

# 16-Simple-CRUD-Full-Version

1. Login - using google - [noor.tushar.khan@gmail.com](mailto:noor.tushar.khan@gmail.com)

The screenshot shows the MongoDB Atlas interface for a project named "Project 0". The left sidebar has a red box around the "Database" item under the "DEPLOYMENT" section. The main area displays "NOORTUSHAR'S ORG - 2024-04-20 > PROJECT 0" and "Database Deployments". It features a green icon of two stacked cylinders with a plus sign. Below it is the text "Create a database" and "Choose your cloud provider, region, and specs.". A red box highlights the "Build a Database" button. At the bottom, a note says "Once your database is up and running, live migrate an existing MongoDB database into Atlas with our [Live Migration Service](#)".

2.

The screenshot shows the MongoDB Atlas interface with a modal dialog titled "Connect to Cluster0". The dialog is divided into three steps: 1. Set up connection security (done), 2. Choose a connection method (not yet selected), and 3. Connect (not yet selected). Step 1 contains instructions to secure the cluster and add a connection IP address. Step 2 contains instructions to create a database user, mentioning "atlasAdmin" permissions. Step 3 contains fields for "Username" (noortusharkhan) and "Password" (ChEzrwzmrjHO93kU), with a "Copy" button. At the bottom are "Cancel" and "Choose a connection method" buttons.

3.

4. username: noortusharkhan
- password: ChEzrwzmrjHO93kU
5. go back to terminal

create a server

named example - simple-crud-server

```
npm init
npm i express cors mongo
```

6. package.json -> "scripts" : { "start": "node index.js" }

7. index.js ->

```
const express = require("express");
const cors = require("cors");

const app = express();

const port = process.env.PORT || 3000;

//middlewares
app.use(cors());
app.use(express.json());

//mongodb

//routes

app.get("/", (req, res) => {
    res.send("Simple Crud Server is working fine");
});

app.listen(port, () => {
    console.log(`app running at port: ${port}`);
});
```

The screenshot shows the MongoDB Atlas Network Access page. On the left, there's a sidebar with various project management and security options. The main area is titled 'Network Access' and contains an 'IP Access List'. A table lists one IP address: '27.147.202.116/32 (includes your current IP address)'. The table includes columns for 'IP Address', 'Comment', 'Status', and 'Actions'. A red box highlights the '+ ADD IP ADDRESS' button in the top right corner of the table header. Another red box highlights the 'Network Access' link in the sidebar.

IP Address	Comment	Status	Actions
27.147.202.116/32 (includes your current IP address)	Created as part of the Auto Setup process	Active	<button>+ EDIT</button> <button>DELETE</button>

## Add IP Access List Entry

Atlas only allows client connections to a cluster from entries in the project's IP Access List. Each entry should either be a single IP address or a CIDR-notated range of addresses. [Learn more.](#)

**ALLOW ACCESS FROM ANYWHERE**

**Access List Entry:** 0.0.0.0/0

**Comment:** Optional comment describing this entry

This entry is temporary and will be deleted in **6 hours**

**Cancel** **Confirm**

Project 0 Data Services App Services Charts

Overview DEPLOYMENT Database SERVICES Device Sync Triggers Data API Data Federation Atlas Search Stream Processing Migration SECURITY Quickstart

NOORTUSHAR'S ORG - 2024-04-20 > PROJECT 0

### Database Deployments

Find a database deployment... Edit Config + Create

**Cluster0** Connect View Monitoring Browse Collections ...

Sample dataset successfully loaded. Access it in Collections or by connecting with the MongoDB Shell.

**Visualize Your Data**

Build dashboards and charts, and embed them in your apps with MongoDB Charts.

R 0 W 0 Lost 17 minutes 218.4/s

Connections 0 Last 17 minutes 1.0

In 0.0 B/s Out 0.0 B/s Lost 17 minutes 460.4 KB/s

Data Size 134.4 MB / 512.0 MB (26%) Last 17 minutes 512.0 MB

## Connect to Cluster0

- Set up connection security
- Choose a connection method
- Connect

### Connect to your application

**Drivers** Access your Atlas data using MongoDB's native drivers (e.g. Node.js, Go, etc.)

### Access your data through tools

**Data Explorer** Browse your Atlas collections without leaving the UI

# Connecting with MongoDB Driver

## 1. Select your driver and version

We recommend installing and using the latest driver version.

Driver	Version
Node.js	5.5 or later

## 2. Install your driver

Run the following on the command line

```
npm install mongodb
```



[View MongoDB Node.js Driver installation instructions.](#)

## 3. Add your connection string into your application code



[View full code sample](#)



```
const { MongoClient, ServerApiVersion } = require('mongodb');
const uri = "mongodb+srv://noortusharkhan:<password>@cluster0.j7c4zww.mongodb.net/?retryWrites=true&w=majority";

// Create a MongoClient with a MongoClientOptions object to set the Stable API version
const client = new MongoClient(uri, {
  serverApi: {
    version: ServerApiVersion.v1,
    strict: true,
    deprecationErrors: true,
  }
});

async function run() {
  try {
    await client.connect();
    const database = client.db("test");
    const collection = database.collection("test");
    await collection.insertOne({ name: "Mango" });
    const result = await collection.find({ name: "Mango" }).toArray();
    console.log(result);
    await client.close();
  } catch (err) {
    console.error(err);
  }
}

run();
```

10. Back to server index.js and paste it

also make sure to replace the password with the actual password

```
12 //mongodb
13
14 const { MongoClient, ServerApiVersion } = require("mongodb");
15 const uri =
16   "mongodb+srv://noortusharkhan:ChEzrwzmrjH093kU@cluster0.j7c4zww.mongodb.net/?retryWrites=true&
17   w=majority&appName=Cluster0";
18
19 // Create a MongoClient with a MongoClientOptions object to set the Stable API version
20 const client = new MongoClient(uri, {
21   serverApi: [
22     {
23       version: ServerApiVersion.v1,
24       strict: true,
25       deprecationErrors: true,
26     },
27   ],
28 });
29
30 async function run() {
31   try {
32     // Connect the client to the server (optional starting in v4.7)
33     await client.connect();
34     // Send a ping to confirm a successful connection
35     await client.db("admin").command({ ping: 1 });
36     console.log(
37       "Pinged your deployment. You successfully connected to MongoDB!"
38     );
39   } finally {
40     // Ensures that the client will close when you finish/error
41     await client.close();
42   }
43   run().catch(console.dir);
44 }
45
46 //routes
```

