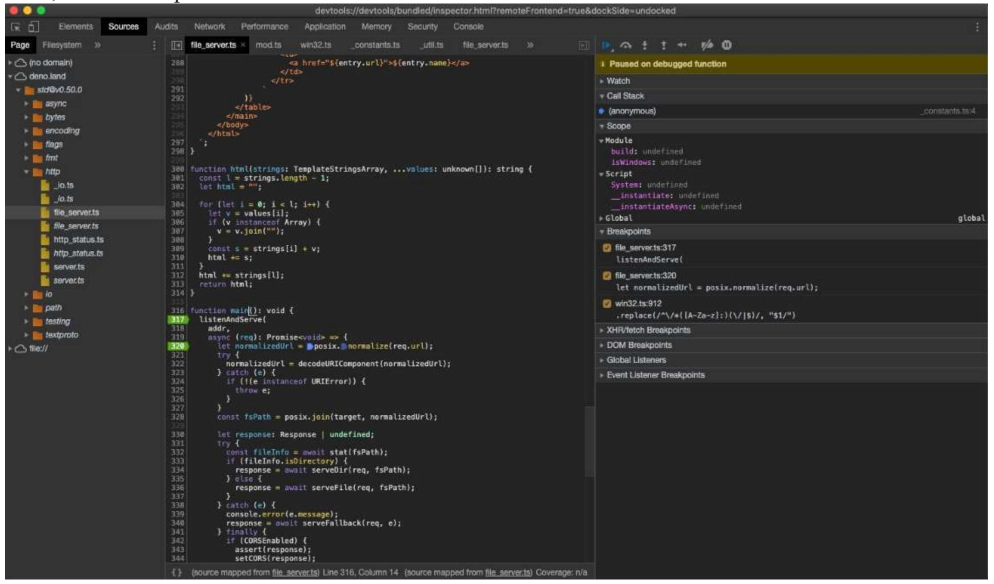
```
build: undefined
isWindows: undefined
System: undefined
__instantiate: undefined
__instantiateAsync: undefined

file_server.ts:317
listenAndServe(
file_server.ts:320
let normalizedUrl = posix.normalize(req.url);
win32.ts:912
.replace(/^\/+([A-Za-z]:)(\/|$)/, "$1/")

Line 300, Column 124                Coverage: n/a
```

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:



```
                  <a href="${entry.url}">${entry.name}</a>
                </td>
              </tr>
            `
          )}
        </table>
      </main>
    </body>
  </html>
  `;
}

function html(strings: TemplateStringsArray, ...values: unknown[]): string {
  const l = strings.length - 1;
  let html = "";

  for (let i = 0; i < l; i++) {
    let v = values[i];
    if (v instanceof Array) {
      v = v.join("");
    }
    const s = strings[i] + v;
    html += s;
  }
  html += strings[l];
  return html;
}

function main(): void {
  listenAndServe(
    addr,
    async (req): Promise<void> => {
      let normalizedUrl = posix.normalize(req.url);
      try {
        normalizedUrl = decodeURIComponent(normalizedUrl);
      } catch (e) {
        if (!(e instanceof URIError)) {
          throw e;
        }
      }
      const fsPath = posix.join(target, normalizedUrl);

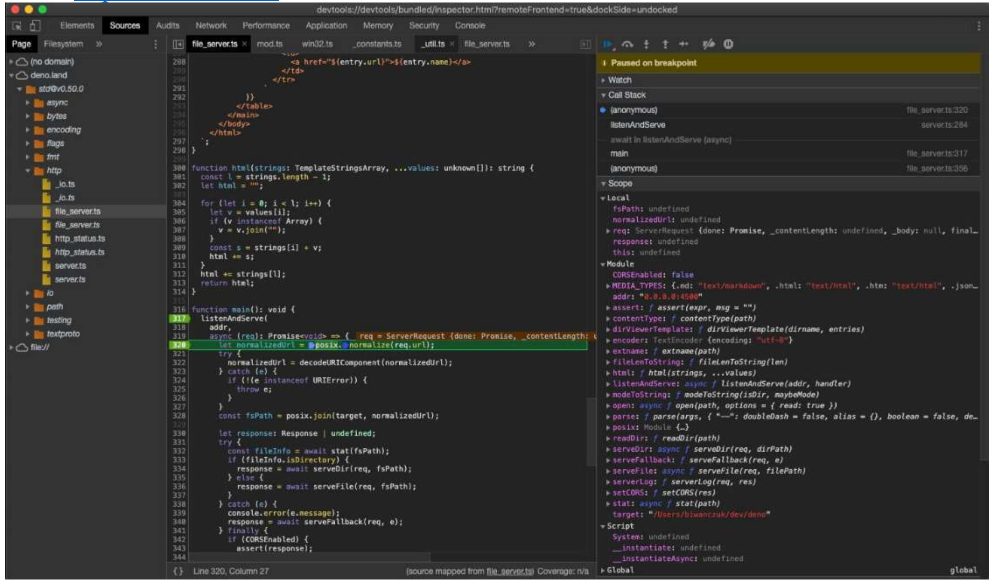
      let response: Response | undefined;
      try {
        const fileInfo = await stat(fsPath);
        if (fileInfo.isDirectory) {
          response = await serveDir(req, fsPath);
        } else {
          response = await serveFile(req, fsPath);
        }
      } catch (e) {
        console.error(e.message);
        response = await serveFallback(req, e);
      } finally {
        if (CORSEnabled) {
          assert(response);
          setCORS(response);
```

```
build: undefined
isWindows: undefined
System: undefined
__instantiate: undefined
__instantiateAsync: undefined

file_server.ts:317
listenAndServe(
file_server.ts:320
let normalizedUrl = posix.normalize(req.url);
win32.ts:912
.replace(/^\/+([A-Za-z]:)(\/|$)/, "$1/")

(source mapped from file_server.ts) Line 316, Column 14  (source mapped from file_server.ts) Coverage: n/a
```

As soon as we've added the breakpoint, DevTools automatically opens 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 can 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, try send a request and inspect it in Devtools:
$ curl http://0.0.0.0:4507/



```
            <a href="${entry.url}">${entry.name}</a>
          </td>
        </tr>
        `
      })
      </table>
    </main>
  </body>
</html>
`;
}

function html(strings: TemplateStringsArray, ...values: unknown[]): string {
  const l = strings.length - 1;
  let html = "";

  for (let i = 0; i < l; i++) {
    let v = values[i];
    if (v instanceof Array) {
      v = v.join("");
    }
    const s = strings[i] + v;
    html += s;
  }
  html += strings[l];
  return html;
}

function main(): void {
  listenAndServe(
    addr,
    async (req): Promise<void> => {  req = ServerRequest {done: Promise, _contentLength:
      let normalizedUrl = posix.normalize(req.url);
      try {
        normalizedUrl = decodeURIComponent(normalizedUrl);
      } catch (e) {
        if (!(e instanceof URIError)) {
          throw e;
        }
      }
      const fsPath = posix.join(target, normalizedUrl);

      let response: Response | undefined;
      try {
        const fileInfo = await stat(fsPath);
        if (fileInfo.isDirectory) {
          response = await serveDir(req, fsPath);
        } else {
          response = await serveFile(req, fsPath);
        }
      } catch (e) {
        console.error(e.message);
        response = await serveFallback(req, e);
      } finally {
        if (CORSEnabled) {
          assert(response);
```

```
▾Local
  fsPath: undefined
  normalizedUrl: undefined
 ▸req: ServerRequest {done: Promise, _contentLength: undefined, _body: null, final…
  response: undefined
  this: undefined
▾Module
  CORSEnabled: false
 ▸MEDIA_TYPES: {.md: "text/markdown", .html: "text/html", .htm: "text/html", .json…
  addr: "0.0.0.0:4500"
 ▸assert: ƒ assert(expr, msg = "")
 ▸contentType: ƒ contentType(path)
 ▸dirViewerTemplate: ƒ dirViewerTemplate(dirname, entries)
 ▸encoder: TextEncoder {encoding: "utf-8"}
 ▸extname: ƒ extname(path)
 ▸fileLenToString: ƒ fileLenToString(len)
 ▸html: ƒ html(strings, ...values)
 ▸listenAndServe: async ƒ listenAndServe(addr, handler)
 ▸modeToString: ƒ modeToString(isDir, maybeMode)
 ▸open: async ƒ open(path, options = { read: true })
 ▸parse: ƒ parse(args, { "--": doubleDash = false, alias = {}, boolean = false, de…
 ▸posix: Module {…}
 ▸readDir: ƒ readDir(path)
 ▸serveDir: async ƒ serveDir(req, dirPath)
 ▸serveFallback: ƒ serveFallback(req, e)
 ▸serveFile: async ƒ serveFile(req, filePath)
 ▸serverLog: ƒ serverLog(req, res)
 ▸setCORS: ƒ setCORS(res)
 ▸stat: async ƒ stat(path)
  target: "/Users/biwanczuk/dev/deno"
▾Script
  System: undefined
  __instantiate: undefined
  __instantiateAsync: undefined
▸Global                                                                  global
```

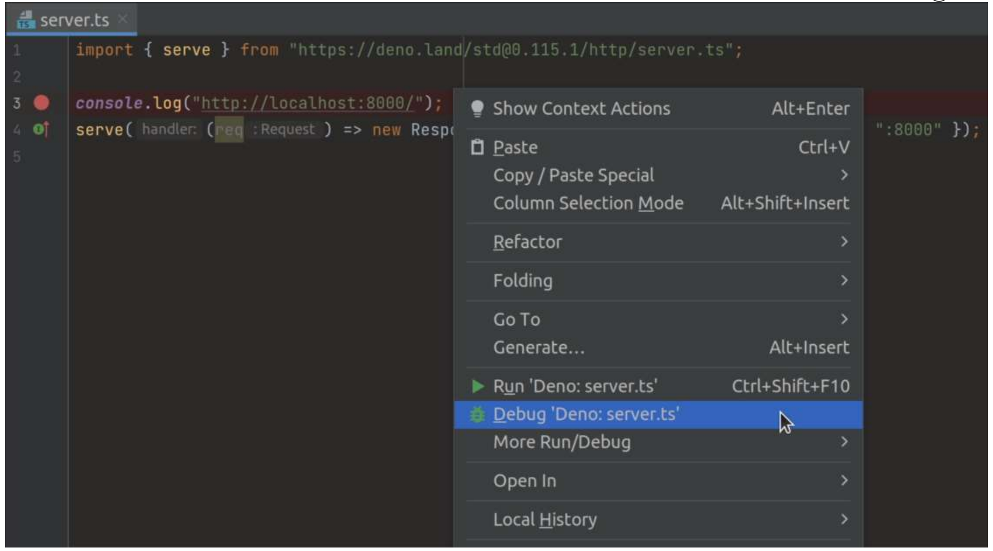
At this point we can introspect the contents of the request and go step-by-step to debug the code.

2.7.2 VSCode

Deno can be debugged using VSCode. This is best done with help from the official vscode_deno extension. Documentation for this can be found here.

2.7.3 JetBrains IDEs

Note: make sure you have this Deno plugin installed and enabled in Preferences / Settings | Plugins. For more information, see this blog post.
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.



```
import { serve } from "https://deno.land/std@0.115.1/http/server.ts";

console.log("http://localhost:8000/");
serve( handler: (req :Request ) => new Resp                    ":8000" });
```

This will create a run/debug configuration with no permission flags set. If you want to configure them, open your run/debug configuration and add the required flags to the Command field.

2.7.4 Other

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

3 The Runtime

Documentation for all runtime functions (Web APIs + Deno global) can be found on doc.deno.land/deno/stable with unstable APIs which are enabled via the --unstable flag at doc.deno.land/deno/unstable.

- Web Platform APIs

For APIs where a web standard already exists, like fetch for HTTP requests, Deno uses these rather than inventing a new proprietary API.
For more details, view the chapter on Web Platform APIs.

- 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, serving HTTP, and executing subprocesses, etc.
The TypeScript definitions for the Deno namespaces can be found in the lib.deno.ns.d.ts file.

3.1 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**.

3.1.1 Standard modules

```
[
  { name: "read", path: "/foo" },
  { name: "read", path: "/bar" },
  { name: "write", path: "/foo" },
];
```

Deno's permission revocation algorithm works by removing every element from this set which is stronger than the argument permission descriptor.

Deno does not allow "fragmented" permission states, where some strong permission is granted with exclusions of weak permissions implied by it. Such a system would prove increasingly complex and unpredictable as you factor in a wider variety of use cases and the "denied" state. This is a calculated trade-off of granularity for security.

3.4 Web Platform APIs

Deno aims to use web platform APIs (like fetch) instead of inventing a new proprietary API where it makes sense. These APIs generally follow the specifications and should match the implementation in Chrome and Firefox. In some cases it makes sense to deviate from the spec slightly, because of the different security model Deno has.

Here is a list of web platform APIs Deno implements:

- Blob
- BroadcastChannel
- Channel Messaging API
- Compression Streams API
- Console
- DOM CustomEvent, EventTarget and EventListener
- Encoding API
- Fetch API
- FormData
- Location API
- Performance API
- setTimeout, setInterval, clearInterval
- Streams API
- URL
- URLPattern
- URLSearchParams
- Web Crypto API
- Web Storage API
- Web Workers API
- WebSocket

3.4.1 fetch API

3.4.2 Overview

The fetch API can be used to make HTTP requests. It is implemented as specified in the WHATWG fetch spec.

You can find documentation about this API on MDN.

3.4.3 Spec deviations

- The Deno user agent does not have a cookie jar. As such, the set-cookie header on a response is not processed, or filtered from the visible response headers.

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

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

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

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

A couple notes on this example:

- addEventListener and onload/onunload are prefixed with globalThis, but you could also use self or no prefix at all. It is not recommended to use window as a prefix.
- You can use addEventListener and/or onload/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 event handler (imported)
got load event in event handler (main)
got load event in onload function (main)
got onbeforeunload event in event handler (imported)
got onbeforeunload event in event handler (main)
got onbeforeunload event in onbeforeunload function (main)
got unload event in event handler (imported)
got unload event in event handler (main)
got unload event in onunload function (main)
```

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

In other words, you can use addEventListener to register multiple "load" or "unload" event handlers, but only the last defined onload, onbeforeunload, onunload event handlers will be executed. It is preferable to use addEventListener when possible for this reason.

3.3 Permission APIs

Permissions are granted from the CLI when running the deno command. User code will often assume its own set of required permissions, but there is no guarantee during execution that the set of granted permissions will align with this.

In some cases, ensuring a fault-tolerant program requires a way to interact with the permission system at runtime.

3.3.1 Permission descriptors

On the CLI, read permission for /foo/bar is represented as --allow-read=/foo/bar. In runtime JS, it is represented as the following:

```
const desc = { name: "read", path: "/foo/bar" } as const;
```

Other examples:

```
// Global write permission.
```

```
const desc1 = { name: "write" } as const;

// Write permission to `$PWD/foo/bar`.
const desc2 = { name: "write", path: "foo/bar" } as const;

// Global net permission.
const desc3 = { name: "net" } as const;

// Net permission to 127.0.0.1:8000.
const desc4 = { name: "net", host: "127.0.0.1:8000" } as const;

// High-resolution time permission.
const desc5 = { name: "hrtime" } as const;
```

⚠ See PermissionDescriptor in API reference for more details.

3.3.2 Query permissions

Check, by descriptor, if a permission is granted or not.

```
// deno run --allow-read=/foo main.ts

const desc1 = { name: "read", path: "/foo" } as const;
console.log(await Deno.permissions.query(desc1));
// PermissionStatus { state: "granted" }

const desc2 = { name: "read", path: "/foo/bar" } as const;
console.log(await Deno.permissions.query(desc2));
// PermissionStatus { state: "granted" }

const desc3 = { name: "read", path: "/bar" } as const;
console.log(await Deno.permissions.query(desc3));
// PermissionStatus { state: "prompt" }
```

3.3.3 Permission states

A permission state can be either "granted", "prompt" or "denied". Permissions which have been granted from the CLI will query to { state: "granted" }. Those which have not been granted query to { state: "prompt" } by default, while { state: "denied" } reserved for those which have been explicitly refused. This will come up in Request permissions.

3.3.4 Permission strength

The intuitive understanding behind the result of the second query in Query permissions is that read access was granted to /foo and /foo/bar is within /foo so /foo/bar is allowed to be read.

We can also say that desc1 is stronger than desc2. This means that for any set of CLI-granted permissions:

1. If desc1 queries to { state: "granted" } then so must desc2.
2. If desc2 queries to { state: "denied" } then so must desc1.

More examples:

```
const desc1 = { name: "write" } as const;
// is stronger than
const desc2 = { name: "write", path: "/foo" } as const;

const desc3 = { name: "net", host: "127.0.0.1" } as const;
```

```
// is stronger than
const desc4 = { name: "net", host: "127.0.0.1:8000" } as const;
```

3.3.5 Request permissions

Request an ungranted permission from the user via CLI prompt.

```
// deno run main.ts

const desc1 = { name: "read", path: "/foo" } as const;
const status1 = await Deno.permissions.request(desc1);
// ⚠ Deno requests read access to "/foo". Grant? [y/n (y = yes allow, n = no deny)] y
console.log(status1);
// PermissionStatus { state: "granted" }

const desc2 = { name: "read", path: "/bar" } as const;
const status2 = await Deno.permissions.request(desc2);
// ⚠ Deno requests read access to "/bar". Grant? [y/n (y = yes allow, n = no deny)] n
console.log(status2);
// PermissionStatus { state: "denied" }
```

If the current permission state is "prompt", a prompt will appear on the user's terminal asking them if they would like to grant the request. The request for desc1 was granted so its new status is returned and execution will continue as if --allow-read=/foo was specified on the CLI. The request for desc2 was denied so its permission state is downgraded from "prompt" to "denied".

If the current permission state is already either "granted" or "denied", the request will behave like a query and just return the current status. This prevents prompts both for already granted permissions and previously denied requests.

3.3.6 Revoke permissions

Downgrade a permission from "granted" to "prompt".

```
// deno run --allow-read=/foo main.ts

const desc = { name: "read", path: "/foo" } as const;
console.log(await Deno.permissions.revoke(desc));
// PermissionStatus { state: "prompt" }
```

What happens when you try to revoke a permission which is partial to one granted on the CLI?

```
// deno run --allow-read=/foo main.ts

const desc = { name: "read", path: "/foo/bar" } as const;
console.log(await Deno.permissions.revoke(desc));
// PermissionStatus { state: "prompt" }
const cliDesc = { name: "read", path: "/foo" } as const;
console.log(await Deno.permissions.revoke(cliDesc));
// PermissionStatus { state: "prompt" }
```

The CLI-granted permission, which implies the revoked permission, was also revoked.

To understand this behaviour, imagine that Deno stores an internal set of explicitly granted permission descriptors. Specifying --allow-read=/foo,/bar on the CLI initializes this set to:

```
[
  { name: "read", path: "/foo" },
  { name: "read", path: "/bar" },
];
```

Granting a runtime request for { name: "write", path: "/foo" } updates the set to:

```
      }, 1000);
    },
    cancel() {
      clearInterval(timer);
    },
  });
  return new Response(body.pipeThrough(new TextEncoderStream()), {
    headers: {
      "content-type": "text/plain; charset=utf-8",
    },
  });
}
```

i Note the cancel function here. This is called when the client hangs up the connection. This is important to make sure that you handle this case, as otherwise the server will keep queuing up messages forever, and eventually run out of memory.

⚠ Beware that the response body stream is "cancelled" when the client hangs up the connection. Make sure to handle this case. This can surface itself as an error in a write() call on a WritableStream object that is attached to the response body ReadableStream object (for example through a TransformStream).

3.5.4 WebSocket support

Deno can upgrade incoming HTTP requests to a WebSocket. This allows you to handle WebSocket endpoints on your HTTP servers.

To upgrade an incoming Request to a WebSocket you use the Deno.upgradeWebSocket function. This returns an object consisting of a Response and a web standard WebSocket object. This response must be returned from the handler for the upgrade to happen. If this is not done, no WebSocket upgrade will take place.

Because the WebSocket protocol is symmetrical, the WebSocket object is identical to the one that can be used for client side communication. Documentation for it can be found on MDN.

i We are aware that this API can be challenging to use, and are planning to switch to WebSocketStream once it is stabilized and ready for use.

```
function handler(req: Request): Response {
  const upgrade = req.headers.get("upgrade") || "";
  let response, socket: WebSocket;
  try {
    ({ response, socket } = Deno.upgradeWebSocket(req));
  } catch {
    return new Response("request isn't trying to upgrade to websocket.");
  }
  socket.onopen = () => console.log("socket opened");
  socket.onmessage = (e) => {
    console.log("socket message:", e.data);
    socket.send(new Date().toString());
  };
  socket.onerror = (e) => console.log("socket errored:", e);
  socket.onclose = () => console.log("socket closed");
  return response;
}
```

- Deno does not follow the same-origin policy, because the Deno user agent currently does not have the concept of origins, and it does not have a cookie jar. This means Deno does not need to protect against leaking authenticated data cross origin. Because of this Deno does not implement the following sections of the WHATWG fetch specification:
- Section 3.1. 'Origin' header.
- Section 3.2. CORS protocol.
- Section 3.5. CORB.
- Section 3.6. 'Cross-Origin-Resource-Policy' header.
- Atomic HTTP redirect handling.
- The opaqueredirect response type.
- A fetch with a redirect mode of manual will return a basic response rather than an opaqueredirect response.
- The specification is vague on how file: URLs are to be handled. Firefox is the only mainstream browser that implements fetching file: URLs, and even then it doesn't work by default. As of Deno 1.16, Deno supports fetching local files. See the next section for details.
- The request and response header guards are implemented, but unlike browsers do not have any constraints on which header names are allowed.
- The referrer, referrerPolicy, mode, credentials, cache, integrity, keepalive, and window properties and their relevant behaviours in RequestInit are not implemented. The relevant fields are not present on the Request object.
- Request body upload streaming is supported (on HTTP/1.1 and HTTP/2). Unlike the current fetch proposal, the implementation supports duplex streaming.
- The set-cookie header is not concatenated when iterated over in the headers iterator. This behaviour is in the process of being specified.

3.4.4 Fetching local files

As of Deno 1.16, Deno supports fetching file: URLs. This makes it easier to write code that uses the same code path on a server as local, as well as easier to author code that works both with the Deno CLI and Deno Deploy.

Deno only supports absolute file URLs, this means that fetch("./some.json") will not work. It should be noted though that if --location is specified, relative URLs use the --location as the base, but a file: URL cannot be passed as the --location.

To be able to fetch some resource, relative to the current module, which would work if the module is local or remote, you would want to use import.meta.url as the base. For example, something like:

```
const response = await fetch(new URL("./config.json", import.meta.url));
const config = await response.json();
```

Notes on fetching local files:

- Permissions are applied to reading resources, so an appropriate --allow-read permission is needed to be able to read a local file.
- Fetching locally only supports the GET method, and will reject the promise with any other method.
- A file that does not exists simply rejects the promise with a vague TypeError. This is to avoid the potential of fingerprinting attacks.
- No headers are set on the response. Therefore it is up to the consumer to determine things like the content type or content length.
- Response bodies are streamed from the Rust side, so large files are available in chunks, and can be cancelled.

3.4.5 CustomEvent, EventTarget and EventListener

3.4.6 Overview

The DOM Event API can be used to dispatch and listen to events happening in an application. It is implemented as specified in the WHATWG DOM spec.
You can find documentation about this API on MDN.

3.4.7 Spec deviations

- Events do not bubble, because Deno does not have a DOM hierarchy, so there is no tree for Events to bubble/capture through.

3.4.8 Typings

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.

3.5 HTTP Server APIs

As of Deno 1.9 and later, native HTTP server APIs were introduced which allow users to create robust and performant web servers in Deno.
The API tries to leverage as much of the web standards as is possible as well as tries to be simple and straightforward.
i These APIs were stabilized in Deno 1.13 and no longer require --unstable flag.

- A "Hello World" server
- Inspecting the incoming request
- Responding with a response
- WebSocket support
- HTTPS support
- HTTP/2 support
- Automatic body compression
- Lower level APIs

3.5.1 A "Hello World" server

To start a HTTP server on a given port, you can use the serve function from std/http. This function takes a handler function that will be called for each incoming request, and is expected to return a response (or a promise resolving to a response).
Here is an example of a handler function that returns a "Hello, World!" response for each request:

```
function handler(req: Request): Response {
  return new Response("Hello, World!");
}
```

i The handler can also return a Promise<Response>, which means it can be an async function.
To then listen on a port and handle requests you need to call the serve function from the https://deno.land/std@0.156.0/http/server.ts module, passing in the handler as the first argument:

```
import { serve } from "https://deno.land/std@0.156.0/http/server.ts";

serve(handler);
```

By default serve will listen on port 8000, but this can be changed by passing in a port number in the second argument options bag:

```
import { serve } from "https://deno.land/std@0.156.0/http/server.ts";

// To listen on port 4242.
serve(handler, { port: 4242 });
```

This same options bag can also be used to configure some other options, such as the hostname to listen on.

3.5.2 Inspecting the incoming request

Most servers will not answer with the same response for every request. Instead they will change their answer depending on various aspects of the request: the HTTP method, the headers, the path, or the body contents.
The request is passed in as the first argument to the handler function. Here is an example showing how to extract various parts of the request:

```
async function handler(req: Request): Promise<Response> {
  console.log("Method:", req.method);

  const url = new URL(req.url);
  console.log("Path:", url.pathname);
  console.log("Query parameters:", url.searchParams);

  console.log("Headers:", req.headers);

  if (req.body) {
    const body = await req.text();
    console.log("Body:", body);
  }

  return new Response("Hello, World!");
}
```

⚠ Be aware that the req.text() call can fail if the user hangs up the connection before the body is fully received. Make sure to handle this case. Do note this can happen in all methods that read from the request body, such as req.json(), req.formData(), req.arrayBuffer(), req.body.getReader().read(), req.body.pipeTo(), etc.

3.5.3 Responding with a response

Most servers also do not respond with "Hello, World!" to every request. Instead they might respond with different headers, status codes, and body contents (even body streams).
Here is an example of returning a response with a 404 status code, a JSON body, and a custom header:

```
function handler(req: Request): Response {
  const body = JSON.stringify({ message: "NOT FOUND" });
  return new Response(body, {
    status: 404,
    headers: {
      "content-type": "application/json; charset=utf-8",
    },
  });
}
```

Response bodies can also be streams. Here is an example of a response that returns a stream of "Hello, World!" repeated every second:

```
function handler(req: Request): Response {
  let timer: number;
  const body = new ReadableStream({
    async start(controller) {
      timer = setInterval(() => {
        controller.enqueue("Hello, World!\n");
      }
    })();
  } catch (err) {
    // The listener has closed
    break;
  }
}
```

Note that in both cases we are using an IIFE to create an inner function to deal with each connection. If we awaited the HTTP requests in the same function scope as the one we were receiving the connections, we would be blocking accepting additional connections, which would make it seem that our server was "frozen". In practice, it might make more sense to have a separate function all together:

```
async function handle(conn: Deno.Conn) {
  const httpConn = Deno.serveHttp(conn);
  for await (const requestEvent of httpConn) {
    // ... handle requestEvent
  }
}

const server = Deno.listen({ port: 8080 });

for await (const conn of server) {
  handle(conn);
}
```

In the examples from this point on, we will focus on what would occur within an example handle() function and remove the listening and connection "boilerplate".

3.6.3.1 HTTP Requests and Responses

HTTP requests and responses in Deno are essentially the inverse of web standard Fetch API. The Deno HTTP Server API and the Fetch API leverage the Request and Response object classes. So if you are familiar with the Fetch API you just need to flip them around in your mind and now it is a server API.

As mentioned above, a Deno.HttpConn asynchronously yields up Deno.RequestEvents. These request events contain a .request property and a .respondWith() method.

The .request property is an instance of the Request class with the information about the request. For example, if we wanted to know what URL path was being requested, we would do something like this:

```
async function handle(conn: Deno.Conn) {
  const httpConn = Deno.serveHttp(conn);
  for await (const requestEvent of httpConn) {
    const url = new URL(requestEvent.request.url);
    console.log(`path: ${url.pathname}`);
  }
}
```

The .respondWith() method is how we complete a request. The method takes either a Response object or a Promise which resolves with a Response object. Responding with a basic "hello world" would look like this:

```
async function handle(conn: Deno.Conn) {
  const httpConn = Deno.serveHttp(conn);
  for await (const requestEvent of httpConn) {
    await requestEvent.respondWith(
      new Response("hello world", {
```

WebSockets are only supported on HTTP/1.1 for now. The connection the WebSocket was created on can not be used for HTTP traffic after a WebSocket upgrade has been performed.

3.5.5 HTTPS support

i To use HTTPS, you will need a valid TLS certificate and a private key for your server.

To use HTTPS, use serveTls from the https://deno.land/std@0.156.0/http/server.ts module instead of serve. This takes two extra arguments in the options bag: certFile and keyFile. These are paths to the certificate and key files, respectively.

```
import { serveTls } from "https://deno.land/std@0.156.0/http/server.ts";

serveTls(handler, {
  port: 443,
  certFile: "./cert.pem",
  keyFile: "./key.pem",
});
```

3.5.6 HTTP/2 support

HTTP/2 support is "automatic" when using the native APIs with Deno. You just need to create your server, and the server will handle HTTP/1 or HTTP/2 requests seamlessly.

3.5.7 Automatic body compression

As of Deno 1.20, the HTTP server has built in automatic compression of response bodies. When a response is sent to a client, Deno determines if the response body can be safely compressed. This compression happens within the internals of Deno, so it is fast and efficient.

Currently Deno supports gzip and brotli compression. A body is automatically compressed if the following conditions are true:

- The request has an Accept-Encoding header which indicates the requestor supports br for brotli or gzip. Deno will respect the preference of the quality value in the header.
- The response includes a Content-Type which is considered compressible. (The list is derived from jshttp/mime-db with the actual list in the code.)
- The response body is greater than 20 bytes.

When the response body is compressed, Deno will set the Content-Encoding header to reflect the encoding as well as ensure the Vary header is adjusted or added to indicate what request headers affected the response.

3.5.7.1 When is compression skipped?

In addition to the logic above, there are a few other reasons why a response won't be compressed automatically:

- The response body is a stream. Currently only static response bodies are supported. We will add streaming support in the future.
- The response contains a Content-Encoding header. This indicates your server has done some form of encoding already.
- The response contains a Content-Range header. This indicates that your server is responding to a range request, where the bytes and ranges are negotiated outside of the control of the internals to Deno.
- The response has a Cache-Control header which contains a no-transform value. This indicates that your server doesn't want Deno or any downstream proxies to modify the response.

3.5.7.2 What happens to an ETag header?

When you set an ETag that is not a weak validator and the body is compressed, Deno will change this to a weak validator (W/). This is to ensure the proper behavior of clients and downstream proxy services when validating the "freshness" of the content of the response body.

3.5.8 Lower level HTTP server APIs

This chapter focuses only on the high level HTTP server APIs. You should probably use this API, as it handles all of the intricacies of parallel requests on a single connection, error handling, and so on.

If you do want to learn more about the low level HTTP server APIs though, you can read more about them here.

3.6 HTTP Server APIs (Low Level)

As of Deno 1.9 and later, native HTTP server APIs were introduced which allow users to create robust and performant web servers in Deno.

ℹ These APIs were stabilized in Deno 1.13 and no longer require --unstable flag.

⚠ You should probably not be using this API, as it is not easy to get right. Use the higher level API in the standard library instead.

3.6.1 Listening for a connection

In order to accept requests, first you need to listen for a connection on a network port. To do this in Deno, you use Deno.listen():

```
const server = Deno.listen({ port: 8080 });
```

ℹ When supplying a port, Deno assumes you are going to listen on a TCP socket as well as bind to the localhost. You can specify transport: "tcp" to be more explicit as well as provide an IP address or hostname in the hostname property as well.

If there is an issue with opening the network port, Deno.listen() will throw, so often in a server sense, you will want to wrap it in the try ... catch block in order to handle exceptions, like the port already being in use.

You can also listen for a TLS connection (e.g. HTTPS) using Deno.listenTls():

```
const server = Deno.listenTls({
  port: 8443,
  certFile: "localhost.crt",
  keyFile: "localhost.key",
  alpnProtocols: ["h2", "http/1.1"],
});
```

The certFile and keyFile options are required and point to the appropriate certificate and key files for the server. They are relative to the CWD for Deno. The alpnProtocols property is optional, but if you want to be able to support HTTP/2 on the server, you add the protocols here, as the protocol negotiation happens during the TLS negotiation with the client and server.

ℹ Generating SSL certificates is outside of the scope of this documentation. There are many resources on the web which address this.

3.6.2 Handling connections

Once we are listening for a connection, we need to handle the connection. The return value of Deno.listen() or Deno.listenTls() is a Deno.Listener which is an async iterable which yields up Deno.Conn connections as well as provide a couple methods for handling connections.

To use it as an async iterable we would do something like this:

```
const server = Deno.listen({ port: 8080 });

for await (const conn of server) {
  // ...handle the connection...
}
```

Every connection made would yield up a Deno.Conn assigned to conn. Then further processing can be applied to the connection.

There is also the .accept() method on the listener which can be used:

```
const server = Deno.listen({ port: 8080 });
```

```
while (true) {
  try {
    const conn = await server.accept();
    // ... handle the connection ...
  } catch (err) {
    // The listener has closed
    break;
  }
}
```

Whether using the async iterator or the .accept() method, exceptions can be thrown and robust production code should handle these using try ... catch blocks. Especially when it comes to accepting TLS connections, there can be many conditions, like invalid or unknown certificates which can be surfaced on the listener and might need handling in the user code.

A listener also has a .close() method which can be used to close the listener.

3.6.3 Serving HTTP

Once a connection is accepted, you can use Deno.serveHttp() to handle HTTP requests and responses on the connection. Deno.serveHttp() returns a Deno.HttpConn. A Deno.HttpConn is like a Deno.Listener in that requests the connection receives from the client are asynchronously yielded up as a Deno.RequestEvent.

To deal with HTTP requests as async iterable it would look something like this:

```
const server = Deno.listen({ port: 8080 });

for await (const conn of server) {
  (async () => {
    const httpConn = Deno.serveHttp(conn);
    for await (const requestEvent of httpConn) {
      // ... handle requestEvent ...
    }
  })();
}
```

The Deno.HttpConn also has the method .nextRequest() which can be used to await the next request. It would look something like this:

```
const server = Deno.listen({ port: 8080 });

while (true) {
  try {
    const conn = await server.accept();
    (async () => {
      const httpConn = Deno.serveHttp(conn);
      while (true) {
        try {
          const requestEvent = await httpConn.nextRequest();
          // ... handle requestEvent ...
        } catch (err) {
          // the connection has finished
          break;
        }
```

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.

Workers currently do not work in compiled executables.

Use of relative module specifiers in the main worker are only supported with --location <href> passed on the CLI. This is not recommended for portability. You can instead use the URL constructor and import.meta.url to easily create a specifier for some nearby script. Dedicated workers, however, have a location and this capability by default.

```
// 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" });
```

As with regular modules, you can use top-level await in worker modules. However, you should be careful to always register the message handler before the first await, since messages can be lost otherwise. This is not a bug in Deno, it's just an unfortunate interaction of features, and it also happens in all browsers that support module workers.

```
import { delay } from "https://deno.land/std@0.136.0/async/mod.ts";

// First await: waits for a second, then continues running the module.
await delay(1000);

// The message handler is only set after that 1s delay, so some of the messages
// that reached the worker during that second might have been fired when no
// handler was registered.
self.onmessage = (evt) => {
  console.log(evt.data);
};
```

3.9.1 Instantiation 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

```
          status: 200,
        }),
      );
    }
  }
```

Note that we awaited the .respondWith() method. It isn't required, but in practice any errors in processing the response will cause the promise returned from the method to be rejected, like if the client disconnected before all the response could be sent. While there may not be anything your application needs to do, not handling the rejection will cause an "unhandled rejection" to occur which will terminate the Deno process, which isn't so good for a server. In addition, you might want to await the promise returned in order to determine when to do any cleanup from for the request/response cycle.

The web standard Response object is pretty powerful, allowing easy creation of complex and rich responses to a client, and Deno strives to provide a Response object that as closely matches the web standard as possible, so if you are wondering how to send a particular response, checkout the documentation for the web standard Response.

3.6.4 HTTP/2 Support

HTTP/2 support is effectively transparent within the Deno runtime. Typically HTTP/2 is negotiated between a client and a server during the TLS connection setup via ALPN. To enable this, you need to provide the protocols you want to support when you start listening via the alpnProtocols property. This will enable the negotiation to occur when the connection is made. For example:

```
const server = Deno.listenTls({
  port: 8443,
  certFile: "localhost.crt",
  keyFile: "localhost.key",
  alpnProtocols: ["h2", "http/1.1"],
});
```

The protocols are provided in order of preference. In practice, the only two protocols that are supported currently are HTTP/2 and HTTP/1.1 which are expressed as h2 and http/1.1.

Currently Deno does not support upgrading a plain-text HTTP/1.1 connection to an HTTP/2 cleartext connection via the Upgrade header (see: #10275), so therefore HTTP/2 support is only available via a TLS/HTTPS connection.

3.6.5 Serving WebSockets

ℹ These APIs were stabilized in Deno 1.14 and no longer require the --unstable flag.

Deno can upgrade incoming HTTP requests to a WebSocket. This allows you to handle WebSocket endpoints on your HTTP servers.

To upgrade an incoming Request to a WebSocket you use the Deno.upgradeWebSocket function. This returns an object consisting of a Response and a web standard WebSocket object. The returned response should be used to respond to the incoming request using the respondWith method. Only once respondWith is called with the returned response, the WebSocket is activated and can be used.

Because the WebSocket protocol is symmetrical, the WebSocket object is identical to the one that can be used for client side communication. Documentation for it can be found on MDN.

Note: We are aware that this API can be challenging to use, and are planning to switch to WebSocketStream once it is stabilized and ready for use.

```
async function handle(conn: Deno.Conn) {
  const httpConn = Deno.serveHttp(conn);
  for await (const requestEvent of httpConn) {
    await requestEvent.respondWith(handleReq(requestEvent.request));
  }
}

function handleReq(req: Request): Response {
  const upgrade = req.headers.get("upgrade") || "";
  if (upgrade.toLowerCase() != "websocket") {
    return new Response("request isn't trying to upgrade to websocket.");
  }
  const { socket, response } = Deno.upgradeWebSocket(req);
  socket.onopen = () => console.log("socket opened");
  socket.onmessage = (e) => {
    console.log("socket message:", e.data);
    socket.send(new Date().toString());
  };
  socket.onerror = (e) => console.log("socket errored:", e);
  socket.onclose = () => console.log("socket closed");
  return response;
}
```

WebSockets are only supported on HTTP/1.1 for now. The connection the WebSocket was created on can not be used for HTTP traffic after a WebSocket upgrade has been performed.

3.7 Location API

Deno supports the location global from the web. Please read on.

3.7.1 Location flag

There is no "web page" whose URL we can use for a location in a Deno process. We instead allow users to emulate a document location by specifying one on the CLI using the --location flag. It can be a http or https URL.

```
// deno run --location https://example.com/path main.ts

console.log(location.href);
// "https://example.com/path"
```

You must pass --location <href> for this to work. If you don't, any access to the location global will throw an error.

```
// deno run main.ts

console.log(location.href);
// error: Uncaught ReferenceError: Access to "location", run again with --location <href>.
```

Setting location or any of its fields will normally cause navigation in browsers. This is not applicable in Deno, so it will throw in this situation.

```
// deno run --location https://example.com/path main.ts

location.pathname = "./foo";
// error: Uncaught NotSupportedError: Cannot set "location.pathname".
```

3.7.2 Extended usage

On the web, resource resolution (excluding modules) typically uses the value of location.href as the root on which to base any relative URLs. This affects some web APIs adopted by Deno.

3.7.2.1 Fetch API

```
// deno run --location https://api.github.com/ --allow-net main.ts
```

```
const response = await fetch("./orgs/denoland");
// Fetches "https://api.github.com/orgs/denoland".
```

The fetch() call above would throw if the --location flag was not passed, since there is no web-analogous location to base it onto.

3.7.2.2 Worker modules

```
// deno run --location https://example.com/index.html --allow-net main.ts

const worker = new Worker("./workers/hello.ts", { type: "module" });
// Fetches worker module at "https://example.com/workers/hello.ts".
```

3.7.3 Only use if necessary

For the above use cases, it is preferable to pass URLs in full rather than relying on --location. You can manually base a relative URL using the URL constructor if needed.

The --location flag is intended for those who have some specific purpose in mind for emulating a document location and are aware that this will only work at application-level. However, you may also use it to silence errors from a dependency which is frivolously accessing the location global.

3.8 Web Storage API

Deno 1.10 introduced the Web Storage API which provides an API for storing string keys and values. Persisting data works similar to a browser, and has a 10MB storage limit. The global sessionStorage object only persists data for the current execution context, while localStorage persists data from execution to execution.

In a browser, localStorage persists data uniquely per origin (effectively the protocol plus hostname plus port). As of Deno 1.16, Deno has a set of rules to determine what is a unique storage location:

- When using the --location flag, the origin for the location is used to uniquely store the data. That means a location of http://example.com/a.ts and http://example.com/b.ts and http://example.com:80/ would all share the same storage, but https://example.com/ would be different.
- If there is no location specifier, but there is a --config configuration file specified, the absolute path to that configuration file is used. That means deno run --config deno.jsonc a.ts and deno run --config deno.jsonc b.ts would share the same storage, but deno run --config tsconfig.json a.ts would be different.
- If there is no configuration or location specifier, Deno uses the absolute path to the main module to determine what storage is shared. The Deno REPL generates a "synthetic" main module that is based off the current working directory where deno is started from. This means that multiple invocations of the REPL from the same path will share the persisted localStorage data.

This means, unlike versions prior to 1.16, localStorage is always available in the main process.

3.8.1 Example

The following snippet accesses the local storage bucket for the current origin and adds a data item to it using setItem().

