

- [Deno.cwd](#) - Get the current working directory
- [Deno.readDir](#) - Get directory listings
- [Deno.readFile](#) - Read a file into memory
- [Deno.readTextFile](#) - Read a text file into memory
- [Deno.open](#) - Open a file for streaming reading
- [Deno.stat](#) - Get file system entry information
- [Deno.lstat](#) - Get file system entry information without following symlinks
- [Deno.realpath](#) - Get the real path of a file after resolving symlinks

5.1.3 Future support

In the future, these APIs will also be added:

- [Cache API](#)
- UDP API:
- `Deno.connectDatagram` for outbound UDP sockets
- Customizable fetch options using `Deno.createHttpClient`

5.1.4 Limitations

Just like the Deno CLI, we do not implement the `__proto__` object field as specified in ECMA Script Annex B.

5.2 BroadcastChannel

In Deno Deploy, code is run in different data centers around the world in order to reduce latency by servicing requests at the data center nearest to the client. In the browser, the BroadcastChannel API allows different tabs with the same origin to exchange messages. In Deno Deploy, the BroadcastChannel API provides a communication mechanism between the various instances; a simple message bus that connects the various Deploy instances world wide.

- [Constructor](#)
- [Parameters](#)
- [Properties](#)
- [Methods](#)
- [Example](#)

5.2.1 Constructor

The BroadcastChannel() constructor creates a new BroadcastChannel instance and connects to (or creates) the provided channel.

let channel = new BroadcastChannel(channelName);

5.2.1.1 Parameters

name	type	description
channelName	string	The name for the underlying broadcast channel connection.

The return type of the constructor is a BroadcastChannel instance.

5.2.2 Properties

name	type	description
name	string	The name of the underlying broadcast channel.
onmessage	function (or null)	The function that's executed when the channel receives a new message ([MessageEvent][messageevent]).
onmessageerror	function (or null)	The function that's executed when the arrived message cannot be deserialized to a JavaScript data structure.

5.2.3 Methods

name	description
close()	Close the connection to the underlying channel. After closing, you can no longer post messages to the channel.
postMessage(message)	Post a message to the underlying channel. The message can be a string, object literal, a number or any kind of [Object][object].

BroadcastChannel extends [EventTarget](#), which allows you to use methods of EventTarget like addEventListener and removeEventListener on an instance of BroadcastChannel.

1 Guide

Deno Deploy is a distributed system that allows you to run JavaScript, TypeScript, and WebAssembly close to users, at the edge, worldwide. Deeply integrated with the V8 runtime, our servers provide minimal latency and eliminate unnecessary abstractions. You can develop your script locally using the Deno CLI, and then deploy it to our managed infrastructure in less than a second, without the need to configure anything.

Built on the same modern systems as the Deno CLI, Deno Deploy provides the latest and greatest in web technologies in a globally scalable way:

- **Builds on the Web:** use fetch, WebSocket, or URL just like in the browser
- **Built-in support for TypeScript and JSX:** type safe code, and intuitive server side rendering without a build step
- **Web compatible ES modules:** import dependencies just like in a browser, without the need for explicit installation
- **Direct GitHub integration:** push to a branch, review a deployed preview, and merge to release to production
- **Extremely fast:** deploy in less than a second, serve globally close to users
- **Deploy from URL:** deploy code with nothing more than a URL

1.1 Hello World

As a first introduction to Deno Deploy, let's deploy a little hello world script. This will just respond to all incoming HTTP requests with Hello World and a 200 OK HTTP status. We will be using the Deno Deploy playground to deploy and edit this script.

Before we start writing the actual script, let's go over some basics: Deno Deploy lets you listen for incoming HTTP requests using the same [server side HTTP API](#) as the Deno CLI. This API is rather low level though, so instead of using this API directly we'll use the high level HTTP API exposed by [std/http](#). This API revolves around the serve function that is exported from [https://deno.land/std@0.140.0/http/server.ts](#).

import { serve } from "https://deno.land/std@0.140.0/http/server.ts";

serve((_req) => { /* .. */ });

Note: the port number we listen on is not important, as Deno Deploy will automatically route requests from the outside world to whatever port we listen on.

The handler function is called with two arguments: a [Request](#) object, and a [ConnInfo](#) object. The Request object contains the request data, and the ConnInfo object contains information about the underlying connection, such as the origin IP address. You must return a [Response](#) object from the handler function.

Let's use this information to finish our hello world script:

import { serve } from "https://deno.land/std@0.140.0/http/server.ts";

```
serve((_req) => {
  return new Response("Hello World!", {
    headers: { "content-type": "text/plain" },
  });
});
```

Let's now deploy this script using a Deno Deploy playground. To get started, create a new playground project by visiting [https://dash.deno.com](#), signing in, and pressing the **New Playground** button. You can now copy the above code into the editor on the left side of the screen. To save and deploy the script, press the **Save & Deploy** button on the right side of the top toolbar (or press Ctrl+S). You can preview the result on the right side of the playground editor, in the preview pane. You will see that if you change the script (for example Hello, World! -> Hello, Galaxy!) and then re-deploy, the

Deno Deploy

preview will automatically update. The URL shown at the top of the preview pane can be used to visit the deployed page from anywhere.

Even in the playground editor, scripts are deployed worldwide across our entire global network. This guarantees fast and reliable performance, no matter the location of your users.

Congratulations. You have now deployed your first script to Deno Deploy 🎉!

1.2 Using JSX

Deno Deploy supports JSX (and TSX) out of the box. You don't need an additional transform step to use JSX with Deno Deploy.

The example below demonstrates the usage of JSX on Deno Deploy.

```
/** @jsx h */
/// <reference no-default-lib="true"/>
/// <reference lib="dom" />
/// <reference lib="dom.asynciterable" />
/// <reference lib="deno.ns" />
```

```
import { serve } from "https://deno.land/std@0.140.0/http/server.ts";
import { h, renderSSR } from "https://deno.land/x/nano_jsx@v0.0.20/mod.ts";
```

```
function App() {
  return (
    <html>
      <head>
        <title>Hello from JSX</title>
      </head>
      <body>
        <h1>Hello world</h1>
      </body>
    </html>
  );
}
```

```
function handler(req) {
  const html = renderSSR(<App />);
  return new Response(html, {
    headers: {
      "content-type": "text/html",
    },
  });
}
```

serve(handler);

You can run the above example with [deno run](#).

deno run --allow-net=:8000 https://deno.com/examples/hello.jsx

It is important that you use a .jsx or .tsx file extension, so Deno Deploy knows to interpret a given file as JSX/TSX.

1.3 Serving static assets

Note: to use this feature, you must link a GitHub repository to your project. URL deployments do not support static assets.

Deno Deploy supports the [Deno.readFile](#) API to read static assets from the file system. This is useful for serving static assets such as images, stylesheets, and JavaScript files. This guide demonstrates how to use this feature.

Imagine the following file structure on a GitHub repository:

5 Runtime

5.1 API Reference

Deno Deploy Runtime helps you write web servers in TypeScript/JavaScript using the Web APIs. It's different from Deno but aims to have similar APIs where applicable.

The following Web and Deno APIs are supported on Deno Deploy Runtime.

5.1.1 Web APIs

- [console](#)
- [atob](#)
- [btoa](#)
- [Fetch API](#)
- [fetch](#)
- [Request](#)
- [Response](#)
- [URL](#)
- [File](#)
- [Blob](#)
- [TextEncoder](#)
- [TextDecoder](#)
- [TextEncoderStream](#)
- [TextDecoderStream](#)
- [Performance](#)
- [Web Crypto API](#)
- [randomUUID\(\)](#)
- [getRandomValues\(\)](#)
- [SubtleCrypto](#)
- [WebSocket API](#)
- [Timers](#) (setTimeout, clearTimeout, and setInterval)
- [Streams API](#)
- [ReadableStream](#)
- [WritableStream](#)
- [TransformStream](#)
- [URLPattern API](#)
- [Import Maps](#)
- Note: import maps are currently only available via [deployctl](#) or [deployctl GitHub Action](#) workflows.

5.1.2 Deno APIs

Note: only stable APIs of Deno are made available in Deploy.

- [Deno.env](#) - Interact with environment variables (secrets).
- [get\(key: string\): string | undefined](#) - get the value of an environment variable.
- [toObject\(\): { \[key: string\]: string }](#) - get all environment variables as an object.
- [Deno.connect](#) - Connect to TCP sockets.
- [Deno.connectTls](#) - Connect to TCP sockets using TLS.
- [Deno.startTls](#) - Start TLS handshake from an existing TCP connection.
- [Deno.resolveDns](#) - Make DNS queries
- [File system API](#)

- **Functionality** – Deno uses these cookies so that we recognize you on our website and remember your previously selected preferences. These could include what language you prefer and location you are in. A mix of first-party and third-party cookies are used.
- **Understanding Usage** – Deno uses these cookies to collect information about your visit to our website, the content you viewed, the links you followed and information about your browser, device, and your IP address.

4.10.10 How to manage cookies

You can set your browser not to accept cookies, and the above website tells you how to remove cookies from your browser. However, in a few cases, some of our website features may not function as a result.

4.10.11 Privacy policies of other websites

The Deno website contains links to other websites. Our privacy policy applies only to deno.land and deno.com, so if you click on a link to another website, you should read their privacy policy.

4.10.12 Changes to our privacy policy

Deno keeps its privacy policy under regular review and places any updates on this web page. This privacy policy was last updated on December 2 2021.

4.10.13 How to contact us

If you have any questions about Deno's privacy policy, the data we hold on you, or you would like to exercise one of your data protection rights, please do not hesitate to contact us at privacy@deno.com.

4.11 Security & Responsible Disclosure

We consider the security of our systems, and all data controlled by those systems a top priority. No matter how much effort we put into system security, it is still possible that security vulnerabilities are present. We appreciate investigative work into system security carried out by well-intentioned, ethical security researchers. If you discover a vulnerability, however small, we would like to know about it so we can address it with appropriate measures, as quickly as possible. This page outlines the method we use to work with the security research community to address our system security.

4.11.1 Reporting a vulnerability

Please email you findings to deploy@deno.com. We strive to resolve all problems as quickly as possible, and are more than happy to play an active role in publication of writeups after the problem is resolved.

4.11.2 Please do the following:

- Do not take advantage of the vulnerability or problem you have discovered. For example only download data that is necessary to demonstrate the vulnerability - do not download any more. Also do not delete, modify, or view other people's data.
- Do not publish or reveal the problem until it has been resolved.
- Do not use attacks on physical security, social engineering, distributed denial of service, spam or applications of third parties.
- Do provide sufficient information to reproduce the problem, so we will be able to resolve it as quickly as possible. Usually, the IP address or the URL of the affected system and a description of the vulnerability will be sufficient, but complex vulnerabilities may require further explanation.

4.11.3 Our commitment

- If you act in accordance with this policy, we will not take legal action against you in regard to your report.
- We will handle your report with strict confidentiality, and not pass on your personal details to third parties without your permission.

```
└─ mod.ts
└─ style.css
```

The contents of mod.ts:

```
import { serve } from "https://deno.land/std@0.140.0/http/server.ts";
```

```
async function handleRequest(request: Request): Promise<Response> {
  const { pathname } = new URL(request.url);
```

```
  // This is how the server works:
  // 1. A request comes in for a specific asset.
  // 2. We read the asset from the file system.
  // 3. We send the asset back to the client.
```

```
  // Check if the request is for style.css.
  if (pathname.startsWith("/style.css")) {
    // Read the style.css file from the file system.
    const file = await Deno.readFile("./style.css");
    // Respond to the request with the style.css file.
    return new Response(file, {
      headers: {
        "content-type": "text/css",
      },
    });
  }
}
```

```
return new Response(
  `<html>
  <head>
    <link rel="stylesheet" href="style.css" />
  </head>
  <body>
    <h1>Example</h1>
  </body>
</html>`,
  {
    headers: {
      "content-type": "text/html; charset=utf-8",
    },
  },
);
}
```

serve(handleRequest);

The path provided to the Deno.readFile API is relative to the root of the repository. You can also specify absolute paths, if they are inside Deno.cwd.

1.4 Running scripts locally

For local development you can use the deno CLI. To install deno, follow the instructions in the [Deno manual](#).

After installation, you can run your scripts locally:

```
$ deno run --allow-net=:8000 https://deno.com/examples/hello.js
```

Listening on http://localhost:8000

To watch for file changes add the --watch flag:

```
$ deno run --allow-net=:8000 --watch ./main.js
```

Listening on http://localhost:8000

For more information about the Deno CLI, and how to configure your development environment and IDE, visit the Deno Manual's [Getting Started](#) section.

2 Tutorials

2.1 Connecting to Postgres

Postgres is a popular database for web applications because of its flexibility and ease of use. This guide will show you how to use Deno Deploy with Postgres.

- [Overview](#)
- [Setup Postgres](#)
- [Write and deploy the application](#)

2.1.1 Overview

We are going to build the API for a simple todo list application. It will have two endpoints: GET /todos will return a list of all todos, and POST /todos will create a new todo.