Normally normal function and async await try catch finally er syntax ta ki?

```
// normal syntax of a function

function name() {
  //
}

name()

// asyn await function

function async func () {
  await //
}

func().catch(err=>console.log(err));

// try catch finally

try {
  // try use korle 'catch' ba 'finally' use korte hobe
  // jeita korar dorkar chilo oita korbe
} catch (error) {
  // korte jeye error face korle ai block e dhora khabe
} finally {
  // jeita korar chilo oita hok ba na hok error khak ba na khak
```

```
// finally ai block ta execute hobe
}
```

so similarly our mongo db code looks like this: an async function

```
16
17 // Create a MongoClient with a MongoClientOptions object to set the Stable API version
18 const client = new MongoClient(uri, {
19   serverApi: {
20     version: ServerApiVersion.v1,
21     strict: true,
22     deprecationErrors: true,
23   },
24 });
25
26 async function run() {
27   try {
28     // Connect the client to the server (optional starting in v4.7)
29     await client.connect();
30     // Send a ping to confirm a successful connection
31     await client.db("admin").command({ ping: 1 });
32     console.log(
33       "Pinged your deployment. You successfully connected to MongoDB!"
34     );
35   } finally {
36     // Ensures that the client will close when you finish/error
37     await client.close();
38   }
39 }
40 run().catch(console.log);
41
```

we can see successfully connected

```
[nodemon] restarting due to changes...
[nodemon] starting `node index.js`
app running at port: 3000
Pinged your deployment. You successfully connected to MongoDB!
[nodemon] restarting due to changes...
[nodemon] starting `node index.js`
app running at port: 3000
Pinged your deployment. You successfully connected to MongoDB!
```

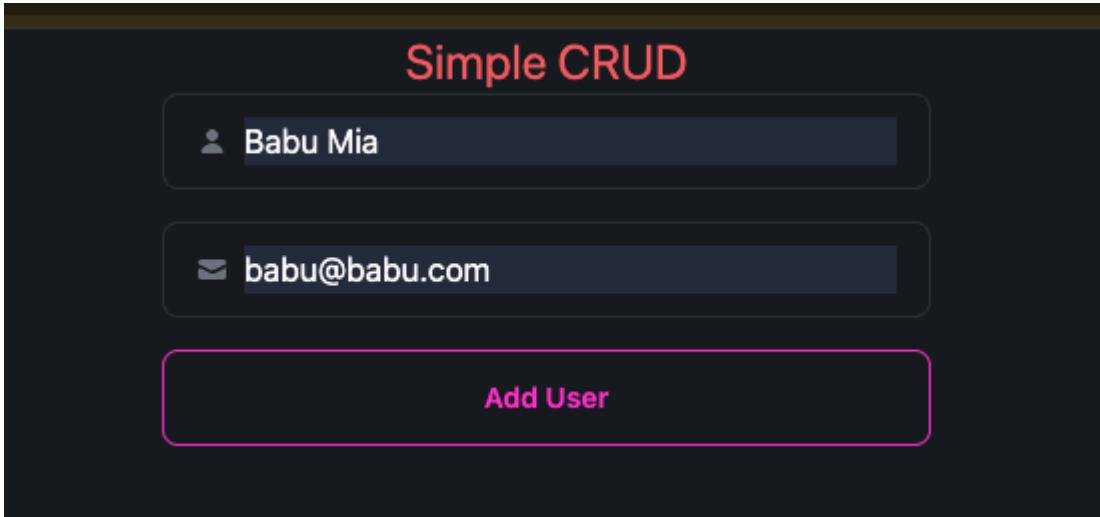
but we want to have the connection between our server and database not to end and thus will comment out  
await client.close()

```
    );
} finally {
  // Ensures that the client will close when you finish/error
  // await client.close();
}
}
```

CLIENT SIDE - lets create one

simple-crud-client

CLIENT SIDE - lets create one



Now we have to send our field data to the backend server then database.

So first we have to create an API:

so akhon kothae korbo api create ta? mogodb r function tar bhitore:

```

25
26  async function run() {
27    try {
28      // Connect the client to the server (optional starting in v4.7)
29      await client.connect();
30
31      // POST API
32      app.post("/users", async (req, res) => {
33        const user = req.body;
34        console.log("new user is here :", user);
35      });
36
37      // Send a ping to confirm a successful connection
38      await client.db("admin").command({ ping: 1 });
39      console.log(
40        "Pinged your deployment. You successfully connected to MongoDB!"

```

CLIENT side e jai akhon:

```

3 const Home = () => {
4     const handleAddUser = (event) => {
5         event.preventDefault();
6
7         const form = event.target;
8         const name = form.name.value;
9         const email = form.email.value;
10
11         const user = { name, email };
12         console.log(user);
13
14         fetch("http://localhost:3000/users", {
15             method: "POST",
16             headers: {
17                 "Content-Type": "application/json",
18             },
19             body: JSON.stringify(user),
20         })
21             .then((res) => res.json())
22             .then((data) => console.log(data))
23             .catch((err) => console.log(err));
24     };
25

```

Add user akhon Client side e click hoile => SERVER side er console e:

```

app running at port: 3000
Pinged your deployment. You successfully connected to MongoDB!
[nodemon] restarting due to changes...
[nodemon] starting `node index.js`
app running at port: 3000
Pinged your deployment. You successfully connected to MongoDB!
new user is here : { name: 'Babu Mia', email: 'babu@babu.com' }

```

Akhon kaaj hoilo server thike database e pathabo.. EXCITED !!!