```
localStorage.setItem("myDemo", "Deno App");
```

The syntax for reading the localStorage item is as follows:

```
const cat = localStorage.getItem("myDemo");
```

The syntax for removing the localStorage item is as follows:

```
localStorage.removeItem("myDemo");
```

The syntax for removing all the localStorage items is as follows:

```
localStorage.clear();
```

3.9 Workers

Deno supports Web Worker API.

And compile it:

```
// unix
cc -c -o add.o add.c
cc -shared -W -o libadd.so add.o
// Windows
cl /LD add.c /link /EXPORT:add
```

Calling the library from Deno:

```
// ffi.ts

// Determine library extension based on
// your OS.
let libSuffix = "";
switch (Deno.build.os) {
  case "windows":
    libSuffix = "dll";
    break;
  case "darwin":
    libSuffix = "dylib";
    break;
  default:
    libSuffix = "so";
    break;
}

const libName = `./libadd.${libSuffix}`;
// Open library and define exported symbols
const dylib = Deno.dlopen(
  libName,
  {
    "add": { parameters: ["isize", "isize"], result: "isize" },
  } as const,
);

// Call the symbol `add`
const result = dylib.symbols.add(35, 34); // 69

console.log(`Result from external addition of 35 and 34: ${result}`);
```

Run with --allow-ffi and --unstable flag:

```
deno run --allow-ffi --unstable ffi.ts
```

3.10.2 Non-blocking FFI

There are many use cases where users might want to run CPU-bound FFI functions in the background without blocking other tasks on the main thread.

As of Deno 1.15, symbols can be marked nonblocking in Deno.dlopen. These function calls will run on a dedicated blocking thread and will return a Promise resolving to the desired result.

Example of executing expensive FFI calls with Deno:

```
// sleep.c
#ifdef _WIN32
```

```
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
```

3.9.2 Using Deno in worker

Starting in v1.22 the Deno namespace is available in worker scope by default. To enable the namespace in earlier versions pass deno: { namespace: true } when creating a new worker.

main.js

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

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 main.js
hello world
```

Starting in v1.23 Deno.exit() no longer exits the process with the provided exit code. Instead is an alias to self.close(), which causes only the worker to shutdown. This better aligns with the Web platform, as there is no way in the browser for a worker to close the page.

3.9.3 Specifying worker permissions

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

The permissions available for the worker are analogous to the CLI permission flags, meaning every permission enabled there can be disabled at the level of the Worker API. You can find a more detailed description of each of the permission options here.

By default a worker will inherit permissions from the thread it was created in, however in order to allow users to limit the access of this worker we provide the deno.permissions option in the worker API.

- For permissions that support granular access you can pass in a list of the desired resources the worker will have access to, and for those who only have the on/off option you can pass true/false respectively.

```
• const worker = new Worker(new URL("./worker.js", import.meta.url).href, {
•   type: "module",
•   deno: {
•     permissions: {
•       net: [
•                     "https://deno.land/",
•                   ],
•                   read: [
•                     new URL("./file_1.txt", import.meta.url),
•                     new URL("./file_2.txt", import.meta.url),
•                   ],
•                   write: false,
•                 },
•               },
});
```

• Granular access permissions receive both absolute and relative routes as arguments, however take into account that relative routes will be resolved relative to the file the worker is instantiated in, not the path the worker file is currently in

```
•         const worker = new Worker(
•           new URL("./worker/worker.js", import.meta.url).href,
•           {
•             type: "module",
•             deno: {
•               permissions: {
•                 read: [
•                   "/home/user/Documents/deno/worker/file_1.txt",
•                   "./worker/file_2.txt",
•                 ],
•               },
•             },
•           },
);
```

• Both deno.permissions and its children support the option "inherit", which implies it will borrow its parent permissions.

```
•         // This worker will inherit its parent permissions
•         const worker = new Worker(new URL("./worker.js", import.meta.url).href, {
•           type: "module",
•           deno: {
•             permissions: "inherit",
•           },
});
// This worker will inherit only the net permissions of its parent
const worker = new Worker(new URL("./worker.js", import.meta.url).href, {
  type: "module",
  deno: {
    permissions: {
      env: false,
      hrtime: false,
      net: "inherit",
      ffi: false,
      read: false,
```

```
      run: false,
      write: false,
    },
  },
});
```

• Not specifying the deno.permissions option or one of its children will cause the worker to inherit by default.

```
•         // This worker will inherit its parent permissions
•         const worker = new Worker(new URL("./worker.js", import.meta.url).href, {
•           type: "module",
});
// This worker will inherit all the permissions of its parent BUT net
const worker = new Worker(new URL("./worker.js", import.meta.url).href, {
  type: "module",
  deno: {
    permissions: {
      net: false,
    },
  },
});
```

• You can disable the permissions of the worker all together by passing "none" to the deno.permissions option.

```
•         // This worker will not have any permissions enabled
•         const worker = new Worker(new URL("./worker.js", import.meta.url).href, {
•           type: "module",
•           deno: {
•             permissions: "none",
•           },
});
```

3.10 Foreign Function Interface API

As of Deno 1.13 and later, the FFI (foreign function interface) API allows users to call libraries written in native languages that support the C ABIs (Rust, C/C++, C#, Zig, Nim, Kotlin, etc) using Deno.dlopen.

3.10.1 Usage

Here's an example showing how to call a Rust function from Deno:

```
// add.rs
#[no_mangle]
pub extern "C" fn add(a: isize, b: isize) -> isize {
    a + b
}
```

Compile it to a C dynamic library (libadd.so on Linux):

```
rustc --crate-type cdylib add.rs
```

In C you can write it as:

```
// add.c
int add(int a, int b) {
  return a + b;
}
```

```
#[deno_bindgen]
struct Input {
  a: i32,
  b: i32,
}

#[deno_bindgen]
fn mul(input: Input) -> i32 {
  input.a * input.b
}
```

Run `deno_bindgen` to generate bindings. You can now directly import them into Deno:

```
// mul.ts
import { mul } from "./bindings/bindings.ts";
mul({ a: 10, b: 2 }); // 20
```

Any issues related to `deno_bindgen` should be reported at https://github.com/denoland/deno_bindgen/issues

4 Linking to External 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@0.156.0/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@0.156.0/testing/asserts.ts
Download https://deno.land/std@0.156.0/fmt/colors.ts
Download https://deno.land/std@0.156.0/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

1. FAQ

1) How do I import a specific version of a module?

```
#include <Windows.h>
#else
#include <time.h>
#endif

int sleep(unsigned int ms) {
  #ifdef _WIN32
  Sleep(ms);
  #else
  struct timespec ts;
  ts.tv_sec = ms / 1000;
  ts.tv_nsec = (ms % 1000) * 1000000;
  nanosleep(&ts, NULL);
  #endif
}
```

Calling it from Deno:

```
// nonblocking_ffi.ts
const library = Deno.dlopen(
  "./sleep.so",
  {
    sleep: {
      parameters: ["usize"],
      result: "void",
      nonblocking: true,
    },
  } as const,
);

library.symbols.sleep(500).then(() => console.log("After"));
console.log("Before");
```

Result:

```
$ deno run --allow-ffi --unstable unblocking_ffi.ts
Before
After
```

3.10.3 Callbacks

Deno FFI API supports creating C callbacks from JavaScript functions for calling back into Deno from dynamic libraries. An example of how callbacks are created and used is as follows:

```
// callback_ffi.ts
const library = Deno.dlopen(
  "./callback.so",
  {
    set_status_callback: {
      parameters: ["function"],
      result: "void",
    },
    start_long_operation: {
      parameters: [],
      result: "void",
    },
    check_status: {
      parameters: [],
      result: "void",
    },
  } as const,
);

const callback = new Deno.UnsafeCallback(
  {
    parameters: ["u8"],
    result: "void",
  } as const,
  (success: number) => {},
);

// Pass the callback pointer to dynamic library
library.symbols.set_status_callback(callback.pointer);
// Start some long operation that does not block the thread
library.symbols.start_long_operation();

// Later, trigger the library to check if the operation is done.
// If it is, this call will trigger the callback.
library.symbols.check_status();
```

If an UnsafeCallback's callback function throws an error, the error will get propagated up to the function that triggered the callback to be called (above it would be check_status()) and can be caught there. If a callback returning a pointer throws then Deno will set the return value to a nullptr. Other return types are not touched on throw and are thus returned in an undefined state after the callback throws.

UnsafeCallback is not deallocated by default as it can cause use-after-free bugs. To properly dispose of an UnsafeCallback its close() method must be called.

```
const callback = new Deno.UnsafeCallback(
  { parameters: [], result: "void" } as const,
  () => {},
);

// After callback is no longer needed
callback.close();
// It is no longer safe to pass the callback as a parameter.
```

It is also possible for native libraries to setup interrupt handlers and to have those directly trigger the callback. However, this is not recommended and may cause unexpected side-effects and undefined behaviour. Preferably any interrupt handlers would only set a flag that can later be polled similarly to how check_status() is used above.

3.10.4 Supported types

Here's a list of types supported currently by the Deno FFI API.

FFI Type	Deno	C	Rust
i8	number	char / signed char	i8
u8	number	unsigned char	u8
i16	number	short int	i16
u16	number	unsigned short int	u16
i32	number	int / signed int	i32
u32	number	unsigned int	u32
i64	number | bigint	long long int	i64
u64	number | bigint	unsigned long long int	u64
usize	number | bigint	size_t	usize
f32	number | bigint	float	f32
f64	number | bigint	double	f64
void[1]	undefined	void	()
pointer[2]	number | bigint | null	const uint8_t *	*const u8
buffer[3]	TypedArray | null	const uint8_t *	*const u8
function[4]	bigint | null	void (*fun)()	Option<extern "C" fn()>

As of Deno 1.25, the pointer type has been split into a pointer and a buffer type to ensure users take advantage of optimizations for Typed Arrays.

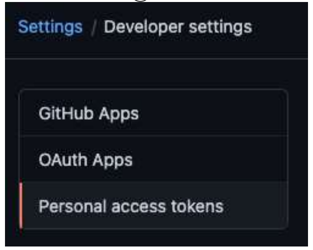
- [1] void type can only be used as a result type.
- [2] pointer type accepts both number and bigint as parameter, while it always returns the latter when used as result type.
- [3] buffer type accepts Typed Arrays as parameter, while it always returns a bigint when used as result type like the pointer type.
- [4] function type parameters and return types are defined using objects, and are passed in as parameters and returned as result types as BigInt pointer values.

3.10.5 deno_bindgen

deno_bindgen is the official tool to simplify glue code generation of Deno FFI libraries written in Rust. It is similar to wasm-bindgen in the Rust WASM ecosystem.

Here's an example showing its usage:

```
// mul.rs
use deno_bindgen::deno_bindgen;
```



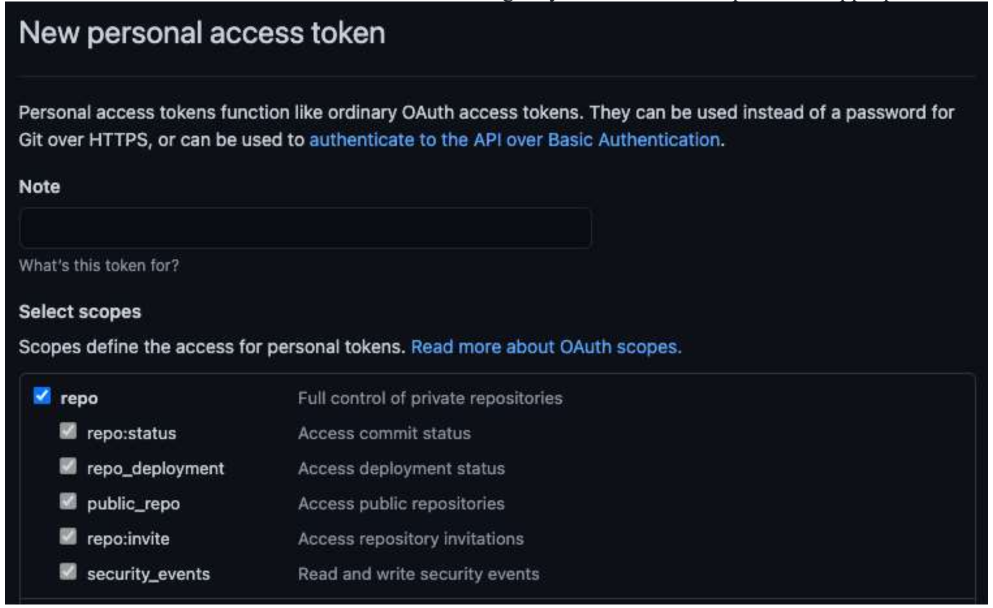
Settings / Developer settings

GitHub Apps

OAuth Apps

Personal access tokens

You would then choose to Generate new token and give your token a description and appropriate access:



New personal access token

Personal access tokens function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to authenticate to the API over Basic Authentication.

Note

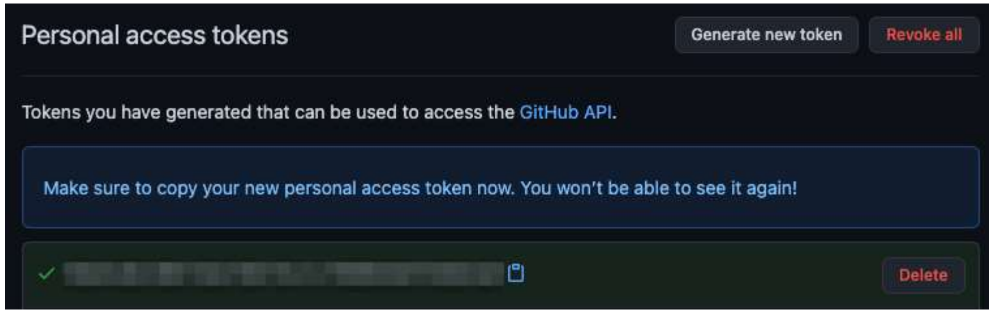
What's this token for?

Select scopes

Scopes define the access for personal tokens. Read more about OAuth scopes.

☑ repo	Full control of private repositories
☑ repo:status	Access commit status
☑ repo_deployment	Access deployment status
☑ public_repo	Access public repositories
☑ repo:invite	Access repository invitations
☑ security_events	Read and write security events

And once created GitHub will display the new token a single time, the value of which you would want to use in the environment variable:



Personal access tokens

Generate new token

Revoke all

Tokens you have generated that can be used to access the GitHub API.

Make sure to copy your new personal access token now. You won't be able to see it again!

✓ [redacted]

Delete

In order to access modules that are contained in a private repository on GitHub, you would want to use the generated token in the DENO_AUTH_TOKENS environment variable scoped to the raw.githubusercontent.com hostname. For example:

```
DENO_AUTH_TOKENS=a1b2c3d4e5f6@raw.githubusercontent.com
```

This should allow Deno to access any modules that the user who the token was issued for has access to. When the token is incorrect, or the user does not have access to the module, GitHub will issue a 404 Not Found status, instead of an unauthorized status. So if you are getting errors that the modules you are

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.

2) 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@0.156.0/testing/asserts.ts" everywhere, you could create a deps.ts file that exports the third-party code:

deps.ts

```
export {
  assert,
  assertEquals,
  assertStringIncludes,
} from "https://deno.land/std@0.156.0/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.

3) 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.

4) 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 using the deno vendor subcommand.

4.1 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 subcommand. It's usage is described below:

4.1.1 To reload everything

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

4.1.2 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 \0.156.0 standard modules:

```
deno cache --reload=https://deno.land/std@0.156.0 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@0.156.0/fs/copy.ts,https://deno.land/std@0.156.0/fmt/colors.ts
my_module.ts
```

4.2 Integrity Checking

4.2.1 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.

4.2.2 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@0.156.0/textproto/mod.ts":
"3118d7a42c03c242c5a49c2ad91c8396110e14acca1324e7aaefd31a999b71a4",
  "https://deno.land/std@0.156.0/io/util.ts":
"ae133d310a0fdcf298cea7bc09a599c49acb616d34e148e263bcb02976f80dee",
  "https://deno.land/std@0.156.0/async/delay.ts":
"35957d585a6e3dd87706858fb1d6b551cb278271b03f52c5a2cb70e65e00c26a",
    ...
}
```

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
```

4.2.3 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.

4.3 Proxies

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

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

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

4.4 Private Modules

There maybe instances where you want to load a remote module that is located in a private repository, like a private repository on GitHub.

Deno supports sending bearer tokens when requesting a remote module. Bearer tokens are the predominant type of access token used with OAuth 2.0 and is broadly supported by hosting services (e.g. GitHub, Gitlab, BitBucket, Cloudsmith, etc.).

4.4.1 DENO_AUTH_TOKENS

The Deno CLI will look for an environment variable named DENO_AUTH_TOKENS to determine what authentication tokens it should consider using when requesting remote modules. The value of the environment variable is in the format of a n number of tokens deliminated by a semi-colon (;) where each token is either:

- a bearer token in the format of {token}@{hostname[:port]}
- basic auth data in the format of {username}:{password}@{hostname[:port]}

For example a single token for would look something like this:

```
DENO_AUTH_TOKENS=a1b2c3d4e5f6@deno.land
```

or

```
DENO_AUTH_TOKENS=username:password@deno.land
```

And multiple tokens would look like this:

```
DENO_AUTH_TOKENS=a1b2c3d4e5f6@deno.land;f1e2d3c4b5a6@example.com:8080,user
name:password@deno.land
```

When Deno goes to fetch a remote module, where the hostname matches the hostname of the remote module, Deno will set the Authorization header of the request to the value of Bearer {token} or Basic {base64EncodedData}. This allows the remote server to recognize that the request is an authorized request tied to a specific authenticated user, and provide access to the appropriate resources and modules on the server.

4.4.2 GitHub

To be able to access private repositories on GitHub, you would need to issue yourself a personal access token. You do this by logging into GitHub and going under Settings -> Developer settings -> Personal access tokens:

want to provide separate entrypoint optimized for use with Deno. As an example, imagine that your package uses `node-fetch`. By providing a conditional `"deno"` export, you can add an entrypoint that doesn't depend on `node-fetch` and instead uses built-in `fetch` API in Deno.

5.1.4 TypeScript support

Currently, the compatibility mode does not support TypeScript.

In the upcoming releases we plan to add support for a `types` field in `package.json`, to automatically lookup types and use them during type checking.

In the long term, we'd like to provide ability to consume TypeScript code authored for the Node runtime.

5.2 The std/node Library

⚠ Starting with v1.15 Deno provides a compatibility mode, that allows to emulate Node environment and consume code authored for Node directly. See Node compatibility mode chapter for details.

The `std/node` part of the Deno standard library is a Node compatibility layer. Its primary focus is providing "polyfills" for Node's built-in modules. It also provides a mechanism for loading CommonJS modules into Deno.

The library is most useful when trying to use your own or private code that was written for Node. If you are trying to consume public npm packages, you are likely to get a better result using a CDN.

5.2.1 Node built-in modules

The standard library provides several "replacement" modules for Node built-in modules. These either replicate the functionality of the built-in or they call the Deno native APIs, returning an interface that is compatible with Node.

The standard library provides modules for the following built-ins:

- assert (partly)
- assert/strict (partly)
- buffer
- console (partly)
- constants
- crypto (partly)
- child_process (partly)
- dns (partly)
- events
- fs (partly)
- fs/promises (partly)
- http (partly)
- module
- net (partly)
- os (partly)
- path
- perf_hooks (partly)
- process (partly)
- querystring
- readline (partly)
- stream
- string_decoder
- sys (partly)
- timers
- timers/promises

trying to access are not found on the command line, check the environment variable settings and the personal access token settings.

In addition, `deno run -L debug` should print out a debug message about the number of tokens that are parsed out of the environment variable. It will print an error message if it feels any of the tokens are malformed. It won't print any details about the tokens for security purposes.

4.5 Import Maps

Deno supports import maps.

You can use import maps with the `--import-map=<FILE>` CLI flag or `importMap` option in the configuration file, the former will take precedence.

Example:

import_map.json

```
{
  "imports": {
    "fmt/": "https://deno.land/std@0.156.0/fmt/"
  }
}
```

color.ts

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

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

Then:

```
$ deno run --import-map=import_map.json color.ts
```

To use your project root for absolute imports:

import_map.json

```
{
  "imports": {
    "/": "./",
    "./": "./"
  }
}
```

main.ts

```
import { MyUtil } from "/util.ts";
```

This causes import specifiers starting with `/` to be resolved relative to the import map's URL or file path.

5 Interoperating with Node and NPM

While Deno is pretty powerful itself, many people will want to leverage code and libraries that are built for Node, in particular the large set of packages available on the NPM registry. In this chapter, we will explore how.

⚠ Starting with v1.15 Deno provides a compatibility mode, that allows to emulate Node environment and consume code authored for Node directly. See Node compatibility mode chapter for details.

The good news is that in many cases, it just works.

There are some foundational things to understand about differences between Node and Deno that can help in understanding what challenges might be faced:

- Current Node supports both CommonJS and ES Modules, while Deno only supports ES Modules. The addition of stabilized ES Modules in Node is relatively recent and most code written for Node is in the CommonJS format.
- Node has quite a few built-in modules that can be imported and they are a fairly expansive set of APIs. On the other hand, Deno focuses on implementing web standards, and where functionality goes beyond the browser, we locate APIs in a single global `Deno` variable/namespace. Lots of code written for Node expects/depends upon these built-in APIs to be available.

- Node has a non-standards based module resolution algorithm, where you can import bare-specifiers (e.g. react or lodash) and Node will look in your local and global node_modules for a path, introspect the package.json and try to see if there is a module named the right way. Deno resolves modules the same way a browser does. For local files, Deno expects a full module name, including the extension. When dealing with remote imports, Deno expects the web server to do any "resolving" and provide back the media type of the code (see the Determining the type of file for more information).
- Node effectively doesn't work without a package.json file. Deno doesn't require an external meta-data file to function or resolve modules.
- Node doesn't support remote HTTP imports. It expects all 3rd party code to be installed locally on your machine using a package manager like npm into the local or global node_modules folder. Deno supports remote HTTP imports (as well as data and blob URLs) and will go ahead and fetch the remote code and cache it locally, similar to the way a browser works.

In order to help mitigate these differences, we will further explore in this chapter:

- Using the std/node modules of the Deno standard library to "polyfill" the built-in modules of Node
- Using CDNs to access the vast majority of npm packages in ways that work under Deno.
- How import maps can be used to provide "bare specifier" imports like Node under Deno, without needing to use a package manager to install packages locally.
- And finally, a general section of frequently asked questions

That being said, some differences cannot be overcome:

- Node has a plugin system that is incompatible with Deno, and Deno will never support Node plugins. If the Node code you want to use requires a "native" Node plugin, it won't work under Deno.
- Node has some built-in modules (e.g. like vm) that are effectively incompatible with the scope of Deno and therefore there aren't easy ways to provide a polyfill of the functionality in Deno.

5.1 Node Compatibility Mode

Starting with v1.15 Deno provides Node compatibility mode that makes it possible to run a subset of programs authored for Node directly in Deno. Compatibility mode can be activated by passing --compat flag in CLI.

⚠ Using compatibility mode currently requires the --unstable flag. If you intend to use CJS modules, the --allow-read flag is needed as well.

⚠ Package management is currently out of scope for Node compatibility mode. For the time being we suggest to keep using your current solution (npm, yarn, pnpm).

5.1.1 Example

eslint is a very popular tool used by most of Node projects. Let's run eslint using Deno in Node compatibility mode. Assuming that eslint is already installed locally (either using npm install eslint or yarn install eslint) we can do so like:

```
$ ls
.eslintrc.json
node_modules
package.json
test.js
test.ts
$ cat test.js
function foo() {}

$ cat test.ts
function bar(): any {}
```

```
$ deno run \
  --compat --unstable \
  --allow-read --allow-write=./ --allow-env \
  node_modules/eslint/bin/eslint.js test.js test.ts

/dev/test.js
  1:10  warning  'foo' is defined but never used   @typescript-eslint/no-unused-vars
  1:16  error    Unexpected empty function 'foo'   @typescript-eslint/no-empty-function

/dev/test.ts
  1:10  warning  'bar' is defined but never used             @typescript-eslint/no-unused-vars
  1:17  warning  Unexpected any. Specify a different type    @typescript-eslint/no-explicit-any
  1:21  error    Unexpected empty function 'bar'              @typescript-eslint/no-empty-function

✖ 5 problems (2 errors, 3 warnings)
```

⚠ Notice that ESLint is run with limited set of permissions. We only give it access to the read from the file system, write to current directory and access environmental variables. Programs run in compatibility mode are subject to Deno's permission model.

5.1.2 How does it work?

When using compatibility mode there Deno does a few things behind the scenes:

- Node globals are set up and made available in the global scope. That means that APIs like process, global, Buffer, setImmediate or clearImmediate are available just like in Node. This is done by executing std/node/global.ts on startup.
- Node built-in modules are set up and made available to import statements and require() calls. Following calls will return appropriate Node modules polyfilled using std/node:
  - import fs from "fs";
  - import os from "node:os";
  - const path = require("path");
  - const http = require("node:http");
- Deno will support Node resolution algorithm so importing packages using "bare" specifiers will work. For details on how module resolution works check Node documentation on CJS and ES modules.

5.1.3 Module resolution

CommonJS resolution is implemented as in Node and there should be no observable differences.
ES module resolution is implemented on top of Deno's regular ESM resolution, leading to a few additional properties compared to Node:

- In addition to file: and data: URL schemes supported in Node; http:, https: and blob: URL schemes will work in the same way if you used Deno without compatibility mode.
- Import maps are supported in the same way if you used Deno without compatibility mode. When resolving "bare" specifiers Deno will first try to resolve them using import map (if one is provided using --import-map flag). Bare specifiers starting with node: prefix are extempt from this rule.
- Deno respects "Conditional exports" field in package.json; in addition to conditions recognized by Node, "deno" condition can be used. This property is useful to the package authors who

While CDNs can make it easy to allow Deno to consume packages and modules from the npm registry, there can still be some things to consider:

- Deno does not (and will not) support Node plugins. If the package requires a native plugin, it won't work under Deno.
- Dependency management can always be a bit of a challenge and a CDN can make it a bit more obfuscated what dependencies are there. You can always use `deno info` with the module or URL to get a full breakdown of how Deno resolves all the code.
- While the Deno friendly CDNs try their best to serve up types with the code for consumption with Deno, lots of types for packages conflict with other packages and/or don't consider Deno, which means you can often get strange diagnostic message when type checking code imported from these CDNs, though skipping type checking will result in the code working perfectly fine. This is a fairly complex topic and is covered in the Types and type declarations section of the manual.

5.4 Using Import Maps

Deno supports import maps which allow you to supply Deno with information about how to resolve modules that overrides the default behavior. Import maps are a web platform standard that is increasingly being included natively in browsers. They are specifically useful with adapting Node code to work well with Deno, as you can use import maps to map "bare" specifiers to a specific module.

When coupled with Deno friendly CDNs import maps can be a powerful tool in managing code and dependencies without need of a package management tool.

5.4.1 Bare and extension-less specifiers

Deno will only load a fully qualified module, including the extension. The import specifier needs to either be relative or absolute. Specifiers that are neither relative or absolute are often called "bare" specifiers. For example "./lodash/index.js" is a relative specifier and `https://cdn.skypack.dev/lodash` is an absolute specifier. Whereas "`lodash`" would be a bare specifier.

Also Deno requires that for local modules, the module to load is fully resolve-able. When an extension is not present, Deno would have to "guess" what the author intended to be loaded. For example does "`./lodash`" mean `./lodash.js`, `./lodash.ts`, `./lodash.tsx`, `./lodash.jsx`, `./lodash/index.js`, `./lodash/index.ts`, `./lodash/index.jsx`, or `./lodash/index.tsx`?

When dealing with remote modules, Deno allows the CDN/web server define whatever semantics around resolution the server wants to define. It just treats a URL, including its query string, as a "unique" module that can be loaded. It expects the CDN/web server to provide it with a valid media/content type to instruct Deno how to handle the file. More information on how media types impact how Deno handles modules can be found in the Determining the type of file section of the manual.

Node does have defined semantics for resolving specifiers, but they are complex, assume unfettered access to the local file system to query it. Deno has chosen not to go down that path.

But, import maps can be used to provide some of the ease of the developer experience if you wish to use bare specifiers. For example, if we want to do the following in our code:

```
import lodash from "lodash";
```

We can accomplish this using an import map, and we don't even have to install the `lodash` package locally. We would want to create a JSON file (for example **import_map.json**) with the following:

```
{
  "imports": {
    "lodash": "https://cdn.skypack.dev/lodash"
  }
}
```

And we would run our program like:

```
> deno run --import-map ./import_map.json example.ts
```

- tty (partly)
- url (partly)
- util (partly)
- worker_threads

Following modules are not yet implemented:

- cluster
- dgram
- http2
- https
- repl
- tls
- vm
- zlib

If you try to run a Node code that requires any of the not implemented modules, please open an issue in https://github.com/denoland/deno_std/issues with example code.

In addition, there is the std/node/global.ts module which provides some of the Node globals like `global`, `process`, and `Buffer`.

If you want documentation for any of the modules, you can simply type `deno doc` and the URL of the module in your terminal:

```
> deno doc https://deno.land/std/fs/move.ts
```

5.2.2 Loading CommonJS modules

Deno only supports natively loading ES Modules, but a lot of Node code is still written in the CommonJS format. As mentioned above, for public npm packages, it is often better to use a CDN to transpile CommonJS modules to ES Modules for consumption by Deno. Not only do you get reliable conversion plus all the dependency resolution and management without requiring a package manager to install the packages on your local machine, you get the advantages of being able to share your code easier without requiring other users to setup some of the Node infrastructure to consume your code with Node.

That being said, the built-in Node module "`module`" provides a function named `createRequire()` which allows you to create a Node compatible `require()` function which can be used to load CommonJS modules, as well as use the same resolution logic that Node uses when trying to load a module. `createRequire()` will also install the Node globals for you.

Example usage would look like this:

```
import { createRequire } from "https://deno.land/std@0.156.0/node/module.ts";

// import.meta.url will be the location of "this" module (like `__filename` in
// Node), and then serve as the root for your "package", where the
// `package.json` is expected to be, and where the `node_modules` will be used
// for resolution of packages.
const require = createRequire(import.meta.url);

// Loads the built-in module Deno "replacement":
const path = require("path");

// Loads a CommonJS module (without specifying the extension):
const cjsModule = require("./my_mod");

// Uses Node resolution in `node_modules` to load the package/module. The
// package would need to be installed locally via a package management tool
// like npm:
const leftPad = require("left-pad");
```

When modules are loaded via the created require(), they are executed in a context which is similar to a Node context, which means that a lot of code written targeting Node will work.

5.3 Packages from CDNs

Because Deno supports remote HTTP modules, and content delivery networks (CDNs) can be powerful tools to transform code, the combination allows an easy way to access code in the npm registry via Deno, usually in a way that works with Deno without any further actions, and often enriched with TypeScript types. In this section we will explore that in detail.

5.3.1 What about deno.land/x/?

The deno.land/x/ is a public registry for code, hopefully code written specifically for Deno. It is a public registry though and all it does is "redirect" Deno to the location where the code exists. It doesn't transform the code in any way. There is a lot of great code on the registry, but at the same time, there is some code that just isn't well maintained (or doesn't work at all). If you are familiar with the npm registry, you know that as well, there are varying degrees of quality.

Because it simply serves up the original published source code, it doesn't really help when trying to use code that didn't specifically consider Deno when authored.

5.3.2 Deno "friendly" CDNs

Deno friendly content delivery networks (CDNs) not only host packages from npm, they provide them in a way that maximizes their integration to Deno. They directly address some of the challenges in consuming code written for Node:

- They provide packages and modules in the ES Module format, irrespective of how they are published on npm.
- They resolve all the dependencies as the modules are served, meaning that all the Node specific module resolution logic is handled by the CDN.
- Often, they inform Deno of type definitions for a package, meaning that Deno can use them to type check your code and provide a better development experience.
- The CDNs also "polyfill" the built-in Node modules, making a lot of code that leverages the built-in Node modules just work.
- The CDNs deal with all the semver matching for packages that a package manager like npm would be required for a Node application, meaning you as a developer can express your 3rd party dependency versioning as part of the URL you use to import the package.

5.3.2.1 esm.sh

esm.sh is a CDN that was specifically designed for Deno, though addressing the concerns for Deno also makes it a general purpose CDN for accessing npm packages as ES Module bundles. esm.sh uses esbuild to take an arbitrary npm package and ensure that it is consumable as an ES Module. In many cases you can just import the npm package into your Deno application:

```
import React from "https://esm.sh/react";

export default class A extends React.Component {
  render() {
    return <div></div>;
  }
}
```

esm.sh supports the use of both specific versions of packages, as well as semver versions of packages, so you can express your dependency in a similar way you would in a package.json file when you import it.

For example, to get a specific version of a package:

```
import React from "https://esm.sh/react@17.0.2";
```

Or to get the latest patch release of a minor release:

```
import React from "https://esm.sh/react@~16.13.0";
```

esm.sh uses the std/node polyfills to replace the built-in modules in Node, meaning that code that uses those built-in modules will have the same limitations and caveats as those modules in std/node.

esm.sh also automatically sets a header which Deno recognizes that allows Deno to be able to retrieve type definitions for the package/module. See Using X-TypeScript-Types header in this manual for more details on how this works.

The CDN is also a good choice for people who develop in mainland China, as the hosting of the CDN is specifically designed to work with "the great firewall of China", as well as esm.sh provides information on self hosting the CDN as well.

Check out the esm.sh homepage for more detailed information on how the CDN can be used and what features it has.

5.3.2.2 Skypack

Skypack.dev is designed to make development overall easier by not requiring packages to be installed locally, even for Node development, and to make it easy to create web and Deno applications that leverage code from the npm registry.

Skypack has a great way of discovering packages in the npm registry, by providing a lot of contextual information about the package, as well as a "scoring" system to try to help determine if the package follows best-practices.

Skypack detects Deno's user agent when requests for modules are received and ensures the code served up is tailored to meet the needs of Deno. The easiest way to load a package is to use the lookup URL for the package:

```
import React from "https://cdn.skypack.dev/react";

export default class A extends React.Component {
  render() {
    return <div></div>;
  }
}
```

Lookup URLs can also contain the semver version in the URL:

```
import React from "https://cdn.skypack.dev/react@~16.13.0";
```

By default, Skypack does not set the types header on packages. In order to have the types header set, which is automatically recognized by Deno, you have to append ?dts to the URL for that package:

```
import { pathToRegexp } from "https://cdn.skypack.dev/path-to-regexp?dts";

const re = pathToRegexp("/path/:id");
```

See Using X-TypeScript-Types header in this manual for more details on how this works.

Skypack docs have a specific page on usage with Deno for more information.

5.3.3 Other CDNs

There are a couple of other CDNs worth mentioning.

5.3.3.1 UNPKG

UNPKG is the most well known CDN for npm packages. For packages that include an ES Module distribution for things like the browsers, many of them can be used directly off of UNPKG. That being said, everything available on UNPKG is available on more Deno friendly CDNs.

5.3.3.2 JSPM

The jspm.io CDN is specifically designed to provide npm and other registry packages as ES Modules in a way that works well with import maps. While it doesn't currently cater to Deno, the fact that Deno can utilize import maps, allows you to use the JSPM.io generator to generate an import-map of all the packages you want to use and have them served up from the CDN.

5.3.4 Considerations

In Deno we handle TypeScript in two major ways. We can type check TypeScript, the default, or you can opt into skipping that checking using the --no-check flag. For example if you had a program you wanted to run, normally you would do something like this:

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

But if you wanted to skip the type checking, you would do something like this:

```
deno run --allow-net --no-check my_server.ts
```

Type checking can take a significant amount of time, especially if you are working on a code base where you are making a lot of changes. We have tried to optimise the type checking, but it still comes at a cost. If you just want to hack at some code, or if you are working in an IDE which is type checking your code as you author it, using --no-check can certainly speed up the process of running TypeScript in Deno.

6.1.3 Determining the type of file

Since Deno supports JavaScript, TypeScript, JSX, TSX modules, Deno has to make a decision about how to treat each of these kinds of files. For local modules, Deno makes this determination based fully on the extension. When the extension is absent in a local file, it is assumed to be JavaScript.

For remote modules, the media type (mime-type) is used to determine the type of the module, where the path of the module is used to help influence the file type, when it is ambiguous what type of file it is.

For example, a .d.ts file and a .ts file have different semantics in TypeScript as well as have different ways they need to be handled in Deno. While we expect to convert a .ts file into JavaScript, a .d.ts file contains no "runnable" code, and is simply describing types (often of "plain" JavaScript). So when we fetch a remote module, the media type for a .ts. and .d.ts file looks the same. So we look at the path, and if we see something that has a path that ends with .d.ts we treat it as a type definition only file instead of "runnable" TypeScript.

6.1.3.1 Supported media types

The following table provides a list of media types which Deno supports when identifying the type of file of a remote module:

Media Type	How File is Handled
application/typescript	TypeScript (with path extension influence)
text/typescript	TypeScript (with path extension influence)
video/vnd.dlna.mpeg-tts	TypeScript (with path extension influence)
video/mp2t	TypeScript (with path extension influence)
application/x-typescript	TypeScript (with path extension influence)
application/javascript	JavaScript (with path extensions influence)
text/javascript	JavaScript (with path extensions influence)
application/ecmascript	JavaScript (with path extensions influence)
text/ecmascript	JavaScript (with path extensions influence)
application/x-javascript	JavaScript (with path extensions influence)
application/node	JavaScript (with path extensions influence)
text/jsx	JSX

If you wanted to manage the versions in the import map, you could do this as well. For example if you were using Skypack CDN, you can used a pinned URL for the dependency in your import map. For example, to pin to lodash version 4.17.21 (and minified production ready version), you would do this:

```
{
  "imports": {
    "lodash": "https://cdn.skypack.dev/pin/lodash@v4.17.21-
K6GEbP02mWFnLA45zAmi/mode=imports,min/optimized/lodash.js"
  }
}
```

5.4.2 Overriding imports

The other situation where import maps can be very useful is the situation where you have tried your best to make something work, but have failed. For example you are using an npm package which has a dependency on some code that just doesn't work under Deno, and you want to substitute another module that "polyfills" the incompatible APIs.

For example, let's say we have a package that is using a version of the built-in "fs" module that we have a local module we want to replace it with when it tries to import it, but we want other code we are loading to use the standard library replacement module for "fs". We would want to create an import map that looked something like this:

```
{
  "imports": {
    "fs": "https://deno.land/std@0.156.0/node/fs.ts"
  },
  "scopes": {
    "https://deno.land/x/example": {
      "fs": "./patched/fs.ts"
    }
  }
}
```

Import maps can be very powerful, check out the official standards README for more information.

5.5 Frequently Asked Questions

5.5.1 Getting errors when type checking like cannot find namespace NodeJS

One of the modules you are using has type definitions that depend upon the NodeJS global namespace, but those types don't include the NodeJS global namespace in their types.

The quickest fix is to skip type checking. You can do this by using the --no-check flag.

Skipping type checking might not be acceptable though. You could try to load the Node types yourself. For example from UNPKG it would look something like this:

```
import type {} from "https://unpkg.com/@types/node/index.d.ts";
```

Or from esm.sh:

```
import type {} from "https://esm.sh/@types/node/index.d.ts";
```

Or from Skypack:

```
import type {} from "https://cdn.skypack.dev/@types/node/index.d.ts";
```

You could also try to provide only specifically what the 3rd party package is missing. For example the package @aws-sdk/client-dynamodb has a dependency on the NodeJS.ProcessEnv type in its type definitions. In one of the modules of your project that imports it as a dependency, you could put something like this in there which will solve the problem:

```
declare global {
  namespace NodeJS {
    type ProcessEnv = Record<string, string>;
  }
}
```

5.5.2 Getting type errors like cannot find document or HTMLElement

The library you are using has dependencies on the DOM. This is common for packages that are designed to run in a browser as well as server-side. By default, Deno only includes the libraries that are directly supported. Assuming the package properly identifies what environment it is running in at runtime it is "safe" to use the DOM libraries to type check the code. For more information on this, check out the Targeting Deno and the Browser section of the manual.

5.6 Node->Deno Cheatsheet

Node	Deno
node file.js	deno run file.ts
npm i -g	deno install
npm i / npm install	n/a [1]
npm run	deno task
eslint	deno lint
prettier	deno fmt
rollup / webpack / etc	deno bundle
package.json	deno.json / deno.jsonc / import_map.json
tsc	deno check [2]
typedoc	deno doc
jest / ava / mocha / tap / etc	deno test
nodemon	deno run/lint/test --watch
nexe / pkg	deno compile
npm explain	deno info
nvm / n / fnm	deno upgrade
tsserver	deno lsp
nyc / c8 / istanbul	deno coverage
benchmarks	deno bench

[1] See Linking to external code, the runtime downloads and caches the code on first use.

[2] Type checking happens automatically, TypeScript compiler is built into the deno binary.

5.7 dnt - Deno to Node Transform

Library authors may want to make their Deno modules available to Node.js users. This is possible by using the dnt build tool.

dnt allows you to develop your Deno module mostly as-is and use a single Deno script to build, type check, and test an npm package in an output directory. Once built, you only need to npm publish the output directory to distribute it to Node.js users.

For more details, see https://github.com/denoland/dnt

6 Using TypeScript

In this chapter we will discuss:

- Overview of TypeScript in Deno
- Configuring TypeScript in Deno
- Types and Type Declarations
- Migrating to/from JavaScript
- FAQs about TypeScript in Deno

6.1 Overview

One of the benefits of Deno is that it treats TypeScript as a first class language, just like JavaScript or Web Assembly, when running code in Deno. What that means is you can run or import TypeScript without installing anything more than the Deno CLI.

But wait a minute, does Deno really run TypeScript? you might be asking yourself. Well, depends on what you mean by run. One could argue that in a browser you don't actually run JavaScript either. The JavaScript engine in the browser translates the JavaScript to a series of operation codes, which it then executes in a sandbox. So it translates JavaScript to something close to assembly. Even Web Assembly goes through a similar translation, in that Web Assembly is architecture agnostic while it needs to be translated into the machine specific operation codes needed for the particular platform architecture it is running on. So when we say TypeScript is a first class language in Deno, we mean that we try to make the user experience in authoring and running TypeScript as easy and straightforward as JavaScript and Web Assembly.

Behind the scenes, we use a combination of technologies, in Rust and JavaScript, to provide that experience.

6.1.1 How does it work?

At a high level, Deno converts TypeScript (as well as TSX and JSX) into JavaScript. It does this via a combination of the TypeScript compiler, which we build into Deno, and a Rust library called swc. When the code has been type checked and transformed, it is stored in a cache, ready for the next run without the need to convert it from its source to JavaScript again.

You can see this cache location by running deno info:

```
> deno info
DENO_DIR location: "/path/to/cache/deno"
Remote modules cache: "/path/to/cache/deno/deps"
TypeScript compiler cache: "/path/to/cache/deno/gen"
```

If you were to look in that cache, you would see a directory structure that mimics that source directory structure and individual .js and .meta files (also potentially .map files). The .js file is the transformed source file while the .meta file contains meta data we want to cache about the file, which at the moment contains a hash of the source module that helps us manage cache invalidation. You might also see a .buildinfo file as well, which is a TypeScript compiler incremental build information file, which we cache to help speed up type checking.

6.1.2 Type Checking

One of the main advantages of TypeScript is that you can make code more type safe, so that what would be syntactically valid JavaScript becomes TypeScript with warnings about being "unsafe".

6.2.3 Using the "lib" property

Deno has several libraries built into it that are not present in other platforms, like tsc. This is what enables Deno to properly check code written for Deno. In some situations though, this automatic behavior can cause challenges, for example like writing code that is intended to also run in a browser. In these situations the "lib" property of a compilerOptions can be used to modify the behavior of Deno when type checking code.

The built-in libraries that are of interest to users:

- "deno.ns" - This includes all the custom Deno global namespace APIs plus the Deno additions to import.meta. This should generally not conflict with other libraries or global types.
- "deno.unstable" - This includes the addition unstable Deno global namespace APIs.
- "deno.window" - This is the "default" library used when checking Deno main runtime scripts. It includes the "deno.ns" as well as other type libraries for the extensions that are built into Deno. This library will conflict with libraries like "dom" and "dom.iterable" that are standard TypeScript libraries.
- "deno.worker" - This is the library used when checking a Deno web worker script. For more information about web workers, check out [Type Checking Web Workers](#).
- "dom.asynciterable" - TypeScript currently does not include the DOM async iterables that Deno implements (plus several browsers), so we have implemented it ourselves until it becomes available in TypeScript.

These are common libraries that Deno doesn't use, but are useful when writing code that is intended to also work in another runtime:

- "dom" - The main browser global library that ships with TypeScript. The type definitions conflict in many ways with "deno.window" and so if "dom" is used, then consider using just "deno.ns" to expose the Deno specific APIs.
- "dom.iterable" - The iterable extensions to the browser global library.
- "scripthost" - The library for the Microsoft Windows Script Host.
- "webworker" - The main library for web workers in the browser. Like "dom" this will conflict with "deno.window" or "deno.worker", so consider using just "deno.ns" to expose the Deno specific APIs.
- "webworker.importscripts" - The library that exposes the importScripts() API in the web worker.
- "webworker.iterable" - The library that adds iterables to objects within a web worker. Modern browsers support this.

6.2.3.1 Targeting Deno and the Browser

A common use case is writing code that works in Deno and the browser: using a conditional check to determine the environment in which the code is executing before using any APIs which are exclusive to one or the other. If that is the case, a common configuration of a compilerOptions would look like this:

```
{
  "compilerOptions": {
    "target": "esnext",
    "lib": ["dom", "dom.iterable", "dom.asynciterable", "deno.ns"]
  }
}
```

This should allow most code to be type checked properly by Deno.

If you expect to run the code in Deno with the --unstable flag, then you will want to add that library to the mix as well:

```
{
  "compilerOptions": {
    "target": "esnext",
```

Media Type	How File is Handled
text/tsx	TSX
text/plain	Attempt to determine that path extension, otherwise unknown
application/octet-stream	Attempt to determine that path extension, otherwise unknown

6.1.4 Strict by default

Deno type checks TypeScript in strict mode by default, and the TypeScript core team recommends strict mode as a sensible default. This mode generally enables features of TypeScript that probably should have been there from the start, but as TypeScript continued to evolve, would be breaking changes for existing code.

6.1.5 Mixing JavaScript and TypeScript

By default, Deno does not type check JavaScript. This can be changed, and is discussed further in [Configuring TypeScript in Deno](#). Deno does support JavaScript importing TypeScript and TypeScript importing JavaScript, in complex scenarios.

An important note though is that when type checking TypeScript, by default Deno will "read" all the JavaScript in order to be able to evaluate how it might have an impact on the TypeScript types. The type checker will do the best it can to figure out what the types are of the JavaScript you import into TypeScript, including reading any JSDoc comments. Details of this are discussed in detail in the [Types and type declarations](#) section.

6.1.6 Diagnostics are terminal

While tsc by default will still emit JavaScript when run while encountering diagnostic (type checking) issues, Deno currently treats them as terminal. It will halt on these warnings, not cache any of the emitted files, and exit the process.

In order to avoid this, you will either need to resolve the issue, utilise the // @ts-ignore or // @ts-expect-error pragmas, or utilise --no-check to bypass type checking all together.

6.1.7 Type resolution

One of the core design principles of Deno is to avoid non-standard module resolution, and this applies to type resolution as well. If you want to utilise JavaScript that has type definitions (e.g. a .d.ts file), you have to explicitly tell Deno about this. The details of how this is accomplished are covered in the [Types and type declarations](#) section.

6.2 [Configuration](#)

TypeScript comes with a lot of different options that can be configured, but Deno strives to make it easy to use TypeScript with Deno. Lots of different options frustrates that goal. To make things easier, Deno configures TypeScript to "just work" and shouldn't require additional configuration.

That being said, Deno does support using a TypeScript configuration file, though like the rest of Deno, the detection and use of a configuration file is not automatic. To use a TypeScript configuration file with Deno, you have to provide a path on the command line. For example:

```
> deno run --config ./deno.json main.ts
```

⚠ Do consider though that if you are creating libraries that require a configuration file, all of the consumers of your modules will require that configuration file too if you distribute your modules as TypeScript. In addition, there could be settings you do in the configuration file that make other TypeScript modules incompatible. Honestly it is best to use the Deno defaults and to think long and hard about using a configuration file.

⚠ Deno v1.14 started supporting a more general configuration file that is no longer confined to specifying TypeScript compiler settings. Using tsconfig.json as a file name will still work, but we recommend to use deno.json or deno.jsonc, as an automatic lookup of this file is planned for an upcoming release.

6.2.1 How Deno uses a configuration file

Deno does not process a TypeScript configuration file like tsc does, as there are lots of parts of a TypeScript configuration file that are meaningless in a Deno context or would cause Deno to not function properly if they were applied.

Deno only looks at the compilerOptions section of a configuration file, and even then it only considers certain compiler options, with the rest being ignored.

Here is a table of compiler options that can be changed, their default in Deno and any other notes about that option:

Option	Default	Notes
allowJs	true	This almost never needs to be changed
allowUnreachableCode	false	
allowUnusedLabels	false	
checkJs	false	If true causes TypeScript to type check JavaScript
jsx	"react"	
jsxFactory	"React.createElement"	
jsxFragmentFactory	"React.Fragment"	
keyofStringsOnly	false	
lib	["deno.window"]	The default for this varies based on other settings in Deno. If it is supplied, it overrides the default. See below for more information.
noFallthroughCasesInSwitch	false	
noImplicitAny	true	
noImplicitReturns	false	
noImplicitThis	true	
noImplicitUseStrict	true	
noStrictGenericChecks	false	
noUnusedLocals	false	
noUnusedParameters	false	
noUncheckedIndexedAccess	false	
reactNamespace	React	
strict	true	
strictBindCallApply	true	
strictFunctionTypes	true	
strictPropertyInitialization	true	
strictNullChecks	true	
suppressExcessPropertyErrors	false	
suppressImplicitAnyIndexErrors	false	
useUnknownInCatchVariables	false	

For a full list of compiler options and how they affect TypeScript, please refer to the TypeScript Handbook.

6.2.2 What an implied tsconfig.json looks like

It is impossible to get tsc to behave like Deno. It is also difficult to get the TypeScript language service to behave like Deno. This is why we have built a language service directly into Deno. That being said, it can be useful to understand what is implied.

If you were to write a tsconfig.json for Deno, it would look something like this:

```
{
  "compilerOptions": {
    "allowJs": true,
    "esModuleInterop": true,
    "experimentalDecorators": true,
    "inlineSourceMap": true,
    "isolatedModules": true,
    "jsx": "react",
    "lib": ["deno.window"],
    "module": "esnext",
    "moduleDetection": "force",
    "strict": true,
    "target": "esnext",
    "useDefineForClassFields": true
  }
}
```

You can't copy paste this into a configuration file and get it to work, specifically because of the built-in type libraries that are custom to Deno which are provided to the TypeScript compiler. This can somewhat be mocked by running deno types on the command line and piping the output to a file and including that in the files as part of the program, removing the "lib" option, and setting the "noLib" option to true.

If you use the --unstable flag, Deno will change the "lib" option to ["deno.window", "deno.unstable"]. If you are trying to load a worker, that is type checked with "deno.worker" instead of "deno.window". See Type Checking Web Workers for more information on this.

6.3.5 Important points

6.3.5.1 Type declaration semantics

Type declaration files (.d.ts files) follow the same semantics as other files in Deno. This means that declaration files are assumed to be module declarations (UMD declarations) and not ambient/global declarations. It is unpredictable how Deno will handle ambient/global declarations.

In addition, if a type declaration imports something else, like another .d.ts file, its resolution follow the normal import rules of Deno. For a lot of the .d.ts files that are generated and available on the web, they may not be compatible with Deno.

To overcome this problem, some solution providers, like the Skypack CDN, will automatically bundle type declarations just like they provide bundles of JavaScript as ESM.

6.3.5.2 Deno Friendly CDNs

There are CDNs which host JavaScript modules that integrate well with Deno.

- Skypack.dev is a CDN which provides type declarations (via the X-TypeScript-Types header) when you append ?dts as a query string to your remote module import statements. For example:

```
import React from "https://cdn.skypack.dev/react?dts";
```

6.3.6 Behavior of JavaScript when type checking

If you import JavaScript into TypeScript in Deno and there are no types, even if you have checkJs set to false (the default for Deno), the TypeScript compiler will still access the JavaScript module and attempt to do some static analysis on it, to at least try to determine the shape of the exports of that module to validate the import in the TypeScript file.

This is usually never a problem when trying to import a "regular" ES module, but in some cases if the module has special packaging, or is a global UMD module, TypeScript's analysis of the module can fail and cause misleading errors. The best thing to do in this situation is provide some form of types using one of the methods mention above.

6.3.6.1 Internals

While it isn't required to understand how Deno works internally to be able to leverage TypeScript with Deno well, it can help to understand how it works.

Before any code is executed or compiled, Deno generates a module graph by parsing the root module, and then detecting all of its dependencies, and then retrieving and parsing those modules, recursively, until all the dependencies are retrieved.

For each dependency, there are two potential "slots" that are used. There is the code slot and the type slot. As the module graph is filled out, if the module is something that is or can be emitted to JavaScript, it fills the code slot, and type only dependencies, like .d.ts files fill the type slot.

When the module graph is built, and there is a need to type check the graph, Deno starts up the TypeScript compiler and feeds it the names of the modules that need to be potentially emitted as JavaScript. During that process, the TypeScript compiler will request additional modules, and Deno will look at the slots for the dependency, offering it the type slot if it is filled before offering it the code slot.

This means when you import a .d.ts module, or you use one of the solutions above to provide alternative type modules for JavaScript code, that is what is provided to TypeScript instead when resolving the module.

6.4 Migrating to/from JavaScript

One of the advantages of Deno is that it treats TypeScript and JavaScript pretty equally. This might mean that transitioning from JavaScript to TypeScript or even from TypeScript to JavaScript is something you want to accomplish. There are several features of Deno that can help with this.

6.4.1 Type checking JavaScript

You might have some JavaScript that you would like to ensure is more type sound but you don't want to go through a process of adding type annotations everywhere.

```
      "lib": [
        "dom",
        "dom.iterable",
        "dom.asynciterable",
        "deno.ns",
        "deno.unstable"
      ]
    }
}
```

Typically when you use the "lib" option in TypeScript, you need to include an "es" library as well. In the case of "deno.ns" and "deno.unstable", they automatically include "esnext" when you bring them in.

The biggest "danger" when doing something like this, is that the type checking is significantly looser, and there is no way to validate that you are doing sufficient and effective feature detection in your code, which may lead to what could be trivial errors becoming runtime errors.

6.2.4 Using the "types" property

The "types" property in "compilerOptions" can be used to specify arbitrary type definitions to include when type checking a program. For more information on this see Using ambient or global types.

6.3 Types and Type Declarations

One of the design principles of Deno is no non-standard module resolution. When TypeScript is type checking a file, it only cares about the types for the file, and the tsc compiler has a lot of logic to try to resolve those types. By default, it expects ambiguous module specifiers with an extension, and will attempt to look for the file under the .ts specifier, then .d.ts, and finally .js (plus a whole other set of logic when the module resolution is set to "node"). Deno deals with explicit specifiers.

This can cause a couple problems though. For example, let's say I want to consume a TypeScript file that has already been transpiled to JavaScript along with a type definition file. So I have mod.js and mod.d.ts. If I try to import mod.js into Deno, it will only do what I ask it to do, and import mod.js, but that means my code won't be as well type checked as if TypeScript was considering the mod.d.ts file in place of the mod.js file.

In order to support this in Deno, Deno has two solutions, of which there is a variation of a solution to enhance support. The two main situations you come across would be:

- As the importer of a JavaScript module, I know what types should be applied to the module.
- As the supplier of the JavaScript module, I know what types should be applied to the module.

The latter case is the better case, meaning you as the provider or host of the module, everyone can consume it without having to figure out how to resolve the types for the JavaScript module, but when consuming modules that you may not have direct control over, the ability to do the former is also required.

6.3.1 Providing types when importing

If you are consuming a JavaScript module and you have either created types (a .d.ts file) or have otherwise obtained the types, you want to use, you can instruct Deno to use that file when type checking instead of the JavaScript file using the @deno-types compiler hint. @deno-types needs to be a single line double slash comment, where when used impacts the next import or re-export statement.

For example if I have a JavaScript modules coolLib.js and I had a separate coolLib.d.ts file that I wanted to use, I would import it like this:

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

When type checking coolLib and your usage of it in the file, the coolLib.d.ts types will be used instead of looking at the JavaScript file.

The pattern matching for the compiler hint is somewhat forgiving and will accept quoted and non-question values for the specifier as well as it accepts whitespace before and after the equals sign.

6.3.2 Providing types when hosting

If you are in control of the source code of the module, or you are in control of how the file is hosted on a web server, there are two ways to inform Deno of the types for a given module, without requiring the importer to do anything special.

6.3.2.1 Using the triple-slash reference directive

Deno supports using the triple-slash reference types directive, which adopts the reference comment used by TypeScript in TypeScript files to include other files and applies it only to JavaScript files.

For example, if I had created coolLib.js and along side of it I had created my type definitions for my library in coolLib.d.ts I could do the following in the coolLib.js file:

```
/// <reference types="./coolLib.d.ts" />

// ... the rest of the JavaScript ...
```

When Deno encounters this directive, it would resolve the ./coolLib.d.ts file and use that instead of the JavaScript file when TypeScript was type checking the file, but still load the JavaScript file when running the program.

i Note this is a repurposed directive for TypeScript that only applies to JavaScript files. Using the triple-slash reference directive of types in a TypeScript file works under Deno as well, but has essentially the same behavior as the path directive.

6.3.2.2 Using X-TypeScript-Types header

Similar to the triple-slash directive, Deno supports a header for remote modules that instructs Deno where to locate the types for a given module. For example, a response for https://example.com/coolLib.js might look something like this:

```
HTTP/1.1 200 OK
Content-Type: application/javascript; charset=UTF-8
Content-Length: 648
X-TypeScript-Types: ./coolLib.d.ts
```

When seeing this header, Deno would attempt to retrieve https://example.com/coolLib.d.ts and use that when type checking the original module.

6.3.3 Using ambient or global types

Overall it is better to use module/UMD type definitions with Deno, where a module expressly imports the types it depends upon. Modular type definitions can express augmentation of the global scope via the declare global in the type definition. For example:

```
declare global {
  var AGlobalString: string;
}
```

This would make AGlobalString available in the global namespace when importing the type definition.

In some cases though, when leveraging other existing type libraries, it may not be possible to leverage modular type definitions. Therefore there are ways to include arbitrary type definitions when type checking programmes.

6.3.3.1 Using a triple-slash directive

This option couples the type definitions to the code itself. By adding a triple-slash types directive near the type of a module, type checking the file will include the type definition. For example:

```
/// <reference types="./types.d.ts" />
```

The specifier provided is resolved just like any other specifier in Deno, which means it requires an extension, and is relative to the module referencing it. It can be a fully qualified URL as well:

```
/// <reference types="https://deno.land/x/pkg@1.0.0/types.d.ts" />
```

6.3.3.2 Using a configuration file

Another option is to use a configuration file that is configured to include the type definitions, by supplying a "types" value to the "compilerOptions". For example:

```
{
  "compilerOptions": {
    "types": [
      "./types.d.ts",
      "https://deno.land/x/pkg@1.0.0/types.d.ts",
      "/Users/me/pkg/types.d.ts"
    ]
  }
}
```

Like the triple-slash reference above, the specifier supplied in the "types" array will be resolved like other specifiers in Deno. In the case of relative specifiers, it will be resolved relative to the path to the config file. Make sure to tell Deno to use this file by specifying --config=path/to/file flag.

6.3.4 Type Checking Web Workers

When Deno loads a TypeScript module in a web worker, it will automatically type check the module and its dependencies against the Deno web worker library. This can present a challenge in other contexts like deno cache, deno bundle, or in editors. There are a couple of ways to instruct Deno to use the worker libraries instead of the standard Deno libraries.

6.3.4.1 Using triple-slash directives

This option couples the library settings with the code itself. By adding the following triple-slash directives near the top of the entry point file for the worker script, Deno will now type check it as a Deno worker script, irrespective of how the module is analyzed:

```
/// <reference no-default-lib="true" />
/// <reference lib="deno.worker" />
```

The first directive ensures that no other default libraries are used. If this is omitted, you will get some conflicting type definitions, because Deno will try to apply the standard Deno library as well. The second instructs Deno to apply the built-in Deno worker type definitions plus dependent libraries (like "esnext").

When you run a deno cache or deno bundle command or use an IDE which uses the Deno language server, Deno should automatically detect these directives and apply the correct libraries when type checking.

The one disadvantage of this, is that it makes the code less portable to other non-Deno platforms like tsc, as it is only Deno which has the "deno.worker" library built into it.

6.3.4.2 Using a configuration file

Another option is to use a configuration file that is configured to apply the library files. A minimal file that would work would look something like this:

```
{
  "compilerOptions": {
    "target": "esnext",
    "lib": ["deno.worker"]
  }
}
```

Then when running a command on the command line, you would need to pass the --config path/to/file argument, or if you are using an IDE which leverages the Deno language server, set the deno.config setting.

If you also have non-worker scripts, you will either need to omit the --config argument, or have one that is configured to meet the needs of your non-worker scripts.

7.1.1 JSX

Created by the React team at Facebook, JSX is a popular DSL (domain specific language) for embedding HTML-like syntax in JavaScript. The TypeScript team also added support for the JSX syntax into TypeScript. JSX has become popular with developers as it allows mixing imperative programming logic with a declarative syntax that looks a lot like HTML.
An example of what a JSX "component" might look like:

```
export function Greeting({ name }) {
  return (
    <div>
      <h1>Hello {name}!</h1>
    </div>
  );
}
```

The main challenge with JSX is that it isn't JavaScript nor is it HTML. It requires transpiling to pure JavaScript before it can be used in a browser, which then has to process that logic to manipulate the DOM in the browser. This is probably less efficient than having a browser render static HTML.
This is where Deno can play a role. Deno supports JSX in both .jsx and .tsx modules and combined with a JSX runtime, Deno can be used to dynamically generate HTML to be sent to a browser client, without the need of shipping the un-transpiled source file, or the JSX runtime library to the browser.
Read the Configuring JSX in Deno section for information on how to customize the configuration of JSX in Deno.

7.1.2 Document Object Model (DOM)

The DOM is the main way a user interface is provided in a browser, and it exposes a set of APIs that allow it to be manipulated via JavaScript, but also allows the direct use of HTML and CSS.
While Deno has a lot of web platform APIs, it does not support most of the DOM APIs related to visual representation. Having said that though, there are a few libraries that provide a lot of the APIs needed to take code that was designed to run in a web browser to be able to run under Deno, in order to generate HTML and CSS which can be shipped to a browser "pre-rendered". We will cover those in the following sections:

- Using LinkeDOM with Deno
- Using deno-dom with Deno
- Using jsdom with Deno

7.1.3 CSS

Cascading Style Sheets (CSS) provide styling for HTML within the DOM. There are tools which make working with CSS in a server side context easier and powerful.

- Parsing and stringifying CSS
- Using Twind with Deno

7.2 Configuring JSX

Deno has built-in support for JSX in both .jsx files and .tsx files. JSX in Deno can be handy for server-side rendering or generating code for consumption in a browser.

7.2.1 Default configuration

The Deno CLI has a default configuration for JSX that is different than the defaults for tsc. Effectively Deno uses the following TypeScript compiler options by default:

```
{
  "compilerOptions": {
    "jsx": "react",
    "jsxFactory": "React.createElement",
    "jsxFragmentFactory": "React.Fragment"
```

Deno supports using the TypeScript type checker to type check JavaScript. You can mark any individual file by adding the check JavaScript pragma to the file:

```
// @ts-check
```

This will cause the type checker to infer type information about the JavaScript code and raise any issues as diagnostic issues.
These can be turned on for all JavaScript files in a program by providing a configuration file with the check JS option enabled:

```
{
  "compilerOptions": {
    "checkJs": true
  }
}
```

And setting the `--config` option on the command line.

6.4.2 Using JSDoc in JavaScript

If you are type checking JavaScript, or even importing JavaScript into TypeScript you can use JSDoc in JavaScript to express more types information than can just be inferred from the code itself. Deno supports this without any additional configuration, you simply need to annotate the code in line with the supported TypeScript JSDoc. For example to set the type of an array:

```
/** @type {string[]} */
const a = [];
```

6.4.3 Skipping type checking

You might have TypeScript code that you are experimenting with, where the syntax is valid but not fully type safe. You can always bypass type checking for a whole program by passing the `--no-check`.
You can also skip whole files being type checked, including JavaScript if you have check JS enabled, by using the no-check pragma:

```
// @ts-nocheck
```

6.4.4 Just renaming JS files to TS files

While this might work in some cases, it has some severe limits in Deno. This is because Deno, by default, runs type checking in what is called strict mode. This means a lot of unclear or ambiguous situations where are not caught in non-strict mode will result in diagnostics being generated, and JavaScript is nothing but unclear and ambiguous when it comes to types.

6.5 Frequently Asked Questions

6.5.1 Can I use TypeScript not written for Deno?

Maybe. That is the best answer, we are afraid. For lots of reasons, Deno has chosen to have fully qualified module specifiers. In part this is because it treats TypeScript as a first class language. Also, Deno uses explicit module resolution, with no magic. This is effectively the same way browsers themselves work, though they don't obviously support TypeScript directly. If the TypeScript modules use imports that don't have these design decisions in mind, they may not work under Deno.
Also, in recent versions of Deno (starting with 1.5), we have started to use a Rust library to do transformations of TypeScript to JavaScript in certain scenarios. Because of this, there are certain situations in TypeScript where type information is required, and therefore those are not supported under Deno. If you are using `tsc` as stand-alone, the setting to use is `"isolatedModules"` and setting it to `true` to help ensure that your code can be properly handled by Deno.
One of the ways to deal with the extension and the lack of Node.js non-standard resolution logic is to use import maps which would allow you to specify "packages" of bare specifiers which then Deno could resolve and load.

6.5.2 What version(s) of TypeScript does Deno support?

Deno is built with a specific version of TypeScript. To find out what this is, type the following on the command line:

> deno --version

The TypeScript version (along with the version of Deno and v8) will be printed. Deno tries to keep up to date with general releases of TypeScript, providing them in the next patch or minor release of Deno.

6.5.3 There was a breaking change in the version of TypeScript that Deno uses, why did you break my program?

We do not consider changes in behavior or breaking changes in TypeScript releases as breaking changes for Deno. TypeScript is a generally mature language and breaking changes in TypeScript are almost always "good things" making code more sound, and it is best that we all keep our code sound. If there is a blocking change in the version of TypeScript and it isn't suitable to use an older release of Deno until the problem can be resolved, then you should be able to use `--no-check` to skip type checking all together.
In addition you can utilize `@ts-ignore` to ignore a specific error in code that you control. You can also replace whole dependencies, using import maps, for situations where a dependency of a dependency isn't being maintained or has some sort of breaking change you want to bypass while waiting for it to be updated.

6.5.4 How do I write code that works in Deno and a browser, but still type checks?

You can do this by using a configuration file with the `--config` option on the command line and adjusting the `"lib"` option in the `"compilerOptions"` in the file. For more information see Targeting Deno and the Browser.

6.5.5 Why are you forcing me to use isolated modules, why can't I use const enums with Deno, why do I need to do export type?

As of Deno 1.5 we defaulted to isolatedModules to `true` and in Deno 1.6 we removed the options to set it back to `false` via a configuration file. The isolatedModules option forces the TypeScript compiler to check and emit TypeScript as if each module would stand on its own. TypeScript has a few type directed emits in the language at the moment. While not allowing type directed emits into the language was a design goal for TypeScript, it has happened anyways. This means that the TypeScript compiler needs to understand the erasable types in the code to determine what to emit, which when you are trying to make a fully erasable type system on top of JavaScript, that becomes a problem.
When people started transpiling TypeScript without `tsc`, these type directed emits became a problem, since the likes of Babel simply try to erase the types without needing to understand the types to direct the emit. In the internals of Deno we have started to use a Rust based emitter which allows us to optionally skip type checking and generates the bundles for things like `deno bundle`. Like all transpilers, it doesn't care about the types, it just tries to erase them. This means in certain situations we cannot support those type directed emits.
So instead of trying to get every user to understand when and how we could support the type directed emits, we made the decision to disable the use of them by forcing the isolatedModules option to `true`.
This means that even when we are using the TypeScript compiler to emit the code, it will follow the same "rules" that the Rust based emitter follows.
This means that certain language features are not supportable. Those features are:

- Re-exporting of types is ambiguous and requires knowing if the source module is exporting runtime code or just type information. Therefore, it is recommended that you use `import type` and `export type` for type only imports and exports. This will help ensure that when the code is emitted, that all the types are erased.
- `const enum` is not supported. `const enums` require type information to direct the emit, as `const enums` get written out as hard coded values. Especially when `const enums` get exported, they are a type system only construct.
- `export =` and `import =` are legacy TypeScript syntax which we do not support.
- Only `declare namespace` is supported. Runtime `namespace` is legacy TypeScript syntax that is not supported.

6.5.6 Why don't you support language service plugins or transformer plugins?

While `tsc` supports language service plugins, Deno does not. Deno does not always use the built-in TypeScript compiler to do what it does, and the complexity of adding support for a language service plugin is not feasible. TypeScript does not support emitter plugins, but there are a few community projects which hack emitter plugins into TypeScript. First, we wouldn't want to support something that TypeScript doesn't support, plus we do not always use the TypeScript compiler for the emit, which would mean we would need to ensure we supported it in all modes, and the other emitter is written in Rust, meaning that any emitter plugin for TypeScript wouldn't be available for the Rust emitter.

6.5.7 How do I combine Deno code with non-Deno code in my IDE?

The Deno language server supports the ability to have a "per-resource" configuration of enabling Deno or not. This also requires a client IDE to support this ability. For Visual Studio Code the official Deno extension supports the vscode concept of multi-root workspace. This means you just need to add folders to the workspace and set the `deno.enable` setting as required on each folder.
For other IDEs, the client extensions needs to support the similar IDE concepts.

7 Using JSX and the DOM

In this chapter we will discuss:

- Overview of JSX and DOM in Deno
- Configuring JSX in Deno
- Using LinkeDOM with Deno
- Using deno-dom with Deno
- Using JSDOM with Deno
- Parsing and Stringifying CSS
- Using Twind with Deno

7.1 Overview

One of the common use cases for Deno is to handle workloads as part of web applications. Because Deno includes many of the browser APIs built-in, there is a lot of power in being able to create isomorphic code that can run both in Deno and in the browser. A powerful workload that can be handled by Deno is performing server side rendering (SSR), where application state is used server side to dynamically render HTML and CSS to be provided to a client.
Server side rendering, when used effectively, can dramatically increase the performance of a web application by offloading heavy calculations of dynamic content to a server process allowing an application developer to minimize the JavaScript and application state that needs to be shipped to the browser.
A web application is generally made up of three key technologies:

- JavaScript
- HTML
- CSS

As well as increasingly, Web Assembly might play a part in a web application.
These technologies combine to allow a developer to build an application in a browser using the web platform. While Deno supports a lot of web platform APIs, it generally only supports web APIs that are usable in a server-side context, but in a client/browser context, the main "display" API is the Document Object Model (or DOM). There are APIs that are accessible to application logic via JavaScript that manipulate the DOM to provide a desired outcome, as well as HTML and CSS are used to structure and style the display of a web application.
In order to facilitate manipulation of the DOM server side and the ability to generate HTML and CSS dynamically, there are some key technologies and libraries that can be used with Deno to achieve this, which we will explore in this chapter.
We will be exploring fairly low-level enabling libraries and technologies, versus a full solution or framework for server side generation of websites.

```
        </button>
      </form>
    </body>
  </html>`);

customElements.define(
  "custom-element",
  class extends HTMLElement {
    connectedCallback() {
      console.log("it works 🥳");
    }
  },
);

document.body.appendChild(document.createElement("custom-element"));

document.toString(); // the string of the document, ready to send to a client
```

7.4 Using deno-dom

deno-dom is an implementation of DOM and HTML parser in Deno. It is implemented in Rust (via Wasm) and TypeScript. There is also a "native" implementation, leveraging the FFI interface.

deno-dom aims for specification compliance, like jsdom and unlike LinkeDOM. Currently, deno-dom is slower than LinkeDOM for things like parsing data structures, but faster at some manipulation operations. Both deno-dom and LinkeDOM are significantly faster than jsdom.

As of deno_dom v0.1.22-alpha supports running on Deno Deploy. So if you want strict standards alignment, consider using deno-dom over LinkeDOM.

7.4.1 Basic example

This example will take a test string and parse it as HTML and generate a DOM structure based on it. It will then query that DOM structure, picking out the first heading it encounters and print out the text content of that heading:

```
import { DOMParser } from "https://deno.land/x/deno_dom/deno-dom-wasm.ts";
import { assert } from "https://deno.land/std@0.156.0/testing/asserts.ts";

const document = new DOMParser().parseFromString(
  `<!DOCTYPE html>
  <html lang="en">
    <head>
      <title>Hello from Deno</title>
    </head>
    <body>
      <h1>Hello from Deno</h1>
      <form>
        <input name="user">
        <button>
          Submit
        </button>
      </form>
    </body>
```

```
  }
}
```

7.2.2 JSX import source

In React 17, the React team added what they called the new JSX transforms. This enhanced and modernized the API for JSX transforms as well as provided a mechanism to automatically import a JSX library into a module, instead of having to explicitly import it or make it part of the global scope. Generally this makes it easier to use JSX in your application.

As of Deno 1.16, initial support for these transforms was added. Deno supports both the JSX import source pragma as well as configuring a JSX import source in a configuration file.

7.2.2.1 JSX runtime

When using the automatic transforms, Deno will try to import a JSX runtime module that is expected to conform to the new JSX API and is located at either `jsx-runtime` or `jsx-dev-runtime`. For example if a JSX import source is configured to `react`, then the emitted code will add this to the emitted file:

```
import { jsx as jsx_ } from "react/jsx-runtime";
```

Deno generally works off explicit specifiers, which means it will not try any other specifier at runtime than the one that is emitted. Which means to successfully load the JSX runtime, `"react/jsx-runtime"` would need to resolve to a module. Saying that, Deno supports remote modules, and most CDNs resolve the specifier easily.

For example, if you wanted to use Preact from the esm.sh CDN, you would use `https://esm.sh/preact` as the JSX import source, and esm.sh will resolve `https://esm.sh/preact/jsx-runtime` as a module, including providing a header in the response that tells Deno where to find the type definitions for Preact.

7.2.2.2 Using the JSX import source pragma

Whether you have a JSX import source configured for your project, or if you are using the default "legacy" configuration, you can add the JSX import source pragma to a .jsx or .tsx module, and Deno will respect it.

The `@jsxImportSource` pragma needs to be in the leading comments of the module. For example to use Preact from esm.sh, you would do something like this:

```
/** @jsxImportSource https://esm.sh/preact */

export function App() {
  return (
    <div>
      <h1>Hello, world!</h1>
    </div>
  );
}
```

7.2.2.3 Using JSX import source in a configuration file

If you want to configure a JSX import source for a whole project, so you don't need to insert the pragma on each module, you can use the `"compilerOptions"` in a configuration file to specify this. For example if you were using Preact as your JSX library from esm.sh, you would configure the following, in the configuration file:

```
{
  "compilerOptions": {
    "jsx": "react-jsx",
    "jsxImportSource": "https://esm.sh/preact"
  }
}
```

7.2.2.4 Using an import map

In situations where the import source plus /jsx-runtime or /jsx-dev-runtime is not resolvable to the correct module, an import map can be used to instruct Deno where to find the module. An import map can also be used to make the import source "cleaner". For example, if you wanted to use Preact from skypack.dev and have skypack.dev include all the type information, you could setup an import map like this:

```
{
  "imports": {
    "preact/jsx-runtime": "https://cdn.skypack.dev/preact/jsx-runtime?dts",
    "preact/jsx-dev-runtime": "https://cdn.skypack.dev/preact/jsx-dev-runtime?dts"
  }
}
```

And then you could use the following pragma:

```
/** @jsxImportSource preact */
```

Or you could configure it in the compiler options:

```
{
  "compilerOptions": {
    "jsx": "react-jsx",
    "jsxImportSource": "preact"
  }
}
```

You would then need to pass the --import-map option on the command line (along with the --config option is using a config file) or set the deno.importMap option (and deno.config option) in your IDE.

7.2.2.5 Current limitations

There are two current limitations of the support of the JSX import source:

- A JSX module that does not have any imports or exports is not transpiled properly when type checking (see: microsoft/TypeScript#46723). Errors will be seen at runtime about _jsx not being defined. To work around the issue, add export {} to the file or use the --no-check flag which will cause the module to be emitted properly.
- Using "jsx-reactdev" compiler option is not supported with --no-emit/bundling/compiling (see: swc-project/swc#2656). Various runtime errors will occur about not being able to load jsx-runtime modules. To work around the issue, use the "jsx-react" compiler option instead, or don't use --no-emit, bundling or compiling.

7.3 Using LinkeDOM

LinkeDOM is a DOM-like namespace to be used in environments, like Deno, which don't implement the DOM.

LinkeDOM focuses on being fast and implementing features useful for server side rendering. It may allow you to do things that are invalid DOM operations. deno-dom and jsdom focus on correctness. While currently deno-dom is slower than LinkeDOM in some cases, both are significantly faster than jsdom, so if you require correctness or features not related to server side rendering, consider deno-dom.

While LinkeDOM works under the Deno CLI, it does not type check. While the provided types work well when using an editor like VSCode, attempting to strictly type check them, like Deno does by default, at runtime, it will fail. This is the same if you were to use tsc to type check the code. The maintainer has indicated they aren't interested in fixing this issue. This means for Deno, you need to use the --no-check=remote to avoid diagnostics stopping the execution of your programme.

LinkedDOM runs under Deno Deploy, along with deno_dom, but jsdom does not.

7.3.1 Basic example

This example will take a test string and parse it as HTML and generate a DOM structure based on it. It will then query that DOM structure, picking out the first heading it encounters and print out the text content of that heading:

```
import { DOMParser } from "https://esm.sh/linkedom";
import { assert } from "https://deno.land/std@0.132.0/testing/asserts.ts";

const document = new DOMParser().parseFromString(
  `<!DOCTYPE html>
  <html lang="en">
    <head>
      <title>Hello from Deno</title>
    </head>
    <body>
      <h1>Hello from Deno</h1>
      <form>
        <input name="user">
        <button>
          Submit
        </button>
      </form>
    </body>
  </html>`,
  "text/html",
);

assert(document);
const h1 = document.querySelector("h1");
assert(h1);

console.log(h1.textContent);
```

7.3.2 Alternative API

For the parseHTML() can be better suited for certain SSR workloads. This is similar to jsdom's JSDOM() function, in the sense it gives you a "sandbox" of a window scope you can use to access API's outside of the scope of the document. For example:

```
import { parseHTML } from "https://esm.sh/linkedom";

const { document, customElements, HTMLElement } = parseHTML(`<!DOCTYPE html>
  <html lang="en">
    <head>
      <title>Hello from Deno</title>
    </head>
    <body>
      <h1>Hello from Deno</h1>
      <form>
        <input name="user">
        <button>
          Submit
```

```
background.value = "white";

console.log(css.stringify(ast));
```

7.6.2 A basic example with deno_css

In this example, we will parse some CSS into an AST and log out the background declaration of the body rule to the console.

```
import * as css from "https://deno.land/x/css@0.3.0/mod.ts";

const ast = css.parse(`
body {
  background: #eee;
  color: #888;
}
`);

const [body] = ast.stylesheet.rules;
const [background] = body.declarations;

console.log(JSON.stringify(background, undefined, "  "));
```

7.7 Using Twind

Twind is a tailwind-in-js solution for using Tailwind. Twind is particularly useful in Deno's server context, where Tailwind and CSS can be easily server side rendered, generating dynamic, performant and efficient CSS while having the usability of styling with Tailwind.

7.7.1 Basic example

In the following example, we will use twind to server side render an HTML page and log it to the console. It demonstrates using the tw function to specify grouped tailwind classes and have it rendered using only the styles specified in the document and no client side JavaScript to accomplish the tailwind-in-js:

```
import { setup, tw } from "https://esm.sh/twind@0.16.16";
import { getStyleTag, virtualSheet } from "https://esm.sh/twind@0.16.16/sheets";

const sheet = virtualSheet();

setup({
  theme: {
    fontFamily: {
      sans: ["Helvetica", "sans-serif"],
      serif: ["Times", "serif"],
    },
  },
  sheet,
});

function renderBody() {
  return `
    <h1 class="${tw`text(3xl blue-500)`}">Hello from Deno</h1>
    <form>
```

```
  </html>`,
  "text/html",
);

assert(document);
const h1 = document.querySelector("h1");
assert(h1);

console.log(h1.textContent);
```

Note: the example uses an unpinned version from deno_land/x, which you likely don't want to do, because the version can change and cause unexpected outcomes. You should use the latest version of available of deno-dom.

7.4.2 Faster startup

Just importing the deno-dom-wasm.ts file bootstraps the Wasm code via top level await. The problem is that top level await blocks the module loading process. Especially with big Wasm projects, it is a lot more performant to initialize the Wasm after module loading is complete.
deno-dom has the solution for that, they provide an alternative version of the library that does not automatically init the Wasm, and requires you to do it in the code:

```
import {
  DOMParser,
  initParser,
} from "https://deno.land/x/deno_dom/deno-dom-wasm-noinit.ts";

(async () => {
  // initialize when you need it, but not at the top level
  await initParser();

  const doc = new DOMParser().parseFromString(
    `<h1>Lorem ipsum dolor...</h1>`,
    "text/html",
  );
})();
```

In addition, using the deno-dom-native.ts (which requires the --allow-ffi flag) will bypass the Wasm startup penalty as well as will not require the init() startup time. This would only work with the Deno CLI and not Deploy.

7.5 Using jsdom

jsdom is a pure JavaScript implementation of many web standards, notably the WHATWG DOM and HTML Standards. It's main goal is to be comprehensive and standards compliant and does not specifically consider performance.
If you are interested in server side rendering, then both deno-dom and LinkeDOM are better choices. If you are trying to run code in a "virtual" browser that needs to be standards based, then it is possible that jsdom is suitable for you.
While jsdom works under the Deno CLI, it does not type check. This means you have to use the --no-check=remote option on the command line to avoid diagnostics stopping the execution of your programme.
Having sound typing in an editor requires some changes to the workflow as well, as the way jsdom types are provided are declared as a global type definition with a globally named module, as well as leveraging the built in types from the built-in DOM libraries.

This means if you want strong typing and intelligent auto-completion in your editor while using the Deno language server, you have to perform some extra steps.

7.5.1.1 Defining an import_map.json

You need to map the bare specifier "jsdom" to the imported version of jsdom. This allows Deno to correctly apply the types to the import in the way they were specified.

