

如何使用 Oak 和 Deno KV 构建 CRUD API

Andy Jiang 前端与Deno 2023年06月16日 08:00 北京

原文: <https://deno.com/blog/build-crud-api-oak-denokv>

作者: [Andy Jiang^{\[1\]}](#) 2023.06.01

Deno KV^[2]是首批直接内置到运行时中的数据库之一。这意味着您无需执行任何额外步骤（例如配置数据库或复制和粘贴 API 密钥）来构建有状态应用程序。要打开与数据存储的连接，您只需编写：

```
const kv = await Deno.openKv();
```

64M

除了作为具有简单而灵活的 **API^[3]** 的键值存储之外，它还是一个具有**原子事务^[4]**、**一致性控制^[5]**和尖端性能的生产就绪数据库。通过本入门教程，您将学习如何使用 Deno KV 构建一个用**Oak^[6]**编写的简单的有状态 CRUD API。我们将涵盖：

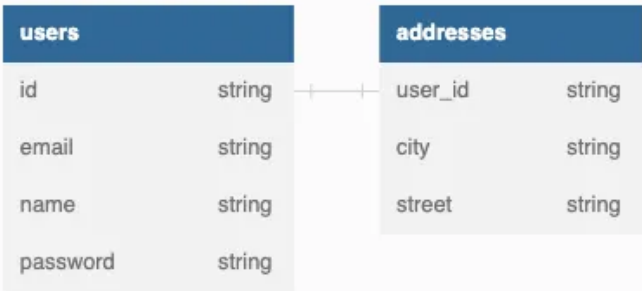
- 二级索引^[7]
- 原子交易^[8]
- 列表和分页^[9]

在我们开始之前，Deno KV 目前可以 `--unstable` 在**Deno 1.33^[10]**及更高版本中使用该标志。如果您有兴趣在 Deno Deploy 上使用 Deno KV，[请加入候补名单^{\[11\]}](#)，因为它仍处于封闭测试阶段。

按照下面的操作或 [查看源代码^{\[12\]}](#)。

设置数据库模型

这个 API 相当简单，使用两个模型，每个模型user都有一个可选的地址。



在一个新的 **repo** 中，创建一个 **db.ts** 文件，其中将包含数据库的所有信息和逻辑。让我们从类型定义开始：

```
export interface User {  
  id: string;  
  email: string;  
  name: string;  
  password: string;  
}  
  
export interface Address {  
  city: string;  
  street: string;  
}
```

创建 API 路由

接下来，让我们创建具有以下功能的 API 路由：

- 更新用户
- 更新绑定到用户的地址
- 列出所有用户
- 按 ID 列出单个用户
- 通过电子邮件列出单个用户
- 按用户 ID 列出地址
- 删除用户和任何关联的地址

我们可以使用自带 **Router**^[13]。

让我们创建一个新文件 **main.ts** 并添加以下路由。我们暂时将路由处理程序中的一些逻辑保留为空白：

```
import {  
  Application,  
  Context,  
  helpers,  
  Router,  
} from "https://deno.land/x/oak@v12.4.0/mod.ts";  
  
const { getQuery } = helpers;  
const router = new Router();  
  
router  
  .get("/users", async (ctx: Context) => {  
  })
```

```
.get("/users/:id", async (ctx: Context) => {
  const { id } = getQuery(ctx, { mergeParams: true });
})
.get("/users/email/:email", async (ctx: Context) => {
  const { email } = getQuery(ctx, { mergeParams: true });
})
.get("/users/:id/address", async (ctx: Context) => {
  const { id } = getQuery(ctx, { mergeParams: true });
})
.post("/users", async (ctx: Context) => {
  const body = ctx.request.body();
  const user = await body.value;
})
.post("/users/:id/address", async (ctx: Context) => {
  const { id } = getQuery(ctx, { mergeParams: true });
  const body = ctx.request.body();
  const address = await body.value;
})
.delete("/users/:id", async (ctx: Context) => {
  const { id } = getQuery(ctx, { mergeParams: true });
});

const app = new Application();

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

await app.listen({ port: 8000 });
```

接下来，让我们通过编写数据库函数来深入了解 Deno KV。

Deno KV

回到我们的db.ts文件，让我们开始在类型定义下添加数据库辅助函数。

```
const kv = await Deno.openKv();

export async function getAllUsers() {
}

export async function getUserById(id: string): Promise<User> {
}

export async function getUserByEmail(email: string) {
}

export async function getAddressByUserId(id: string) {
}
```

```
export async function upsertUser(user: User) {  
}  
  
export async function updateUserAndAddress(user: User, address: Address) {  
}  
  
export async function deleteUserById(id: string) {  
}
```

让我们从填写开始getUserById:

```
export async function getUserById(id: string): Promise<User> {  
  const key = ["user", id];  
  return (await kv.get<User>(key)).value!;  
}
```

这相对简单，我们使用键前缀"user"和带有kv.get()的id。

但是我们如何添加呢getUserByEmail?

添加二级索引

二级索引^[14]是不是主索引的索引，可能包含重复项。在这种情况下，我们的二级索引是email。

由于 Deno KV 是一个简单的键值存储，我们将创建第二个键前缀， "user_by_email"它用于email创建键并返回关联的用户id。这是一个例子：

```
const user = (await kv<User>.get(["user", "1"])).value!;  
// {  
//   "id": "1",  
//   "email": "andy@deno.com",  
//   "name": "andy",  
//   "password": "12345"  
// }  
  
const id = (await kv.get(["user_by_email", "andy@deno.com"])).value;  
// 1
```

然后，为了获取User，我们将在第一个索引上执行单独的kv.get()。

有了这两个索引，我们现在可以写getUserByEmail:

```
const id = (await kv.get(["user_by_email", email])).value;
```

```
export async function getUserByEmail(email: string) {  
  const userByEmailKey = ["user_by_email", email];  
  const id = (await kv.get(userByEmailKey)).value as string;  
  const userKey = ["user", id];  
  return (await kv<User>.get(userKey)).value!;  
}
```

现在，当我们`upsertUser`时，我们必须更新主键前缀`user`中的`"user"`。如果`email`不同，那么我们还必须更新辅助键前缀`"user_by_email"`。

但是，当两个更新事务同时发生时，我们如何确保我们的数据不会不同步呢？

使用原子事务

我们将使用`kv.atomic()`^[15]，它保障事务中的所有操作都成功完成，或者事务在发生故障时回滚到其初始状态，而数据库保持不变。

以下是我们的定义`upsertUser`：

```
export async function upsertUser(user: User) {  
  const userKey = ["user", user.id];  
  const userByEmailKey = ["user_by_email", user.email];  
  
  const oldUser = await kv.get<User>(userKey);  
  
  if (!oldUser.value) {  
    const ok = await kv.atomic()  
      .check(oldUser)  
      .set(userByEmailKey, user.id)  
      .set(userKey, user)  
      .commit();  
    if (!ok) throw new Error("Something went wrong.");  
  } else {  
    const ok = await kv.atomic()  
      .check(oldUser)  
      .delete(["user_by_email", oldUser.value.email])  
      .set(userByEmailKey, user.id)  
      .set(userKey, user)  
      .commit();  
    if (!ok) throw new Error("Something went wrong.");  
  }  
}
```

我们首先获取 `oldUser` 并检查其是否存在。如果不存在，则使用用户和用户 ID 对 `"user"` 和 `"user_by_email"` 的键前缀进行 `.set()`。否则，由于用户的电子邮件可能已更改，我们通过删除键 `["user_by_email", oldUser.value.email]` 上的值来删除 `"user_by_email"` 中的值。

我们使用 `.check(oldUser)` 来执行所有这些操作以确保另一个客户端没有更改值。否则，我们容易受到竞争条件的影响，从而导致更新错误的记录。如果 `.check()` 通过并且值保持不变，则可以使用 `.set()` 和 `.delete()` 完成事务。

`kv.atomic()` 是确保正确性的绝佳方法，特别是在多个客户端发送写事务的情况下，例如银行/金融和其它数据敏感应用程序。

列表和分页

接下来，让我们定义 `getAllUsers`。我们可以用 来做到这一点 `kv.list()`^[16]，它返回一个键迭代器，我们可以枚举它来获取我们 `push()` 放入 `users` 数组的值：

```
export async function getAllUsers() {
  const users = [];
  for await (const res of kv.list({ prefix: ["user"] })) {
    users.push(res.value);
  }
  return users;
}
```

请注意，这个简单的函数遍历并返回整个 KV 存储。如果此 API 与前端交互，我们可以传递一个 `{ limit: 50 }` 选项来检索前 50 个项目：

```
let iter = await kv.list({ prefix: ["user"] }, { limit: 50 });
```

当用户需要更多数据时，使用以下方法检索下一批数据 `iter.cursor`：

```
iter = await kv.list({ prefix: ["user"] }, { limit: 50, cursor: iter.cursor });
```

添加第二个模型，Address

让我们将第二个模型添加 `Address` 到我们的数据库中。我们将使用一个新的键前缀，`"user_address"` 后跟标识符 `user_id` (`["user_address", user_id]`) 作为这两个 KV 子空间之间的“连接”。

现在，让我们编写 `getAddressByUser` 函数：

```
let iter = await kv.list({ prefix: ["user_address", user_id] });
```

```
export async function getAddressByUserId(id: string) {  
  const key = ["user_address", id];  
  return (await kv<Address>.get(key)).value!;  
}
```

我们可以编写`updateUserAndAddress`函数。请注意，我们需要使用`kv.atomic()`，因为我们想要更新三个键前缀为"user"，"user_by_email"和"user_address"的KV条目。

```
export async function updateUserAndAddress(user: User, address: Address) {  
  const userKey = ["user", user.id];  
  const userByEmailKey = ["user_by_email", user.email];  
  const addressKey = ["user_address", user.id];  
  
  const oldUser = await kv.get<User>(userKey);  
  
  if (!oldUser.value) {  
    const ok = await kv.atomic()  
      .check(oldUser)  
      .set(userByEmailKey, user.id)  
      .set(userKey, user)  
      .set(addressKey, address)  
      .commit();  
  
    if (!ok) throw new Error("Something went wrong.");  
  } else {  
    const ok = await kv.atomic()  
      .check(oldUser)  
      .delete(["user_by_email", oldUser.value.email])  
      .set(userByEmailKey, user.id)  
      .set(userKey, user)  
      .set(addressKey, address)  
      .commit();  
  
    if (!ok) throw new Error("Something went wrong.");  
  }  
}
```

添加`kv.delete()`

最后，为了完善我们应用程序的CRUD功能，让我们定义`deleteByUserId`。类似于其他变异函数，我们将检索`userRes`并在`.delete()`三个键之前使用`.atomic().check(userRes)`。

```
export async function deleteUserById(id: string) {  
  const userKey = ["user", id];  
  const userRes = await kv.get(userKey);  
  if (!userRes.value) return;  
  const userByEmailKey = ["user_by_email", userRes.value.email];  
  const addressKey = ["user_address", id];
```

```
await kv.atomic()
  .check(userRes)
  .delete(userKey)
  .delete(userByEmailKey)
  .delete(addressKey)
  .commit();
}
```

更新路由处理程序

现在我们已经定义了数据库函数，让我们导入它们到 `main.ts` 并在我们的路由处理程序中填写其余的功能。这是完整的 `main.ts` 文件：

```
import {
  Application,
  Context,
  helpers,
  Router,
} from "https://deno.land/x/oak@v12.4.0/mod.ts";

import {
  deleteUserById,
  getAddressByUserId,
  getAllUsers,
  getUserByEmail,
  getUserById,
  updateUserAndAddress,
  upsertUser,
} from "./db.ts";

const { getQuery } = helpers;
const router = new Router();

router
  .get("/users", async (ctx: Context) => {
    ctx.response.body = await getAllUsers();
  })
  .get("/users/:id", async (ctx: Context) => {
    const { id } = getQuery(ctx, { mergeParams: true });
    ctx.response.body = await getUserById(id);
  })
  .get("/users/email/:email", async (ctx: Context) => {
    const { email } = getQuery(ctx, { mergeParams: true });
    ctx.response.body = await getUserByEmail(email);
  })
  .get("/users/:id/address", async (ctx: Context) => {
    const { id } = getQuery(ctx, { mergeParams: true });
    ctx.response.body = await getAddressByUserId(id);
  })
  .post("/users", async (ctx: Context) => {
```



```
const body = ctx.request.body();
const user = await body.value;
await upsertUser(user);
})
.post("/users/:id/address", async (ctx: Context) => {
  const { id } = getQuery(ctx, { mergeParams: true });
  const body = ctx.request.body();
  const address = await body.value;
  const user = await getUserById(id);
  await updateUserAndAddress(user, address);
})
.delete("/users/:id", async (ctx: Context) => {
  const { id } = getQuery(ctx, { mergeParams: true });
  await deleteUserById(id);
});

const app = new Application();

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

await app.listen({ port: 8000 });
```

测试我们的 API

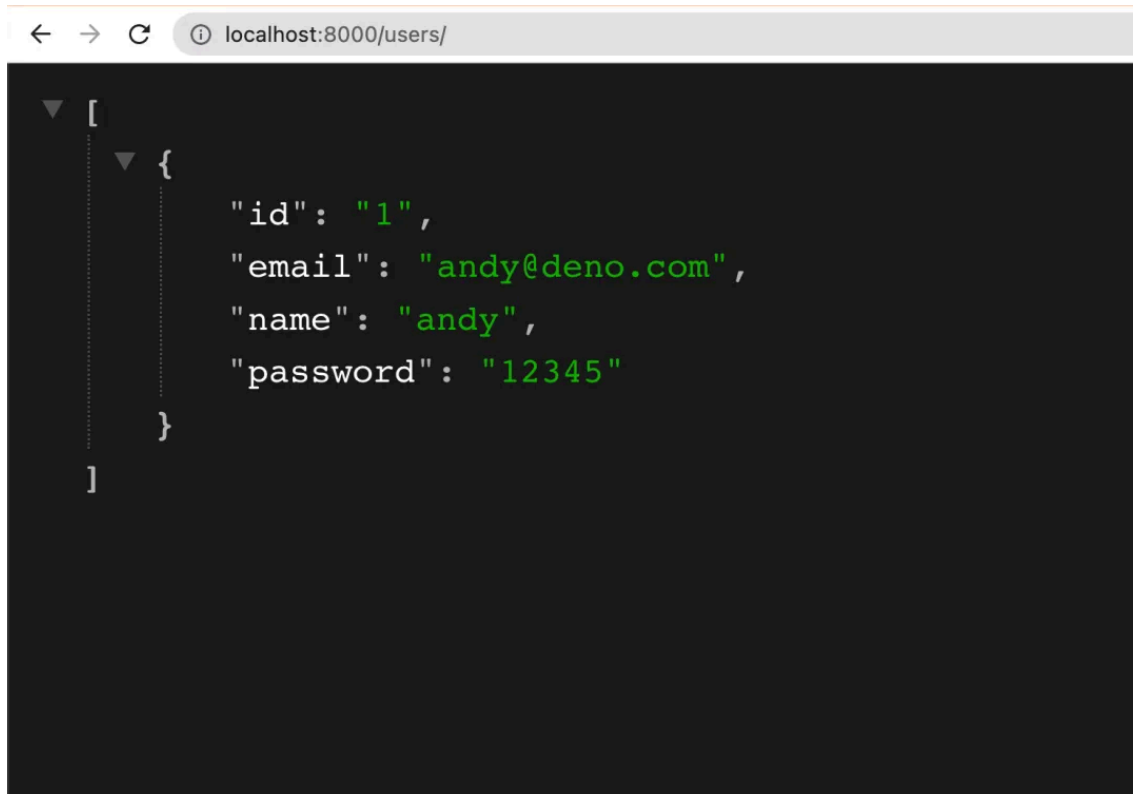
让我们运行我们的应用程序并测试它。运行它：

```
deno run --allow-net --watch --unstable main.ts
```

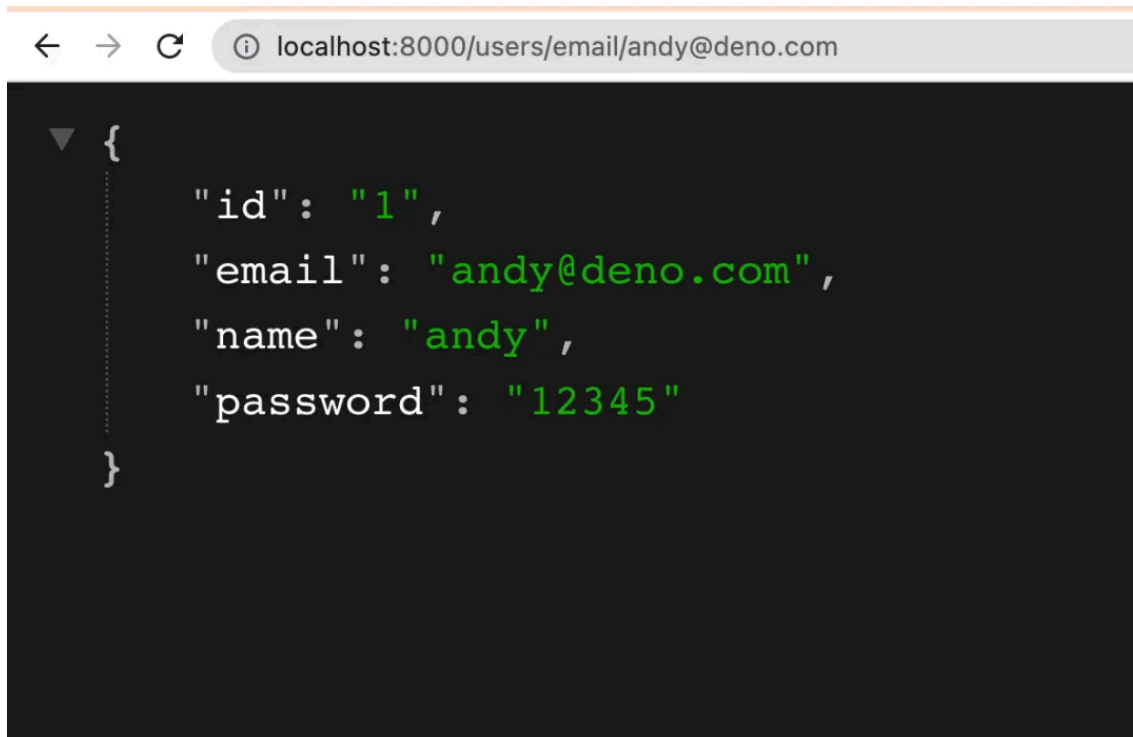
我们可以使用 CURL 测试我们的应用程序。让我们添加一个新用户：

```
curl -X POST http://localhost:8000/users -H "Content-Type: application/json" -d '{"id": "1", "em
```

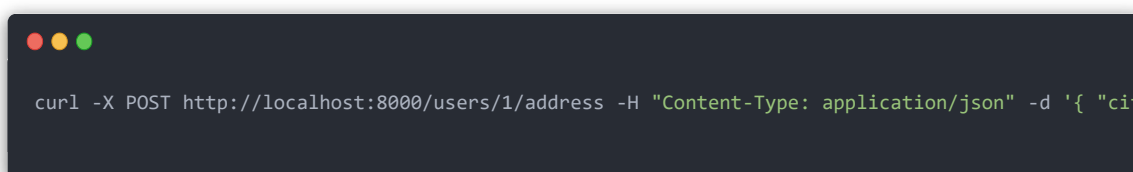
当我们将浏览器指向localhost:8000/users时，我们应该看到：



让我们看看是否可以通过将浏览器指向以下地址来通过电子邮件检索用户
localhost:8000/users/email/andy@deno.com:



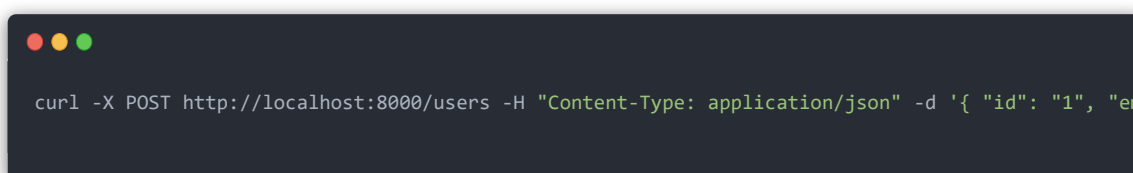
让我们发送一个 POST 请求来为这个用户添加一个地址:



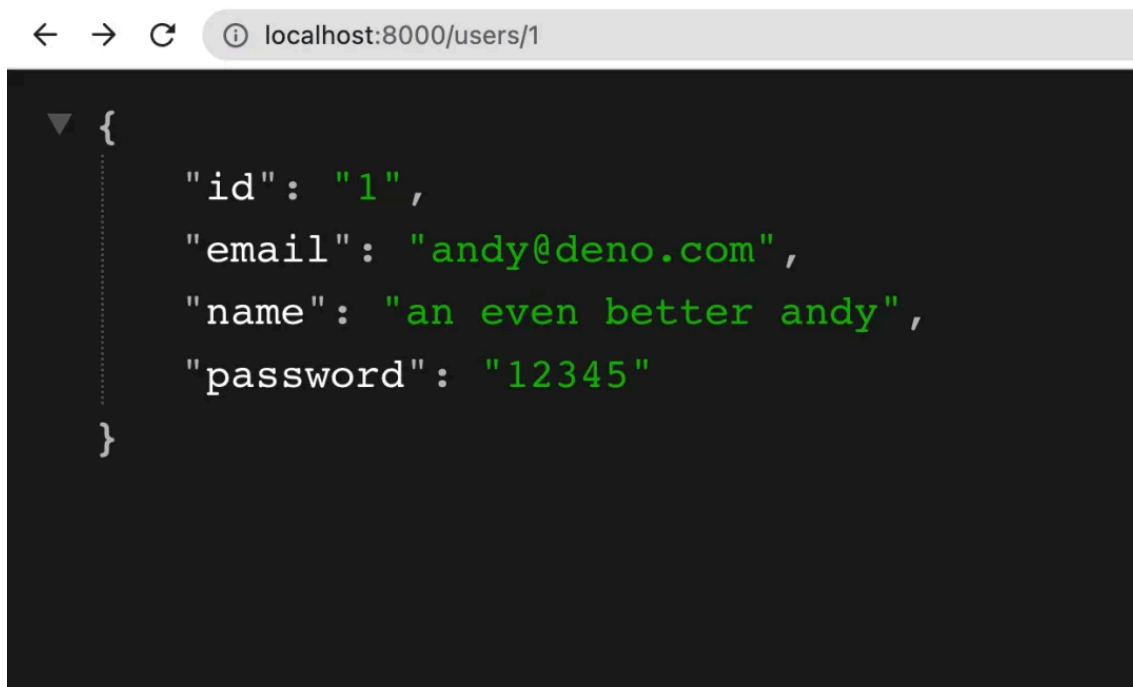
让我们看看这是否有效localhost:8000/users/1/address:



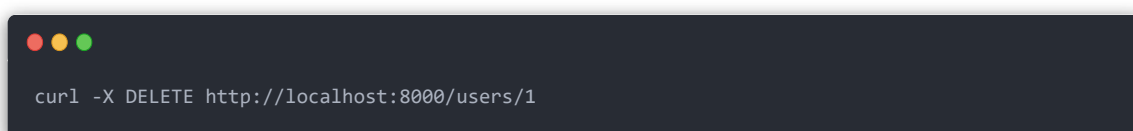
让我们使用新名称更新具有 id 的同一用户：



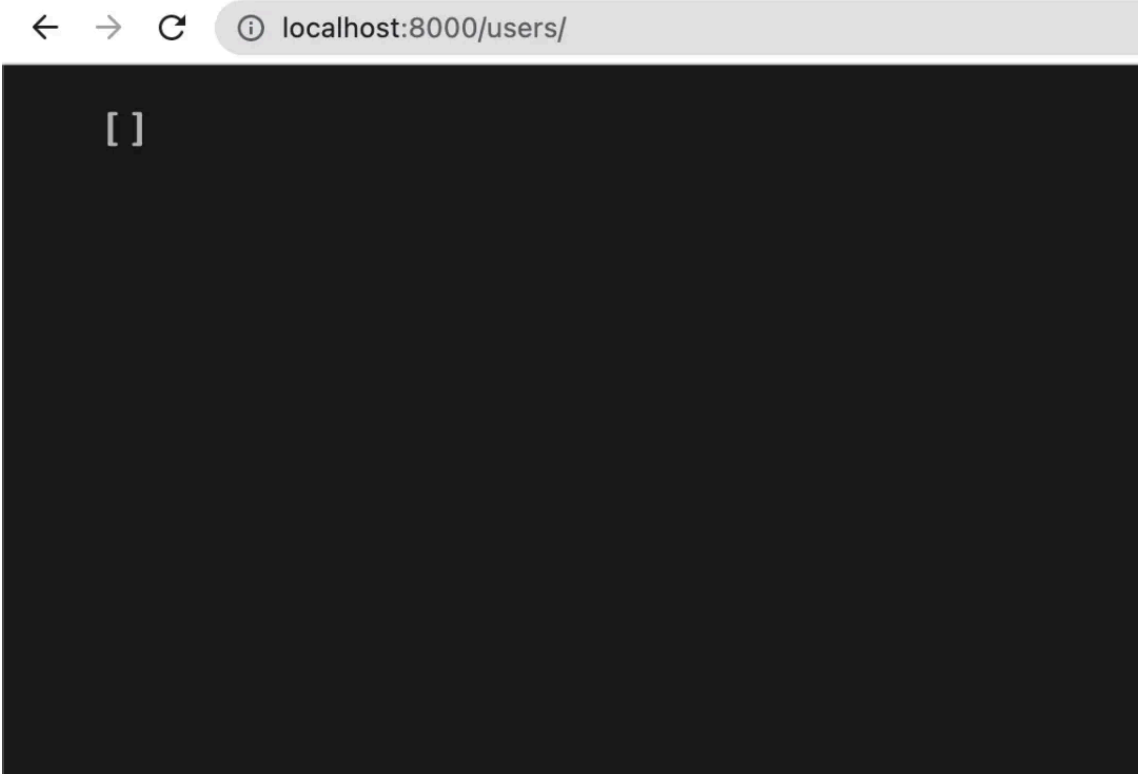
我们可以在浏览器中看到该更改localhost:8000/users/1:



最后，让我们删除用户：



当我们将浏览器指向localhost:8000/users时，我们应该什么也看不到：

A screenshot of a web browser window. The address bar shows 'localhost:8000/users/'. The main content area is dark and displays a large, white, empty array notation '[]'.

下一步是什么

这只是对使用 Deno KV 构建有状态 API 的介绍，但希望您能看到它是多么快速和容易上手。

使用此 CRUD API，您可以创建一个简单的前端客户端来与数据进行交互。

参考资料

- [1] Andy Jiang GitHub地址: <https://github.com/lambtron>
- [2] Deno KV: <https://deno.com/kv>
- [3] Deno KV API: <https://deno.com/manualruntime/kv/operations>
- [4] 原子事务: <https://deno.com/manual/runtime/kv/transactions>
- [5] 一致性控制: <https://deno.com/manual/runtime/kv/operations#get>
- [6] Oak: <https://deno.land/x/oak>
- [7] 二级索引: <https://deno.com/blog/build-crud-api-oak-denokv#add-a-secondary-index>
- [8] 原子交易: <https://deno.com/blog/build-crud-api-oak-denokv#use-atomic-transactions>
- [9] 列表和分页: <https://deno.com/blog/build-crud-api-oak-denokv#list-and-pagination>
- [10] Deno 1.33: <https://deno.com/blog/v1.33>
- [11] 请加入候补名单: <https://deno.com/kv>
- [12] with-oak-deno-kv源代码: <https://github.com/denoland/examples/tree/main/with-oak-deno-kv>
- [13] Oak Router: <https://deno.land/x/oak#router>
- [14] 索引: https://deno.com/manual@/runtime/kv/secondary_indexes
- [15] kv.atomic(): <https://deno.com/manual/runtime/kv/transactions>
- [16] kv.list(): <https://deno.com/manual/runtime/kv/operations#list>

[# Deno 73](#) [# Deno官方博客 76](#)

Deno · 目录 ≡

[< 上一篇](#)

Deno使用Puppeteer开发实践

[下一篇 >](#)

Fresh 1.2 - 欢迎全职维护者、岛屿之间共享
状态、有限的 npm 支持等更新

[Read more](#)