<https://www.mongodb.com/docs/drivers/node/current/fundamentals/crud/>

The screenshot shows the MongoDB Documentation website for the Node.js Driver. The left sidebar has a red box around the 'Insert Operations' section, which contains a sub-section 'Insert a Document' also highlighted with a red box. The main content area displays the 'Insert a Document' page, which includes a red box around the first paragraph explaining the `collection.insertOne()` method. Below this, there's a section about query options and another about compatibility, both of which have red boxes around their respective descriptions.

```

1 import { MongoClient } from "mongodb";
2
3 // Replace the uri string with your MongoDB deployment's connection string.
4 const uri = "<connection string uri>";
5
6 // Create a new client and connect to MongoDB
7 const client = new MongoClient(uri);
8
9 async function run() {
10   try {
11     // Connect to the "insertDB" database and access its "haiku" collection
12     const database = client.db("insertDB");
13     const haiku = database.collection("haiku");
14
15     // Create a document to insert
16     const doc = {
17       title: "Record of a Shriveled Datum",
18       content: "No bytes, no problem. Just insert a document, in MongoDB",
19     }
20     // Insert the defined document into the "haiku" collection
21     const result = await haiku.insertOne(doc);
22
23     // Print the ID of the inserted document
24     console.log(`A document was inserted with the _id: ${result.insertedId}`)
25   } finally {
26     // Close the MongoDB client connection
27     await client.close();
28   }
29 }
30 // Run the function and handle any errors
31 run().catch(console.dir);

```



so we already have similar code setup in our express now going to edit here and there:

```

25
26 async function run() {
27   try {
28     // Connect the client to the server (optional starting in v4.7)
29     await client.connect();
30
31     // Database and Collection
32     const database = client.db("usersDB");
33     const userCollection = database.collection("users");
34
35     // POST API
36     app.post("/users", async (req, res) => {
37       const user = req.body;
38       console.log("new user is here :", user);
39       const result = await userCollection.insertOne(user);
40       res.send(result);
41     });
42
43     // Send a ping to confirm a successful connection
44     await client.db("admin").command({ ping: 1 });
45     console.log(
46       "Pinged your deployment. You successfully connected to MongoDB!"
47     );
48   } finally {
49     // Ensures that the client will close when you finish/error
50     // await client.close();
51   }
52 }
53 run().catch(console.dir);

```



UI:

The screenshot shows the browser's developer tools console tab. It displays the following log entries:

- [vite] connecting...
- [vite] connected.
- WebSocket connection to 'ws://127.0.0.1:5500/ws' failed.
- Third-party cookie will be blocked. Learn more in the Issues tab.
- Third-party cookie will be blocked. Learn more in the Issues tab.

Below the log, there are two entries representing inserted user documents:

- ▶ {name: '', email: ''}
- ▶ {acknowledged: true, insertedId: '6623b2cebb966590264ed0e7'}
- ▶ {name: 'Babu Mia', email: 'babu@babu.com'}
- ▶ {acknowledged: true, insertedId: '6623b2e4bb966590264ed0e8'}

Two red arrows point from the bottom of the question text to the 'insertedId' fields in the last two log entries.

in atlas database:

The screenshot shows the MongoDB Compass interface. The left sidebar shows the database structure:

- usersDB
- users

The main area shows the following details:

- STORAGE SIZE: 20KB
- LOGICAL DATA SIZE: 11IB
- TOTAL DOCUMENTS: 2
- INDEXES TOTAL SIZE: 20KB

Below this, there are tabs for Find, Indexes, Schema Anti-Patterns, Aggregation, and Search Indexes. An 'INSERT DOCUMENT' button is visible. A search bar at the top says 'Search Namespaces'.

The 'Find' tab is active, showing a 'Filter' field with the placeholder 'Type a query: { field: 'value' }'. Below it, the 'QUERY RESULTS: 1-2 OF 2' section displays two documents:

- ▶ \_id: ObjectId('6623b2cebb966590264ed0e7')  
name : ""  
email : ""
- ▶ \_id: ObjectId('6623b2e4bb966590264ed0e8')  
name : "Babu Mia"  
email : "babu@babu.com"

Yes we did it !

on the UI we are getting a response as data, and inside the data we have `insertedId`

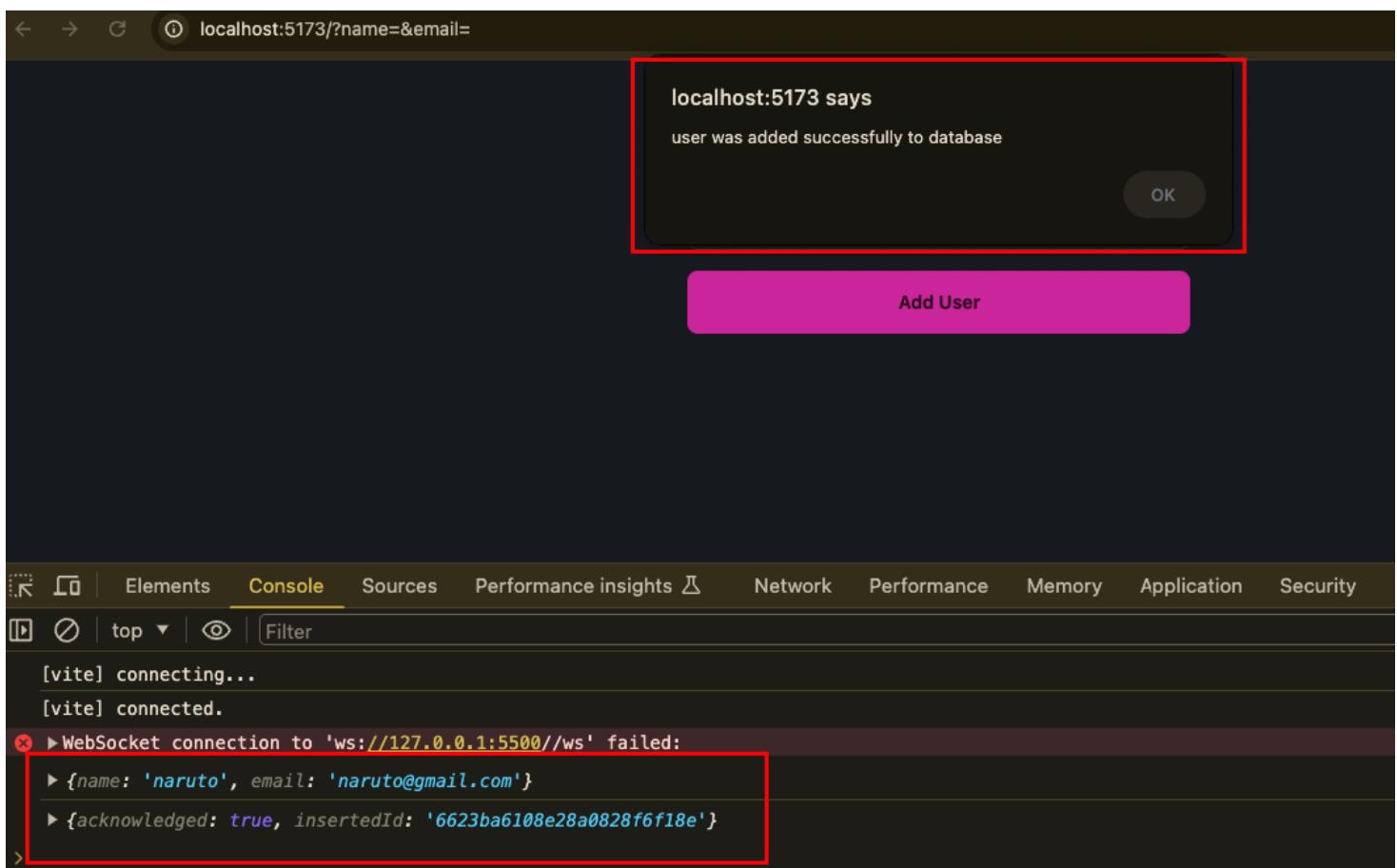
The screenshot shows the browser's developer tools console tab. It displays the following log entries:

- ▶ {name: '', email: ''}
- ▶ {acknowledged: true, insertedId: '6623b2cebb966590264ed0e7'}
- ▶ {name: 'Babu Mia', email: 'babu@babu.com'}
- ▶ {acknowledged: true, insertedId: '6623b2e4bb966590264ed0e8'}

The 'insertedId' field in the second entry is highlighted with a red box.

so we can write an if statement and an alert which would let us know that an user was added.

```
13
14     fetch("http://localhost:3000/users", {
15         method: "POST",
16         headers: {
17             "Content-Type": "application/json",
18         },
19         body: JSON.stringify(user),
20     })
21     .then((res) => res.json())
22     .then((data) => {
23         console.log(data);
24         if (data.insertedId) {
25             alert("user was added successfully to database");
26             form.reset();
27         }
28     })
29     .catch((err) => console.log(err)); TypeError { stack: 'TypeError: Failed
30
31     };
```



mongoDb atlas:

```
_id: ObjectId('6623b2cebb966590264ed0e7')
name : ""
email : ""
```

```
_id: ObjectId('6623b2e4bb966590264ed0e8')
name : "Babu Mia"
email : "babu@babu.com"
```

```
_id: ObjectId('6623ba6108e28a0828f6f18e')
name : "naruto"
email : "naruto@gmail.com"
```

Now our aim is to show multiple users when we go to the route "/users"

so first e amra GET users route ta kore feli server side e.

```
25
26 async function run() {
27   try {
28     // Connect the client to the server (optional starting in v4.7)
29     await client.connect();
30
31     // Database and Collection
32     const database = client.db("usersDB");
33     const userCollection = database.collection("users");
34
35     // GET API for Users
36     app.get('/users', (req, res) => {
37       // how to get the data?
38     })
39
40   }
41
42   // POST API
43   app.post("/users", async (req, res) => {
44     const user = req.body;
```

mongo r documentation:

[← Back To MongoDB Drivers](#)

## Node.js Driver

v6.5 (current) ▾

Quick Start

Quick Reference

What's New

Usage Examples

▼ Find Operations

Find a Document

Find Multiple Documents

▶ Insert Operations

▶ Update &amp; Replace Operations

Delete Operations

Count Documents

Retrieve Distinct Values of

## Find Multiple Documents

You can query for multiple documents in a collection with `collection.find()`. The `find()` method uses a query document that you provide to match the subset of the documents in the collection that match the query. If you don't provide a query document (or if you provide an empty document), MongoDB returns all documents in the collection. For more information on querying MongoDB, see our [documentation on query documents](#).

You can also define more query options such as `sort` and `projection` to configure the result set. You can specify these in the options parameter in your `find()` method call in `sort` and `projection` objects. See [collection.find\(\)](#) for more information on the parameters you can pass to the method.

The `find()` method returns a [FindCursor](#) that manages the results of your query. You can iterate through the matching documents using the `for await...of` syntax, or one of the following `cursor` methods:

- `next()`
- `toArray()`

If no documents match the query, `find()` returns an empty cursor.

## Compatibility

```
1
2 8  async function run() {
3 9    try {
4
5 10      // Get the database and collection on which to run the operation
6 11      const database = client.db("sample_mflix");
7 12      const movies = database.collection("movies");
8
9 13      // Query for movies that have a runtime less than 15 minutes
10 14      const query = { runtime: { $lt: 15 } };
11
12 15      const options = {
13        // Sort returned documents in ascending order by title (A->Z)
14        sort: { title: 1 },
15        // Include only the `title` and `imdb` fields in each returned document
16        projection: { _id: 0, title: 1, imdb: 1 },
17      };
18
19      // Execute query
20      const cursor = movies.find(query, options);
21
22      // Print a message if no documents were found
23      if ((await movies.countDocuments(query)) === 0) {
24        console.log("No documents found!");
25      }
26
27      // Print returned documents
28      for await (const doc of cursor) {
29        console.dir(doc);
30      }
31
32      // Finally, close the client
33    } finally {
34      await client.close();
35    }
36  }
```

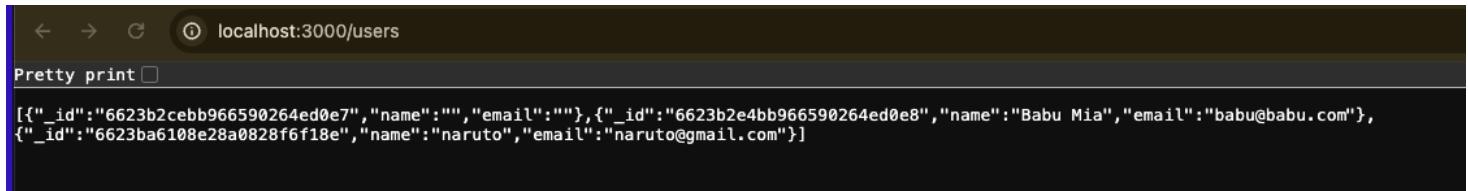
so amra pura mil na korleo amra amader moto code lekhbo:

```

26  async function run() {
27    try {
28      // Connect the client to the server (optional starting in v4.7)
29      await client.connect();
30
31      // Database and Collection
32      const database = client.db("usersDB");
33      const userCollection = database.collection("users");
34
35      // GET API for Users
36      app.get("/users", async (req, res) => {
37        // how to get the data?
38        const cursor = userCollection.find();
39        const result = await cursor.toArray();
40        res.send(result);
41      });
42
43      // POST API
44      app.post("/users", async (req, res) => {

```

apato amra find  
e kono argument  
pass na korei try  
kortesi

localhost:3000/users

Pretty print □

```
[{"_id": "6623b2cebb966590264ed0e7", "name": "", "email": ""}, {"_id": "6623b2e4bb966590264ed0e8", "name": "Babu Mia", "email": "babu@babu.com"}, {"_id": "6623ba6108e28a0828f6f18e", "name": "naruto", "email": "naruto@gmail.com"}]
```

on client side:

```
{
  path: "/users",
  element: <Users></Users>,
  loader: () => {
    return fetch("http://localhost:3000/users");
  },
}
```

```

1 import { useLoaderData } from "react-router-dom";
2
3 const Users = () => {
4     const users = useLoaderData();
5     return (
6         <div className="text-center py-[40px]">
7             <h3>Total Users: {users.length}</h3>
8             <div className="grid grid-cols-3">
9                 {users.map((user) => [
0                     return (
1                         <div
2                             key={user._id}
3                             className="card w-96 bg-base-100
4                             shadow-xl border border-secondary"
5                         >
6                             <div className="card-body">
7                                 <h2 className="card-title">{user.
8                                     name}</h2>
9                                     <p>{user.email}</p>
10                                </div>
11                            </div>
12                        );
13                    ]);
14                );
15            </div>
16        );
17    );

```



UI:

Home    Users

Total Users: 2

Babu Mia babu@babu.com	naruto naruto@gmail.com
---------------------------	----------------------------

UI: Client Side:

Total Users: 4

**Babu Mia**  
babu@babu.com  
[Delete](#)

**naruto**  
naruto@gmail.com  
[Delete](#)

**bulbuli**  
bulbuli@bulbuli.com  
[Delete](#)

**Hello Kumar**  
hkumar@gmail.com  
[Delete](#)

Amra delete button create korbo and click korle amader `id` ta kintu pathaite hobe jei id dhore pore delete ta hobe.

```
const handleDeleteUser = (id) => {
  console.log(id);
};
```

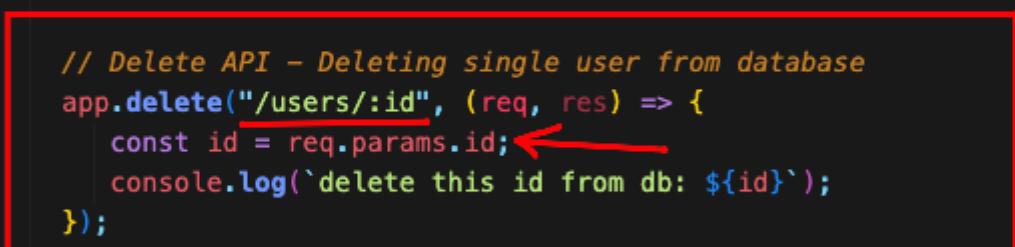
```
<button
  onClick={() => {
    handleDeleteUser(user._id);
  }}
  className="btn btn-primary"
>
  Delete
</button>
```

Now delete operation to backend:

```
// POST API
app.post("/users", async (req, res) => {
  const user = req.body;
  console.log("new user is here :", user);
  const result = await userCollection.insertOne(user);
  res.send(result);
});

// Delete API - Deleting single user from database
app.delete("/users/:id", (req, res) => {
  const id = req.params.id; ←
  console.log(`delete this id from db: ${id}`);
});

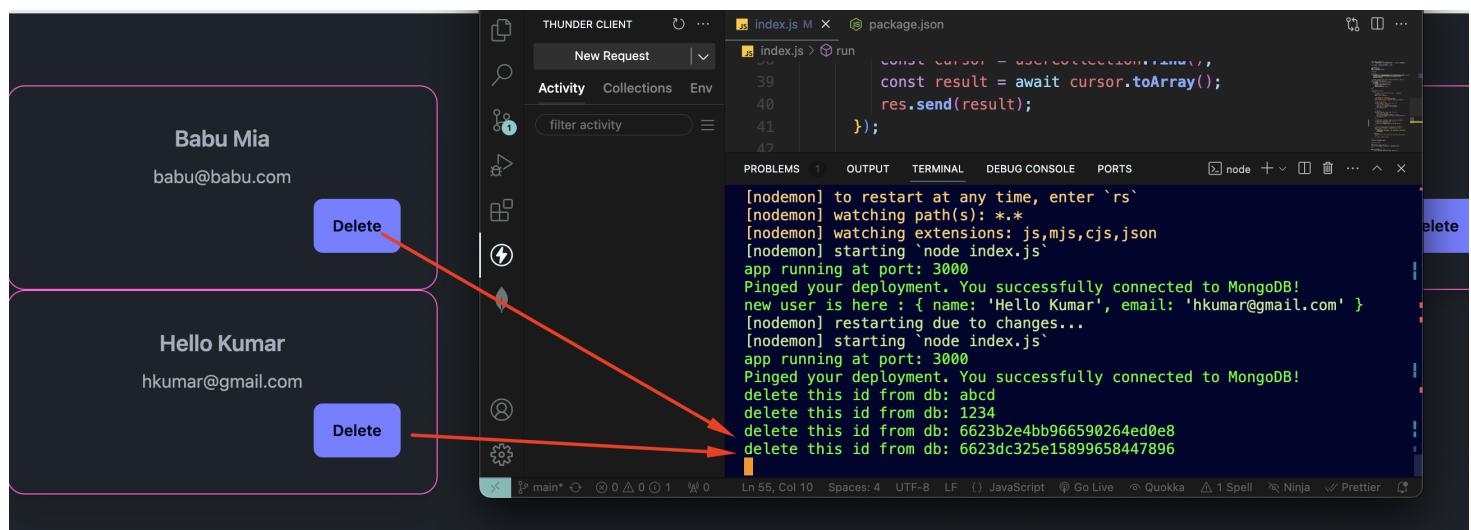
// Send a ping to confirm a successful connection
await client.db("admin").command({ ping: 1 });
console.log(
  "Pinged your deployment. You successfully connected to MongoDB!"
);
```



so going back to the handleDeleteUser button:

```
const handleDeleteUser = (id) => {
  console.log(id); // 6623dc325e15899658447896
  fetch(`http://localhost:3000/users/${id}`, {
    method: "DELETE",
  })
    .then((res) => res.json())
    .then((data) => console.log(data))
    .catch((err) => console.error(err));
};
```

server log



so amra kintu id ta pathaite partesi client thike server e. nice!

now mongo

```

1 // Delete a document
2
3 import { MongoClient } from "mongodb";
4
5 // Replace the uri string with your MongoDB deployment's connection string
6 const uri = "<connection string uri>";
7
8 const client = new MongoClient(uri);
9
10 async function run() {
11   try {
12     const database = client.db("sample_mflix");
13     const movies = database.collection("movies");
14
15     /* Delete the first document in the "movies" collection that matches
16     the specified query document */
17     const query = { title: "Annie Hall" };
18     const result = await movies.deleteOne(query);
19
20     /* Print a message that indicates whether the operation deleted a
21     document */
22     if (result.deletedCount === 1) {
23       console.log("Successfully deleted one document.");
24     } else {
25       console.log("No documents matched the query. Deleted 0 documents.");
26     }
27   } finally {
28     // Close the connection after the operation completes
29     await client.close();
30   }
31 }

```

code:

but amra iccha kore id ta bhul disi cause originally database e id number ta wrap kora `ObjectId(' ')` ai string e.

```

47         const result = await userCollection.insertOne(user);
48         res.send(result);
49     });
50
51     // Delete API - Deleting single user from database
52     app.delete("/users/:id", async (req, res) => {
53       // getting which id to delete
54       const id = req.params.id;
55       console.log(`delete this id from db: ${id}`);
56
57       // query (field name in database: value we seek)
58       const query = { _id: id };
59       // but amra iccha kore id ta bhul disi cause originally
60       // database e id number ta wrap kora ObjectId(' ') ai string e.
61       const result = await userCollection.deleteOne(query);
62       res.send(result);
63     });
64
65     // Send a ping to confirm a successful connection

```

and front end e akhon jodi dekhi je ki response pacchi jodi delete button click kori:

Total Users: 4

Babu Mia  
babu@babu.com  
Delete

naruto  
naruto@gmail.com  
Delete

Hello Kumar  
hkumar@gmail.com  
Delete

[vite] connecting...  
[vite] connected.  
✖ ► WebSocket connection to 'ws://127.0.0.1:5500//ws' failed:  
6623b2e4bb966590264ed0e8  
▶ {acknowledged: true, deletedCount: 0} ← deleteCount: 0

we can show an alert system:

```
const handleDeleteUser = (id) => {
  console.log(id);
  fetch(`http://localhost:3000/users/${id}`, {
    method: "DELETE",
  })
    .then((res) => res.json())
    .then((data) => {
      console.log(data);
      if (data.deletedCount > 0) {
        alert("Data removed from database");
      }
    })
    .catch((err) => console.error(err));
};
```

so akhon solve korbo kibhabe ObjectId r issue ta? By importing `ObjectId` and using the `new` keyword backend:

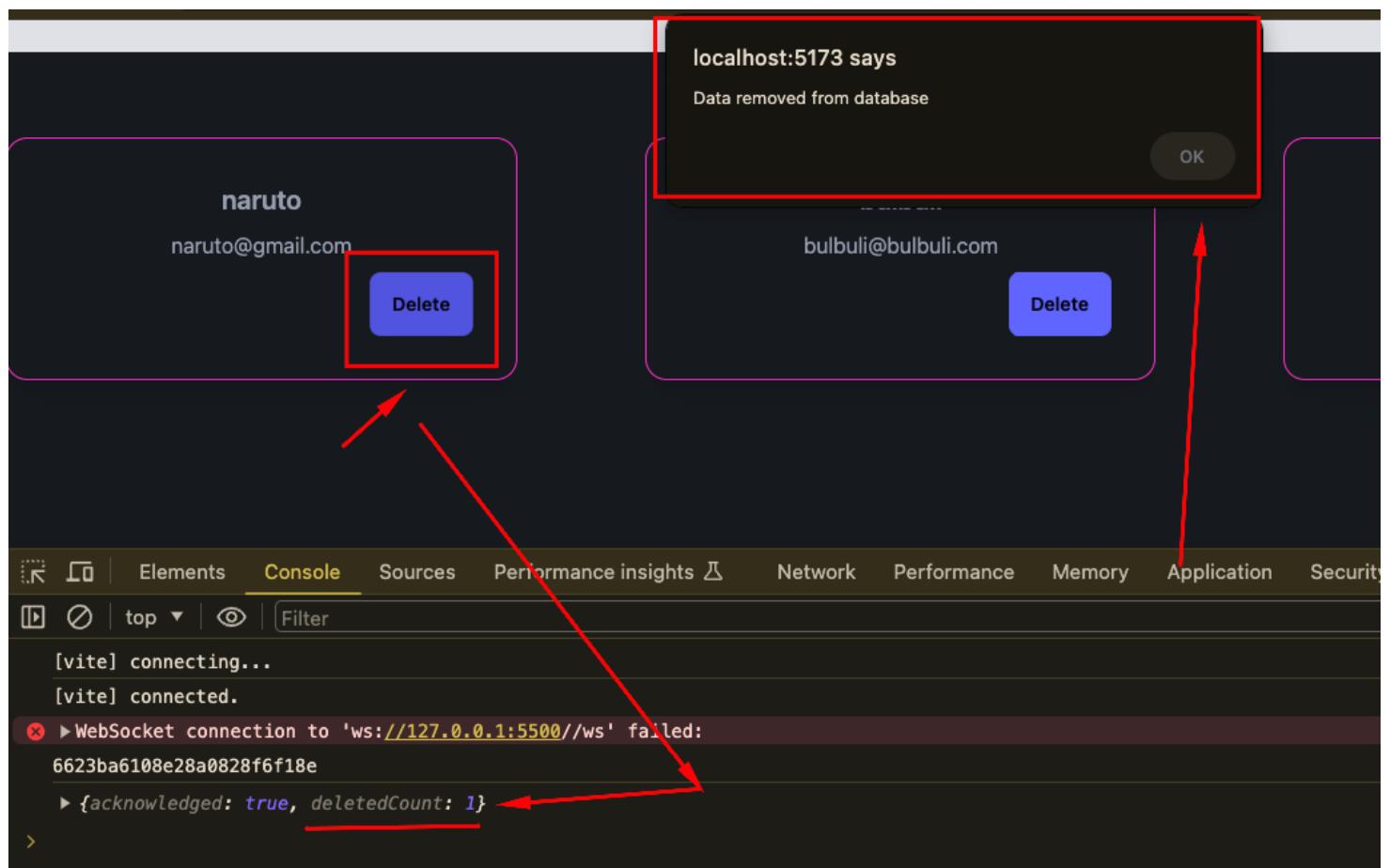
```
const app = express();
const { MongoClient, ServerApiVersion, ObjectId } = require("mongodb");
```

```
// Delete API - Deleting single user from database
app.delete("/users/:id", async (req, res) => {
  // getting which id to delete
  const id = req.params.id;
  console.log(`delete this id from db: ${id}`);

  // query (field name in database: value we seek)
  const query = { _id: new ObjectId(id) };

  const result = await userCollection.deleteOne(query);
  res.send(result);
});
```

now:



Great, but User is showing in UI till we refresh,

To Solve this: set a state

```
const Users = () => {
  const loadedUsers = useLoaderData();
  const [users, setUsers] = useState(loadedUsers); - (1)
```

```
  const handleDeleteUser = (id) => {
    console.log(id); '6623c550b33dc0b666408586'
    fetch(`http://localhost:3000/users/${id}`, {
      method: "DELETE",
    })
      .then((res) => res.json())
      .then((data) => {
        console.log(data); { acknowledged: true, deletedCount: 1 }
        if (data.deletedCount > 0) {
          alert("Data removed from database");
          const remaining = users.filter((user) => user._id !== id);
          setUsers(remaining); - (2) - (3)
        }
      })
      .catch((err) => console.error(err));
  };
}

return ()
```

Perfect.

Now interesting part -> CREATING A SINGLE DATA API in the backend

then loading that data in the client side and updating it.

v6.5 (current)

Quick Start

Quick Reference

What's New

Usage Examples

Find Operations

**Find a Document**

Find Multiple Documents

Insert Operations

Update &amp; Replace Operations

Delete Operations

Count Documents

Retrieve Distinct Values of a Field

Run a Command

Watch for Changes

Perform Bulk Operations

Perform a Transaction

Fundamentals

Aggregation Tutorials

API Documentation

FAQ

```

import { MongoClient } from "mongodb";

// Replace the uri string with your MongoDB deployment's connection string.
const uri = "<connection string uri>";

const client = new MongoClient(uri);

async function run() {
  try {

    // Get the database and collection on which to run the operation
    const database = client.db("sample_mflix");
    const movies = database.collection("movies");

    // Query for a movie that has the title 'The Room'
    const query = { title: "The Room" };

    const options = {
      // Sort matched documents in descending order by rating
      sort: { "imdb.rating": -1 },
      // Include only the 'title' and 'imdb' fields in the returned document
      projection: { _id: 0, title: 1, imdb: 1 },
    };

    // Execute query
    const movie = await movies.findOne(query, options);

    // Print the document returned by findOne()
    console.log(movie);
  } finally {
    await client.close();
  }
}

```

*must**optional**must**optional*

our code backend:

```

// GET API for a single User:
app.get("/users/:id", async (req, res) => {
  const id = req.params.id;
  const query = { _id: new ObjectId(id) };
  const user = await userCollection.findOne(query);

  res.send(user);
});

// POST API

```

so now we got an API which can show single data

localhost:3000/users/6623dc325e15899658447896

Pretty print □

```
{"_id": "6623dc325e15899658447896", "name": "Hello Kumar", "email": "hkumar@gmail.com"}
```

Now let us make this route and get dynamic id in the client side:

root.jsx:

```
        },
    },
    {
        path: "/users/:id",
        element: <Update></Update>,
        loader: ({ params }) => {
            return fetch(`http://localhost:3000/users/${params.id}`);
        },
    },
],
},
```

update.jsx:

```
const Update = () => {
    const user = useLoaderData();

    return (
        <div>
            <h3 className="text-2xl mt-4 text-red-400 text-center">
                Profile of Username: {user.name}
            </h3>
        </div>
    );
};

export default Update;
```

users.jsx:

```
<p>{user.email}</p>
<div className="card-actions justify-end">
    <Link to={`/users/${user._id}`}>
        <button className="btn btn-secondary">
            Update User
        </button>
    </Link>
    <button
        onClick={() => {
            handleDeleteUser(user.id);
        }}>
        Delete User
    </button>
</div>
```

UI:

Now let us show the user info in a form and using default values.

```
const Update = () => {
  const user = useLoaderData();

  const handleUpdateUser = (event) => {
    event.preventDefault();
    const form = event.target;
    const name = form.name.value;
    const email = form.email.value;
    console.log(name, email); // 'Hello Kumar' | 'hkumar@gmail.com'
  };

  return (
    <div>
      <h3 className="text-2xl mt-4 text-red-400 text-center">...
      </h3>
      <form
        onSubmit={handleUpdateUser}
        className="max-w-sm mx-auto space-y-4 mt-4"
      >
        {/* Name field */}
        <label className="input input-bordered flex items-center gap-2">
          <svg ...
          </svg>
          <input
            type="text"
            name="name"
            className="grow"
            defaultValue={user.name}
            placeholder="Username"
          />
        </label>
      </form>
    </div>
  );
}
```

## Profile of Username: Hello Kumar

Hello Kumar

hkumar@gmail.com

Update User

PUT- thakle update korbe nahole purata entry korbe

PATCH - update parts

so first e amra backend e PUT API create korbo:

```
// PUT API
app.put("/users/:id", async (req, res) => {
  const id = req.params.id;
  const updatedUser = req.body;
  console.log("user received in backend to update: ", updatedUser);
});
```

client side e akhon code korbo jate backend e user info ta pathano jae:

```
2
3  const Update = () => {
4    const user = useLoaderData();
5
6    const handleUpdateUser = (event) => {
7      event.preventDefault();
8      const form = event.target;
9      const name = form.name.value;
10     const email = form.email.value;
11     const updatedUser = { name, email };
12
13     fetch(`http://localhost:3000/users/${user._id}`, {
14       method: "PUT",
15       headers: { "content-type": "application/json" },
16       body: JSON.stringify(updatedUser),
17     })
18       .then((res) => res.json())
19       .then((data) => {
20         console.log(data);
21       });
22     };
23
24   return (
25     <div>
```

akhon jodi amra update click kori, request ta kintu backend e show kore:

## Profile of Username: hulu

hulu

hulu@gmail.com

Update User

filter activity

```

52
53 // POST API
54 app.post("/users", async (req, res) => {
55   const user = req.body;
56   console.log("new user is here :", user);
57   const result = await userCollection.insertOne(user);
58   res.send(result);
59 }
60
61 // PUT API
62 app.put("/users/:id", async (req, res) => {
63   const id = req.params.id;
64   const updatedUser = req.body;
65   console.log("user received in backend to update: ", updatedUser);
66 }
67
68 // Delete API - Deleting single user from database
69 app.delete("/users/:id", async (req, res) => {
70   // getting which id to delete

```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE PORTS

```

[nodemon] to restart at any time, enter `rs`  

[nodemon] watching path(s): *.*  

[nodemon] watching extensions: js,mjs,cjs,json  

[nodemon] starting `node index.js`  

app running at port: 3000  

Pinged your deployment. You successfully connected to MongoDB!  

[nodemon] restarting due to changes...  

[nodemon] starting `node index.js`  

app running at port: 3000  

Pinged your deployment. You successfully connected to MongoDB!  

[nodemon] restarting due to changes...  

[nodemon] starting `node index.js`  

app running at port: 3000  

Pinged your deployment. You successfully connected to MongoDB!  

user received in backend to update: { name: 'hulu', email: 'hulu@gmail.com' }

```

akhon backend e jehutu id ta paisi PUT korar jonne, kaaj akhon mongor

The screenshot shows the MongoDB Node.js Driver documentation page. On the left, there's a sidebar with navigation links like 'Quick Start', 'Quick Reference', 'What's New', 'Usage Examples' (which is expanded), 'Find Operations', 'Insert Operations', 'Update & Replace Operations' (which is also expanded), and 'Replace a Document'. The main content area displays a code snippet for updating a document in a MongoDB collection.

The code is as follows:

```

12 const client = Client.create("sampledb");
13 const movies = database.collection("movies");
14
15 // Create a filter for movies with the title "Random Harvest"
16 const filter = { title: "Random Harvest" };
17
18 /* Set the upsert option to insert a document if no documents match
19 the filter */
20 const options = { upsert: true };
21
22 // Specify the update to set a value for the plot field
23 const updateDoc = {
24   $set: {
25     plot: `A harvest of random numbers, such as: ${Math.random()}`,
26   },
27 };
28
29 // Update the first document that matches the filter
30 const result = await movies.updateOne(filter, updateDoc, options);
31
32 // Print the number of matching and modified documents
33 console.log(` ${result.matchedCount} document(s) matched the filter, updated ${result.modifiedCount}`);
34
35 } finally {
36   // Close the connection after the operation completes
37   await client.close();
38 }

```

The code is annotated with red boxes and circled numbers:

- Box 1:** Surrounds the line `const filter = { title: "Random Harvest" };`.
- Box 2:** Surrounds the line `const options = { upsert: true };`.
- Box 3:** Surrounds the block starting with `// Specify the update to set a value for the plot field`.
- Box 4:** Surrounds the line `const result = await movies.updateOne(filter, updateDoc, options);`.

our code:

```
9
0 // PUT API
1 app.put("/users/:id", async (req, res) => {
2   const id = req.params.id;
3   const user = req.body;
4   console.log("user received in backend to update: ", user);
5
6   const filter = { _id: new ObjectId(id) }; — ①
7   const options = { upsert: true }; — ②
8
9   const updatedUser = {
10     $set: {
11       name: user.name,
12       email: user.email,
13     },
14   };
15
16   const result = await userCollection.updateOne( — ④
17     filter,
18     updatedUser,
19     options
20   );
21
22   res.send(result); — ⑤
23 });
24
```

UI:

The screenshot shows a web application interface and its corresponding developer tools. At the top, there's a navigation bar with 'Home' and 'Users'. Below it is a profile card for 'huluhulu' with fields for 'Name' (huluhulu) and 'Email' (hulu@gmail.com). A red arrow points from the 'Name' field to the 'name' key in the code. Another red arrow points from the 'Email' field to the 'email' key in the code. At the bottom of the profile card is a pink button labeled 'Update User'. The developer tools at the bottom show the browser's network activity, with a red arrow pointing to a successful WebSocket connection message: 'WebSocket connection to 'ws://127.0.0.1:5500//ws' failed: > {acknowledged: true, modifiedCount: 1, upsertedId: null, upsertedCount: 0, matchedCount: 1}'.

we can slow alert if we want:

The screenshot shows a web application interface with a modal dialog and a browser's developer tools console.

**Modal Dialog:**

- Header: "localhost:5173 says"
- Content: "user is being updated"
- Buttons: "OK" (top right)

**Form Fields:**

- Email input: "mister@gmail.com"
- Update button: "Update User" (pink background)

**Developer Tools Console:**

- Logs:
  - [vite] connecting...
  - [vite] connected.
  - ✖ WebSocket connection to 'ws://127.0.0.1:5500//ws' failed:
    - ▶ {acknowledged: true, modifiedCount: 1, upsertedId: null, upsertedCount: 0, matchedCount: 1}
  - [vite] hot updated: /src/components/Update/Update.jsx
  - [vite] hot updated: /src/index.css
  - ▶ {acknowledged: true, modifiedCount: 1, upsertedId: null, upsertedCount: 0, matchedCount: 1}
- Filter: "Filter" (dropdown menu)