Created with a trial version of Syncfusion Essential PDF

DENO

SUCCINCTLY

BY **MARK LEWIN**

SUCCINCTLY EBOOK SERIES

Syncfusion

Deno Succinctly

By
Mark Lewin

Foreword by Daniel Jebaraj



Copyright © 2021 by Syncfusion, Inc.

2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA
All rights reserved.

ISBN: 978-1-64200-214-0

Important licensing information. Please read.

This book is available for free download from www.syncfusion.com on completion of a registration form.

If you obtained this book from any other source, please register and download a free copy from www.syncfusion.com.

This book is licensed for reading only if obtained from www.syncfusion.com.

This book is licensed strictly for personal or educational use.

Redistribution in any form is prohibited.

The authors and copyright holders provide absolutely no warranty for any information provided.

The authors and copyright holders shall not be liable for any claim, damages, or any other liability arising from, out of, or in connection with the information in this book.

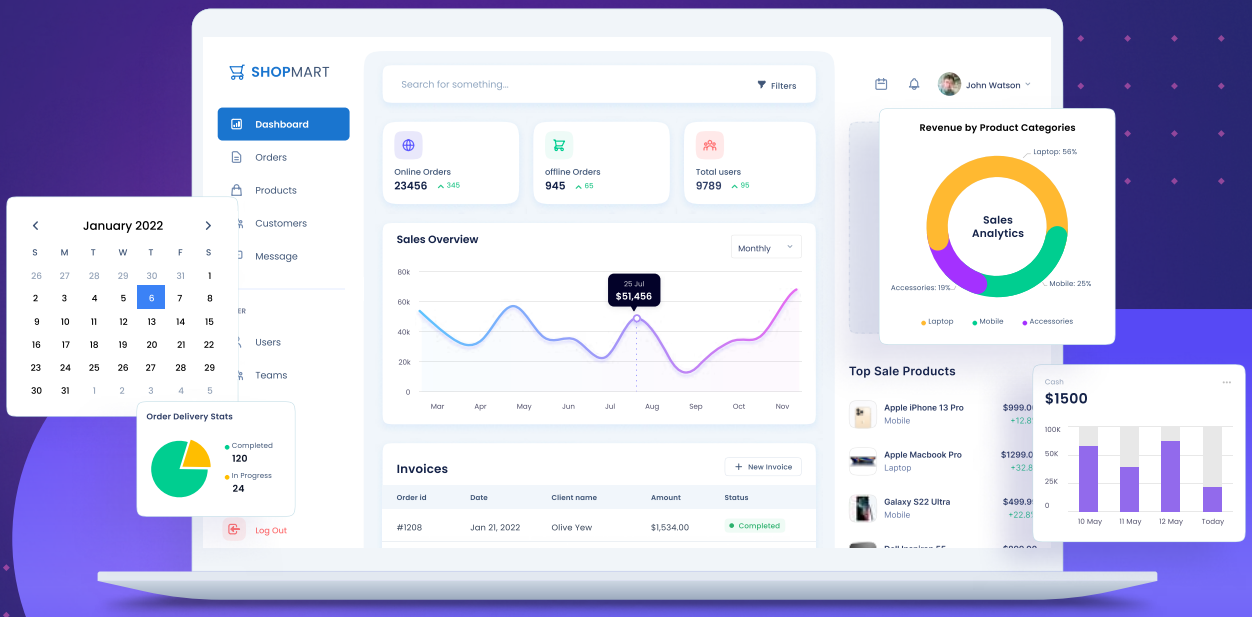
Please do not use this book if the listed terms are unacceptable.

Use shall constitute acceptance of the terms listed.

SYNCFUSION, SUCCINCTLY, DELIVER INNOVATION WITH EASE, ESSENTIAL, and .NET ESSENTIALS are the registered trademarks of Syncfusion, Inc.

Technical Reviewer: James McCaffrey
Copy Editor: Courtney Wright
Acquisitions Coordinator: Tres Watkins, VP of content, Syncfusion, Inc.
Proofreader: Graham High, senior content producer, Syncfusion, Inc.

THE WORLD'S BEST UI COMPONENT SUITE FOR BUILDING POWERFUL APPS



GET YOUR FREE .NET AND JAVASCRIPT UI COMPONENTS

syncfusion.com/communitylicense

4.6 out of 5 stars

- 1,700+ components for mobile, web, and desktop platforms
- Support within 24 hours on all business days
- Uncompromising quality
- Hassle-free licensing
- 28000+ customers
- 20+ years in business

Trusted by the world's leading companies

Syncfusion

Table of Contents

The Story Behind the *Succinctly* Series of Books **7**

About the Author **9**

Chapter 1 Introducing Deno **10**

What is Deno, and why should I care? 10

The Deno runtime and the V8 engine 11

Architecture and internals 12

Installing Deno 13

Windows 13

macOS/Linux 14

Verifying the installation 14

Upgrading Deno 15

Getting help 15

Your first app 16

Running Deno code 16

Creating a simple server application 17

Chapter 2 Deno and Node.js **20**

Dependencies 20

How do I find modules? 23

How are modules versioned? 23

Security 24

Deno tools 25

Deno: the end of Node? 26

Chapter 3 Create a URL Shortener Console Application **28**

Creating your project 28

Accepting command-line arguments....29
Validating the URL....32
Shortening the URL....33
Top-level await....35
Logging....35
Deploying our console application....37
Summary....38
Chapter 4 Code a Static Site Generator....39
Configuring the application....40
Creating the input files....44
Reading the input files....46
Parsing the post metadata....47
Using templates....51
Creating the home page....55
Summary....63
Chapter 5 Build a RESTful API....64
Deno API frameworks....64
The Dinosaur API....66
Creating the server....66
Testing your API using Postman....68
Structuring the application....70
Managing dependencies....70
Refactoring the codebase....71
Setting up the database....73
Storing the database credentials....84
Creating the schema....85

Connecting to the database....86
Listing all the dinosaurs....87
Adding dinosaurs....90
Finding a specific dinosaur....94
Editing a dinosaur....96
Deleting a dinosaur....100
Adding extra functionality with middleware....102
Route not found middleware....103
Request logging middleware....104
Summary....107
Chapter 6 Conclusion and Resources....108

The Story Behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!





About the Author

Mark Lewin has been developing, teaching, and writing about software for more than 20 years. His main focus these days is documenting APIs and teaching developers how to use them. He currently works as a technical writer for a leading provider of communication APIs.

Mark is the author of the Pluralsight "Google Maps API: Get Started" course and several titles in the Syncfusion *Succinctly* series.

Contact him at mark@marklewin.com or on Twitter at @devtechwriter.

Chapter 1 Introducing Deno

What is Deno, and why should I care?

Deno is a new, secure runtime for JavaScript. It was invented by Ryan Dahl. Does that name sound familiar? It should! Dahl is the creator of the incredibly successful Node. Node (also called Node.js) was released in 2009 as a way of writing server applications in JavaScript, thereby enabling developers to write both front- and back-end code in the same programming language. It was immediately successful and continues to be incredibly popular.



Figure 1: Deno is pronounced "Dean-oh" (to rhyme with “chino”) and not "Dino" (as in “dinosaur”), even though its mascot is this rather cute dinosaur!

So, what is Deno, apart from an anagram of Node? After over a decade of Node, Dahl took time to reflect on his creation and consider things that he would have done better, if he had known back in 2009 what he knows now. In that decade, web development itself also moved on, and better technologies became available to support some of the things that Node was designed to do. The result of those reflections was Deno—a new way of writing server-side applications in JavaScript that is like Node but improves upon it in several ways.

These are some of the ways in which Deno improves on Node:

- **Is secure by default**. Deno doesn’t allow programs to access your file system, network, or environment unless you explicitly give that program permission to do so.
- **Supports TypeScript out of the box**. Of course, you can write Node applications in TypeScript too, but you need to install extra packages to do so—and keep them up to date.
- **Ships as a single executable file.**
- **Includes built-in tools.** Includes tools such as a dependency inspector (**`deno info`**) and a code formatter (**`deno fmt`**).
- **Includes standard modules.** Deno has a set of audited standard **modules** that are guaranteed to work.

That's what developers notice when they use Deno. Under the hood, there are significant architectural changes that make Deno an entirely different beast than Node. But I think it's still safe to say that if you know Node, you'll find it easy to get up and running in Deno. In fact, I'd go further and say that you are likely to find the whole experience liberating! That's the purpose of this book: to give you insight into what Deno is, how it works, and why its future looks so promising.

When someone with as much cachet as Ryan Dahl has the humility to say of his creation "Yes, it was good, but it could be better," it's hard to ignore! Deno is still in its early days, and it is arguably not quite ready for mainstream use. However, I'm excited about the possibilities for Deno, and I hope that after reading this book, you will be too.

The Deno runtime and the V8 engine

As we've already established, Deno is a secure runtime for JavaScript. Node is also a JavaScript runtime. Both are built on the V8 engine. So, what is this V8 engine? Well, before we look at V8 specifically, let's discuss what an *engine* is.

An engine takes our source code and compiles it into instructions that can be understood and processed by the machine that it is running on. Initially, all JavaScript code was intended to be run within a web browser. Every browser has an engine, although that engine varies depending on the browser you use. That's why you can fire up the developer tools in any browser and start writing JavaScript straight away.

V8 is just one such engine, specifically the one that drives the Google Chrome web browser. It's a highly optimized engine written in C++ that, like all engines, understands how to turn source code into machine-readable instructions.

However, the purpose of technologies like Deno and Node is to let you run code *independently* of the web browser. So, the Deno and Node runtimes have their own instances of the V8 engine, and you can think of those runtimes as a kind of "wrapper" around this V8 engine. Without a runtime, your machine doesn't understand what JavaScript is, or how to run it.

Equipped with its own version of the V8 engine, the Deno runtime can turn your JavaScript into machine code. But it also provides extra tools and functionality to make your life as a developer easier. For example, it gives you the ability to write your code in TypeScript. The V8 engine doesn't understand TypeScript, but Deno can take your TypeScript code, convert it into JavaScript, and then hand that to the V8 engine for compilation.



Figure 2: The V8 engine

Architecture and internals

The main building blocks of Deno are **Rust**, **Tokio**, and **V8**:

- **Rust** is a modern systems programming language developed by Mozilla, with a focus on high performance and safe concurrency. Rust shares a similar syntax with C++, but has memory safety features that don't rely on garbage collection. The initial choice for Deno was the Go programming language, but issues with garbage collection forced a switch to Rust.
- **Tokio** is an "event-driven, non-blocking I/O platform for writing asynchronous applications with the Rust programming language. At a high level, it provides a few major components: Tools for working with asynchronous tasks, including synchronization primitives and channels and timeouts, delays, and intervals." (Description taken from Tokio documentation.)
- **V8**, as we have already learned, is Google's open-source JavaScript engine, written in C++, and which features in the Chrome web browser.

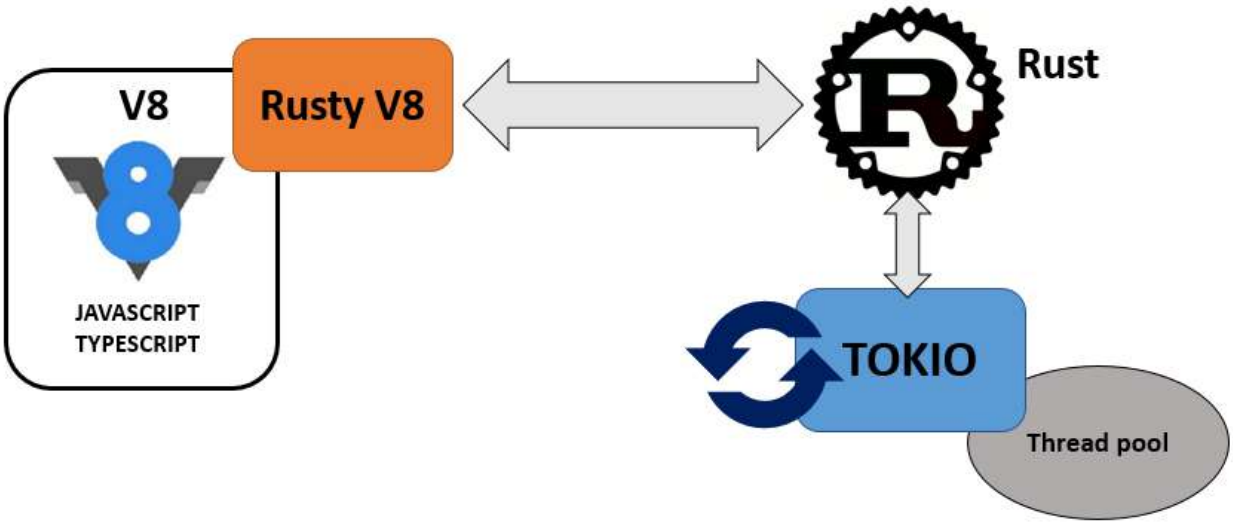


Figure 3: The building blocks of Deno

You can consider programs written in **TypeScript** or **JavaScript** as being Deno's front end. If you create a TypeScript (.ts) file and submit it to Deno, then Deno will first compile it to JavaScript and then submit it to V8. If you give it a JavaScript (.js) file, it will skip the compilation step. Anything you do in TypeScript or JavaScript is considered "unprivileged," in that it doesn't have access to the network, file system, or anything else outside of Deno's sandbox, so it is secure by default.

Anything V8 needs to do outside of this unprivileged environment needs to be submitted to Deno's **Rust** back end. For example, if you want to access files, send emails, create servers, set timeouts, and so on, all that fun stuff needs to be handled by Rust. The communication between the unprivileged side (JavaScript/TypeScript) and the privileged side (Rust) is managed by a set of JavaScript bindings for Rust called **Rusty_v8**.

JavaScript is a single-threaded application, which is fine for a single browser instance, but totally inadequate for server applications that need to respond to requests from multiple clients simultaneously. For that we need asynchronous I/O running in an *event loop*.

The event loop:

- Is an endless loop that waits for tasks, executes them, and then sleeps until it receives more tasks.
- Executes tasks from the event queue starting from the oldest first, but only when the call stack is empty—that is, there is no ongoing task.
- Uses callbacks and promises to signify the completion status of the task.

In Deno, this event loop is provided by **Tokio**.

At an extremely high level, that's Deno. We don't need to dwell too much on the internals here. There are plenty of in-depth articles available online if you want to geek out about some of the architectural decisions that went into creating Deno. I hope, however, that this has given you some idea of what's going on behind the scenes.

Installing Deno

The **deno** executable is just a single binary with no external dependencies, and therefore installation is a breeze.

Windows

Probably the easiest way to install Deno on Windows is by using a package manager such as Chocolatey:

Code Listing 1

```
choco install deno
```

You can also install it from the command line in PowerShell. Be sure to run the shell with Administrator privileges:

Code Listing 2

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

Note these code approaches install the *latest* version of Deno. If you want to install a specific version (for example, version 1.0.0), you need to do the following:

Code Listing 3

```
$v="1.0.0"; iwr https://deno.land/x/install/install.ps1 -useb | iex
```

macOS/Linux

If you are on a Mac and have Homebrew installed, you can install the latest version of Deno with the following command:

Code Listing 4

```
brew install deno
```

You should also be able to do something similar with your package manager on Linux, the command for which will vary depending on the version you are running.

You can also install it using **`curl:`**

Code Listing 5

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

To install a specific version, use the **`-s`** argument:

Code Listing 6

```
curl -fsSL https://deno.land/x/install/install.sh | sh -s v1.0.0
```

Verifying the installation

Once you have installed Deno, check that everything is okay by running:

Code Listing 7

```
deno --version
```

This command should display the version number of Deno, and of V8 and TypeScript. If it does, you're good to go! If not, please review the installation instructions in the Deno manual.

Deno also gives you a REPL (Read Evaluate Print Loop) that you can use to execute commands interactively. Start the REPL by executing **`deno`**, and test it with a simple **`console.log`** statement:

Code Listing 8

```
$ deno
Deno 1.5.3
exit using ctrl+d or close()
```

```
> console.log("Hello from Deno!")
Hello from Deno!
undefined
>
```

Finally, you'll want to fine-tune your development environment. This involves setting environment variables, enabling autocomplete for your shell, and getting your chosen IDE or editor to recognize Deno code. Because these steps vary depending on your OS and the tools you use, I'll refer you to the Deno manual.

Upgrading Deno

To upgrade your installation to a specific version of Deno, run the following, where 1.5.9 is the version you want to upgrade to:

Code Listing 9

```
$ deno upgrade –version 1.5.9
```

If you just want to upgrade to the latest version, run:

Code Listing 10

```
$ deno upgrade
```

Getting help

If you forget the URL for the docs, how to start the REPL, which command-line options are available, or anything else for that matter, run **`deno help`**:

Code Listing 11

```
$ deno help
deno 1.5.8
A secure JavaScript and TypeScript runtime

Docs: https://deno.land/manual
Modules: https://deno.land/std/ https://deno.land/x/
Bugs: https://github.com/denoland/deno/issues

To start the REPL:
  deno

To execute a script:
```

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

To evaluate code in the shell:
  deno eval "console.log(30933 + 404)"

USAGE:
    deno [OPTIONS] [SUBCOMMAND]

OPTIONS:
    -h, --help                       Prints help information
    -L, --log-level <log-level>      Set log level [possible values: debug,
info]
    -q, --quiet                      Suppress diagnostic output
    -V, --version                    Prints version information

…
```



Tip: As well as `deno help` and the Deno manual, another great place to get help is the community Discord group.

Now that you have your Deno development environment all set up, it's time to write a little test app.

Your first app

Let's first create the traditional "Hello Word" example, and then we'll extend it to run as a server application.

Running Deno code

Create a directory for the code you'll write in this book and, within that directory, create a file called **`HelloDeno.ts`**. Note the **`.ts`** extension, which tells Deno that this is a TypeScript file. Deno has first-class support for TypeScript, and all its standard library modules (which we'll talk about in the next chapter) are written in TypeScript. As you've seen, Deno can handle JavaScript files too, but in this book, we're going to take advantage of the strong typing available in TypeScript.



Tip: If you're not familiar with TypeScript, I can thoroughly recommend* TypeScript Succinctly*, which you can download for free.

In **HelloDeno.ts**, write the following code:

Code Listing 12

```
console.log("Hello world ... from Deno!");
```

Then execute your program using **deno run**:

Code Listing 13

```
$ deno run HelloDeno.ts
Check file: .../HelloDeno.ts
Hello world ... from Deno!
```

The **Check file** step is Deno compiling your TypeScript into vanilla JavaScript. If you supply your code as a JavaScript file (with the **.js** extension), Deno omits this step:

Code Listing 14

```
$ deno run HelloDeno.js
Hello world ... from Deno!
```

Creating a simple server application

Let's now make this first example a bit more "Deno-ish" by rewriting it as a server application. We'll spin up a server, let it listen for incoming requests and have our app respond with some JSON that contains our **Hello world … from Deno!** message.

Create a new file called **server.ts** that contains the following code:

Code Listing 15

```
import { serve } from "https://deno.land/std/http/server.ts";
const s = serve({ port: 3000 });
console.log("Listening on port 3000");

for await (const req of s) {
  req.respond({ body: JSON.stringify({ msg: "Hello world ... from Deno!" })
 });
}
```

There are a few things there that might look unfamiliar. First, the import of the **serve** package from a URL instead of a module installed locally—that's one of the key differences between Deno and what you might be used to with Node. We'll talk about module dependencies in the next chapter.

The other is the use of the **`await`** command without any **`async`**. What's going on there? That's another feature of Deno: top-level await.



Note: We'll have more to say about top-level await in Chapter 3.

Let's run it:

Code Listing 16

```
$ deno run HelloDeno.ts
Download https://deno.land/std@0.70.0/http/server.ts
Download https://deno.land/std@0.70.0/encoding/utf8.ts
Download https://deno.land/std@0.70.0/io/bufio.ts
Download https://deno.land/std@0.70.0/_util/assert.ts
Download https://deno.land/std@0.70.0/async/mod.ts
Download https://deno.land/std@0.70.0/http/_io.ts
Download https://deno.land/std@0.70.0/textproto/mod.ts
Download https://deno.land/std@0.70.0/http/http_status.ts
Download https://deno.land/std@0.70.0/async/deferred.ts
Download https://deno.land/std@0.70.0/async/delay.ts
Download https://deno.land/std@0.70.0/async/mux_async_iterator.ts
Download https://deno.land/std@0.70.0/async/pool.ts
Download https://deno.land/std@0.70.0/bytes/mod.ts
Check file:../server.ts
error: Uncaught PermissionDenied: network access to "0.0.0.0:3000", run
again with the --allow-net flag
    at Object.jsonOpSync (core.js:247:13)
    at opListen (deno:cli/rt/30_net.js:32:17)
    at Object.listen (deno:cli/rt/30_net.js:207:17)
    at serve (server.ts:287:25)
```

Deno first downloads all the dependencies that the server package requires, compiles the TypeScript into JavaScript—and then it fails with a **`PermissionDenied`** error. Oh dear.

Remember what we said earlier in this chapter about Deno being "secure by default?" This is what's going on here. Anything you write in TypeScript or JavaScript is considered "unprivileged" code. It runs in a sandbox with no access either to your file system or the network unless you explicitly grant it permission. To get your program to run, you must give it network access with the **`--allow-net`** flag:

Code Listing 17

```
$ deno run --allow-net server.ts
Listening on port 3000
```

If you then visit **`http://localhost:3000`** in your browser, you should see a JSON object containing our message:

Code Listing 18

```
{"msg":"Hello world ... from Deno!"}
```

Congratulations! You have written your first Deno program and on the way, you learned a little bit about the rationale behind Deno. You can find the source code for this chapter here.

In the next chapter, we'll dig into some Deno specifics and try to address the differences between Deno and Node.

Chapter 2 Deno and Node.js

In Chapter 1, we took a brief look at the (short) history of Deno and some of the rationale behind its development. In this chapter, I'd like to do a bit of a deeper dive into how Deno differs from Node.js and how I see the future of both runtimes unfolding.

If you've never developed in Node, don't worry. This chapter will still be useful to you because it will teach you the key concepts that you need to know to be able to write and run your applications. We'll be looking at:

- How Deno handles dependencies using modules.
- What APIs are available to Deno.
- Why your program needs explicit permissions to perform certain tasks, and how to grant those permissions.
- What tools Deno includes out of the box to help you write, test, and run your code.

Dependencies

Let's start off by talking about the difference in how Deno manages modules and dependencies, which is one of the most radical differences between Deno and Node.

When Node was invented in 2009, JavaScript didn't have a standard module system. Incredible as it might seem now, there was no real way to directly reference or include one JavaScript file within another. To do it, developers had to rely either on HTML **`<script>`** tags or "bundlers" like Babel, Grunt, and webpack.

Node wanted modules to make it easy for developers to share code, so Node adopted CommonJS for this purpose—one of several workarounds that the community came up with for JavaScript's lack of modules.

Ultimately, this led to the development of Node's package management system, which enabled developers to discover, use, and publish modules in a centralized repository. The Node Package Manager (**`npm`**) is incredibly well-supported, with over 1.5 million modules at the time of writing, and is arguably one of the key reasons why Node has been so widely adopted.

And Deno does away with it.

"What?" I hear you say. It's true—Deno has no package manager. It adopts an entirely different approach to Node for working with modules that does not require one.

You see, things have changed a lot in the world of JavaScript since 2009. And one of the biggest changes is the addition of ES6 modules. ES6 modules let you import from either an absolute or relative URL:

Code Listing 19

```
import { something } from https://somewhere.com/my_module.js;
```

This relies on the script referenced by that URL using an export function to make the imported functions or other values available externally. For example:

Code Listing 20

```
export function something() {
   // does stuff
}
```

This is the approach Deno uses. On the face of it, this might not seem massively different from how CommonJS works, except that CommonJS modules and their dependencies are loaded on demand from the file system *while* your code is running. ES6 modules are parsed and any dependencies are downloaded *before* your code executes.



Note: Just to be clear, it's also possible to use ES6 modules in Node.js these days. But it's a bit complex because Node must support CommonJS modules at the same time, so it involves using different file extensions to differentiate between the two, and it's still in the experimental stages.

Whenever an **`import`** statement is encountered by a script, Deno downloads the required module, compiles it, and caches it in a global directory that all Deno programs running on that machine can access. The location of this is your system's cache directory:

- On Linux/Redox: **`$XDG_CACHE_HOME/deno`** or **`$HOME/.cache/deno`**
- On Windows: **`%LOCALAPPDATA%/deno`** (**`%LOCALAPPDATA% = FOLDERID_LocalAppData`**)
- On macOS: **`$HOME/Library/Caches/deno`**
- If something fails, it falls back to **`$HOME/.deno`**

This means that only one instance of a module is ever present on your server, even if dozens of programs need it, and that module is available before your code runs.



Tip: If ever you need to force Deno to re-fetch an import, you can do it by executing *`deno cache --reload my_module.ts`.*

Compare this to how Node.js does things. Every Node program has a **`node_modules`** folder, and all your dependencies are installed there. If you've worked with Node much in the past, you will have noticed that **`node_modules`** can quickly get out of hand: every module your program requires has dependencies on specific versions of other modules, and this can lead to **`node_modules`** becoming very large—sometimes several hundred megabytes:

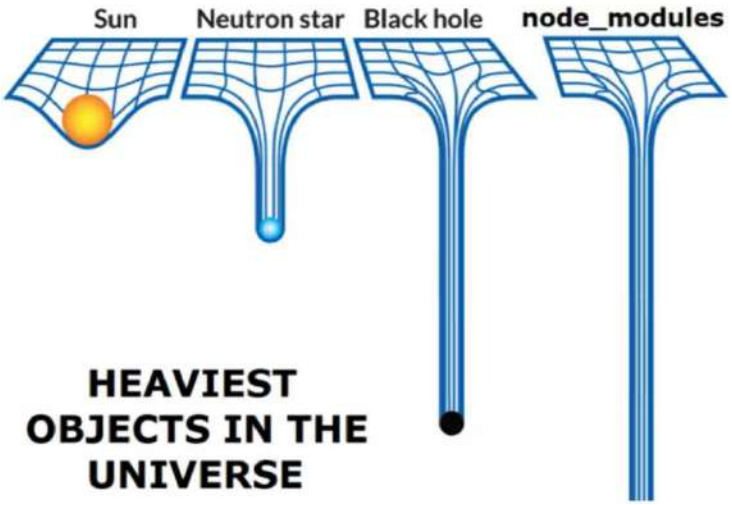


Figure 4: `node_modules` can get rather large. Source: Ryan Dahl's talk, 10 Things I Regret About Node.js, at JSConf EY 2018.

But what happens if a module URL suddenly becomes unavailable? With Node.js, you would normally just check **`node_modules`** into GitHub so that mission-critical applications can continue to run.

Well, you can do that with Deno too. All you need to do is set the **`DENO_DIR`** environment variable to a directory within your project. For example:

OS X/Linux:

Code Listing 21