GET /todos

returns a list of all todos

```
[
  {
    "id": 1,
    "title": "Buy bread"
  },
  {
    "id": 2,
    "title": "Buy rice"
  },
  {
    "id": 3,
    "title": "Buy spices"
  }
]
```

POST /todos

creates a new todo

"Buy milk"

returns a 201 status code

In this tutorial, we will be:

- Creating and setting up a [Postgres](#) instance on [Supabase](#).
- Using a [Deno Deploy](#) Playground to develop and deploy the application.
- Testing our application using [cURL](#).

2.1.2 Setup Postgres

This tutorial will focus entirely on connecting to Postgres unencrypted. If you would like to use encryption with a custom CA certificate, use the documentation [here](#).

To get started we need to create a new Postgres instance for us to connect to. For this tutorial we will be using [Supabase](#) as they provide free, managed Postgres instances. If you like to host your database somewhere else, you can do that too.

1. Visit <https://app.supabase.io/> and click "New project".
2. Select a name, password, and region for your database. Make sure to save the password, as you will need it later.
3. Click "Create new project". Creating the project can take a while, so be patient.
4. Once the project is created, navigate to the "Database" tab on the left.
5. Go to the "Connection Pooling" settings, and copy the connection string from the "Connection String" field. This is the connection string you will use to connect to your database. Insert the password you saved earlier into this string, and then save the string somewhere - you will need it later.

4.10.2 How do we collect your data?

- Your email address is collected during registration to Deno Deploy.
- Usage information, cookies, and device information are collected automatically when viewing our websites.

4.10.3 How will we use your data?

Deno collects your data so that we can:

- Manage your account. Deno Deploy requires your email address for notifications, like when being invited an organization.
 - Analyze usage patterns of our products and services.
- Deno will not share your information with other parties.

4.10.4 How do we store your data?

Deno takes measures reasonably necessary to protect User Personal Information from unauthorized access, alteration, or destruction.

Deno will keep your email address indefinitely. Contact us at privacy@deno.com to request your information be deleted.

4.10.5 Marketing

Deno does not currently send out marketing information. However, Deno may in the future send you information about products and services of ours that we think you might like.

You have the right at any time to stop Deno from contacting you for marketing purposes. Please contact us at privacy@deno.com

4.10.6 What are your data protection rights?

Deno would like to make sure you are fully aware of all of your data protection rights. Every user is entitled to the following:

- The right to access – You have the right to request Deno for copies of your personal data.
- The right to rectification – You have the right to request that Deno correct any information you believe is inaccurate. You also have the right to request Deno to complete the information you believe is incomplete.
- The right to erasure – You have the right to request that Deno erase your personal data, under certain conditions.
- The right to restrict processing – You have the right to request that Our Company restrict the processing of your personal data, under certain conditions.
- The right to object to processing – You have the right to object to Our Company's processing of your personal data, under certain conditions.
- The right to data portability – You have the right to request that Deno transfer the data that we have collected to another organization, or directly to you, under certain conditions.
- If you make a request, we have one month to respond to you. If you would like to exercise any of these rights, please contact us at our email: privacy@deno.com

4.10.7 Cookies

Cookies are text files placed on your computer to collect standard Internet log information and visitor behavior information. When you visit our websites, we may collect information from you automatically through cookies or similar technology

For further information, visit allaboutcookies.org.

4.10.8 How do we use cookies?

Deno uses cookies in a range of ways to improve your experience on our website, including:

- Keeping you signed in
- Understanding how you use our website

4.10.9 What types of cookies do we use?

There are a number of different types of cookies, however, our website uses:

4.9 Fair Use Policy

The public beta for the Deno Deploy service includes resources (CPU time, request counts) that are subject to this Fair Use policy. This document can give a rough estimate to what we consider as "Fair Use", and what we do not.

4.9.1 Examples of Fair Use

- ✓ Server-side rendered websites
- ✓ Jamstack sites and apps
- ✓ Single page applications
- ✓ APIs that query a DB or external API
- ✓ A personal blog
- ✓ A company website
- ✓ An e-commerce site

4.9.2 Not Fair Use

- ✗ Crypto mining
- ✗ Highly CPU-intensive load (e.g. machine learning)
- ✗ Media hosting for external sites
- ✗ Scrapers
- ✗ Proxy or VPN

4.9.3 Guidelines

We expect most projects to fall well within the usage limits. We will notify you if your projects usage significantly deviates from the norm. We will reach out to you where possible before taking any action to address unreasonable burdens on our infrastructure.

4.10 Privacy Policy

Deno Land Inc is a corporation registered in Delaware, USA doing business as "Deno". This privacy policy will explain how our organization uses the personal data we collect from you when you use our website.

This privacy policy applies to all services provided at deno.com and deno.land.

Topics:

- What data do we collect?
- How do we collect your data?
- How will we use your data?
- How do we store your data?
- Marketing
- What are your data protection rights?
- What are cookies?
- How do we use cookies?
- What types of cookies do we use?
- How to manage your cookies
- Privacy policies of other websites
- Changes to our privacy policy
- How to contact us
- What data do we collect?

4.10.1 What data do we collect?

For users of Deno Deploy, we collect your email address and GitHub login. We also automatically collect from you your usage information, cookies, and device information, subject, where necessary, to your consent.

2.1.3 Write and deploy the application

We can now start writing our application. To start, we will create a new Deno Deploy playground in the control panel: press the "New Playground" button on <https://dash.deno.com/projects>. This will open up the playground editor. Before we can actually start writing code, we'll need to put our Postgres connection string into the environment variables. To do this, click on the project name in the top left corner of the editor. This will open up the project settings.

From here, you can navigate to the "Settings" -> "Environment Variable" tab via the left navigation menu. Enter "DATABASE_URL" into the "Key" field, and paste your connection string into the "Value" field. Now, press "Add". Your environment variables is now set.

Let's return back to the editor: to do this, go to the "Overview" tab via the left navigation menu, and press "Open Playground". Let's start by the std/http module so we can start serving HTTP requests:

```
import { serve } from "https://deno.land/std@0.140.0/http/server.ts";
```

```
serve(async (req) => {
  return new Response("Not Found", { status: 404 });
});
```

You can already save this code using Ctrl+S (or Cmd+S on Mac). You should see the preview page on the right refresh automatically: it now says "Not Found".

Next, let's import the Postgres module, read the connection string from the environment variables, and create a connection pool.

```
import { serve } from "https://deno.land/std@0.140.0/http/server.ts";
import * as postgres from "https://deno.land/x/postgres@v0.14.0/mod.ts";
```

```
// Get the connection string from the environment variable "DATABASE_URL"
const databaseUrl = Deno.env.get("DATABASE_URL")!;
```

```
// Create a database pool with three connections that are lazily established
const pool = new postgres.Pool(databaseUrl, 3, true);
```

```
serve(async (req) => {
```

Again, you can save this code now, but this time you should see no changes. We are creating a connection pool, but we are not actually running any queries against the database yet. Before we can do that, we need to set up our table schema.

We want to store a list of todos. Let's create a table called todos with an auto-increment id column and a title column:

```
const pool = new postgres.Pool(databaseUrl, 3, true);
```

```
// Connect to the database
const connection = await pool.connect();
try {
  // Create the table
  await connection.queryObject`
    CREATE TABLE IF NOT EXISTS todos (
      id SERIAL PRIMARY KEY,
      title TEXT NOT NULL
    )
  `;
} finally {
  // Release the connection back into the pool
  connection.release();
}
```



```
serve(async (req) => {
  Now that we have a table, we can add the HTTP handlers for the GET and POST endpoints.
  serve(async (req) => {
    // Parse the URL and check that the requested endpoint is /todos. If it is
    // not, return a 404 response.
    const url = new URL(req.url);
    if (url.pathname !== "/todos") {
      return new Response("Not Found", { status: 404 });
    }

    // Grab a connection from the database pool
    const connection = await pool.connect();
```

```
  try {
    switch (req.method) {
      case "GET": { // This is a GET request. Return a list of all todos.
        // Run the query
        const result = await connection.queryObject`
          SELECT * FROM todos
        `;

        // Encode the result as JSON
        const body = JSON.stringify(result.rows, null, 2);

        // Return the result as JSON
        return new Response(body, {
          headers: { "content-type": "application/json" },
        });
      }
      case "POST": { // This is a POST request. Create a new todo.
        // Parse the request body as JSON. If the request body fails to parse,
        // is not a string, or is longer than 256 chars, return a 400 response.
        const title = await req.json().catch(() => null);
        if (typeof title !== "string" || title.length > 256) {
          return new Response("Bad Request", { status: 400 });
        }

        // Insert the new todo into the database
        await connection.queryObject`
          INSERT INTO todos (title) VALUES (${title})
        `;

        // Return a 201 Created response
        return new Response("", { status: 201 });
      }
      default: // If this is neither a POST, or a GET return a 405 response.
        return new Response("Method Not Allowed", { status: 405 });
    }
  } catch (err) {
    console.error(err);
    // If an error occurs, return a 500 response
    return new Response(`Internal Server Error\n\n${err.message}`, {
```

4.6.1 When is compression skipped?

Deno Deploy skips the compression if:

- The response has [Content-Encoding](#) header.
- The response has [Content-Range](#) header.
- The response's [Cache-Control](#) header has [no-transform](#) value (e.g. cache-control: public, no-transform).

4.6.2 What happens to my Etag header?

When you set an Etag header with the response, we convert the header value to a Weak Etag if we apply compression to your response body. If it is already a Weak Etag, we don't touch the header.

4.7 Regions

Deno Deploy deploys your code throughout the world. Each new request is served from the closest region to your user. Deploy is presently located in the following regions:

1. Taiwan (asia-east1)	2. Hong Kong (asia-east2)
3. Tokyo (asia-northeast1)	4. Osaka (asia-northeast2)
5. Seoul (asia-northeast3)	6. Mumbai (asia-south1)
7. Delhi (asia-south2)	8. Singapore (asia-southeast1)
9. Jakarta (asia-southeast2)	10. Sydney (australia-southeast1)
11. Melbourne (australia-southeast2)	12. Warsaw (europe-central2)
13. Finland (europe-north1)	14. Belgium (europe-west1)
15. London (europe-west2)	16. Frankfurt (europe-west3)
17. Netherlands (europe-west4)	18. Zurich (europe-west6)
19. Milan (europe-west8)	20. Paris (europe-west9)
21. Madrid (europe-southwest1)	22. Montréal (northamerica-northeast1)
23. Toronto (northamerica-northeast2)	24. São Paulo (southamerica-east1)
25. Chile (southamerica-west1)	26. Iowa (us-central1)
27. South Carolina (us-east1)	28. North Virginia (us-east4)
29. Ohio (us-east5)	30. Texas (us-south1)
31. Oregon (us-west1)	32. California (us-west2)
33. Utah (us-west3)	34. Nevada (us-west4)

We will update the list as we add more regions.

4.8 Pricing & Limits

See [the pricing page](#) for the overview of the available features in Free and Pro plans. No uptime guarantees are provided during the initial public beta for Deno Deploy. Access to the service will be controlled by [our fair use policy](#). Any user we deem to be in violation of this policy, runs the risk of having their account terminated. During the initial public beta, the following hard limits apply. If any runtime limits are exceeded, all related requests will be immediately terminated, and a warning will be logged to the deployment's logs.

Feature	Free	Pro
Request count	100k req/day, 1000 req/min	\$2/million requests
Memory	512 MB	512 MB
CPU Time per request	10 ms	50 ms
Environment variable size	8 KB	8 KB
ES modules per deployment	1000	1000
Deployment script size	20 MB	20 MB
Deployments per hour	30	30
Custom domains	50	50
Crash Reports	100 crash reports per deployment, 100 logs per crash report	The same as Free
BroadcastChannel	64KB/sec send rate per isolate, no limit on receive	The same as Free

Isolate: an instance of your deployment running in any one of the [available regions](#). Isolates are created and destroyed on demand based on traffic to your deployment. If you have a use case that exceeds any of these limits, [please reach out](#).

Deno Deploy

Doing this will create a new GitHub repository containing the playground code. This project will be automatically turned into a git project that is linked to this new GitHub repository. Environment variables and domains will be retained.

The new GitHub repository will be created in your personal account, and will be set to private. You can change these settings later in the GitHub repository settings.

After exporting a playground, you can no longer use the Deno Deploy playground editor for this project. This is a one-way operation.

To export the playground visit the playground settings page in the Deno Deploy dashboard or select **Deploy: Export to GitHub** from the command palette (press F1 in the editor).

Here you can enter a name for the new GitHub repository. This name will be used to create the repository on GitHub. The repository must not already exist.

Press **Export** to export the playground to GitHub.

4.5 Organizations

Organizations allow you to collaborate with other users. A project created in an organization is accessible to all members of the organization. Users should first signup for Deno Deploy before they can be added to an organization.

Currently, all organization members have full access to the organization. They can add/remove members, and create/delete/modify all projects in the organization.

4.5.1 Create an organization

1. On your Deploy dashboard, click on the + **New** button in the navigation bar.
2. Select **New Organization**.
3. Enter a name for your organization and click on **Create**.