```
{
  "jsdom": "https://esm.sh/jsdom"
}
```

7.5.1.2 Setting up a configuration file

You will want to set up a deno.jsonc configuration file in the root of your workspace with both TypeScript library information as well as specifying the import map defined above:

```
{
  "compilerOptions": {
    "lib": [
      "deno.ns",
      "dom",
      "dom.iterable",
      "dom.asynciterable"
    ]
  },
  "importMap": "./import_map.json"
}
```

Note: we are using an unpinned version of jsdom above. You should consider pinning the version to the version you know you want to use.

7.5.2 Basic example

This example will take a test string and parse it as HTML and generate a DOM structure based on it. It will then query that DOM structure, picking out the first heading it encounters and print out the text content of that heading:

```
import { JSDOM } from "jsdom";
import { assert } from "https://deno.land/std@0.132.0/testing/asserts.ts";

const { window: { document } } = new JSDOM(
  `<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Hello from Deno</title>
  </head>
  <body>
    <h1>Hello from Deno</h1>
    <form>
      <input name="user">
      <button>
        Submit
      </button>
    </form>
  </body>
</html>`,
  {
```

```
    url: "https://example.com/",
    referrer: "https://example.org/",
    contentType: "text/html",
    storageQuota: 10000000,
  },
);

const h1 = document.querySelector("h1");
assert(h1);

console.log(h1.textContent);
```

7.6 Parsing CSS

If you want to parse CSS to a abstract syntax tree (AST) then there are two solutions you might want to consider:

- reworkcss/css
- deno_css

reworkcss/css was written originally for Node.js but work well when consumed from a CDN. Importing from esm.sh also automatically combines the type definitions from DefinitelyTyped. It should be noted though that types on DefinitelyTyped are not very good as many union types that should be tagged union types are just union types which leave the types very ambiguous and require a lot of type casting.

Also, if you want to take an AST and generate CSS, reworkcss/css also provides capability to stringify the AST it generates.

deno_css is authored in TypeScript specifically for Deno and is available on deno.land/x.

7.6.1 Basic example with reworkcss/css

In this example, we will parse some CSS into an AST and make a modification to the background declaration of the body rule, to change the color to white. Then we will stringify the modified CSS AST and output it to the console:

```
import * as css from "https://esm.sh/css@3.0.0";
import { assert } from "https://deno.land/std@0.132.0/testing/asserts.ts";

declare global {
  interface AbortSignal {
    reason: unknown;
  }
}

const ast = css.parse(`
body {
  background: #eee;
  color: #888;
}
`);

assert(ast.stylesheet);
const body = ast.stylesheet.rules[0] as css.Rule;
assert(body.declarations);
const background = body.declarations[0] as css.Declaration;
```

```
error: TS2339 [ERROR]: Property 'utime' does not exist on type 'typeof Deno'. 'Deno.utime' is
an unstable API. Did you forget to run with the '--unstable' flag?
    await Deno.utime(dest, statInfo.atime, statInfo.mtime);
                ~~~~~
    at https://deno.land/std@0.156.0/fs/copy.ts:92:16

TS2339 [ERROR]: Property 'utimeSync' does not exist on type 'typeof Deno'.
'Deno.utimeSync' is an unstable API. Did you forget to run with the '--unstable' flag?
    Deno.utimeSync(dest, statInfo.atime, statInfo.mtime);
         ~~~~~~~~~
    at https://deno.land/std@0.156.0/fs/copy.ts:103:10
```

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. Feel free to omit the flag if the particular modules you depend on compile successfully without it.

10 Examples

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

10.1.1 Basic

- Hello World
- Import and Export Modules
- Manage Dependencies
- Fetch Data
- Read and Write Files
- Shebang

10.1.2 Advanced

- Unix cat Program
- HTTP Web Server
- File Server
- TCP echo Server
- Creating a Subprocess
- OS Signals
- File System Events
- Module Metadata

10.2 Hello World

10.2.1 Concepts

- Deno can run JavaScript or TypeScript out of the box with no additional tools or config required.

10.2.2 Overview

Deno is a secure runtime for both JavaScript and TypeScript. As the hello world examples below highlight the same functionality can be created in JavaScript or TypeScript, and Deno will execute both.

10.2.3 JavaScript

In this JavaScript example the message Hello [name] is printed to the console and the code ensures the name provided is capitalized.

Command: deno run hello-world.js

```
/**
```

```
      <input name="user">
      <button class="${tw`text(2xl red-500)`}">
        Submit
      </button>
    </form>
  `;
}

function ssr() {
  sheet.reset();
  const body = renderBody();
  const styleTag = getStyleTag(sheet);

  return `<!DOCTYPE html>
    <html lang="en">
      <head>
        <title>Hello from Deno</title>
        ${styleTag}
      </head>
      <body>
        ${body}
      </body>
    </html>`;
}

console.log(ssr());
```

8 Using WebAssembly

Designed to be used alongside JavaScript to speed up key application components, WebAssembly can have much higher, and more consistent execution speed than JavaScript, similar to C, C++, or Rust. Deno can execute WebAssembly modules with the same interfaces that browsers provide.

In this chapter we will discuss:

1. Using WebAssembly in Deno
2. Using the Streaming WebAssembly APIs
3. Helpful Resources

8.1 Using WebAssembly in Deno

To run WebAssembly in Deno, all you need is a binary to run. WebAssembly is a binary data format. This means that .wasm files are not directly human readable, and not intended to be written by hand. Instead a compiler for a language like Rust, C++, or Go emits .wasm files.

The following binary exports a main function that just returns 42 upon invocation:

```
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);

const main = wasmInstance.exports.main as CallableFunction;
console.log(main().toString());
```

As the code above shows, the following steps need to be performed in order to load WebAssembly in a JavaScript program:

1. Fetching the binary (usually in the form of a .wasm file, though we are using a simple byte array for now)
2. Compiling the binary into a WebAssembly.Module object
3. Instantiating the WebAssembly module

For more complex scenarios you will probably want to write in a programming language that compiles down to WebAssembly instead of hand writing instructions. A number of languages exist that can do this, such as Rust, Go or AssemblyScript. As an example, a Rust program that compiles to the aforementioned bytes would look something like this:

```
pub fn main() -> u32 {    // u32 stands for an unsigned integer using 32 bits of memory.
  42
}
```

Aside from the methods shown in the preceding example, it is also possible to use the streaming methods of the WebAssembly API, as will be shown on the next page.

8.2 Using the Streaming WebAssembly APIs

The most efficient way to fetch, compile and instantiate a WebAssembly module is to use the streaming variants of the WebAssembly API. For example, you can use instantiateStreaming combined with fetch to perform all three steps in one go:

```
const { instance, module } = await WebAssembly.instantiateStreaming(
  fetch("https://wpt.live/wasm/incrementer.wasm"),
);

const increment = instance.exports.increment as (input: number) => number;
console.log(increment(41));
```

Note that the .wasm file must be served with the application/wasm MIME type. If you want to do additional work on the module before instantiation you can instead use compileStreaming:

```
const module = await WebAssembly.compileStreaming(
  fetch("https://wpt.live/wasm/incrementer.wasm"),
);

/* do some more stuff */

const instance = await WebAssembly.instantiate(module);
instance.exports.increment as (input: number) => number;
```

If for some reason you cannot make use of the streaming methods you can fall back to the less efficient compile and instantiate methods. See for example the MDN docs. For a more in-depth look on what makes the streaming methods more performant, see for example this post.

8.3 Helpful Resources

This page contains some further information that is helpful when using and/or developing WebAssembly modules.

8.3.1 WebAssembly API

Further information on all parts of the WebAssembly API can be found on MDN.

8.3.2 Working with Non-Numeric Types

The code samples in this chapter only used numeric types in the WebAssembly modules. To run WebAssembly with more complex types (strings, classes) you will want to use tools that generate type bindings between JavaScript and the language used to compile to WebAssembly.

An example on how to create type bindings between JavaScript and Rust, compiling it into a binary and calling it from a JavaScript program can be found on MDN.

If you plan to do a lot of work with Web APIs in Rust+WebAssembly, you may find the web_sys and js_sys Rust crates useful. web_sys contains bindings to most of the Web APIs that are available in Deno, while js_sys provides bindings to JavaScript's standard, built-in objects.

8.3.3 Optimization

For production builds it can be a good idea to perform optimizations on WebAssembly binaries. If you're mainly serving binaries over networks then optimizing for size can make a real difference, whereas if you're mainly executing WebAssembly on a server to perform computationally intensive tasks, optimizing for speed can be beneficial. You can find a good guide on optimizing (production) builds here. In addition, the rust-wasm group has a list of tools that can be used to optimize and manipulate WebAssembly binaries.

9 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/

9.1.1 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. The standard library is released each time Deno is released.

We strongly suggest to always use imports with pinned version of standard library to avoid unintended changes. For example, rather than linking to the default branch of code, which may change at any time, potentially causing compilation errors or unexpected behavior:

```
// import the latest release, this should be avoided
import { copy } from "https://deno.land/std/fs/copy.ts";
```

instead, use a version of the std library which is immutable and will not change:

```
// imports from v0.156.0 of std, never changes
import { copy } from "https://deno.land/std@0.156.0/fs/copy.ts";
```

9.1.2 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 in the Deno namespace do not exist:

```
// main.ts
import { copy } from "https://deno.land/std@0.156.0/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@0.156.0/fs/copy.ts
Download https://deno.land/std@0.156.0/fs/ensure_dir.ts
Download https://deno.land/std@0.156.0/fs/_util.ts
```

big projects with many dependencies it will become cumbersome and time consuming to update modules if they are all imported individually into individual modules.

The standard practice for solving this problem in Deno is to create a deps.ts file. All required remote dependencies are referenced in this file and the required methods and classes are re-exported. The dependent local modules then reference the deps.ts rather than the remote dependencies. If now for example one remote dependency is used in several files, upgrading to a new version of this remote dependency is much simpler as this can be done just within deps.ts.

With all dependencies centralized in deps.ts, managing these becomes easier. Dev dependencies can also be managed in a separate dev_deps.ts file, allowing clean separation between dev only and production dependencies.

10.4.3 Example

```
/**
 * deps.ts
 *
 * This module re-exports the required methods from the dependant remote Ramda module.
 */
export {
  add,
  multiply,
} from "https://x.nest.land/ramda@0.27.0/source/index.js";
```

In this example the same functionality is created as is the case in the local and remote import examples. But in this case instead of the Ramda module being referenced directly it is referenced by proxy using a local deps.ts module.

Command: deno run example.ts

```
/**
 * example.ts
 */

import { add, multiply } from "./deps.ts";

function totalCost(outbound: number, inbound: number, tax: number): number {
  return multiply(add(outbound, inbound), tax);
}

console.log(totalCost(19, 31, 1.2));
console.log(totalCost(45, 27, 1.15));

/**
 * Output
 *
 * 60
 * 82.8
 */
```

10.5 Fetch Data

10.5.1 Concepts

- Like browsers, Deno implements web standard APIs such as fetch.
- Deno is secure by default, meaning explicit permission must be granted to access the network.

```
 * hello-world.js
 */
function capitalize(word) {
  return word.charAt(0).toUpperCase() + word.slice(1);
}

function hello(name) {
  return "Hello " + capitalize(name);
}

console.log(hello("john"));
console.log(hello("Sarah"));
console.log(hello("kai"));

/**
 * Output:
 *
 * Hello John
 * Hello Sarah
 * Hello Kai
 */
```

10.2.4 TypeScript

This TypeScript example is exactly the same as the JavaScript example above, the code just has the additional type information which TypeScript supports.

The deno run command is exactly the same, it just references a *.ts file rather than a *.js file.

Command: deno run hello-world.ts

```
/**
 * hello-world.ts
 */
function capitalize(word: string): string {
  return word.charAt(0).toUpperCase() + word.slice(1);
}

function hello(name: string): string {
  return "Hello " + capitalize(name);
}

console.log(hello("john"));
console.log(hello("Sarah"));
console.log(hello("kai"));

/**
 * Output:
 *
 * Hello John
 * Hello Sarah
 * Hello Kai
```

```
  */
```

10.3 Import and Export Modules

10.3.1 Concepts

- import allows you to include and use modules held elsewhere, on your local file system or remotely.
- Imports are URLs or file system paths.
- export allows you to specify which parts of your module are accessible to users who import your module.

10.3.2 Overview

Deno by default standardizes the way modules are imported in both JavaScript and TypeScript using the ECMAScript 6 import/export standard.
It adopts browser-like module resolution, meaning that file names must be specified in full. You may not omit the file extension and there is no special handling of index.js.

```
import { add, multiply } from "./arithmetic.ts";
```

Dependencies are also imported directly, there is no package management overhead. Local modules are imported in exactly the same way as remote modules. As the examples show below, the same functionality can be produced in the same way with local or remote modules.

10.3.3 Local Import

In this example the add and multiply functions are imported from a local arithmetic.ts module.
Command: deno run local.ts

```
/**
  * local.ts
  */
import { add, multiply } from "./arithmetic.ts";

function totalCost(outbound: number, inbound: number, tax: number): number {
   return multiply(add(outbound, inbound), tax);
}

console.log(totalCost(19, 31, 1.2));
console.log(totalCost(45, 27, 1.15));

/**
  * Output
  *
  * 60
  * 82.8
  */
```

10.3.4 Remote Import

In the local import example above an add and multiply method are imported from a locally stored arithmetic module. The same functionality can be created by importing add and multiply methods from a remote module too.
In this case the Ramda module is referenced, including the version number. Also note a JavaScript module is imported directly into a TypeScript module, Deno has no problem handling this.
Command: deno run ./remote.ts

```
/**
  * remote.ts
  */
import {
   add,
   multiply,
} from "https://x.nest.land/ramda@0.27.0/source/index.js";

function totalCost(outbound: number, inbound: number, tax: number): number {
   return multiply(add(outbound, inbound), tax);
}

console.log(totalCost(19, 31, 1.2));
console.log(totalCost(45, 27, 1.15));

/**
  * Output
  *
  * 60
  * 82.8
  */
```

10.3.5 Export

In the local import example above the add and multiply functions are imported from a locally stored arithmetic module. To make this possible the functions stored in the arithmetic module must be exported.
To do this just add the keyword export to the beginning of the function signature as is shown below.

```
/**
  * arithmetic.ts
  */
export function add(a: number, b: number): number {
   return a + b;
}

export function multiply(a: number, b: number): number {
   return a * b;
}
```

All functions, classes, constants and variables which need to be accessible inside external modules must be exported. Either by prepending them with the export keyword or including them in an export statement at the bottom of the file.

10.4 Manage Dependencies

10.4.1 Concepts

- Deno uses URLs for dependency management.
- One convention places all these dependent URLs into a local deps.ts file. Functionality is then exported out of deps.ts for use by local modules.
- Continuing this convention, dev only dependencies can be kept in a dev_deps.ts file.
- See also Linking to external code

10.4.2 Overview

In Deno there is no concept of a package manager as external modules are imported directly into local modules. This raises the question of how to manage remote dependencies without a package manager. In

- Modules can be run directly from remote URLs.

10.7.2 Example

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

```
/**
 * cat.ts
 */
import { copy } from "https://deno.land/std@0.156.0/streams/conversion.ts";
for (const filename of Deno.args) {
  const file = await Deno.open(filename);
  await copy(file, Deno.stdout);
  file.close();
}
```

To run the program:

```
deno run --allow-read https://deno.land/std@0.156.0/examples/cat.ts /etc/passwd
```

10.8 HTTP Web Server

10.8.1 Concepts

- Use Deno's integrated HTTP server to run your own web server.

10.8.2 Overview

With just a few lines of code you can run your own HTTP web server with control over the response status, request headers and more.

10.8.3 Sample web server

In this example, the user-agent of the client is returned to the client:

webserver.ts:

```
// Start listening on port 8080 of localhost.
const server = Deno.listen({ port: 8080 });
console.log(`HTTP webserver running.   Access it at:   http://localhost:8080/`);

// Connections to the server will be yielded up as an async iterable.
for await (const conn of server) {
  // In order to not be blocking, we need to handle each connection individually
  // without awaiting the function
  serveHttp(conn);
}

async function serveHttp(conn: Deno.Conn) {
  // This "upgrades" a network connection into an HTTP connection.
  const httpConn = Deno.serveHttp(conn);
  // Each request sent over the HTTP connection will be yielded as an async
  // iterator from the HTTP connection.
  for await (const requestEvent of httpConn) {
    // The native HTTP server uses the web standard `Request` and `Response`
    // objects.
    const body = `Your user-agent is:\n\n${
      requestEvent.request.headers.get("user-agent") ?? "Unknown"
    }`;
    // The requestEvent's `.respondWith()` method is how we send the response
```

- See also: Deno's permissions model.

10.5.2 Overview

When building any sort of web application developers will usually need to retrieve data from somewhere else on the web. This works no differently in Deno than in any other JavaScript application, just call the fetch() method. For more information on fetch read the MDN documentation.

The exception with Deno occurs when running a script which makes a call over the web. Deno is secure by default which means access to IO (Input / Output) is prohibited. To make a call over the web Deno must be explicitly told it is ok to do so. This is achieved by adding the --allow-net flag to the deno run command.

10.5.3 Example

Command: deno run --allow-net fetch.ts

```
/**
 * Output: JSON Data
 */
const jsonResponse = await fetch("https://api.github.com/users/denoland");
const jsonData = await jsonResponse.json();
console.log(jsonData);

/**
 * Output: HTML Data
 */
const textResponse = await fetch("https://deno.land/");
const textData = await textResponse.text();
console.log(textData);

/**
 * Output: Error Message
 */
try {
  await fetch("https://does.not.exist/");
} catch (error) {
  console.log(error);
}
```

10.5.4 Files and Streams

Like in browsers, sending and receiving large files is possible thanks to the Streams API. The standard library's streams module can be used to convert a Deno file into a writable or readable stream.

Command: deno run --allow-read --allow-write --allow-net fetch_file.ts

```
/**
 * Receiving a file
 */
import { writableStreamFromWriter } from "https://deno.land/std@0.156.0/streams/mod.ts";

const fileResponse = await fetch("https://deno.land/logo.svg");

if (fileResponse.body) {
  const file = await Deno.open("./logo.svg", { write: true, create: true });
  const writableStream = writableStreamFromWriter(file);
  await fileResponse.body.pipeTo(writableStream);
}

/**
 * Sending a file
 */
import { readableStreamFromReader } from "https://deno.land/std@0.156.0/streams/mod.ts";

const file = await Deno.open("./logo.svg", { read: true });
const readableStream = readableStreamFromReader(file);

await fetch("https://example.com/", {
  method: "POST",
  body: readableStream,
});
```

10.6 Read and Write Files

10.6.1 Concepts

- Deno's runtime API provides the Deno.readTextFile and Deno.writeTextFile asynchronous functions for reading and writing entire text files.
- Like many of Deno's APIs, synchronous alternatives are also available. See Deno.readTextFileSync and Deno.writeTextFileSync.
- Use --allow-read and --allow-write permissions to gain access to the file system.

10.6.2 Overview

Interacting with the filesystem to read and write files is a common requirement. Deno provides a number of ways to do this via the standard library and the Deno runtime API.

As highlighted in the Fetch Data example Deno restricts access to Input / Output by default for security reasons. Therefore when interacting with the filesystem the --allow-read and --allow-write flags must be used with the deno run command.

10.6.3 Reading a text file

The Deno runtime API makes it possible to read text files via the Deno.readTextFile() method, it just requires a path string or URL object. The method returns a promise which provides access to the file's text data.

Command: deno run --allow-read read.ts

```
/**
 * read.ts
 */
const text = await Deno.readTextFile("./people.json");
console.log(text);

/**
 * Output:
 *
 * [
 *    {"id": 1, "name": "John", "age": 23},
 *    {"id": 2, "name": "Sandra", "age": 51},
 *    {"id": 5, "name": "Devika", "age": 11}
 * ]
```

```
 */
```

10.6.4 Writing a text file

The Deno runtime API allows developers to write text to files via the Deno.writeTextFile() method. It just requires a file path and text string. The method returns a promise which resolves when the file was successfully written.

To run the command the --allow-write flag must be supplied to the deno run command.

Command: deno run --allow-write write.ts

```
/**
 * write.ts
 */
await Deno.writeTextFile("./hello.txt", "Hello World!");
console.log("File written to ./hello.txt");

/**
 * Output: File written to ./hello.txt
 */
```

By combining Deno.writeTextFile and JSON.stringify you can easily write serialized JSON objects to a file. This example uses synchronous Deno.writeTextFileSync, but this can also be done asynchronously using await Deno.writeTextFile.

To execute the code the deno run command needs the write flag.

Command: deno run --allow-write write.ts

```
/**
 * write.ts
 */
function writeJson(path: string, data: object): string {
  try {
    Deno.writeTextFileSync(path, JSON.stringify(data));

    return "Written to " + path;
  } catch (e) {
    return e.message;
  }
}

console.log(writeJson("./data.json", { hello: "World" }));

/**
 * Output: Written to ./data.json
 */
```

10.7 Unix cat Program

An Implementation of the Unix "cat" Program

10.7.1 Concepts

- Use the Deno runtime API to output the contents of a file to the console.
- Deno.args accesses the command line arguments.
- Deno.open is used to get a handle to a file.
- copy is used to transfer data from the file to the output stream.
- Files should be closed when you are finished with them

```
/**
 * subprocess_simple.ts
 */

// define command used to create the subprocess
const cmd = ["echo", "hello"];

// create subprocess
const p = Deno.run({ cmd });

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

Note: If using Windows, the command above would need to be written differently because echo is not an executable binary (rather, it is a built-in shell command):

```
// define command used to create the subprocess
const cmd = ["cmd", "/c", "echo hello"];
```

Run it:

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

10.11.3 Security

The --allow-run permission is required for creation of a subprocess. Be aware that subprocesses are not run in a Deno sandbox and therefore have the same permissions as if you were to run the command from the command line yourself.

10.11.4 Communicating with subprocesses

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

```
/**
 * subprocess.ts
 */
const fileNames = Deno.args;

const p = Deno.run({
  cmd: [
    "deno",
    "run",
    "--allow-read",
    "https://deno.land/std@0.156.0/examples/cat.ts",
    ...fileNames,
  ],
  stdout: "piped",
  stderr: "piped",
});

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

// Reading the outputs closes their pipes
const rawOutput = await p.output();
```

```
      // back to the client.
      requestEvent.respondWith(
        new Response(body, {
          status: 200,
        }),
      );
    }
}
```

Then run this with:

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

Then navigate to http://localhost:8080/ in a browser.

10.8.3.1 Using the std/http library

ℹ Since the stabilization of native HTTP bindings in ^1.13.x, std/http now supports a native HTTP server from ^0.107.0. The legacy server module was removed in 0.117.0.

webserver.ts:

```
import { serve } from "https://deno.land/std@0.156.0/http/server.ts";

const port = 8080;

const handler = (request: Request): Response => {
  const body = `Your user-agent is:\n\n${
    request.headers.get("user-agent") ?? "Unknown"
  }`;

  return new Response(body, { status: 200 });
};

console.log(`HTTP webserver running. Access it at: http://localhost:8080/`);
await serve(handler, { port });
```

Then run this with:

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

10.9 File Server

10.9.1 Concepts

- Use Deno.open to read a file's content in chunks.
- Transform a Deno file into a ReadableStream.
- Use Deno's integrated HTTP server to run your own file server.

10.9.2 Overview

Sending files over the network is a common requirement. As seen in the Fetch Data example, because files can be of any size, it is important to use streams in order to prevent having to load entire files into memory.

10.9.3 Example

Command: deno run --allow-read --allow-net file_server.ts

```
// Start listening on port 8080 of localhost.
const server = Deno.listen({ port: 8080 });
console.log("File server running on http://localhost:8080/");

for await (const conn of server) {
    handleHttp(conn).catch(console.error);
}

async function handleHttp(conn: Deno.Conn) {
    const httpConn = Deno.serveHttp(conn);
    for await (const requestEvent of httpConn) {
        // Use the request pathname as filepath
        const url = new URL(requestEvent.request.url);
        const filepath = decodeURIComponent(url.pathname);

        // Try opening the file
        let file;
        try {
            file = await Deno.open("." + filepath, { read: true });
        } catch {
            // If the file cannot be opened, return a "404 Not Found" response
            const notFoundResponse = new Response("404 Not Found", { status: 404 });
            await requestEvent.respondWith(notFoundResponse);
            return;
        }

        // Build a readable stream so the file doesn't have to be fully loaded into
        // memory while we send it
        const readableStream = file.readable;

        // Build and send the response
        const response = new Response(readableStream);
        await requestEvent.respondWith(response);
    }
}
```

10.9.4 Using the std/http file server

The Deno standard library provides you with a file server so that you don't have to write your own.
To use it, first install the remote script to your local file system. This will install the script to the Deno installation root's bin directory, e.g. /home/alice/.deno/bin/file_server.

```
deno install --allow-net --allow-read https://deno.land/std@0.156.0/http/file_server.ts
```

You can now run the script with the simplified script name. Run it:

```
$ file_server .
Downloading https://deno.land/std@0.156.0/http/file_server.ts...
[...]
HTTP server listening on http://0.0.0.0:4507/
```

Now go to http://0.0.0.0:4507/ in your web browser to see your local directory contents.
The complete list of options are available via:

```
file_server --help
```

Example output:

```
Deno File Server
    Serves a local directory in HTTP.
  INSTALL:
    deno install --allow-net --allow-read https://deno.land/std/http/file_server.ts
  USAGE:
    file_server [path] [options]
  OPTIONS:
    -h, --help              Prints help information
    -p, --port <PORT>       Set port
    --cors                  Enable CORS via the "Access-Control-Allow-Origin" header
    --host       <HOST>     Hostname (default is 0.0.0.0)
    -c, --cert <FILE>       TLS certificate file (enables TLS)
    -k, --key    <FILE>     TLS key file (enables TLS)
    --no-dir-listing        Disable directory listing
    All TLS options are required when one is provided.
```

10.10 TCP echo Server

10.10.1 Concepts

- Listening for TCP port connections with Deno.listen.
- Use copy to take inbound data and redirect it to be outbound data.

10.10.2 Example

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

```
/**
 * echo_server.ts
 */
import { copy } from "https://deno.land/std@0.156.0/streams/conversion.ts";
const listener = Deno.listen({ port: 8080 });
console.log("listening on 0.0.0.0:8080");
for await (const conn of listener) {
    copy(conn, conn).finally(() => conn.close());
}
```

Run with:

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

To test it, try sending data to it with netcat (Linux/MacOS only). Below 'hello world' is sent over the connection, which is then echoed back to the user:

```
$ 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.

10.11 Creating a Subprocess

10.11.1 Concepts

- Deno is capable of spawning a subprocess via Deno.run.
- --allow-run permission is required to spawn a subprocess.
- Spawned subprocesses do not run in a security sandbox.
- Communicate with the subprocess via the stdin, stdout and stderr streams.
- Use a specific shell by providing its path/name and its string input switch,

e.g. Deno.run({cmd: ["bash", "-c", "ls -la"]});

10.11.2 Simple example

This example is the equivalent of running 'echo hello' from the command line.

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

10.14 Module Metadata

10.14.1 Concepts

- import.meta can provide information on the context of the module.
- The boolean import.meta.main will let you know if the current module is the program entry point.
- The string import.meta.url will give you the URL of the current module.
- The import.meta.resolve allows you to resolve specifier relative to the current module. This function takes into account an import map (if one was provided on startup).
- The string Deno.mainModule will give you the URL of the main module entry point, i.e. the module invoked by the deno runtime.

10.14.2 Example

The example below uses two modules to show the difference between import.meta.url, import.meta.main and Deno.mainModule. In this example, module_a.ts is the main module entry point:

```
/**
 * module_b.ts
 */
export function outputB() {
  console.log("Module B's import.meta.url", import.meta.url);
  console.log("Module B's mainModule url", Deno.mainModule);
  console.log(
    "Is module B the main module via import.meta.main?",
    import.meta.main,
  );
}
/**
 * module_a.ts
 */
import { outputB } from "./module_b.ts";

function outputA() {
  console.log("Module A's import.meta.url", import.meta.url);
  console.log("Module A's mainModule url", Deno.mainModule);
  console.log(
    "Is module A the main module via import.meta.main?",
    import.meta.main,
  );
  console.log("Resolved specifier for ./module_b.ts", import.meta.resolve("./module_b.ts"));
}

outputA();
console.log("");
```

```
const rawError = await p.stderrOutput();

if (code === 0) {
  await Deno.stdout.write(rawOutput);
} else {
  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)
```

10.11.5 Piping to files

This example is the equivalent of running yes &> ./process_output in bash.

```
/**
 * subprocess_piping_to_file.ts
 */

import {
  readableStreamFromReader,
  writableStreamFromWriter,
} from "https://deno.land/std@0.156.0/streams/conversion.ts";
import { mergeReadableStreams } from "https://deno.land/std@0.156.0/streams/merge.ts";

// create the file to attach the process to
const file = await Deno.open("./process_output.txt", {
  read: true,
  write: true,
  create: true,
});
const fileWriter = await writableStreamFromWriter(file);

// start the process
const process = Deno.run({
  cmd: ["yes"],
  stdout: "piped",
  stderr: "piped",
});
```

```
// example of combining stdout and stderr while sending to a file
const stdout = readableStreamFromReader(process.stdout);
const stderr = readableStreamFromReader(process.stderr);
const joined = mergeReadableStreams(stdout, stderr);
// returns a promise that resolves when the process is killed/closed
joined.pipeTo(fileWriter).then(() => console.log("pipe join done"));

// manually stop process "yes" will never end on its own
setTimeout(async () => {
  process.kill("SIGINT");
}, 100);
```

Run it:

```
$ deno run --allow-run ./subprocess_piping_to_file.ts
```

10.12 OS Signals

Windows only supports listening for SIGINT and SIGBREAK as of Deno v1.23.

10.12.1 Concepts

- Deno.addSignalListener() can be used to capture and monitor OS signals.
- Deno.removeSignalListener() can be used to stop watching the signal.

10.12.2 Set up an OS signal listener

APIs for handling OS signals are modelled after already familiar addEventListener and removeEventListener APIs.

⚠ Note that listening for OS signals doesn't prevent event loop from finishing, ie. if there are no more pending async operations the process will exit.

You can use Deno.addSignalListener() function for handling OS signals:

```
/**
 * add_signal_listener.ts
 */
console.log("Press Ctrl-C to trigger a SIGINT signal");

Deno.addSignalListener("SIGINT", () => {
  console.log("interrupted!");
  Deno.exit();
});

// Add a timeout to prevent process exiting immediately.
setTimeout(() => {}, 5000);
```

Run with:

```
deno run add_signal_listener.ts
```

You can use Deno.removeSignalListener() function to unregister previously added signal handler.

```
/**
 * signal_listeners.ts
 */
console.log("Press Ctrl-C to trigger a SIGINT signal");

const sigIntHandler = () => {
  console.log("interrupted!");
  Deno.exit();
```

```
};
Deno.addSignalListener("SIGINT", sigIntHandler);

// Add a timeout to prevent process existing immediately.
setTimeout(() => {}, 5000);

// Stop listening for a signal after 1s.
setTimeout(() => {
  Deno.removeSignalListener("SIGINT", sigIntHandler);
}, 1000);
```

Run with:

```
deno run signal_listeners.ts
```

10.12.3 Async iterator example

If you prefer to handle signals using an async iterator, you can use signal() API available in deno_std:

```
/**
 * async_iterator_signal.ts
 */
import { signal } from "https://deno.land/std@0.156.0/signal/mod.ts";

const sig = signal("SIGUSR1", "SIGINT");

// Add a timeout to prevent process exiting immediately.
setTimeout(() => {}, 5000);

for await (const _ of sig) {
  console.log("interrupt or usr1 signal received");
}
```

Run with:

```
deno run async_iterator_signal.ts
```

10.13 File System Events

10.13.1 Concepts

- Use Deno.watchFs to watch for file system events.
- Results may vary between operating systems.

10.13.2 Example

To poll for file system events in the current directory:

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

Run with:

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

Now try adding, removing and modifying files in the same directory as watcher.ts.

```
    hostname: "localhost",
    port: 5432,
  });
  await client.connect();

  // provide a step name and function
  await t.step("insert user", async () => {
    const users = await client.queryObject<User>(
      "INSERT INTO users (name) VALUES ('Deno') RETURNING *",
    );
    assertEquals(users.rows.length, 1);
    assertEquals(users.rows[0].name, "Deno");
  });

  // or provide a test definition
  await t.step({
    name: "insert book",
    fn: async () => {
      const books = await client.queryObject<Book>(
        "INSERT INTO books (name) VALUES ('The Deno Manual') RETURNING *",
      );
      assertEquals(books.rows.length, 1);
      assertEquals(books.rows[0].title, "The Deno Manual");
    },
    ignore: false,
    // these default to the parent test or step's value
    sanitizeOps: true,
    sanitizeResources: true,
    sanitizeExit: true,
  });

  // nested steps are also supported
  await t.step("update and delete", async (t) => {
    await t.step("update", () => {
      // even though this test throws, the outer promise does not reject
      // and the next test step will run
      throw new Error("Fail.");
    });

    await t.step("delete", () => {
      // ...etc...
    });
  });

  // steps return a value saying if they ran or not
  const testRan = await t.step({
    name: "copy books",
```

```
outputB();
```

If module_a.ts is located in /home/alice/deno then the output of deno run --allow-read module_a.ts is:

```
Module A's import.meta.url file:///home/alice/deno/module_a.ts
Module A's mainModule url file:///home/alice/deno/module_a.ts
Is module A the main module via import.meta.main? true
Resolved specifier for ./module_b.ts file:///home/alice/deno/module_b.ts

Module B's import.meta.url file:///home/alice/deno/module_b.ts
Module B's mainModule url file:///home/alice/deno/module_a.ts
Is module B the main module via import.meta.main? false
```

10.15 Shebang

Making Scripts Executable With #!Shebang

10.15.1 Concepts

- Deno.env provides the environment variables.
- env runs a program in a modified environment.

10.15.2 Overview

Making Deno scripts executable can be useful if you want to make, for example, small tools.

10.15.3 Example

In this program we give the context permission to access the environment variables and print the Deno installation path.

```
#!/usr/bin/env -S deno run --allow-env

/**
 *  shebang.ts
 */

const path = Deno.env.get("DENO_INSTALL");

console.log("Deno Install Path:", path);
```

10.15.3.1 Permissions

You may require to give the script execution permissions.

10.15.3.1.1 Linux

```
sudo chmod +x shebang.ts
```

10.15.3.2 Execute

Start the script by calling it like any other command:

```
./shebang.ts
```

10.15.4 Details

- A shebang has to be placed in the first line.
- -S splits the command into arguments.

11 Testing

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

1. Quickstart

Firstly, let's create a file url_test.ts and register a test case using Deno.test() function.

```
// url_test.ts
import { assertEquals } from "https://deno.land/std@0.156.0/testing/asserts.ts";
```

```
Deno.test("url test", () => {
  const url = new URL("./foo.js", "https://deno.land/");
  assertEquals(url.href, "https://deno.land/foo.js");
});
```

Secondly, run the test using deno test subcommand.

```
$ deno test url_test.ts
running 1 test from file:///dev/url_test.js
test url test ... ok (2ms)

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out (9ms)
```

2. Writing tests

To define a test you need to register it with a call to Deno.test API. There are multiple overloads of this API to allow for greatest flexibility and easy switching between the forms (eg. when you need to quickly focus a single test for debugging, using only: true option):

```
import { assertEquals } from "https://deno.land/std@0.156.0/testing/asserts.ts";

// Compact form: name and function
Deno.test("hello world #1", () => {
  const x = 1 + 2;
  assertEquals(x, 3);
});

// Compact form: named function.
Deno.test(function helloWorld3() {
  const x = 1 + 2;
  assertEquals(x, 3);
});

// Longer form: test definition.
Deno.test({
  name: "hello world #2",
  fn: () => {
    const x = 1 + 2;
    assertEquals(x, 3);
  },
});

// Similar to compact form, with additional configuration as a second argument.
Deno.test("hello world #4", { permissions: { read: true } }, () => {
  const x = 1 + 2;
  assertEquals(x, 3);
});

// Similar to longer form, with test function as a second argument.
Deno.test(
  { name: "hello world #5", permissions: { read: true } },
```

```
  () => {
    const x = 1 + 2;
    assertEquals(x, 3);
  },
);

// Similar to longer form, with a named test function as a second argument.
Deno.test({ permissions: { read: true } }, function helloWorld6() {
  const x = 1 + 2;
  assertEquals(x, 3);
});
```

1) 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@0.156.0/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");
  }
});
```

2) Test steps

The test steps API provides a way to report distinct steps within a test and do setup and teardown code within that test.

```
import { assertEquals } from "https://deno.land/std@0.156.0/testing/asserts.ts";
import { Client } from "https://deno.land/x/postgres@v0.15.0/mod.ts";

interface User {
  id: number;
  name: string;
}

interface Book {
  id: number;
  title: string;
}

Deno.test("database", async (t) => {
  const client = new Client({
    user: "user",
    database: "test",
export function foo(fn) {
  fn();
}
```

This way, we can call foo(bar) in the application code or wrap a spy function around bar and call foo(spy) in the testing code:

```
import sinon from "https://cdn.skypack.dev/sinon";
import { assertEquals } from "https://deno.land/std@0.156.0/testing/asserts.ts";
import { bar, foo } from "./my_file.js";

Deno.test("calls bar during execution of foo", () => {
  // create a test spy that wraps 'bar'
  const spy = sinon.spy(bar);

  // call function 'foo' and pass the spy as an argument
  foo(spy);

  assertEquals(spy.called, true);
  assertEquals(spy.getCalls().length, 1);
});
```

If you prefer not to add additional parameters for testing purposes only, you can also use sinon to wrap a method on an object instead. In other JavaScript environments bar might have been accessible via a global such as window and callable via sinon.spy(window, "bar"), but in Deno this will not work and instead you can export an object with the functions to be tested. This means rewriting my_file.js to something like this:

```
// my_file.js
function bar() {/*...*/}

export const funcs = {
  bar,
};

// 'foo' no longer takes a parameter, but calls 'bar' from an object
export function foo() {
  funcs.bar();
}
```

And then import in a test file:

```
import sinon from "https://cdn.skypack.dev/sinon";
import { assertEquals } from "https://deno.land/std@0.156.0/testing/asserts.ts";
import { foo, funcs } from "./my_file.js";

Deno.test("calls bar during execution of foo", () => {
  // create a test spy that wraps 'bar' on the 'funcs' object
  const spy = sinon.spy(funcs, "bar");

  // call function 'foo' without an argument
  foo();
```

```
    fn: () => {
      // ...etc...
    },
    ignore: true, // was ignored, so will return `false`
  });

  // steps can be run concurrently if sanitizers are disabled on sibling steps
  const testCases = [1, 2, 3];
  await Promise.all(testCases.map((testCase) =>
    t.step({
      name: `case ${testCase}`,
      fn: async () => {
        // ...etc...
      },
      sanitizeOps: false,
      sanitizeResources: false,
      sanitizeExit: false,
    })
  ));

  client.end();
});
```

Outputs:

```
test database ...
  test insert user ... ok (2ms)
  test insert book ... ok (14ms)
  test update and delete ...
    test update ... FAILED (17ms)
      Error: Fail.
          at <stack trace omitted>
    test delete ... ok (19ms)
  FAILED (46ms)
  test copy books ... ignored (0ms)
  test case 1 ... ok (14ms)
  test case 2 ... ok (14ms)
  test case 3 ... ok (14ms)
FAILED (111ms)
```

Notes:

1. Test steps **must be awaited** before the parent test/step function resolves or you will get a runtime error.
2. Test steps cannot be run concurrently unless sanitizers on a sibling step or parent test are disabled.
3. If nesting steps, ensure you specify a parameter for the parent step.
4. ` Deno.test("my test", async (t) => {`
5. `   await t.step("step", async (t) => {`
6. `     // note the `t` used here is for the parent step and not the outer `Deno.test``
7. `     await t.step("sub-step", () => {`
8. `     });`

```
9.                });
});
```

Nested test steps

3. Running tests

To run the test, call deno test with the file that contains your test function. You can also omit the file name, in which case all tests in the current directory (recursively) that match the glob {*_,*.,}test.{ts, tsx, mts, js, mjs, jsx, cjs, cts} will be run. If you pass a directory, all files in the directory that match this glob will be run.

The glob expands to:

- files named test.{ts, tsx, mts, js, mjs, jsx, cjs, cts},
- or files ending with .test.{ts, tsx, mts, js, mjs, jsx, cjs, cts},
- or files ending with _test.{ts, tsx, mts, js, mjs, jsx, cjs, cts}

```
# Run all tests in the current directory and all sub-directories
deno test

# Run all tests in the util directory
deno test util/

# Run just my_test.ts
deno test my_test.ts

# Run test modules in parallel
deno test --parallel
```

Note that starting in Deno v1.24, some test options can be configured via [a configuration file](#).

⚠ If you want to pass additional CLI arguments to the test files use -- to inform Deno that remaining arguments are scripts arguments.

```
# Pass additional arguments to the test file
deno test my_test.ts -- -e --foo --bar
```

deno test uses the same permission model as deno run and therefore will require, for example, --allow-write to write to the file system during testing.

To see all runtime options with deno test, you can reference the command line help:

```
deno help test
```

4. Filtering

There are a number of options to filter the tests you are running.

1) Command line filtering

Tests can be run individually or in groups using the command line --filter option.

The filter flags accept a string or a pattern as value.

Assuming the following tests:

```
Deno.test({ name: "my-test", fn: myTest });
Deno.test({ name: "test-1", fn: test1 });
Deno.test({ name: "test-2", fn: test2 });
```

This command will run all of these tests because they all contain the word "test".

```
deno test --filter "test" tests/
```

On the flip side, the following command uses a pattern and will run the second and third tests.

```
deno test --filter "/test-*\d/" tests/
```

To let Deno know that you want to use a pattern, wrap your filter with forward-slashes like the JavaScript syntactic sugar for a REGEX.

2) Test definition filtering

Within the tests themselves, you have two options for filtering.

A. Filtering out (Ignoring these 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() {
    // do MacOS feature here
  },
});
```

B. Filtering in (Only run these tests)

Sometimes you may be in the middle of a problem within a large test class and you would like to focus on just that test and ignore the rest for now. For this you can use the only option to tell the test framework to only run tests with this set to true. Multiple tests can set this option. While the test run will report on the success or failure of each test, the overall test run will always fail if any test is flagged with only, as this is a temporary measure only which disables nearly all of your tests.

```
Deno.test({
  name: "Focus on this test only",
  only: true,
  fn() {
    // test complicated stuff here
  },
});
```

5. Failing fast

If you have a long-running test suite and wish for it to stop on the first failure, you can specify the --fail-fast flag when running the suite.

```
deno test --fail-fast
```

6. Integration with testing libraries

Deno's test runner works with popular testing libraries like [Chai](#), [Sinon.JS](#) or [fast-check](#).

For example integration see:

- https://deno.land/std@0.156.0/testing/chai_example.ts
- https://deno.land/std@0.156.0/testing/sinon_example.ts
- https://deno.land/std@0.156.0/testing/fast_check_example.ts

1) Example: spying on a function with Sinon

Test spies are function stand-ins that are used to assert if a function's internal behavior matches expectations. Sinon is a widely used testing library that provides test spies and can be used in Deno by importing it from a CDN, such as Skypack:

```
import sinon from "https://cdn.skypack.dev/sinon";
```

Say we have two functions, foo and bar and want to assert that bar is called during execution of foo. There are a few ways to achieve this with Sinon, one is to have function foo take another function as a parameter:

```
// my_file.js
export function bar() {/*...*/}
```

```
Deno.test("Test Assert Array Contains", () => {
  assertArrayIncludes([1, 2, 3], [1]);
  assertArrayIncludes([1, 2, 3], [1, 2]);
  assertArrayIncludes(Array.from("Hello World"), Array.from("Hello"));
});
```

11.1.5 Regex

You can assert regular expressions via assertMatch() and assertNotMatch() assertions.

```
Deno.test("Test Assert Match", () => {
  assertMatch("abcdefghi", new RegExp("def"));

  const basicUrl = new RegExp("^https?://[a-z.]+.com$");
  assertMatch("https://www.google.com", basicUrl);
  assertMatch("http://facebook.com", basicUrl);
});

Deno.test("Test Assert Not Match", () => {
  assertNotMatch("abcdefghi", new RegExp("jkl"));

  const basicUrl = new RegExp("^https?://[a-z.]+.com$");
  assertNotMatch("https://deno.land/", basicUrl);
});
```

11.1.6 Object

Use assertObjectMatch to check that a JavaScript object matches a subset of the properties of an object.

```
// Simple subset
assertObjectMatch(
  { foo: true, bar: false },
  {
    foo: true,
  },
);
```

11.1.7 Throws

There are two ways to assert whether something throws an error in
Deno, assertThrows() and assertRejects(). Both assertions allow you to check an Error has been thrown, the type of error thrown and what the message was.

The difference between the two assertions is assertThrows() accepts a standard function
and assertRejects() accepts a function which returns a Promise.

The assertThrows() assertion will check an error has been thrown, and optionally will check the thrown error is of the correct type, and assert the error message is as expected.

```
Deno.test("Test Assert Throws", () => {
  assertThrows(
    () => {
      throw new Error("Panic!");
    },
    Error,
    "Panic!",
  );
});
```

```
  assertEquals(spy.called, true);
  assertEquals(spy.getCalls().length, 1);
});
```

11.1 Assertions

To help developers write tests the Deno standard library comes with a built-in assertions module which can be imported from https://deno.land/std@0.156.0/testing/asserts.ts.