```
DENO_DIR= ~/myproject/deno_modules
```

Windows cmd shell:

Code Listing 22

```
set DENO_DIR="C:\myproject#deno_modules"
```

Windows PowerShell:

Code Listing 23

```
$env:DENO_DIR="C:\myproject\deno_modules"
```

Deno will cache modules in that directory when your program executes, and you can check it into your version control system.

How do I find modules?

So, no package manager?

Yes, that's right. No package manager. There are advantages and disadvantages to that, in my opinion. Certainly, it's very flexible: you can create packages and share them via any URL you like without having to publish them to a centralized repository. But discovering new packages won't be as easy without one.

The closest thing Deno has to a centralized repository for packages is at https://deno.land. Packages are stored in two discrete areas:

- **The Deno Standard Library**: This is a set of standard modules that are audited by the core team and guaranteed to work with Deno. It includes modules like **`http`** (a lightweight server for files over HTTP) and **`fs`** (for working with the file system).
- **Third-Party Modules**: This a hosting service for Deno scripts. It caches releases of open-source modules stored on GitHub and serves them via a single, easy-to-remember domain. Many of the modules here are direct ports of popular Node.js modules, like **`lodash`** (a utility library for working with JavaScript and widely adopted) and **`oak`** (an implementation of Node's Express web framework that we will be looking at in a later chapter).

We'll be using several modules from both sources in the practical examples in subsequent chapters.

How are modules versioned?

At the time of writing this, Deno itself is versioned differently from the modules in the standard library, which is not yet considered stable. However, a new version of the standard library is released every time a new version of Deno is released.

The Deno core team recommends that you refer to specific versions of modules rather than the master branch to avoid your application breaking if there are changes to its dependencies.

So, in production, avoid this:

Code Listing 24

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

Instead, refer to a specific version (in this example, version 0.71.0):

Code Listing 25

```
import { copy } from "https://deno.land/std@0.71.0/fs/copy.ts";
```

Deno's third-party modules in https://deno.land/x work in the same way.

Security

As we have already established, Deno is secure by default. Everything runs in a sandbox without permissions to the network or file system unless you explicitly grant them. Compare this to Node, where your programs have access to everything.

To grant your Deno program access to the network, execute it with the **`--allow-net`** argument, like this:

Code Listing 26

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

To enable access to the network and reading local files, grant those permissions using two arguments, like this:

Code Listing 27

```
deno run --allow-net --allow-read myprogram.ts
```

The other permissions available are shown in Table 1.

Table 1: Deno run permissions

Permission	Enables
`-A, --allow-all`	All permissions. The equivalent to disabling all security.
`--allow-env`	Getting and setting environment variables.
`--allow-hrtime`	High-resolution time measurement (to help with timing attacks and fingerprinting).
`--allow-net`	Network access. Optionally accepts a comma-separated whitelist of domains.
`--allow-plugin`	Loading plugins (unstable at the time of writing).
`--allow-read`	File system write access.

Permission	Enables
`--allow-run`	Executing subprocesses.
`--allow-write`	File system read access.

Deno tools

Deno ships with a bunch of command-line tools to make your life as a developer easier. Doubtless you will have used tools like these in your Node development efforts too, but most of them would have been from third parties.

Deno gives you just about everything you need to be productive right out of the box, including the following:

- **Bundler**: Bundles the script you specify and its dependencies into a single file.
- **Formatter**: Beautifies your TypeScript and JavaScript code in a consistent way, which is especially useful for teams and stops the “tabs versus spaces” arguments that a lot of teams are wont to have.
- **Linter**: A code linter that helps you catch errors and inconsistencies in your code.
- **Dependency Inspector**: Specify a module, and it shows you a dependency tree. Super helpful.
- **Debugger**: Debug your Deno programs with Chrome DevTools, VS Code, and other tools (Node also provides a debugger).
- **Documentation Generator**: If you provide JSDoc annotations in your code files, this will parse them and produce documentation.
- **Test Runner**: Run tests on your code. Uses the assertions module in the standard library.

You can see all the tools available by executing:

Code Listing 28

```
deno help
```

Use **`deno help <tool>`** to learn how to use a specific tool. For example, the formatter:

Code Listing 29

```
$ deno help fmt
deno-fmt
Auto-format JavaScript/TypeScript source code.
  deno fmt
  deno fmt myfile1.ts myfile2.ts
```

```
  deno fmt --check

Format stdin and write to stdout:
  cat file.ts | deno fmt -

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

Ignore formatting a file by adding an ignore comment at the top of the
file:
  // deno-fmt-ignore-file

USAGE:
    deno fmt [OPTIONS] [files]...

OPTIONS:
        --check                    Check if the source files are formatted
    -h, --help                     Prints help information
        --ignore=<ignore>          Ignore formatting particular source
files. Use with --unstable
    -L, --log-level <log-level>    Set log level [possible values: debug,
info]
    -q, --quiet                    Suppress diagnostic output
        --unstable                 Enable unstable features and APIs

ARGS:
    <files>...
```

Deno: the end of Node?

I'll try and introduce more features of Deno as we use them in subsequent chapters, but I wanted to highlight some of the key differences between it and Node so that you have some idea of what to expect.

Really though, if you know Node, you're going to appreciate and hopefully one day, fall in love with Deno. Even if you don't know Node, then learning Deno could well be a valuable investment of your time as it continues to mature and gain adoption.

But if Deno is so promising, does it mean the end of Node? It's a valid question. They're both coming from the same place and trying to solve similar problems. But I think that's extremely unlikely, at least for the foreseeable future. Node is *incredibly* popular. It's mature, it's robust, and what it does, it does very well. Deno, as exciting as it is, is still largely untried in production.

Another thing that suggests that Node could be around for a long time to come is the Node Package Manager (`npm`). Ironic, since Deno does without the one thing that really helped Node gain such a widespread following in the first place. There are well over a million packages in the `npm` registry, and that's a huge draw for developers who are looking for pre-built solutions to common problems so that they are not constantly reinventing the wheel. If Deno is looking to compete with that, then first, it's got a long way to go. And second, there's the issue of discoverability. Sure, it's great that you can create modules and publish them anywhere, but how are developers going to know those modules exist without a Deno equivalent of `npm`? It will be interesting to see how that plays out.

In general, though, there are more similarities than differences between Deno and Node. As we start to write Deno code in the remaining chapters, try to keep the following differences between Node and Deno in mind:

- You'll be writing code in TypeScript rather than JavaScript. (You can write JavaScript if you prefer, but that's not really the "Deno way.")
- You'll be importing modules (standard library and third party) via a URL rather than downloading and installing them locally with a package manager.
- You have built-in tools at your disposal to help you develop your application.
- When you execute your application, you'll need to ensure that it has the appropriate permissions, or it won't run.

Enough background—let's build some cool applications and learn by doing!

Chapter 3 Create a URL Shortener Console Application

In this chapter, we'll build a URL shortener using Deno and turn it into a self-contained console application called **surl** that you can run in your terminal. It should allow you to enter a URL and return both the original URL and the new, shortened version:

Code Listing 30

```
$ surl https://deno.land/manual/tools/dependency_inspector
SUCCESS: Your long URL: https://deno.land/manual/tools/dependency_inspector
... is now shorter: https://cleanuri.com/M06pJb
```

Our application will use the **cleanuri** API to do the actual shortening, and Deno tools to install it as a single executable. We'll also provide a logging facility, so that you can see a record of the original URLs and their shortened versions.



Note: We'll use this example as a demonstration of some of Deno's features. But, for the most part, if you're familiar with JavaScript (and especially with TypeScript), I suspect that you'll feel right at home here. If TypeScript is entirely new to you, then I recommend that you read* TypeScript Succinctly*, which you can download for free here.

Creating your project

In your favorite code editor or IDE, create a folder called **surl**. Within that folder, create two files: a **mod.ts** file for your code, and a **urls.txt** file with which to log your original URLs and their shortened counterparts.

Why **mod.ts** and not **index.ts** or similar? If you've come from the world of Node, then you'll know that the **index.js** file is special in that it specifies the default entry point for your application. This is especially important when you **require** Node modules.

When you pass a directory to Node's **require** function, then it has some work to do to figure out how to run the module that you have specified. First, it checks **package.json** for an endpoint. If that isn't defined, it then looks for the presence of **index.js**. If it doesn't find an **index.js** file, it makes a last-ditch attempt to find **index.node** (a C++ extension format).

You can see that in the absence of a **package.json** entry, the most likely entry point for your code is **index.js**—hence the importance of that name in Node.

Ryan Dahl has said that placing such great importance on the **`index.js`** file as the default entry point for a module's code is a major regret. It added a lot of complexity to the module loading system. For this reason, Deno doesn't care what you call your main code file. In fact, you must explicitly name this file as part of your **`import`** statement. For example, to use the **`csv`** module in the standard API:

Code Listing 31

```
import { readCSVObjects } from "https://deno.land/x/csv/mod.ts";
```

Having said that though, developers like to standardize wherever possible, so you'll find that a lot of Deno code uses **`mod.ts`** as the entry point—and we'll do the same in this book. This is not always the case, however, especially for third-party modules, so please bear that in mind.



Note: You can listen to Ryan Dahl lamenting his choices behind `index.js` in his talk, 10 Things I Regret About Node.js.

Accepting command-line arguments

For our tool to be useful, we need to provide a way for the user to enter the URL they want to shorten as a command-line argument.

For this, as in so many other common use cases, Deno has us covered. We can use the **`Deno.args`** variable in the runtime API to see which arguments are passed to our command-line program.

But wait a minute. What's that **`Deno`** namespace? To answer that question, let's have a look at how the runtime API is structured.

The Deno runtime consists of web APIs and the **`Deno`** global namespace. Deno wants you to use familiar APIs wherever possible, so where a web standard exists (such as **`fetch`** for HTTP requests), Deno supports these rather than forcing you to use a new proprietary API.

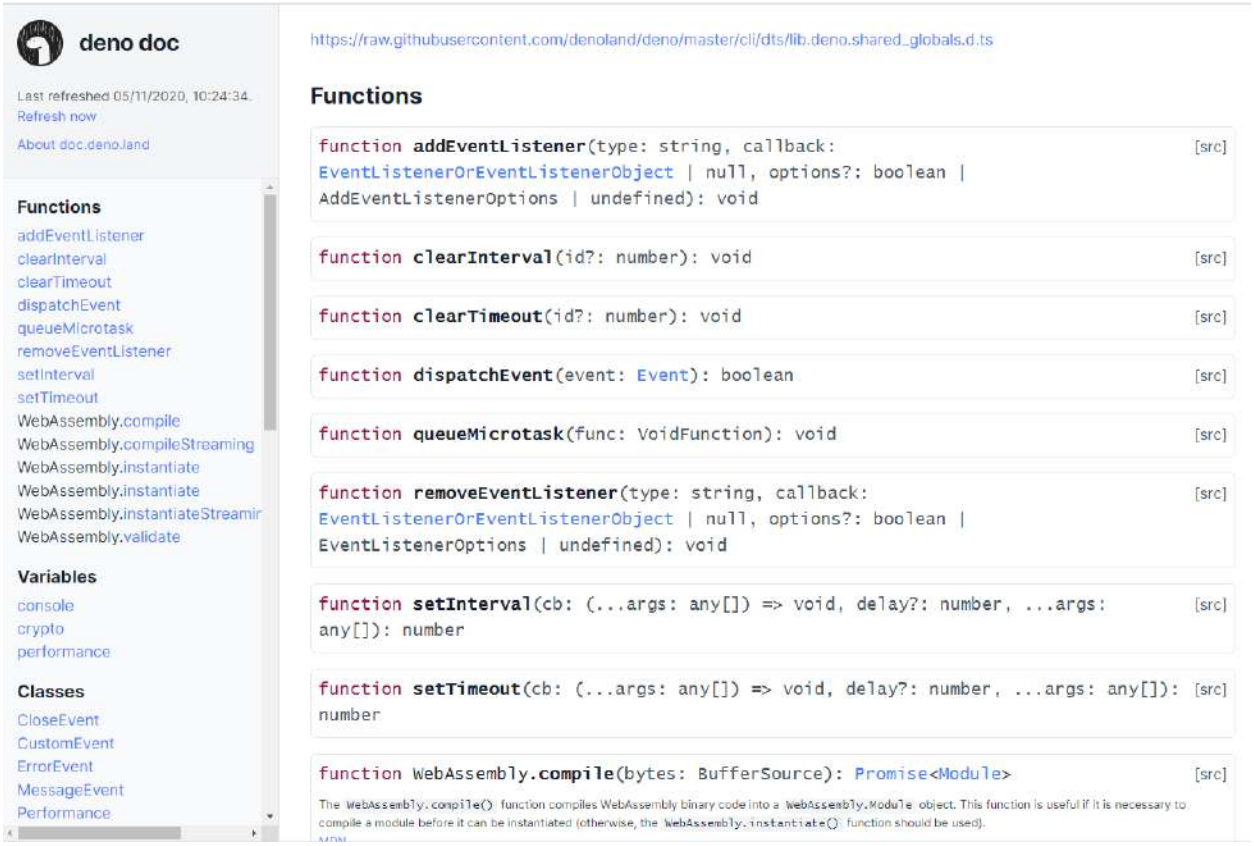


Figure 5: Deno's web APIs

All the APIs that Deno supports that are *not* web standard are contained in the **Deno** global namespace. These include APIs for reading from files, opening TCP sockets, and many operations that you, as a developer, are likely to use often.

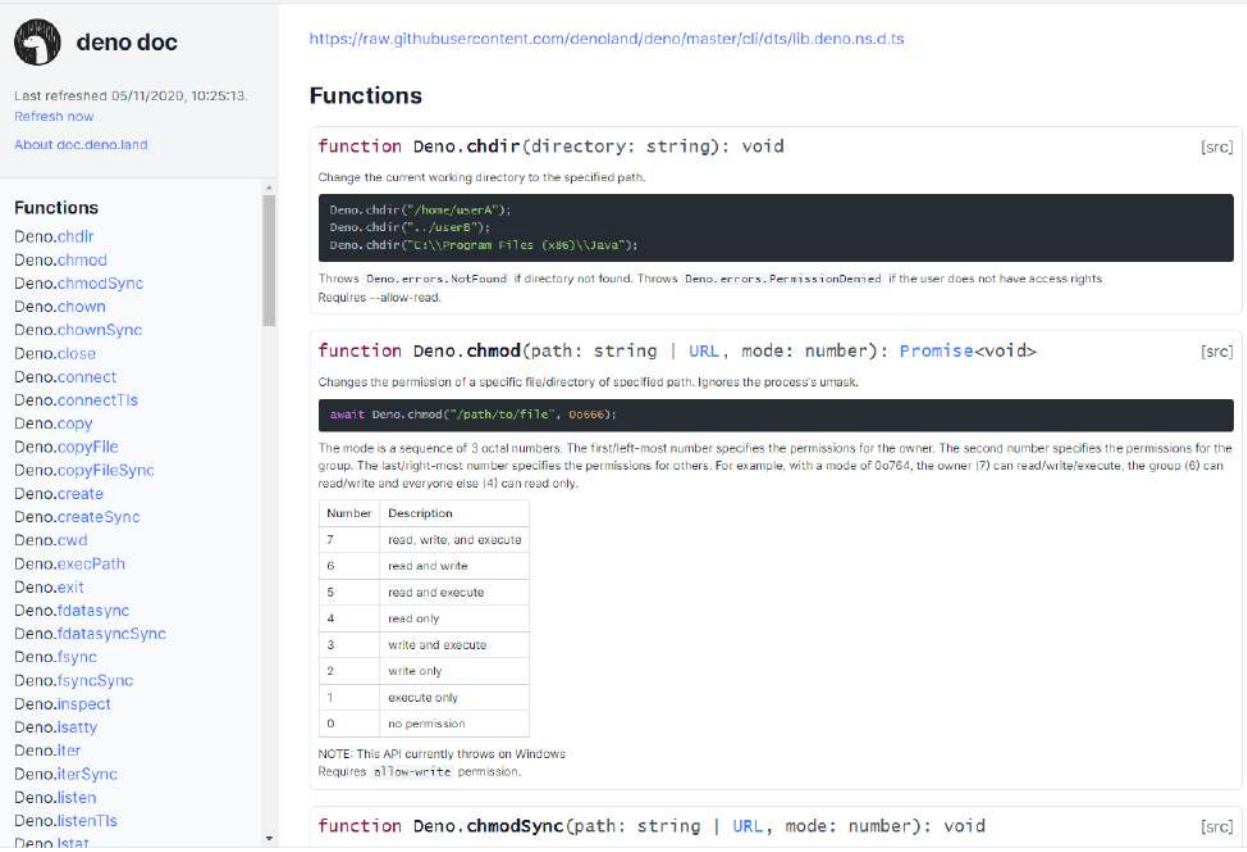


Figure 6: The Deno global namespace

So, back to accepting command-line arguments. The Deno global namespace has a variable called **`Deno.args`**, which accepts an array of strings, one for each argument passed to your program. We can test it out as shown in the following code.

In **`mod.ts`**, include this single line of code:

Code Listing 32

```
console.log(Deno.args);
```

Run it with **`deno run mod.ts firstArgument secondArgument`**. It should return an array containing the two arguments passed to it:

Code Listing 33

```
$ deno run mod.ts firstArgument secondArgument
Check file:///.../surl/mod.ts
[ "firstArgument", "secondArgument" ]
```

Validating the URL

Now we know how to accept a URL argument to our shortener. The next thing we need to worry about is whether the URL supplied is a valid one. We need a regular expression for that. Thankfully (because of my rather lacking regex skills) a suitable expression is easy to find on the web:

Code Listing 34

```
^(https?://)?(www\\.)?([-a-z0-9]{1,63}\\.)*?[a-z0-9][-a-z0-9]{0,61}[a-z0-9]\\.[a-z]{2,6}(/[-\\w@\\+\\.~#\\?&/=%]*)?$
```

We just need to wrap it up in a function and pass our submitted URL to it:

Code Listing 35

```
const { args: [url] } = Deno;

function isUrlValid(input: string) {
  const regex =
    "^(https?://)?(www\\.)?([-a-z0-9]{1,63}\\.)*?[a-z0-9][-a-z0-9]{0,61}[a-z0-9]\\.[a-z]{2,6}(/[-\\w@\\+\\.~#\\?&/=%]*)?$";
  var url = new RegExp(regex, "i");
  return url.test(input);
}

function shorten(url: string) {
  if (url === "" || url === undefined) {
    console.error("No URL provided");
    return;
  }

  if (!isUrlValid(url)) {
    console.error(`${url} is invalid`);
  } else {
    console.log(`${url} is a valid URL`);
  }
}

shorten(url);
```



Tip: The string enclosed by backticks in the `console.log` and `console.error` statements is an ES6 feature that enables string interpolation.

If we run this code, we should be able to enter any URL and have our program determine whether it is valid or not:

Code Listing 36

```
$ deno run mod.ts https:www.bbc.co.uk
https:www.bbc.co.uk is invalid

$ deno run mod.ts https://www.bbc.co.uk
https://www.bbc.co.uk is a valid URL

$ deno run mod.ts
No URL provided
```

There's nothing particularly Deno-esque going on there. In fact, the only reference to anything Deno is the **`Deno.args`** global environment variable that we're using to accept a URL as input.

In this example, however, we are using the *destructuring assignment* syntax in ECMAScript 6 to access **`Deno.args`**. This syntax enables you to unpack values from arrays, or properties from objects, into distinct variables. Here, we're only interested in the first element in the **`Deno.args`** array, which we call **`url`**, so we unpack that accordingly:

Code Listing 37

```
const { args: [url] } = Deno;
```

If we include extra arguments when we execute the program, they are ignored:

Code Listing 38

```
$ deno run mod.ts https://www.bbc.co.uk https://google.com
https://slashdot.org
https://www.bbc.co.uk is a valid URL
```

Shortening the URL

Now that we can pass a URL into our program, we want to submit it to a third-party API to shorten it for us. I favor **`cleanuri`** for this, at least for demonstration purposes, because it doesn't require any signup, API key, or secret to use.



Note: The cleanuri API docs are here.

Let's refactor the **`shorten()`** method in our code to submit any valid URL our program receives to cleanuri.com for shortening:

Code Listing 39

```
async function shorten(url: string): Promise<{ result_url: string }> {
  if (url === "" || url === undefined) {
    throw { error: "No URL provided" };
  }

  if (!isUrlValid(url)) {
    throw { error: "The URL provided is invalid" };
  }

  const options = {
    method: "POST",
    headers: {
      "Content-Type": "application/json",
    },
    body: JSON.stringify({ url }),
  };
  const res = await fetch("https://cleanuri.com/api/v1/shorten", options);
  const data = await res.json();
  return data;
}

try {
  const { result_url } = await shorten(url);
  console.log(`SUCCESS: Your long URL: ${url}`);
  console.log(`... is now shorter: ${result_url} `);
} catch (error) {
  console.log(`ERROR: ${error}`);
}
```

If we run it, everything should work nicely:

Code Listing 40

```
$ deno run --allow-net mod.ts "https://www.bbc.co.uk"
SUCCESS: Your long URL: https://www.bbc.co.uk
... is now shorter: https://cleanuri.com/x60ZOd
```



Tip: Now that the program wants to access an external API over the internet, you must grant it network access. Make sure that you include the `--allow-net` flag, or it won't work!

As you can see, we're using a standard **`fetch`** to make a **`POST`** request to **`cleanuri`** with our JSON-formatted URL as the payload. Because we don't want the rest of our code to execute before we've got the results back from the API call, we use **`await`** and declare the surrounding **`shorten`** function as **`async`**, and its return type as **`Promise`**.

We then call **`surl`** asynchronously too, with another **`await`**. But hold on a moment! We know that the async/await paradigm in JavaScript requires that any function that uses an **`await`** should be marked as **`async`**. How have we gotten away with it here?

This is another cool feature of Deno: *top-level await*, which we encountered briefly back in Chapter 1.

Top-level await

At the time of writing, this is something that even Node cannot do, even though support for it has been added to the V8 engine. There are workarounds which require Node developers to use immediately invoked functions tagged as **`async`**, but this is messy.

With Deno, the **`async`** declaration in your top-level code is implicit, which leads to much tidier code overall. It means that you can perform asynchronous tasks such as connect to a database, make an API request, and so on from your startup code while your application loads—without having to wrap those operations in an **`async`** function just to get them to work. It also makes it easy to check for missing dependencies and fall back gracefully:

Code Listing 41

```
let myModule = null;
try {
    myModule = await import('https://url1');
} catch (exception) {
    myModule = await import('https://url2');
}
```

Logging

There's just one thing left to do: start logging the URLs submitted to **`surl`** and their shortened counterparts. We just want to write these to a local text file. That's a common requirement, so our first port of call should be to the Deno runtime API documentation to see if there's anything there that can help us.

You'll see that there are a number of different functions for reading from and writing to files. **`Deno.writeTextFile()`** looks perfect for our needs:



Figure 7: The Deno.writeTextFile() function

All we need to do is invoke that function at the end of the **`try`** clause in our **`try`**/**`catch`** block, passing in the name of the file to write to (**`urls.txt`**: put it in your **`HOME`** directory to make it easy to locate), and the original (**`url`**) and shortened (**`result_url`**) URLs:

Code Listing 42

```
try {
  const { result_url } = await  shorten(url);
  console.log(`SUCCESS: Your long URL: ${url}`);
  console.log(`... is now shorter: ${result_url} `);

  await Deno.writeTextFile(
    "~/urls.txt",
    `${url} -> ${result_url}\n`,
    { append: true },
  );
} catch (error) {
  console.log(`ERROR: ${error}`);
}
```

Now, when we run the application, we should see those URLs logged to **`urls.txt`**:

Code Listing 43

```
$ deno run --allow-net mod.ts "https://www.slashdot.org"
Check file:///.../surl/mod.ts
SUCCESS: Your long URL: https://www.slashdot.org
... is now shorter: https://cleanuri.com/5mkmpd
```

```
ERROR: PermissionDenied: write access to "urls.txt", run again with the --
allow-write flag
```

Oops! Now that we're asking our Deno program to write to the file system, we need to give it the appropriate permission to do so:

Code Listing 44

```
$ deno run --allow-net --allow-write mod.ts "https://www.slashdot.org"
SUCCESS: Your long URL: https://www.slashdot.org
... is now shorter: https://cleanuri.com/5mkmpd
```

And then we can examine the log entry in **`urls.txt`**:

Code Listing 45

```
https://www.slashdot.org -> https://cleanuri.com/5mkmpd
```

Deploying our console application

Now that we've created a useful tool, let's package it up so that we can share it. Nobody wants to execute **`deno run --allow-net --allow-write mod.ts "https://www.example.com"`** every time they want to shorten a URL.

To do this, we'll use the **`deno install`** command to create a script that will run our code with the correct permissions.

There are a couple of different ways of doing this, depending on where you want to install your script, and what name you want to give it.

In our example, we can just run the following from within our code directory:

Code Listing 46

```
$ deno install --allow-net --allow-write mod.ts
```

This will create a script called **`surl`** (based on the directory where **`mod.ts`** is located) in the **`.deno/bin`** folder, in your computer's home directory. If you examine the contents of that file, you will see the following:

Code Listing 47

```
% generated by deno install %
@deno.exe "run" "--allow-write" "--allow-net" "file:///<path>/surl/mod.ts"
%*
```

In this instance, **`deno install`** looked at the name of the TypeScript file, saw it was just a generic **`mod.ts`** file, and decided to use the name of the parent directory instead (**`surl`**).

If you then add that script to your **`PATH`**, you can run it anywhere on your machine as follows:

Code Listing 48