4.5.2 Add members

1. Select the desired organization in the organization dropdown in the top left of the screen, in the navigation bar.
2. Click on **Organization Settings**.
3. Under the **Members** panel, click on + **Invite member**.

Note: Users should first signup for Deno Deploy using [this link](#) before you invite them.

4. Enter the GitHub username of the user and click on **Invite**.

Deploy will send the user an invite email. They can then either accept or decline your invite. Once they accept the invite, they're added to your organization and shown in the members panel.

Pending invites are displayed in the **Invites** panel. You can revoke pending invites by clicking on the delete icon next to the pending invite.

4.5.3 Remove members

1. Select the desired organization in the organization dropdown in the top left of the screen, in the navigation bar.
2. Click on **Organization Settings**.
3. In the **Members** panel, click on the delete button beside the user you want to remove.

4.6 Compression

Compressing the response body to save bandwidth is a common practice. To take some work off your shoulder, we built the capabilities directly into Deploy.

Deno Deploy supports brotli and gzip compression. Compression is applied when the following conditions are met.

1. The request to your deployment has **Accept-Encoding** header set to either br (brotli) or gzip.
2. The response from your deployment includes the **Content-Type** header.
3. The provided content type is compressible; we use [this database](#) to determine if the content type is compressible.
4. The response body size is greater than 20 bytes.

When Deploy compresses the response body, it will set Content-Encoding: gzip or Content-Encoding: br header to the response based on the compression algorithm used.

```
status: 500,
});
} finally {
  // Release the connection back into the pool
  connection.release();
}
});
```

And there we go - application done. Deploy this code by saving the editor. You can now POST to the /todos endpoint to create a new todo, and you can get a list of all todos by making a GET request to /todos:

```
$ curl -X GET https://tutorial-postgres.deno.dev/todos
```

[↵]

```
$ curl -X POST -d "Buy milk" https://tutorial-postgres.deno.dev/todos
```

```
$ curl -X GET https://tutorial-postgres.deno.dev/todos
```

```
[
  {
    "id": 1,
    "title": "Buy milk"
  }
]
```

It's all working 🎉

The full code for the tutorial:

As an extra challenge, try add a DELETE /todos/:id endpoint to delete a todo. The [URLPattern](#) API can help with this.

2.2 Discord Slash Command

Discord has a new feature called **Slash Commands**. They allow you to type / followed by a command name to perform some action. For example, you can type /giphy cats (a built-in command) to get some cat gifs. It's pretty cool. :)

Discord Slash Commands work by making a request to a URL whenever someone issues a command. You don't need your app to be running all the time for Slash Commands to work which makes Deno Deploy a perfect solution to build such commands.

In this post let's see how we can build a hello world Slash Command using Deno Deploy.

1. [Create an application on Discord Developer Portal](#)
2. [Create and deploy the hello world Slash Command](#)
3. [Install the Slash Command on your Discord server](#)

2.2.1 Create an application on Discord Developer Portal

1. Go to <https://discord.com/developers/applications> (login using your discord account if required).
2. Click on **New Application** button available at left side of your profile picture.
3. Name your application and click on **Create**.
4. Go to **Bot** section, click on **Add Bot**, and finally on **Yes, do it!** to confirm.

That's it. A new application is created which will hold our Slash Command. Don't close the tab as we need information from this application page throughout our development.

Before we can write some code, we need to curl a discord endpoint to register a Slash Command in our app.

Fill BOT_TOKEN with the token available in the **Bot** section and CLIENT_ID with the ID available on the **General Information** section of the page and run the command on your terminal.

```
BOT_TOKEN='replace_me_with_bot_token'
```

```
CLIENT_ID='replace_me_with_client_id'
```

```
curl -X POST \
```

```
-H 'Content-Type: application/json' \
-H "Authorization: Bot $BOT_TOKEN" \
-d '{"name":"hello","description":"Greet a person","options":[{"name":"name","description":"The name of the person","type":3,"required":true}]}' \
"https://discord.com/api/v8/applications/$CLIENT_ID/commands"
This will register a Slash Command named hello that accepts a parameter named name of type string.
```

2.2.2 Create and deploy the hello world Slash Command

To brief up: we created an application on Discord Developer Portal, registered a Slash Command, and created a project in Deno Deploy dashboard.

Whenever someone issues a command, Discord makes a POST request with the payload in JSON format. We need to respond to Discord in the same JSON format. The below code demonstrates the example that responds with a greeting.

```
// Sift is a small routing library that abstracts away details like starting a
// listener on a port, and provides a simple function (serve) that has an API
// to invoke a function for a specific path.
```

```
import {
  json,
  serve,
  validateRequest,
} from "https://deno.land/x/sift@0.6.0/mod.ts";
// TweetNaCl is a cryptography library that we use to verify requests
// from Discord.
import nacl from "https://cdn.skypack.dev/tweetnacl@v1.0.3?dts";
```

```
// For all requests to "/" endpoint, we want to invoke home() handler.
serve({
  "/": home,
});
```

// The main logic of the Discord Slash Command is defined in this function.

```
async function home(request: Request) {
  // validateRequest() ensures that a request is of POST method and
  // has the following headers.
  const { error } = await validateRequest(request, {
    POST: {
      headers: ["X-Signature-Ed25519", "X-Signature-Timestamp"],
    },
  });
  if (error) {
    return json({ error: error.message }, { status: error.status });
  }
}
```

```
// verifySignature() verifies if the request is coming from Discord.
// When the request's signature is not valid, we return a 401 and this is
// important as Discord sends invalid requests to test our verification.
const { valid, body } = await verifySignature(request);
if (!valid) {
  return json(
    { error: "Invalid request" },
    {
      status: 401,
    },
  );
}
```

If you chose **GitHub Actions** mode, you will now need to add the Deno Deploy GitHub Action to your workflow file. You can find an explanation on how to do this in the Deno Deploy dashboard, and the [documentation for the Deno Deploy Action](#).

4.3.5.2 Disabling

You can disable the Git integration by pressing the **Unlink** button on the Git settings page of your project.

4.4 Playgrounds

Playgrounds are an easy way to play around with Deno Deploy, and to create small projects. Using playgrounds you can write code, run it, and see the output fully inside the browser.

Playgrounds have the full power of Deno Deploy: they support all the same features as a normal project, including environment variables, custom domains, logs, and crash reports.

Playgrounds are also just as performant as all other projects on Deno Deploy: they make full use of our global network to run your code as close to users as possible.

- [Creating a playground](#)
- [Using the playground editor](#)
- [Making a playground public](#)
- [Exporting a playground to GitHub](#)

4.4.1 Creating a playground

To create a new playground press the **New Playground** button in the top right corner of the [project overview page](#).

This will create a new playground with a randomly generated name. You can change this name in the project settings later.

4.4.2 Using the playground editor

The playground editor is opened automatically when you create a new playground. You can also open it by navigating to your project's overview page and clicking the **Edit** button.

The editor consists of two main areas: the editor on the left, and the preview panel on the right. The editor is where you write your code, and the preview panel is where you can see the output of your code through a browser window.

There is also a logs panel underneath the editor panel on the left side. This panel shows the console output of your code, and is useful for debugging your code.

After editing your code, you need to save and deploy it so the preview on the right updates. You can do this by clicking the **Save & Deploy** button in the top right, by pressing Ctrl + S, or opening the command palette with F1 and selecting **Deploy: Save & Deploy**.

In the tool bar in the top right of the editor you can see the current deployment status of your project while saving.

The preview panel on the right will refresh automatically every time you save and deploy your code.

The language dropdown in the top right of the editor allows you to switch between JavaScript, JSX, TypeScript, and TSX. The default selected language is TSX which will work for most cases.

4.4.3 Making a playground public

Playgrounds can be shared with other users by making them public. This means that anyone can view the playground and its preview. Public playgrounds can not be edited by anyone: they can still only be edited by you. Logs are also only shown to you. Users have the option to fork a public playground to make a private copy of it that they can edit.

To make a playground public, press the **Share** button in the top tool bar in the editor. The URL to your playground will be copied to your clipboard automatically.

You can also change the playground visibility from the playground settings page in the Deno Deploy dashboard. This can be used to change the visibility of a playground from public to private again.

4.4.4 Exporting a playground to GitHub

Playgrounds can be exported to GitHub. This is useful if your project is starting to outgrow the single file limit of the playground editor.

Deno Deploy

To update DNS records of your domain, go to the DNS configuration panel of your domain registrar (or the service you're using to manage DNS) and enter the records as described on the domain setup page. Once you have the DNS records updated, click the "Validate" button on the domain setup page. It will check if the DNS records are correctly set and updates the status to "Validated, needs a certificate".

At this point you have two options: let us automatically provision a certificate using Let's Encrypt, or you can manually upload a certificate and private key.

To automatically provision a certificate, press the "Provision Automatically" button. Provisioning a TLS certificate can take up to a minute. It is possible that the provisioning fails if your domain specifies a CAA record that prevents [Let's Encrypt](#) from provisioning certificates. Certificates will be automatically renewed around 30 days before the certificate expires.

To manually upload a certificate chain and private key, press the "Upload Manually" button. You will be prompted to upload a certificate chain and private key. The certificate chain needs to be complete and valid, and your leaf certificate needs to be at the top of the chain.

4.3.4 Environment Variables

Environment variables are useful to store values like access tokens of web services. You can create them in the project dashboard and access them in your code via the Deno.env API. They are made available to both production and preview deployments.

To add an environment variable to your project, click on the "Settings" button on the project page and then on "Environment Variables" from the sidebar. Fill in the key/value fields and click on "Add" to add an environment variable to your project. A new production deployment will be created automatically with the new environment variables.

4.3.4.1 Preset Variables

Every deployment has the following environment variables preset, which you can access from your code.

1. `DENO_REGION`

It holds the region code of the region in which the deployment is running. You can use this variable to serve region-specific content.

You can refer to the region code from the [regions page](#).

1. `DENO_DEPLOYMENT_ID`

It holds the ID of the deployment.

4.3.5 Git Integration

The Git integration enables deployment of code changes that are pushed to a GitHub repository. Commits on the production branch will be deployed as a production deployment. Commits on all other branches will be deployed as a preview deployment.

There are two modes of operation for the Git integration:

- **Automatic:** Deno Deploy will automatically pull code and assets from your repository source every time you push, and deploy it. This mode is very fast, but does not allow for a build step. This is the recommended mode for most users.

- **GitHub Actions:** In this mode you push your code and assets to Deno Deploy from a GitHub Actions workflow. This allows you to perform a build step before deploying.

4.3.5.1 Enabling

To enable the Git integration, follow these steps:

1. Click on the **Settings** button on the project page and then select **Git** from the sidebar.
2. Select your organization name, and repository. If your repository or organization does not show up, make sure the [Deno Deploy GitHub App](#) is installed on your repository.
3. Select a production branch. Code deployed from this branch will be deployed as a production deployment instead of a preview deployment.
4. Choose either **Automatic** or **GitHub Actions** deployment mode. The difference between these modes is explained above.
5. Depending on the mode, do the following:
 - If you chose **Automatic**, select the endpoint for your project.
 - If you chose **GitHub Actions**, press the **Link** button.

```

    );
  }

  const { type = 0, data = { options: [] } } = JSON.parse(body);
  // Discord performs Ping interactions to test our application.
  // Type 1 in a request implies a Ping interaction.
  if (type === 1) {
    return json({
      type: 1, // Type 1 in a response is a Pong interaction response type.
    });
  }

  // Type 2 in a request is an ApplicationCommand interaction.
  // It implies that a user has issued a command.
  if (type === 2) {
    const { value } = data.options.find((option) => option.name === "name");
    return json({
      // Type 4 responds with the below message retaining the user's
      // input at the top.
      type: 4,
      data: {
        content: `Hello, ${value}!`,
      },
    });
  }

  // We will return a bad request error as a valid Discord request
  // shouldn't reach here.
  return json({ error: "bad request" }, { status: 400 });
}

/** Verify whether the request is coming from Discord. */
async function verifySignature(
  request: Request,
): Promise<{ valid: boolean; body: string }> {
  const PUBLIC_KEY = Deno.env.get("DISCORD_PUBLIC_KEY")!;
  // Discord sends these headers with every request.
  const signature = request.headers.get("X-Signature-Ed25519")!;
  const timestamp = request.headers.get("X-Signature-Timestamp")!;
  const body = await request.text();
  const valid = nacl.sign.detached.verify(
    new TextEncoder().encode(timestamp + body),
    hexToUint8Array(signature),
    hexToUint8Array(PUBLIC_KEY),
  );

  return { valid, body };
}


```

```

/** Converts a hexadecimal string to Uint8Array. */
function hexToUint8Array(hex: string) {
  return new Uint8Array(hex.match(/.{1,2}/g)!.map((val) => parseInt(val, 16)));
}

```

2.2.2.1 Deploy the code

1. Click on the button below and you'll be navigated to Deno Deploy. 
2. Input DISCORD_PUBLIC_KEY env variable field. The value should be the public key available in **General Information** section in the Discord application page.
3. Click on **Create** to create the project, then on **Deploy** to deploy the script.
4. Grab the URL that's displayed under **Domains** in Production Deployment card.