```
import { assert } from "https://deno.land/std@0.156.0/testing/asserts.ts";

Deno.test("Hello Test", () => {
  assert("Hello");
});
```

⚠️ Some popular assertion libraries, like Chai, can be used in Deno too, for example usage see https://deno.land/std@0.156.0/testing/chai_example.ts.

The assertions module provides 14 assertions:

- assert(expr: unknown, msg = ""): asserts expr
- assertEquals(actual: unknown, expected: unknown, msg?: string): void
- assertExists(actual: unknown, msg?: string): void
- assertNotEquals(actual: unknown, expected: unknown, msg?: string): void
- assertStrictEquals(actual: unknown, expected: unknown, msg?: string): void
- assertAlmostEquals(actual: number, expected: number, epsilon = 1e-7, msg?: string): void
- assertInstanceOf(actual: unknown, expectedType: unknown, msg?: string): void
- assertStringIncludes(actual: string, expected: string, msg?: string): void
- assertArrayIncludes(actual: unknown[], expected: unknown[], msg?: string): void
- assertMatch(actual: string, expected: RegExp, msg?: string): void
- assertNotMatch(actual: string, expected: RegExp, msg?: string): void
- assertObjectMatch(actual: Record<PropertyKey, unknown>, expected: Record<PropertyKey, unknown>): void
- assertThrows(fn: () => void, ErrorClass?: Constructor, msgIncludes?: string | undefined, msg?: string | undefined): Error
- assertRejects(fn: () => Promise<unknown>, ErrorClass?: Constructor, msgIncludes?: string | undefined, msg?: string | undefined): Promise<void>

In addition to the above assertions, the snapshot module also exposes an assertSnapshot function. The documentation for this module can be found here.

11.1.1 Assert

The assert method is a simple 'truthy' assertion and can be used to assert any value which can be inferred as true.

```
Deno.test("Test Assert", () => {
  assert(1);
  assert("Hello");
  assert(true);
});
```

11.1.2 Exists

The assertExists can be used to check if a value is not null or undefined.

```
assertExists("Denosaurus");
Deno.test("Test Assert Exists", () => {
  assertExists("Denosaurus");
  assertExists(false);
  assertExists(0);
});
```

11.1.3 Equality

There are three equality assertions
available, assertEquals(), assertNotEquals() and assertStrictEquals().
The assertEquals() and assertNotEquals() methods provide a general equality check and are capable of asserting equality between primitive types and objects.

```
Deno.test("Test Assert Equals", () => {
  assertEquals(1, 1);
  assertEquals("Hello", "Hello");
  assertEquals(true, true);
  assertEquals(undefined, undefined);
  assertEquals(null, null);
  assertEquals(new Date(), new Date());
  assertEquals(new RegExp("abc"), new RegExp("abc"));

  class Foo {}
  const foo1 = new Foo();
  const foo2 = new Foo();

  assertEquals(foo1, foo2);
});

Deno.test("Test Assert Not Equals", () => {
  assertNotEquals(1, 2);
  assertNotEquals("Hello", "World");
  assertNotEquals(true, false);
  assertNotEquals(undefined, "");
  assertNotEquals(new Date(), Date.now());
  assertNotEquals(new RegExp("abc"), new RegExp("def"));
});
```

By contrast assertStrictEquals() provides a simpler, stricter equality check based on the === operator.
As a result it will not assert two instances of identical objects as they won't be referentially the same.

```
Deno.test("Test Assert Strict Equals", () => {
  assertStrictEquals(1, 1);
  assertStrictEquals("Hello", "Hello");
  assertStrictEquals(true, true);
  assertStrictEquals(undefined, undefined);
});
```

The assertStrictEquals() assertion is best used when you wish to make a precise check against two primitive types.

11.1.3.1 Equality for numbers

When testing equality between numbers, it is important to keep in mind that some of them cannot be accurately depicted by IEEE-754 double-precision floating-point representation.
That's especially true when working with decimal numbers, where assertStrictEquals() may work in some cases but not in others:

```
import {
  assertStrictEquals,
  assertThrows,
} from "https://deno.land/std@0.156.0/testing/asserts.ts";

Deno.test("Test Assert Strict Equals with float numbers", () => {
  assertStrictEquals(0.25 + 0.25, 0.25);
  assertThrows(() => assertStrictEquals(0.1 + 0.2, 0.3));
  //0.1 + 0.2 will be stored as 0.30000000000000004 instead of 0.3
});
```

Instead, assertAlmostEquals() provides a way to test that given numbers are close enough to be considered equals. Default tolerance is set to 1e-7 though it is possible to change it by passing a third optional parameter.

```
import {
  assertAlmostEquals,
  assertThrows,
} from "https://deno.land/std@0.156.0/testing/asserts.ts";

Deno.test("Test Assert Almost Equals", () => {
  assertAlmostEquals(0.1 + 0.2, 0.3);
  assertAlmostEquals(0.1 + 0.2, 0.3, 1e-16);
  assertThrows(() => assertAlmostEquals(0.1 + 0.2, 0.3, 1e-17));
});
```

11.1.3.2 Instance types

To check if an object is an instance of a specific constructor, you can use assertInstanceOf(). This has the added benefit that it lets TypeScript know the passed in variable has a specific type:

```
import { assertInstanceOf } from "https://deno.land/std@0.156.0/testing/asserts.ts";

Deno.test("Test Assert Instance Type", () => {
  const variable = new Date() as unknown;

  assertInstanceOf(variable, Date);

  // This won't cause type errors now that
  // it's type has been asserted against.
  variable.getDay();
});
```

11.1.4 Contains

There are two methods available to assert a value contains a
value, assertStringIncludes() and assertArrayIncludes().
The assertStringIncludes() assertion does a simple includes check on a string to see if it contains the expected string.

```
Deno.test("Test Assert String Contains", () => {
  assertStringIncludes("Hello World", "Hello");
});
```

The assertArrayIncludes() assertion is slightly more advanced and can find both a value within an array and an array of values within an array.

```
Deno.test({
  name: "leaky operation test",
  fn() {
    crypto.subtle.digest(
      "SHA-256",
      new TextEncoder().encode("a".repeat(100000000)),
    );
  },
  sanitizeOps: false,
});
```

11.4.3 Exit sanitizer

There's also the exit sanitizer which ensures that tested code doesn't call Deno.exit() signaling a false test success.

This is enabled by default for all tests, but can be disabled by setting the sanitizeExit boolean to false in the test definition.

```
Deno.test({
  name: "false success",
  fn() {
    Deno.exit(0);
  },
  sanitizeExit: false,
});

// This test never runs, because the process exits during "false success" test
Deno.test({
  name: "failing test",
  fn() {
    throw new Error("this test fails");
  },
});
```

11.5 Behavior-Driven Development

With the bdd.ts module you can write your tests in a familiar format for grouping tests and adding setup/teardown hooks used by other JavaScript testing frameworks like Jasmine, Jest, and Mocha.

The describe function creates a block that groups together several related tests. The it function registers an individual test case.

11.5.1 Hooks

There are 4 types of hooks available for test suites. A test suite can have multiples of each type of hook, they will be called in the order that they are registered. The afterEach and afterAll hooks will be called whether or not the test case passes. The *All hooks will be called once for the whole group while the *Each hooks will be called for each individual test case.

- beforeAll: Runs before all of the tests in the test suite.
- afterAll: Runs after all of the tests in the test suite finish.
- beforeEach: Runs before each of the individual test cases in the test suite.
- afterEach: Runs after each of the individual test cases in the test suite.

If a hook is registered at the top level, a global test suite will be registered and all tests will belong to it. Hooks registered at the top level must be registered before any individual test cases or test suites.

11.5.2 Focusing tests

The assertRejects() assertion is a little more complicated, mainly because it deals with Promises. But basically it will catch thrown errors or rejections in Promises. You can also optionally check for the error type and error message. This can be used similar to assertThrows() but with async functions.

```
Deno.test("Test Assert Throws Async", () => {
  await assertRejects(
    () => {
      return new Promise(() => {
        throw new Error("Panic! Threw Error");
      });
    },
    Error,
    "Panic! Threw Error",
  );

  await assertRejects(
    () => {
      return Promise.reject(new Error("Panic! Reject Error"));
    },
    Error,
    "Panic! Reject Error",
  );
});
```

11.1.8 Custom Messages

Each of Deno's built-in assertions allow you to overwrite the standard CLI error message if you wish. For instance this example will output "Values Don't Match!" rather than the standard CLI error message.

```
Deno.test("Test Assert Equal Fail Custom Message", () => {
  assertEquals(1, 2, "Values Don't Match!");
});
```

11.1.9 Custom Tests

While Deno comes with powerful assertions modules but there is always something specific to the project you can add. Creating custom assertion function can improve readability and reduce the amount of code.

```
import { AssertionError } from "https://deno.land/std@0.156.0/testing/asserts.ts";

function assertPowerOf(actual: number, expected: number, msg?: string): void {
  let received = actual;
  while (received % expected === 0) received = received / expected;
  if (received !== 1) {
    if (!msg) {
      msg = `actual: "${actual}" expected to be a power of : "${expected}"`;
    }
    throw new AssertionError(msg);
  }
}
```

Use this matcher in your code like this:

```
Deno.test("Test Assert PowerOf", () => {
  assertPowerOf(8, 2);
  assertPowerOf(11, 4);
});
```

11.2 Coverage

Deno will collect test coverage into a directory for your code if you specify the --coverage flag when starting deno test.

This coverage information is acquired directly from the JavaScript engine (V8) which is very accurate.

This can then be further processed from the internal format into well known formats by the deno coverage tool.

⚠ To ensure consistent coverage results, make sure to process coverage data immediately after running tests. Otherwise source code and collected coverage data might get out of sync and unexpectedly show uncovered lines.

```
# Go into your project's working directory
git clone https://github.com/oakserver/oak && cd oak

# Collect your coverage profile with deno test --coverage=<output_directory>
deno test --coverage=cov_profile

# From this you can get a pretty printed diff of uncovered lines
deno coverage cov_profile

# Or generate an lcov report
deno coverage cov_profile --lcov --output=cov_profile.lcov

# Which can then be further processed by tools like genhtml
genhtml -o cov_profile/html cov_profile.lcov
```

By default, deno coverage will exclude any files matching the regular expression test\.(ts|tsx|mts|js|mjs|jsx|cjs|cts) and only consider including specifiers matching the regular expression ^file: - ie. remote files will be excluded from coverage report.

These filters can be overridden using the --exclude and --include flags. A module specifier must match the include_regular expression and not match the exclude_ expression for it to be a part of the report.

11.3 Documentation

Deno supports type-checking your documentation examples.

This makes sure that examples within your documentation are up to date and working.

The basic idea is this:

```
/**
 * # Examples
 *
 * ```ts
 * const x = 42;
 * ```
 */
```

The triple backticks mark the start and end of code blocks, the language is determined by the language identifier attribute which may be any of the following:

- js
- jsx
- ts
- tsx

If no language identifier is specified then the language is inferred from media type of the source document that the code block is extracted from.

If this example was in a file named foo.ts, running deno test --doc foo.ts will extract this example, and then type-check it as a standalone module living in the same directory as the module being documented.

To document your exports, import the module using a relative path specifier:

```
/**
 * # Examples
 *
 * ```ts
 * import { foo } from "./foo.ts";
 * ```
 */
export function foo(): string {
  return "foo";
}
```

11.4 Sanitizers

The test runner offers several sanitizers to ensure that the test behaves in a reasonable and expected way.

11.4.1 Resource sanitizer

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.

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

11.4.2 Op sanitizer

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.

```
const userTests = describe("User");

it(userTests, "users initially empty", () => {
  assertEquals(User.users.size, 0);
});

it(userTests, "constructor", () => {
  try {
    const user = new User("Kyle");
    assertEquals(user.name, "Kyle");
    assertStrictEquals(User.users.get("Kyle"), user);
  } finally {
    User.users.clear();
  }
});

const ageTests = describe({
  name: "age",
  suite: userTests,
  beforeEach(this: { user: User }) {
    this.user = new User("Kyle");
  },
  afterEach() {
    User.users.clear();
  },
});

it(ageTests, "getAge", function () {
  const { user } = this;
  assertThrows(() => user.getAge(), Error, "Age unknown");
  user.age = 18;
  assertEquals(user.getAge(), 18);
});

it(ageTests, "setAge", function () {
  const { user } = this;
  user.setAge(18);
  assertEquals(user.getAge(), 18);
});
```

11.5.6.3 **Mixed test grouping**

Both nested test grouping and flat test grouping can be used together. This can be useful if you'd like to create deep groupings without all the extra indentation in front of each line.

```
// https://deno.land/std@0.156.0/testing/bdd_examples/user_mixed_test.ts
import {
  assertEquals,
  assertStrictEquals,
```

If you would like to run only specific test cases, you can do so by calling it.only instead of it. If you would like to run only specific test suites, you can do so by calling describe.only instead of describe.
There is one limitation to this when using the flat test grouping style. When describe is called without being nested, it registers the test with Deno.test. If a child test case or suite is registered with it.only or describe.only, it will be scoped to the top test suite instead of the file. To make them the only tests that run in the file, you would need to register the top test suite with describe.only too.

11.5.3 Ignoring tests

If you would like to not run specific individual test cases, you can do so by calling it.ignore instead of it. If you would like to not run specific test suites, you can do so by calling describe.ignore instead of describe.

11.5.4 Sanitization options

Like Deno.TestDefinition, the DescribeDefinition and ItDefinition have sanitization options. They work in the same way.

- sanitizeExit: Ensure the test case does not prematurely cause the process to exit, for example via a call to Deno.exit. Defaults to true.
- sanitizeOps: Check that the number of async completed ops after the test is the same as number of dispatched ops. Defaults to true.
- sanitizeResources: Ensure the test case does not "leak" resources - ie. the resource table after the test has exactly the same contents as before the test. Defaults to true.

11.5.5 Permissions option

Like Deno.TestDefinition, the DescribeDefintion and ItDefinition have a permissions option. They specify the permissions that should be used to run an individual test case or test suite. Set this to "inherit" to keep the calling thread's permissions. Set this to "none" to revoke all permissions. This setting defaults to "inherit".
There is currently one limitation to this, you cannot use the permissions option on an individual test case or test suite that belongs to another test suite. That's because internally those tests are registered with t.step which does not support the permissions option.

11.5.6 Comparing to Deno.test

The default way of writing tests is using Deno.test and t.step. The describe and it functions have similar call signatures to Deno.test, making it easy to switch between the default style and the behavior-driven development style of writing tests. Internally, describe and it are registering tests with Deno.test and t.step.
Below is an example of a test file using Deno.test and t.step. In the following sections there are examples of how the same test could be written with describe and it using nested test grouping, flat test grouping, or a mix of both styles.

```
// https://deno.land/std@0.156.0/testing/bdd_examples/user_test.ts
import {
  assertEquals,
  assertStrictEquals,
  assertThrows,
} from "https://deno.land/std@0.156.0/testing/asserts.ts";
import { User } from "https://deno.land/std@0.156.0/testing/bdd_examples/user.ts";

Deno.test("User.users initially empty", () => {
  assertEquals(User.users.size, 0);
});
```

```
Deno.test("User constructor", () => {
  try {
    const user = new User("Kyle");
    assertEquals(user.name, "Kyle");
    assertStrictEquals(User.users.get("Kyle"), user);
  } finally {
    User.users.clear();
  }
});

Deno.test("User age", async (t) => {
  const user = new User("Kyle");

  await t.step("getAge", () => {
    assertThrows(() => user.getAge(), Error, "Age unknown");
    user.age = 18;
    assertEquals(user.getAge(), 18);
  });

  await t.step("setAge", () => {
    user.setAge(18);
    assertEquals(user.getAge(), 18);
  });
});
```

11.5.6.1 **Nested test grouping**

Tests created within the callback of a describe function call will belong to the new test suite it creates. The hooks can be created within it or be added to the options argument for describe.

```
// https://deno.land/std@0.156.0/testing/bdd_examples/user_nested_test.ts
import {
  assertEquals,
  assertStrictEquals,
  assertThrows,
} from "https://deno.land/std@0.156.0/testing/asserts.ts";
import {
  afterEach,
  beforeEach,
  describe,
  it,
} from "https://deno.land/std@0.156.0/testing/bdd.ts";
import { User } from "https://deno.land/std@0.156.0/testing/bdd_examples/user.ts";

describe("User", () => {
  it("users initially empty", () => {
    assertEquals(User.users.size, 0);
  });

  it("constructor", () => {
```

```
    try {
      const user = new User("Kyle");
      assertEquals(user.name, "Kyle");
      assertStrictEquals(User.users.get("Kyle"), user);
    } finally {
      User.users.clear();
    }
  });

  describe("age", () => {
    let user: User;

    beforeEach(() => {
      user = new User("Kyle");
    });

    afterEach(() => {
      User.users.clear();
    });

    it("getAge", function () {
      assertThrows(() => user.getAge(), Error, "Age unknown");
      user.age = 18;
      assertEquals(user.getAge(), 18);
    });

    it("setAge", function () {
      user.setAge(18);
      assertEquals(user.getAge(), 18);
    });
  });
});
```

11.5.6.2 **Flat test grouping**

The describe function returns a unique symbol that can be used to reference the test suite for adding tests to it without having to create them within a callback. The gives you the ability to have test grouping without any extra indentation in front of the grouped tests.

```
// https://deno.land/std@0.156.0/testing/bdd_examples/user_flat_test.ts
import {
  assertEquals,
  assertStrictEquals,
  assertThrows,
} from "https://deno.land/std@0.156.0/testing/asserts.ts";
import {
  describe,
  it,
} from "https://deno.land/std@0.156.0/testing/bdd.ts";
import { User } from "https://deno.land/std@0.156.0/testing/bdd_examples/user.ts";
```

```
  assertSpyCalls(multiplySpy, 1);
});
```

One difference you may have noticed between these two examples is that in the second we call the restore method on multiplySpy function. That is needed to remove the spy wrapper from the _internals object's multiply method. The restore method is called in a finally block to ensure that it is restored whether or not the assertion in the try block is successful. The restore method didn't need to be called in the first example because the multiply function was not modified in any way like the _internals object was in the second example.

11.6.2 Stubbing

Say we have two functions, randomMultiple and randomInt, if we want to assert that randomInt is called during execution of randomMultiple we need a way to spy on the randomInt function. That could be done with either of the spying techniques previously mentioned. To be able to verify that the randomMultiple function returns the value we expect it to for what randomInt returns, the easiest way would be to replace the randomInt function's behavior with more predictable behavior.

You could use the first spying technique to do that but that would require adding a randomInt parameter to the randomMultiple function.

You could also use the second spying technique to do that, but your assertions would not be as predictable due to the randomInt function returning random values.

Say we want to verify it returns correct values for both negative and positive random integers. We could easily do that with stubbing. The below example is similar to the second spying technique example but instead of passing the call through to the original randomInt function, we are going to replace randomInt with a function that returns pre-defined values.

```
// https://deno.land/std@0.156.0/testing/mock_examples/random.ts
export function randomInt(lowerBound: number, upperBound: number): number {
  return lowerBound + Math.floor(Math.random() * (upperBound - lowerBound));
}

export function randomMultiple(value: number): number {
  return value * _internals.randomInt(-10, 10);
}

export const _internals = { randomInt };
```

The mock module includes some helper functions to make creating common stubs easy. The returnsNext function takes an array of values we want it to return on consecutive calls.

```
// https://deno.land/std@0.156.0/testing/mock_examples/random_test.ts
import {
  assertSpyCall,
  assertSpyCalls,
  returnsNext,
  stub,
} from "https://deno.land/std@0.156.0/testing/mock.ts";
import { assertEquals } from "https://deno.land/std@0.156.0/testing/asserts.ts";
import {
  _internals,
  randomMultiple,
} from "https://deno.land/std@0.156.0/testing/mock_examples/random.ts";
```

```
  assertThrows,
} from "https://deno.land/std@0.156.0/testing/asserts.ts";
import {
  describe,
  it,
} from "https://deno.land/std@0.156.0/testing/bdd.ts";
import { User } from "https://deno.land/std@0.156.0/testing/bdd_examples/user.ts";

describe("User", () => {
  it("users initially empty", () => {
    assertEquals(User.users.size, 0);
  });

  it("constructor", () => {
    try {
      const user = new User("Kyle");
      assertEquals(user.name, "Kyle");
      assertStrictEquals(User.users.get("Kyle"), user);
    } finally {
      User.users.clear();
    }
  });

  const ageTests = describe({
    name: "age",
    beforeEach(this: { user: User }) {
      this.user = new User("Kyle");
    },
    afterEach() {
      User.users.clear();
    },
  });

  it(ageTests, "getAge", function () {
    const { user } = this;
    assertThrows(() => user.getAge(), Error, "Age unknown");
    user.age = 18;
    assertEquals(user.getAge(), 18);
  });

  it(ageTests, "setAge", function () {
    const { user } = this;
    user.setAge(18);
    assertEquals(user.getAge(), 18);
  });
});
```

11.6 Mocking

Test spies are function stand-ins that are used to assert if a function's internal behavior matches expectations. Test spies on methods keep the original behavior but allow you to test how the method is called and what it returns. Test stubs are an extension of test spies that also replaces the original method's behavior.

11.6.1 Spying

Say we have two functions, square and multiply, if we want to assert that the multiply function is called during execution of the square function we need a way to spy on the multiply function. There are a few ways to achieve this with Spies, one is to have the square function take the multiply as a parameter.

```
// https://deno.land/std@0.156.0/testing/mock_examples/parameter_injection.ts
export function multiply(a: number, b: number): number {
  return a * b;
}

export function square(
  multiplyFn: (a: number, b: number) => number,
  value: number,
): number {
  return multiplyFn(value, value);
}
```

This way, we can call square(multiply, value) in the application code or wrap a spy function around the multiply function and call square(multiplySpy, value) in the testing code.

```
// https://deno.land/std@0.156.0/testing/mock_examples/parameter_injection_test.ts
import {
  assertSpyCall,
  assertSpyCalls,
  spy,
} from "https://deno.land/std@0.156.0/testing/mock.ts";
import { assertEquals } from "https://deno.land/std@0.156.0/testing/asserts.ts";
import {
  multiply,
  square,
} from "https://deno.land/std@0.156.0/testing/mock_examples/parameter_injection.ts";

Deno.test("square calls multiply and returns results", () => {
  const multiplySpy = spy(multiply);

  assertEquals(square(multiplySpy, 5), 25);

  // asserts that multiplySpy was called at least once and details about the first call.
  assertSpyCall(multiplySpy, 0, {
    args: [5, 5],
    returned: 25,
  });

  // asserts that multiplySpy was only called once.
  assertSpyCalls(multiplySpy, 1);
```

```
});
```

If you prefer not adding additional parameters for testing purposes only, you can use spy to wrap a method on an object instead. In the following example, the exported _internals object has the multiply function we want to call as a method and the square function calls _internals.multiply instead of multiply.

```
// https://deno.land/std@0.156.0/testing/mock_examples/internals_injection.ts
export function multiply(a: number, b: number): number {
  return a * b;
}

export function square(value: number): number {
  return _internals.multiply(value, value);
}

export const _internals = { multiply };
```

This way, we can call square(value) in both the application code and testing code. Then spy on the multiply method on the _internals object in the testing code to be able to spy on how the square function calls the multiply function.

```
// https://deno.land/std@0.156.0/testing/mock_examples/internals_injection_test.ts
import {
  assertSpyCall,
  assertSpyCalls,
  spy,
} from "https://deno.land/std@0.156.0/testing/mock.ts";
import { assertEquals } from "https://deno.land/std@0.156.0/testing/asserts.ts";
import {
  _internals,
  square,
} from "https://deno.land/std@0.156.0/testing/mock_examples/internals_injection.ts";

Deno.test("square calls multiply and returns results", () => {
  const multiplySpy = spy(_internals, "multiply");

  try {
    assertEquals(square(5), 25);
  } finally {
    // unwraps the multiply method on the _internals object
    multiplySpy.restore();
  }

  // asserts that multiplySpy was called at least once and details about the first call.
  assertSpyCall(multiplySpy, 0, {
    args: [5, 5],
    returned: 25,
  });

  // asserts that multiplySpy was only called once.
```

The result of the serializer function will be written to the snapshot file in update mode, and in assert mode will be compared to the snapshot stored in the snapshot file.

```
// example_test.ts
import { assertSnapshot, serialize } from "https://deno.land/std@0.156.0/testing/snapshot.ts";
import { stripColor } from "https://deno.land/std@0.156.0/fmt/colors.ts";

/**
 * Serializes `actual` and removes ANSI escape codes.
 */
function customSerializer(actual: string) {
  return serialize(stripColor(actual));
}

Deno.test("Custom Serializer", async function (t): Promise<void> {
  const output = "\x1b[34mHello World!\x1b[39m";
  await assertSnapshot(t, output, {
    serializer: customSerializer,
  });
});
```

```
// __snapshots__/example_test.ts.snap
export const snapshot = {};

snapshot[`Custom Serializer 1`] = `"Hello World!"`;
```

Custom serializers can be useful in a variety of cases. One possible use case is to discard information which is not relevant and/or to present the serialized output in a more human readable form.

For example, the above code snippet shows how a custom serializer could be used to remove ANSI escape codes (which encode font color and styles in CLI applications), making the snapshot more readable than it would be otherwise.

Other common use cases would be to obfuscate values which are non-deterministic or which you may not want to write to disk for other reasons. For example, timestamps or file paths.

Note that the default serializer is exported from the snapshot module so that its functionality can be easily extended.

dir and path

The dir and path options allow you to control where the snapshot file will be saved to and read from. These can be absolute paths or relative paths. If relative, the they will be resolved relative to the test file.

For example, if your test file is located at /path/to/test.ts and the dir option is set to snapshots, then the snapshot file would be written to /path/to/snapshots/test.ts.snap.

As shown in the above example, the dir option allows you to specify the snapshot directory, while still using the default format for the snapshot file name.

In contrast, the path option allows you to specify the directory and file name of the snapshot file.

For example, if your test file is located at /path/to/test.ts and the path option is set to snapshots/test.snapshot, then the snapshot file would be written to /path/to/snapshots/test.snapshot.

If both dir and path are specified, the dir option will be ignored and the path option will be handled as normal.

mode

The mode option can be either assert or update. When set, the --update and -u flags will be ignored.

```
Deno.test("randomMultiple uses randomInt to generate random multiples between -10 and 10 times the value", () => {
  const randomIntStub = stub(_internals, "randomInt", returnsNext([-3, 3]));

  try {
    assertEquals(randomMultiple(5), -15);
    assertEquals(randomMultiple(5), 15);
  } finally {
    // unwraps the randomInt method on the _internals object
    randomIntStub.restore();
  }

  // asserts that randomIntStub was called at least once and details about the first call.
  assertSpyCall(randomIntStub, 0, {
    args: [-10, 10],
    returned: -3,
  });
  // asserts that randomIntStub was called at least twice and details about the second call.
  assertSpyCall(randomIntStub, 1, {
    args: [-10, 10],
    returned: 3,
  });

  // asserts that randomIntStub was only called twice.
  assertSpyCalls(randomIntStub, 2);
});
```

11.6.3 Faking time

Say we have a function that has time based behavior that we would like to test. With real time, that could cause tests to take much longer than they should. If you fake time, you could simulate how your function would behave over time starting from any point in time. Below is an example where we want to test that the callback is called every second.

```
// https://deno.land/std@0.156.0/testing/mock_examples/interval.ts
export function secondInterval(cb: () => void): number {
  return setInterval(cb, 1000);
}
```

With FakeTime we can do that. When the FakeTime instance is created, it splits from real time. The Date, setTimeout, clearTimeout, setInterval and clearInterval globals are replaced with versions that use the fake time until real time is restored. You can control how time ticks forward with the tick method on the FakeTime instance.

```
// https://deno.land/std@0.156.0/testing/mock_examples/interval_test.ts
import {
  assertSpyCalls,
  spy,
} from "https://deno.land/std@0.156.0/testing/mock.ts";
import { FakeTime } from "https://deno.land/std@0.156.0/testing/time.ts";
import { secondInterval } from
"https://deno.land/std@0.156.0/testing/mock_examples/interval.ts";
```

```
Deno.test("secondInterval calls callback every second and stops after being cleared", () => {
  const time = new FakeTime();

  try {
    const cb = spy();
    const intervalId = secondInterval(cb);
    assertSpyCalls(cb, 0);
    time.tick(500);
    assertSpyCalls(cb, 0);
    time.tick(500);
    assertSpyCalls(cb, 1);
    time.tick(3500);
    assertSpyCalls(cb, 4);

    clearInterval(intervalId);
    time.tick(1000);
    assertSpyCalls(cb, 4);
  } finally {
    time.restore();
  }
});
```

11.7 Snapshots

The Deno standard library comes with a snapshot module, which enables developers to write tests which assert a value against a reference snapshot. This reference snapshot, is a serialized representation of the original value and is stored alongside the test file.

Snapshot testing can be useful in many cases, as it enables catching a wide array of bugs with very little code. It is particularly helpful in situations where it is difficult to precisely express what should be asserted, without requiring a prohibitive amount of code, or where the assertions a test makes are expected to change often. It therefore lends itself especially well to use in the development of front ends and CLIs.

11.7.1 Basic usage

The assertSnapshot function will create a snapshot of a value and compare it to a reference snapshot, which is stored alongside the test file in the __snapshots__ directory.

```
// example_test.ts
import { assertSnapshot } from "https://deno.land/std@0.156.0/testing/snapshot.ts";

Deno.test("isSnapshotMatch", async function (t): Promise<void> {
  const a = {
    hello: "world!",
    example: 123,
  };
  await assertSnapshot(t, a);
});
```

```
// __snapshots__/example_test.ts.snap
export const snapshot = {};
```

```
snapshot[`isSnapshotMatch 1`] = `
{
  example: 123,
  hello: "world!",
}
`;
```

Calling assertSnapshot in a test will throw an AssertionError, causing the test to fail, if the snapshot created during the test does not match the one in the snapshot file.

11.7.2 Creating and updating snapshots

When adding new snapshot assertions to your test suite, or when intentionally making changes which cause your snapshots to fail, you can update your snapshots by running the snapshot tests in update mode. Tests can be run in update mode by passing the --update or -u flag as an argument when running the test. When this flag is passed, then any snapshots which do not match will be updated.

```
deno test --allow-all -- --update
```

Additionally, new snapshots will only be created when this flag is present.

11.7.3 Permissions

When running snapshot tests, the --allow-read permission must be enabled, or else any calls to assertSnapshot will fail due to insufficient permissions. Additionally, when updating snapshots, the --allow-write permission must also be enabled, as this is required in order to update snapshot files.

The assertSnapshot function will only attempt to read from and write to snapshot files. As such, the allow list for --allow-read and --allow-write can be limited to only include existing snapshot files, if so desired.

11.7.4 Version Control

Snapshot testing works best when changes to snapshot files are comitted alongside other code changes. This allows for changes to reference snapshots to be reviewed along side the code changes that caused them, and ensures that when others pull your changes, their tests will pass without needing to update snapshots locally.

11.7.5 Advanced Usage

11.7.5.1 Options

The assertSnapshot function can also be called with an options object which offers greater flexibility and enables some non standard use cases.

```
import { assertSnapshot } from "https://deno.land/std@0.156.0/testing/snapshot.ts";

Deno.test("isSnapshotMatch", async function (t): Promise<void> {
  const a = {
    hello: "world!",
    example: 123,
  };
  await assertSnapshot(t, a, {
    // options
  });
});
```

serializer

The serializer option allows you to provide a custom serializer function. This will be called by assertSnapshot and be passed the value being asserted. It should return a string. It is important that the serializer function is deterministic i.e. that it will always produce the same output, given the same input.

```
          This is an example of how Deno.customInspect could be used to snapshot an
intermediate SSR representation
        </p>
    </body>
</html>
`;
```

In contrast, when we remove the Deno.customInspect method, the test will produce the following snapshot.

```
// __snapshots__/example_test.ts.snap
export const snapshot = {};

snapshot[`Page HTML Tree 1`] = `
HTMLTag {
  children: [
    HTMLTag {
      children: [
        HTMLTag {
          children: [
            "Simple SSR Example",
          ],
          name: "title",
        },
      ],
      name: "head",
    },
    HTMLTag {
      children: [
        HTMLTag {
          children: [
            "Simple SSR Example",
          ],
          name: "h1",
        },
        HTMLTag {
          children: [
            "This is an example of how Deno.customInspect could be used to snapshot an
intermediate SSR representation",
          ],
          name: "p",
        },
      ],
      name: "body",
    },
  ],
  name: "html",
}
`;
```

If the mode option is set to assert, then assertSnapshot will always behave as though the update flag is not passed i.e. if the snapshot does not match the one saved in the snapshot file, then an AssertionError will be thrown.

If the mode option is set to update, then assertSnapshot will always behave as though the update flag has been passed i.e. if the snapshot does not match the one saved in the snapshot file, then the snapshot will be updated after all tests have run.

name

The name option specifies the name of the snapshot. By default, the name of the test step will be used. However, if specified, the name option will be used instead.

```
// example_test.ts
import { assertSnapshot } from "https://deno.land/std@0.156.0/testing/snapshot.ts";

Deno.test("isSnapshotMatch", async function (t): Promise<void> {
  const a = {
    hello: "world!",
    example: 123,
  };
  await assertSnapshot(t, a, {
    name: "Test Name"
  });
});
// __snapshots__/example_test.ts.snap
export const snapshot = {};

snapshot[`Test Name 1`] = `
{
  example: 123,
  hello: "world!",
}
`;
```

When assertSnapshot is run multiple times with the same value for name, then the suffix will be incremented as normal. i.e. Test Name 1, Test Name 2, Test Name 3, etc.

msg

Allows setting a custom error message to use. This will overwrite the default error message, which includes the diff for failed snapshots.

11.7.5.2 Default Options

You can configure default options for assertSnapshot.

```
// example_test.ts
import { createAssertSnapshot } from "https://deno.land/std@0.156.0/testing/snapshot.ts";

const assertSnapshot = createAssertSnapshot({
  // options
});
```

When configuring default options like this, the resulting assertSnapshot function will function the same as the default function exported from the snapshot module. If passed an optional options object, this will take precedence over the default options, where the value provded for an option differs.

It is possible to "extend" an assertSnapshot function which has been configured with default options.

```
// example_test.ts
```

```
import { createAssertSnapshot } from "https://deno.land/std@0.156.0/testing/snapshot.ts";
import { stripColor } from "https://deno.land/std@0.156.0/fmt/colors.ts";

const assertSnapshot = createAssertSnapshot({
  dir: ".snaps",
});

const assertMonochromeSnapshot = createAssertSnapshot<string>(
  { serializer: stripColor },
  assertSnapshot,
);

Deno.test("isSnapshotMatch", async function (t): Promise<void> {
  const a = "\x1b[32mThis green text has had it's colours stripped\x1b[39m";
  await assertMonochromeSnapshot(t, a);
});
```

```
// .snaps/example_test.ts.snap
export const snapshot = {};

snapshot[`isSnapshotMatch 1`] = `This green text has had it's colours stripped`;
```

11.7.5.3 Serialization with Deno.customInspect

The default serialization behaviour can be customised in two ways. The first is by specifying the serializer option. This allows you to control the serialisation of any value which is passed to a specific assertSnapshot call. See the above documentation on the correct usage of this option.

The second option is to make use of Deno.customInspect. Because the default serializer used by assertSnapshot uses Deno.inspect under the hood, you can set property Symbol.for("Deno.customInspect") to a custom serialization function.

Doing so will ensure that the custom serialization will, by default, be used whenever the object is passed to assertSnapshot. This can be useful in many cases. One example is shown in the code snippet below.

```
// example_test.ts
import { assertSnapshot } from "https://deno.land/std@0.156.0/testing/snapshot.ts";

class HTMLTag {
  constructor(
    public name: string,
    public children: Array<HTMLTag | string> = [],
  ) {}

  public render(depth: number) {
    const indent = "    ".repeat(depth);
    let output = `${indent}<${this.name}>\n`;
    for (const child of this.children) {
      if (child instanceof HTMLTag) {
        output += `${child.render(depth + 1)}\n`;
      } else {
        output += `${indent}    ${child}\n`;
      }
    }
    output += `${indent}</${this.name}>`;
    return output;
  }

  public [Symbol.for("Deno.customInspect")]() {
    return this.render(0);
  }
}

Deno.test("Page HTML Tree", async (t) => {
  const page = new HTMLTag("html", [
    new HTMLTag("head", [
      new HTMLTag("title", [
        "Simple SSR Example",
      ]),
    ]),
    new HTMLTag("body", [
      new HTMLTag("h1", [
        "Simple SSR Example",
      ]),
      new HTMLTag("p", [
        "This is an example of how Deno.customInspect could be used to snapshot an
intermediate SSR representation",
      ]),
    ]),
  ]);

  await assertSnapshot(t, page);
});
```

This test will produce the following snapshot.

```
// __snapshots__/example_test.ts.snap
export const snapshot = {};

snapshot[`Page HTML Tree 1`] = `
<html>
  <head>
    <title>
      Simple SSR Example
    </title>
  </head>
  <body>
    <h1>
      Simple SSR Example
    </h1>
    <p>
```

Identifier	Description
_	Yields the last evaluated expression
_error	Yields the last thrown error

```
Deno 1.14.3
exit using ctrl+d or close()
> "hello world!"
"hello world!"
> _
"hello world!"
> const foo = "bar";
undefined
> _
undefined
```

12.3.2 Special functions

The REPL provides several functions in the global scope:

Function	Description
clear()	Clears the entire terminal screen
close()	Close the current REPL session

12.3.3 --eval flag

--eval flag allows you to run some code in the runtime before you are dropped into the REPL. This is useful for importing some code you commonly use in the REPL, or modifying the runtime in some way:

```
$ deno repl --eval 'import { assert } from "https://deno.land/std@0.156.0/testing/asserts.ts"'
Deno 1.14.3
exit using ctrl+d or close()
> assert(true)
undefined
> assert(false)
Uncaught AssertionError
    at assert (https://deno.land/std@0.110.0/testing/asserts.ts:224:11)
    at <anonymous>:2:1
```

12.3.4 --eval-file flag

--eval-file flag allows you to run code from specified files before you are dropped into the REPL. Like the --eval flag, this is useful for importing code you commonly use in the REPL, or modifying the runtime in some way.

Files can be specified as paths or URLs. URL files are cached and can be reloaded via the --reload flag.

If --eval is also specified, then --eval-file files are run before the --eval code.

```
$ deno repl --eval-file=https://examples.deno.land/hello-
world.ts,https://deno.land/std@0.156.0/encoding/ascii85.ts
Download https://examples.deno.land/hello-world.ts
Hello, World!
Download https://deno.land/std@0.156.0/encoding/ascii85.ts
Deno 1.20.5
exit using ctrl+d or close()
```

You can see that this snapshot is much less readable. This is because:

1. The keys are sorted alphabetically, so the name of the element is displayed after its children
2. It includes a lot of extra information, causing the snapshot to be more than twice as long
3. It is not an accurate serialization of the HTML which the data represents

Note that in this example it would be entirely possible to achieve the same result by calling:

```
await assertSnapshot(t, page.render(0));
```

However, depending on the public API you choose to expose, this may not be practical in other cases.

It is also worth considering that this will have an impact beyond just snapshot testing. For example, Deno.customInspect is also used to serialize objects when calling console.log and in some other cases. This may or may not be desirable.

12 Tools

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

- benchmarker (deno bench)
- bundler (deno bundle)
- compiling executables (deno compile)
- installer (deno install)
- dependency inspector (deno info)
- documentation generator (deno doc)
- formatter (deno fmt)
- linter (deno lint)
- repl (deno repl)
- task runner (deno task)
- test runner (deno test)
- vendoring dependencies (deno vendor)

12.1 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@0.156.0/http/file_server.ts
[1/1] Compiling https://deno.land/std@0.156.0/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@0.156.0/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.
- If the resulting name has an '@...' suffix, strip it.

To change the installation root, use --root:

```
deno install --allow-net --allow-read --root /usr/local
https://deno.land/std@0.156.0/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@0.156.0/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
```

12.1.1 Uninstall

You can uninstall the script with deno uninstall command.

```
$ deno uninstall file_server
deleted /Users/deno/.deno/bin/file_server
✅ Successfully uninstalled file_server
```

See deno uninstall -h for more details.

12.2 Formatter

Deno ships with a built-in code formatter that will auto-format the following files:

File Type	Extension
JavaScript	.js
TypeScript	.ts
JSX	.jsx
TSX	.tsx
Markdown	.md, .markdown
JSON	.json
JSONC	.jsonc

In addition, deno fmt can format code snippets in Markdown files. Snippets must be enclosed in triple backticks and have a language attribute.

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

12.2.1 Ignoring Code

Ignore formatting code by preceding it with a // deno-fmt-ignore comment in TS/JS/JSONC:

```
// 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.

In markdown you may use a <!-- deno-fmt-ignore --> comment or ignore a whole file with a <!-- deno-fmt-ignore-file --> comment. To ignore a section of markdown, surround the code with <!-- deno-fmt-ignore-start --> and <!-- deno-fmt-ignore-end --> comments.

12.2.2 Configuration

ℹ It is recommended to stick with default options.

Starting with Deno v1.14 a formatter can be customized using either a configuration file or following CLI flags:

- --options-use-tabs - Whether to use tabs. Defaults to false (using spaces).
- --options-line-width - The width of a line the printer will try to stay under. Note that the printer may exceed this width in certain cases. Defaults to 80.
- --options-indent-width - The number of characters for an indent. Defaults to 2.
- --options-single-quote - Whether to use single quote. Defaults to false (using double quote).
- --options-prose-wrap={always,never,preserve} - Define how prose should be wrapped in Markdown files. Defaults to "always".

12.3 Read-Eval-Print-Loop

deno repl starts a read-eval-print-loop, which lets you interactively build up program state in the global context, it is especially useful for quick prototyping and checking snippets of code.

⚠ Deno REPL supports JavaScript as well as TypeScript, however TypeScript code is not type-checked, instead it is transpiled to JavaScript behind the scenes.

⚠ To make it easier to copy-paste code samples, Deno REPL supports import and export declarations. It means that you can paste code containing import ... from ...;, export class ... or export function ... and it will work as if you were executing a regular ES module.

12.3.1 Special variables

The REPL provides a couple of special variables, that are always available:

- Web Storage API

12.6 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.

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 is used to generate module documentation.