```
$ surl https://www.amazon.com
```

> 💡 ***Tip:** `deno install` **uses the name of the parent directory for the script if the file you referred to is called either** `main`**,** `mod`**,** `index`**, or** `cli`**. By default, it installs to ~/.deno/bin. You can read the full documentation for** `deno install` **here.***

You can also override the default script name by using the **`--name`** (or **`-n`**) flag:

Code Listing 49

```
$ deno install --allow-net --allow-write -n myexec mod.ts
```

Summary

Congratulations! You built a useful little command-line tool in Deno. Along the way you learned about:

- Top-level **`await`**.
- Writing to a CSV file.
- Distributing executable code with **`deno install`**.

You can find the complete source code here.

Chapter 4 Code a Static Site Generator

Hopefully, you're feeling all fired up after building your first real Deno application! Let's continue our exploration of Deno by building something a little more complex.

In this chapter, we're going to build a bare-bones static site generator for blogging with. If you're not familiar with the term, static site generators are all the rage nowadays. They're something of a reaction against bloated content management systems in general (and WordPress in particular) that rely upon storing website content in a database and rendering it on the fly when users visit a page. With a static site generator, all the content is served as plain HTML. When you update the site, the site generator recreates the HTML with your latest additions and changes. The result is a site that's super fast and easy to migrate from one host to another.

Our simple static site generator will enable us to write our blog posts in Markdown, and then convert them into individual HTML files. Along with the Markdown, we'll also be able to provide metadata about each post, including things like the title, date, author, and so on. We'll then create a home page that links to each of those posts.

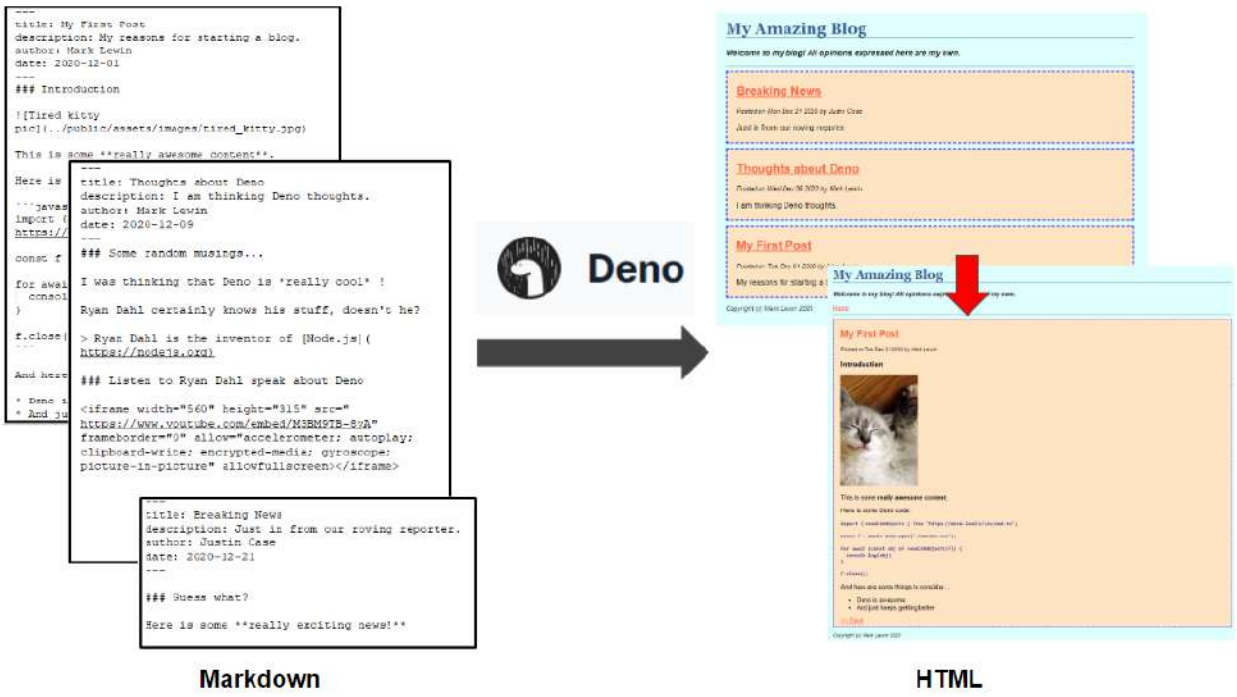


Figure 8: Static site generator

On our journey, we'll learn about some interesting and useful Deno modules. Let's get started!

Configuring the application

To be useful to an end user, our application needs to be configurable. In our "alpha" version, the only things we'll enable the user to configure are:

- The name of the blog.
- A description of the blog.
- Copyright information.
- The location of the source Markdown files.
- The location of the destination HTML files.

But because my goal for this exercise is to create something you could use as the basis of something much more sophisticated, we'll make it easy to add more configuration options in the future.

A common practice for configuring applications is to include the configuration options in a "dot env" file, **`.env`**. Because of this, you might reasonably expect that there's a Deno module somewhere that can read this type of file and do something sensible with the contents. And you'd be right! It's called **`dotenv`**, and it's an almost identical port from the Node module of the same name. **`dotenv`** is a zero-dependency module that loads environment variables from a **`.env`** file—either into Deno's process environment, or a named object variable of your choice.

Let's build a project and get the configuration bit working first. Create a directory called **`static-site-gen`** with the following contents:

- A directory called **`content`** that contains our Markdown content ready for publishing as HTML.
- A directory called **`public`** that contains the HTML files generated from the Markdown files in the **`content`** directory.
    - A directory under **`public`** called **`assets`**:
        - Two subfolders within **`assets`**: **`images`** (for images) and **`styles`** (for the site CSS).
- A directory called **`views`** to store the template files we will use to render HTML.
- A **`.env`** file for configuration settings (the GitHub repository for this project contains a file called **`example.env`** you can adapt).
- A **`mod.ts`** file to contain our application code.

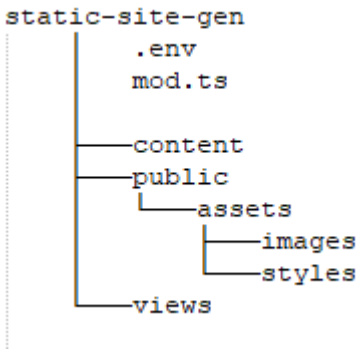


Figure 9: The static-site-gen project directory

Then, configure **.env** with the **POSTS_DIR** and **PUBLISH_DIR** settings as shown, but with whatever else you want to include to personalize your blog:

Code Listing 50

```
BLOG=My Amazing Blog
SUMMARY=Welcome to my blog! All opinions expressed here are my own.
COPYRIGHT=Copyright (c) Mark Lewin 2021
POSTS_DIR=./content/
PUBLISH_DIR=./public/
```

In **mod.ts**, write the following code:

Code Listing 51

```
import "https://deno.land/x/dotenv/load.ts";

console.log(Deno.env.get("BLOG"));
console.log(Deno.env.get("COPYRIGHT"));
console.log(Deno.env.get("SUMMARY"));
console.log(Deno.env.get("POSTS_DIR"));
console.log(Deno.env.get("PUBLISH_DIR"));
```

Run it, ensuring that you enable the **--allow-read** and **--allow-env** options:

Code Listing 52

```
$ deno run --allow-read --allow-env mod.ts
...
My Amazing Blog
Copyright (c) Mark Lewin 2021
Welcome to my blog! All opinions expressed here are my own.
./content/
./public/
```

The settings in **.env** are read into Deno's environment variables (**Deno.env**) and displayed in the console. Note how we use the **Deno.env.get()** method to read values from Deno's environment variables.

You might be wondering what other environment variables Deno is using. Let's find out! We can assign **Deno.env** to a variable and log its contents to the console:

Code Listing 53

```
import "https://deno.land/x/dotenv/load.ts";

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

```
console.log(env);
```

There are too many to list here, but note how our settings are included in there (in no particular order):

Code Listing 54

```
{
  CommonProgramW6432: "C:\\Program Files\\Common Files",
  SYSTEMROOT: "C:\\WINDOWS",
  USERPROFILE: "C:\\Users\\mplew",
  COMPUTERNAME: "DESKTOP-JETHKMM",
  SSH_ASKPASS: "C:/Program Files/Git/mingw64/libexec/git-core/git-gui--
askpass",
  EXEPATH: "C:\\Program Files\\Git",
  PUBLISH_DIR: "./public/",
  "CommonProgramFiles(x86)": "C:\\Program Files (x86)\\Common Files",
  PROCESSOR_ARCHITECTURE: "AMD64",
  _: "C:/Users/mplew/.deno/bin/deno",
  MSYSTEM: "MINGW64",
  NUMBER_OF_PROCESSORS: "4",
  POSTS_DIR: "./content/",
  ...
  LANG: "en_GB.UTF-8"
}
```



Tip: The `Deno.env` object also has `set()` and `delete()` methods that let you change the values of those environment variables. Although you probably won't use those methods regularly, they can be useful for debugging.

There is another way to use **dotenv**, which, to my mind, is a little cleaner than trying to access Deno's environment variables, and that is to import the configuration using the **config()** function:

Code Listing 55

```
import { config } from "https://deno.land/x/dotenv/mod.ts";

console.log(config());
```



Note: With this approach we need to import the `config()` function from the `mod.ts` file, instead of importing `load.ts`, which we used for autoloading `.env`.

When we run that, the **`config()`** function returns an object representing the contents of the **`.env`** file:

Code Listing 56

```
$ deno run --allow-read --allow-env mod.ts
Check file:///../static-site-gen/mod.ts
{
  BLOG: "My Amazing Blog",
  SUMMARY: "Welcome to my blog! All opinions expressed here are my own.",
  COPYRIGHT: "Copyright (c) Mark Lewin 2021",
  POSTS_DIR: "./content/",
  PUBLISH_DIR: "./public/"
}
```

To make our code cleaner, we can use object destructuring to refer to those configuration variables in our code directly by name instead of as object properties:

Code Listing 57

```
import { config } from "https://deno.land/x/dotenv/mod.ts";

const { BLOG, SUMMARY, COPYRIGHT, POSTS_DIR, PUBLISH_DIR } = config();

console.log(`BLOG: ${BLOG}`);
console.log(`SUMMARY: ${SUMMARY}`);
console.log(`COPYRIGHT: ${COPYRIGHT}`);
console.log(`POSTS_DIR: ${POSTS_DIR}`);
console.log(`PUBLISH_DIR: ${PUBLISH_DIR}`);
```

When you run it, you see that those settings are now directly available to you:

Code Listing 58

```
$ deno run --allow-read mod.ts
Check file:///../static-site-gen/mod.ts
BLOG: My Amazing Blog
SUMMARY: Welcome to my blog! All opinions expressed here are my own.
COPYRIGHT: Copyright (c) Mark Lewin 2021
POSTS_DIR: ./content/
PUBLISH_DIR: ./public/
```

Note: Another benefit of not loading these settings into `Deno.env` is that we can avoid having to specify the `--Load-env` flag when running the program.

Creating the input files

As I said earlier, we want to be able to write our site content in Markdown and have our program convert it into HTML. We also want to be able to include some metadata in each file that will include a title, description, and date for each post that will be ignored by the HTML renderer.

Many other static site generators use YAML (Yet Another Markup Language) for this metadata, and we'll do the same. Here are the three sample posts I will be using in this chapter:

Code Listing 59

post1.md

````
---
title: My First Post
description: My reasons for starting a blog.
author: Mark Lewin
date: 2020-12-01
---
### Introduction

![Tired kitty pic](../public/assets/images/tired_kitty.jpg)

This is some **really awesome content**.

Here is some Deno code:

```javascript
import { readCSVObjects } from "https://deno.land/x/csv/mod.ts";

const f = await Deno.open("./movies.csv");

for await (const obj of readCSVObjects(f)) {
  console.log(obj);
}

f.close();
```

And here are some things to consider....

* Deno is awesome
* And just keeps getting better
````

In this first post, you can see that I'm including an image file called **tired_kitty.jpg**. In your **PUBLISH_DIR** (**public** in this example) you should have a folder called **assets**, and a subfolder within **assets** called **images**. Place any images that you want to include in your posts in there.

In my second post, I'm going to embed a YouTube video of one of Ryan Dahl's talks:

Code Listing 60

post2.md

```
---
title: Thoughts about Deno
description: I am thinking Deno thoughts.
author: Mark Lewin
date: 2020-12-09
---
### Some random musings...

I was thinking that Deno is *really cool* !

Ryan Dahl certainly knows his stuff, doesn't he?

> Ryan Dahl is the inventor of [Node.js](https://nodejs.org)

### Listen to Ryan Dahl speak about Deno

<iframe width="560" height="315" src="https://www.youtube.com/embed/M3BM9TB
-8yA" frameborder="0" allow="accelerometer; autoplay; clipboard-
write; encrypted-media; gyroscope; picture-in-
picture" allowfullscreen></iframe>
```

And I'll include a third post, just so that we have a good selection of posts to show off:

Code Listing 61

post3.md

```
---
title: Breaking News
description: Just in from our roving reporter.
author: Justin Case
date: 2020-12-21
---

### Guess what?

Here is some **really exciting news!**
```

Reading the input files

Now that we have some Markdown content to render, the first thing we need to figure out is how to read in those files from the **POSTS_DIR** and write them to the **PUBLISH_DIR**. Reading and writing files is a common use case, and as we have already seen from our URL shortener example, there are functions in the Deno runtime API that can help us.

Let's start by writing some code that will simply read the files from **POSTS_DIR** and write them back out, unchanged, to **PUBLISH_DIR**. We'll worry about converting from Markdown to HTML in a later step.

Code Listing 62

```
import { config } from "https://deno.land/x/dotenv/mod.ts";

const { BLOG, SUMMARY, COPYRIGHT, POSTS_DIR, PUBLISH_DIR } = config();

async function generatePosts() {
  const decoder = new TextDecoder("utf-8");

  // Read in the files from the POSTS_DIR
  for await (const file of Deno.readDir(POSTS_DIR)) {
    const markdown = decoder.decode(await Deno.readFile(POSTS_DIR + file.na
me));
    console.log(`Read ${POSTS_DIR}${file.name}`);

    // Write the file to the PUBLISH_DIR
    Deno.writeTextFile(PUBLISH_DIR + file.name, markdown);
    console.log(`Wrote ${PUBLISH_DIR}${file.name}`);
  }
}

await generatePosts();
```

Here we have defined a function called **generatePosts()** that uses the Deno core function **Deno.readDir()** to read each file from **POSTS_DIR** (using **Deno.readFile()**) and write it to **PUBLISH_DIR** (using **Deno.writeTextFile()**). The **Deno.readFile()** function reads in the file as an array of bytes, so we need to use a **TextDecoder** to decode it.

If we run it and examine the contents of **PUBLISH_DIR**, we should see our unaltered Markdown files in there:

Code Listing 63

```
$ deno run --allow-read --allow-write mod.ts
Check file:///../static-site-gen/mod.ts
Read ./content/post1.md
Wrote ./public/post1.md
Read ./content/post2.md
```

```
Wrote ./public/post2.md
Read ./content/post3.md
Wrote ./public/post3.md

$ ls ./public/
assets/  post1.md  post2.md  post3.md
```

Parsing the post metadata

Great, but so far, we're not doing anything with our Markdown files—we're just copying them from one location to another. We want to convert them to HTML.

To do this, we're going to use a third-party Deno module called, appropriately enough, **`markdown`**. It is a port of a popular Node package called **`marked`**.

The trouble is that these files aren't just Markdown: they also have some metadata (often called "front matter") that we need to deal with. As a first step towards doing something useful with this metadata, we will parse it to get the title of the post, and use this as the basis of the file name for the rendered HTML.

Luckily, **`markdown`** can do this for us!

Let's import the **`markdown`** module and use its **`parse()`** method on the contents of the input Markdown files and see what it returns:

Code Listing 64

```
import { config } from "https://deno.land/x/dotenv/mod.ts";
import { Marked } from "https://deno.land/x/markdown/mod.ts";

const { BLOG, SUMMARY, COPYRIGHT, POSTS_DIR, PUBLISH_DIR } = config();

async function generatePosts() {
  const decoder = new TextDecoder("utf-8");

  // Read in the files from the POSTS_DIR
  for await (const file of Deno.readDir(POSTS_DIR)) {
    const markdown = decoder.decode(await Deno.readFile(POSTS_DIR + file.na
me));
    const markup = Marked.parse(markdown);

    console.log(`Read ${POSTS_DIR}${file.name}`);
    console.log(markup);

    // Write the file to the PUBLISH_DIR
    Deno.writeTextFile(PUBLISH_DIR + file.name, markdown);
    console.log(`Wrote ${PUBLISH_DIR}${file.name}`);
```

```
  }
}

await generatePosts();
```

When we run this, we can see that it gives us an object with two properties: **content**, which is the rendered HTML, and **meta**, which contains our front matter in a nested object:

Code Listing 65

```
$ deno run --allow-read --allow-write mod.ts
Check file:///../static-site-gen/mod.ts
Read ./content/post1.md
{
  content: '<h3 id="introduction">Introduction</h3>\n<p><img
src="../public/assets/images/tired_kitty.jpg" alt="T...',
  meta: {
    title: "My First Post",
    description: "My reasons for starting a blog.",
    author: "Mark Lewin",
    date: 2020-12-01T00:00:00.000Z
  }
}
Wrote ./public/post1.md
...
```

Let's write that rendered HTML in **markup.content** to an HTML file named after the title of the blog post (using the **meta.title** property). All we'll do is turn the title into lowercase and replace any spaces with hyphens.

For example: "My First Post" will become **my-first-post.html**.

Code Listing 66

```
import { config } from "https://deno.land/x/dotenv/mod.ts";
import { Marked } from "https://deno.land/x/markdown/mod.ts";

const { BLOG, SUMMARY, COPYRIGHT, POSTS_DIR, PUBLISH_DIR } = config();

async function generatePosts() {
  const decoder = new TextDecoder("utf-8");

  // Read in the files from the POSTS_DIR
  for await (const file of Deno.readDir(POSTS_DIR)) {
    const markdown = decoder.decode(await Deno.readFile(POSTS_DIR + file.na
me));
    const markup = Marked.parse(markdown);
```

```
    console.log(`Read ${POSTS_DIR}${file.name}`);

    // Determine the file name from the metadata
    const newPostFileName = markup.meta.title.toLowerCase().replace(/ /g, "
-") +
      ".html";

    // Write the file to the PUBLISH_DIR
    Deno.writeTextFile(PUBLISH_DIR + newPostFileName, markup.content);
    console.log(`Wrote ${PUBLISH_DIR}${newPostFileName}`);
  }
}

await generatePosts();
```

Let's run it!

Code Listing 67

```
$ deno run --allow-read --allow-write mod.ts
Check file:///../static-site-gen/mod.ts
Read ./content/post1.md
Wrote ./public/my-first-post.html
Read ./content/post2.md
Wrote ./public/thoughts-about-deno.html
Read ./content/post3.md
Wrote ./public/breaking-news.html
```

Check the contents of each **.html** file and ensure that they look like HTML. If they do, open them in your browser.

Here is **my-first-post.html**, which was created from **post1.md**:

Code Listing 68

```
<h3 id="introduction">Introduction</h3>
<p><img src="../public/assets/images/tired_kitty.jpg" alt="Tired kitty pic"
></p>
<p>This is some <strong>really awesome content</strong>.</p>
<p>Here is some Deno code:</p>

<pre><code class="lang-
javascript">import { readCSVObjects } from &amp;quot;https://deno.land/x/csv/mo
d.ts&amp;quot;;

const f = await Deno.open(&amp;quot;./movies.csv&amp;quot;);

for await (const obj of readCSVObjects(f)) {
```

```
  console.log(obj);
}

f.close();
</code></pre>
<p>And here are some things to consider....</p>

<ul>
<li>Deno is awesome</li>
<li>And just keeps getting better</li>
</ul>
```

And when I open it in a browser:

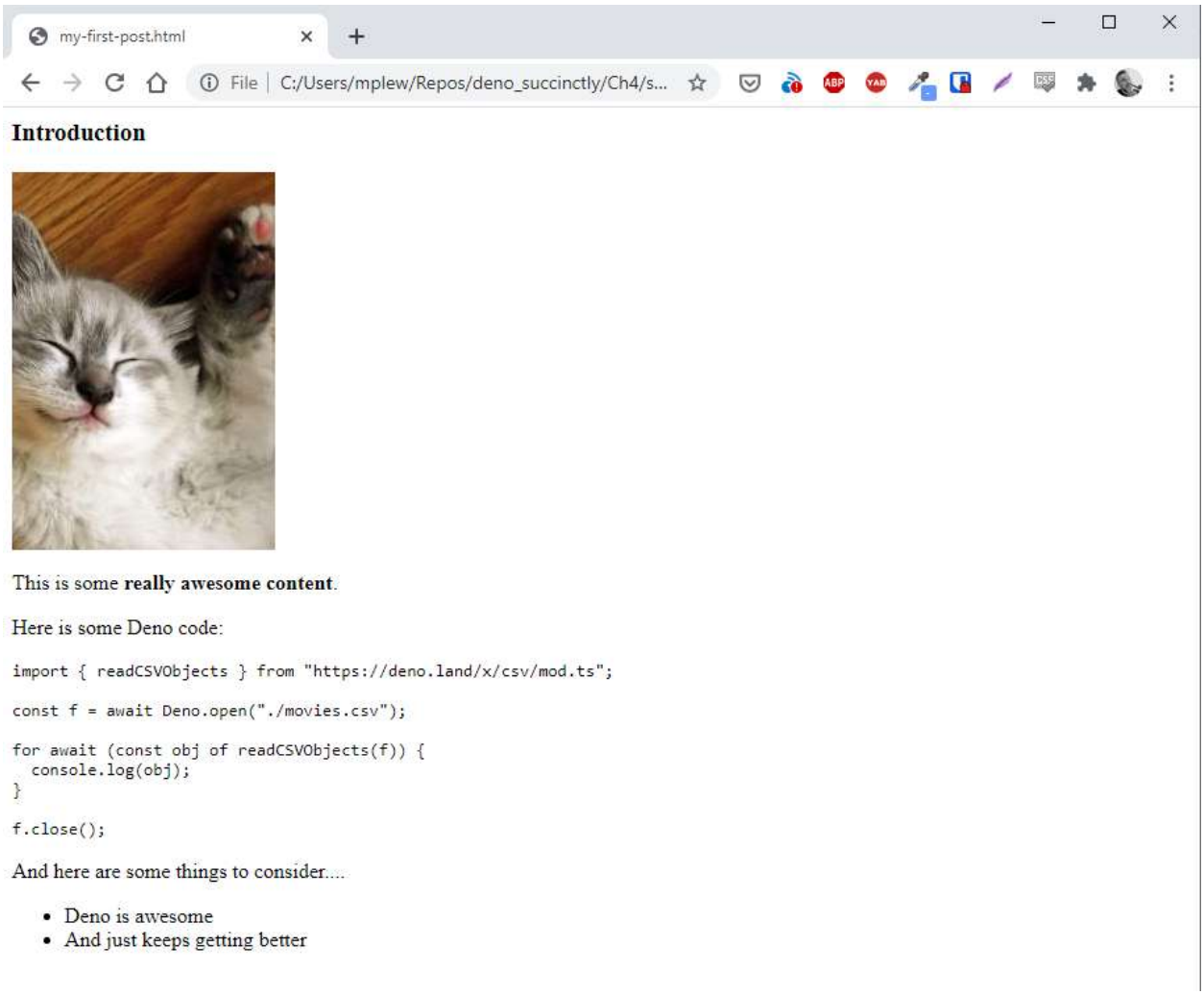


Figure 10: The rendered HTML page

You can go ahead and delete any **`.md`** files in **`PUBLISH_DIR`**—you're now writing real HTML!

We now have a way to convert our Markdown to HTML. But the results aren't pretty—yet.

Using templates

If you look at the HTML that's being rendered, you'll notice that it's just an HTML fragment and not a full webpage. We should do something about that. Also, it would be nice if we could assign classes to some of the HTML tags so that we can style them using CSS. We could do that in the Markdown, but it's not ideal: we would have to repeat the styling information in every blog post. It would also remove the main benefit of writing our blog posts in Markdown: a nice, simple format that anyone can learn without much effort.

By far the best way to do this is in our application. And the easiest way to achieve it is by using a template engine. A template engine enables you to use static template files in your application. At runtime, the template engine replaces variables in a template file with actual values and transforms the template into an HTML file. It's a great way of keeping your application logic separate from the presentation, which makes applications much easier to maintain.

There are many such engines: Handlebars, Mustache, EJS, and Pug are all extremely popular. Deno has a fair selection too, including ports of these, but we're going to use one called Eta (which apparently means "tiny" in Esperanto). Eta claims to be a "lightweight and blazing-fast embedded JS templating engine that works inside Node, Deno, and the browser. Created by the developers of Squirrelly, it's written in TypeScript and emphasizes phenomenal performance, configurability, and low bundle size."

The essence of Eta (and other template engines) is that it enables you to programmatically apply data to a template, and then render it as HTML:

Code Listing 69

```
<!-- views/template.eta -->
My name is <%= it.name %>

<!-- include other template files as "partials" -->
<%~ includeFile('./footer') %>

<!-- views/footer.eta -->
<footer>This is the footer!</footer>


// mod.ts

import { renderFile, configure } from "https://deno.land/x/eta@v1.11.0/mod.
ts"

const viewPath = `${Deno.cwd()}/views/`

configure({
```

```
  // This tells Eta to look for templates
  // In the /views directory
  views: viewPath
})

// Eta assumes the .eta extension if you don't specify an extension
// You could also write renderFile("template.eta"),
// renderFile("/template"), etc.

let templateResult = await renderFile("./template", { name: "Mark" })

console.log(templateResult)
/*
My name is Mark
<footer>This is the footer!</footer>
*/
```

Eta also supports conditionals (which we won't be using) and loops (which we will use a bit later when we create our home page).



Tip: You can find out more about Eta here.