2.2.2.2 Configure Discord application to use our URL as interactions endpoint URL

1. Go back to your application (Greeter) page on Discord Developer Portal
2. Fill **INTERACTIONS ENDPOINT URL** field with the URL and click on **Save Changes**.

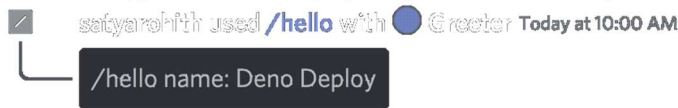
The application is now ready. Let's proceed to the next section to install it.

2.2.3 Install the Slash Command on your Discord server

So to use the hello Slash Command, we need to install our Greeter application on our Discord server.

Here are the steps:

1. Go to **OAuth2** section of the Discord application page on Discord Developer Portal
 2. Select applications.commands scope and click on the **Copy** button below.
 3. Now paste and visit the URL on your browser. Select your server and click on **Authorize**.
- Open Discord, type /hello Deno Deploy and press **Enter**. The output will look something like below.



Congratulations for completing the tutorial! Go ahead and build some awesome Discord Slash Commands! And do share them with us on **deploy** channel of [the Deno Discord server](#).

2.3 Persist data using FaunaDB

FaunaDB calls itself "The data API for modern applications". It's a database with a GraphQL interface that enables you to use GraphQL to interact with it. Since we communicate with it using HTTP requests, we don't need to manage connections which suits very well for serverless applications.

The tutorial assumes that you've [FaunaDB](#) and Deno Deploy accounts, Deno Deploy CLI installed, and some basic knowledge of GraphQL.

- [Overview](#)
- [Build the API Endpoints](#)
- [Use FaunaDB for Persistence](#)
- [Deploy the API](#)

2.3.1 Overview

In this tutorial, let's build a small quotes API with endpoints to insert and retrieve quotes. And later use FaunaDB to persist the quotes.

Let's start by defining the API endpoints.

A POST request to the endpoint should insert the quote to the list.

POST /quotes/

Body of the request.

```
{
  "quote": "Don't judge each day by the harvest you reap but by the seeds that you plant.",
  "author": "Robert Louis Stevenson"
}
```

with:

```
project: my-project
entrypoint: index.js
root: dist
```

The entrypoint can either be a relative path or file name, or a an absolute URL. If it is a relative path, it will be resolved relative to the root. Both absolute file:/// and https:// URLs are supported.

To deploy the ./dist directory using the [std/http/file_server.ts](#) module, you can use the following configuration:

```
- name: Deploy to Deno Deploy
  uses: denoland/deployctl@v1
  with:
    project: my-project
    entrypoint: https://deno.land/std/http/file_server.ts
    root: dist
```

See [deployctl README](#) for more details.

4.3 Projects

Projects are the most central resource in Deno Deploy. They group [deployments](#), [custom domains](#), and [environment variables](#). Projects can be linked to a GitHub repository to enable the Deno Deploy [Git integration](#).

- [Creating a Project](#)
- [Settings](#)
- [Domains](#)
- [Environment Variables](#)
- [Git Integration](#)

4.3.1 Creating a Project

To create a project click on the **New Project** button on the Deno Deploy dashboard, or navigate to <https://dash.deno.com/new>.

Project names must be between 3 and 24 characters long, only contain a-z, 0-9 and -, and must not start or end with a hyphen (-). The project name will determine a project's production URL. It has the form \$PROJECT_ID.deno.dev (e.g. <https://hello.deno.dev>). The production URL is the URL that your production deployment can be reached at.

4.3.2 Settings

In the project settings a project can be renamed, transferred to a different organization, or deleted. When renaming a project, the production URL of the project will be changed and traffic to the old URL will be dropped.

To transfer a project to a different organization select the receiving organization from the dropdown menu and press **Transfer**. You need to be a member of the receiving organization to transfer a project. Deleting a project also deletes all linked custom domains, environment variables and deployments.

4.3.3 Domains

By default a deployment can be reached by it's preview URL, or by the project URL if the project has a production deployment. These domains always end in .deno.dev. It is also possible to use custom domains to serve traffic.

You must own the domain that you want to add to a project. If you do not own a domain yet, you can register one at a domain registrar like Google Domains, Namecheap, or gandi.net.

4.3.3.1 Adding a Domain

Domains can be added in the domain settings page of a project. To navigate there, click the "Settings" button on the project page, then select "Domains" from the sidebar.

Enter the domain name you wish to add to the project and press "Add". The domain is added to the domains list and will have a "Needs setup" badge. Click on the "Needs setup" badge to visit the domain setup page which will display the list of DNS records that need to be created/updated for your domain.

4.1.4 Crash Reports

When a deployment crashes (you get a 502 status code as a response) due to an uncaught exception or uncaught rejection in your code, or we terminate the deployment due to overconsumption of resources, we generate a crash report containing the last 100 logs from your deployment.

We generate new crash reports for a deployment if we see errors different from the previously generated crash reports. If same crash report is generated in different regions, we show you the crash report of the first region. We retain the latest hundred crash reports per deployment. Old crash reports are deleted in order of age when you reach the limit of 100 crash reports for a deployment.

4.2 deployctl

deployctl is a command line tool for deploying your code to Deno Deploy, and it's also a GitHub

Action for the same purpose.

The default GitHub integration of Deno Deploy automatically uploads the files in your repository as [Static Assets](#) and you can access those files in Deno Deploy. However this static assets support is limited to the files in your repository with the default GitHub integration.

If you like to **dynamically** generate the static assets during the build steps, you need to use deployctl.

4.2.1 deployctl CLI

You can install deployctl command with the below command:

```
deno install --allow-read --allow-write --allow-env --allow-net --allow-run --no-check -r -f
```

<https://deno.land/x/deploy/deployctl.ts>

You also need to set your Personal Access Token to `DENO_DEPLOY_TOKEN` environment variable.

You can generate your Personal Access Token in <https://dash.deno.com/user/access-tokens>.

4.2.2 Usages

To deploy a local script:

```
deployctl deploy --project=helloworld main.ts
```

To deploy a remote script:

```
deployctl deploy --project=helloworld https://deno.com/examples/hello.js
```

To deploy a remote script without static files:

```
deployctl deploy --project=helloworld --no-static https://deno.com/examples/hello.js
```

To ignore the node_modules directory while deploying:

```
deployctl deploy --project=helloworld --exclude=node_modules main.tsx
```

See the help message (deployctl -h) for more details.

4.2.3 deployctl GitHub Action

There's also [deployctl GitHub Action](#) which wraps and automates the above command line usages.

You just need to include the deployctl GitHub Action as a step in your workflow.

You do **not** need to set up any secrets for this to work. You **do** need to link your GitHub repository to your Deno Deploy project and choose the "GitHub Actions" deployment mode. You can do this in your project settings on <https://dash.deno.com>.

job:

```
permissions:
  id-token: write # This is required to allow the GitHub Action to authenticate with Deno Deploy.
  contents: read
steps:
  - name: Deploy to Deno Deploy
    uses: denoland/deployctl@v1
    with:
      project: my-project # the name of the project on Deno Deploy
      entrypoint: main.ts # the entrypoint to deploy
```

By default the entire contents of the repository will be deployed. This can be changed by specifying the root option.

```
- name: Deploy to Deno Deploy
  uses: denoland/deployctl@v1
```

A GET request to the endpoint should return all the quotes from the database.

GET /quotes/

Response of the request.

```
{
  "quotes": [
    {
      "quote": "Don't judge each day by the harvest you reap but by the seeds that you plant.",
      "author": "Robert Louis Stevenson"
    }
  ]
}
```

Now that we understand how the endpoint should behave, let's proceed to build it.

2.3.2 Build the API Endpoints

First, create a file named quotes.ts and paste the following content.

Read through the comments in the code to understand what's happening.

```
import {
  json,
  serve,
  validateRequest,
} from "https://deno.land/x/sift@0.6.0/mod.ts";
```

```
serve({
  "/quotes": handleQuotes,
});
```

// To get started, let's just use a global array of quotes.

```
const quotes = [
  {
    quote: "Those who can imagine anything, can create the impossible.",
    author: "Alan Turing",
  },
  {
    quote: "Any sufficiently advanced technology is equivalent to magic.",
    author: "Arthur C. Clarke",
  },
];
```

```
async function handleQuotes(request: Request) {
  // Make sure the request is a GET request.
  const { error } = await validateRequest(request, {
    GET: {},
  });
  // validateRequest populates the error if the request doesn't meet
  // the schema we defined.
  if (error) {
    return json({ error: error.message }, { status: error.status });
  }

  // Return all the quotes.
  return json({ quotes });
}
```

Run the above program using [the Deno CLI](#).

```

Deno Deploy
deno run --allow-net=:8000 ./path/to/quotes.ts
# Listening on http://0.0.0.0:8000/
And curl the endpoint to see some quotes.
curl http://127.0.0.1:8000/quotes
# {"quotes":[
# {"quote":"Those who can imagine anything, can create the impossible.", "author":"Alan Turing"},
# {"quote":"Any sufficiently advanced technology is equivalent to magic.", "author":"Arthur C. Clarke"}
# ]}

```

Let's proceed to handle the POST request.

Update the `validateRequest` function to make sure a POST request follows the provided body scheme.

```

- const { error } = await validateRequest(request, {
+ const { error, body } = await validateRequest(request, {
  GET: {},
  POST: {
    body: ["quote", "author"]
  }
});

```

Handle the POST request by updating `handleQuotes` function with the following code.

```

async function handleQuotes(request: Request) {
  const { error, body } = await validateRequest(request, {
    GET: {},
    POST: {
      body: ["quote", "author"],
    },
  });
  if (error) {
    return json({ error: error.message }, { status: error.status });
  }
}

```

```

+ // Handle POST requests.
+ if (request.method === "POST") {
+   const { quote, author } = body as { quote: string; author: string };
+   quotes.push({ quote, author });
+   return json({ quote, author }, { status: 201 });
+ }

```

```

  return json({ quotes });
}

```

Let's test it by inserting some data.

```

curl --dump-header - --request POST --data '{"quote": "A program that has not been tested does not
work.", "author": "Bjarne Stroustrup"}' http://127.0.0.1:8000/quotes

```

The output might look like something below.

```

HTTP/1.1 201 Created
transfer-encoding: chunked
content-type: application/json; charset=utf-8

```

```

{"quote":"A program that has not been tested does not work.", "author":"Bjarne Stroustrup"}
Awesome! We built our API endpoint, and it's working as expected. Since the data is stored in memory,
it will be lost after a restart. Let's use FaunaDB to persist our quotes.

```

2.3.3 Use FaunaDB for Persistence

Let's define our database schema using GraphQL Schema.

We're creating a new type named `Quote` to represent a quote and its author.

4 Platform

4.1 Deployments

A deployment is a snapshot of all code and environment variables required to run an application.

Deployments are immutable after they have been created. To deploy a new version of the code for an application, a new deployment must be created.

All deployments have a preview URL that can be used to view this specific deployment. Preview URLs have the format `{project_name}-{deployment_id}.deno.dev`. There can also be other URLs that can point to a deployment, like custom domains or a project URL. These are explained in the [Projects](#) chapter.

- [Creating Deployments](#)
- [Git Integration](#)
- [Production Vs. Preview Deployments](#)
- [Logs](#)
- [Crash Reports](#)

4.1.1 Creating Deployments

There are multiple ways to create a deployment: through [the GitHub integration](#),

through [deployctl](#) command (or GitHub Action), or through a [playground](#). In all cases the environment variables are copied from the project at the time of creation. Changing environment variables in a project will not have any impact on the environment variables of previously created deployments.

4.1.1.1 Playgrounds

The easiest way to deploy some code, is via a Deno Deploy playground. You can learn more about the playgrounds [here](#).

4.1.1.2 Git Integration

The simplest way to create deployments for more complex projects is to link a project to a GitHub repository (more about this [projects chapter](#)). In this case, whenever a new commit is pushed to the linked repository, we will automatically pull the application code from the repository and create a new deployment. The status of this process is reported via status check on the commit. This status check can be viewed in the GitHub UI, and it contains a link to the preview URL for that deployment. If a deployment fails (for example, Deno Deploy could not download a dependency), we will show the error in the GitHub status check of the commit and add a comment to the relevant commit in GitHub.

For pushes to the production branch of your repository (usually `main` or the one chosen by you during the link process), we will create a **Production Deployment**. Pushes to any other branch will create a **Preview Deployment**.

4.1.1.3 [deployctl](#)

`deployctl` is a command line tool for deploying your code to Deno Deploy, and it's also a GitHub Action for the same purpose. You can control more details of your deployment than the above automatic GitHub integration by using `deployctl`. See [the `deployctl` page](#) for more details.

4.1.2 Production vs Preview Deployments

A deployment can either be a production or a preview deployment. These deployments do not have any differences in runtime functionality. The only distinguishing factor is that a project's production deployment will receive traffic from the project URL (e.g. `myproject.deno.dev`), and from custom domains in addition to traffic to the deployment's preview URL.

