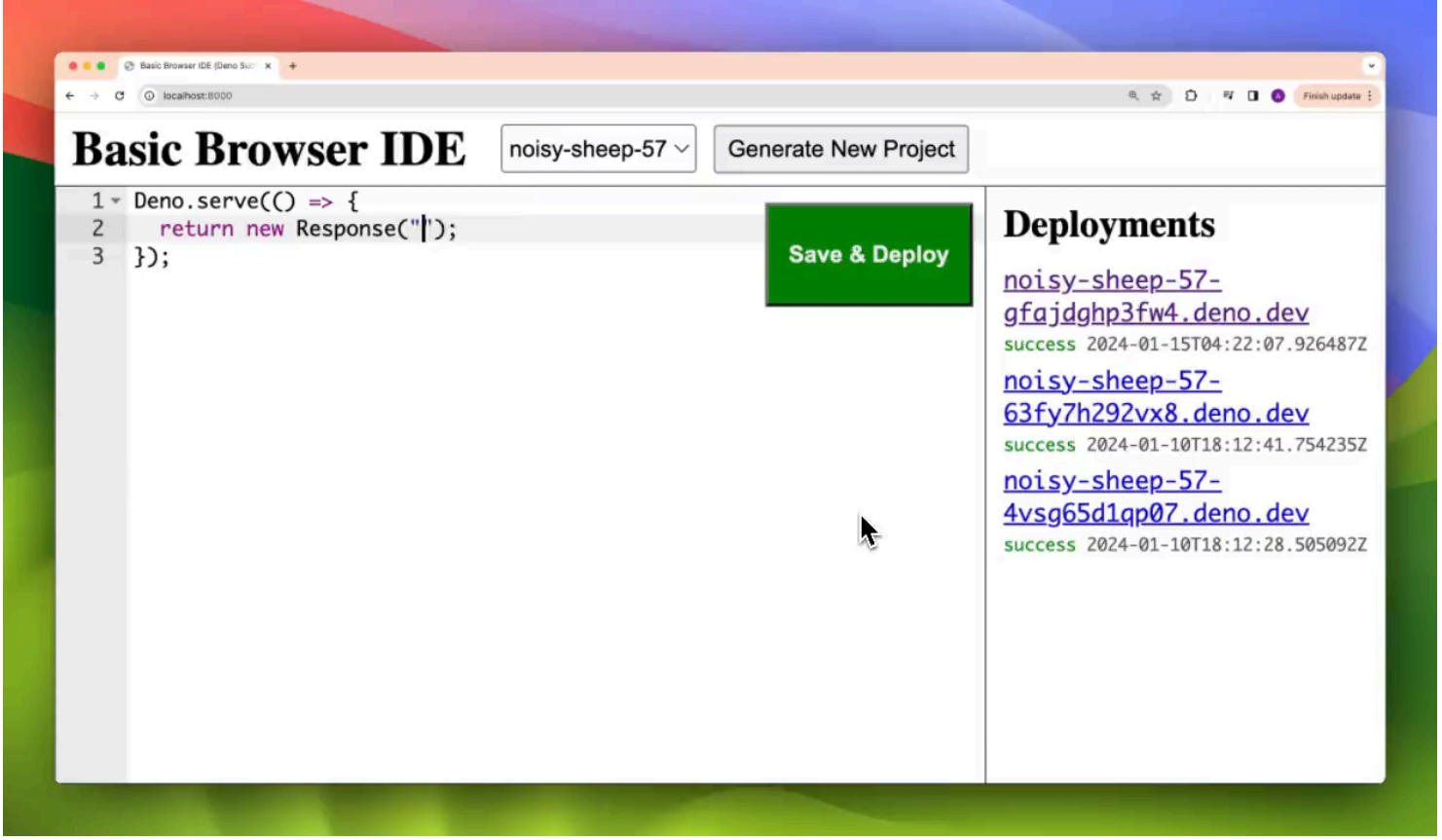
Build Your Own Cloud IDE with the Subhosting API

January 16, 2024

Kevin Whinnery Andy Jiang

How To Subhosting

More and more SaaS platforms are allowing their users to customize their product with code, such as creating bespoke workflows or via an app/integrations marketplace. And a popular approach to enabling code-level customization with minimal friction is through a cloud IDE directly in their product.