Let's create a template, which we will apply to each of our blog posts as part of the rendering process. Create the **`post.eta`** file in the **`views`** folder:

Code Listing 70

```
<!DOCTYPE html>
<html lang="en">
   <head>
      <meta charset="UTF-8" />
      <meta name="viewport" content="width=device-width, initial-
scale=1.0" />
      <link rel="stylesheet" href="assets/styles/style.css">
      <title><%= it.title | it.blog %></title>
   </head>
   <body>
      <header>
         <h1><%= it.blog %></h1>
         <p class="slug"><%= it.summary %></p>
         <a href="index.html">Home</a>
         <hr />
      </header>
      <article>
         <h2><%= it.title %></h2>
         <p class="smalltext">Posted on <%= it.date.toDateString() %> by <%
= it.author %></p>
```

```
        <div><%~ it.content %></div>
        <a href="index.html">&lt;&lt; Back</a>
      </article>
      <footer>
        <p class="smalltext"><%= it.copyright %></p>
      </footer>
    </body>
</html>
```

Everything prefixed by **it** in the template is a variable placeholder that we must supply to the template in our program code. They are enclosed by **<%= it.[variable] %>** or **<%~ it.[variable] %>** tags, which have different behaviors:

- **<%= %>** are *interpolation tags*. You just place the value you want to output between them, and they are automatically escaped if necessary.
- **<%~ %>** are known as *raw interpolation tags*. They output unescaped values, which is useful when injecting HTML into the template.

Having created our post template, let's consider how to populate it. We do that by calling **renderFile()** and passing the template file we want to render (the **.eta** extension is optional) and an object containing the variables that the template requires:

Code Listing 71

```
let templateResult = await renderFile("./template", { blog: "My Amazing
Blog" });
```

Let's import the **eta** module and write some code to first tell Deno where to find our template (in the **views** folder), and then populate it with the values we retrieved from **Marked.parse()**:

Code Listing 72

```
import { config } from "https://deno.land/x/dotenv/mod.ts";
import { Marked } from "https://deno.land/x/markdown/mod.ts";
import { configure, renderFile } from "https://deno.land/x/eta/mod.ts";

const { BLOG, SUMMARY, COPYRIGHT, POSTS_DIR, PUBLISH_DIR } = config();

const viewPath = `${Deno.cwd()}/views/`;

// Configure ETA
configure({
  views: viewPath,
});

async function generatePosts() {
  const decoder = new TextDecoder("utf-8");
```

```
  // Read in the files from the POSTS_DIR
  for await (const file of Deno.readDir(POSTS_DIR)) {
    const markdown = decoder.decode(await Deno.readFile(POSTS_DIR + file.na
me));
    const markup = Marked.parse(markdown);

    console.log(`Read ${POSTS_DIR}${file.name}`);

    // Determine the file name from the metadata
    const newPostFileName = markup.meta.title.toLowerCase().replace(/ /g, "
-") +
      ".html";

    let templateResult = await renderFile(
      "./post",
      {
        blog: BLOG,
        title: markup.meta.title,
        date: markup.meta.date,
        author: markup.meta.author,
        summary: SUMMARY,
        copyright: COPYRIGHT,
        content: markup.content,
      },
    );

    // Write the file to the PUBLISH_DIR
    await Deno.writeTextFile(
      PUBLISH_DIR + newPostFileName,
      templateResult,
    );

    console.log(`Wrote ${PUBLISH_DIR}${newPostFileName}`);
  }
}

await generatePosts();
```

Note that at the time of writing, not all Deno features are production-ready, and Deno protects us from those features by default. This is the case for the **eta** module, so you'll need to run it using the **--unstable** command-line flag:

Code Listing 73

```
$ deno run --allow-read --allow-write --unstable mod.ts
Check file:///../static-site-gen/mod.ts
Read ./content/post1.md
Wrote ./public/my-first-post.html
```

```
Read ./content/post2.md
Wrote ./public/thoughts-about-deno.html
Read ./content/post3.md
Wrote ./public/breaking-news.html
```

When you examine the generated HTML files, you should see that they are now full webpages with some classes applied to certain elements that we can use to style them. We'll get to that in a bit!

Code Listing 74

```
<!DOCTYPE html>
<html lang="en">
   <head>
      <meta charset="UTF-8" />
      <meta name="viewport" content="width=device-width, initial-
scale=1.0" />
      <link rel="stylesheet" href="assets/styles/style.css">
      <title>0</title>
   </head>
   <body>
      <header>
         <h1>My Amazing Blog</h1>
         <p class="slug">Welcome to my blog! All opinions expressed here ar
e my own.</p>
         <a href="index.html">Home</a>
         <hr />
      </header>
      <article>
         <h2>Breaking News</h2>
         <p class="smalltext">Posted on Mon Dec 21 2020 by Justin Case</p>
         <div><h3 id="guess-what-">Guess what?</h3>
<p>Here is some <strong>really exciting news!</strong></p>
</div>
         <a href="index.html">&lt;&lt; Back</a>
      </article>
      <footer>
         <p class="smalltext">Copyright (c) Mark Lewin 2021</p>
      </footer>
   </body>
</html>
```

Creating the home page

So now we have several individual HTML files, each containing a blog post. We now want to create a central home page where we can link to each post, ordering the posts so that the most recent one is listed first.

We'll use another template for the home page. Create **index.eta** in the **views** directory and populate it as shown:

Code Listing 75

```
<!DOCTYPE html>
<html lang="en">
   <head>
      <meta charset="UTF-8" />
      <meta name="viewport" content="width=device-width, initial-
scale=1.0" />
      <link rel="stylesheet" href="assets/styles/style.css">
      <title><%= it.blog %></title>
   </head>
   <body>
      <header>
         <h1><%= it.blog %></h1>
         <p class="slug"><%= it.summary %></p>
         <hr />
      </header>
      <div>
         <% it.posts.forEach((post)=>{ %>
         <div class="post">
            <h2><a href="<%= post.url %>"><%= post.title %></a></h2>
            <p class="smalltext">Posted on <%= post.date.toDateString() %>
by <%= post.author %></p>
            <p><%= post.description %></p>
         </div>
         <% }) %>
      </div>
      <footer>
         <p class="smalltext"><%= it.copyright %></p>
      </footer>
   </body>
</html>
```

You can see that in addition to the settings in **.env**, which apply to the whole site, we also want to populate variables with metadata from the individual posts. We also need to know the URL of each blog post so that we can link to it from the home page. To achieve this, we're going to have to accumulate that information as we process each post.

Create a type called **PostDetails** with the metadata that we want to store for each post, and a collection called **posts** to store them in:

Code Listing 76

```
import { config } from "https://deno.land/x/dotenv/mod.ts";
import { Marked } from "https://deno.land/x/markdown/mod.ts";
```

```
import { configure, renderFile } from "https://deno.land/x/eta/mod.ts";

const { BLOG, SUMMARY, COPYRIGHT, POSTS_DIR, PUBLISH_DIR } = config();

const viewPath = `${Deno.cwd()}/views/`;

// Configure ETA
configure({
  views: viewPath,
});

// Store the metadata for individual posts
type PostDetails = {
  title: string;
  description: string;
  date: Date;
  url: string;
};

// Details of all posts
let posts: PostDetails[] = [];

async function generatePosts() {
  const decoder = new TextDecoder("utf-8");

  // Read in the files from the POSTS_DIR
  for await (const file of Deno.readDir(POSTS_DIR)) {
    const markdown = decoder.decode(await Deno.readFile(POSTS_DIR + file.na
me));
    const markup = Marked.parse(markdown);

    console.log(`Read ${POSTS_DIR}${file.name}`);

    // Determine the file name from the metadata
    const newPostFileName = markup.meta.title.toLowerCase().replace(/ /g, "
-") +
      ".html";

    const post = {
      title: markup.meta.title,
      description: markup.meta.description,
      author: markup.meta.author,
      date: markup.meta.date,
      url: `./${newPostFileName}`,
    };

    let templateResult = await renderFile(
      "./post",
      {
```

```
        blog: BLOG,
        title: post.title,
        date: post.date,
        author: post.author,
        summary: SUMMARY,
        copyright: COPYRIGHT,
        content: markup.content,
      },
    );
    // Write the file to the PUBLISH_DIR
    await Deno.writeTextFile(
      PUBLISH_DIR + newPostFileName,
      templateResult,
    );

    console.log(`Wrote ${PUBLISH_DIR}${newPostFileName}`);

    posts.push(post);
  }

  console.log(posts);
}

await generatePosts();
```

Run your program and ensure that the posts collection is being populated correctly:

Code Listing 77

```
$ deno run --allow-read --allow-write --unstable mod.ts
Check file:///C:/Users/mplew/Repos/deno_succinctly/Ch4/static-site-
gen/mod.ts
Read ./content/post1.md
Wrote ./public/my-first-post.html
Read ./content/post2.md
Wrote ./public/thoughts-about-deno.html
Read ./content/post3.md
Wrote ./public/breaking-news.html
[
  {
    title: "My First Post",
    description: "My reasons for starting a blog.",
    author: "Mark Lewin",
    date: 2020-12-01T00:00:00.000Z,
    url: "./my-first-post.html"
  },
  {
    title: "Thoughts about Deno",
```

```
    description: "I am thinking Deno thoughts.",
    author: "Mark Lewin",
    date: 2020-12-09T00:00:00.000Z,
    url: "./thoughts-about-deno.html"
  },
  {
    title: "Breaking News",
    description: "Just in from our roving reporter.",
    author: "Justin Case",
    date: 2020-12-21T00:00:00.000Z,
    url: "./breaking-news.html"
  }
]
```

Now that we have all the information we require to render the home page, we need to write the code to do it. We'll create a separate function for this called **`generateHomepage()`**. In that function we will sort the posts by date (most recent first), render the template, and then write it to an **`index.html`** file in our **`PUBLISH_DIR`**:

Code Listing 78

```
async function generateHomepage() {
  // Sort posts by date: most recent posts first
  posts.sort((a, b) => {
    return (b.date.getTime() - a.date.getTime());
  });

  let templateResult = await renderFile(
    "./index",
    {
      blog: BLOG,
      summary: SUMMARY,
      copyright: COPYRIGHT,
      posts: posts,
    },
  );

  // Write the file to the PUBLISH_DIR
  await Deno.writeTextFile(
    PUBLISH_DIR + "index.html",
    templateResult,
  );
}

await generatePosts();
generateHomepage();
```

Running this should produce an **`index.html`** file that contains links to the individual blog posts:

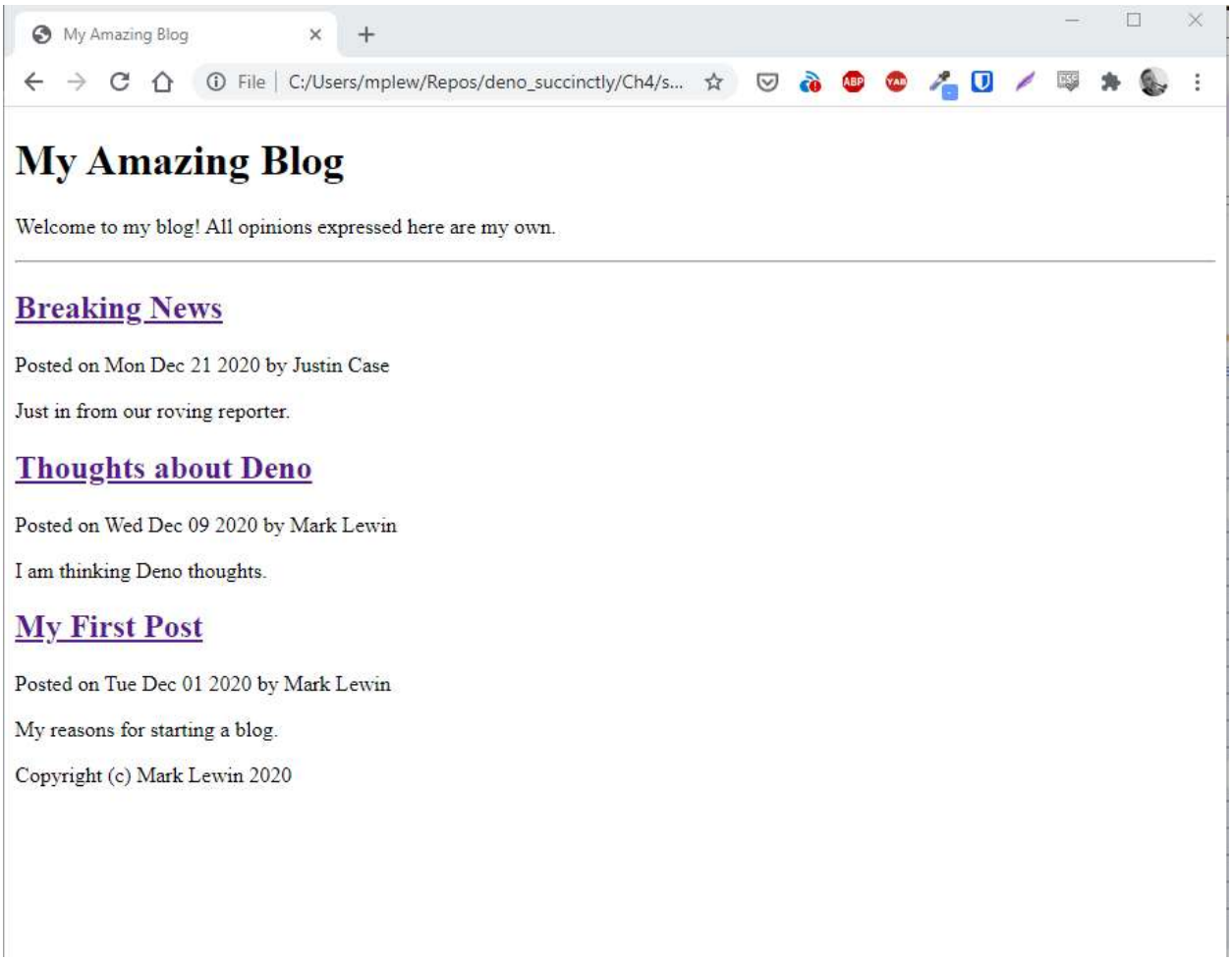


Figure 11: The site home page

The only thing left to do now is to style our site. In the **`public/assets/styles`** directory, create a file called **`style.css`** with the following stylesheet:

Code Listing 79

```
body {
    font-family: Helvetica, Arial, sans-serif;
    width:50%;
    margin: 0 auto;
    background-color: lightcyan;
}

h1 {
    color: #375e97;
    font-family: Georgia, 'Times New Roman', Times, serif;
    border-bottom: 1px solid #375e97;
```

```
}

h2 {
    color: #fb6542;
}

.slug {
    color: #0c0707;
    font-weight: bold;
    font-style: italic;
}

code {
   color: darkblue;
   font-weight: bolder;
}

.post {
    border: 2px dashed blue;
    padding: 5px 20px;
    margin-bottom: 10px;
    background-color: bisque;
}

article {
    border: 2px dashed blue;
    padding: 5px 20px;
    background-color: bisque;
}

blockquote {
    background-color: lemonchiffon;
    border: 2px solid #fb6542;;
    padding: 5px;
    margin-left: 0%;
}

.smalltext {
    font-style: italic;
    font-size: small;
}

a:link, a:visited {
    color: #fb6542;
}

a:hover {
    text-decoration: none;
}
```

Now we have the beginnings of a blog worthy of the name!

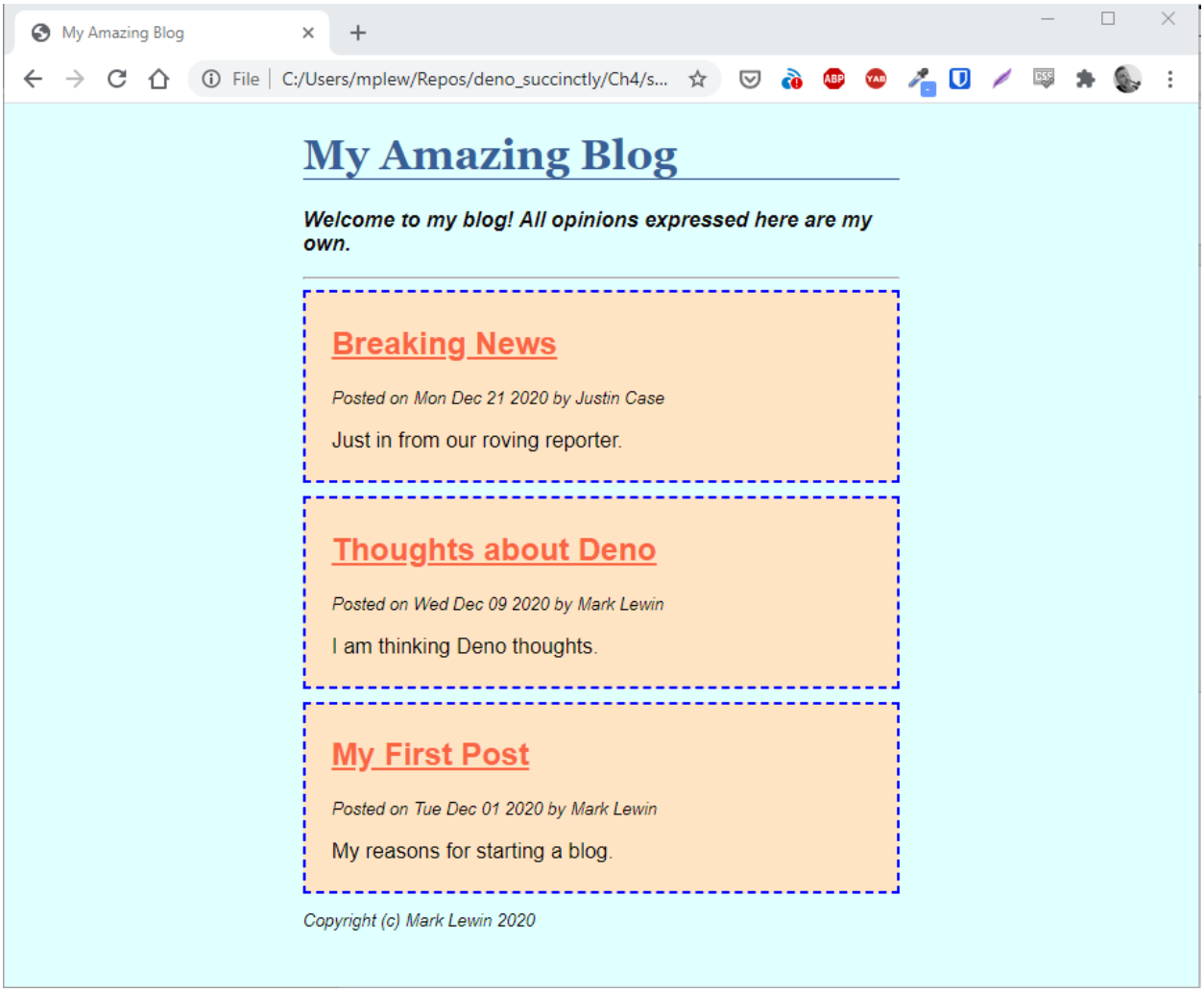


Figure 12: The styled home page

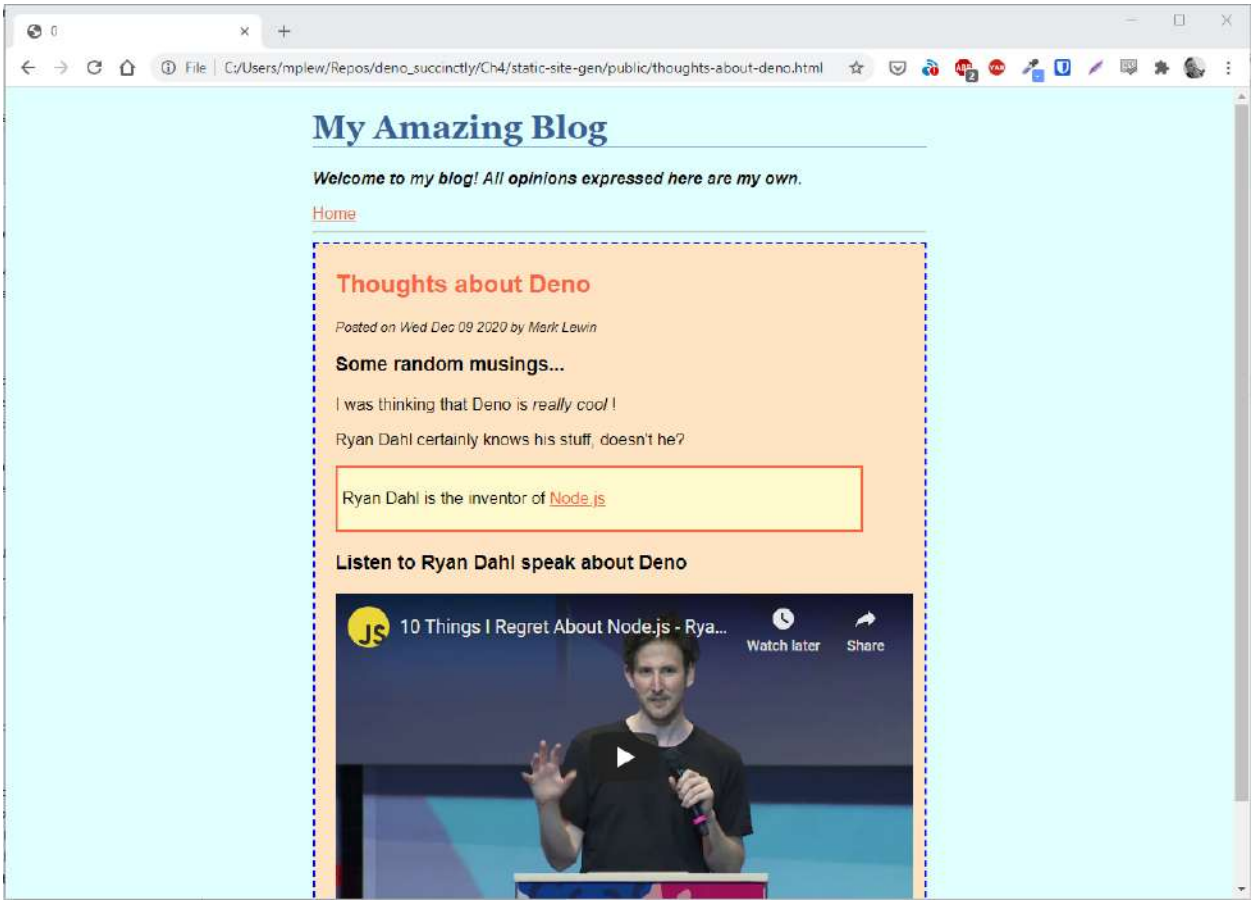


Figure 13: A styled post

Summary

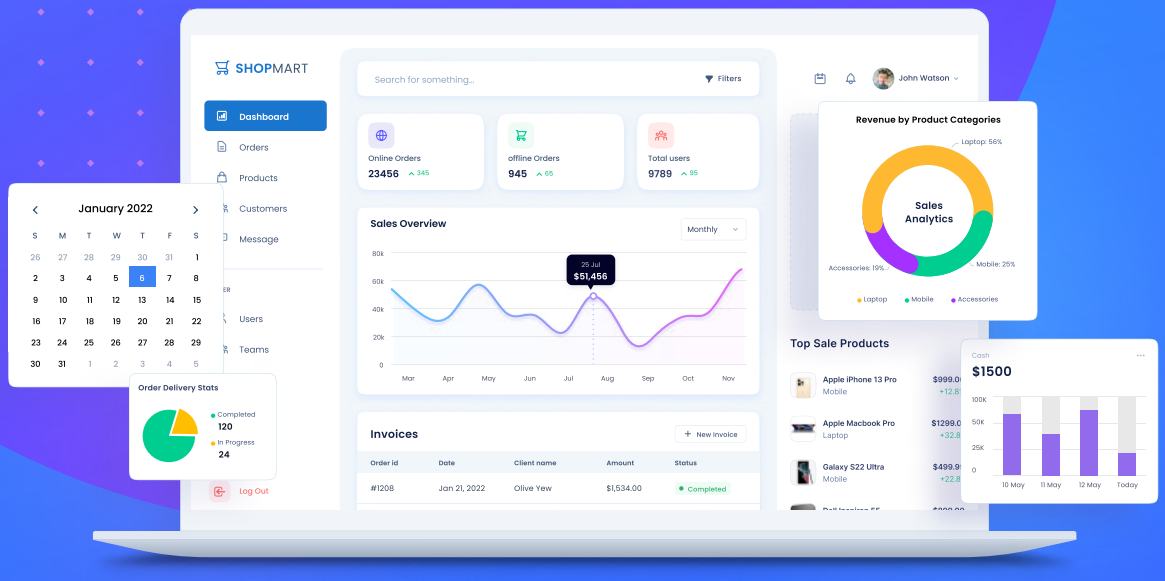
In this chapter you built a bare-bones static site generator. It's very usable but rather limited in features, so hopefully you'll feel inclined to expand on it! Other than restyling the thing (design was never my strong suit), you could consider adding extra pages (such as an About and Contact page), implementing post categories, having a featured post, or just about anything else you've seen in other static site generators like Jekyll or Hugo.

As well as continuing to practice writing and running Deno programs, you also learned about three particularly useful modules: **`env`** for configuration, and **`markdown`** and **`eta`** for page templating.

You can find the source code for this project here.

THE WORLD'S BEST
UI COMPONENT SUITE
FOR BUILDING
POWERFUL APPS

G2 4.6 out of 5 stars



GET YOUR FREE .NET AND JAVASCRIPT UI COMPONENTS

syncfusion.com/communitylicense

- 1,700+ components for mobile, web, and desktop platforms
- Support within 24 hours on all business days
- Uncompromising quality
- Hassle-free licensing
- 28000+ customers
- 20+ years in business

Trusted by the world's leading companies

Apple IBM SIEMENS intel VISA

Syncfusion

Chapter 5 Build a RESTful API

One of the primary reasons for Node's swift rise to prominence in the last decade is its suitability for developing APIs. As more than one IT journalist noted during that time, if Marc Andreessen of Netscape had been commenting on the software landscape in recent years, then his famous observation that "software is eating the world" would have been revised to "APIs are eating the world."

Indeed, APIs are everywhere. And any language or framework that can make it easier for developers to create good APIs is always going to be a hit. Node delivered on this very well, with its non-blocking architecture allowing it to serve many concurrent requests at great speed and scale nicely. Because Deno has inherited all the good things about Node (and hopefully discarded some of the sketchier bits), you can rest assured that Deno is (or at least will be) a very solid choice for API development.