12.7 Dependency Inspector

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

```
deno info https://deno.land/std@0.67.0/http/file_server.ts
Download https://deno.land/std@0.67.0/http/file_server.ts
...
local:
/home/deno/.cache/deno/deps/https/deno.land/f57792e36f2dbf28b14a75e2372a479c6392780d
4712d76698d5031f943c0020
type: TypeScript
emit:
/home/deno/.cache/deno/gen/https/deno.land/f57792e36f2dbf28b14a75e2372a479c6392780d4
712d76698d5031f943c0020.js
dependencies: 23 unique (total 139.89KB)
https://deno.land/std@0.67.0/http/file_server.ts (10.49KB)
├─┬ https://deno.land/std@0.67.0/path/mod.ts (717B)
│ ├── https://deno.land/std@0.67.0/path/_constants.ts (2.35KB)
│ ├─┬ https://deno.land/std@0.67.0/path/win32.ts (27.36KB)
│ │ ├── https://deno.land/std@0.67.0/path/_interface.ts (657B)
│ │ ├── https://deno.land/std@0.67.0/path/_constants.ts *
│ │ ├─┬ https://deno.land/std@0.67.0/path/_util.ts (3.3KB)
│ │ │ ├── https://deno.land/std@0.67.0/path/_interface.ts *
│ │ │ └── https://deno.land/std@0.67.0/path/_constants.ts *
│ │ └── https://deno.land/std@0.67.0/_util/assert.ts (405B)
│ ├─┬ https://deno.land/std@0.67.0/path/posix.ts (12.67KB)
│ │ ├── https://deno.land/std@0.67.0/path/_interface.ts *
│ │ ├── https://deno.land/std@0.67.0/path/_constants.ts *
```

```
> rfc1924 // local (not exported) variable defined in ascii85.ts
"0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz!#$%&(
)*+-;<=>?@^_`{|}~"
```

12.3.4.1 Relative Import Path Resolution

If --eval-file specifies a code file that contains relative imports, then the runtime will try to resolve the imports relative to the current working directory. It will not try to resolve them relative to the code file's location. This can cause "Module not found" errors when --eval-file is used with module files:

```
$ deno repl --eval-file=https://deno.land/std@0.156.0/hash/md5.ts
error in --eval-file file https://deno.land/std@0.156.0/hash/md5.ts. Uncaught TypeError:
Module not found "file:///home/encoding/hex.ts".
    at async <anonymous>:2:13
Deno 1.20.5
exit using ctrl+d or close()
> close()
$ deno repl --eval-file=https://deno.land/std@0.156.0/encoding/hex.ts
Download https://deno.land/std@0.156.0/encoding/hex.ts
Deno 1.20.5
exit using ctrl+d or close()
>
```

12.3.5 Tab completions

Tab completions are crucial feature for quick navigation in REPL. After hitting tab key, Deno will now show a list of all possible completions.

```
$ deno repl
Deno 1.14.3
exit using ctrl+d or close()
> Deno.read
readTextFile      readFile          readDirSync       readLinkSync      readAll
read
readTextFileSync  readFileSync      readDir           readLink
readAllSync        readSync
```

12.3.6 Keyboard shortcuts

Keystroke	Action
Ctrl-A, Home	Move cursor to the beginning of line
Ctrl-B, Left	Move cursor one character left
Ctrl-C	Interrupt and cancel the current edit
Ctrl-D	If if line is empty, signal end of line
Ctrl-D, Del	If line is not empty, delete character under cursor
Ctrl-E, End	Move cursor to end of line
Ctrl-F, Right	Move cursor one character right
Ctrl-H, Backspace	Delete character before cursor

Keystroke	Action
Ctrl-I, Tab	Next completion
Ctrl-J, Ctrl-M, Enter	Finish the line entry
Ctrl-K	Delete from cursor to end of line
Ctrl-L	Clear screen
Ctrl-N, Down	Next match from history
Ctrl-P, Up	Previous match from history
Ctrl-R	Reverse Search history (Ctrl-S forward, Ctrl-G cancel)
Ctrl-T	Transpose previous character with current character
Ctrl-U	Delete from start of line to cursor
Ctrl-V	Insert any special character without performing its associated action
Ctrl-W	Delete word leading up to cursor (using white space as a word boundary)
Ctrl-X Ctrl-U	Undo
Ctrl-Y	Paste from Yank buffer
Ctrl-Y	Paste from Yank buffer (Meta-Y to paste next yank instead)
Ctrl-Z	Suspend (Unix only)
Ctrl-_	Undo
Meta-0, 1, ..., -	Specify the digit to the argument. – starts a negative argument.
Meta-<	Move to first entry in history
Meta->	Move to last entry in history
Meta-B, Alt-Left	Move cursor to previous word
Meta-Backspace	Kill from the start of the current word, or, if between words, to the start of the previous word
Meta-C	Capitalize the current word
Meta-D	Delete forwards one word
Meta-F, Alt-Right	Move cursor to next word
Meta-L	Lower-case the next word
Meta-T	Transpose words

Keystroke	Action
Meta-U	Upper-case the next word
Meta-Y	See Ctrl-Y
Ctrl-S	Insert a new line

12.4 Bundler

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

```
deno bundle https://deno.land/std@0.156.0/examples/colors.ts colors.bundle.js
Bundle https://deno.land/std@0.156.0/examples/colors.ts
Download https://deno.land/std@0.156.0/examples/colors.ts
Download https://deno.land/std@0.156.0/fmt/colors.ts
Emit "colors.bundle.js" (9.83KB)
```

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

12.4.1 Bundling for the Web

The output of deno bundle is intended for consumption in Deno and not for use in a web browser or other runtimes. That said, depending on the input it may work in other environments.

If you wish to bundle for the web, we recommend other solutions such as esbuild.

12.5 Compiling Executables

deno compile [--output <OUT>] <SRC> will compile the script into a self-contained executable.

```
> deno compile https://deno.land/std/examples/welcome.ts
```

If you omit the OUT parameter, the name of the executable file will be inferred.

12.5.1 Flags

As with deno install, the runtime flags used to execute the script must be specified at compilation time. This includes permission flags.

```
> deno compile --allow-read --allow-net https://deno.land/std/http/file_server.ts
```

Script arguments can be partially embedded.

```
> deno compile --allow-read --allow-net https://deno.land/std/http/file_server.ts -p 8080
> ./file_server --help
```

12.5.2 Cross Compilation

You can compile binaries for other platforms by adding the --target CLI flag. Deno currently supports compiling to Windows x64, macOS x64, macOS ARM and Linux x64. Use deno compile --help to list the full values for each compilation target.

12.5.3 Unavailable in executables

- Workers
- Dynamic Imports

12.9.4.1 Boolean lists

Boolean lists provide a way to execute additional commands based on the exit code of the initial command. They separate commands using the && and || operators.

The && operator provides a way to execute a command and if it succeeds (has an exit code of 0) it will execute the next command:

```
deno run --allow-read=. --allow-write=. collect.ts && deno run --allow-read=. analyze.ts
```

The || operator is the opposite. It provides a way to execute a command and only if it fails (has a non-zero exit code) it will execute the next command:

```
deno run --allow-read=. --allow-write=. collect.ts || deno run play_sad_music.ts
```

12.9.4.2 Sequential lists

Sequential lists are similar to boolean lists, but execute regardless of whether the previous command in the list passed or failed. Commands are separated with a semi-colon (;).

```
deno run output_data.ts ; deno run --allow-net server/main.ts
```

12.9.4.3 Async commands

Async commands provide a way to make a command execute asynchronously. This can be useful when starting multiple processes. To make a command asynchronous, add an & to the end of it. For example the following would execute sleep 1 && deno run --allow-net client/main.ts and deno run --allow-net server/main.ts at the same time:

```
sleep 1 && deno run --allow-net client/main.ts & deno run --allow-net server/main.ts
```

12.9.4.4 Environment variables

Environment variables are defined like the following:

```
export VAR_NAME=value
```

Here's an example of using one in a task with shell variable substitution and then with it being exported as part of the environment of the spawned Deno process (note that in the JSON configuration file the double quotes would need to be escaped with backslashes):

```
export VAR=hello && echo $VAR && deno eval "console.log('Deno: ' +
Deno.env.get('VAR'))"
```

Would output:

```
hello
Deno: hello
```

12.9.4.4.1 Setting environment variables for a command

To specify environment variable(s) before a command, list them like so:

```
VAR=hello VAR2=bye deno run main.ts
```

This will use those environment variables specifically for the following command.

12.9.4.5 Shell variables

Shell variables are similar to environment variables, but won't be exported to spawned commands. They are defined with the following syntax:

```
VAR_NAME=value
```

If we use a shell variable instead of an environment variable in a similar example to what's shown in the previous "Environment variables" section:

```
VAR=hello && echo $VAR && deno eval "console.log('Deno: ' + Deno.env.get('VAR'))"
```

We will get the following output:

```
hello
Deno: undefined
```

Shell variables can be useful when we want to re-use a value, but don't want it available in any spawned processes.

12.9.4.6 Pipelines

Pipelines provide a way to pipe the output of one command to another.

The following command pipes the stdout output "Hello" to the stdin of the spawned Deno process:

```
│ │ └── https://deno.land/std@0.67.0/path/_util.ts *
│ ├─┬ https://deno.land/std@0.67.0/path/common.ts (1.14KB)
│ │ └─┬ https://deno.land/std@0.67.0/path/separator.ts (264B)
│ │   └── https://deno.land/std@0.67.0/path/_constants.ts *
│ ├── https://deno.land/std@0.67.0/path/separator.ts *
│ ├── https://deno.land/std@0.67.0/path/_interface.ts *
│ └─┬ https://deno.land/std@0.67.0/path/glob.ts (8.12KB)
│   ├── https://deno.land/std@0.67.0/path/_constants.ts *
│   ├── https://deno.land/std@0.67.0/path/mod.ts *
│   └── https://deno.land/std@0.67.0/path/separator.ts *
├─┬ https://deno.land/std@0.67.0/http/server.ts (10.23KB)
│ ├── https://deno.land/std@0.67.0/encoding/utf8.ts (433B)
│ ├─┬ https://deno.land/std@0.67.0/io/bufio.ts (21.15KB)
│ │ ├── https://deno.land/std@0.67.0/bytes/mod.ts (4.34KB)
│ │ └── https://deno.land/std@0.67.0/_util/assert.ts *
│ ├── https://deno.land/std@0.67.0/_util/assert.ts *
│ ├─┬ https://deno.land/std@0.67.0/async/mod.ts (202B)
│ │ ├── https://deno.land/std@0.67.0/async/deferred.ts (1.03KB)
│ │ ├── https://deno.land/std@0.67.0/async/delay.ts (279B)
│ │ ├─┬ https://deno.land/std@0.67.0/async/mux_async_iterator.ts (1.98KB)
│ │ │ └── https://deno.land/std@0.67.0/async/deferred.ts *
│ │ └── https://deno.land/std@0.67.0/async/pool.ts (1.58KB)
│ └─┬ https://deno.land/std@0.67.0/http/_io.ts (11.25KB)
│   ├── https://deno.land/std@0.67.0/io/bufio.ts *
│   ├─┬ https://deno.land/std@0.67.0/textproto/mod.ts (4.52KB)
│   │ ├── https://deno.land/std@0.67.0/io/bufio.ts *
│   │ ├── https://deno.land/std@0.67.0/bytes/mod.ts *
│   │ └── https://deno.land/std@0.67.0/encoding/utf8.ts *
│   ├── https://deno.land/std@0.67.0/_util/assert.ts *
│   ├── https://deno.land/std@0.67.0/encoding/utf8.ts *
│   ├── https://deno.land/std@0.67.0/http/server.ts *
│   └── https://deno.land/std@0.67.0/http/http_status.ts (5.93KB)
├─┬ https://deno.land/std@0.67.0/flags/mod.ts (9.54KB)
│ └── https://deno.land/std@0.67.0/_util/assert.ts *
└── https://deno.land/std@0.67.0/_util/assert.ts *
```

Dependency inspector works with any local or remote ES modules.

12.7.1 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"
```

12.8 Linter

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

```
# lint all JS/TS files in the current directory and subdirectories
deno lint
# lint specific files
```

```
deno lint myfile1.ts myfile2.ts
# lint all JS/TS files in specified directory and subdirectories
deno lint src/
# print result as JSON
deno lint --json
# read from stdin
cat file.ts | deno lint -
```

For more detail, run deno lint --help.

12.8.1 Available rules

For a complete list of supported rules visit [the deno_lint rule documentation](https://lint.deno.land/).

12.8.2 Ignore directives

12.8.2.1 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 statement 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 {
  // ...
}
```

You can also ignore certain diagnostics in the whole file

```
// deno-lint-ignore-file no-explicit-any no-empty

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

12.8.2.2 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) {
  // ...
}
```

12.8.3 Configuration

Starting with Deno v1.14 a linter can be customized using either [a configuration file](#) or following CLI flags:

- --rules-tags - List of tag names that will be run. Empty list disables all tags and will only use rules from include. Defaults to "recommended".
- --rules-exclude - List of rule names that will be excluded from configured tag sets. If the same rule is in include it will be run.
- --rules-include - List of rule names that will be run. Even if the same rule is in exclude it will be run.

12.9 [Task Runner](#)

deno task was introduced in Deno v1.20 and is unstable. It may drastically change in the future.

deno task provides a cross platform way to define and execute custom commands specific to a codebase.

To get started, define your commands in your codebase's [Deno configuration file](#) under a "tasks" key.

For example:

```
{
  "tasks": {
    "data": "deno task collect && deno task analyze",
    "collect": "deno run --allow-read=. --allow-write=. scripts/collect.js",
    "analyze": "deno run --allow-read=. scripts/analyze.js"
  }
}
```

12.9.1 Listing tasks

To get an output showing all the defined tasks, run:

```
deno task
```

12.9.2 Executing a task

To execute a specific task, run:

```
deno task task-name [additional args]...
```

In the example above, to run the data task we would do:

```
deno task data
```

12.9.3 Specifying the current working directory

By default, deno task executes commands with the directory of the Deno configuration file (ex. deno.json) as the current working directory. This allows tasks to use relative paths and continue to work regardless of where in the directory tree you happen to execute the deno task from. In some scenarios, this may not be desired and this behavior can be overridden by providing a --cwd <path> flag.

For example, given a task called wasmbuild in a deno.json file:

```
# use the sub directory project1 as the cwd for the task
deno task --cwd project1 wasmbuild
# use the cwd (project2) as the cwd for the task
cd project2 && deno task --cwd . wasmbuild
```

Note: Be sure to provide this flag before the task name.

12.9.4 Syntax

deno task uses a cross platform shell that's a subset of sh/bash to execute defined tasks.

```
// Similar to longer form, with a named test function as a second argument.
Deno.bench({ permissions: { read: true } }, function helloWorld6() {
  new URL("https://deno.land");
});
```

12.11.2.1 Async functions

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

```
Deno.bench("async hello world", async () => {
  await 1;
});
```

12.11.3 Grouping and baselines

When registering a bench case, it can be assigned to a group, using Deno.BenchDefinition.group option:

```
// url_bench.ts
Deno.bench("url parse", { group: "url" }, () => {
  new URL("https://deno.land");
});
```

It is useful to assign several cases to a single group and compare how they perform against a "baseline" case.

In this example we'll check how performant is Date.now() compared to performance.now(), to do that we'll mark the first case as a "baseline" using Deno.BechnDefintion.baseline option:

```
// time_bench.ts
Deno.bench("Date.now()", { group: "timing", baseline: true }, () => {
  Date.now();
});

Deno.bench("performance.now()", { group: "timing" }, () => {
  performance.now();
});
```

```
$ deno bench --unstable time_bench.ts
cpu: Apple M1 Max
runtime: deno 1.21.0 (aarch64-apple-darwin)

file:///dev/deno/time_bench.ts
benchmark                  time (avg)             (min … max)       p75       p99
p995
-------------------------------------------------------- -----------------------------
Date.now()           125.24 ns/iter (118.98 ns … 559.95 ns) 123.62 ns 150.69 ns 156.63 ns
performance.now()      2.67 µs/iter      (2.64 µs … 2.82 µs)     2.67 µs     2.82 µs     2.82
µs

summary
  Date.now()
   21.29x times faster than performance.now()
```

You can specify multiple groups in the same file.

12.11.4 Running benchmarks

```
echo Hello | deno run main.ts
```

To pipe stdout and stderr, use |& instead:

```
deno eval 'console.log(1); console.error(2);' |& deno run main.ts
```

12.9.4.7 Command substitution

The $(command) syntax provides a way to use the output of a command in other commands that get executed.

For example, to provide the output of getting the latest git revision to another command you could do the following:

```
deno run main.ts $(git rev-parse HEAD)
```

Another example using a shell variable:

```
REV=$(git rev-parse HEAD) && deno run main.ts $REV && echo $REV
```

12.9.4.8 Negate exit code

To negate the exit code, add an exclamation point and space before a command:

```
# change the exit code from 1 to 0
! deno eval 'Deno.exit(1);'
```

12.9.4.9 Redirects

Redirects provide a way to pipe stdout and/or stderr to a file.

For example, the following redirects stdout of deno run main.ts to a file called file.txt on the file system:

```
deno run main.ts > file.txt
```

To instead redirect stderr, use 2>:

```
deno run main.ts 2> file.txt
```

To redirect both stdout and stderr, use &>:

```
deno run main.ts &> file.txt
```

To append to a file, instead of overwriting an existing one, use two right angle brackets instead of one:

```
deno run main.ts >> file.txt
```

Suppressing either stdout, stderr, or both of a command is possible by redirecting to /dev/null. This works in a cross platform way including on Windows.

```
# suppress stdout
deno run main.ts > /dev/null
# suppress stderr
deno run main.ts 2> /dev/null
# suppress both stdout and stderr
deno run main.ts &> /dev/null
```

Note that redirecting input and multiple redirects are currently not supported.

12.9.4.10 Future syntax

We are planning to support glob expansion in the future.

12.9.5 Built-in commands

deno task ships with several built-in commands that work the same out of the box on Windows, Mac, and Linux.

- cp - Copies files.
- mv - Moves files.
- rm - Remove files or directories.
- Ex: rm -rf [FILE]... - Commonly used to recursively delete files or directories.
- mkdir - Makes directories.
- Ex. mkdir -p DIRECTORY... - Commonly used to make a directory and all its parents with no error if it exists.
- pwd - Prints the name of the current/working directory.

- sleep - Delays for a specified amount of time.
- Ex. sleep 1 to sleep for 1 second or sleep 0.5 to sleep for half a second
- echo - Displays a line of text.
- cat - Concatenates files and outputs them on stdout. When no arguments are provided it reads and outputs stdin.
- exit - Causes the shell to exit.
- xargs - Builds arguments from stdin and executes a command.

If you find a useful flag missing on a command or have any suggestions for additional commands that should be supported out of the box, then please open an issue on the deno_task_shell repo.

Note that if you wish to execute any of these commands in a non-cross platform way on Mac or Linux, then you may do so by running it through sh: sh -c <command> (ex. sh -c cp source destination).

12.10 Vendoring Dependencies

deno vendor <specifiers>... will download all remote dependencies of the specified modules into a local vendor folder. For example:

```
# Vendor the remote dependencies of main.ts
$ deno vendor main.ts

# Example file system tree
$ tree
.
├── main.ts
└── vendor
    ├── deno.land
    ├── import_map.json
    └── raw.githubusercontent.com

# Check the directory into source control
$ git add -u vendor
$ git commit
```

To then use the vendored dependencies in your program, just add --import-map=vendor/import_map.json to your Deno invocations. You can also add --no-remote to your invocation to completely disable fetching of remote modules to ensure it's using the modules in the vendor directory.

```
deno run --no-remote --import-map=vendor/import_map.json main.ts
```

Note that you may specify multiple modules and remote modules when vendoring.

```
deno vendor main.ts test.deps.ts https://deno.land/std/path/mod.ts
```

Run deno vendor --help for more details.

12.11 Benchmarking

⚠ deno bench was introduced in Deno v1.20 and currently requires --unstable flag.

Deno has a built-in benchmark runner that you can use for checking performance of JavaScript or TypeScript code.

12.11.1 Quickstart

Firstly, let's create a file url_bench.ts and register a bench using the Deno.bench() function.

```
// url_bench.ts
```

```
Deno.bench("URL parsing", () => {
  new URL("https://deno.land");
});
```

Secondly, run the benchmark using the deno bench subcommand.

```
deno bench --unstable url_bench.ts
cpu: Apple M1 Max
runtime: deno 1.21.0 (aarch64-apple-darwin)

file:///dev/deno/url_bench.ts
benchmark          time (avg)             (min … max)       p75       p99
p995
-------------------------------------------------- -----------------------------
URL parsing     17.29 µs/iter   (16.67 µs … 153.62 µs)   17.25 µs   18.92 µs   22.25 µs
```

12.11.2 Writing benchmarks

To define a benchmark you need to register it with a call to the Deno.bench API. There are multiple overloads of this API to allow for the greatest flexibility and easy switching between the forms (eg. when you need to quickly focus a single bench for debugging, using the only: true option):

```
// Compact form: name and function
Deno.bench("hello world #1", () => {
  new URL("https://deno.land");
});

// Compact form: named function.
Deno.bench(function helloWorld3() {
  new URL("https://deno.land");
});

// Longer form: test definition.
Deno.bench({
  name: "hello world #2",
  fn: () => {
    new URL("https://deno.land");
  },
});

// Similar to compact form, with additional configuration as a second argument.
Deno.bench("hello world #4", { permissions: { read: true } }, () => {
  new URL("https://deno.land");
});

// Similar to longer form, with test function as a second argument.
Deno.bench(
  { name: "hello world #5", permissions: { read: true } },
  () => {
    new URL("https://deno.land");
  },
);
```

```
jobs:
  build:
    runs-on: ${{ matrix.os }}
    continue-on-error: ${{ matrix.canary }} # Continue in case the canary run does not
succeed
    strategy:
      matrix:
        os: [ ubuntu-22.04, macos-12, windows-2022 ]
        deno-version: [ v1.x ]
        canary: [ false ]
        include:
          - deno-version: canary
            os: ubuntu-22.04
            canary: true
```

13.3 Speeding up Deno pipelines

13.3.1 Reducing repetition

In cross-platform runs, certain steps of a pipeline do not need to run for each OS necessarily. For example, generating the same test coverage report on Linux, MacOS and Windows is a bit redundant. You can use the if conditional keyword of GitHub Actions in these cases. The example below shows how to run code coverage generation and upload steps only on the ubuntu (Linux) runner:

```
- name: Generate coverage report
  if: matrix.os == 'ubuntu-22.04'
  run: deno coverage --lcov cov > cov.lcov

- name: Upload coverage to Coveralls.io
  if: matrix.os == 'ubuntu-22.04'
  # Any code coverage service can be used, Coveralls.io is used here as an example.
  uses: coverallsapp/github-action@master
  with:
    github-token: ${{ secrets.GITHUB_TOKEN }} # Generated by GitHub.
    path-to-lcov: cov.lcov
```

13.3.2 Caching dependencies

As a project grows in size, more and more dependencies tend to be included. Deno will download these dependencies during testing and if a workflow is run many times a day, this can become a time-consuming process. A common solution to speed things up is to cache dependencies so that they do not need to be downloaded anew.

Deno stores dependencies locally in a cache directory. In a pipeline the cache can be preserved between workflows by setting the DENO_DIR environment variable and adding a caching step to the workflow:

```
# Set DENO_DIR to an absolute or relative path on the runner.
env:
  DENO_DIR: my_cache_directory

steps:
  - name: Cache Deno dependencies
    uses: actions/cache@v2
    with:
```

To run a benchmark, call deno bench with the file that contains your bench function. You can also omit the file name, in which case all benchmarks in the current directory (recursively) that match the glob {*_,*.,}bench.{ts, tsx, mts, js, mjs, jsx, cjs, cts} will be run. If you pass a directory, all files in the directory that match this glob will be run.

The glob expands to:

- files named bench.{ts, tsx, mts, js, mjs, jsx, cjs, cts},
- or files ending with .bench.{ts, tsx, mts, js, mjs, jsx, cjs, cts},
- or files ending with _bench.{ts, tsx, mts, js, mjs, jsx, cjs, cts}

```
# Run all benches in the current directory and all sub-directories
deno bench

# Run all benches in the util directory
deno bench util/

# Run just my_bench.ts
deno bench my_bench.ts
```

⚠ If you want to pass additional CLI arguments to the bench files use -- to inform Deno that remaining arguments are scripts arguments.

```
# Pass additional arguments to the bench file
deno bench my_test.ts -- -e --foo --bar
```

deno bench uses the same permission model as deno run and therefore will require, for example, --allow-write to write to the file system during benching.

To see all runtime options with deno bench, you can reference the command line help:

```
deno help bench
```

12.11.5 Filtering

There are a number of options to filter the benches you are running.

12.11.5.1 Command line filtering

Benches can be run individually or in groups using the command line --filter option.

The filter flags accept a string or a pattern as value.

Assuming the following benches:

```
Deno.bench({
  name: "my-bench",
  fn: () => {/* bench function zero */},
});
Deno.bench({
  name: "bench-1",
  fn: () => {/* bench function one */},
});
Deno.bench({
  name: "bench2",
  fn: () => {/* bench function two */},
});
```

This command will run all of these benches because they all contain the word "bench".

```
deno bench --filter "bench" benchmarks/
```

On the flip side, the following command uses a pattern and will run the second and third benchmarks.

```
deno bench --filter "/bench-*\d/" benchmarks/
```

To let Deno know that you want to use a pattern, wrap your filter with forward-slashes like the JavaScript syntactic sugar for a regex.

12.11.5.2 Bench definition filtering

Within the benches themselves, you have two options for filtering.

12.11.5.2.1 Filtering out (ignoring these benches)

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

```
Deno.bench({
  name: "bench windows feature",
  ignore: Deno.build.os !== "windows",
  fn() {
    // do windows feature
  },
});
```

12.11.5.2.2 Filtering in (only run these benches)

Sometimes you may be in the middle of a performance problem within a large bench class and you would like to focus on just that single bench and ignore the rest for now. For this you can use the `only` option to tell the benchmark harness to only run benches with this set to true. Multiple benches can set this option. While the benchmark run will report on the success or failure of each bench, the overall benchmark run will always fail if any bench is flagged with `only`, as this is a temporary measure only which disables nearly all of your benchmarks.

```
Deno.bench({
  name: "Focus on this bench only",
  only: true,
  fn() {
    // bench complicated stuff
  },
});
```

13 Continuous Integration

Deno's built-in tools make it easy to set up Continuous Integration (CI) pipelines for your projects. Testing, linting and formatting of code can all be done with the corresponding commands `deno test`, `deno lint` and `deno fmt`. In addition, you can generate code coverage reports from test results with `deno coverage` in pipelines.

On this page we will discuss:

- Setting up a basic pipeline
- Cross-platform workflows
- Speeding up Deno pipelines
    - Reducing repetition
    - Caching dependencies

13.1 Setting up a basic pipeline

This page will show you how to set up basic pipelines for Deno projects in GitHub Actions. The concepts explained on this page largely apply to other CI providers as well, such as Azure Pipelines, CircleCI or GitLab.

Building a pipeline for Deno generally starts with checking out the repository and installing Deno:

```
name: Build

on: push

jobs:
  build:
    runs-on: ubuntu-22.04
    steps:
      - uses: actions/checkout@v3
      - uses: denoland/setup-deno@v1.1.0
        with:
          deno-version: v1.x # Run with latest stable Deno.
```

To expand the workflow just add any of the `deno` subcommands that you might need:

```
      # Check if the code is formatted according to Deno's default
      # formatting conventions.
      - run: deno fmt --check

      # Scan the code for syntax errors and style issues. If
      # you want to use a custom linter configuration you can add a configuration file with --
config <myconfig>
      - run: deno lint

      # Run all test files in the repository and collect code coverage. The example
      # runs with all permissions, but it is recommended to run with the minimal permissions
your program needs (for example --allow-read).
      - run: deno test --allow-all --coverage=cov/

      # This generates a report from the collected coverage in `deno test --coverage`. It is
      # stored as a .lcov file which integrates well with services such as Codecov, Coveralls
and Travis CI.
      - run: deno coverage --lcov cov/ > cov.lcov
```

13.2 Cross-platform workflows

As a Deno module maintainer, you probably want to know that your code works on all of the major operating systems in use today: Linux, MacOS and Windows. A cross-platform workflow can be achieved by running a matrix of parallel jobs, each one running the build on a different OS:

```
jobs:
  build:
    runs-on: ${{ matrix.os }}
    strategy:
      matrix:
        os: [ ubuntu-22.04, macos-12, windows-2022 ]
    steps:
      - run: deno test --allow-all --coverage cov/
```

Note: GitHub Actions has a known issue with handling Windows-style line endings (CRLF). This may cause issues when running `deno fmt` in a pipeline with jobs that run on `windows`. To prevent this, configure the Actions runner to use Linux-style line endings before running the `actions/checkout@v3` step:

```
git config --system core.autocrlf false
git config --system core.eol lf
```

If you are working with experimental or unstable Deno APIs, you can include a matrix job running the canary version of Deno. This can help to spot breaking changes early on:

```
    let someCondition = true;
    assert(someCondition);
  },
});
```

You will see a code lens like the following just above the test:

```
▶ Run Test
```

This is a link that if you click it, the extension will start up the Deno CLI to run the test for you and display the output. Based on your other settings, the extension will try to run your test with the same settings. If you need to adjust the arguments provided when doing deno test, you can do so by setting the deno.codeLens.testArgs setting.

The extension will also try to track if in the same module you destructure the Deno.test function or assign it to a variable. So you can do something like this and still have the code lens work:

```
const { test: denoTest } = Deno;

denoTest({
  name: "example test",
  fn() {},
});
```

If you want to disable this feature, you can do so by unsetting the Deno > CodeLens: Test/deno.codeLens.test setting.

14.0.7 Using the debugger

[TBC]

14.0.8 Tasks

The extension communicates directly to the language server, but for some development tasks, you might want to execute the CLI directly. The extension provides a task definition for allowing you to create tasks that execute the deno CLI from within the editor.

14.0.8.1 Deno CLI tasks

The template for the Deno CLI tasks has the following interface, which can be configured in a tasks.json within your workspace:

```
interface DenoTaskDefinition {
  type: "deno";
  // This is the `deno` command to run (e.g. `run`, `test`, `cache`, etc.)
  command: string;
  // Additional arguments pass on the command line
  args?: string[];
  // The current working directory to execute the command
  cwd?: string;
  // Any environment variables that should be set when executing
  env?: Record<string, string>;
}
```

Several of the commands that are useful in the editor are configured as templates and can be added to your workspace by select Tasks: Configure Task in the command palette and searching for deno tasks.

And example of what a deno run mod.ts would look like in a tasks.json:

```
{
  "version": "2.0.0",
  "tasks": [
    {
      "type": "deno",
```

```
      path: ${{ env.DENO_DIR }}
      key: my_cache_key
```

At first, when this workflow runs the cache is still empty and commands like deno test will still have to download dependencies, but when the job succeeds the contents of DENO_DIR are saved and any subsequent runs can restore them from cache instead of re-downloading.

There is still an issue in the workflow above: at the moment the name of the cache key is hardcoded to my_cache_key, which is going to restore the same cache every time, even if one or more dependencies are updated. This can lead to older versions being used in the pipeline even though you have updated some dependencies. The solution is to generate a different key each time the cache needs to be updated, which can be achieved by using a lockfile and by using the hashFiles function provided by GitHub Actions:

```
key: ${{ hashFiles('lock.json') }}
```

To make this work you will also need a have a lockfile in your Deno project, which is discussed in detail here. Now, if the contents of lock.json are changed, a new cache will be made and used in subsequent pipeline runs thereafter.

To demonstrate, let's say you have a project that uses the logger from deno.land/std:

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

In order to increment this version, you can update the import statement and then reload the cache and update the lockfile locally:

```
deno cache --reload --lock=lock.json --lock-write deps.ts
```

You should see changes in the lockfile's contents after running this. When this is committed and run through the pipeline, you should then see the hashFiles function saving a new cache and using it in any runs that follow.

13.3.2.1 Clearing the cache

Occasionally you may run into a cache that has been corrupted or malformed, which can happen for various reasons. There is no option in GitHub Actions UI to clear a cache yet, but to create a new cache you can simply change the name of the cache key. A practical way of doing so without having to forcefully change your lockfile is to add a variable to the cache key name, which can be stored as a GitHub secret and which can be changed if a new cache is needed:

```
key: ${{ secrets.CACHE_VERSION }}-${{ hashFiles('lock.json') }}
```

14 Using Visual Studio Code

In this section we are going to go into depth about developing Deno applications using Visual Studio Code and the official vscode_deno extension.

14.0

14.0.1 Installing

The vscode extension integrates directly to the Deno CLI using the language server protocol. This helps ensure that the information you get about your code aligns to how that code will work when you try to run it under the Deno CLI.

The Deno extension is installed like other extensions in vscode, by browsing the extensions in vscode and choosing to install the Deno extension. Or if you have vscode installed, you can view the extension via this link and install it if you haven't already done so.

Once you install the extension for the first time, you should receive a splash page that welcomes you to the extension. (If you missed it, or want to see it again, just use the Deno: Welcome command from the command palette.)

14.0.2 Configuring the extension

The following sections will detail out how to configure the extension to work best for you and will cover most of the settings available.

14.0.2.1 Deno enabling a workspace

We realize that not every project you might edit with vscode is a Deno project. By default, vscode comes with a built-in TypeScript/JavaScript language service which is used when editing TypeScript or JavaScript files.
In order to have support for Deno APIs as well as the ability to resolve modules as the Deno CLI does, you need to enable Deno for the workspace. The most direct way to do this is to use the Deno: Initialize Workspace Configuration from the vscode command palette. This will activate a helper which will ask if you want to also enable linting and the Deno unstable APIs for the project. This command will instruct vscode to store these settings in the workspace configuration (your workspace root .vscode/settings.json). Once the helper is finished, you will get a notification that Deno is setup for the project.
These settings (and other settings) are available via the vscode settings panel. In the panel the setting is Deno: Enable and when manually editing the JSON, the setting is deno.enable.
⚠ vscode has user and workspace settings. You probably don't want to enable Deno in the user settings, as then by default, every workspace will be Deno enabled.
When a project is enabled, the extension will get information directly from the installed Deno CLI. The extension will also mute the built-in TypeScript/JavaScript extension.

14.0.2.2 Partially Deno enabling a workspace

While vscode supports Workspace Folders, they can be challenging to configure and use. Because of this, the option Deno: Enable Paths has been introduced (or "deno.enablePaths" if manually editing). In a given workspace (or workspace folder), sub-paths can be enabled for Deno, while code outside those paths will be not be enabled and the vscode built-in JavaScript/TypeScript language server will be used.
For example if you have a project like this:

```
project
├── worker
└── front_end
```

Where you only want to enabled the worker path (and its subpaths) to be Deno enabled, you will want to add ./worker to the list of Deno: Enable Paths in the configuration.

14.0.2.3 Using linting

The same engine that provides the diagnostics when using deno lint can also be used via the extension. By enabling the Deno: Lint setting in the settings panel (or deno.lint if editing settings in JSON) the editor should start to display lint "warnings" in your code. See the Linter section for more information on how to use the Deno linter.

14.0.2.4 Using import maps

It is possible to use import maps in the editor. The option Deno: Import Map (or deno.importMap if manually editing) should be set to the value of the import map file. If the path is a relative path, it will be resolved relative to the root of the workspace.

14.0.2.5 Using a configuration file

Typically a configuration file is not required for a Deno project. There are a few scenarios though where it might be useful, and if you want to have the same settings applied as when specifying the --config option on the command line, the Deno: Config option can be used (or deno.config if manually editing).
The Deno extension will also auto-identify and apply a deno.jsonc or deno.json by looking in the workspace root for the configuration file and applying it. Manually specifying a Deno: Config option will override this automatic behavior.

14.0.2.6 Using formatting

The Deno CLI comes with a built-in formatter which can be accessed using deno fmt but can also be configured to be used by vscode. Deno should be on the drop down list for the Editor: Default formatter setting (or if you are editing settings manually, it would be "editor.defaultFormatter": "denoland.vscode-deno").

See the Code formatter for more information on how to use the formatter.

14.0.2.7 Setting a path to the Deno CLI

The extension looks for the Deno CLI executable in the host's PATH, but sometimes that isn't desirable and the Deno: Path can be set (or deno.path if manually editing) to point to the Deno executable. If the path provided is relative, it will be resolved relative to the root of the workspace.

14.0.3 Import suggestions

When attempting to import a module, the extension will offer suggestions to complete the import. Local relative files will be included in the suggestions, plus also any cached remote files.
The extension supports registry auto-completions, where a remote registry/website of modules can optionally provide metadata that allows a client to discover modules. By default, the extension will check hosts/origins to see if they support suggestions, and if it does, the extension will prompt you to see if you want to enable it. This behavior can be changed by unsetting Deno > Suggest > Imports: Auto Discover (or deno.suggest.imports.autoDiscover if manually editing).
Individual hosts/origins can be enabled or disabled by editing the Deno > Suggest > Imports: Hosts/deno.suggest.imports.hosts setting in the appropriate settings.json.

14.0.4 Caching remote modules

Deno supports remote modules and will fetch remote modules and store them locally in a cache. When you do something like deno run, deno test, deno info or deno cache on the command line, the Deno CLI will go and try to fetch any remote modules and their dependencies and populate the cache.
While developing code in the editor, if the module is not in the cache, you will get a diagnostic like Uncached or missing remote URL: "https://deno.land/example/mod.ts" for any missing remote modules. Deno will not automatically try to cache the module, unless it is a completion from a registry import suggestion (see above).
In addition to running a command on a command line, the extension provides ways to cache dependencies within the editor. A missing dependency will have a quick fix which is to have Deno try to cache the dependency. Fixes can be accessed by pressing CTRL . or ⌘ . when the editor is positioned in the import specifier, or hovering over the specifier and selecting Quick Fix....
There is also the Deno: Cache Dependencies command in the command palette which will attempt to cache any dependencies of the module currently active in the editor.

14.0.5 Code lenses

The language server currently supports several code lenses (actionable contextual information interspersed in the code) that allow you to get greater insight into the code. Most are disabled by default, but can easily be enabled:

- Deno > Code Lens: Implementations/deno.codeLens.implementations - Provides a lens that will list out any implementations of an item elsewhere in the code.
- Deno > Code Lens: References/deno.codeLens.references - Provides a lens that will list out any references to an item elsewhere in the code.
- Deno > Code Lens: References All Functions/deno.codeLens.referencesAllFunctions - Provides a lens that will list out all references to all functions in the code. All functions are excluded from just References mention above.

14.0.6 Testing code lens

The Deno CLI includes a built-in testing API available under Deno.test. The extension and language server have a code lens enabled by default which provides the ability to run a test from within the editor.
When you have a block of code that provides a test, like:

```
import { assert } from "https://deno.land/std@0.156.0/testing/asserts.ts";

Deno.test({
  name: "a test case",
  fn() {
```

When the language server is started, a LanguageServer instance is created which holds all of the state of the language server. It also defines all of the methods that the client calls via the Language Server RPC protocol.

15.1.2 Settings

There are several settings that the language server supports for a workspace:

- deno.enable
- deno.enablePaths
- deno.cache
- deno.certificateStores
- deno.config
- deno.importMap
- deno.internalDebug
- deno.codeLens.implementations
- deno.codeLens.references
- deno.codeLens.referencesAllFunctions
- deno.codeLens.test
- deno.suggest.completeFunctionCalls
- deno.suggest.names
- deno.suggest.paths
- deno.suggest.autoImports
- deno.suggest.imports.autoDiscover
- deno.suggest.imports.hosts
- deno.lint
- deno.tlsCertificate
- deno.unsafelyIgnoreCertificateErrors
- deno.unstable

There are settings that are supported on a per resource basis by the language server:

- deno.enable
- deno.enablePaths
- deno.codeLens.test

There are several points in the process where Deno analyzes these settings. First, when the initialize request from the client, the initializationOptions will be assumed to be an object that represents the deno namespace of options. For example, the following value:

```
{
  "enable": true,
  "unstable": true
}
```

Would enable Deno with the unstable APIs for this instance of the language server.

When the language server receives a workspace/didChangeConfiguration notification, it will assess if the client has indicated if it has a workspaceConfiguration capability. If it does, it will send a workspace/configuration request which will include a request for the workspace configuration as well as the configuration of all URIs that the language server is currently tracking.

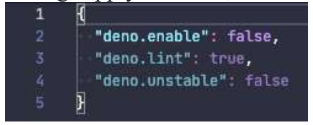
If the client has the workspaceConfiguration capability, the language server will send a configuration request for the URI when it received the textDocument/didOpen notification in order to get the resources specific settings.

```
        "command": "run",
        "args": [
          "mod.ts"
        ],
        "problemMatcher": [
          "$deno"
        ],
        "label": "deno: run"
      }
    ]
}
```

14.0.9 Workspace folders

The Deno language server and this extension supports multi-root workspaces configuration, where certain settings can be applied to workspace folders within a workspace.

When you add folders to your workspace and open the settings, you will have access to the per folder settings. If you look at the .vscode/settings.json in a folder, you will see a visual indication of what settings apply to folder, versus those that come from the workspace configuration:



```
{
  "deno.enable": false,
  "deno.lint": true,
  "deno.unstable": false
}
```

14.0.9.1 Workspace folder settings

These are the settings that can be set on a workspace folder. The rest of the settings currently only apply to the workspace:

- deno.enable - Controls if the Deno Language Server is enabled. When enabled, the extension will disable the built-in vscode JavaScript and TypeScript language services, and will use the Deno language server instead. boolean, default false
- deno.enablePaths - Controls if the Deno Language Server is enabled for only specific paths of the workspace folder. Defaults to an empty list.
- deno.codeLens.test - Controls if the test code lens is enabled. boolean, default true
- deno.codeLens.testArgs - The list of arguments that are passed to deno test when activating a test code lens. string array, default ["--allow-all"]

14.0.9.2 Mixed-Deno projects

While you can use this feature to enable mixed-Deno projects, you might want to consider partially Deno enabling a workspace. But with this feature, you can have a mixed Deno project, where some of the workspace folders are Deno enabled and some are not. This is useful when creating a project that might have a front-end component, where you want a different configuration for that front end code.

In order to support this, you would create a new workspace (or add a folder to an existing workspace) and in the settings configure one of the folders to have deno.enable set to true and one set to false. Once you save the workspace configuration, you notice that the Deno language server only applies diagnostics to the enabled folders, while the other folder will use the built-in TypeScript compiler of vscode to supply diagnostics for TypeScript and JavaScript files.

14.0.10 Using a development container

Using a development container with vscode is a great way to have an isolated development environment without having to worry about having to install the Deno CLI on your local system.

To use development containers, you need to have a few prerequisites installed:

- Docker Desktop
- Visual Studio Code or Visual Studio Code Insiders

- Remote Development extension pack

The way a development container is configured is by having a .devcontainer folder as part of the workspace with configuration information in the folder. If you are opening a project that already contains a dev container with Deno, you will be prompted to build the dev container and access the project under that. Everything should "just work".
If you have an existing Deno project that you would like to add dev container support to, you will want to execute the command Remote-Containers: Add Development Container Configuration Files... in the command palette and then choose Show All Definitions... and then search for the Deno definition. This will setup a baseline .devcontainer configuration for you that will install the latest version of the Deno CLI in the container.
Once added, vscode will prompt if you want to open the project in a dev container. If you choose to, vscode will build the development container and re-open the workspace using the development container, which will have the Deno CLI and the vscode_deno extension installed in it.

14.0.11 Troubleshooting

The following sections cover challenges you might face when using the extension and try to give likely causes.

Errors/diagnostics like An import path cannot end with a '.ts' extension. or Cannot find name 'Deno'.