A demo of the starter ide template.

In this post, we'll show you how to build your own cloud IDE using the Deno Subhosting API, which allows you to programmatically deploy and run code within seconds on Deno Deploy's global v8 isolate cloud. We'll walk through our Subhosting IDE Starter template, built on Hono, Ace Editor, and Deno.

Or view the accompanying video tutorial on YouTube.

Setting up your project

Before we get started, we'll need the following things:

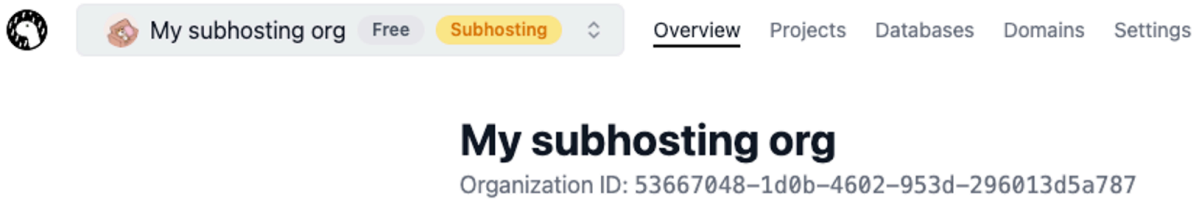
- a Deno Deploy account
- a Deno Deploy Subhosting organization, which you can create from your Deno Deploy dashboard

Create the following project structure in a new folder:

```
subhosting_starter_ide/
├── .env
├── App.tsx
├── deno.json
└── main.tsx
```

Next, you have to create the following environmental variables:

- a `DEPLOY_ACCESS_TOKEN`, which you can generate in your Deno Deploy account
- a `DEPLOY_ORG_ID`, which you can grab from your organization page:



Once you have these values, add them to your `.env` file:

```
DEPLOY_ACCESS_TOKEN = ddp_xxxxxx;
DEPLOY_ORG_ID = ed63948c - xxx - xxx - xxx - xxxxxx;
```

Let's setup our `deno.json` to include a command to run the server and an import map to import Hono:

```
{
  "tasks": {
    "dev": "deno run -A --watch --env main.tsx"
  },
```

```
  "imports": {
    "$hono/": "https://deno.land/x/hono@v3.12.0/"
  }
}
```

Building the server in `main.tsx`

The main logic of our cloud IDE will be in `main.tsx`, which creates the server and the following routes for our application:

- `GET /`: list all projects
- `GET /deployments`: list all deployments for the given project
- `POST /deployment`: create a new deployment for the given project
- `POST /project`: create a new project for the given org

And we need to be able to serve static assets from the `./static` folder, which contains client-side JavaScript and CSS necessary such as Ace.js:

```
/** @jsx jsx */
import { Hono } from "$hono/mod.ts";
import { jsx } from "$hono/jsx/index.ts";
import { serveStatic } from "$hono/middleware.ts";
import App from "./App.tsx";

const app = new Hono();

app.get("/", async (c) => {
});

// Poll deployment data from Subhosting API
app.get("/deployments", async (c) => {
});

// Create deployment for the given project with the Subhosting API
app.post("/deployment", async (c) => {
});

// Create project for the given org with the Subhosting API
app.post("/project", async (c) => {
});

app.use("/*", serveStatic({ root: "./static" }));
```

```
Deno.serve(app.fetch);
```

Next, let's fill in the logic for each route handler. To simplify this for now, we'll import a wrapper library around the Subhosting API that we'll create later:

```
import Client from "./subhosting.ts";

const shc = new Client();
```