Deno API frameworks

Technically, you could probably create your API without using a framework. However, the role of a framework is to harness the core language's ability to do things like interact with the file system and accept network requests while abstracting away some of the complexity involved in coding them. It can also extend the core language features with additional capabilities.

In the Node world, the most popular framework for creating APIs is undoubtedly Express. Express calls itself a "fast, un-opinionated, minimalist framework for Node." This framework doesn't offer a lot of enhancements, giving you the bare minimum of features and focusing mainly on performance. However, it does enable you to create extensive middleware chains that intercept requests before they reach your handlers to perform additional processing. We'll talk more about middleware later.

Developed by the same team behind Express, Koa attempts to do things better in much the same way as Deno in an attempt to "fix" Node. It takes advantage of some of the new ES6 language features that weren't around when Express was conceived to ditch callbacks in favor of function generators and async/await and to generally create a much nicer programming experience.

In Deno, Koa has become Oak (see what they did there?), and is probably the most popular framework for developing APIs right now. For that reason, it's the one that we'll use to build our API in this chapter.

However, there is another contender that is gaining popularity, and that is Drash. Unlike most of the frameworks available for Deno, it's not just a direct port of a Node framework. Instead, Drash wants to do things differently and, in my opinion, it's worth keeping an eye on.

Tip: You can find out more about Drash here.

One of the nicest things about Drash is that you can use it to create either APIs or web applications, and it gives you the appropriate generator script to scaffold your project:

Code Listing 80

```
$ deno run --allow-read --allow-write --allow-run --allow-net
https://deno.land/x/drash/create_app.ts --api
```

This creates the following structure for your project:

```
\---drash-app
    |   app.ts
    |   config.ts
    |   deps.ts
    |   server.ts
    |
    +---middleware
    +---resources
    |       home_resource.ts
    |
    \---tests
        \---resources
                home_resource_test.ts
```

Figure 14: The Drash project skeleton

Drash creates boilerplate code to help you get started. It encourages you to use object-oriented techniques to create your application. For instance, by looking at the autogenerated **`home_resource.ts`** in the **`resources`** folder, you can see how you can extend **`Drash.Http.Resource`** to start building out your API's endpoints:

Code Listing 81

```
import { Drash } from "../deps.ts";

export default class HomeResource extends Drash.Http.Resource {
  static paths = ["/"];

  public GET() {
    this.response.body = { message: "GET request received!" };
    return this.response;
  }

  public POST() {
    this.response.body = { message: "POST method not implemented." };
```

```
    return this.response;
  }

  public DELETE() {
    this.response.body = { message: "DELETE method not implemented." };
    return this.response;
  }

  public PUT() {
    this.response.body = { message: "PUT method not implemented." };
    return this.response;
  }
}
```

It's a nice little framework and well worth looking into.

The Dinosaur API

In case you haven't noticed already, Deno is full of dinosaurs. As well as the project logo, many of its third-party modules are named after dinosaur themes.

Let's play along by creating a Dinosaur API. This will be a simple CRUD (create, read, update, and delete) API that will enable us to view and manage a collection of dinosaur information by making HTTP calls to REST endpoints.

To build this, we're going to use the following:

- Oak: The Deno version of Koa, to create our routes and handlers.
- MongoDB: To store our collection of dinosaurs.
- **`mongo`**: A database driver for MongoDB.
- **`dotenv`**: Our old friend from Chapter 4, which will keep our database credentials private in a **`.env`** file.

We'll start off by building the endpoints and getting them to talk to our database. Then we'll talk about middleware and write some that will log all API requests and provide error handling for missing endpoints.

Let's go!

Creating the server

We will start by creating the server that will accept incoming requests and create an initial handler for the root endpoint.

Create a new directory called **dinosaur-api** and a file within it called **server.ts**. In **server.ts**, write the following code:

Code Listing 82

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

const app = new Application();
const router = new Router();

app.use(router.routes());
app.use(router.allowedMethods());

router
  .get("/", (ctx) => {
    ctx.response.body = { message: "Welcome to the Dinosaur API" };
    ctx.response.status = 200;
  });

app.listen({ port: 3000 });
console.log("Server is running....");
```

The **Application** class in Oak is a wrapper around Deno's built-in **http** package. It has two methods: **.use()** and **.listen()**.

The **.use()** method adds what's known as a *middleware function*. Middleware functions have access to the request object (from an incoming HTTP request) and the response object (the response that your server sends to that request). Middleware functions are usually chained together, and any middleware function can also access any subsequent middleware functions in the chain.

We'll create our own middleware functions later in this chapter and learn a bit more about them then, but for now, just appreciate that they can intercept the request and response objects to add extra functionality to your server.

In this code, we're adding two pieces of middleware using functions on the **Router** object. The first (**router.routes()**) returns middleware that will do all the route (or endpoint) processing that we have defined. The second (**router.allowedMethods()**) returns middleware that instructs the server to only allow HTTP requests using the methods (**GET**, **POST**, etc.) that we have defined for each route.

We then define those routes. Here we have a handler for a **GET** request on our root endpoint (that is, our server host without any path denoting a specific resource). It is passed a **RouterContext** object called **ctx**, which contains the **request** and **response** objects. Our route handler is merely returning a **response** object to any client that requests this route, which consists of the status code for the request (**200 OK**) and some JSON with a welcome message.

Having set up the middleware and defined the route, we then instruct our server to start listening for incoming requests using the **app.listen()** method.

Run the program:

Code Listing 83

```
$ deno run --allow-net server.ts
Check file:///C:/Users/mplew/Repos/deno_succinctly/dinosaur-api/server.ts
Server is running....
```

And visit http://localhost:3000 in your browser. You should see the following:

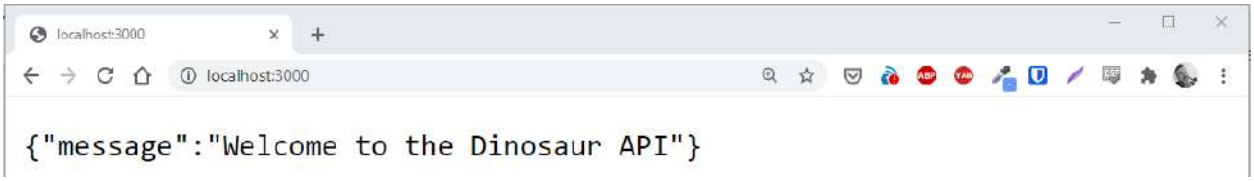


Figure 15: Your first route

Testing your API using Postman

For a simple API request like the one you just made, a browser will do. However, when you want to submit more sophisticated requests with more parameters and, especially, those in the body of a request, then you will want to test them with a dedicated client.

You could use a command-line HTTP client like **`curl`** for this, but I'm going to suggest that you use a free graphical tool called Postman. It enables you to easily create HTTP requests, view the responses, and save them in collections so that you can reuse them.

Download and install Postman here. Run it and click the **Add Collection** link to create a new collection. Call it **Dinosaur API** and click the **Create** button:

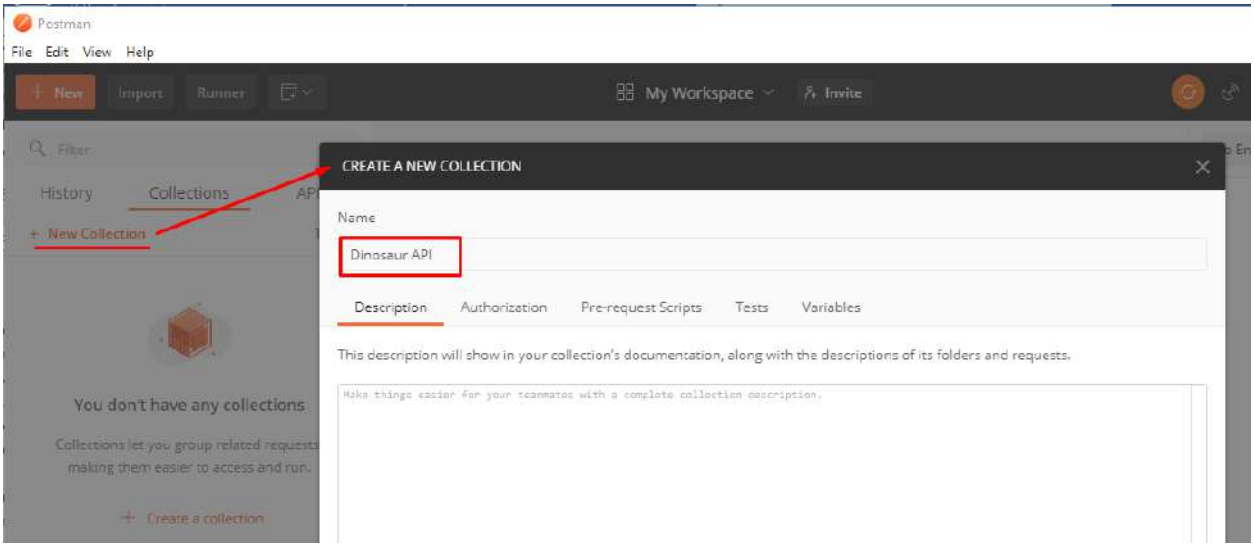


Figure 16: Creating a collection in Postman

Click the plus symbol to create a new request:

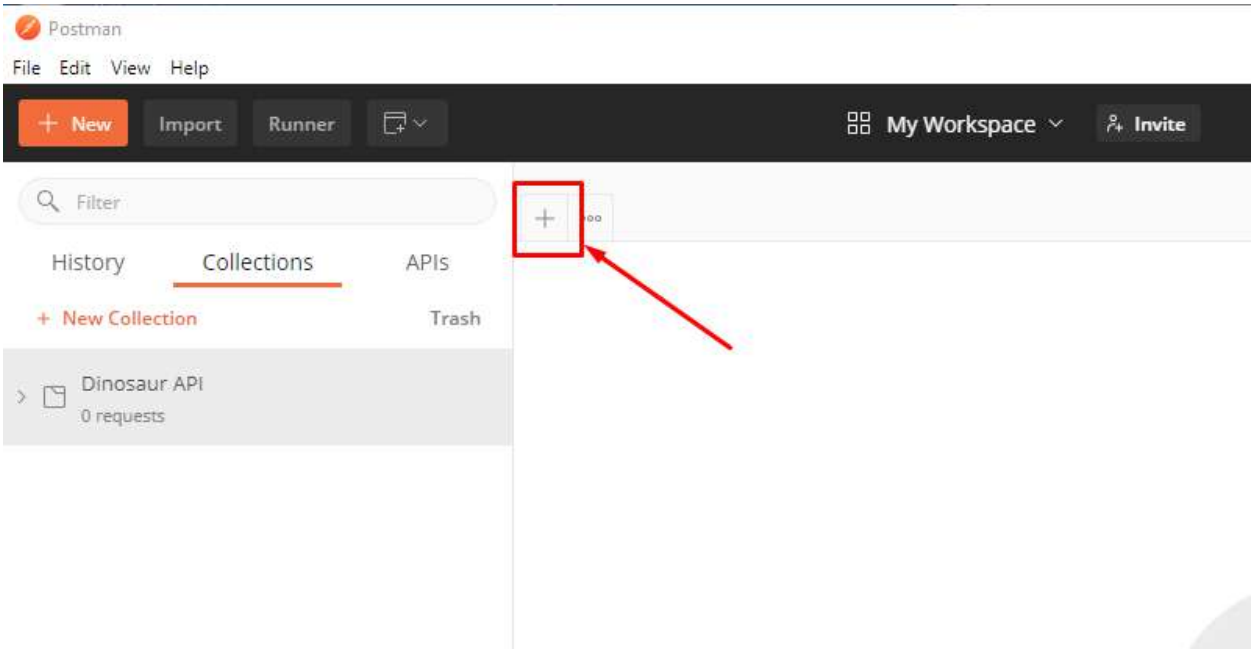


Figure 17: Creating a new request

In the tab that appears, select **GET** from the drop-down menu and type **http://localhost:3000**:

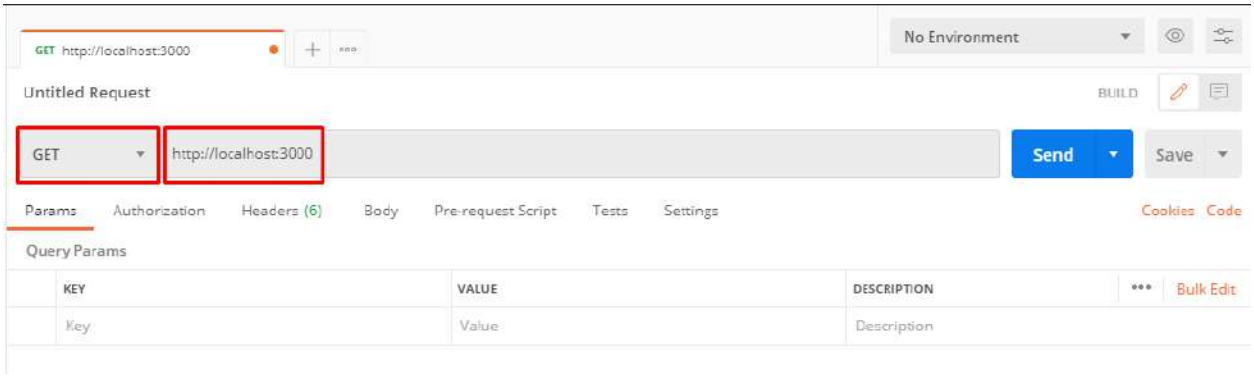


Figure 18: The request HTTP method and endpoint

Ensure that your server is still running. If it is not, restart it:

Code Listing 84

```
$ deno run --allow-net server.ts
Check file:///C:/Users/mplew/Repos/deno_succinctly/dinosaur-api/server.ts
Server is running....
```

Then, click **Send** in Postman to submit your request. In the response area at the bottom of the tab, you should see the response from your API (in the JSON tab):



Figure 19: The response body in Postman

Note the status (**`200 OK`**) and timing information.

Save your request by clicking the **Save** button. Give it a name (such as **Root**) and save it in the **Dinosaur API** collection. You can now re-run this request at any time.

Structuring the application

As soon as you start to build anything non-trivial, it makes sense to consider the organization of your code so that it's easy to extend and maintain.

Let's do that by creating a couple new code files and moving some of the logic in our **`server.ts`** file into them.

Managing dependencies

One immediate side effect of splitting our code into several different files is that we need to ensure that each file is importing the correct modules. This can be a real pain.

So far, we've just been grabbing the latest version of each module, like this:

Code Listing 85

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

That's okay while we're in development, but in production you'll likely want to refer to specific versions of modules so that you can guarantee that your code works:

Code Listing 86

```
import { Application, Router } from "https://deno.land/x/oak@6.3.2/mod.ts";
```

But what happens if you're using that module widely across your application? It means that you need to change every code file that refers to it. If you get it wrong, you might have different parts of your application using different versions of a module. If you're relying on a feature in one version that another doesn't have, then your application could be unpredictable.

One way to avoid this is to have all your dependencies in a single file. The TypeScript convention for this is to use a file called **deps.ts**. Essentially, you *export* all your dependencies from **deps.ts**, and then you *import* any modules you need in your code from **deps.ts**.

Create a new **deps.ts** file in the root of your application. In it, export the **oak** module:

Code Listing 87

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

Then, in **server.ts**, import **oak** from **deps.ts**:

Code Listing 88

```
// server.ts
import { Application, Router } from "./deps.ts";
```

Run the server and ensure that everything still works as before. As we add more dependencies to our application, we will include them in **deps.ts**.

Refactoring the codebase

Okay, calling it a “codebase” at this stage might be stretching it, but let’s see if we can come up with a good structure for the application we’re building.

Let’s leave the **server.ts** file responsible only for what its name suggests: setting up and running the server, and providing the initial entry point for our application. It’s going to need access to our defined routes though, so we’ll import those from **routes.ts**, which we’ll create shortly.

Code Listing 89

```
// server.ts
import { Application } from "./deps.ts";
import router from "./routes.ts";

const app = new Application();
app.use(router.routes());
app.use(router.allowedMethods());

app.listen({ port: 3000 });
console.log("Server is running....");
```

We'll store all our routes in **routes.ts** and create a handler function for each route to keep this file simple, so we can see immediately which routes our API supports. Of course, we'll need to import the handlers here too:

Code Listing 90

```
// routes.ts
import { Router } from "./deps.ts";
import { showWelcome } from "./handlers.ts";

const router = new Router();

router
  .get("/", showWelcome);

export { router };
```

We'll keep our route handlers in **handlers.ts**:

```
// handlers.ts
import { RouterContext } from "./deps.ts";

const showWelcome = async (ctx: RouterContext) => {
  ctx.response.body = { message: "Welcome to the Dinosaur API" };
  ctx.response.status = 200;
};

export { showWelcome };
```

Of course, we need access to that **RouterContext** type so that we can process the request and response. It wasn't a problem when this code was in **server.ts** because we were importing Oak's **Application** object, and the **RouterContext** type came with it. We could export the entire **Application** module here, but that's overkill, so let's just use **deps.ts** to export the type:

Code Listing 91