4.1.3 Logs

Applications can generate logs at runtime using the console API. These logs can be viewed in real time by navigating to the Logs panel of a project or deployment. Logs will be streamed directly from an application to the log panel.

These logs are **not persisted**. Only logs that are generated after the logs page is opened can be viewed. After closing the logs page, all streamed logs are discarded.

Log messages have a maximum size of 2kb. Messages larger than this are trimmed to 2kb.

```

Deno Deploy
  headers: { "content-type": "text/html; charset=utf-8" },
});
}

case "POST": {
  const body = await req.formData();
  const name = body.get("name") || "anonymous";
  return new Response(`Hello ${name}!`);
}

default:
  return new Response("Invalid method", { status: 405 });
}
}

```

3.1.6 [Proxying to other servers](#)

A HTTP server that proxies requests to a different server.

```

async function handler(req: Request): Promise<Response> {
  const url = new URL(req.url);
  url.protocol = "https:";
  url.hostname = "example.com";
  url.port = "443";
  return await fetch(url.href, {
    headers: req.headers,
    method: req.method,
    body: req.body,
  });
}

```

3.1.7 [Server side rendering with JSX](#)

A HTTP server that renders a HTML page on the server with JSX (using Preact).

```

import { h } from "https://esm.sh/preact@10.5.15";
import { renderToString } from "https://esm.sh/preact-render-to-string@5.1.19?deps=preact@10.5.15";

```

```

function handler(_req: Request): Response {
  const page = (
    <div>
      <h1>Current time</h1>
      <p>{new Date().toLocaleString()}</p>
    </div>
  );
  const html = renderToString(page);
  return new Response(html, {
    headers: { "content-type": "text/html; charset=utf-8" },
  });
}

```

3.1.8 [Wildcard Domain](#)

A HTTP server that serves a wildcard domain.

```

function handler(req: Request) {
  const url = new URL(req.url);
  if (url.hostname === "a.example.com") {
    return new Response("website 1");
  } else if (url.hostname === "b.example.com") {
    return new Response("website 2");
  }
  return new Response("website infinity");
}

```

```

type Quote {
  quote: String!
  author: String!
}

```

```

type Query {
  # A new field in the Query operation to retrieve all quotes.
  allQuotes: [Quote!]
}

```

Fauna has a graphql endpoint for its database, and it generates essential mutations like create, update, delete for a data type defined in the schema. For example, fauna will generate a mutation named createQuote to create a new quote in the database for the data type Quote. And we're additionally defining a query field named allQuotes that returns all the quotes in the database.

Let's get to writing the code to interact with fauna from Deno Deploy applications.

To interact with fauna, we need to make a POST request to its graphql endpoint with appropriate query and parameters to get the data in return. So let's construct a generic function that will handle those things.

```

async function queryFauna(
  query: string,
  variables: { [key: string]: unknown },
): Promise<{
  data?: any;
  error?: any;
}> {
  // Grab the secret from the environment.
  const token = Deno.env.get("FAUNA_SECRET");
  if (!token) {
    throw new Error("environment variable FAUNA_SECRET not set");
  }
}

```

```

try {
  // Make a POST request to fauna's graphql endpoint with body being
  // the query and its variables.
  const res = await fetch("https://graphql.fauna.com/graphql", {
    method: "POST",
    headers: {
      authorization: `Bearer ${token}`,
      "content-type": "application/json",
    },
    body: JSON.stringify({
      query,
      variables,
    }),
  });
}

```

```

const { data, errors } = await res.json();
if (errors) {
  // Return the first error if there are any.
  return { data, error: errors[0] };
}

```

```

return { data };
} catch (error) {

```



```

    return { error };
  }
}

```

Add this code to the quotes.ts file. Now let's proceed to update the endpoint to use fauna.

```

async function handleQuotes(request: Request) {
  const { error, body } = await validateRequest(request, {
    GET: {},
    POST: {
      body: ["quote", "author"],
    },
  });
  if (error) {
    return json({ error: error.message }, { status: error.status });
  }

  if (request.method === "POST") {
    const { quote, author, error } = await createQuote(
      body as { quote: string; author: string }
    );
    if (error) {
      return json({ error: "couldn't create the quote" }, { status: 500 });
    }

    return json({ quote, author }, { status: 201 });
  }

  return json({ quotes });
}

```

```

+async function createQuote({
+  quote,
+  author,
+}): {
+  quote: string;
+  author: string;
+}): Promise<{ quote?: string; author?: string; error?: string }> {
+  const query = `
+    mutation($quote: String!, $author: String!) {
+      createQuote(data: { quote: $quote, author: $author }) {
+        quote
+        author
+      }
+    }
+  `;
+
+  const { data, error } = await queryFauna(query, { quote, author });
+  if (error) {
+    return { error };
+  }
+
+  return data;
+}

```

3 Examples

3.1 Examples Gallery

```
import { serve } from "https://deno.land/std@0.140.0/http/server.ts";
```

```
// codes
```

```
serve(handler);
```

A collection of code snippets demonstrating certain functions of Deno Deploy.

3.1.1 [Hello World](#)

A HTTP server that responds with a static "Hello World" string to all requests.

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

3.1.2 [Respond with JSON](#)

A HTTP server that responds to requests with pretty printed JSON.

```
function handler(_req: Request) {
  const data = {
    Hello: "World!",
  };
  const body = JSON.stringify(data, null, 2);
  return new Response(body, {
    headers: { "content-type": "application/json; charset=utf-8" },
  });
}
```

3.1.3 [Redirects](#)

A HTTP server that that redirects users to https://example.com

```
function handler(_req: Request) {
  return Response.redirect("https://example.com", 307);
}
```

3.1.4 [Get client IP address](#)

A HTTP server that responds to requests with the client's IP address.

```
function handler(_req: Request, connInfo: ConnInfo) {
  const addr = connInfo.remoteAddr as Deno.NetAddr;
  const ip = addr.hostname;
  return new Response(`Your IP address is <b>${ip}</b>`, {
    headers: { "content-type": "text/html" },
  });
}
```

3.1.5 [Handling <form> submissions](#)

A HTTP server that serves a HTML form and handles the form submission via a POST request.

```
const html = `
<form method="POST" action="/">
  <input type="text" name="name" placeholder="Your name">
  <button type="submit">Submit</button>
</form>
`;
```

```
async function handler(req: Request): Promise<Response> {
  switch (req.method) {
    case "GET": {
      return new Response(html, {
```

Deno Deploy

appId: "APPID",
};
You would need to set the value of the string to this (noting that spacing and new lines are not required):
{
 "apiKey": "APIKEY",
 "authDomain": "example-12345.firebaseio.com",
 "projectId": "example-12345",
 "storageBucket": "example-12345.appspot.com",
 "messagingSenderId": "1234567890",
 "appId": "APPID"
}

2.5.6 Deploy the application

Now let's deploy the application:
1. Go to <https://gist.github.com/new> and create a new gist, ensuring the filename of the gist ends with .js.
For convenience the whole application is hosted at <https://deno.com/examples/firebase.js>. You can skip creating a gist if you want to try the example without any modification, or click the link at the bottom of the tutorial.
2. Copy the Raw link of the saved gist.
3. In your project on dash.deno.com, click the **Deploy URL** button and enter the link to the raw gist in the URL field.
4. Click the **Deploy** button and copy one of the URLs displayed in the **Domains** section of the project panel.
Now let's take our API for a spin.
We can create a new song:
curl --request POST \
 --header "Content-Type: application/json" \
 --data '{"title": "Old Town Road", "artist": "Lil Nas X", "album": "7", "released": "2019", "genres": "Country rap, Pop"}' \
 --dump-header \
 - https://<project_name>.deno.dev/songs
And we can get all the songs in our collection:
curl https://<project_name>.deno.dev/songs
And we get specific information about a title we created:
curl https://<project_name>.deno.dev/songs/Old%20Town%20Road

Deno Deploy

Now that we've updated the code to insert new quotes let's set up a fauna database before proceeding to test the code.
Create a new database:
1. Go to <https://dashboard.fauna.com> (login if required) and click on **New Database**
2. Fill the **Database Name** field and click on **Save**.
3. Click on **GraphQL** section visible on the left sidebar.
4. Create a file ending with .gql extension with the content being the schema we defined above.
Generate a secret to access the database:
1. Click on **Security** section and click on **New Key**.
2. Select **Server** role and click on **Save**. Copy the secret.
Let's now run the application with the secret.
FAUNA_SECRET=<the_secret_you_just_obtained> deno run --allow-net=:8000 --watch quotes.ts
Listening on http://0.0.0.0:8000
curl --dump-header - --request POST --data '{"quote": "A program that has not been tested does not work.", "author": "Bjarne Stroustrup"}' http://127.0.0.1:8000/quotes
Notice how the quote was added to your collection in FaunaDB.
Let's write a new function to get all the quotes.
async function getAllQuotes() {
 const query = `
 query {
 allQuotes {
 data {
 quote
 author
 }
 }
 }
 `;

 const {
 data: {
 allQuotes: { data: quotes },
 },
 error,
 } = await queryFauna(query, {});
 if (error) {
 return { error };
 }

 return { quotes };
}

And update the handleQuotes function with the following code.
-// To get started, let's just use a global array of quotes.
-const quotes = [
- {
- quote: "Those who can imagine anything, can create the impossible.",
- author: "Alan Turing",
- },
- {
- quote: "Any sufficiently advanced technology is equivalent to magic.",
- author: "Arthur C. Clarke",
- },
-

-];

```

async function handleQuotes(request: Request) {
  const { error, body } = await validateRequest(request, {
    GET: {},
    POST: {
      body: ["quote", "author"],
    },
  });
  if (error) {
    return json({ error: error.message }, { status: error.status });
  }

  if (request.method === "POST") {
    const { quote, author, error } = await createQuote(
      body as { quote: string; author: string },
    );
    if (error) {
      return json({ error: "couldn't create the quote" }, { status: 500 });
    }

    return json({ quote, author }, { status: 201 });
  }

  // It's assumed that the request method is "GET".
  {
    const { quotes, error } = await getAllQuotes();
    if (error) {
      return json({ error: "couldn't fetch the quotes" }, { status: 500 });
    }

    return json({ quotes });
  }
}

```

```
curl http://127.0.0.1:8000/quotes
```

You should see all the quotes we've inserted into the database. The final code of the API is available at <https://deno.com/examples/fauna.ts>.

2.3.4 Deploy the API

The process of deploying the API involves creating a new Deno Deploy project and a secret to hold our FaunaDB secret.

Create a project and a secret:

1. Go to <https://dash.deno.com/new> (Sign in with GitHub if you didn't already) and click on **Create**.
2. Now click on **Settings** button available on the project page.
3. Navigate to **Environment Variables** Section and add the following secrets.

- **FAUNA_SECRET** - The value should be the secret we created in the previous step or a new one.

Don't close this tab yet.

Deploy the code:

1. Create a gist (make sure the extension of the file is .ts) at <https://gist.github.com/new> with your code and grab the raw link of it.

For convenience, the code is also hosted at <https://deno.com/examples/fauna.ts> so you can skip creating a gist if you just want to try out the above example without making changes to it.

2. Go back to Deno Deploy **Settings** screen where we created our secrets.

```

const songsRef = db.collection("songs");
await songsRef.add(song);
ctx.response.status = Status.NoContent;
});

```

Ok, we are almost done. We just need to create our middleware application, and add the localStorage middleware we imported:

```

const app = new Application();
app.use(virtualStorage());

```

And then we need to add middleware to authenticate the user. In this tutorial we are simply grabbing the username and password from the environment variables we will be setting up, but this could easily be adapted to redirect a user to a sign-in page if they are not logged in:

```

app.use(async (ctx, next) => {
  const signedInUid = ctx.cookies.get("LOGGED_IN_UID");
  const signedInUser = signedInUid != null ? users.get(signedInUid) : undefined;
  if (!signedInUid || !signedInUser || !auth.currentUser) {
    const creds = await auth.signInWithEmailAndPassword(
      Deno.env.get("FIREBASE_USERNAME"),
      Deno.env.get("FIREBASE_PASSWORD"),
    );
    const { user } = creds;
    if (user) {
      users.set(user.uid, user);
      ctx.cookies.set("LOGGED_IN_UID", user.uid);
    } else if (signedInUser && signedInUser.uid !== auth.currentUser?.uid) {
      await auth.updateCurrentUser(signedInUser);
    }
  }
  return next();
});

```

Now let's add our router to the middleware application and set the application to listen on port 8000:

```

app.use(router.routes());
app.use(router.allowedMethods());
await app.listen({ port: 8000 });

```

Now we have an application that should serve up our APIs.

2.5.5 Create a Project in Deno Deploy

1. Go to <https://dash.deno.com/new> (Sign in with GitHub if you didn't already) and click on **Create**.
2. Now click on **Settings** button available on the project page.
3. Navigate to **Environment Variables** Section and add the following:

FIREBASE_USERNAME

The Firebase user (email address) that was added above.

FIREBASE_PASSWORD

The Firebase user password that was added above.

FIREBASE_CONFIG

The configuration of the Firebase application as a JSON string.