Using our `shc` wrapper library, we can add the logic for each route handler:

```
app.get("/", async (c) => {
  const projects = await (await shc.listProjects()).json();
  return c.html(<App projects={projects} />);
});

// Poll deployment data from Subhosting API
app.get("/deployments", async (c) => {
  const projectId = c.req.query("projectId") || "";
  const dr = await shc.listDeployments(projectId, {
    order: "desc",
  });
  const deployments = await dr.json();
  return c.json(deployments);
});

// Create deployment for the given project with the Subhosting API
app.post("/deployment", async (c) => {
  const body = await c.req.json();

  const dr = await shc.createDeployment(body.projectId, {
    entryPointUrl: "main.ts", // maps to `main.ts` under `assets`
    assets: {
      "main.ts": {
        "kind": "file",
        "content": body.code,
        "encoding": "utf-8",
      },
    },
    envVars: {}, // if you need the code to have access to credentials, etc.
  });
  const deploymentResponse = await dr.json();

  return c.json(deploymentResponse);
});
```

```
// Create project for the given org with the Subhosting API
app.post("/project", async (c) => {
  const body = await c.req.parseBody();

  const pr = await shc.createProject(body.name as string);
  const projectResponse = await pr.json();
  console.log(projectResponse);

  return c.redirect("/");
});
```

Before we move on, let's dig into the payload we're sending to create a deployment:

```
entryPointUrl: "main.ts", // maps to `main.ts` under `assets`
assets: {
  "main.ts": {
    "kind": "file",
    "content": body.code,
    "encoding": "utf-8",
  },
},
envVars: {}, // if you need the code to have access to credentials, etc.
```

- **entryPointUrl**: this string is the filename that serves as the entrypoint to deploy. Note that this value must map to a key under `assets`
- **assets**: this is a JSON object of files, scripts, whatever the deployment needs to run. Our example is very simple, so it's a single file (`main.ts`), but for more complex deployments, can become very large with many files.
- **envVars**: you can specify env vars here, which will be present when the code is executed. This is useful in case you want your code to have access to API credentials or other configuration level information to work properly.

To learn more about creating a deployment with the Subhosting API, check out our documentation.

Next, let's create our Subhosting client in `subhosting.ts` .

Creating the Subhosting API wrapper library

Let's create a new `subhosting.ts` file to the root of the project, which will serve as a wrapper around the Subhosting API. In this file, we'll define an interface for `ClientOptions`, as well as a `Client` class, which will have these fields `accessToken`, `orgId`, and `clientOptions`, in addition to a constructor that initializes the instance variables for the class with simple error handling:

```
export interface ClientOptions {
  endpoint?: string;
}

export default class Client {
  accessToken: string;
  orgId: string;
  clientOptions: ClientOptions;

  constructor(accessToken?: string, orgId?: string, options?: ClientOptions) {
    const at = accessToken ?? Deno.env.get("DEPLOY_ACCESS_TOKEN");
    if (!at) {
      throw new Error(
        "A Deno Deploy access token is required (or set DEPLOY_ACCESS_TOKEN env variab
      );
    }

    const org = orgId ?? Deno.env.get("DEPLOY_ORG_ID");
    if (!org) {
      throw new Error(
        "Deno Subhosting org ID is required (or set DEPLOY_ORG_ID env variable).",
      );
    }

    this.accessToken = at;
    this.orgId = org;
    this.clientOptions = Object.assign({
      endpoint: "https://api.deno.com/v1",
    }, options);
  }
}
```

Next, let's create the functions that we've imported and are using in `main.tsx`. Before we do that, let's import the following helper functions, `urlJoin` and `normalize` at the top of the file:

```
import { normalize, urlJoin } from "https://deno.land/x/url_join@1.0.0/mod.ts";
```

Note in our GitHub repo, we've inlined both functions since they were pretty simple.

Let's define a convenience getter, `orgUrl`, which returns the org URL fragment:

```
export default class Client {
  // ...

  get orgUrl() {
    return `/organizations/${this.orgId}`;
  }

  // ...
}
```

Once we've done that, we can define the functions that we've imported and are using in `main.tsx`:

- `fetch`
- `listProjects`
- `createProject`
- `listDeployments`
- `listAppLogs`
- `createDeployment`

With the additional functions, your `Client` will look like this:

```
export default class Client {
  // ...

  /**
   * A wrapper around "fetch", preconfigured with your subhosting API info.
   */
  async fetch(url: string, options?: RequestInit): Promise<Response> {
    const finalUrl = urlJoin(this.clientOptions.endpoint, url);
    const finalHeaders = Object.assign({
      Authorization: `Bearer ${this.accessToken}`,
      "Content-Type": "application/json",
    }, options?.headers || {});
    const finalOptions = Object.assign({}, options, { headers: finalHeaders });

    return await fetch(finalUrl, finalOptions);
  }

  /**
```

```
 * Get a list of projects for the configured org, with optional query params
 */
// deno-lint-ignore no-explicit-any
async listProjects(query?: any): Promise<Response> {
  const qs = new URLSearchParams(query).toString();
  return await this.fetch(`${this.orgUrl}/projects?${qs}`, { method: "GET" });
}

/**
 * Create a project within the configured organization for the client.
 */
async createProject(name?: string): Promise<Response> {
  return await this.fetch(`${this.orgUrl}/projects`, {
    method: "POST",
    body: JSON.stringify({ name }),
  });
}

/**
 * Get a list of deployments for the given project, with optional query params.
 */
// deno-lint-ignore no-explicit-any
async listDeployments(projectId: string, query?: any): Promise<Response> {
  const qs = new URLSearchParams(query).toString();
  return await this.fetch(`/projects/${projectId}/deployments?${qs}`, {
    method: "GET",
  });
}

/**
 * Get a list of logs for the given deployment, with optional query params
 */
// deno-lint-ignore no-explicit-any
async listAppLogs(deploymentId: string, query?: any): Promise<Response> {
  const qs = new URLSearchParams(query).toString();
  return await this.fetch(`/deployments/${deploymentId}/app_logs?${qs}`, {
    method: "GET",
  });
}

/**
 * Create a new deployment for the given project by ID.
 */
async createDeployment(
  projectId: string,
  // deno-lint-ignore no-explicit-any
  deploymentOptions: any,
): Promise<Response> {
  return await this.fetch(`/projects/${projectId}/deployments`, {
    method: "POST",
    body: JSON.stringify(deploymentOptions),
  });
}
```

For the full `subhosting.ts` code, which includes TSDoc style comments, please refer to the GitHub repo. If you're interested in digging into the Subhosting API endpoints, check out our API reference.

The logic for our server's route handlers should finally be complete. The next step is to define our frontend components.

Building the frontend in `App.tsx`

Let's create the `App` JSX component, which we import in `main.tsx`.

This is a simple server-side rendered JSX component. A few things to point out:

1. There are two `<script>` tags that import:
   - `/ace/ace.js`, which is a full featured in-browser IDE library, and
   - `app.js`, some vanilla JavaScript for naive client-side interactions, which we'll dive into later
2. The only props passed into this component are `projects`, which is an array of objects representing your Subhosting projects. We'll use `map` to return a list of `<option>` elements, which is added to a `<select>` element:
3. Note that the `<div id="deployments">` is the parent element of the list of deployments. We'll use vanilla JavaScript in `app.js` to continuously set its the `innerHTML`.

Your `App.tsx` should look something like this:

```
/** @jsx jsx */
import { jsx } from "$hono/jsx/index.ts";

// deno-lint-ignore no-explicit-any
export default function App({ projects }: { projects?: any }) {
  // deno-lint-ignore no-explicit-any
  const projList = projects?.map((p: any) => {
    return <option value={p.id}>{p.name}</option>;
  });

  return (
    <html>
      <head>
        <title>Basic Browser IDE (Deno Subhosting)</title>
        <link rel="stylesheet" href="/styles.css" />
```

```
        <script src="/ace/ace.js"></script>
        <script src="/app.js"></script>
      </head>
      <body>
        <nav>
          <h1>
            Basic Browser IDE
          </h1>
          <div id="project-selector">
            <select id="project-list">
              {projList}
            </select>
            <form action="/project" method="POST">
              <button type="submit" id="new-project">
                Generate New Project
              </button>
            </form>
          </div>
        </nav>
        <main>
          <div style="position:relative;height:100%;width:100%;">
            <div id="editor-container">
              <div id="editor"></div>
            </div>
            <div id="deployments-container">
              <h3>Deployments</h3>
              <div id="deployments"></div>
            </div>
            <button id="deploy-button">Save & Deploy</button>
          </div>
        </main>
      </body>
    </html>
  );
}
```

Next, let's create our client-side JavaScript.

Client-side JavaScript with Ace and `app.js`

Let's create a new directory, `static`, in which we'll add:

- the ace library
- app.js
- styles.css

Let's start with `app.js`. When the window loads, we'll need to initialize the editor, bind event handlers to the `#deploy-button` and `#project-list`, and call `pollData()` (which we'll define shortly) every five seconds to get a list of deployments for the current `projectId`:

```
let editor;

window.onload = function () {
  // Initialize editor
  editor = ace.edit("editor");
  editor.session.setTabSize(2);
  editor.setTheme("ace/theme/chrome");
  editor.session.setMode("ace/mode/typescript");
  editor.setValue(
    `Deno.serve(() => {
  console.log("Responding hello...");
  return new Response("Hello, subhosting!");
});`,
    -1,
  );

  // Attach event handler for deploy button
  document.getElementById("deploy-button").addEventListener(
    "click",
    saveAndDeploy,
  );

  // Immediately refresh deployments when new project selected
  document.getElementById("project-list").addEventListener("change", pollData);

  // Poll for deployment and log data for the selected project
  setInterval(pollData, 5000);
  pollData();
};
```

Next, let's define the following functions:

- `pollData`: get a list of deployments from the `/deployments` endpoint given the current `projectId` and display them with `setDeployments`
- `saveAndDeploy`: grab the `projectId` and `code` then create a deployment with a `POST` request to the `/deployment` endpoint
- `getProjectId`: get the project ID from the `<select id="project-list">`
- `setDeployments`: given an array of deployments, create the HTML needed to display deployment information, such as a link to the deployment URL, the deployment status, and when the deployment was created

```
async function pollData() {
  const projectId = getProjectId();

  try {
    // Get list of all deployments
    const dr = await fetch(`/deployments?projectId=${projectId}`);
    const deployments = await dr.json();
    setDeployments(deployments);
  } catch (e) {
    console.error(e);
  }
}

async function saveAndDeploy(e) {
  const $t = document.getElementById("deployments");
  const currentHtml = $t.innerHTML;
  $t.innerHTML = "<p>Creating deployment...</p>" + currentHtml;

  const projectId = getProjectId();

  const dr = await fetch(`/deployment`, {
    method: "POST",
    body: JSON.stringify({
      projectId,
      code: editor.getValue(),
    }),
  });
  const deployResult = await dr.json();
}

function getProjectId() {
  const $project = document.getElementById("project-list");
  return $project.value;
}

function setDeployments(deployments) {
  const $t = document.getElementById("deployments");

  if (!deployments || deployments.length < 1) {
    $t.innerHTML = "<p>No deployments for this project.</p>";
  } else {
    let html = "";
    deployments.forEach((deployment) => {
      html += `<div class="deployment-line">
        <a href="https://${deployment.domains[0]}" target="_blank">
          ${deployment.domains[0] || "URL pending..."}
        </a>
        <span class="timestamp">
          <span class="status ${deployment.status}">${deployment.status}</span>
          ${deployment.updatedAt}
        </span>
      </div>`;
    });
    $t.innerHTML = html;
```

```
    }
  }
```

With all of that, your app should be complete. To start the server, run the command `deno task dev`.

A note on deployment limitations

As of January 2024, active deployments are capped at 50 per day on the free Subhosting plan. This may cause problems during your testing, as each time you save your code in the browser, a new deployment is created. We are working on changes to our billing structure to avoid this problem, but if you run into any problems with this limit, please reach out to support@deno.com.

What's next?

Cloud IDEs are becoming more common as a frictionless way to edit, write, and deploy code. They can improve the developer experience in situations where your developer would need to build and setup a server in their own workflow outside of your product. While you could build your own infrastructure to deploy and run third party code, you would also have to maintain and scale it, as well as consider the security implications of running untrusted code.

Building a cloud IDE to deploy and run third party code is made simple with Deno Subhosting, which is designed for maximum security and can initiate deployments programmatically via a REST API. We hope this tutorial and starter template is a good foundation to build your advanced cloud IDEs or integrate a cloud IDE into your product.

All systems operational