This is normally a situation where Deno is not enabled on a Deno project. If you look at the source of the diagnostic you are probably going to see a ts(2691). The ts indicates that it is coming from the built-in TypeScript/JavaScript engine in vscode. You will want to check that your configuration is set properly and the Deno: Enable/deno.enable is true.
You can also check what the Deno language server thinks is your current active configuration by using Deno: Language Server Status from the command palette. This will display a document from the language server with a section named Workspace Configuration. This will provide you with what vscode is reporting the configuration is to the language server.
If "enable" is set to true in there, and the error message still persists, you might want to try restarting vscode, as the part of the extension that "mutes" the built-in TypeScript diagnostics for files is not working as designed. If the issue still persists after a restart, you may have encountered a bug that we didn't expect and searching the issues and reporting a bug at https://github.com/denoland/vscode_deno is the next step.

14.1 Testing API

The vscode_deno extension implements a client for the vscode Testing API and when using a version of Deno that supports the testing API, tests in your project will be displayed within your IDE for Deno enabled projects.

14.1.1 Test display

When both the editor and the version of Deno support the testing API, the Test Explorer view will activate represented by a beaker icon, which will provide you with a side panel of tests that have been discovered in your project.
Also, next to tests identified in the code, there will be decorations which allow you to run and see the status of each test, as well as there will be entries in the command pallette for tests.

14.1.2 Discovering tests

Currently, Deno will only discover tests that are part of the "known" modules inside a workspace. A module becomes "known" when it is opened in the editor, or another module which imports that module is "known" inside the editor.
In the future, tests will be discovered in a similar fashion to the way the deno test subcommand discovers tests as part of the root of the workspace.

14.1.3 Running tests

You can run tests from the Test Explorer view, from the decorations next to the tests when viewing the test code, or via the command pallette. You can also use the filter function in the Text Explorer view to exclude certain tests from a test run.
Currently, Deno only supports the "run" test capability. We will be adding a debug run mode as well as a coverage run mode in the future. We will also be integrating the benchmarking tests as a tag, so they can be run (or excluded) from your test runs.
The Deno language server does not spin up a new CLI subprocess. It instead spawns a new thread and JavaScript runtime per test module to execute the tests.

14.1.4 Test output

Any console.log() that occurs in your tests will be sent to the test output window within vscode.
When a test fails, the failure message, including the stack trace, will be available when inspecting the test results in vscode.

14.1.5 How tests are structured

Test will be displayed in the Test Explorer at the top level with the module that contains the test. Inside the module will be all the tests that have been discovered, and if you are using test steps, they will be included under the test.
In most cases, the Deno language server will be able to statically identify tests, but if you are generating tests dynamically, Deno may not be aware of them until runtime. In these cases it may not be possible to filter these tests out of a run, but they will be added to the explorer view as they are encountered.

14.1.6 Configuration

By default, tests are executed in a similar fashion to if you were to use deno test --allow-all on the command line. These default arguments can be changed by setting the Deno > Testing: Args option in your user or workspace settings (or deno.testing.args if you are configuring manually). Add individual arguments here which you would have used with the deno test subcommand.
Based on other settings that you have, those options will be automatically merged into the "command line" used when running tests unless explicitly provided in the Deno > Testing: Args setting. For example if you have a Deno: Import Map (deno.importMap) set, the value of that will be used unless you have provided an explicit --import-map value in the testing args setting.

14.1.7 Known limitations and caveats

Because of the way the Deno test runner runs, it is not possible to exclude (or explicitly include) a test step. While the vscode UI will allow you to do this, by for example, choosing to run a specific test step, all test steps in that test will be run (but vscode will not update the results for them). So if there are other side effects in the test case, they may occur.

15 Language Server

The Deno CLI comes with a built in language server that can provide an intelligent editing experience as well as a way to easily access the other tools that come built in with Deno. For most users, using the language server would be via your editor like Visual Studio Code or other editors. This section of the manual is designed for those creating integrations to the language server or providing a package registry for Deno that integrates intelligently.
In this section we will cover:

- An Overview of the Language Server
- Import Sompletions and Intelligent Package Registries

15.1 Overview of the Language Server

The Deno Language Server provides a server implementation of the Language Server Protocol which is specifically tailored to provide a Deno view of code. It is integrated into the command line and can be started via the lsp sub-command.
Most users will never interact with the server directly, but instead will via vscode_deno or another editor extension. This documentation is for those implementing a editor client.

15.1.1 Structure

```
  ]
}
```

- "variables" - for the keys defined in the schema, a corresponding variable needs to be defined, which informs the language server where to fetch completions for that part of the module specifier. In the example above, we had 3 variables of package, version and path, so we would expect a variable definition for each.

15.2.4.3 Variables

In the configuration schema, the "variables" property is an array of variable definitions, which are objects with two required properties:

- "key" - a string which matches the variable key name specifier in the "schema" property.
- "documentation" - An optional URL where the language server can fetch the documentation for an individual variable entry. Variables can be substituted in to build the final URL. Variables with a single brace format like ${variable} will be added as matched out of the string, and those with double format like ${{variable}} will be percent-encoded as a URI component part.
- "url" - A URL where the language server can fetch the completions for the variable. Variables can be substituted in to build the URL. Variables with a single brace format like ${variable} will be added as matched out of the string, and those with double format like ${{variable}} will be percent-encoded as a URI component part. If the variable the value of the "key" is included, then the language server will support incremental requests for partial modules, allowing the server to provide completions as a user types part of the variable value. If the URL is not fully qualified, the URL of the schema file will be used as a base. In our example above, we had three variables and so our variable definition might look like:

```
{
  "version": 1,
  "registries": [
    {
      "schema": "/:package([a-z0-9_]*)@:version?/:path*",
      "variables": [
        {
          "key": "package",
          "documentation":
"https://api.example.com/docs/packages/${package}",
          "url": "https://api.example.com/packages/${package}"
        },
        {
          "key": "version",
          "url": "https://api.example.com/packages/${package}/versions"
        },
        {
          "key": "path",
          "documentation":
"https://api.example.com/docs/packages/${package}/${{version}}/paths/${path}",
          "url":
"https://api.example.com/packages/${package}/${{version}}/paths/${path}"
        }
      ]
```

If the client does not have the workspaceConfiguration capability, the language server will assume the workspace setting applies to all resources.

15.1.3 Commands

There are several commands that might be issued by the language server to the client, which the client is expected to implement:

- deno.cache - This is sent as a resolution code action when there is an un-cached module specifier that is being imported into a module. It will be sent with and argument that contains the resolved specifier as a string to be cached.
- deno.showReferences - This is sent as the command on some code lenses to show locations of references. The arguments contain the specifier that is the subject of the command, the start position of the target and the locations of the references to show.
- deno.test - This is sent as part of a test code lens to, of which the client is expected to run a test based on the arguments, which are the specifier the test is contained in and the name of the test to filter the tests on.

15.1.4 Requests

The LSP currently supports the following custom requests. A client should implement these in order to have a fully functioning client that integrates well with Deno:

- deno/cache - This command will instruct Deno to attempt to cache a module and all of its dependencies. If a referrer only is passed, then all dependencies for the module specifier will be loaded. If there are values in the uris, then only those uris will be cached.

It expects parameters of:

```
interface CacheParams {
  referrer: TextDocumentIdentifier;
  uris: TextDocumentIdentifier[];
}
```

- deno/performance - Requests the return of the timing averages for the internal instrumentation of Deno.

It does not expect any parameters.

- deno/reloadImportRegistries - Reloads any cached responses from import registries.

It does not expect any parameters.

- deno/virtualTextDocument - Requests a virtual text document from the LSP, which is a read only document that can be displayed in the client. This allows clients to access documents in the Deno cache, like remote modules and TypeScript library files built into Deno. The Deno language server will encode all internal files under the custom schema deno:, so clients should route all requests for the deno: schema back to the deno/virtualTextDocument API.

It also supports a special URL of deno:/status.md which provides a markdown formatted text document that contains details about the status of the LSP for display to a user.

It expects parameters of:

```
interface VirtualTextDocumentParams {
  textDocument: TextDocumentIdentifier;
}
```

15.1.5 Notifications

There is currently one custom notification that is sent from the server to the client:

- deno/registryState - when deno.suggest.imports.autoDiscover is true and an origin for an import being added to a document is not explicitly set in deno.suggest.imports.hosts, the origin will be checked and the notification will be sent to the client of the status.

When receiving the notification, if the param suggestion is true, the client should offer the user the choice to enable the origin and add it to the configuration for deno.suggest.imports.hosts.

If suggestion is false the client should add it to the configuration of as false to stop the language server from attempting to detect if suggestions are supported.
The params for the notification are:

```
interface RegistryStatusNotificationParams {
  origin: string;
  suggestions: boolean;
}
```

15.1.6 Language IDs

The language server supports diagnostics and formatting for the following text document language IDs:

- "javascript"
- "javascriptreact"
- "jsx" non standard, same as javascriptreact
- "typescript"
- "typescriptreact"
- "tsx" non standard, same as typescriptreact

The language server supports only formatting for the following language IDs:

- "json"
- "jsonc"
- "markdown"

15.2 Import Suggestions and Intelligent Registries

The language server, supports completions for URLs.

15.2.1 Local import completions

When attempting to import a relative module specifier (one that starts with ./ or ../), import completions are provided for directories and files that Deno thinks it can run (ending with the extensions .ts, .js, .tsx, .jsx, or .mjs).

15.2.2 Workspace import completions

When attempting to import a remote URL that isn't configured as a registry (see below), the extension will provide remote modules that are already part of the workspace.

15.2.3 Module registry completions

Module registries that support it can be configured for auto completion. This provides a handy way to explore a module registry from the "comfort" of your IDE.

15.2.3.1 Auto-discovery

The Deno language server, by default, will attempt to determine if a server supports completion suggestions. If the host/origin has not been explicitly configured, it will check the server, and if it supports completion suggestions you will be prompted to choose to enable it or not.
You should only enable this for registries you trust, as the remote server could provide suggestions for modules which are an attempt to get you to run un-trusted code.

15.2.3.2 Configuration

Settings for configuring registries for auto completions:

- deno.suggest.imports.autoDiscover - If enabled, when the language server discovers a new origin that isn't explicitly configured, it will check to see if that origin supports import completions and prompt you to enable it or not. This is true by default.
- deno.suggest.imports.hosts - These are the origins that are configured to provide import completions. The target host needs to support Deno import completions (detailed below). The value is an object where the key is the host and the value is if it is enabled or not. For example:

```
•     {
•       "deno.suggest.imports.hosts": {
```

```
•         "https://deno.land": true
•       }
}
```

15.2.3.3 How does it work?

On startup of the extension and language server, Deno will attempt to fetch /.well-known/deno-import-intellisense.json from any of the hosts that are configured and enabled. This file provides the data necessary to form auto completion of module specifiers in a highly configurable way (meaning you aren't tied into any particular module registry in order to get a rich editor experience).
As you build or edit your module specifier, Deno will go and fetch additional parts of the URL from the host based on what is configured in the JSON configuration file.
When you complete the module specifier, if it isn't already cached locally for you, Deno will attempt to fetch the completed specifier from the registry.

15.2.3.4 Does it work with all remote modules?

No, as the extension and Deno need to understand how to find modules. The configuration file provides a highly flexible way to allow people to describe how to build up a URL, including supporting things like semantic versioning if the module registry supports it.

15.2.4 Registry support for import completions

In order to support having a registry be discoverable by the Deno language server, the registry needs to provide a few things:

- A schema definition file. This file needs to be located at /.well-known/deno-import-intellisense.json. This file provides the configuration needed to allow the Deno language server query the registry and construct import specifiers.
- A series of API endpoints that provide the values to be provided to the user to complete an import specifier.

15.2.4.1 Configuration schema

The JSON response to the schema definition needs to be an object with two required properties:

- "version" - a number, which must be equal to 1 or 2.
- "registries" - an array of registry objects which define how the module specifiers are constructed for this registry.

There is a JSON Schema document which defines this schema available as part of the CLI's source code.
While the v2 supports more features than v1 did, they were introduced in a non-breaking way, and the language server automatically handles v1 or v2 versions, irrespective of what version is supplied in the "version" key, so technically a registry could profess itself to be v1 but use all the v2 features. This is not recommended though, because while there is no specific branches in code to support the v2 features currently, that doesn't mean there will not be in the future in order to support a v3 or whatever.

15.2.4.2 Registries

In the configuration schema, the "registries" property is an array of registries, which are objects which contain two required properties:

- "schema" - a string, which is an Express-like path matching expression, which defines how URLs are built on the registry. The syntax is directly based on path-to-regexp. For example, if the following was the specifier for a URL on the registry:

```
https://example.com/a_package@v1.0.0/mod.ts
```

The schema value might be something like this:

```
{
  "version": 1,
  "registries": [
    {
      "schema": "/:package([a-z0-9_]*)@:version?/:path*"
    }
  experimental?: {
    testingApi: boolean;
  };
}
```

When a version of Deno that supports the testing API encounters a client which supports the capability, it will initialize the code which handles the test detection and will start providing the notifications which enable it.
It should also be noted that when the testing API capabilities are enabled, the testing code lenses will no longer be sent to the client.

15.3.2 Settings

There are specific settings which change the behavior of the language server:

- deno.testing.args - An array of strings which will be provided as arguments when executing tests. This works in the same fashion as the deno test subcommand.
- deno.testing.enable - A binary flag that enables or disables the testing server

15.3.3 Notifications

The server will send notifications to the client under certain conditions.

15.3.3.1 deno/testModule

When a module containing tests is discovered by the server, it will notify the client by sending a deno/testModule notification along with a payload of TestModuleParams.
Deno structures in this fashion:

- A module can contain n tests.
- A test can contain n steps.
- A step can contain n steps.

When Deno does static analysis of a test module, it attempts to identify any tests and test steps. Because of the dynamic way tests can be declared in Deno, they cannot always be statically identified and can only be identified when the module is executed. The notification is designed to handle both of these situations when updating the client. When tests are discovered statically, the notification kind will be "replace", when tests or steps are discovered at execution time, the notification kind will be "insert".
As a test document is edited in the editor, and textDocument/didChange notifications are received from the client, the static analysis of those changes will be performed server side and if the tests have changed, the client will receive a notification.
When a client receives a "replace" notification, it can safely "replace" a test module representation, where when an "insert" it received, it should recursively try to add to existing representations.
For test modules the textDocument.uri should be used as the unique ID for any representation (as it the string URL to the unique module). TestData items contain a unique id string. This id string is a SHA-256 hash of identifying information that the server tracks for a test.

```
interface TestData {
  /** The unique ID for this test/step. */
  id: string;

  /** The display label for the test/step. */
  label: string;

  /** Any test steps that are associated with this test/step */
  steps?: TestData[];

  /** The range of the owning text document that applies to the test. */
  range?: Range;
```

```
•                    }
•                ]
}
```

15.2.4.3.1 URL endpoints

The response from each URL endpoint needs to be a JSON document that is an array of strings or a completions list:

```
interface CompletionList {
  /** The list (or partial list) of completion items. */
  items: string[];
  /** If the list is a partial list, and further queries to the endpoint will
   * change the items, set `isIncomplete` to `true`. */
  isIncomplete?: boolean;
  /** If one of the items in the list should be preselected (the default
   * suggestion), then set the value of `preselect` to the value of the item. */
  preselect?: string;
}
```

Extending our example from above the URL https://api.example.com/packages would be expected to return something like:

```
[
  "a_package",
  "another_package",
  "my_awesome_package"
]
```

Or something like this:

```
{
  "items": [
    "a_package",
    "another_package",
    "my_awesome_package"
  ],
  "isIncomplete": false,
  "preselect": "a_package"
}
```

And a query to https://api.example.com/packages/a_package/versions would return something like:

```
[
  "v1.0.0",
  "v1.0.1",
  "v1.1.0",
  "v2.0.0"
]
```

Or:

```
{
  "items": [
    "v1.0.0",
    "v1.0.1",
    "v1.1.0",
    "v2.0.0"
  ],
  "preselect": "v2.0.0"
}
```

And a query to https://api.example.com/packages/a_package/versions/v1.0.0/paths would return something like:

```
[
  "a.ts",
  "b/c.js",
  "d/e.ts"
]
```

Or:

```
{
  "items": [
    "a.ts",
    "b/c.js",
    "d/e.ts"
  ],
  "isIncomplete": true,
  "preselect": "a.ts"
}
```

15.2.4.3.2 Multi-part variables and folders

Navigating large file listings can be a challenge for the user. With the registry V2, the language server has some special handling of returned items to make it easier to complete a path to file in sub-folders easier.

When an item is returned that ends in /, the language server will present it to the client as a "folder" which will be represented in the client. So a registry wishing to provide sub-navigation to a folder structure like this:

```
examples/
└─┬─ first.ts
  └─ second.ts
sub-mod/
└─┬─ mod.ts
  └─ tests.ts
mod.ts
```

And had a schema like /:package([a-z0-9_]*)@:version?/:path* and an API endpoint for path like https://api.example.com/packages/${package}/${{version}}/${path} would want to respond to the path of /packages/pkg/1.0.0/ with:

```
{
  "items": [
    "examples/",
    "sub-mod/",
    "mod.ts"
  ],
  "isIncomplete": true
}
```

And to a path of /packages/pkg/1.0.0/examples/ with:

```
{
  "items": [
    "examples/first.ts",
    "examples/second.ts"
  ],
  "isIncomplete": true
}
```

This would allow the user to select the folder examples in the IDE before getting the listing of what was in the folder, making it easier to navigate the file structure.

15.2.4.3.3 Documentation endpoints

Documentation endpoints should return a documentation object with any documentation related to the requested entity:

```
interface Documentation {
  kind: "markdown" | "plaintext";
  value: string;
}
```

For extending the example from above, a query to https://api.example.com/packages/a_package would return something like:

```
{
  "kind": "markdown",
  "value": "some _markdown_ `documentation` here..."
}
```

15.2.4.4 Schema validation

When the language server is started up (or the configuration for the extension is changed) the language server will attempt to fetch and validate the schema configuration for the domain hosts specifier in the configuration.

The validation attempts to make sure that all registries defined are valid, that the variables contained in those schemas are specified in the variables, and that there aren't extra variables defined that are not included in the schema. If the validation fails, the registry won't be enabled and the errors will be logged to the Deno Language Server output in vscode.

If you are a registry maintainer and need help, advice, or assistance in setting up your registry for auto-completions, feel free to open up an issue and we will try to help.

15.2.5 Known registries

The following is a list of registries known to support the scheme. All you need to do is add the domain to deno.suggest.imports.hosts and set the value to true:

- https://deno.land/ - both the 3rd party /x/ registry and the /std/ library registry are available.
- https://nest.land/ - a module registry for Deno on the blockweave.
- https://crux.land/ - a free open-source registry for permanently hosting small scripts.

15.3 Testing API

The Deno language server supports a custom set of APIs to enable testing. These are built around providing information to enable the vscode's Testing API but can be used by other language server clients to provide a similar interface.

15.3.1 Capabilities

Both the client and the server should support the experimental testingApi capability:

```
interface ClientCapabilities {
  experimental?: {
    testingApi: boolean;
  };
}
interface ServerCapabilities {
  id: number;

  /** The message*/
  message: TestRunProgressMessage;
}
```

15.3.4 Requests

The server handles two different requests:

15.3.4.1 deno/testRun

To request the language server to perform a set of tests, the client sends a deno/testRun request, which includes that ID of the test run to be used in future responses to the client, the type of the test run, and any test modules or tests to include or exclude.

Currently Deno only supports the "run" kind of test run. Both "debug" and "coverage" are planned to be supported in the future.

When there are no test modules or tests that are included, it implies that all discovered tests modules and tests should be executed. When a test module is included, but not any test ids, it implies that all tests within that test module should be included. Once all the tests are identified, any excluded tests are removed and the resolved set of tests are returned in the response as "enqueued".

It is not possible to include or exclude test steps via this API, because of the dynamic nature of how test steps are declared and run.

```
interface TestRunRequestParams {
  /** The id of the test run to be used for future messages. */
  id: number;

  /** The run kind. Currently Deno only supports `"run"` */
  kind: "run" | "coverage" | "debug";

  /** Test modules or tests to exclude from the test run. */
  exclude?: TestIdentifier[];

  /** Test modules or tests to include in the test run. */
  include?: TestIdentifier[];
}

interface EnqueuedTestModule {
  /** The test module the enqueued test IDs relate to */
  textDocument: TextDocumentIdentifier;

  /** The test IDs which are now enqueued for testing */
  ids: string[];
}

interface TestRunResponseParams {
  /** Test modules and test IDs that are now enqueued for testing. */
  enqueued: EnqueuedTestModule[];
}
```

15.3.5 deno/testRunCancel

```
}

interface TestModuleParams {
  /** The text document identifier that the tests are related to. */
  textDocument: TextDocumentIdentifier;

  /** A indication if tests described are _newly_ discovered and should be
   * _inserted_ or if the tests associated are a replacement for any existing
   * tests. */
  kind: "insert" | "replace";

  /** The text label for the test module. */
  label: string;

  /** An array of tests that are owned by this test module. */
  tests: TestData[];
}
```

15.3.3.2 deno/testModuleDelete

When a test module is deleted that the server is observing, the server will issue a deno/testModuleDelete notification. When receiving the notification the client should remove the representation of the test module and all of its children tests and test steps.

```
interface TestModuleDeleteParams {
  /** The text document identifier that has been removed. */
  textDocument: TextDocumentIdentifier;
}
```

15.3.3.3 deno/testRunProgress

When a deno/testRun is requested from the client, the server will support progress of that test run via the deno/testRunProgress notification.

The client should process these messages and update any UI representation.

The state change is represented in the .message.kind property of the TestRunProgressParams. The states are:

- "enqueued" - A test or test step has been enqueued for testing.
- "skipped" - A test or test step was skipped. This occurs when the Deno test has the ignore option set to true.
- "started" - A test or test step has started.
- "passed" - A test or test step has passed.
- "failed" - A test or test step has failed. This is intended to indicate an error with the test harness instead of the test itself, but Deno currently does not support this distinction.
- "errored" - The test or test step has errored. Additional information about the error will be in the .message.messages property.
- "end" - The test run has ended.

```
interface TestIdentifier {
  /** The test module the message is related to. */
  textDocument: TextDocumentIdentifier;

  /** The optional ID of the test. If not present, then the message applies to
   * all tests in the test module. */
  id?: string;

  /** The optional ID of the step. If not present, then the message only applies
   * to the test. */
  stepId?: string;
}

interface TestMessage {
  /** The content of the message. */
  message: MarkupContent;

  /** An optional string which represents the expected output. */
  expectedOutput?: string;

  /** An optional string which represents the actual output. */
  actualOutput?: string;

  /** An optional location related to the message. */
  location?: Location;
}

interface TestEnqueuedStartedSkipped {
  /** The state change that has occurred to a specific test or test step.
   *
   * - `"enqueued"` - the test is now enqueued to be tested
   * - `"started"` - the test has started
   * - `"skipped"` - the test was skipped
   */
  type: "enqueued" | "started" | "skipped";

  /** The test or test step relating to the state change. */
  test: TestIdentifier;
}

interface TestFailedErrored {
  /** The state change that has occurred to a specific test or test step.
   *
   * - `"failed"` - The test failed to run properly, versus the test erroring.
   *     currently the Deno language server does not support this.
   * - `"errored"` - The test errored.
   */
  type: "failed" | "errored";

  /** The test or test step relating to the state change. */
  test: TestIdentifier;

  /** Messages related to the state change. */
```

```
  messages: TestMessage[];

  /** An optional duration, in milliseconds from the start to the current
   * state. */
  duration?: number;
}

interface TestPassed {
  /** The state change that has occurred to a specific test or test step. */
  type: "passed";

  /** The test or test step relating to the state change. */
  test: TestIdentifier;

  /** An optional duration, in milliseconds from the start to the current
   * state. */
  duration?: number;
}

interface TestOutput {
  /** The test or test step has output information / logged information. */
  type: "output";

  /** The value of the output. */
  value: string;

  /** The associated test or test step if there was one. */
  test?: TestIdentifier;

  /** An optional location associated with the output. */
  location?: Location;
}

interface TestEnd {
  /** The test run has ended. */
  type: "end";
}

type TestRunProgressMessage =
  | TestEnqueuedStartedSkipped
  | TestFailedErrored
  | TestPassed
  | TestOutput
  | TestEnd;

interface TestRunProgressParams {
  /** The test run ID that the progress message applies to. */
```

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).

19.1.1 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:
4. git config --global core.symlinks true

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

19.1.2 Prerequisites

Deno requires the progressively latest stable release of Rust. Deno does not support the Rust Nightly Releases.

Update or Install Rust. Check that Rust installed/updated correctly:

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

For Apple aarch64 users lld must be installed.

```
brew install llvm
# Add /opt/homebrew/opt/llvm/bin/ to $PATH
```

19.1.3 Building Deno

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 3 for running WPT tests. Ensure that a suffix-less python/python.exe exists in your PATH and it refers to Python 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 Command Line Tools installed. (XCode already includes CLT. Run xcode-select --install to install it without XCode.)

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.)

If a client wishes to cancel a currently running test run, it sends a deno/testRunCancel request with the test ID to cancel. The response back will be a boolean of true if the test is cancelled or false if it was not possible. Appropriate test progress notifications will still be sent as the test is being cancelled.

```
interface TestRunCancelParams {
  /** The test id to be cancelled. */
  id: number;
}
```

16 Publishing Modules

Deno is not prescriptive about how developers make their modules available—modules may be imported from any source. To help publish and distribute modules, separate standalone solutions are provided.

16.1 Publishing on deno.land/x

A common way to publish Deno modules is via the official https://deno.land/x hosting service. It caches releases of open source modules and serves them at one easy to remember domain.

To use it, modules must be developed and hosted in public repositories on GitHub. Their source is then published to deno.land/x on tag creation. They can then be accessed by using a url in the following format:

```
https://deno.land/x/<module_name>@<tag_name>/<file_path>
```

Module versions are persistent and immutable. It is thus not possible to edit or delete a module (or version), to prevent breaking programs that rely on this module. Modules may be removed if there is a legal reason to do so (for example copyright infringement).

For more details, see Adding a Module.

16.2 Publishing Deno modules for Node.js

See dnt - Deno to Node Transform.

17 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.

18 Help

Stuck? Lost? Get Help from the Community.

18.1 Stack Overflow

Stack Overflow is a popular forum to ask code-level questions or if you’re stuck with a specific error. ask your own!

18.2 Community Discord

Ask questions and chat with community members in real-time.

18.3 DEV's Deno Community

A great place to find interesting articles about best practices, application architecture and new learnings. Post your articles with the tag deno.

19 Contributing

We welcome and appreciate all contributions to Deno.
This page serves as a helper to get you started on contributing.

19.0 Projects

There are numerous repositories in the denoland organization that are part of the Deno ecosystem. Repositories have different scopes, use different programming languages and have varying difficulty level when it comes to contributions.
To help you decide which repository might be the best to start contributing (and/or falls into your interest), here's a short comparison (**languages in bold comprise most of the codebase**):

19.0.1 deno

This is the main repository that provides the `deno` CLI.
If you want to fix a bug or add a new feature to `deno` this is the repository you want to contribute to.
Languages: **Rust**, **JavaScript**

19.0.2 deno_std

The standard library for Deno.
Languages: **TypeScript**, WebAssembly.

19.0.3 dotland

Frontend for official Deno webpage: https://deno.land/
Languages: **TypeScript**, TSX, CSS

19.0.4 deno_lint

Linter that powers `deno lint` subcommand.
Languages: **Rust**

19.0.5 deno_doc

Documentation generator that powers `deno doc` subcommand and https://doc.deno.land
Languages: **Rust**

19.0.6 docland

Frontend for documentation generator: https://doc.deno.land
Languages: **TypeScript**, TSX, CSS

19.0.7 rusty_v8

Rust bindings for the V8 JavaScript engine. Very technical and low-level.
Languages: **Rust**

19.0.8 serde_v8

Library that provides bijection layer between V8 and Rust objects. Based on serde library. Very technical and low-level.
Languages: **Rust**

19.0.9 deno_docker

Official Docker images for Deno.

19.0.10 General remarks

- 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.
- If you are going to work on a new feature, create an issue and discuss with other contributors before you start working on the feature; we appreciate all contributions, but not all proposed features are getting accepted. We don't want you to spend hours working on a code that might not be accepted.
- Please be professional in the forums. We follow Rust's code of conduct (CoC). Have a problem? Email ry@tinyclouds.org.

19.0.11 Submitting a pull request

Before submitting a PR to any of the repos, please make sure the following is done:

1. Give the PR a descriptive title.

Examples of good PR title:

- fix(std/http): Fix race condition in server
- docs(console): Update docstrings
- feat(doc): Handle nested re-exports

Examples of bad PR title:

- fix #7123
- update docs
- fix bugs

1. Ensure there is a related issue and it is referenced in the PR text.
2. Ensure there are tests that cover the changes.

19.0.11.1 Submitting a PR to deno

Additionally to the above make sure that:

1. `cargo test` passes - this will run full test suite for `deno` including unit tests, integration tests and Web Platform Tests
2. Run `./tools/format.js` - this will format all of the code to adhere to the consistent style in the repository
3. Run `./tools/lint.js` - this will check Rust and JavaScript code for common mistakes and errors using `clippy` (for Rust) and `dlint` (for JavaScript)

19.0.12 Submitting a PR to deno_std

Additionally to the above make sure that:

1. All of the code you wrote is in `TypeScript` (ie. don't use `JavaScript`)
2. `deno test --unstable --allow-all` passes - this will run full test suite for the standard library
3. Run `deno fmt` in the root of repository - this will format all of the code to adhere to the consistent style in the repository.
4. Run `deno lint` - this will check TypeScript code for common mistakes and errors.

19.0.13 Documenting APIs

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

19.0.13.1 JavaScript and TypeScript

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/

19.0.13.2 Rust

Use this guide for writing documentation comments in Rust code.

19.1 Building from Source

```
  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,
```

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

19.1.4 Building

Build with Cargo:

```
# Build:
cargo build -vv

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

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

19.2 Web Platform Tests

Deno uses a custom test runner for Web Platform Tests. It can be found at ./tools/wpt.ts.

19.2.1 Running tests

If you are on Windows, or your system does not support shebangs, prefix all ./tools/wpt.ts commands with `deno run --unstable --allow-write --allow-read --allow-net --allow-env --allow-run`.

Before attempting to run WPT tests for the first time, please run the WPT setup. You must also run this command every time the ./test_util/wpt submodule is updated:

```
./tools/wpt.ts setup
```

To run all available web platform tests, run the following command:

```
./tools/wpt.ts run

# You can also filter which test files to run by specifying filters:
./tools/wpt.ts run -- streams/piping/general hr-time
```

The test runner will run each web platform test and record its status (failed or ok). It will then compare this output to the expected output of each test as specified in the ./tools/wpt/expectation.json file. This file is a nested JSON structure that mirrors the ./test_utils/wpt directory. It describes for each test file, if it should pass as a whole (all tests pass, `true`), if it should fail as a whole (test runner encounters an exception outside of a test or all tests fail, `false`), or which tests it expects to fail (a string array of test case names).

19.2.2 Updating enabled tests or expectations

You can update the ./tools/wpt/expectation.json file manually by changing the value of each of the test file entries in the JSON structure. The alternative and preferred option is to have the WPT runner run all, or a filtered subset of tests, and then automatically update the `expectation.json` file to match the current reality. You can do this with the `./wpt.ts update` command. Example:

```
./tools/wpt.ts update -- hr-time
```

After running this command the `expectation.json` file will match the current output of all the tests that were run. This means that running `wpt.ts run` right after a `wpt.ts update` should always pass.

19.2.3 Subcommands

19.2.3.1 setup

Validate that your environment is configured correctly, or help you configure it.

This will check that the python3 (or `python.exe` on Windows) is actually Python 3.

You can specify the following flags to customize behaviour:

--rebuild
  Rebuild the manifest instead of downloading. This can take up to 3 minutes.

--auto-config
    Automatically configure /etc/hosts if it is not configured (no prompt will be shown).

19.2.3.2 run

Run all tests like specified in expectation.json.
You can specify the following flags to customize behaviour:

--release
    Use the ./target/release/deno binary instead of ./target/debug/deno

--quiet
    Disable printing of `ok` test cases.

--json=<file>
    Output the test results as JSON to the file specified.

You can also specify exactly which tests to run by specifying one of more filters after a --:
./tools/wpt.ts run -- hr-time streams/piping/general

19.2.3.3 update

Update the expectation.json to match the current reality.
You can specify the following flags to customize behaviour:

--release
    Use the ./target/release/deno binary instead of ./target/debug/deno

--quiet
    Disable printing of `ok` test cases.

--json=<file>
    Output the test results as JSON to the file specified.

You can also specify exactly which tests to run by specifying one of more filters after a --:
./tools/wpt.ts update -- hr-time streams/piping/general

19.2.4 FAQ

19.2.4.1 Upgrading the wpt submodule:

```
cd test_util/wpt/
# Rebase to retain our modifications
git rebase origin/master
git push denoland
```

All contributors will need to rerun ./tools/wpt.ts setup after this.

19.3 Style Guide

⚠ Note that this is the style guide for **internal runtime code** in the Deno runtime, and in the Deno standard library. This is not meant as a general style guide for users of Deno.

19.3.1 Copyright Headers

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

```
// Copyright 2018-2022 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.

19.3.2 Use underscores, not dashes in filenames.

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

19.3.3 Add tests for new features.

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

19.3.4 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.
```

19.3.5 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.

19.3.6 Inclusive code

Please follow the guidelines for inclusive code outlined at https://chromium.googlesource.com/chromium/src/+/master/styleguide/inclusive_code.md.

19.3.7 Rust

Follow Rust conventions and be consistent with existing code.

19.3.8 TypeScript

The TypeScript portion of the code base is the standard library std.

19.3.8.1 Use TypeScript instead of JavaScript.

19.3.8.2 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.

19.3.8.3 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.

19.3.8.4 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",
function convertURL(url: URL) {
  return url.href;
}
```

19.4 Architecture

19.4.1 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

19.4.1.1 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.

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

19.4.1.2 Metrics

Metrics is Deno's internal counter for various statistics.

```
> console.table(Deno.metrics())
┌─────────────────────────┬────────┐
│         (idx)           │ Values │
├─────────────────────────┼────────┤
│      opsDispatched      │   9    │
│    opsDispatchedSync    │   0    │
│   opsDispatchedAsync    │   0    │
│ opsDispatchedAsyncUnref │   0    │
│      opsCompleted       │   9    │
│    opsCompletedSync     │   0    │
│    opsCompletedAsync    │   0    │
│ opsCompletedAsyncUnref  │   0    │
│    bytesSentControl     │  504   │
│      bytesSentData      │   0    │
│      bytesReceived      │  856   │
└─────────────────────────┴────────┘
```

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

19.3.8.5 Export all interfaces that are used as parameters to an exported member

Whenever you are using interfaces that are included in the parameters or return type of an exported member, you should export the interface that is used. Here is an example:

```
// my_file.ts
export interface Person {
  name: string;
  age: number;
}

export function createPerson(name: string, age: number): Person {
  return { name, age };
}

// mod.ts
export { createPerson } from "./my_file.ts";
export type { Person } from "./my_file.ts";
```

19.3.8.6 Minimize dependencies; do not make circular imports.

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

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

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.

19.3.8.8 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 JSDoc. 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 utilize markdown format, like so:

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

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

19.3.8.9 Resolve linting problems using directives

Currently, the building process uses dlint to validate linting problems in the code. If the task requires code that is non-conformant to linter use deno-lint-ignore <code> directive to suppress the warning.

```
// deno-lint-ignore no-explicit-any
let x: any;
```

This ensures the continuous integration process doesn't fail due to linting problems, but it should be used scarcely.

19.3.8.10 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 std module should go in std/tests due to their different contexts; otherwise, it should just be a sibling to the tested module.

19.3.8.11 Unit Tests should be explicit.

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

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

Example of test:

```
import { assertEquals } from "https://deno.land/std@0.156.0/testing/asserts.ts";
import { foo } from "./mod.ts";

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

19.3.8.12 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";
}
```

19.3.8.13 std

19.3.8.13.1 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.

19.3.8.13.2 Document and maintain browser compatibility.

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.

19.3.8.13.3 Prefer # over private

We prefer the private fields (#) syntax over private keyword of TypeScript in the standard modules codebase. The private fields make the properties and methods private even at runtime. On the other hand, private keyword of TypeScript guarantee it private only at compile time and the fields are publicly accessible at runtime.

Good:

```
class MyClass {
  #foo = 1;
  #bar() {}
}
```

Bad:

```
class MyClass {
  private foo = 1;
  private bar() {}
}
```

19.3.8.13.4 Naming convention

Always use camel or pascal case. Some Web APIs use uppercase acronyms (JSON, URL, URL.createObjectURL() etc.). Deno does not follow this convention and also uses camel or pascal case.

Good:

```
class HttpObject {
}
```

Bad:

```
class HTTPObject {
}
```

Good:

```
function convertUrl(url: URL) {
  return url.href;
}
```

Bad:

```
  --future (Implies all staged features that we want to ship in the not-too-far future)
        type: bool  default: false
  --assert-types (generate runtime type assertions to test the typer)
        type: bool  default: false
  --allocation-site-pretenuring (pretenure with allocation sites)
        type: bool  default: true
  --page-promotion (promote pages based on utilization)
        type: bool  default: true
  --always-promote-young-mc (always promote young objects during mark-compact)
        type: bool  default: true
  --page-promotion-threshold (min percentage of live bytes on a page to enable fast
evacuation)
        type: int  default: 70
  --trace-pretenuring (trace pretenuring decisions of HAllocate instructions)
        type: bool  default: false
  --trace-pretenuring-statistics (trace allocation site pretenuring statistics)
        type: bool  default: false
  --track-fields (track fields with only smi values)
        type: bool  default: true
  --track-double-fields (track fields with double values)
        type: bool  default: true
  --track-heap-object-fields (track fields with heap values)
        type: bool  default: true
  --track-computed-fields (track computed boilerplate fields)
        type: bool  default: true
  --track-field-types (track field types)
        type: bool  default: true
  --trace-block-coverage (trace collected block coverage information)
        type: bool  default: false
  --trace-protector-invalidation (trace protector cell invalidations)
        type: bool  default: false
  --feedback-normalization (feed back normalization to constructors)
        type: bool  default: false
  --enable-one-shot-optimization (Enable size optimizations for the code that will only be
executed once)
        type: bool  default: false
  --unbox-double-arrays (automatically unbox arrays of doubles)
        type: bool  default: true
  --interrupt-budget (interrupt budget which should be used for the profiler counter)
        type: int  default: 147456
  --jitless (Disable runtime allocation of executable memory.)
        type: bool  default: false
  --use-ic (use inline caching)
        type: bool  default: true
  --budget-for-feedback-vector-allocation (The budget in amount of bytecode executed by a
function before we decide to allocate feedback vectors)
        type: int  default: 1024
```

19.4.2 Schematic diagram

Anqi:the original document lost the picture.

19.4.3 Conference

- Ryan Dahl. (May 27, 2020). An interesting case with Deno. Deno Israel.
- Bartek Iwańczuk. (Oct 6, 2020). Deno internals - how modern JS/TS runtime is built. Paris Deno.

19.5 Profiling

19.5.1 Perf profiling:

Tools that can be used to generate/ visualise perf results:

- flamegraph-rs (https://github.com/flamegraph-rs/flamegraph)
- flamescope (https://github.com/Netflix/flamescope)

Example using perf on micro_bench_ops and visualising using flamescope:

```
# build `examples/micro_bench_ops`
cargo build --release --example micro_bench_ops

# run `examples/micro_bench_ops` using perf
sudo perf record -F 49 -a -g -- ./target/release/examples/micro_bench_ops
sudo perf script --header > micro_bench_ops_perf

# now open the file using flamescope
```

Example running deno_tcp.ts in combination with flamegraph (script.sh):

```
sudo flamegraph -o flamegraph.svg target/debug/deno run --allow-net cli/bench/deno_tcp.ts &
sleep 1
./third_party/prebuilt/linux64/wrk http://localhost:4500/
sleep 1
kill `pgrep perf`
```

19.5.2 v8 profiling:

Example using v8 profiling on micro_bench_ops:

```
# build `examples/micro_bench_ops`
cargo build --release --example micro_bench_ops

# run `examples/micro_bench_ops`
./target/release/examples/micro_bench_ops --prof
```

Example using v8 profiling on deno_tcp.ts:

```
# build `deno`
cargo build --release

# run `deno_tcp.ts`
./target/release/deno --v8-flags=--prof --allow-net cli/bench/deno_tcp.ts &
sleep 1
./third_party/prebuilt/linux64/wrk http://localhost:4500/
sleep 1
kill `pgrep deno`
```

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

```
node --prof-process isolate-0x7fad98242400-v8.log > prof.log
```

prof.log will contain information about tick distribution of different calls.
To view the log with Web UI, generate JSON file of the log:
Open rusty_v8/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 profiling, check out the following links:

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

19.5.3 Debugging with LLDB

To debug the deno binary, 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
```

19.5.4 V8 flags

V8 has many many internal command-line flags:

```
$ deno run --v8-flags=--help _
SSE3=1 SSSE3=1 SSE4_1=1 SSE4_2=1 SAHF=1 AVX=1 FMA3=1 BMI1=1 BMI2=1 LZCNT=1 POPCNT=1 ATOM=0
Synopsis:
  shell [options] [--shell] [<file>...]
  d8 [options] [-e <string>] [--shell] [[--module] <file>...]

  -e        execute a string in V8
  --shell   run an interactive JavaScript shell
  --module  execute a file as a JavaScript module

Note: the --module option is implicitly enabled for *.mjs files.

The following syntax for options is accepted (both '-' and '--' are ok):
```

```
  --flag         (bool flags only)
  --no-flag      (bool flags only)
  --flag=value   (non-bool flags only, no spaces around '=')
  --flag value   (non-bool flags only)
  --             (captures all remaining args in JavaScript)

Options:
  --use-strict (enforce strict mode)
        type: bool  default: false
  --es-staging (enable test-worthy harmony features (for internal use only))
        type: bool  default: false
  --harmony (enable all completed harmony features)
        type: bool  default: false
  --harmony-shipping (enable all shipped harmony features)
        type: bool  default: true
  --harmony-regexp-sequence (enable "RegExp Unicode sequence properties" (in progress))
        type: bool  default: false
  --harmony-weak-refs-with-cleanup-some (enable "harmony weak references with FinalizationRegistry.prototype.cleanupSome" (in progress))
        type: bool  default: false
  --harmony-regexp-match-indices (enable "harmony regexp match indices" (in progress))
        type: bool  default: false
  --harmony-top-level-await (enable "harmony top level await")
        type: bool  default: false
  --harmony-namespace-exports (enable "harmony namespace exports (export * as foo from 'bar')")
        type: bool  default: true
  --harmony-sharedarraybuffer (enable "harmony sharedarraybuffer")
        type: bool  default: true
  --harmony-import-meta (enable "harmony import.meta property")
        type: bool  default: true
  --harmony-dynamic-import (enable "harmony dynamic import")
        type: bool  default: true
  --harmony-promise-all-settled (enable "harmony Promise.allSettled")
        type: bool  default: true
  --harmony-promise-any (enable "harmony Promise.any")
        type: bool  default: true
  --harmony-private-methods (enable "harmony private methods in class literals")
        type: bool  default: true
  --harmony-weak-refs (enable "harmony weak references")
        type: bool  default: true
  --harmony-string-replaceall (enable "harmony String.prototype.replaceAll")
        type: bool  default: true
  --harmony-logical-assignment (enable "harmony logical assignment")
        type: bool  default: true
  --lite-mode (enables trade-off of performance for memory savings)
        type: bool  default: false
        type: int  default: 30
  --max-optimized-bytecode-size (maximum bytecode size to be considered for optimization;
too high values may cause the compiler to hit (release) assertions)
        type: int  default: 61440
  --min-inlining-frequency (minimum frequency for inlining)
        type: float  default: 0.15
  --polymorphic-inlining (polymorphic inlining)
        type: bool  default: true
  --stress-inline (set high thresholds for inlining to inline as much as possible)
        type: bool  default: false
  --trace-turbo-inlining (trace TurboFan inlining)
        type: bool  default: false
  --turbo-inline-array-builtins (inline array builtins in TurboFan code)
        type: bool  default: true
  --use-osr (use on-stack replacement)
        type: bool  default: true
  --trace-osr (trace on-stack replacement)
        type: bool  default: false
  --analyze-environment-liveness (analyze liveness of environment slots and zap dead values)
        type: bool  default: true
  --trace-environment-liveness (trace liveness of local variable slots)
        type: bool  default: false
  --turbo-load-elimination (enable load elimination in TurboFan)
        type: bool  default: true
  --trace-turbo-load-elimination (trace TurboFan load elimination)
        type: bool  default: false
  --turbo-profiling (enable basic block profiling in TurboFan)
        type: bool  default: false
  --turbo-profiling-verbose (enable basic block profiling in TurboFan, and include each
function's schedule and disassembly in the output)
        type: bool  default: false
  --turbo-verify-allocation (verify register allocation in TurboFan)
        type: bool  default: false
  --turbo-move-optimization (optimize gap moves in TurboFan)
        type: bool  default: true
  --turbo-jt (enable jump threading in TurboFan)
        type: bool  default: true
  --turbo-loop-peeling (Turbofan loop peeling)
        type: bool  default: true
  --turbo-loop-variable (Turbofan loop variable optimization)
        type: bool  default: true
  --turbo-loop-rotation (Turbofan loop rotation)
        type: bool  default: true
  --turbo-cf-optimization (optimize control flow in TurboFan)
        type: bool  default: true
  --turbo-escape (enable escape analysis)
        type: bool  default: true
```

```
  --lazy-feedback-allocation (Allocate feedback vectors lazily)
        type: bool  default: true
  --ignition-elide-noneffectful-bytecodes (elide bytecodes which won't have any external
effect)
        type: bool  default: true
  --ignition-reo (use ignition register equivalence optimizer)
        type: bool  default: true
  --ignition-filter-expression-positions (filter expression positions before the bytecode
pipeline)
        type: bool  default: true
  --ignition-share-named-property-feedback (share feedback slots when loading the same
named property from the same object)
        type: bool  default: true
  --print-bytecode (print bytecode generated by ignition interpreter)
        type: bool  default: false
  --enable-lazy-source-positions (skip generating source positions during initial compile but
regenerate when actually required)
        type: bool  default: true
  --stress-lazy-source-positions (collect lazy source positions immediately after lazy compile)
        type: bool  default: false
  --print-bytecode-filter (filter for selecting which functions to print bytecode)
        type: string  default: *
  --trace-ignition-codegen (trace the codegen of ignition interpreter bytecode handlers)
        type: bool  default: false
  --trace-ignition-dispatches (traces the dispatches to bytecode handlers by the ignition
interpreter)
        type: bool  default: false
  --trace-ignition-dispatches-output-file (the file to which the bytecode handler dispatch table
is written (by default, the table is not written to a file))
        type: string  default: nullptr
  --fast-math (faster (but maybe less accurate) math functions)
        type: bool  default: true
  --trace-track-allocation-sites (trace the tracking of allocation sites)
        type: bool  default: false
  --trace-migration (trace object migration)
        type: bool  default: false
  --trace-generalization (trace map generalization)
        type: bool  default: false
  --turboprop (enable experimental turboprop mid-tier compiler.)
        type: bool  default: false
  --concurrent-recompilation (optimizing hot functions asynchronously on a separate thread)
        type: bool  default: true
  --trace-concurrent-recompilation (track concurrent recompilation)
        type: bool  default: false
  --concurrent-recompilation-queue-length (the length of the concurrent compilation queue)
        type: int  default: 8
  --concurrent-recompilation-delay (artificial compilation delay in ms)
```

```
            type: int    default: 0
    --block-concurrent-recompilation (block queued jobs until released)
            type: bool    default: false
    --concurrent-inlining (run optimizing compiler's inlining phase on a separate thread)
            type: bool    default: false
    --max-serializer-nesting (maximum levels for nesting child serializers)
            type: int    default: 25
    --trace-heap-broker-verbose (trace the heap broker verbosely (all reports))
            type: bool    default: false
    --trace-heap-broker-memory (trace the heap broker memory (refs analysis and zone
numbers))
            type: bool    default: false
    --trace-heap-broker (trace the heap broker (reports on missing data only))
            type: bool    default: false
    --stress-runs (number of stress runs)
            type: int    default: 0
    --deopt-every-n-times (deoptimize every n times a deopt point is passed)
            type: int    default: 0
    --print-deopt-stress (print number of possible deopt points)
            type: bool    default: false
    --opt (use adaptive optimizations)
            type: bool    default: true
    --turbo-sp-frame-access (use stack pointer-relative access to frame wherever possible)
            type: bool    default: false
    --turbo-control-flow-aware-allocation (consider control flow while allocating registers)
            type: bool    default: true
    --turbo-filter (optimization filter for TurboFan compiler)
            type: string    default: *
    --trace-turbo (trace generated TurboFan IR)
            type: bool    default: false
    --trace-turbo-path (directory to dump generated TurboFan IR to)
            type: string    default: nullptr
    --trace-turbo-filter (filter for tracing turbofan compilation)
            type: string    default: *
    --trace-turbo-graph (trace generated TurboFan graphs)
            type: bool    default: false
    --trace-turbo-scheduled (trace TurboFan IR with schedule)
            type: bool    default: false
    --trace-turbo-cfg-file (trace turbo cfg graph (for C1 visualizer) to a given file name)
            type: string    default: nullptr
    --trace-turbo-types (trace TurboFan's types)
            type: bool    default: true
    --trace-turbo-scheduler (trace TurboFan's scheduler)
            type: bool    default: false
    --trace-turbo-reduction (trace TurboFan's various reducers)
            type: bool    default: false
    --trace-turbo-trimming (trace TurboFan's graph trimmer)
```

```
            type: bool    default: false
    --trace-turbo-jt (trace TurboFan's jump threading)
            type: bool    default: false
    --trace-turbo-ceq (trace TurboFan's control equivalence)
            type: bool    default: false
    --trace-turbo-loop (trace TurboFan's loop optimizations)
            type: bool    default: false
    --trace-turbo-alloc (trace TurboFan's register allocator)
            type: bool    default: false
    --trace-all-uses (trace all use positions)
            type: bool    default: false
    --trace-representation (trace representation types)
            type: bool    default: false
    --turbo-verify (verify TurboFan graphs at each phase)
            type: bool    default: false
    --turbo-verify-machine-graph (verify TurboFan machine graph before instruction selection)
            type: string    default: nullptr
    --trace-verify-csa (trace code stubs verification)
            type: bool    default: false
    --csa-trap-on-node (trigger break point when a node with given id is created in given stub.
The format is: StubName,NodeId)
            type: string    default: nullptr
    --turbo-stats (print TurboFan statistics)
            type: bool    default: false
    --turbo-stats-nvp (print TurboFan statistics in machine-readable format)
            type: bool    default: false
    --turbo-stats-wasm (print TurboFan statistics of wasm compilations)
            type: bool    default: false
    --turbo-splitting (split nodes during scheduling in TurboFan)
            type: bool    default: true
    --function-context-specialization (enable function context specialization in TurboFan)
            type: bool    default: false
    --turbo-inlining (enable inlining in TurboFan)
            type: bool    default: true
    --max-inlined-bytecode-size (maximum size of bytecode for a single inlining)
            type: int    default: 500
    --max-inlined-bytecode-size-cumulative (maximum cumulative size of bytecode considered
for inlining)
            type: int    default: 1000
    --max-inlined-bytecode-size-absolute (maximum cumulative size of bytecode considered for
inlining)
            type: int    default: 5000
    --reserve-inline-budget-scale-factor (maximum cumulative size of bytecode considered for
inlining)
            type: float    default: 1.2
    --max-inlined-bytecode-size-small (maximum size of bytecode considered for small
function inlining)
```

```
    --wasm-atomics-on-non-shared-memory (allow atomic operations on non-shared
WebAssembly memory)
              type: bool    default: true
    --wasm-grow-shared-memory (allow growing shared WebAssembly memory objects)
              type: bool    default: true
    --wasm-simd-post-mvp (allow experimental SIMD operations for prototyping that are not
included in the current proposal)
              type: bool    default: false
    --wasm-code-gc (enable garbage collection of wasm code)
              type: bool    default: true
    --trace-wasm-code-gc (trace garbage collection of wasm code)
              type: bool    default: false
    --stress-wasm-code-gc (stress test garbage collection of wasm code)
              type: bool    default: false
    --wasm-max-initial-code-space-reservation (maximum size of the initial wasm code space
reservation (in MB))
              type: int    default: 0
    --frame-count (number of stack frames inspected by the profiler)
              type: int    default: 1
    --stress-sampling-allocation-profiler (Enables sampling allocation profiler with X as a
sample interval)
              type: int    default: 0
    --lazy-new-space-shrinking (Enables the lazy new space shrinking strategy)
              type: bool    default: false
    --min-semi-space-size (min size of a semi-space (in MBytes), the new space consists of two
semi-spaces)
              type: size_t    default: 0
    --max-semi-space-size (max size of a semi-space (in MBytes), the new space consists of two
semi-spaces)
              type: size_t    default: 0
    --semi-space-growth-factor (factor by which to grow the new space)
              type: int    default: 2
    --max-old-space-size (max size of the old space (in Mbytes))
              type: size_t    default: 0
    --max-heap-size (max size of the heap (in Mbytes) both max_semi_space_size and
max_old_space_size take precedence. All three flags cannot be specified at the same time.)
              type: size_t    default: 0
    --initial-heap-size (initial size of the heap (in Mbytes))
              type: size_t    default: 0
    --huge-max-old-generation-size (Increase max size of the old space to 4 GB for x64 systems
withthe physical memory bigger than 16 GB)
              type: bool    default: true
    --initial-old-space-size (initial old space size (in Mbytes))
              type: size_t    default: 0
    --global-gc-scheduling (enable GC scheduling based on global memory)
              type: bool    default: true
    --gc-global (always perform global GCs)
```

```
    --turbo-allocation-folding (Turbofan allocation folding)
              type: bool    default: true
    --turbo-instruction-scheduling (enable instruction scheduling in TurboFan)
              type: bool    default: false
    --turbo-stress-instruction-scheduling (randomly schedule instructions to stress dependency
tracking)
              type: bool    default: false
    --turbo-store-elimination (enable store-store elimination in TurboFan)
              type: bool    default: true
    --trace-store-elimination (trace store elimination)
              type: bool    default: false
    --turbo-rewrite-far-jumps (rewrite far to near jumps (ia32,x64))
              type: bool    default: true
    --stress-gc-during-compilation (simulate GC/compiler thread race related to
https://crbug.com/v8/8520)
              type: bool    default: false
    --turbo-fast-api-calls (enable fast API calls from TurboFan)
              type: bool    default: false
    --reuse-opt-code-count (don't discard optimized code for the specified number of deopts.)
              type: int    default: 0
    --turbo-nci (enable experimental native context independent code.)
              type: bool    default: false
    --turbo-nci-as-highest-tier (replace default TF with NCI code as the highest tier for testing
purposes.)
              type: bool    default: false
    --print-nci-code (print native context independent code.)
              type: bool    default: false
    --trace-turbo-nci (trace native context independent code.)
              type: bool    default: false
    --turbo-collect-feedback-in-generic-lowering (enable experimental feedback collection in
generic lowering.)
              type: bool    default: false
    --optimize-for-size (Enables optimizations which favor memory size over execution speed)
              type: bool    default: false
    --untrusted-code-mitigations (Enable mitigations for executing untrusted code)
              type: bool    default: false
    --expose-wasm (expose wasm interface to JavaScript)
              type: bool    default: true
    --assume-asmjs-origin (force wasm decoder to assume input is internal asm-wasm format)
              type: bool    default: false
    --wasm-num-compilation-tasks (maximum number of parallel compilation tasks for wasm)
              type: int    default: 128
    --wasm-write-protect-code-memory (write protect code memory on the wasm native heap)
              type: bool    default: false
    --wasm-async-compilation (enable actual asynchronous compilation for
WebAssembly.compile)
              type: bool    default: true
```

```
  --wasm-test-streaming (use streaming compilation instead of async compilation for tests)
        type: bool  default: false
  --wasm-max-mem-pages (maximum initial number of 64KiB memory pages of a wasm instance)
        type: uint  default: 32767
  --wasm-max-mem-pages-growth (maximum number of 64KiB pages a Wasm memory can grow to)
        type: uint  default: 65536
  --wasm-max-table-size (maximum table size of a wasm instance)
        type: uint  default: 10000000
  --wasm-max-code-space (maximum committed code space for wasm (in MB))
        type: uint  default: 1024
  --wasm-tier-up (enable tier up to the optimizing compiler (requires --liftoff to have an effect))
        type: bool  default: true
  --trace-wasm-ast-start (start function for wasm AST trace (inclusive))
        type: int  default: 0
  --trace-wasm-ast-end (end function for wasm AST trace (exclusive))
        type: int  default: 0
  --liftoff (enable Liftoff, the baseline compiler for WebAssembly)
        type: bool  default: true
  --trace-wasm-memory (print all memory updates performed in wasm code)
        type: bool  default: false
  --wasm-tier-mask-for-testing (bitmask of functions to compile with TurboFan instead of Liftoff)
        type: int  default: 0
  --wasm-expose-debug-eval (Expose wasm evaluator support on the CDP)
        type: bool  default: false
  --validate-asm (validate asm.js modules before compiling)
        type: bool  default: true
  --suppress-asm-messages (don't emit asm.js related messages (for golden file testing))
        type: bool  default: false
  --trace-asm-time (log asm.js timing info to the console)
        type: bool  default: false
  --trace-asm-scanner (log tokens encountered by asm.js scanner)
        type: bool  default: false
  --trace-asm-parser (verbose logging of asm.js parse failures)
        type: bool  default: false
  --stress-validate-asm (try to validate everything as asm.js)
        type: bool  default: false
  --dump-wasm-module-path (directory to dump wasm modules to)
        type: string  default: nullptr
  --experimental-wasm-eh (enable prototype exception handling opcodes for wasm)
        type: bool  default: false
  --experimental-wasm-simd (enable prototype SIMD opcodes for wasm)
        type: bool  default: false
  --experimental-wasm-return-call (enable prototype return call opcodes for wasm)
```

```
        type: bool  default: false
  --experimental-wasm-compilation-hints (enable prototype compilation hints section for wasm)
        type: bool  default: false
  --experimental-wasm-gc (enable prototype garbage collection for wasm)
        type: bool  default: false
  --experimental-wasm-typed-funcref (enable prototype typed function references for wasm)
        type: bool  default: false
  --experimental-wasm-reftypes (enable prototype reference type opcodes for wasm)
        type: bool  default: false
  --experimental-wasm-threads (enable prototype thread opcodes for wasm)
        type: bool  default: false
  --experimental-wasm-type-reflection (enable prototype wasm type reflection in JS for wasm)
        type: bool  default: false
  --experimental-wasm-bigint (enable prototype JS BigInt support for wasm)
        type: bool  default: true
  --experimental-wasm-bulk-memory (enable prototype bulk memory opcodes for wasm)
        type: bool  default: true
  --experimental-wasm-mv (enable prototype multi-value support for wasm)
        type: bool  default: true
  --wasm-staging (enable staged wasm features)
        type: bool  default: false
  --wasm-opt (enable wasm optimization)
        type: bool  default: false
  --wasm-bounds-checks (enable bounds checks (disable for performance testing only))
        type: bool  default: true
  --wasm-stack-checks (enable stack checks (disable for performance testing only))
        type: bool  default: true
  --wasm-math-intrinsics (intrinsify some Math imports into wasm)
        type: bool  default: true
  --wasm-trap-handler (use signal handlers to catch out of bounds memory access in wasm (currently Linux x86_64 only))
        type: bool  default: true
  --wasm-fuzzer-gen-test (generate a test case when running a wasm fuzzer)
        type: bool  default: false
  --print-wasm-code (Print WebAssembly code)
        type: bool  default: false
  --print-wasm-stub-code (Print WebAssembly stub code)
        type: bool  default: false
  --asm-wasm-lazy-compilation (enable lazy compilation for asm-wasm modules)
        type: bool  default: false
  --wasm-lazy-compilation (enable lazy compilation for all wasm modules)
        type: bool  default: false
  --wasm-lazy-validation (enable lazy validation for lazily compiled wasm functions)
        type: bool  default: false
```

--compact-code-space (Compact code space on full collections)
type: bool default: true
--flush-bytecode (flush of bytecode when it has not been executed recently)
type: bool default: true
--stress-flush-bytecode (stress bytecode flushing)
type: bool default: false
--use-marking-progress-bar (Use a progress bar to scan large objects in increments when incremental marking is active.)
type: bool default: true
--stress-per-context-marking-worklist (Use per-context worklist for marking)
type: bool default: false
--force-marking-deque-overflows (force overflows of marking deque by reducing it's size to 64 words)
type: bool default: false
--stress-compaction (stress the GC compactor to flush out bugs (implies --force_marking_deque_overflows))
type: bool default: false
--stress-compaction-random (Stress GC compaction by selecting random percent of pages as evacuation candidates. It overrides stress_compaction.)
type: bool default: false
--stress-incremental-marking (force incremental marking for small heaps and run it more often)
type: bool default: false
--fuzzer-gc-analysis (prints number of allocations and enables analysis mode for gc fuzz testing, e.g. --stress-marking, --stress-scavenge)
type: bool default: false
--stress-marking (force marking at random points between 0 and X (inclusive) percent of the regular marking start limit)
type: int default: 0
--stress-scavenge (force scavenge at random points between 0 and X (inclusive) percent of the new space capacity)
type: int default: 0
--gc-experiment-background-schedule (new background GC schedule heuristics)
type: bool default: false
--gc-experiment-less-compaction (less compaction in non-memory reducing mode)
type: bool default: false
--disable-abortjs (disables AbortJS runtime function)
type: bool default: false
--randomize-all-allocations (randomize virtual memory reservations by ignoring any hints passed when allocating pages)
type: bool default: false
--manual-evacuation-candidates-selection (Test mode only flag. It allows an unit test to select evacuation candidates pages (requires --stress_compaction).)
type: bool default: false
--fast-promotion-new-space (fast promote new space on high survival rates)
type: bool default: false
--clear-free-memory (initialize free memory with 0)

type: bool default: false
--random-gc-interval (Collect garbage after random(0, X) allocations. It overrides gc_interval.)
type: int default: 0
--gc-interval (garbage collect after <n> allocations)
type: int default: -1
--retain-maps-for-n-gc (keeps maps alive for <n> old space garbage collections)
type: int default: 2
--trace-gc (print one trace line following each garbage collection)
type: bool default: false
--trace-gc-nvp (print one detailed trace line in name=value format after each garbage collection)
type: bool default: false
--trace-gc-ignore-scavenger (do not print trace line after scavenger collection)
type: bool default: false
--trace-idle-notification (print one trace line following each idle notification)
type: bool default: false
--trace-idle-notification-verbose (prints the heap state used by the idle notification)
type: bool default: false
--trace-gc-verbose (print more details following each garbage collection)
type: bool default: false
--trace-gc-freelists (prints details of each freelist before and after each major garbage collection)
type: bool default: false
--trace-gc-freelists-verbose (prints details of freelists of each page before and after each major garbage collection)
type: bool default: false
--trace-evacuation-candidates (Show statistics about the pages evacuation by the compaction)
type: bool default: false
--trace-allocations-origins (Show statistics about the origins of allocations. Combine with --no-inline-new to track allocations from generated code)
type: bool default: false
--trace-allocation-stack-interval (print stack trace after <n> free-list allocations)
type: int default: -1
--trace-duplicate-threshold-kb (print duplicate objects in the heap if their size is more than given threshold)
type: int default: 0
--trace-fragmentation (report fragmentation for old space)
type: bool default: false
--trace-fragmentation-verbose (report fragmentation for old space (detailed))
type: bool default: false
--minor-mc-trace-fragmentation (trace fragmentation after marking)
type: bool default: false
--trace-evacuation (report evacuation statistics)
type: bool default: false
--trace-mutator-utilization (print mutator utilization, allocation speed, gc speed)

```
        type: bool  default: false
  --incremental-marking (use incremental marking)
        type: bool  default: true
  --incremental-marking-wrappers (use incremental marking for marking wrappers)
        type: bool  default: true
  --incremental-marking-task (use tasks for incremental marking)
        type: bool  default: true
  --incremental-marking-soft-trigger (threshold for starting incremental marking via a task in percent of available space: limit - size)
        type: int  default: 0
  --incremental-marking-hard-trigger (threshold for starting incremental marking immediately in percent of available space: limit - size)
        type: int  default: 0
  --trace-unmapper (Trace the unmapping)
        type: bool  default: false
  --parallel-scavenge (parallel scavenge)
        type: bool  default: true
  --scavenge-task (schedule scavenge tasks)
        type: bool  default: true
  --scavenge-task-trigger (scavenge task trigger in percent of the current heap limit)
        type: int  default: 80
  --scavenge-separate-stack-scanning (use a separate phase for stack scanning in scavenge)
        type: bool  default: false
  --trace-parallel-scavenge (trace parallel scavenge)
        type: bool  default: false
  --write-protect-code-memory (write protect code memory)
        type: bool  default: true
  --concurrent-marking (use concurrent marking)
        type: bool  default: true
  --concurrent-array-buffer-sweeping (concurrently sweep array buffers)
        type: bool  default: true
  --concurrent-allocation (concurrently allocate in old space)
        type: bool  default: false
  --local-heaps (allow heap access from background tasks)
        type: bool  default: false
  --stress-concurrent-allocation (start background threads that allocate memory)
        type: bool  default: false
  --parallel-marking (use parallel marking in atomic pause)
        type: bool  default: true
  --ephemeron-fixpoint-iterations (number of fixpoint iterations it takes to switch to linear ephemeron algorithm)
        type: int  default: 10
  --trace-concurrent-marking (trace concurrent marking)
        type: bool  default: false
  --concurrent-store-buffer (use concurrent store buffer processing)
        type: bool  default: true
  --concurrent-sweeping (use concurrent sweeping)
```

```
        type: bool  default: true
  --parallel-compaction (use parallel compaction)
        type: bool  default: true
  --parallel-pointer-update (use parallel pointer update during compaction)
        type: bool  default: true
  --detect-ineffective-gcs-near-heap-limit (trigger out-of-memory failure to avoid GC storm near heap limit)
        type: bool  default: true
  --trace-incremental-marking (trace progress of the incremental marking)
        type: bool  default: false
  --trace-stress-marking (trace stress marking progress)
        type: bool  default: false
  --trace-stress-scavenge (trace stress scavenge progress)
        type: bool  default: false
  --track-gc-object-stats (track object counts and memory usage)
        type: bool  default: false
  --trace-gc-object-stats (trace object counts and memory usage)
        type: bool  default: false
  --trace-zone-stats (trace zone memory usage)
        type: bool  default: false
  --zone-stats-tolerance (report a tick only when allocated zone memory changes by this amount)
        type: size_t  default: 1048576
  --track-retaining-path (enable support for tracking retaining path)
        type: bool  default: false
  --concurrent-array-buffer-freeing (free array buffer allocations on a background thread)
        type: bool  default: true
  --gc-stats (Used by tracing internally to enable gc statistics)
        type: int  default: 0
  --track-detached-contexts (track native contexts that are expected to be garbage collected)
        type: bool  default: true
  --trace-detached-contexts (trace native contexts that are expected to be garbage collected)
        type: bool  default: false
  --move-object-start (enable moving of object starts)
        type: bool  default: true
  --memory-reducer (use memory reducer)
        type: bool  default: true
  --memory-reducer-for-small-heaps (use memory reducer for small heaps)
        type: bool  default: true
  --heap-growing-percent (specifies heap growing factor as (1 + heap_growing_percent/100))
        type: int  default: 0
  --v8-os-page-size (override OS page size (in KBytes))
        type: int  default: 0
  --always-compact (Perform compaction on every full GC)
        type: bool  default: false
  --never-compact (Never perform compaction on full GC - testing only)
        type: bool  default: false
```

```
            type: bool    default: false
   --hard-abort (abort by crashing)
            type: bool    default: true
   --expose-inspector-scripts (expose injected-script-source.js for debugging)
            type: bool    default: false
   --stack-size (default size of stack region v8 is allowed to use (in kBytes))
            type: int    default: 984
   --max-stack-trace-source-length (maximum length of function source code printed in a stack
trace.)
            type: int    default: 300
   --clear-exceptions-on-js-entry (clear pending exceptions when entering JavaScript)
            type: bool    default: false
   --histogram-interval (time interval in ms for aggregating memory histograms)
            type: int    default: 600000
   --heap-profiler-trace-objects (Dump heap object allocations/movements/size_updates)
            type: bool    default: false
   --heap-profiler-use-embedder-graph (Use the new EmbedderGraph API to get embedder
nodes)
            type: bool    default: true
   --heap-snapshot-string-limit (truncate strings to this length in the heap snapshot)
            type: int    default: 1024
   --sampling-heap-profiler-suppress-randomness (Use constant sample intervals to eliminate
test flakiness)
            type: bool    default: false
   --use-idle-notification (Use idle notification to reduce memory footprint.)
            type: bool    default: true
   --trace-ic (trace inline cache state transitions for tools/ic-processor)
            type: bool    default: false
   --modify-field-representation-inplace (enable in-place field representation updates)
            type: bool    default: true
   --max-polymorphic-map-count (maximum number of maps to track in POLYMORPHIC
state)
            type: int    default: 4
   --native-code-counters (generate extra code for manipulating stats counters)
            type: bool    default: false
   --thin-strings (Enable ThinString support)
            type: bool    default: true
   --trace-prototype-users (Trace updates to prototype user tracking)
            type: bool    default: false
   --trace-for-in-enumerate (Trace for-in enumerate slow-paths)
            type: bool    default: false
   --trace-maps (trace map creation)
            type: bool    default: false
   --trace-maps-details (also log map details)
            type: bool    default: true
   --allow-natives-syntax (allow natives syntax)
            type: bool    default: false
```

```
            type: bool    default: false
   --young-generation-large-objects (allocates large objects by default in the young generation
large object space)
            type: bool    default: true
   --debug-code (generate extra code (assertions) for debugging)
            type: bool    default: false
   --code-comments (emit comments in code disassembly; for more readable source positions
you should add --no-concurrent_recompilation)
            type: bool    default: false
   --enable-sse3 (enable use of SSE3 instructions if available)
            type: bool    default: true
   --enable-ssse3 (enable use of SSSE3 instructions if available)
            type: bool    default: true
   --enable-sse4-1 (enable use of SSE4.1 instructions if available)
            type: bool    default: true
   --enable-sse4-2 (enable use of SSE4.2 instructions if available)
            type: bool    default: true
   --enable-sahf (enable use of SAHF instruction if available (X64 only))
            type: bool    default: true
   --enable-avx (enable use of AVX instructions if available)
            type: bool    default: true
   --enable-fma3 (enable use of FMA3 instructions if available)
            type: bool    default: true
   --enable-bmi1 (enable use of BMI1 instructions if available)
            type: bool    default: true
   --enable-bmi2 (enable use of BMI2 instructions if available)
            type: bool    default: true
   --enable-lzcnt (enable use of LZCNT instruction if available)
            type: bool    default: true
   --enable-popcnt (enable use of POPCNT instruction if available)
            type: bool    default: true
   --arm-arch (generate instructions for the selected ARM architecture if available: armv6,
armv7, armv7+sudiv or armv8)
            type: string    default: armv8
   --force-long-branches (force all emitted branches to be in long mode (MIPS/PPC only))
            type: bool    default: false
   --mcpu (enable optimization for specific cpu)
            type: string    default: auto
   --partial-constant-pool (enable use of partial constant pools (X64 only))
            type: bool    default: true
   --sim-arm64-optional-features (enable optional features on the simulator for testing: none or
all)
            type: string    default: none
   --enable-source-at-csa-bind (Include source information in the binary at CSA bind
locations.)
            type: bool    default: false
   --enable-armv7 (deprecated (use --arm_arch instead))
```

```
            type: maybe_bool    default: unset
    --enable-vfp3 (deprecated (use --arm_arch instead))
            type: maybe_bool    default: unset
    --enable-32dregs (deprecated (use --arm_arch instead))
            type: maybe_bool    default: unset
    --enable-neon (deprecated (use --arm_arch instead))
            type: maybe_bool    default: unset
    --enable-sudiv (deprecated (use --arm_arch instead))
            type: maybe_bool    default: unset
    --enable-armv8 (deprecated (use --arm_arch instead))
            type: maybe_bool    default: unset
    --enable-regexp-unaligned-accesses (enable unaligned accesses for the regexp engine)
            type: bool    default: true
    --script-streaming (enable parsing on background)
            type: bool    default: true
    --stress-background-compile (stress test parsing on background)
            type: bool    default: false
    --finalize-streaming-on-background (perform the script streaming finalization on the
background thread)
            type: bool    default: false
    --disable-old-api-accessors (Disable old-style API accessors whose setters trigger through
the prototype chain)
            type: bool    default: false
    --expose-gc (expose gc extension)
            type: bool    default: false
    --expose-gc-as (expose gc extension under the specified name)
            type: string    default: nullptr
    --expose-externalize-string (expose externalize string extension)
            type: bool    default: false
    --expose-trigger-failure (expose trigger-failure extension)
            type: bool    default: false
    --stack-trace-limit (number of stack frames to capture)
            type: int    default: 10
    --builtins-in-stack-traces (show built-in functions in stack traces)
            type: bool    default: false
    --experimental-stack-trace-frames (enable experimental frames (API/Builtins) and stack
trace layout)
            type: bool    default: false
    --disallow-code-generation-from-strings (disallow eval and friends)
            type: bool    default: false
    --expose-async-hooks (expose async_hooks object)
            type: bool    default: false
    --expose-cputracemark-as (expose cputracemark extension under the specified name)
            type: string    default: nullptr
    --allow-unsafe-function-constructor (allow invoking the function constructor without
security checks)
            type: bool    default: false
```

```
    --force-slow-path (always take the slow path for builtins)
            type: bool    default: false
    --test-small-max-function-context-stub-size (enable testing the function context size
overflow path by making the maximum size smaller)
            type: bool    default: false
    --inline-new (use fast inline allocation)
            type: bool    default: true
    --trace (trace javascript function calls)
            type: bool    default: false
    --trace-wasm (trace wasm function calls)
            type: bool    default: false
    --lazy (use lazy compilation)
            type: bool    default: true
    --max-lazy (ignore eager compilation hints)
            type: bool    default: false
    --trace-opt (trace lazy optimization)
            type: bool    default: false
    --trace-opt-verbose (extra verbose compilation tracing)
            type: bool    default: false
    --trace-opt-stats (trace lazy optimization statistics)
            type: bool    default: false
    --trace-deopt (trace optimize function deoptimization)
            type: bool    default: false
    --trace-file-names (include file names in trace-opt/trace-deopt output)
            type: bool    default: false
    --always-opt (always try to optimize functions)
            type: bool    default: false
    --always-osr (always try to OSR functions)
            type: bool    default: false
    --prepare-always-opt (prepare for turning on always opt)
            type: bool    default: false
    --trace-serializer (print code serializer trace)
            type: bool    default: false
    --compilation-cache (enable compilation cache)
            type: bool    default: true
    --cache-prototype-transitions (cache prototype transitions)
            type: bool    default: true
    --parallel-compile-tasks (enable parallel compile tasks)
            type: bool    default: false
    --compiler-dispatcher (enable compiler dispatcher)
            type: bool    default: false
    --trace-compiler-dispatcher (trace compiler dispatcher activity)
            type: bool    default: false
    --cpu-profiler-sampling-interval (CPU profiler sampling interval in microseconds)
            type: int    default: 1000
    --trace-side-effect-free-debug-evaluate (print debug messages for side-effect-free debug-
evaluate for testing)
```

--map-counters (Map counters to a file)
type: string default:
--mock-arraybuffer-allocator (Use a mock ArrayBuffer allocator for testing.)
type: bool default: false
--mock-arraybuffer-allocator-limit (Memory limit for mock ArrayBuffer allocator used to simulate OOM for testing.)
type: size_t default: 0
--gdbjit (enable GDBJIT interface)
type: bool default: false
--gdbjit-full (enable GDBJIT interface for all code objects)
type: bool default: false
--gdbjit-dump (dump elf objects with debug info to disk)
type: bool default: false
--gdbjit-dump-filter (dump only objects containing this substring)
type: string default:
--log (Minimal logging (no API, code, GC, suspect, or handles samples).)
type: bool default: false
--log-all (Log all events to the log file.)
type: bool default: false
--log-api (Log API events to the log file.)
type: bool default: false
--log-code (Log code events to the log file without profiling.)
type: bool default: false
--log-handles (Log global handle events.)
type: bool default: false
--log-suspect (Log suspect operations.)
type: bool default: false
--log-source-code (Log source code.)
type: bool default: false
--log-function-events (Log function events (parse, compile, execute) separately.)
type: bool default: false
--prof (Log statistical profiling information (implies --log-code).)
type: bool default: false
--detailed-line-info (Always generate detailed line information for CPU profiling.)
type: bool default: false
--prof-sampling-interval (Interval for --prof samples (in microseconds).)
type: int default: 1000
--prof-cpp (Like --prof, but ignore generated code.)
type: bool default: false
--prof-browser-mode (Used with --prof, turns on browser-compatible mode for profiling.)
type: bool default: true
--logfile (Specify the name of the log file.)
type: string default: v8.log
--logfile-per-isolate (Separate log files for each isolate.)
type: bool default: true
--ll-prof (Enable low-level linux profiler.)
type: bool default: false

--allow-natives-for-differential-fuzzing (allow only natives explicitly allowlisted for differential fuzzers)
type: bool default: false
--parse-only (only parse the sources)
type: bool default: false
--trace-sim (Trace simulator execution)
type: bool default: false
--debug-sim (Enable debugging the simulator)
type: bool default: false
--check-icache (Check icache flushes in ARM and MIPS simulator)
type: bool default: false
--stop-sim-at (Simulator stop after x number of instructions)
type: int default: 0
--sim-stack-alignment (Stack alignment in bytes in simulator (4 or 8, 8 is default))
type: int default: 8
--sim-stack-size (Stack size of the ARM64, MIPS64 and PPC64 simulator in kBytes (default is 2 MB))
type: int default: 2048
--log-colour (When logging, try to use coloured output.)
type: bool default: true
--trace-sim-messages (Trace simulator debug messages. Implied by --trace-sim.)
type: bool default: false
--async-stack-traces (include async stack traces in Error.stack)
type: bool default: true
--stack-trace-on-illegal (print stack trace when an illegal exception is thrown)
type: bool default: false
--abort-on-uncaught-exception (abort program (dump core) when an uncaught exception is thrown)
type: bool default: false
--correctness-fuzzer-suppressions (Suppress certain unspecified behaviors to ease correctness fuzzing: Abort program when the stack overflows or a string exceeds maximum length (as opposed to throwing RangeError). Use a fixed suppression string for error messages.)
type: bool default: false
--randomize-hashes (randomize hashes to avoid predictable hash collisions (with snapshots this option cannot override the baked-in seed))
type: bool default: true
--rehash-snapshot (rehash strings from the snapshot to override the baked-in seed)
type: bool default: true
--hash-seed (Fixed seed to use to hash property keys (0 means random)(with snapshots this option cannot override the baked-in seed))
type: uint64 default: 0
--random-seed (Default seed for initializing random generator (0, the default, means to use system random).)
type: int default: 0
--fuzzer-random-seed (Default seed for initializing fuzzer random generator (0, the default, means to use v8's random number generator seed).)

```
            type: int    default: 0
   --trace-rail (trace RAIL mode)
            type: bool    default: false
   --print-all-exceptions (print exception object and stack trace on each thrown exception)
            type: bool    default: false
   --detailed-error-stack-trace (includes arguments for each function call in the error stack
frames array)
            type: bool    default: false
   --adjust-os-scheduling-parameters (adjust OS specific scheduling params for the isolate)
            type: bool    default: true
   --runtime-call-stats (report runtime call counts and times)
            type: bool    default: false
   --rcs (report runtime call counts and times)
            type: bool    default: false
   --rcs-cpu-time (report runtime times in cpu time (the default is wall time))
            type: bool    default: false
   --profile-deserialization (Print the time it takes to deserialize the snapshot.)
            type: bool    default: false
   --serialization-statistics (Collect statistics on serialized objects.)
            type: bool    default: false
   --serialization-chunk-size (Custom size for serialization chunks)
            type: uint    default: 4096
   --regexp-optimization (generate optimized regexp code)
            type: bool    default: true
   --regexp-mode-modifiers (enable inline flags in regexp.)
            type: bool    default: false
   --regexp-interpret-all (interpret all regexp code)
            type: bool    default: false
   --regexp-tier-up (enable regexp interpreter and tier up to the compiler after the number of
executions set by the tier up ticks flag)
            type: bool    default: true
   --regexp-tier-up-ticks (set the number of executions for the regexp interpreter before tiering-
up to the compiler)
            type: int    default: 1
   --regexp-peephole-optimization (enable peephole optimization for regexp bytecode)
            type: bool    default: true
   --trace-regexp-peephole-optimization (trace regexp bytecode peephole optimization)
            type: bool    default: false
   --trace-regexp-bytecodes (trace regexp bytecode execution)
            type: bool    default: false
   --trace-regexp-assembler (trace regexp macro assembler calls.)
            type: bool    default: false
   --trace-regexp-parser (trace regexp parsing)
            type: bool    default: false
   --trace-regexp-tier-up (trace regexp tiering up execution)
            type: bool    default: false
   --testing-bool-flag (testing_bool_flag)
```

```
            type: bool    default: true
   --testing-maybe-bool-flag (testing_maybe_bool_flag)
            type: maybe_bool    default: unset
   --testing-int-flag (testing_int_flag)
            type: int    default: 13
   --testing-float-flag (float-flag)
            type: float    default: 2.5
   --testing-string-flag (string-flag)
            type: string    default: Hello, world!
   --testing-prng-seed (Seed used for threading test randomness)
            type: int    default: 42
   --testing-d8-test-runner (test runner turns on this flag to enable a check that the function was
prepared for optimization before marking it for optimization)
            type: bool    default: false
   --fuzzing (Fuzzers use this flag to signal that they are ... fuzzing. This causes intrinsics to
fail silently (e.g. return undefined) on invalid usage.)
            type: bool    default: false
   --embedded-src (Path for the generated embedded data file. (mksnapshot only))
            type: string    default: nullptr
   --embedded-variant (Label to disambiguate symbols in embedded data file. (mksnapshot
only))
            type: string    default: nullptr
   --startup-src (Write V8 startup as C++ src. (mksnapshot only))
            type: string    default: nullptr
   --startup-blob (Write V8 startup blob file. (mksnapshot only))
            type: string    default: nullptr
   --target-arch (The mksnapshot target arch. (mksnapshot only))
            type: string    default: nullptr
   --target-os (The mksnapshot target os. (mksnapshot only))
            type: string    default: nullptr
   --target-is-simulator (Instruct mksnapshot that the target is meant to run in the simulator and
it can generate simulator-specific instructions. (mksnapshot only))
            type: bool    default: false
   --minor-mc-parallel-marking (use parallel marking for the young generation)
            type: bool    default: true
   --trace-minor-mc-parallel-marking (trace parallel marking for the young generation)
            type: bool    default: false
   --minor-mc (perform young generation mark compact GCs)
            type: bool    default: false
   --help (Print usage message, including flags, on console)
            type: bool    default: true
   --dump-counters (Dump counters on exit)
            type: bool    default: false
   --dump-counters-nvp (Dump counters as name-value pairs on exit)
            type: bool    default: false
   --use-external-strings (Use external strings for source code)
            type: bool    default: false
```

```
  --gc-fake-mmap (Specify the name of the file for fake gc mmap used in ll_prof)
        type: string  default: /tmp/__v8_gc__
  --log-internal-timer-events (Time internal events.)
        type: bool  default: false
  --redirect-code-traces (output deopt information and disassembly into file code-<pid>-<isolate id>.asm)
        type: bool  default: false
  --redirect-code-traces-to (output deopt information and disassembly into the given file)
        type: string  default: nullptr
  --print-opt-source (print source code of optimized and inlined functions)
        type: bool  default: false
  --vtune-prof-annotate-wasm (Used when v8_enable_vtunejit is enabled, load wasm source map and provide annotate support (experimental).)
        type: bool  default: false
  --win64-unwinding-info (Enable unwinding info for Windows/x64)
        type: bool  default: true
  --interpreted-frames-native-stack (Show interpreted frames on the native stack (useful for external profilers).)
        type: bool  default: false
  --predictable (enable predictable mode)
        type: bool  default: false
  --predictable-gc-schedule (Predictable garbage collection schedule. Fixes heap growing, idle, and memory reducing behavior.)
        type: bool  default: false
  --single-threaded (disable the use of background tasks)
        type: bool  default: false
  --single-threaded-gc (disable the use of background gc tasks)
        type: bool  default: false
```

Particularly useful ones:

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

19.6 Release Schedule

A new minor release for the `deno` cli is released every month, on the third Thursday of the month.
See Milestones on Deno's GitHub for the upcoming releases.
There are usually two or three patch releases (done weekly) after a minor releases; after that a merge window for new features opens for the upcoming minor release.
Stable releases can be found on the GitHub releases page.

19.6.1 Canary channel

In addition to the stable channel described above, canaries are released multiple times daily (for each commit on main). You can upgrade to the latest canary release by running:

```
deno upgrade --canary
```

To update to a specific canary, pass the commit hash in the `--version` option:

```
deno upgrade --canary --version=973af61d8bb03c1709f61e456581d58386ed4952
```

To switch back to the stable channel, run `deno upgrade`.
Canaries can be downloaded from https://dl.deno.land.