The configuration needs to be a valid JSON string to be readable by the application. If the code snippet given when setting up looked like this:

```

var firebaseConfig = {
  apiKey: "APIKEY",
  authDomain: "example-12345.firebaseio.com",
  projectId: "example-12345",
  storageBucket: "example-12345.appspot.com",
  messagingSenderId: "1234567890",

```

Deno Deploy

```
const auth = firebase.auth(firebaseApp);
const db = firebase.firestore(firebaseApp);
```

We are also going to setup the application to handle signed in users per request. So we will create a map of users that we have previously signed in in this deployment. While in this tutorial we will only ever have one signed in user, the code can easily be adapted to allow clients to sign-in individually:

```
const users = new Map();
```

Let's create our middleware router and create three different middleware handlers to support GET and POST of /songs and a GET of a specific song on /songs/{title}:

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

```
// Returns any songs in the collection
```

```
router.get("/songs", async (ctx) => {
  const querySnapshot = await db.collection("songs").get();
  ctx.response.body = querySnapshot.docs.map((doc) => doc.data());
  ctx.response.type = "json";
});
```

```
// Returns the first document that matches the title
```

```
router.get("/songs/:title", async (ctx) => {
  const { title } = ctx.params;
  const querySnapshot = await db.collection("songs").where("title", "==", title)
    .get();
  const song = querySnapshot.docs.map((doc) => doc.data())[0];
  if (!song) {
    ctx.response.status = 404;
    ctx.response.body = `The song titled "${ctx.params.title}" was not found.`;
    ctx.response.type = "text";
  } else {
    ctx.response.body = querySnapshot.docs.map((doc) => doc.data())[0];
    ctx.response.type = "json";
  }
});
```

```
function isSong(value) {
  return typeof value === "object" && value !== null && "title" in value;
}
```

```
// Removes any songs with the same title and adds the new song
```

```
router.post("/songs", async (ctx) => {
  const body = ctx.request.body();
  if (body.type !== "json") {
    ctx.throw(Status.BadRequest, "Must be a JSON document");
  }
  const song = await body.value;
  if (!isSong(song)) {
    ctx.throw(Status.BadRequest, "Payload was not well formed");
  }
  const querySnapshot = await db
    .collection("songs")
    .where("title", "=", song.title)
    .get();
  await Promise.all(querySnapshot.docs.map((doc) => doc.ref.delete()));
```

Deno Deploy

- Click on your project name on the **Settings** page to go back to the dashboard of your project.
 - Click on **Deploy URL**, paste the raw link and click on **Deploy**.
 - Click on Visit to see your project live on Deno Deploy (remember to append /quotes to the deployment URL to see the content of your FaunaDB)
- That's it.

Congrats on building and deploying the Quotes API!

<https://dash.deno.com/login?redirect=%2Fnew>

2.3.4.1 Deploy JavaScript Globally

Deno Deploy is a distributed system that runs JavaScript, TypeScript, and WebAssembly at the edge, worldwide.

- Github integration with public & private repos
- Runs in all 32 network locations
- Free deno.dev subdomain & custom domains
- Automatic HTTPS / TLS
- Unlimited production deployments & previews
- Up to 10ms CPU time per request

Continue with Github <https://dash.deno.com/new>

2.4 Persist data using DynamoDB

Amazon DynamoDB is a fully managed NoSQL database. In this tutorial let's take a look at how we can use it to build a small API that has endpoints to insert and retrieve information.

The tutorial assumes that you've an AWS and Deno Deploy account.

- [Overview](#)
- [Setup DynamoDB](#)
- [Create a Project in Deno Deploy](#)
- [Write the Application](#)
- [Deploy the Application](#)

2.4.1 Overview

We're going to build an API with a single endpoint that accepts GET/POST requests and returns appropriate information

A GET request to the endpoint should return the details of the song based on its title.

GET /songs?title=Song%20Title # '%20' == space

```
# response
{
  title: "Song Title"
  artist: "Someone"
  album: "Something",
  released: "1970",
  genres: "country rap",
}
```

A POST request to the endpoint should insert the song details.

POST /songs

post request body

```
{
  title: "A New Title"
  artist: "Someone New"
  album: "Something New",
  released: "2020",
  genres: "country rap",
}
```

2.4.2 Setup DynamoDB

Our first step in the process is to generate AWS credentials to programmatically access DynamoDB.

Generate Credentials:

1. Go to <https://console.aws.amazon.com/iam/> and go to "Users" section.
2. Click on **Add user** button, fill the **User name** field (maybe use denamo) and select **Programmatic access** type.
3. Click on **Next: Permissions**, then on **Attach existing policies directly**, search for AmazonDynamoDBFullAccess and select it.
4. Click on **Next: Tags**, then on **Next: Review** and finally **Create user**.
5. Click on **Download .csv** button to download the credentials.

Create database table:

1. Go to <https://console.aws.amazon.com/dynamodb> and click on **Create table** button.
2. Fill the **Table name** field with Songs and **Primary key** with title.
3. Scroll down and click on **Create**. That's it.

2.4.3 Create a Project in Deno Deploy

1. Go to <https://dash.deno.com/new> (Sign in with GitHub if you didn't already) and click on **Create**.
2. Now click on **Settings** button available on the project page.
3. Navigate to **Environment Variables** Section and add the following secrets.
 - **AWS_ACCESS_KEY_ID** - Use the value that's available under **Access key ID** column in the downloaded CSV.
 - **AWS_SECRET_ACCESS_KEY** - Use the value that's available under **Secret access key** column in the downloaded CSV.

Now click on the project name to go back to the project dashboard. Keep this tab open as we will come back here later in the deploy step.

2.4.4 Write the Application

```
import {
  json,
  serve,
  validateRequest,
} from "https://deno.land/x/sift@0.6.0/mod.ts";
// AWS has an official SDK that works with browsers. As most Deno Deploy's
// APIs are similar to browser's, the same SDK works with Deno Deploy.
// So we import the SDK along with some classes required to insert and
// retrieve data.
import {
  DynamoDBClient,
  GetItemCommand,
  PutItemCommand,
} from "https://cdn.skypack.dev/@aws-sdk/client-dynamodb?dts";

// Create a client instance by providing your region information.
// The credentials are obtained from environment variables which
// we set during our project creation step on Deno Deploy.
const client = new DynamoDBClient({
  region: "ap-south-1",
  credentials: {
    accessKeyId: Deno.env.get("AWS_ACCESS_KEY_ID"),
    secretAccessKey: Deno.env.get("AWS_SECRET_ACCESS_KEY"),
  },
});

serve({
```

2.5.3 Setup Firebase

[Firebase](#) is a feature rich platform. All the details of Firebase administration are beyond the scope of this tutorial. We will cover what it needed for this tutorial.

1. Create a new project under the [Firebase console](#).
2. Add a web application to your project. Make note of the firebaseConfig provided in the setup wizard. It should look something like the below. We will use this later:
3.

```
var firebaseConfig = {
```
4.

```
  apiKey: "APIKEY",
```
5.

```
  authDomain: "example-12345.firebaseio.com",
```
6.

```
  projectId: "example-12345",
```
7.

```
  storageBucket: "example-12345.appspot.com",
```
8.

```
  messagingSenderId: "1234567890",
```
9.

```
  appId: "APPID",
```
10.

```
};
```
10. Under Authentication in the administration console for, you will want to enable the Email/Password sign-in method.
11. You will want to add a user and password under Authentication and then Users section, making note of the values used for later.
12. Add Firestore Database to your project. The console will allow you to setup in production mode or test mode. It is up to you how you configure this, but production mode will require you to setup further security rules.
13. Add a collection to the database named songs. This will require you to add at least one document. Just set the document with an Auto ID.

Note depending on the status of your Google account, there maybe other setup and administration steps that need to occur.

2.5.4 Write the application

We want to create our application as a JavaScript file in our favorite editor.

The first thing we will do is import the XMLHttpRequest polyfill that Firebase needs to work under Deploy as well as a polyfill for localStorage to allow the Firebase auth to persist logged in users:

```
import "https://deno.land/x/xhr@0.1.1/mod.ts";
import { installGlobals } from "https://deno.land/x/virtualstorage@0.1.0/mod.ts";
installGlobals();
```

i we are using the current version of packages at the time of the writing of this tutorial. They may not be up-to-date and you may want to double check current versions.

Because Deploy has a lot of the web standard APIs, it is best to use the web libraries for Firebase under deploy. Currently v9 is in still in beta for Firebase, so we will use v8 in this tutorial:

```
import firebase from "https://cdn.skypack.dev/firebase@8.7.0/app";
import "https://cdn.skypack.dev/firebase@8.7.0/auth";
import "https://cdn.skypack.dev/firebase@8.7.0/firestore";
```

We are also going to use [oak](#) as the middleware framework for creating the APIs, including middleware that will take the localStorage values and set them as client cookies:

```
import {
  Application,
  Router,
  Status,
} from "https://deno.land/x/oak@v7.7.0/mod.ts";
import { virtualStorage } from "https://deno.land/x/virtualstorage@0.1.0/middleware.ts";
```

Now we need to setup our Firebase application. We will be getting the configuration from environment variables we will setup later under the key FIREBASE_CONFIG and get references to the parts of Firebase we are going to use:

```
const firebaseConfig = JSON.parse(Deno.env.get("FIREBASE_CONFIG"));
const firebaseApp = firebase.initializeApp(firebaseConfig, "example");
```


A POST request to the endpoint should insert the song details.

POST /songs

post request body

```
{
  title: "A New Title"
  artist: "Someone New"
  album: "Something New",
  released: "2020",
  genres: "country rap",
}
```

In this tutorial, we will be:

- Creating and setting up a [Firebase Project](#).
- Using a text editor to create our application.
- Creating a [gist](#) to "host" our application.
- Deploying our application on [Deno Deploy](#).
- Testing our application using [cURL](#).

2.5.2 Concepts

There are a few concepts that help in understanding why we take a particular approach in the rest of the tutorial, and can help in extending the application. You can skip ahead to [Setup Firebase](#) if you want.

2.5.2.1 Deploy is browser-like

Even though Deploy runs in the cloud, in many aspects the APIs it provides are based on web standards. So when using Firebase, the Firebase APIs are more compatible with the web than those that are designed for server run times. That means we will be using the Firebase web libraries in this tutorial.

2.5.2.2 Firebase uses XHR

Firebase uses a wrapper around Closure's [WebChannel](#) and WebChannel was originally built around [XMLHttpRequest](#). While WebChannel supports the more modern `fetch()` API, current versions of Firebase for the web do not uniformly instantiate WebChannel with `fetch()` support, and instead use `XMLHttpRequest`.

While Deploy is browser-like, it does not support `XMLHttpRequest`. `XMLHttpRequest` is a "legacy" browser API that has several limitations and features that would be difficult to implement in Deploy, which means it is unlikely that Deploy will ever implement that API.

So, in this tutorial we will be using a limited polyfill that provides enough of the `XMLHttpRequest` feature set to allow Firebase/WebChannel to communicate with the server.

2.5.2.3 Firebase auth

Firebase offers quite [a few options](#) around authentication. In this tutorial we are going to be using email and password authentication.

When a user is logged in, Firebase can persist that authentication. Because we are using the web libraries for Firebase, persisting the authentication allows a user to navigate away from a page and not need to re-log in when returning. Firebase allows authentication to be persisted in local storage, session storage or none.

In a Deploy context, it is a little different. A Deploy deployment will remain "active" meaning that in-memory state will be present from request to request on some requests, but under various conditions a new deployment can be started up or shutdown. Currently, Deploy doesn't offer any persistence outside of in-memory allocation. In addition it doesn't currently offer the global `localStorage` or `sessionStorage`, which is what is used by Firebase to store the authentication information.

In order to reduce the need to re-authenticate but also ensure that we can support multiple-users with a single deployment, we are going to use a polyfill that will allow us to provide a `localStorage` interface to Firebase, but store the information as a cookie in the client.

```
"/songs": handleRequest,
});
```

```
async function handleRequest(request) {
  // The endpoint allows GET and POST request. A parameter named "title"
  // for a GET request to be processed. And body with the fields defined
  // below are required to process POST request.
  // validateRequest ensures that the provided terms are met by the request.
  const { error, body } = await validateRequest(request, {
    GET: {
      params: ["title"],
    },
    POST: {
      body: ["title", "artist", "album", "released", "genres"],
    },
  });
  if (error) {
    return json({ error: error.message }, { status: error.status });
  }
}
```

// Handle POST request.

```
if (request.method === "POST") {
  try {
    // When we want to interact with DynamoDB, we send a command using the client
    // instance. Here we are sending a PutItemCommand to insert the data from the
    // request.
    const {
      $metadata: { httpStatusCode },
    } = await client.send(
      new PutItemCommand({
        TableName: "Songs",
        Item: {
          // Here 'S' implies that the value is of type string
          // and 'N' implies a number.
          title: { S: body.title },
          artist: { S: body.artist },
          album: { S: body.album },
          released: { N: body.released },
          genres: { S: body.genres },
        },
      }),
    );

    // On a successful put item request, dynamo returns a 200 status code (weird).
    // So we test status code to verify if the data has been inserted and respond
    // with the data provided by the request as a confirmation.
    if (httpStatusCode === 200) {
      return json({ ...body }, { status: 201 });
    }
  } catch (error) {
    // If something goes wrong while making the request, we log
    // the error for our reference.
  }
}
```

```

    console.log(error);
  }

  // If the execution reaches here it implies that the insertion wasn't successful.
  return json({ error: "couldn't insert data" }, { status: 500 });
}

// Handle GET request.
try {
  // We grab the title form the request and send a GetItemCommand
  // to retrieve the information about the song.
  const { searchParams } = new URL(request.url);
  const { Item } = await client.send(
    new GetItemCommand({
      TableName: "Songs",
      Key: {
        title: { S: searchParams.get("title") },
      },
    }),
  );

  // The Item property contains all the data, so if it's not undefined,
  // we proceed to returning the information about the title
  if (Item) {
    return json({
      title: Item.title.S,
      artist: Item.artist.S,
      album: Item.album.S,
      released: Item.released.S,
      genres: Item.genres.S,
    });
  }
} catch (error) {
  console.log(error);
}

// We might reach here if an error is thrown during the request to database
// or if the Item is not found in the database.
// We reflect both conditions with a general message.
return json(
  {
    message: "couldn't find the title",
  },
  { status: 404 },
);
}

```

2.4.5 Deploy the Application

Now that we've everything in place, let's get to deploying the application.