```
//deps.ts

export { Application, Router } from "https://deno.land/x/oak/mod.ts";
export type { RouterContext } from "https://deno.land/x/oak/mod.ts";
```

That will do for now. If we expanded our application, we could further refine our project by assigning code files to subfolders, but this is a good basis for what we want to achieve now.

Setting up the database

We're going to need somewhere to store our dinosaurs! (It will have to be somewhere pretty big.) Let's use MongoDB.

MongoDB is a document-oriented "NoSQL" database, which is great for storing and retrieving large amounts of data at high speed. Unlike traditional databases that use tables and rows, MongoDB stores data in collections and documents. A *document* is just a set of key/value pairs that you can think of as a record in relational database terminology. A *collection* contains sets of documents, and is the functional equivalent of a table.

To make things simpler, rather than downloading, installing, and configuring MongoDB on our local machine, we'll be using MongoDB Atlas, a cloud-based instance of MongoDB.

Register for a MongoDB Atlas account here, or sign in if you already have one:

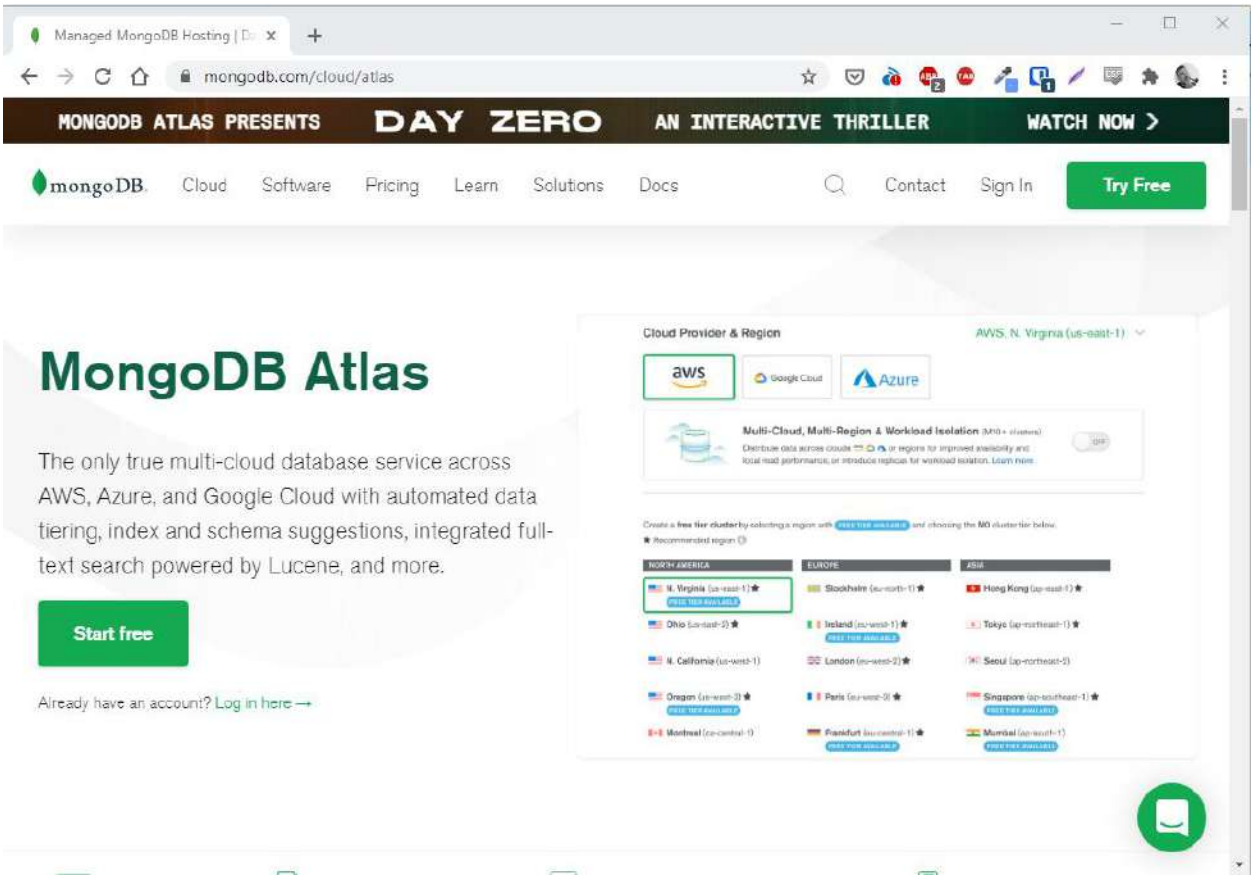


Figure 20: MongoDB Atlas

Create an organization (think of it as a kind of namespace) and a project. I've called mine **Mark's Org** and **Dinosaur API**, respectively:

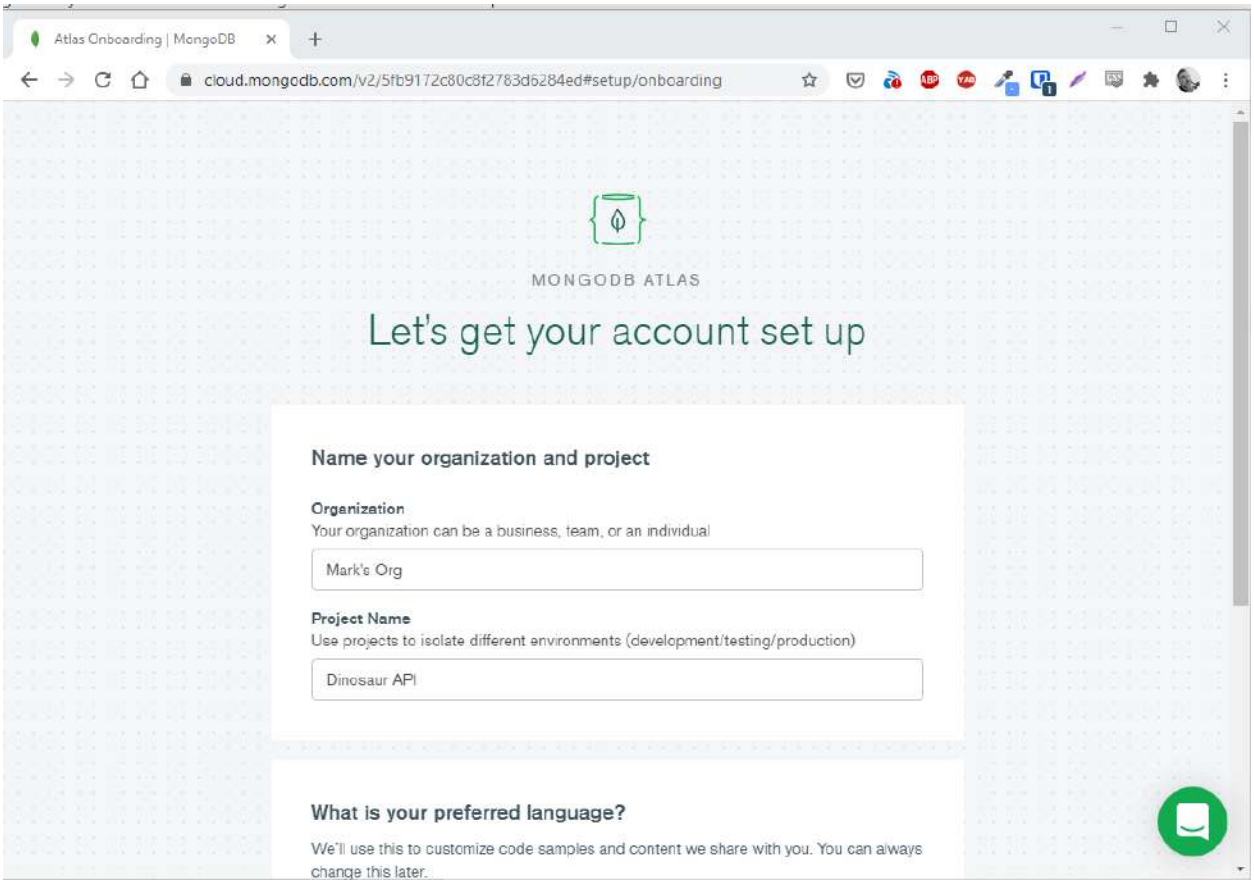


Figure 21: Creating an organization

Click to create the free shared cluster tier, and click **Create a cluster**:

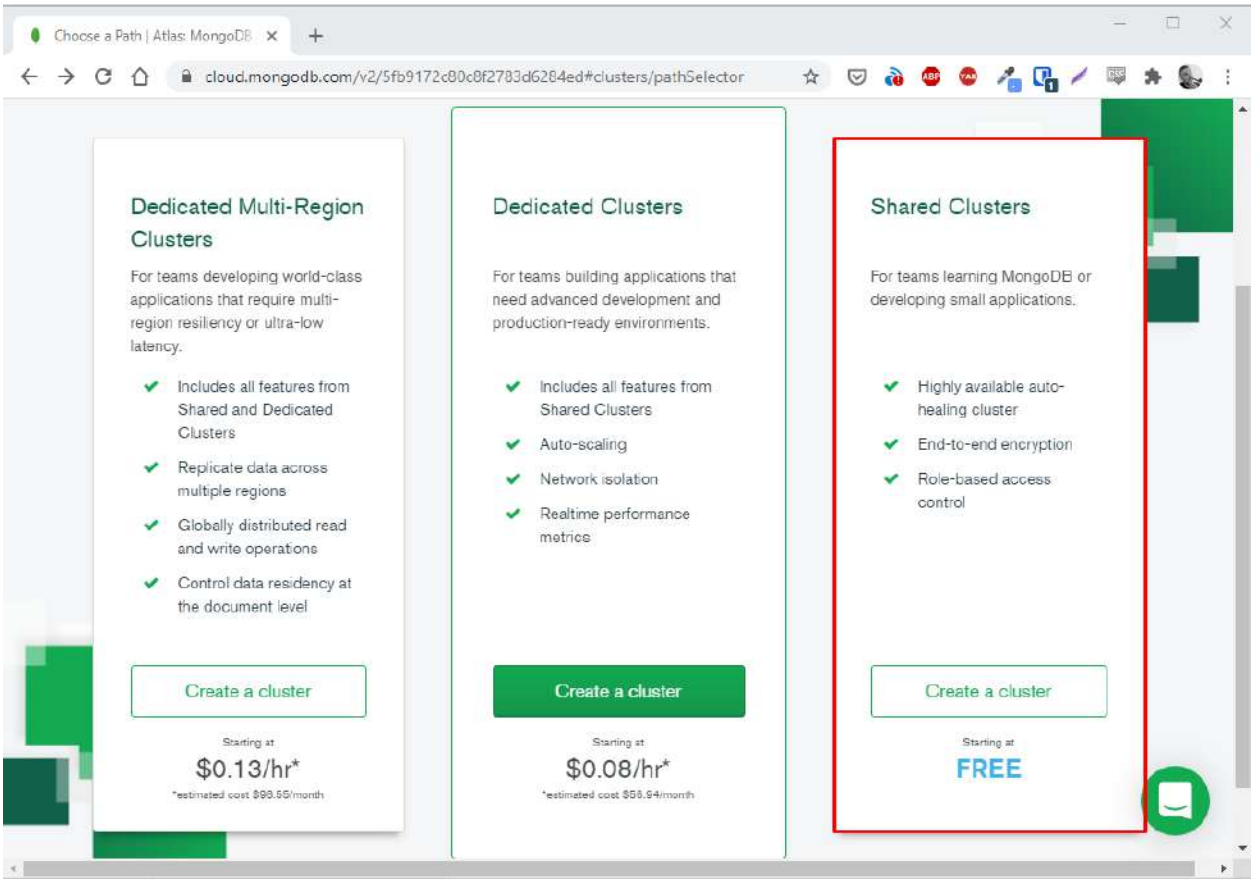


Figure 22: Select "Shared Clusters"

Select a cloud provider (AWS, Google Cloud, or Azure). It doesn't really matter which one you choose. Select a region close to you and click **Create Cluster**:

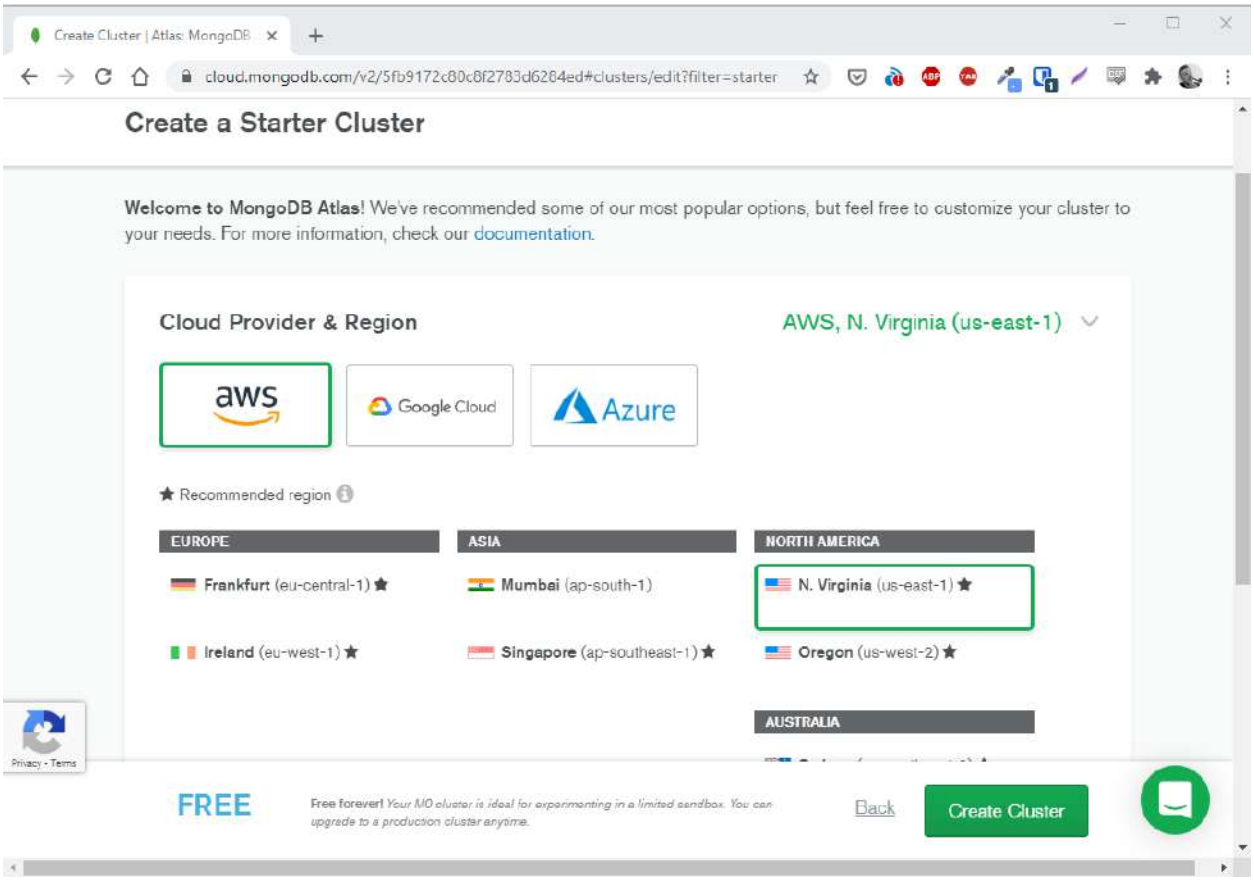


Figure 23: Selecting a cloud provider

MongoDB Atlas then builds your cluster, which will enable you to host your databases in the cloud. This step can take a few minutes:

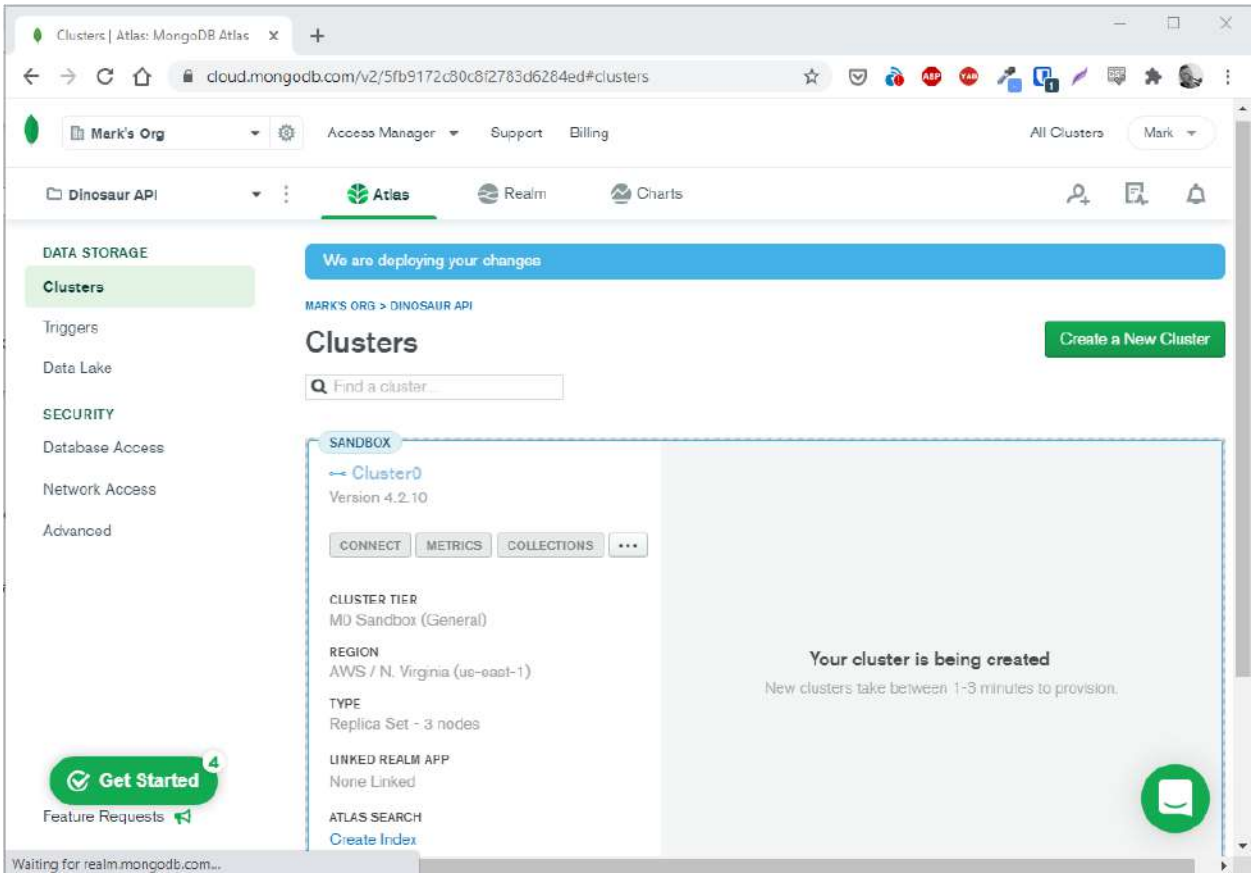


Figure 24: Waiting for the cluster to build

Select **Database Access** in the left-hand menu and click **Add New Database User**:

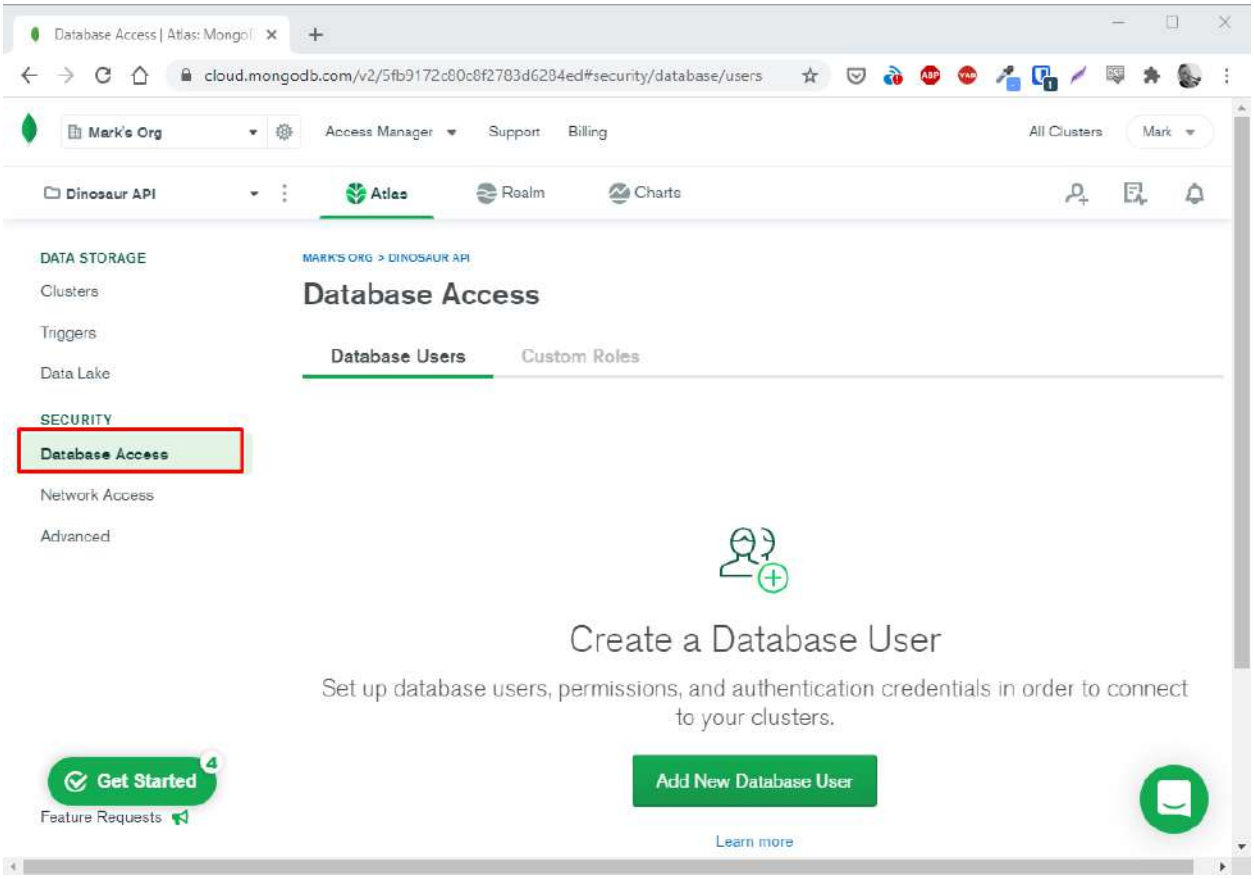


Figure 25: Creating a new user

Select **Password Authentication**, create a new user called **admin**, and autogenerate a secure password, which you should copy somewhere—you'll need it in a bit. Ensure that the admin user has the **Read and write to any database** privilege. Then click **Create User**.

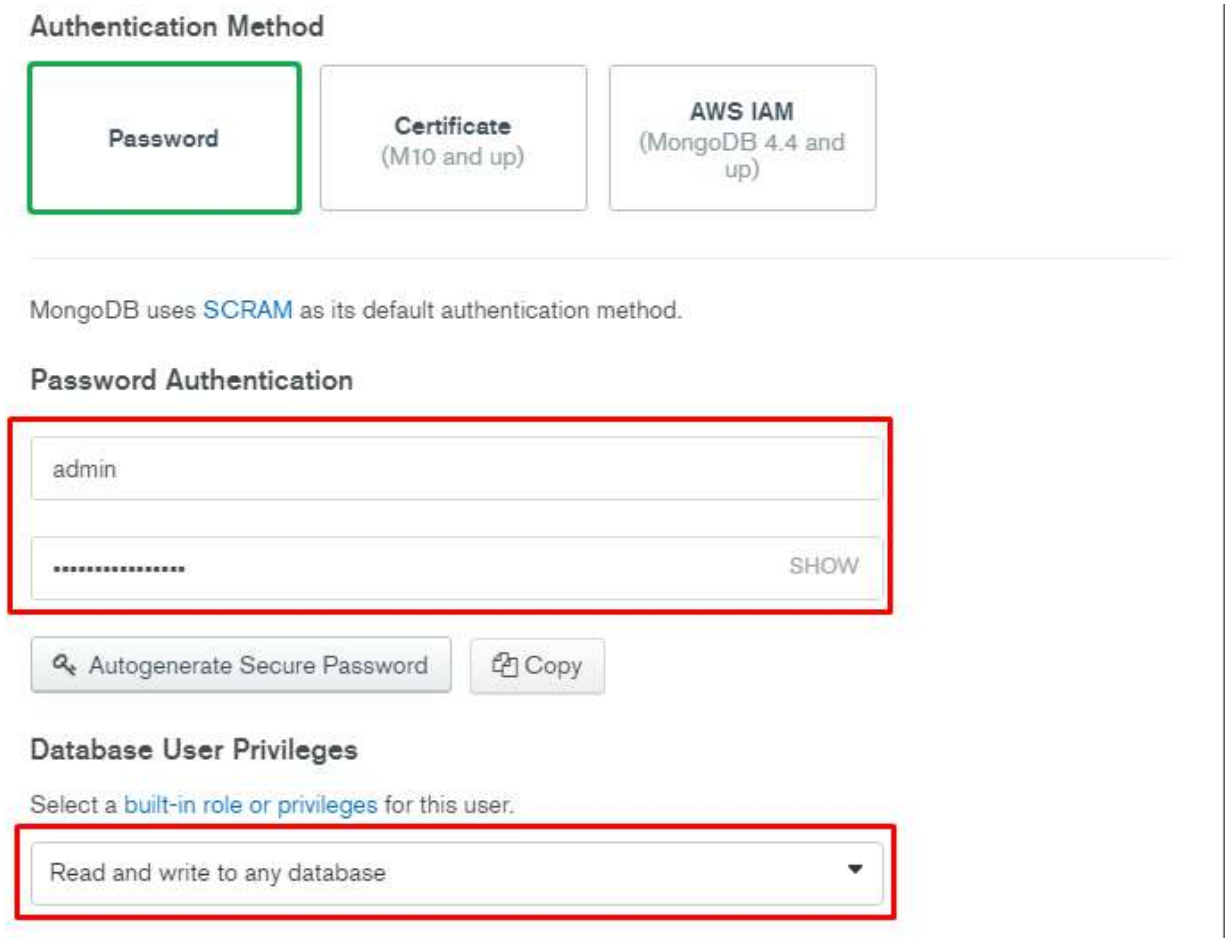


Figure 26: Assigning user privileges

Now you need to add your IP address to an allow list so that your application will be able to connect to the cluster. Select **Network Access** in the left-hand menu and click **Add IP Address**:

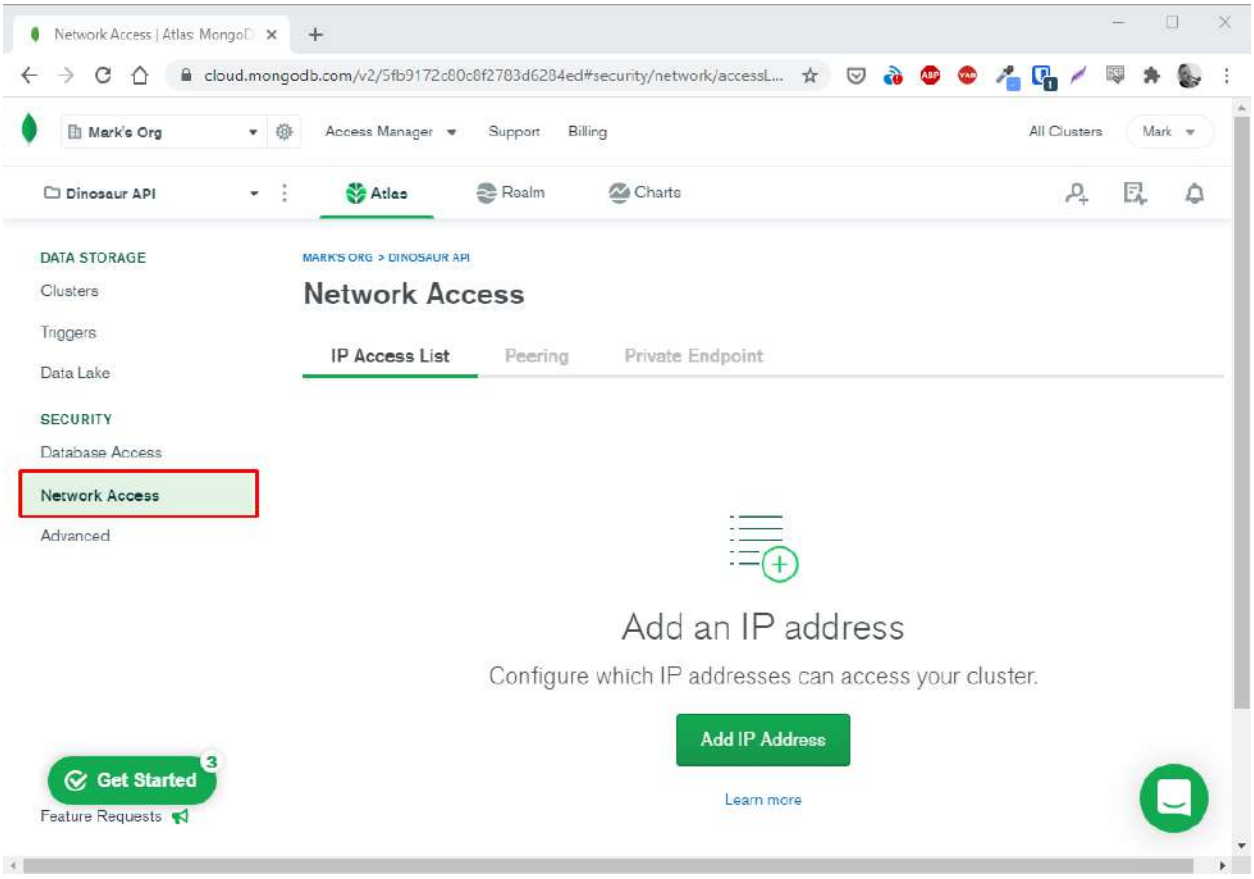


Figure 27: Adding your IP address to the allow list

During development, we'll allow access from anywhere. Click **Allow Access From Anywhere** and this will enter 0.0.0.0/0 in the **Access List Entry** field. Click **Confirm**.

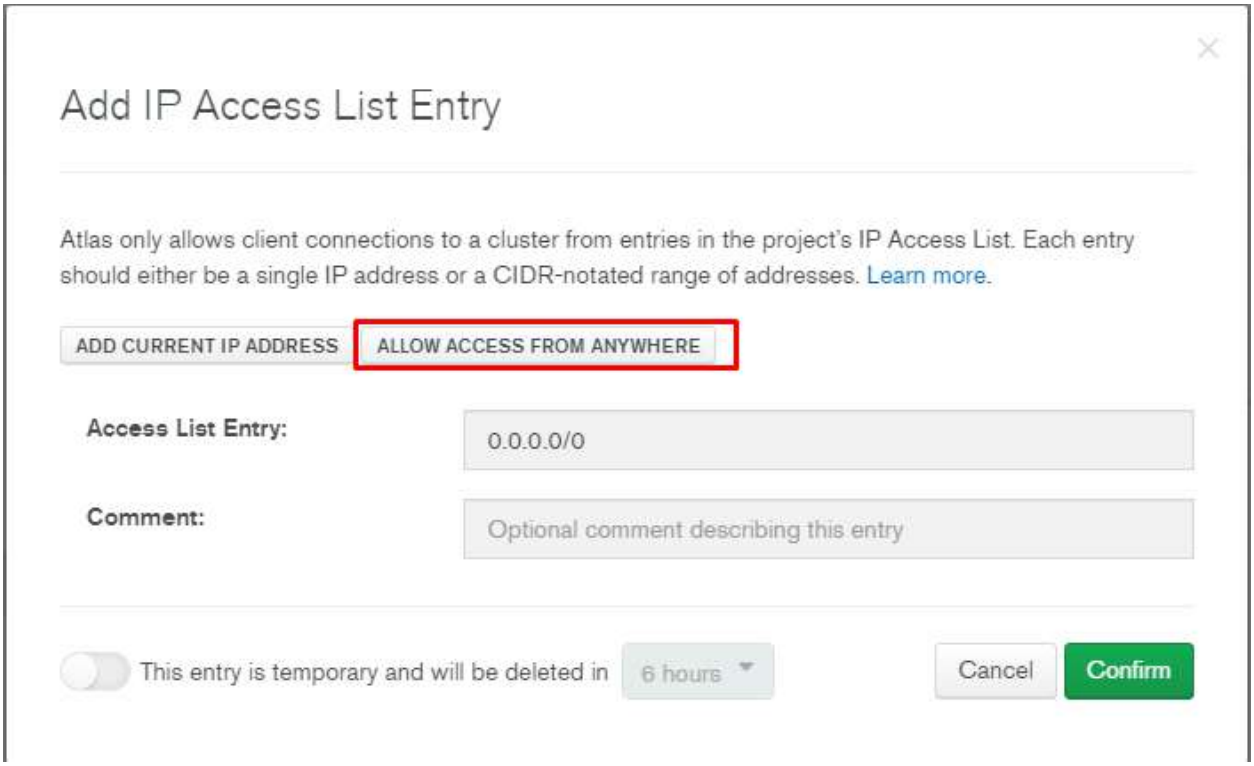


Figure 28: Allowing access from anywhere during development

Click **Clusters** in the left-hand menu to return to your cluster and then click **Connect**:

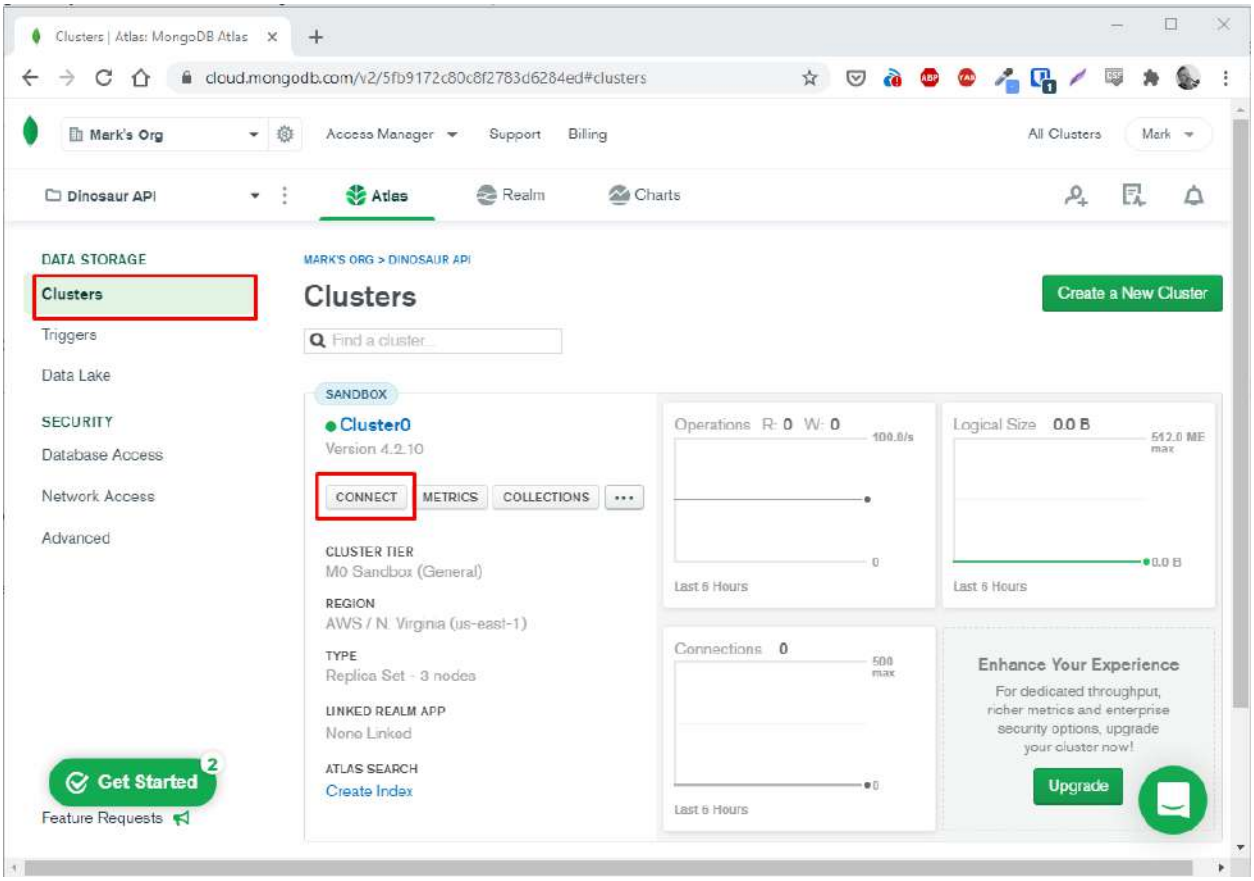


Figure 29: Connecting a cluster

Select **Connect your application**:

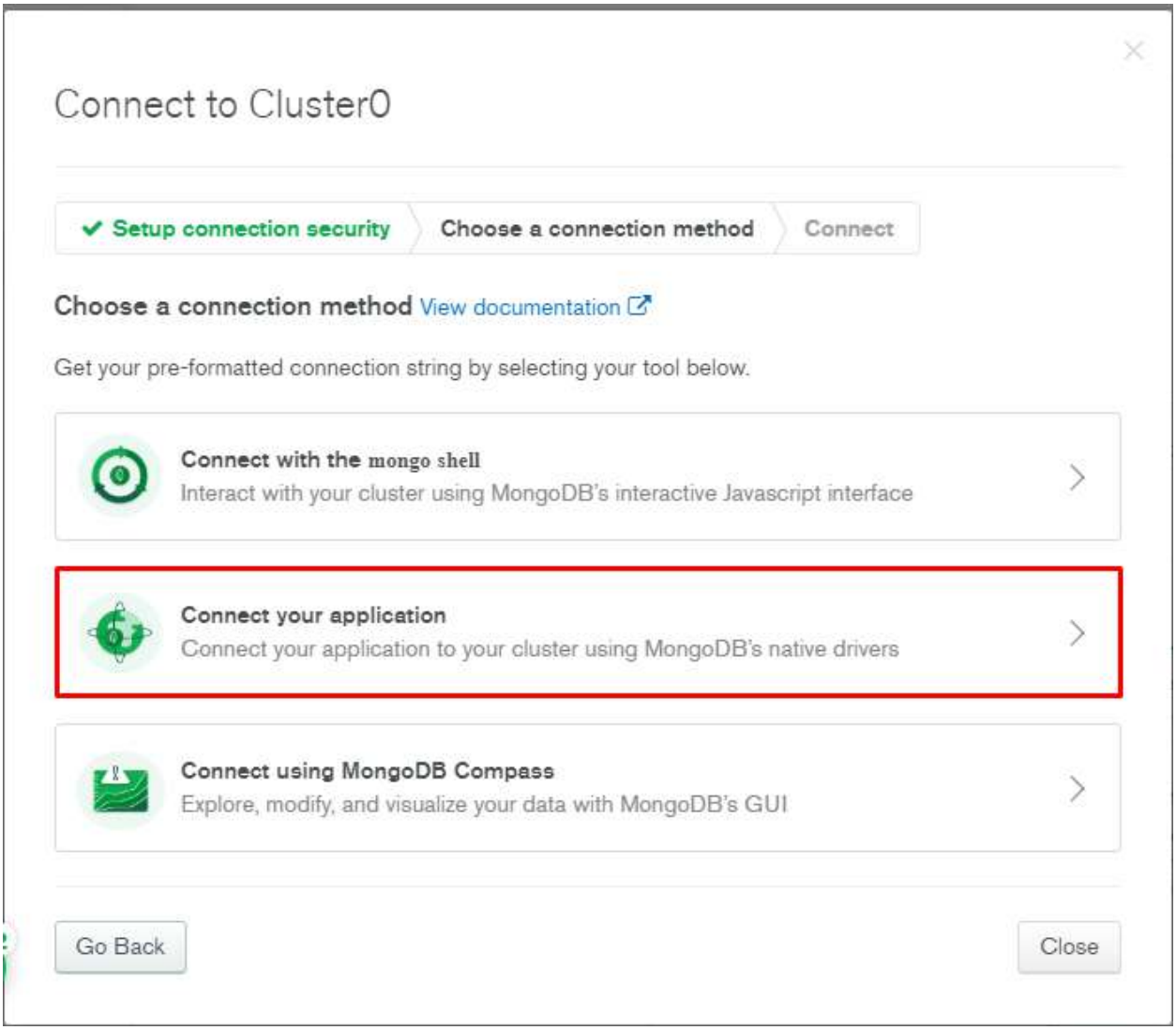


Figure 30: Select "Connect your application"

In the **Connect to Cluster** dialog, leave the driver and version as the default Node.js version and click **Copy** to copy the connection string. Paste it somewhere safe, along with the password you created earlier.

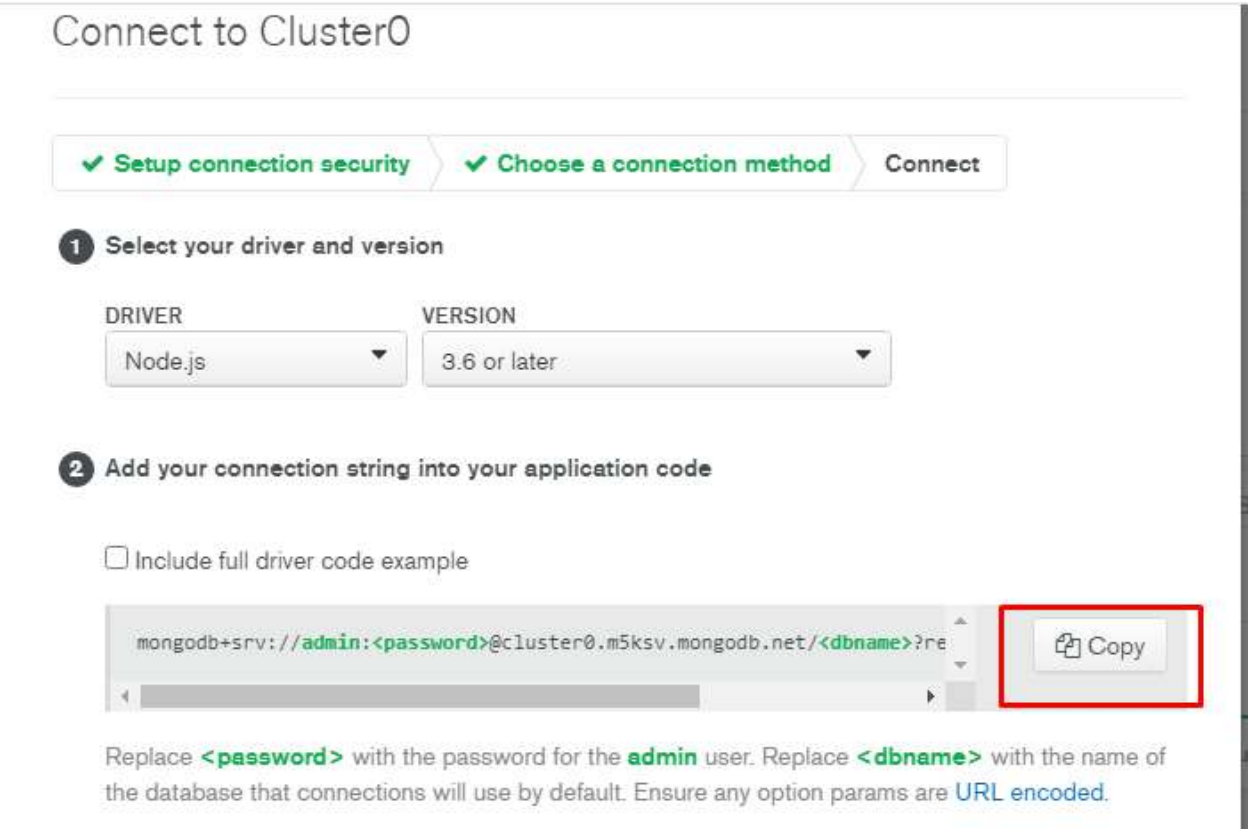


Figure 31: Copying the connection string

Storing the database credentials

Now that you've set up your MongoDB cluster, you need to get your application talking to it. Since we'll be storing the authentication details for the database in our application, we're going to use the **`dotenv`** module to store them in a configuration file and read them in at runtime.

Create a file called **`.env`** in the root of your application and enter the following:

Code Listing 92

```
DB=dinopedia
DB_USER=admin
DB_PASSWORD=T1WMgkpXX2gT9rBf
```

Add the **dotenv** module to **deps.ts**:

Code Listing 93

```
//deps.ts
export { Application, Router } from "https://deno.land/x/oak/mod.ts";
export type { RouterContext } from "https://deno.land/x/oak/mod.ts";
export { config } from "https://deno.land/x/dotenv/mod.ts";
```

Create a new file called **db.ts** for all your database code and import the configuration settings from **.env**:

Code Listing 94

```
// db.ts

import { config } from "./deps.ts";

const { DB, DB_USER, DB_PASSWORD } = config();
```

Creating the schema

While MongoDB is a schema-less database, we want to ensure that our API is consistent when reading documents from or writing documents to our collection. To do that, we're going to define an interface to represent a dinosaur document. Create a file called **schema.ts** and define the **DinosaurSchema** interface:

Code Listing 95

```
// schema.ts

interface DinosaurSchema {
  _id: { $oid: string };
  name: string;
  epoch: string;
  habitat: string;
}

export default DinosaurSchema;
```

As you can see, of all the fascinating traits of dinosaurs, we're only interested in three in our API: the name, the age (or "epoch") in which they lived, and whereabouts in the world they are known to have stomped around. The other field, the **_id** object, is an internal unique identifier that MongoDB requires.

Import the **DinosaurSchema** into **db.ts**:

Code Listing 96

```
// db.ts

import { config } from "./deps.ts";
import DinosaurSchema from "./schema.ts";

const { DB, DB_USER, DB_PASSWORD } = config();
```

Connecting to the database

To connect to the database, we're going to need a driver. We'll use the **deno_mongo** driver, specifically the **MongoClient** class, which you should export from **deps.ts**:

Code Listing 97

```
//deps.ts

export { Application, Router } from "https://deno.land/x/oak/mod.ts";
export type { RouterContext } from "https://deno.land/x/oak/mod.ts";
export { config } from "https://deno.land/x/dotenv/mod.ts";
export { MongoClient } from "https://deno.land/x/mongo/mod.ts";
```

Import it into **db.ts** and write the code to connect to the database, using the credentials in **.env**.

Code Listing 98

```
// db.ts

import { config } from "./deps.ts";
import DinosaurSchema from "./schema.ts";
import { MongoClient } from "./deps.ts";

const { DB, DB_USER, DB_PASSWORD } = config();

const connectStr =
  `mongodb+srv://${DB_USER}:${DB_PASSWORD}@cluster0.fmukv.mongodb.net/${DB}
?retryWrites=true&w=majority`;

const client = new MongoClient();
client.connectWithUri(connectStr);

const db = client.database(DB);
const dinosaurCollection = db.collection<DinosaurSchema>("dinosaurs");
```

```
export { db, dinosaurCollection };
```

Note how we're using the connection string you copied when you configured the connection to your MongoDB cluster. We're basically replacing the user, password, and database names in the string with the settings in **`.env`**.

We create a connection to Mongo, switch to the **`dinopedia`** database (again, from **`.env`**), and create a collection called **`dinosaurs`**. Neither of these yet exist, but accessing them in this way creates them the first time we execute this code. Subsequent executions will use the existing database and collection.

Finally, we export references to the database and collection because we want to create some new routes whose handlers will interact with the database.

Listing all the dinosaurs

We're now going to create a route that lists all the dinosaurs in the collection. The route will be accessible via a **`GET`** request to the following URL:

```
http://localhost/dinosaurs
```

First, add the route to the **`routes.ts`** file:

Code Listing 99

```
// routes.ts

import { Router } from "./deps.ts";
import { getDinosaurs, showWelcome } from "./handlers.ts";

const router = new Router();

router
  .get("/", showWelcome)
  .get("/dinosaurs", getDinosaurs);

export { router };
```

Then we need to add the **`getDinosaurs()`** handler for this route to **`handlers.ts`**. To interact with the dinosaur collection, we need to first import it from **`db.ts`**:

Code Listing 100

```
// handlers.ts

import { RouterContext } from "./deps.ts";
import { dinosaurCollection } from "./db.ts";

...

const getDinosaurs = async (ctx: RouterContext) => {
  const dinosaurs = await dinosaurCollection.find();

  ctx.response.body = dinosaurs;
  ctx.response.status = 200;
};

export { getDinosaurs, showWelcome };
```

In this handler, we're using the **`find()`** method on the collection with no arguments to return all the documents in the collection. We send the results back in the response body with a **`200 OK`** HTTP status code.

Let's try this out. First, run the server:

Code Listing 101

```
$ deno run --allow-net --allow-env --allow-read --allow-write --allow-
plugin --unstable server.ts
INFO load deno plugin "deno_mongo" from local
"C:\Users\mplew\Repos\deno_succinctly\dinosaur-
api\.deno_plugins\deno_mongo_ce02adbd9ca9967016cbf45958ee753b.dll"
Server is running....
```

Wow. Look at all those arguments. Especially that **`--unstable`** flag. That's because, at the time of writing, there is code that the MongoDB driver uses that hasn't yet been certified as stable by Deno. You must specify **`--unstable`** to run it.



Tip: Supplying all those permissions every time you execute your program can be a real pain. So, during development, you can grant* all *permissions to your Deno program by executing it with the `-A` flag. (You still need to specify `--unstable` where required, however.) For example: `deno run -A --unstable mod.ts`.

Also, note how the Deno driver has installed a plugin. You can find it in your project directory in the **`.deno_plugins`** folder. Don't expect to be able to view the contents, however, because it's a binary file:

```
\---dinosaur-api
    |   .env
    |   db.ts
    |   deps.ts
    |   handlers.ts
    |   routes.ts
    |   schema.ts
    |   server.ts
    |
    \---.deno_plugins
            deno_mongo_ce02adbd9ca9967016cbf45958ee753b.dll
```

Figure 32: The MongoDB plugin DLL

Now that your server is running, create a new request in Postman. It should be a **`GET`** request to the **`http://localhost:3000/dinosaurs`** endpoint. Save it as **Get all dinosaurs** in your Dinosaur API collection:

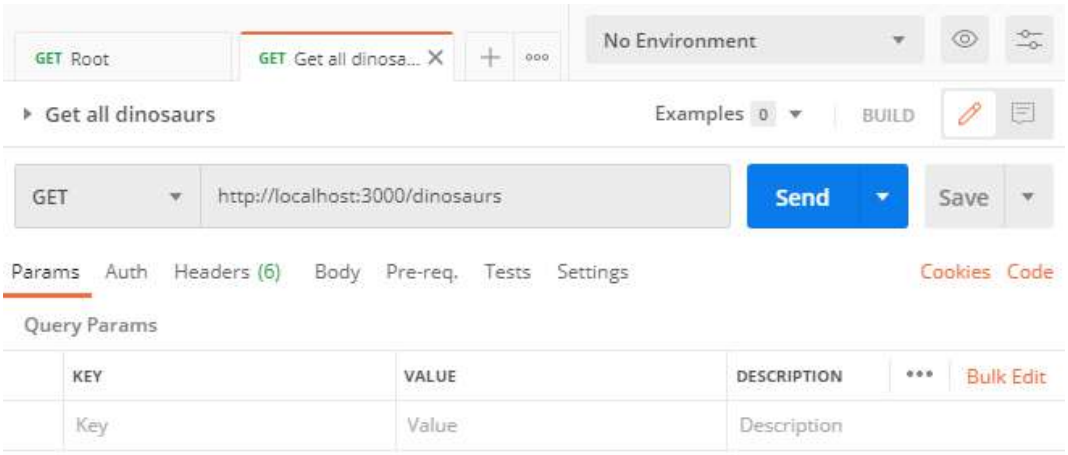


Figure 33: Retrieving all dinosaurs

Click **Send** and look at the response:



Figure 34: No dinosaurs!

If all goes well, you should see an empty array (`[]`). That's because our dinosaur collection is empty!



Tip: If you see an `ECONNREFUSED` error, your server probably is not running. Run the server and try the request again.

Now that we've got our application talking to the database, we need to figure out how to add some dinosaurs!

Adding dinosaurs

Users will be able to add a new dinosaur to the collection by making a **`POST`** request to **`http://localhost/dinosaurs`** and including the dinosaur details in the body of the request.

Let's create a route and a handler for this.

First, in **`routes.ts`**:

Code Listing 102

```
// routes.ts

import { Router } from "./deps.ts";
import { addDinosaur, getDinosaurs, showWelcome } from "./handlers.ts";

const router = new Router();

router
  .get("/", showWelcome)
  .get("/dinosaurs", getDinosaurs)
  .post("/dinosaurs", addDinosaur);

export { router };
```

And then create the **`addDinosaur()`** handler in **`handlers.ts`**:

```
// handlers.ts

import { RouterContext } from "./deps.ts";
import { dinosaurCollection } from "./db.ts";

...

const addDinosaur = async (ctx: RouterContext) => {
  if (!ctx.request.hasBody) {
    ctx.response.body = { message: "No data" };
```

```
    ctx.response.status = 400;
    return;
  }

  const { name, epoch, habitat } = await ctx.request.body().value;
  const result = await dinosaurCollection.insertOne({
    name,
    epoch,
    habitat,
  });
  ctx.response.status = 201;
  ctx.response.body = { message: `Added dinosaur with ID: ${result.$oid}` }
;
};

export { addDinosaur, getDinosaurs, showWelcome };
```

This uses the **`insertOne()`** method on the collection to take the fields in the request and add them to a new document in the collection. First, we check if there is any request body. If not, we return a **`400 Bad Request`** error and exit the handler.

Otherwise, we return a **`201 Created`** status in the response, together with the ID of the document that is autogenerated by MongoDB.

Try this out by running the server and creating a new **`POST`** request in Postman called **Add dinosaur**. The endpoint is **http://localhost:3000/dinosaurs**.

Click the **Body** link and select **raw** and **JSON** from the drop-down lists. Add your dinosaur JSON to the editor pane:

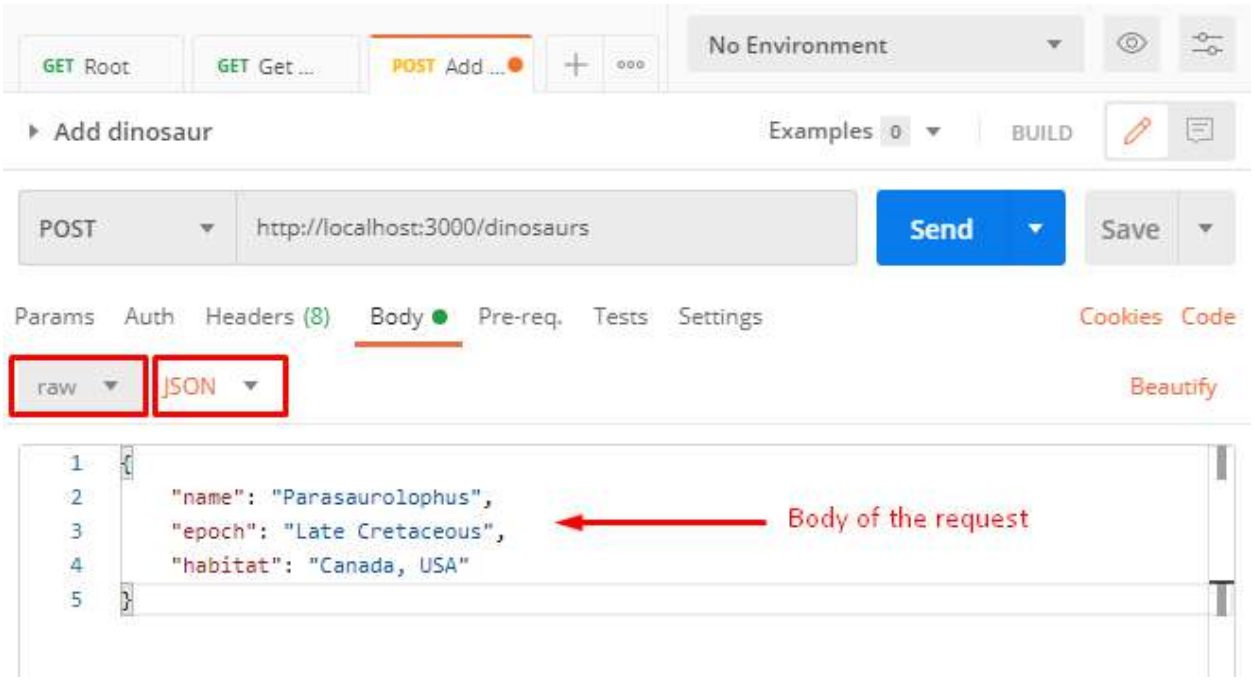


Figure 35: Adding a dinosaur

Save your request as **Add dinosaur**, ensure that your server is running, and then click **Send**. If all goes according to plan, you should receive the ID of the new document:

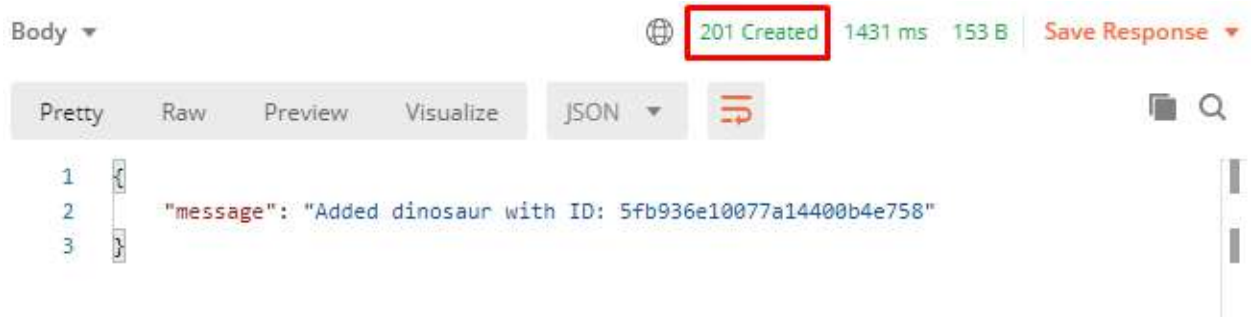


Figure 36: Successfully adding a dinosaur

Still in Postman, open your **Get all dinosaurs** request and click **Send**. The newly created dinosaur appears in the response:



Figure 37: The new dinosaur appears in the response

If you want to add a few more dinosaurs, then here are some more you can try:

Code Listing 103

```
{
	"name": "Omeisaurus",
	"epoch": "Mid Jurassic",
	"habitat": "China"
}

{
	"name": "Talarurus",
	"epoch": "Late Cretaceous",
	"habitat": "Mongolia"
}

{
	"name": "Velociraptor",
	"epoch": "Late Cretaceous",
	"habitat": "Mongolia"
}

{
	"name": "Kotasaurus",
	"epoch": "Early Jurassic",
	"habitat": "India"
}

{
	"name": "Becklespinax",
	"epoch": "Early Cretaceous",
```

```
      "habitat": "England"
}
```



Tip: To find out more about these dinosaurs and discover many others, visit the UK Natural History Museum's Dino Directory.

Finding a specific dinosaur

We can now add a dinosaur and list all the dinosaurs in the collection. Let's add a path to retrieve a specific dinosaur by the ID that MongoDB generated for it.

This will be another **`GET`** request, this time to the following URL: **`http://localhost/dinosaurs/<ID>`**.

First, add the route, with a handler called **`getDinosaur()`**. This route will include a path parameter for the ID portion of the URL, which we denote by prefixing it with a colon: **`http://localhost/dinosaurs/:id`**.

Code Listing 104

```
// routes.ts

import { Router } from "./deps.ts";
import {
  addDinosaur,
  getDinosaur,
  getDinosaurs,
  showWelcome,
} from "./handlers.ts";

const router = new Router();

router
  .get("/", showWelcome)
  .get("/dinosaurs", getDinosaurs)
  .get("/dinosaurs/:id", getDinosaur)
  .post("/dinosaurs", addDinosaur);

export { router };
```

And then code the route handler, **`getDinosaur()`**:

Code Listing 105

```
// handlers.ts

import { RouterContext } from "./deps.ts";
import { dinosaurCollection } from "./db.ts";

...

const getDinosaur = async (ctx: RouterContext) => {
  const id = ctx.params.id;

  const dinosaur = await dinosaurCollection.findOne({
    _id: {
      $oid: `${id}`,
    },
  });

  if (!dinosaur) {
    ctx.response.body = { message: "Dinosaur not found" };
    ctx.response.status = 404;
    return;
  }
  ctx.response.body = dinosaur;
  ctx.response.status = 200;
};

export { addDinosaur, getDinosaur, getDinosaurs, showWelcome };
```

This handler uses the collection's **`findOne()`** method to find a specific document based on the path parameter supplied in the request. The **`findOne()`** method expects the JSON element that you are searching for. In this case it's the **`_id`** object that MongoDB uses to uniquely identify documents.

If no such document exists, it returns a **`404 Not Found`** error. Otherwise, it returns the document itself.

Test it by creating a new **`GET`** request to **`http://localhost:3000/dinosaurs/<id>`**, using the ID of one of the dinosaurs in your collection (which you can find by submitting the "List all dinosaurs" request). Save this request as **Find dinosaur** in Postman. When you send it, the response should include the document you requested:

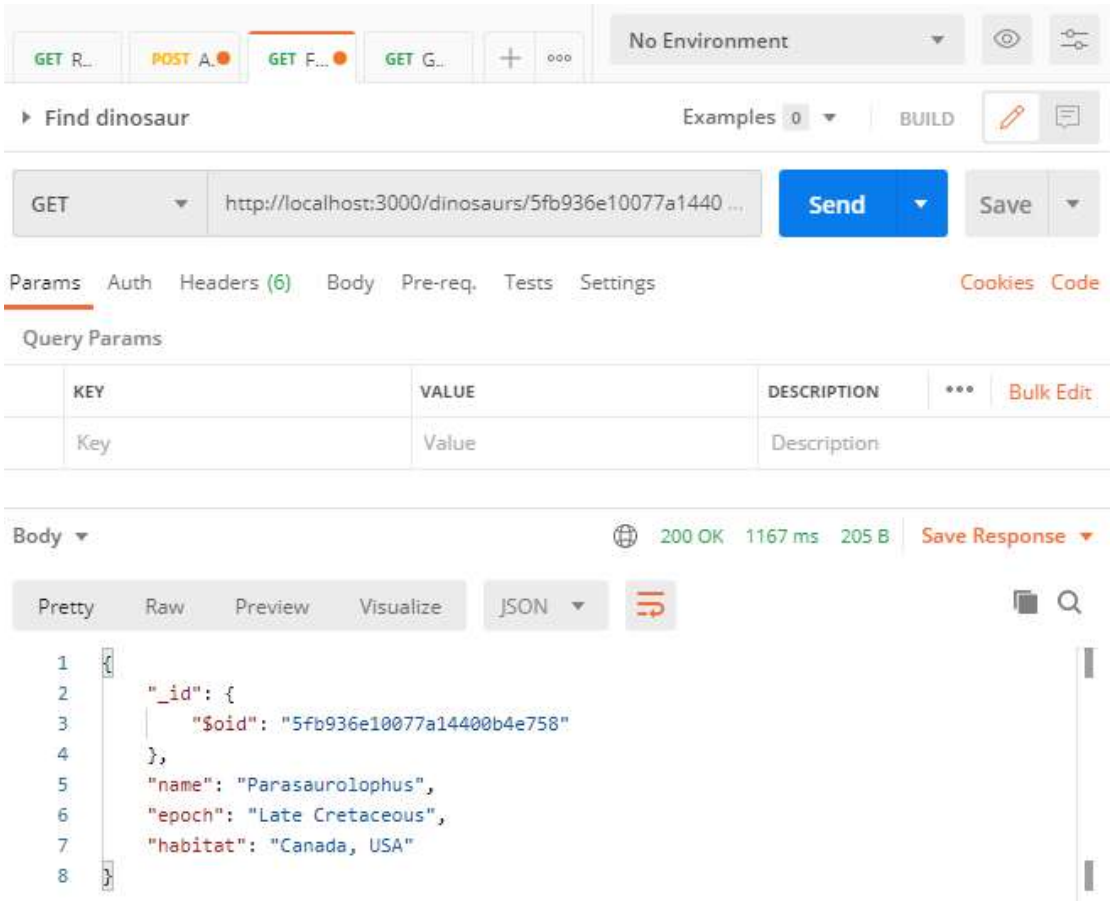


Figure 38: Finding a dinosaur

Editing a dinosaur

Maybe some new research has come to light and we want to update the details of a dinosaur. Let's add an endpoint for that. It will be similar to the one to retrieve a specific dinosaur, in that we'll specify the ID in the path. However, this time we will make it a **PUT** request and include the changes that we want to make in the body of the request.

Define the new route in **`routes.ts`**:

Code Listing 106

```
// routes.ts

import { Router } from "./deps.ts";
import {
  addDinosaur,
  getDinosaur,
  getDinosaurs,
  showWelcome,
```

```
  updateDinosaur,
} from "./handlers.ts";

const router = new Router();

router
  .get("/", showWelcome)
  .get("/dinosaurs", getDinosaurs)
  .get("/dinosaurs/:id", getDinosaur)
  .put("/dinosaurs/:id", updateDinosaur)
  .post("/dinosaurs", addDinosaur);

export { router };
```

Then, code the route handler **`updateDinosaur()`** in **`handlers.ts`**:

```
// handlers.ts

import { RouterContext } from "./deps.ts";
import { dinosaurCollection } from "./db.ts";

...

const updateDinosaur = async (ctx: RouterContext) => {
  const id = ctx.params.id;

  if (!ctx.request.hasBody) {
    ctx.response.body = { message: "No data" };
    ctx.response.status = 400;
    return;
  }

  const { name, epoch, habitat } = await ctx.request.body().value;
  const { modifiedCount } = await dinosaurCollection.updateOne({
    _id: {
      $oid: `${id}`,
    },
  }, { $set: { name, epoch, habitat } });

  if (!modifiedCount) {
    ctx.response.body = { message: "Dinosaur not found" };
    ctx.response.status = 404;
    return;
  }
  ctx.response.body = { message: "Dinosaur updated" };
  ctx.response.status = 200;
};
```

```
export { addDinosaur, getDinosaur, getDinosaurs, showWelcome, updateDinosaur };
```

First, we check for a request body. If there is none, then we return a 400 error code indicating no data found.

Then we use the **`updateOne()`** method on the collection to provide the **`_id`** of the document we want to update in the first parameter and an object representing the changes we want to make to the document in the second parameter.

Note the addition of the **`$set`** property in the second object. The default behavior of MongoDB is just to overwrite the existing document with whatever we include here. So, if we only want to change the **`habitat`**, and only provide a value for that property, then MongoDB will believe that we want to keep **`habitat`** and discard the other fields. Clearly that is not what we want to do in this example, and we don't want to have to retrieve the current field values just to write them back out to the database. The **`$set`** basically tells Mongo that if any of the properties are unchanged, they should still be included in the updated document.

The **`updateOne()`** method returns a count of the number of documents that it modified. We can use this as an indicator of whether the operation is successful or not, and return the appropriate response.

Try it out by creating a new **`PUT`** request called **`Update dinosaur`** in Postman. Supply the ID of an existing dinosaur in the collection and the modification that you want to make in the request body. For example:

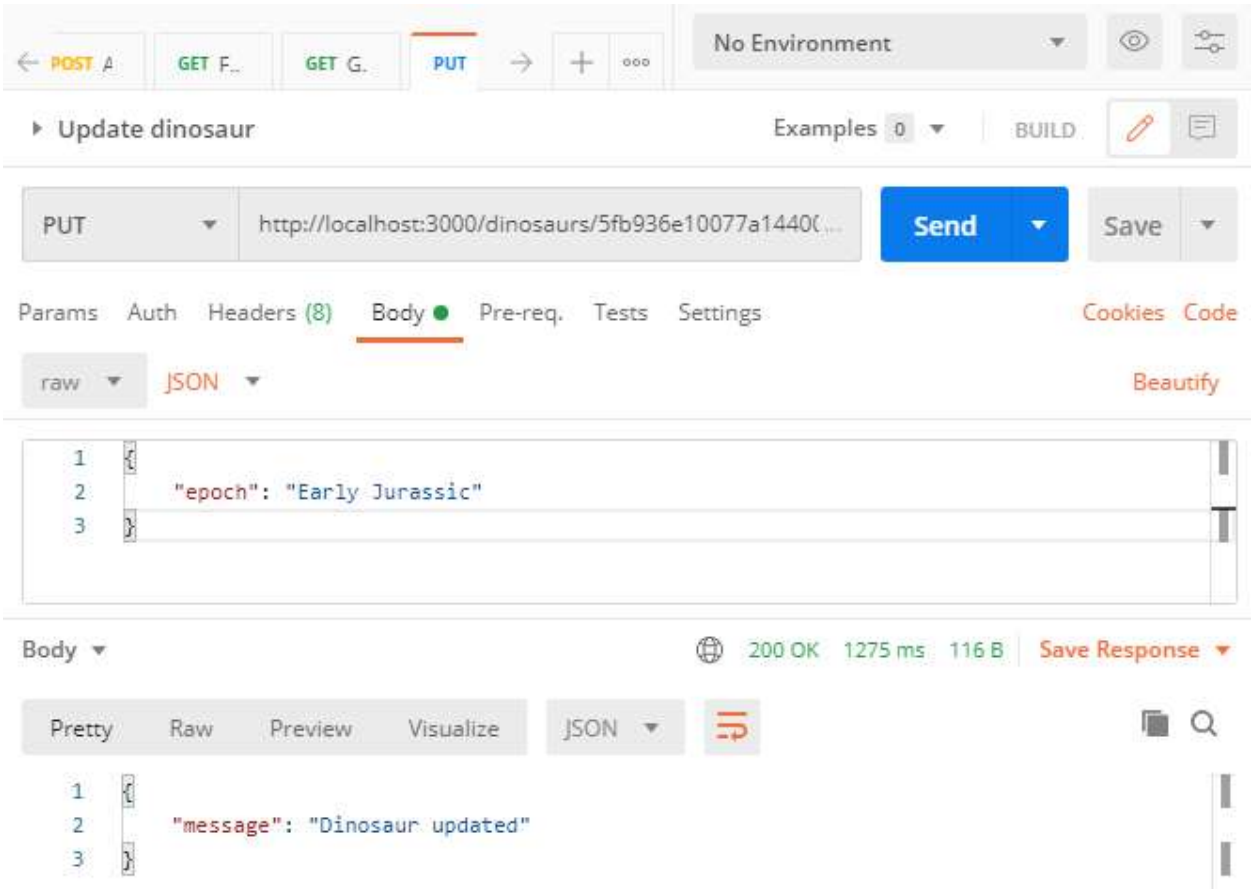


Figure 39: Editing a dinosaur

Use the **`Find dinosaur`** request to locate the dinosaur and verify that it applied your updates:

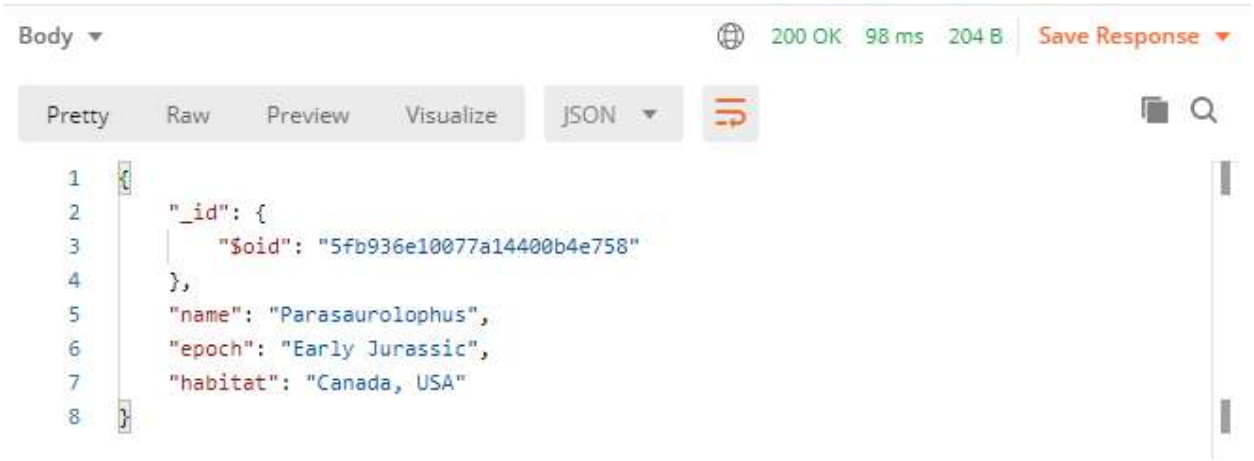


Figure 40: Verifying your edits

Deleting a dinosaur

There is just one more operation to implement before we can truly call our API a CRUD API, and that's the ability to delete a dinosaur. We'll do this via a **`DELETE`** request, passing the ID of the document we want to delete as a path parameter.

Define the route in **`routes.ts`**:

Code Listing 107

```
// routes.ts

import { Router } from "./deps.ts";
import {
  addDinosaur,
  deleteDinosaur,
  getDinosaur,
  getDinosaurs,
  showWelcome,
  updateDinosaur,
} from "./handlers.ts";

const router = new Router();

router
  .get("/", showWelcome)
  .get("/dinosaurs", getDinosaurs)
  .get("/dinosaurs/:id", getDinosaur)
  .put("/dinosaurs/:id", updateDinosaur)
  .post("/dinosaurs", addDinosaur)
  .delete("/dinosaurs/:id", deleteDinosaur);

export { router };
```

Then code the **`deleteDinosaur()`** route handler in **`handlers.ts`**:

Code Listing 108

```
// handlers.ts
import { RouterContext } from "./deps.ts";
import { dinosaurCollection } from "./db.ts";

...

const deleteDinosaur = async (ctx: RouterContext) => {
  const id = ctx.params.id;
  const deleteCount = await dinosaurCollection.deleteOne({
    _id: {
      $oid: `${id}`,
```

```
    },
  });

  if (!deleteCount) {
    ctx.response.body = { message: "Dinosaur not found" };
    ctx.response.status = 404;
    return;
  }
  ctx.response.body = { message: "Dinosaur deleted" };
  ctx.response.status = 200;
};

export {
  addDinosaur,
  deleteDinosaur,
  getDinosaur,
  getDinosaurs,
  showWelcome,
  updateDinosaur,
};
```

The **`deleteDinosaur()`** handler uses **`deleteOne()`** on the collection and works in a similar way to **`findOne()`**, except that instead of returning the document, it deletes it! And then it returns a count of the records deleted, which we can check to see if the operation was successful.

Create a new Postman request called **`Delete dinosaur`** that uses a **`DELETE`** request and points to resource **`http://localhost:3000/dinosaurs/<id>`**. Test it using your least favorite dinosaur and ensure that sucker is not only extinct, but never existed in the first place!

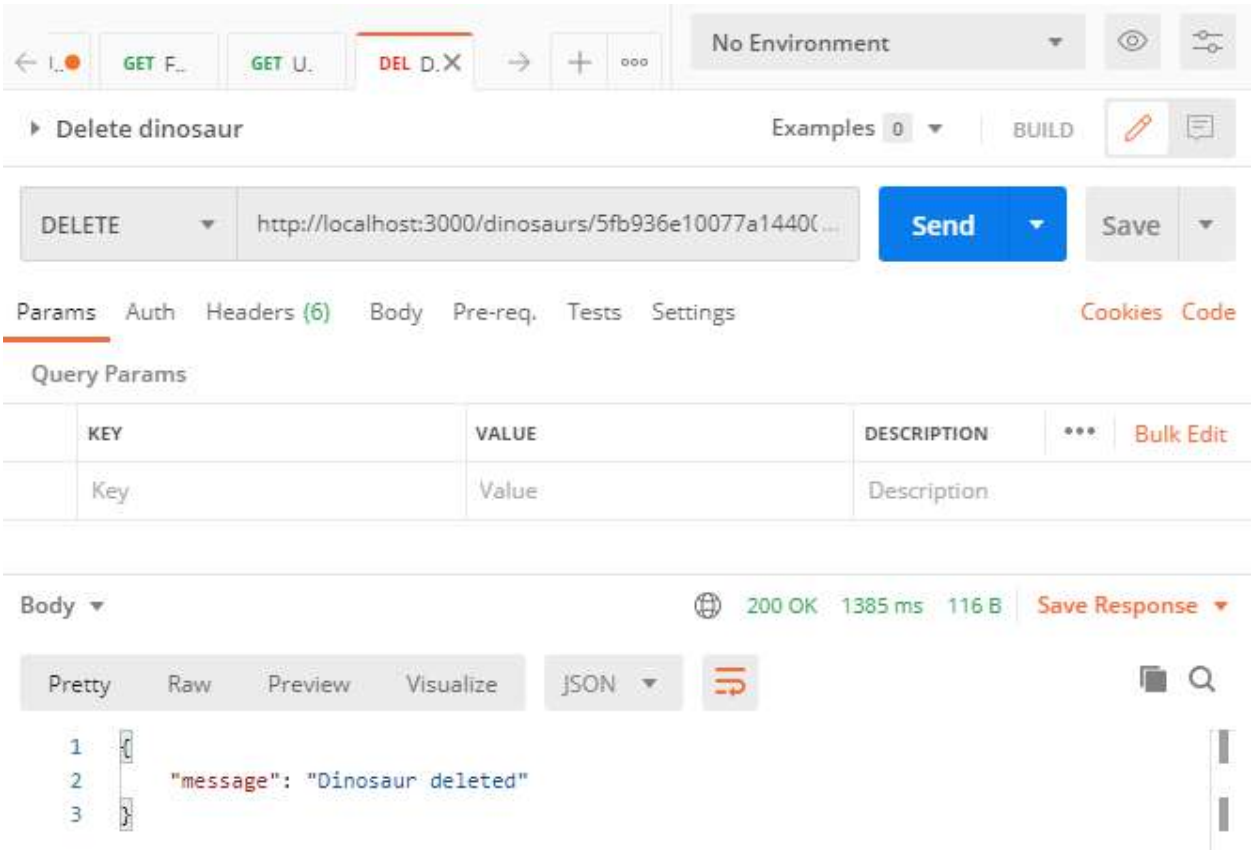


Figure 41: Deleting a dinosaur

Adding extra functionality with middleware

Let's talk a bit about middleware.

If you were building this API using Deno's standard library **`http`** module, you would have to write a single request handler for your entire application. A framework like Oak lets you write multiple request handlers for different routes so that you can give them all specific jobs, like authenticating a user, or reading dinosaur details in from a database.

In your vanilla Deno **`http`** application, your request handler has just one object representing the incoming request, and one representing the response that you will send back to the client. But in an Oak application, these objects are fed into an array where each element is a function. This array of functions is known as the *middleware stack*, and it allows us to intercept these objects and do useful things with them. For example, once a user has logged in, you can fetch their profile information from the database and store it in the request object as **`request.user`**.

In short, middleware functions let you write code to perform further processing on each request, or on each request for a specific route, and manipulate these request and response objects.

Middleware functions have three standard parameters: **`request`**, **`response`**, and a **`next()`** function that can call the next middleware function in the stack. You add each function to the stack by using the **`Application`** object's **`use()`** function in the order that you want them to execute.

Each function is called in turn and makes whatever changes are required to the request and response objects, and passes them on to the next function in the chain by calling **`next()`**. The last function in the stack sends the response back to the browser.



Tip: It's important to remember to call `next()` (unless it's the last function in the chain), or your request will hang and ultimately time out.

Let's add two middleware functions to our API.

Route not found middleware

First, we'll create a middleware function that returns a **`HTTP 400 Not Found`** error if someone tries to access an endpoint that doesn't exist.

Create a directory called **`middleware`** and, inside it, a file called **`notFound.ts`**. In **`notFound.ts`**, write the following code:

Code Listing 109

```
// notFound.ts

export default ({ response }: { response: any }) => {
  response.status = 404;
  response.body = {
    message: "Endpoint not found.",
  };
};
```

Place this after your existing calls to **`app.use()`** in **`server.ts`**. (Yes, you're already using middleware in your application!)

So your application first invokes any matching routes (**`app.use(router.routes())`**), retrieves the collection of HTTP verbs (**`GET`**, **`POST`**, etc.) that are permitted by the routes and sends them to the browser (**`app.use(router.allowedMethods())`**), and then calls your **`notfound`** function:

Code Listing 110

```
// server.ts

import { Application } from "./deps.ts";
import { router } from "./routes.ts";
```

```
import notFound from "./middleware/notFound.ts";

const app = new Application();
app.use(router.routes());
app.use(router.allowedMethods());
app.use(notFound);

app.listen({ port: 3000 });
console.log("Server is running....");
```

If the route is found, the route's handler will have already returned the response, so your **notfound** middleware will never be executed. There's no need to call **next()**, because it's the last middleware function in the stack.

Test it by creating a request in Postman to a non-existent route. You should receive the following response:

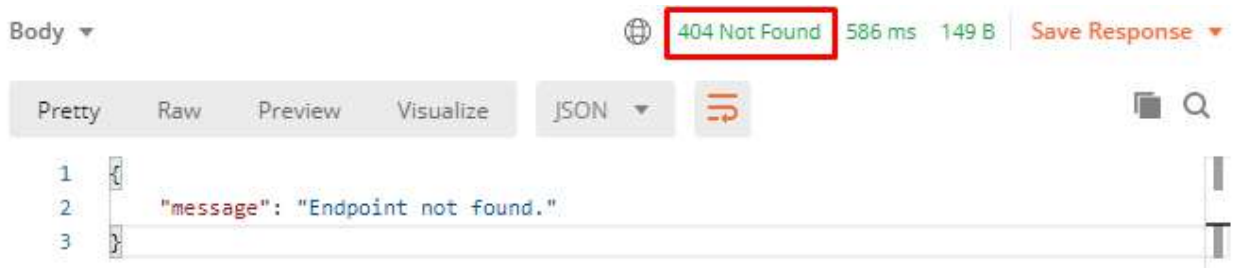


Figure 42: Testing your 404 middleware

Request logging middleware

Now we will write some more middleware that logs request data, including timings. We'll just dump the results to the console, but obviously we could also write this information to a file or a database. We'll invest some effort in making it look attractive though, with the aid of the **colors** module.

First, import **colors** in **deps.ts**:

Code Listing 111

```
//deps.ts

export { Application, Router } from "https://deno.land/x/oak/mod.ts";
export type { RouterContext } from "https://deno.land/x/oak/mod.ts";
export { config } from "https://deno.land/x/dotenv/mod.ts";
export { MongoClient } from "https://deno.land/x/mongo/mod.ts";
export {
  bgBlue,
  bgGreen,
```

```
  bgRed,
  cyan,
  green,
  white,
} from "https://deno.land/std/fmt/colors.ts";
```

Create another file in your middleware folder called **logging.ts**, and enter the following code:

Code Listing 112

```
// logging.ts

import { bgBlue, bgGreen, bgRed, cyan, green, white } from "../deps.ts";

const X_RESPONSE_TIME: string = "X-Response-Time";

export default {
  logger: async (
    { response, request }: { response: any; request: any },
    next: Function,
  ) => {
    await next();
    const responseTime = response.headers.get(X_RESPONSE_TIME);
    let statusText = "";

    if (response.status == 200 || response.status == 201) {
      statusText = `${bgGreen(white(String(response.status)))}`;
    } else {
      statusText = `${bgRed(white(String(response.status)))}`;
    }

    console.log(
      `${statusText} | ${green(request.method)} ${
        cyan(request.url.pathname)
      } -- ${bgBlue(white(String(responseTime)))}`,
    );
  },
  responseTime: async (
    { response }: { response: any },
    next: Function,
  ) => {
    const start = Date.now();
    await next();
    const msec: number = Date.now() - start;
    response.headers.set(X_RESPONSE_TIME, `${msec}ms`);
  },
};
```

There are actually two middleware functions in this module: **`logger()`** and **`responseTime()`**, which you need to add to the stack in **`server.ts`**:

Code Listing 113

```
// server.ts

import { Application } from "./deps.ts";
import { router } from "./routes.ts";
import notFound from "./middleware/notFound.ts";
import logging from "./middleware/logging.ts";

const app = new Application();
app.use(logging.logger);
app.use(logging.responseTime);
app.use(router.routes());
app.use(router.allowedMethods());
app.use(notFound);

app.listen({ port: 3000 });
console.log("Server is running....");
```

Tip: Be careful about the order of the middleware functions here. The `logger()` and `responseTime()` functions must run before the route handlers are invoked.

The **`logger()`** function executes first. The code immediately hits **`await next()`**, which passes control to the **`responseTime()`** function as the next middleware function in the stack.

In **`responseTime()`**, it records the current time and jumps to the next piece of middleware, which is the route handler for whatever API endpoint the client is accessing. Having executed the code in the handler, the flow of execution returns to **`responseTime()`** and again records the time now that the route handler has done its work. It calculates the difference between the two timestamps and injects this information into the request using a header called **`X-RESPONSE-TIME`**. It then returns control to **`logger()`**, which then formats the information and logs it to the console:

```
$ deno run -A --unstable server.ts
Check file:///C:/Users/mplew/Repos/deno_succinctly/dinosaur-api/server.ts
INFO load deno plugin "deno_mongo" from local "C:\Users\mplew\Repos\deno_su
o_plugins\deno_mongo_ce02adbd9ca9967016cbf45958ee753b.dll"
Server is running....
200 | GET /dinosaurs -- 815ms
201 | POST /dinosaurs -- 149ms
404 | PUT /dinosaurs/5fb936e10077a14400b4e758 -- 88ms
404 | DELETE /dinosaus/5fb936e10077a14400b4e758 -- 1ms
404 | DELETE /dinosaurs/5fb936e10077a14400b4e758 -- 157ms
200 | GET /dinosaurs -- 100ms
```

Figure 43: Testing your logging middleware

Summary

Well done! You built a working CRUD API with some nice extra features.

Along the way, you learned about:

- Structuring a Deno project, including managing external dependencies.
- The Oak framework, Deno's implementation of Node's Koa.
- Storing data in and retrieving data from MongoDB.
- Middleware—what it is, and how to write your own.

...and hopefully a bit more about dinosaurs!

You can find the source code for this project here.

Chapter 6 Conclusion and Resources

Thank you for joining me on this whistle-stop tour of Deno! I hope you found it interesting and useful and that it has excited you about the potential of Deno.

If you have any comments about the contents of this book or questions about Deno in general, I would be delighted to hear from you. You can reach me at mark@marklewin.com.

In the meantime, here are some resources that I found invaluable while getting to know Deno and I hope that you will find them useful too:

- https://deno.land: The official Deno website and documentation portal.
- https://youtu.be/M3BM9TB-8yA: Ryan Dahl's talk, "Ten Things I Regret About Node.js."
- https://www.youtube.com/watch?v=AOvg_GbnsbA&t=35m13s: A deep dive into Deno's internals by Bartek Iwanczuk.
- https://denoweekly.com/: A weekly(ish) roundup of what's new in the Denoverse.
- https://github.com/denolib/awesome-deno: An exhaustive, curated list of Deno links.
- https://discord.gg/deno: A Deno discussion group.