The steps:

1. Go to <https://gist.github.com/new> and create a new gist with the above code (make sure the file ends with .js).

For convenience, the code is also hosted at <https://deno.com/examples/dynamo.js> so you can skip creating a gist if you just want to try out the above example without making changes to it.

2. Go to the project (that we previously created) dashboard in Deno Deploy.
3. Click on **Deploy URL** and paste the raw link of the gist.
4. Click on **Deploy** and copy the URL that's displayed under **Domains** section.

Let's test the API.

POST some data.

```
curl --request POST --data \
```

```
'{"title": "Old Town Road", "artist": "Lil Nas X", "album": "7", "released": "2019", "genres": "Country rap, Pop"}'\
```

```
--dump-header - https://<project_name>.deno.dev/songs
```

GET information about the title.

```
curl https://<project_name>.deno.dev/songs?title=Old%20Town%20Road
```

Congratulations on learning how to use DynamoDB with Deno Deploy!

2.5 Persist data using Firebase

Firebase is a platform developed by Google for creating mobile and web applications. You can persist data on the platform using Firestore. In this tutorial let's take a look at how we can use it to build a small API that has endpoints to insert and retrieve information.

- [Overview](#)
- [Concepts](#)
- [Setup Firebase](#)
- [Write the application](#)
- [Deploy the application](#)

2.5.1 Overview

We are going to build an API with a single endpoint that accepts GET and POST requests and returns a JSON payload of information:

A GET request to the endpoint without any sub-path should return the details

of all songs in the store:

GET /songs

response

```
[
  {
    title: "Song Title",
    artist: "Someone",
    album: "Something",
    released: "1970",
    genres: "country rap",
  }
]
```

A GET request to the endpoint with a sub-path to the title should return the

details of the song based on its title.

GET /songs/Song%20Title # '%20' == space

response

```
{
  title: "Song Title"
  artist: "Someone"
  album: "Something",
  released: "1970",
  genres: "country rap",
}
```

A small example:

6.1.5 fresh

Preact, but super edgy.

[Fresh](#) is a web framework that lets you build projects very fast, highly dynamic, and without the need of a build step. Fresh embraces isomorphic JavaScript like never before. Write a JSX component, have it render on the edge just-in-time, and then enhance it with client side JS for great interactivity.

It has support for file-system routing à la Next.js and TypeScript out of the box.

An example routes/index.tsx would look like this:

import Counter from "../islands/Counter.tsx";

```
export default function Home() {
  return (
    <div>
      <p>
        Welcome to `fresh`. Try update this message in the ./routes/index.tsx
        file, and refresh.
      </p>
      <Counter />
    </div>
  );
}
```

This file is rendered on the server only, but can include "islands" of interactivity that will also be client rendered. Here is one of these islands in the "islands/Counter.tsx" file.

import { useState } from "preact/hooks";

import { IS_BROWSER } from "\$fresh/runtime.ts";

```
interface CounterProps {
  start: number;
}
```

```
export default function Counter(props: CounterProps) {
  const [count, setCount] = useState(props.start);
  return (
    <div>
      <p>{count}</p>
      <button onClick={() => setCount(count - 1)} disabled={!IS_BROWSER}>
        -1
      </button>
      <button onClick={() => setCount(count + 1)} disabled={!IS_BROWSER}>
        +1
      </button>
    </div>
  );
}
```

Note: this file alone is not enough to run a fresh project. Check out a full example in the [project repository](#).

5.2.4 Example

A small example that has an endpoint to send a new message to all other actively running instances in different regions and another to fetch all messages from an instance.

import { serve } from "https://deno.land/std@0.140.0/http/server.ts";

```
const messages = [];
// Create a new broadcast channel named earth.
const channel = new BroadcastChannel("earth");
// Set onmessage event handler.
channel.onmessage = (event: MessageEvent) => {
  // Update the local state when other instances
  // send us a new message.
  messages.push(event.data);
};
```

```
function handler(req: Request): Response {
  const { pathname, searchParams } = new URL(req.url);
```

```
  // Handle /send?message=<message> endpoint.
  if (pathname.startsWith("/send")) {
    const message = searchParams.get("message");
    if (!message) {
      return new Response("?message not provided", { status: 400 });
    }
```

```
    // Update local state.
    messages.push(message);
    // Inform all other active instances of the deployment
    // about the new message.
    channel.postMessage(message);
    return new Response("message sent");
  }
```

```
  // Handle /messages request.
  if (pathname.startsWith("/messages")) {
    return new Response(JSON.stringify(messages), {
      "content-type": "application/json",
    });
  }
```

```
  return new Response("not found", { status: 404 });
}
```

serve(handler);

You can test this example by making an HTTP request

to https://broadcast.deno.dev/send?message=Hello_from_<region> and then making another request

to <https://broadcast.deno.dev/messages> from a different region (by using a VPN or some other way) to check if the first request's message is present in the second region.

We built [a small chat application](#) that you can play with at <https://denochat.deno.dev/>

5.3 Fetch API

The [Fetch API](#) allows you to make outbound HTTP requests in Deno Deploy. It is a web standard and has the following interfaces:

Runtime => Fetch API

- `fetch()` - The method that allows you to make outbound HTTP requests
- [Request](#) - represents a request resource of `fetch()`
- [Response](#) - represents a response resource of `fetch()`
- [Headers](#) - represents HTTP Headers of requests and responses.

This page shows usage for the `fetch()` method. You can click above on the other interfaces to learn more about them.

Fetch also supports fetching from file URLs to retrieve static files. For more info on static files, see the [filesystem API documentation](#).

5.3.1 `fetch()`

The `fetch()` method initiates a network request to the provided resource and returns a promise that resolves after the response is available.

```
function fetch(
  resource: Request | string,
  init?: RequestInit,
): Promise<Response>;
```

5.3.1.1 Parameters

name	type	optional	description
resource	Request		
[USVString][usvstring]	false		The resource can either be a request object or a URL string.
init	RequestInit	true	The init object lets you apply optional parameters to the request.

The return type of `fetch()` is a promise that resolves to a [Response](#).

5.3.2 Examples

The Deno Deploy script below makes a `fetch()` request to the GitHub API for each incoming request, and then returns that response from the handler function.

```
import { serve } from "https://deno.land/std@0.140.0/http/server.ts";
```

```
async function handler(req: Request): Promise<Response> {
  const resp = await fetch("https://api.github.com/users/denoland", {
    // The init object here has an headers object containing a
    // header that indicates what type of response we accept.
    // We're not specifying the method field since by default
    // fetch makes a GET request.
    headers: {
      accept: "application/json",
    },
  });
  return new Response(resp.body, {
    status: resp.status,
    headers: {
      "content-type": "application/json",
    },
  });
}
```

`serve(handler)`;
Also checkout [Make an outbound request](#) post for more examples.

5.4 Request

The [Request](#) interface is part of the Fetch API and represents the request of `fetch()`.

- [Constructor](#)
- [Parameters](#)

5.8.9 `Deno.FileInfo`

The `Deno.FileInfo` interface is used to represent a file system entry's metadata. It is returned by the `Deno.stat()` and `Deno.lstat()` functions. It can represent either a file, a directory, or a symlink. In Deno Deploy, only the file type, and size properties are available. The size property behaves the same way it does on Linux.

```
interface FileInfo {
  isDirectory: boolean;
  isFile: boolean;
  isSymlink: boolean;
  size: number;
}
```

5.8.10 `Deno.realpath`

`Deno.realpath()` returns the resolved absolute path to a file, after following symlinks.

The function definition is the same as [Deno](#).

```
function Deno.realpath(path: string | URL): Promise<string>
```

The path can be a relative or absolute. It can also be a file: URL.

5.8.10.1 Example

This example calls `Deno.realpath()` to get the absolute path of a file in the root of the repository. The result is returned as the response body.

```
import { serve } from "https://deno.land/std@0.140.0/http/server.ts";
```

```
async function handler(_req) {
  const path = await Deno.realpath("./README.md");

  return new Response(`The fully resolved path for ./README.md is ${path}`);
}
```

`serve(handler)`;

6 Additional Resources

6.1 HTTP Frameworks

If you want to build something more complicated, or like the concept of a higher order framework to create your application, then there are frameworks that support Deno Deploy.

6.1.1 oak

[oak](#) is an expressive middleware framework including a router and is heavily influenced by koa. It provides support for Deno Deploy.

A little example of a Deno Deploy "hello world" script would be:

More information can be found in the [oak and Deno Deploy documentation](#).

6.1.2 router

[Router](#) by [denosaurs](#) is a tiny router framework for Deno Deploy. It can route based on method, route, and has customizable handlers for cases such as no match, errors, or unknown methods.

A small example using router to handle POST and GET requests separately:

More information can be found in the [documentation](#).

6.1.3 nanossr

[nanossr](#) is a tiny server side rendering framework for Deno Deploy. It uses JSX for templating the HTML, and [twind](#) for TailwindCSS like styling. It is fully integrated into a single import, making it ideal for throwing together small landing pages quickly.

An example that renders Hello {name} to the screen, where {name} depemds on the name query parameter:

6.1.4 sift

[Sift](#) is a routing and utility library for Deno Deploy. Its route handler signature is simple and easy to understand. Handlers accept a Request and return a Response.

```
Deno Deploy
const body = readableStreamFromReader(file);

return new Response(body);
}
```

```
serve(handler);
5.8.6 Deno.File
Deno.File is a file handle returned from Deno.open(). It can be used to read chunks of the file using
the read() method. The file handle can be closed using the close() method.
The interface is similar to Deno, but it doesn't writing to the file, or seeking. Support for the latter will be
added in the future.
class File {
  readonly rid: number;

  close(): void;
  read(p: Uint8Array): Promise<number | null>;
}
```

```
The path can be a relative or absolute. It can also be a file: URL.
5.8.6.1 Deno.File#read()
The read method is used to read a chunk of the file. It should be passed a buffer to read the data into. It
returns the number of bytes read, or null if the end of the file has been reached.
function read(p: Uint8Array): Promise<number | null>;
5.8.6.2 Deno.File#close()
The close method is used to close the file handle. Closing the handle will interrupt all ongoing reads.
function close(): void;
5.8.7 Deno.stat
Deno.stat() reads a file system entry's metadata. It returns a Deno.FileInfo object. Symlinks are followed.
The function definition is the same as Deno. It does not return modification time, access time, or creation
time values.
```

```
function Deno.stat(path: string | URL): Promise<Deno.FileInfo>
The path can be a relative or absolute. It can also be a file: URL.
5.8.7.1 Example
This example gets the size of a file, and returns the result as the response body.
import { serve } from "https://deno.land/std@0.140.0/http/server.ts";
```

```
async function handler(_req) {
  // Get file info of the README.md at the root of the repository.
  const info = await Deno.stat("./README.md");

  // Get the size of the file in bytes.
  const size = info.size;

  return new Response(`README.md is ${size} bytes large`);
}
```

```
serve(handler);
5.8.8 Deno.lstat
Deno.lstat() is similar to Deno.stat(), but it does not follow symlinks.
The function definition is the same as Deno. It does not return modification time, access time, or creation
time values.
function Deno.lstat(path: string | URL): Promise<Deno.FileInfo>
The path can be a relative or absolute. It can also be a file: URL.
```

- [Properties](#)
- [Methods](#)
- [Example](#)

5.4.1 Constructor

The Request() constructor creates a new Request instance.

let request = new Request(input, init);

5.4.1.1 Parameters

name	type	optional	description
resource	Request or USVString	false	The resource can either be a request object or a URL string.
init	RequestInit	true	The init object lets you set optional parameters to apply to the request.

The return type is a Request instance.

5.4.1.1.1 RequestInit

name	type	default	description
[method][method]	string	GET	The method of the request.
[headers][headers]	Headers or { [key: string]: string }	none	Th Headers for the request.
[body][body]	Blob, BufferSource, FormData, URLSearchParams, USVString, or ReadableStream	none	The body of the request.
[cache][cache]	string	none	The cache mode of the request.
[credentials][credentials]	string	same-origin	The credentials mode of the request.
[integrity][integrity]	string	none	The cryptographic hash of the request's body.
[mode][mode]	string	cors	The request mode you want to use.
[redirect][redirect]	string	follow	The mode of how redirects are handled.
[referrer][referrer]	string	about:client	A USVString specifying a referrer, client or a URL.

5.4.2 Properties

name	type	description
[cache][cache]	string	The cache mode indicates how the (default, no-cache, etc) request should be cached by browser.
[credentials][credentials]	string	The credentials (omit, same-origin, etc) indicate whether user agent should send cookies in case of CORs of the request.
[destination][destination]	[RequestDestination][requestdestination]	The string indicates the type of content being requested.
[body][body]	[ReadableStream][readablestream]	The getter exposes a ReadableStream of the body contents.
[bodyUsed][bodyused]	boolean	Indicates whether the body content is read.
[url][url]	USVString	The URL of the request.
[headers][headers]	Headers	The headers associated with the request.
[integrity][integrity]	string	The cryptographic hash of the request's body.
[method][method]	string	The request's method (POST, GET, etc).
[mode][mode]	string	Indicates the mode of the request (e.g. cors).
[redirect][redirect]	string	The mode of how redirects are handled.
[referrer][referrer]	string	The referrer of the request.
[referrerPolicy][referrerpolicy]	string	The referrer policy of the request

All the above properties are read only.

5.4.3 Methods

name	description
[arrayBuffer()][arraybuffer]	Reads the body stream to its completion and returns an ArrayBuffer object.
[blob()][blob]	Reads the body stream to its completion and returns a Blob object.
[formData()][formdata]	Reads the body stream to its completion and returns a FormData object.
[json()][json]	Reads the body stream to its completion, parses it as JSON and returns a JavaScript object.
[text()][text]	Reads the body stream to its completion and returns a USVString object (text).
[clone()][clone]	Clones the Request object.

5.4.4 Example

```
import { serve } from "https://deno.land/std@0.140.0/http/server.ts";

function handler(_req) {
  // Create a post request
  const request = new Request("https://post.deno.dev", {
    method: "POST",
    body: JSON.stringify({
      message: "Hello world!",
    }),
    headers: {
      "content-type": "application/json",
    },
  });

  console.log(request.method); // POST
  console.log(request.headers.get("content-type")); // application/json

  return fetch(request);
}
```

serve(handler);

5.5 Response

The [Response](#) interface is part of the Fetch API and represents a response resource of `fetch()`.

- [Constructor](#)
- [Parameters](#)
- [Properties](#)
- [Methods](#)
- [Example](#)

5.5.1 Constructor

The `Response()` constructor creates a new `Response` instance.
`let response = new Response(body, init);`

5.5.1.1 Parameters

name	type	optional	description
body	Blob, BufferSource, FormData, ReadableStream, URLSearchParams, or USVString	true	The body of the response. The default value is null.
init	ResponseInit	true	An optional object that allows setting status and headers of the response.

The return type is a `Response` instance.

5.5.1.1.1 ResponseInit

name	type	optional	description
status	number	true	The status code of the response.
statusText	string	true	The status message representative of the status code.
headers	Headers or string[][] or Record<string, string>	false	The HTTP headers of the response.

```
import { serve } from "https://deno.land/std@0.140.0/http/server.ts";

async function handler(_req) {
  // Let's read the README.md file available at the root
  // of the repository to explore the available methods.

  // Relative paths are relative to the root of the repository
  const readmeRelative = await Deno.readFile("./README.md");
  // Absolute paths.
  // The content of the repository is available under at Deno.cwd().
  const readmeAbsolute = await Deno.readFile(`${Deno.cwd()}/README.md`);
  // File URLs are also supported.
  const readmeFileUrl = await Deno.readFile(
    new URL(`file://${Deno.cwd()}/README.md`),
  );

  // Decode the Uint8Array as string.
  const readme = new TextDecoder().decode(readmeRelative);
  return new Response(readme);
}
```

serve(handler);

5.8.4 Deno.readFile

This function is similar to [Deno.readFile](#) except it decodes the file contents as a UTF-8 string.
`function Deno.readFile(path: string | URL): Promise<string>`

5.8.4.1 Example

This example reads a text file into memory and returns the contents as the response body.
`import { serve } from "https://deno.land/std@0.140.0/http/server.ts";`

```
async function handler(_req) {
  const readme = await Deno.readFile("./README.md");
  event.respondWith(new Response(readme));
}
```

serve(handler);

5.8.5 Deno.open

`Deno.open()` allows you to open a file, returning a file handle. This file handle can then be used to read the contents of the file. See [Deno.File](#) for information on the methods available on the file handle. The function definition is similar to [Deno](#), but it doesn't support [OpenOptions](#) for the time being. Support will be added in the future.

`function Deno.open(path: string | URL): Promise<Deno.File>`
The path can be a relative or absolute. It can also be a file: URL.

5.8.5.1 Example

This example opens a file, and then streams the content as the response body.
`import { serve } from "https://deno.land/std@0.140.0/http/server.ts";`
`import { readableStreamFromReader } from "https://deno.land/std@0.140.0/streams/conversion.ts";`

```
async function handler(_req) {
  // Open the README.md file available at the root of the repository.
  const file = await Deno.open("./README.md");

  // Turn the `Deno.File` into a `ReadableStream`. This will automatically close
  // the file handle when the response is done sending.
```

Runtime => File System API

- The files in your GitHub repository, if you deploy via the GitHub integration.
- The entrypoint file in a playground deployment.

The APIs that are available are:

- [Deno.cwd](#)
- [Deno.readDir](#)
- [Deno.readFile](#)
- [Deno.readTextFile](#)
- [Deno.open](#)
- [Deno.stat](#)
- [Deno.lstat](#)
- [Deno.realpath](#)

5.8.1 Deno.cwd

Deno.cwd() returns the current working directory of your deployment. It is located at the root of your deployment's root directory. For example, if you deployed via the GitHub integration, the current working directory is the root of your GitHub repository.

5.8.2 Deno.readDir

Deno.readDir() allows you to list the contents of a directory.

The function is fully compatible with [Deno](#).

function Deno.readDir(path: string | URL): AsyncIterable<DirEntry>

The path can be a relative or absolute. It can also be a file: URL.

5.8.2.1 Example

This example lists the contents of a directory and returns this list as a JSON object in the response body.

import { serve } from "https://deno.land/std@0.140.0/http/server.ts";

```
async function handler(_req) {
  // List the posts in the `blog` directory located at the root
  // of the repository.
  const posts = [];
  for await (const post of Deno.readDir(`./blog`)) {
    posts.push(post);
  }

  // Return JSON.
  return new Response(JSON.stringify(posts, null, 2), {
    headers: {
      "content-type": "application/json",
    },
  });
}
```

serve(handler);

5.8.3 Deno.readFile

Deno.readFile() allows you to read a file fully into memory.

The function definition is similar to [Deno](#), but it doesn't support [ReadFileOptions](#) for the time being.

Support will be added in the future.

function Deno.readFile(path: string | URL): Promise<Uint8Array>

The path can be a relative or absolute. It can also be a file: URL.

5.8.3.1 Example

This example reads the contents of a file into memory as a byte array, and then returns it as the response body.

5.5.2 Properties

name	type	read only	description
[body][body]	ReadableStream	true	The getter exposes a ReadableStream of the body contents.
[bodyUsed][bodyused]	boolean	true	Indicates whether the body content is read.
[url][url]	USVString	true	The URL of the response.
[headers][headers]	Headers	true	The headers associated with the response.
[ok][ok]	boolean	true	Indicates if the response is successful (200-299 status).
[redirected][redirected]	boolean	true	Indicates if the response is the result of a redirect.
[status][status]	number	true	The status code of the response
[statusText][statustext]	string	true	The status message of the response
[type][type]	string	true	The type of the response.

5.5.3 Methods

name	description
[arrayBuffer()][arraybuffer]	Reads the body stream to its completion and returns an ArrayBuffer object.
[blob()][blob]	Reads the body stream to its completion and returns a Blob object.
[formData()][formdata]	Reads the body stream to its completion and returns a FormData object.
[json()][json]	Reads the body stream to its completion, parses it as JSON and returns a JavaScript object.
[text()][text]	Reads the body stream to its completion and returns a USVString object (text).
[clone()][clone]	Clones the response object.
[error()][error]	Returns a new response object associated with a network error.
[redirect(url: string, status?: number)][redirect]	Creates a new response that redirects to the provided URL.

5.5.4 Example

import { serve } from "https://deno.land/std@0.140.0/http/server.ts";

```
function handler(_req) {
  // Create a response with html as its body.
  const response = new Response("<html> Hello </html>", {
    status: 200,
    headers: {
      "content-type": "text/html",
    },
  });

  console.log(response.status); // 200
  console.log(response.headers.get("content-type")); // text/html

  return response;
}
```

serve(handler);

5.6 Headers

The [Headers](#) interface is part of the Fetch API. It allows you create and manipulate the HTTP headers of request and response resources of fetch().

- [Constructor](#)
- [Parameters](#)
- [Methods](#)
- [Example](#)

5.6.1 Constructor

The Header() constructor creates a new Header instance.

let headers = new Headers(init);

5.6.1.1 Parameters

name	type	optional	description
init	Headers / { [key: string]: string }	true	The init option lets you initialize the headers object with an existing Headers or an object literal.

The return type of the constructor is a Headers instance.

5.6.2 Methods

name	description
<code>append(name: string, value: string)</code>	Appends a header (overwrites existing one) to the Headers object.
<code>delete(name: string)</code>	Deletes a header from the Headers object.
<code>set(name: string, value: string)</code>	Create a new header in the Headers object.
<code>get(name: string)</code>	Get the value of the header in the Headers object.
<code>has(name: string)</code>	Check if the header exists in the Headers objects.
<code>entries()</code>	Get the headers as key-value pair. The result is iterable.
<code>keys()</code>	Get all the keys of the Headers object. The result is iterable.

5.6.3 Example

// Create a new headers object from an object literal.

```
const myHeaders = new Headers({
  accept: "application/json",
});
```

// Append a header to the headers object.

```
myHeaders.append("user-agent", "Deno Deploy");
```

// Print the headers of the headers object.

```
for (const [key, value] of myHeaders.entries()) {
  console.log(key, value);
}
```

// You can pass the headers instance to Response or Request constructors.

```
const request = new Request("https://api.github.com/users/denoland", {
  method: "POST",
  headers: myHeaders,
});
```

5.7 Sockets API

Deno Deploy supports outbound TCP and TLS connections. These APIs allow you to use databases like PostgreSQL, SQLite, MongoDB, etc., with Deploy.

5.7.1 Deno.connect

Make outbound TCP connections.

The function definition is same as [Deno](#) with the limitation that transport option can only be tcp and hostname cannot be localhost or empty.

```
function Deno.connect(options: ConnectOptions): Promise<Conn>
```

5.7.1.1 Example

```
import { serve } from "https://deno.land/std@0.140.0/http/server.ts";
```

```
async function handler(_req) {
  // Make a TCP connection to example.com
  const connection = await Deno.connect({
    port: 80,
    hostname: "example.com",
  });
```

// Send raw HTTP GET request.

```
const request = new TextEncoder().encode(
  "GET / HTTP/1.1\nHost: example.com\r\n\r\n",
);
const _bytesWritten = await connection.write(request);
```

// Read 15 bytes from the connection.

```
const buffer = new Uint8Array(15);
await connection.read(buffer);
connection.close();
```

// Return the bytes as plain text.

```
return new Response(buffer, {
  headers: {
    "content-type": "text/plain;charset=utf-8",
  },
});
}
```

serve(handler);

5.7.2 Deno.connectTls

Make outbound TLS connections.

The function definition is the same as [Deno](#) with the limitation that hostname cannot be localhost or empty.

```
function Deno.connectTls(options: ConnectTlsOptions): Promise<Conn>
```

5.7.2.1 Example

```
import { serve } from "https://deno.land/std@0.140.0/http/server.ts";
```

```
async function handler(_req) {
  // Make a TLS connection to example.com
  const connection = await Deno.connectTls({
    port: 443,
    hostname: "example.com",
  });
```

// Send raw HTTP GET request.

```
const request = new TextEncoder().encode(
  "GET / HTTP/1.1\nHost: example.com\r\n\r\n",
);
const _bytesWritten = await connection.write(request);
```

// Read 15 bytes from the connection.

```
const buffer = new Uint8Array(15);
await connection.read(buffer);
connection.close();
```

// Return the bytes as plain text.

```
return new Response(buffer, {
  headers: {
    "content-type": "text/plain;charset=utf-8",
  },
});
}
```

serve(handler);

5.8 File System API

Deno Deploy supports a limited set of the file system APIs available in Deno. These file system APIs can access static files from your deployments. Static files are for example: