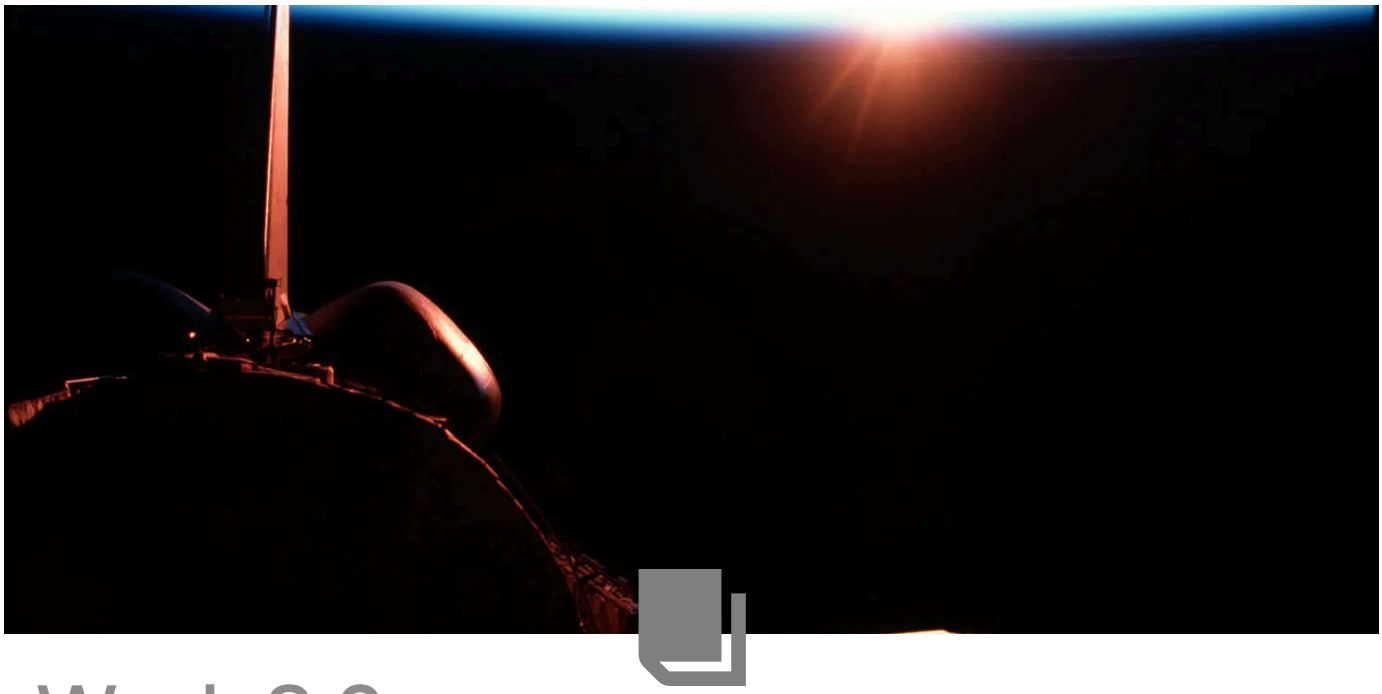
Show Contents

Filter



Week 8.2

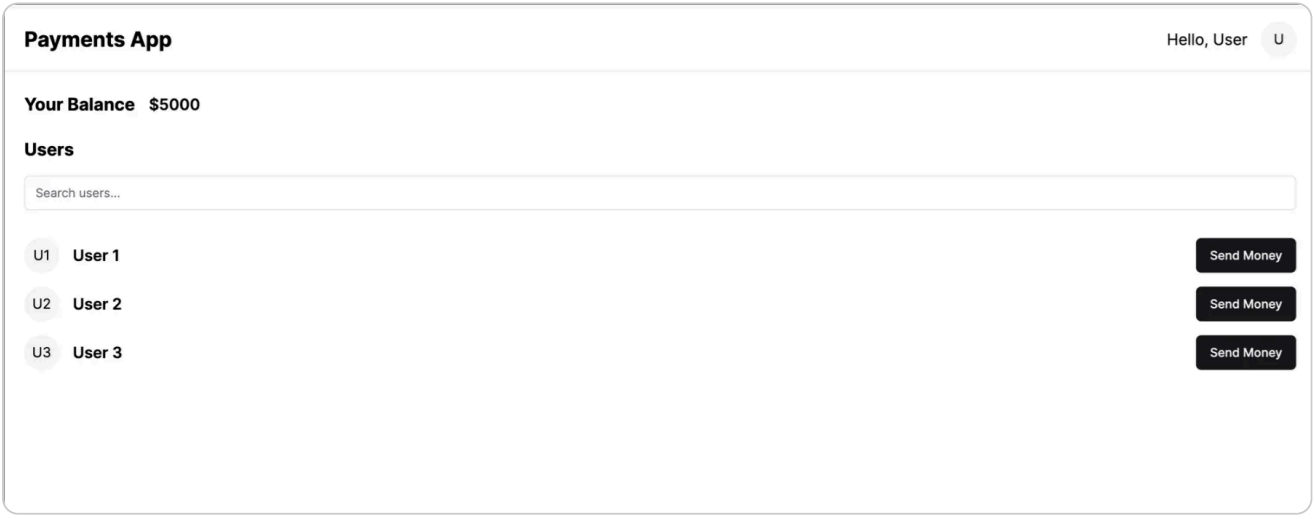
Recap Everything, Build PayTM Backend

In this lecture, Harkirat guides us through an `end-to-end tutorial` on building a comprehensive `full-stack application` resembling `Paytm` While there are no specific notes provided for this section, a mini guide is outlined below to assist you in navigating through each step of the tutorial. Therefore, it is strongly advised to actively follow along during the lecture for a hands-on learning experience.

> It's important to note that this session primarily focuses on the backend

Step 1 - What are we building, Clone the starter repo

We're building a PayTM like application that let's users send money to each other given an initial dummy balance



Things to do

Clone the 8.2 repository from https://github.com/100xdevs-cohort-2/paytm

```
git clone https://github.com/100xdevs-cohort-2/paytm
```

2. There is a Dockerfile in the codebase, you can run mongo locally using it.

Explore the repository

The repo is a basic **express + react + tailwind boilerplate**

Backend

1. Express - HTTP Server
2. mongoose - ODM to connect to MongoDB
3. zod - Input validation

```
// index.js
const express = require("express");
const app = express();
```

Frontend

1. React - Frontend framework
2. Tailwind - Styling framework

```
// App.jsx
function App() {

  return (
    <div>
        Hello world
    </div>
  )
}
```

Step 2 - User Mongoose schemas

We need to support 3 routes for user authentication

1. Allow user to sign up.
2. Allow user to sign in.
3. Allow user to update their information (firstName, lastName, password).

To start off, create the mongo schema for the users table

1. Create a new file (db.js) in the root folder
2. Import mongoose and connect to a database of your choice
3. Create the mongoose schema for the users table
4. Export the mongoose model from the file (call it User)

▶

Solution

Step 3 - Create routing file structure

In the index.js file, route all the requests to `/api/v1` to a apiRouter defined in `backend/routes/index.js`

Step 1

Create a new file `backend/routes/index.js` that exports a new express router.

(How to create a router - https://www.geeksforgeeks.org/express-js-express-router-function/)

Solution

Step 2

Import the router in index.js and route all requests from `/api/v1` to it

▶

Solution

Step 4 - Route user requests

1. Create a new user router

Define a new router in `backend/routes/user.js` and import it in the index router.

Route all requests that go to `/api/v1/user` to the user router.

▶

Solution

2. Create a new user router

Import the userRouter in `backend/routes/index.js` so all requests to `/api/v1/user` get routed to the userRouter.

▶

Solution

```
const userRouter = require("./user");

const router = express.Router();

router.use("/user", userRouter)

module.exports = router;
```

Step 5 - Add cors, body parser and jsonwebtoken

1. Add cors

Since our frontend and backend will be hosted on separate routes, add the `cors` middleware to `backend/index.js`

▶

Hint

▶

Solution

2. Add body-parser

Since we have to support the JSON body in post requests, add the express body parser middleware to `backend/index.js`
You can use the `body-parser` npm library, or use express.json

.....

▶

Solution

3. Add jsonwebtoken

We will be adding authentication soon to our application, so install jsonwebtoken library. It'll be useful in the next slide

```
npm install jsonwebtoken
```

4. Export JWT_SECRET

Export a JWT_SECRET from a new file `backend/config.js`

▶

Solution

5. Listen on port `3000`

Make the express app listen on PORT 3000 of your machine

▶

Solution

1. Signup

This route needs to get user information, do input validation using zod and store the information in the database provided

1. Inputs are correct (validated via zod)
2. Database doesn't already contain another user

If all goes well, we need to return the user a jwt which has their user id encoded as follows -

```
{
    userId: "userId of newly added user"
}
```

💡

Note - We are not hashing passwords before putting them in the database. This is standard practise that should be done, you can find more details here - https://mojoauth.com/blog/hashing-passwords-in-nodejs/

Method: POST
Route: /api/v1/user/signup
Body:

```
{
    username: "name@gmail.com",
    firstName: "name",
    lastName: "name",
    password: "123456"
}
```

Response:

Status code - 200

Status code - 411

```
{
    message: "Email already taken / Incorrect inputs"
}
```

▶

Solution

2. Route to sign in

Let's an existing user sign in to get back a token.

Method: POST
Route: /api/v1/user/signin
Body:

```
{
    username: "name@gmail.com",
    password: "123456"
}
```

Response:

Status code - 200

```
    token: "jwt"
}
```

Status code - 411

```
{
    message: "Error while logging in"
}
```

▶

Solution

By the end, `routes/user.js` should look like follows

▶

Solution

Step 7 - Middleware

Now that we have a user account, we need to `gate` routes which authenticated users can hit.

For this, we need to introduce an auth middleware

Create a `middleware.js` file that exports an `authMiddleware` function

```
Header -
Authorization: Bearer <actual token>
```

▶

Solution

Step 8 - User routes

1. Route to update user information

User should be allowed to `optionally` send either or all of

1. password
2. firstName
3. lastName

Whatever they send, we need to update it in the database for the user.
Use the `middleware` we defined in the last section to authenticate the user

Method: PUT
Route: /api/v1/user
Body:

```
{
    password: "new_password",
    firstName: "updated_first_name",
    lastName: "updated_first_name",
}
```

```
}
```

Status code - 411 (Password is too small...)

```
{
    message: "Error while updating information"
}
```

▶

Solution

2. Route to get users from the backend, filterable via firstName/lastName

This is needed so users can search for their friends and send them money

Method: GET
Route: /api/v1/user/bulk
Query Parameter: `?filter=harkirat`

Response:

Status code - 200

```
    users: [{
        firstName: "",
        lastName: "",
        _id: "id of the user"
    }]
}
```

▶

Hints

▶

Solution

Step 9 - Create Bank related Schema

Update the `db.js` file to add one new schemas and export the respective models

Accounts table

The `Accounts` table will store the INR balances of a user.

The schema should look something like this -

```
{
    userId: ObjectId (or string),
    balance: float/number
}
```

```
There is a certain precision that you need to support (which fo
2/4 decimal places) and this allows you to get rid of precision
errors by storing integers in your DB
```

You should reference the users table in the schema (Hint - https://medium.com/@mendes.develop/joining-tables-in-mongodb-with-mongoose-489d72c84b60)

▶

Solution

▶

By the end of it, `db.js` should look lie this

Step 10 - Transactions in databases

A lot of times, you want multiple databases transactions to be `atomic`

Either all of them should update, or none should

This is super important in the case of a `bank`

Can you guess what's wrong with the following code -

```
const mongoose = require('mongoose');
const Account = require('./path-to-your-account-model');
```

```
    // Increment the balance of the toAccount
    await Account.findByIdAndUpdate(toAccountId, { $inc: { bala
}

// Example usage
transferFunds('fromAccountID', 'toAccountID', 100);
```

▶

Solution

Step 11 - Initialize balances on signup

Update the `signup` endpoint to give the user a random balance between 1 and 10000.

This is so we don't have to integrate with banks and give them random balances to start with.

▶

Solution

Step 12 - Create a new router for accounts

1. Create a new router

All user balances should go to a different express router (that handles all requests on `/api/v1/account`).

Create a new router in `routes/account.js` and add export it

▶

2. Route requests to it

Send all requests from `/api/v1/account/*` in `routes/index.js` to the router created in step 1.

▶

Solution

Step 13 - Balance and transfer Endpoints

Here, you'll be writing a bunch of APIs for the core user balances. There are 2 endpoints that we need to implement

1. An endpoint for user to get their balance.

Method: GET
Route: /api/v1/account/balance

Response:

Status code - 200

```
{
    balance: 100
}
```

▶

Solution

2. An endpoint for user to transfer money to another

Body

```
{
    to: string,
    amount: number
}
```

Response:

Status code - 200

```
{
    message: "Transfer successful"
}
```

Status code - 400

```
{
    message: "Insufficient balance"
}
```

Status code - 400

```
{
    message: "Invalid account"
}
```

▶

Bad Solution (doesn't use transactions)

▶

Problems you might run into

Final Solution

▶

Finally, the account.js file should look like this

Experiment to ensure transactions are working as expected

Try running this code locally. It calls transfer twice on the same account ~almost concurrently

▶

Code

▶

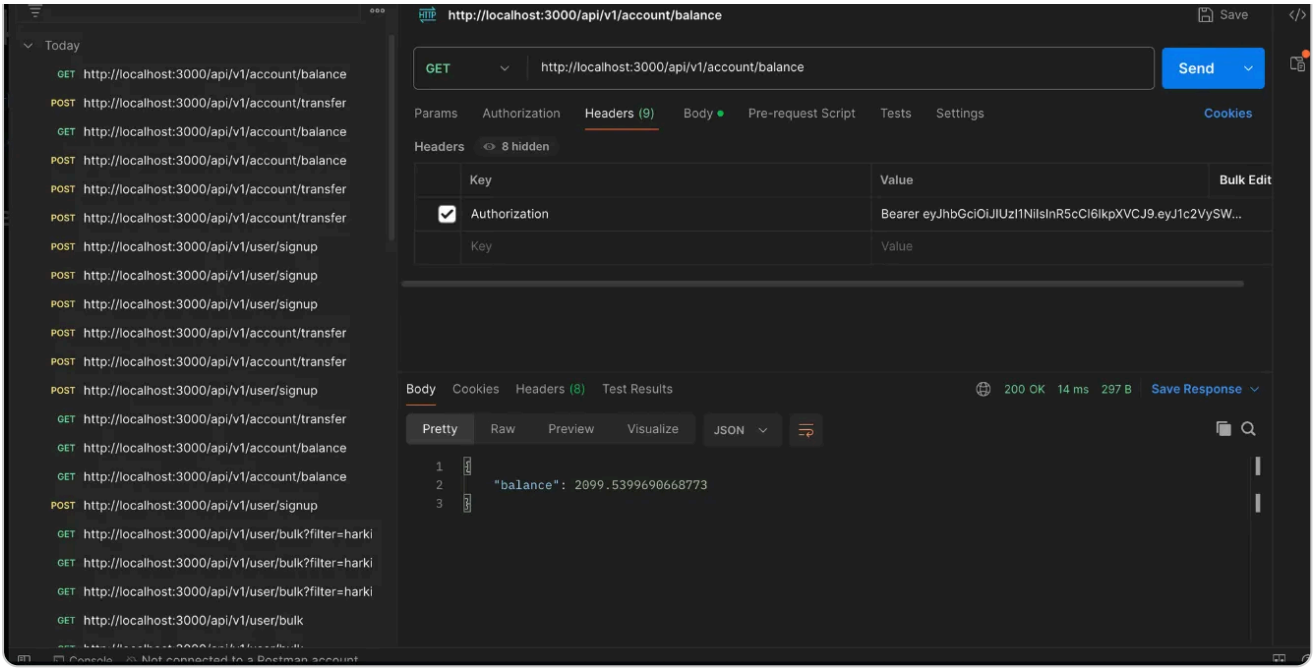
Error

Step 14 - Checkpoint your solution

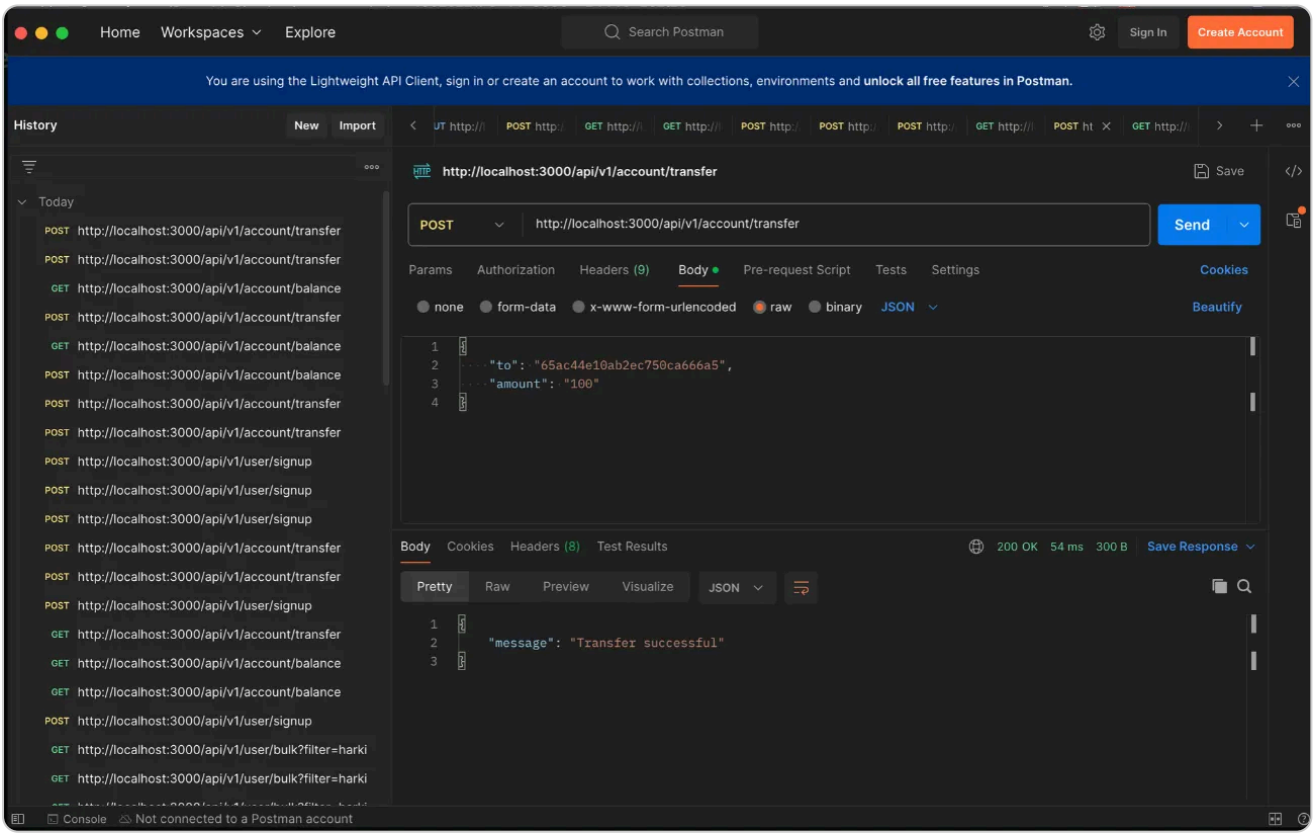
A completely working backend can be found here - https://github.com/100xdevs-cohort-2/paytm/tree/backend-solution

Try to send a few calls via postman to ensure you are able to sign up/sign in/get balance

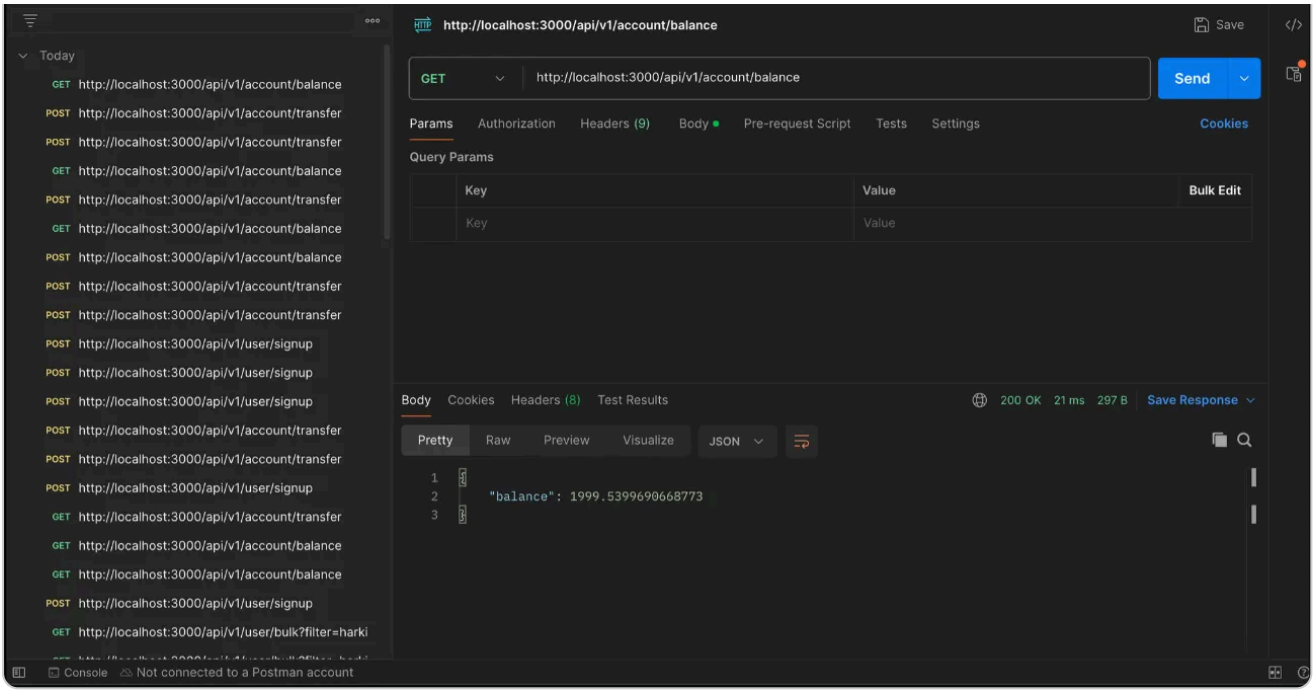
Get balance



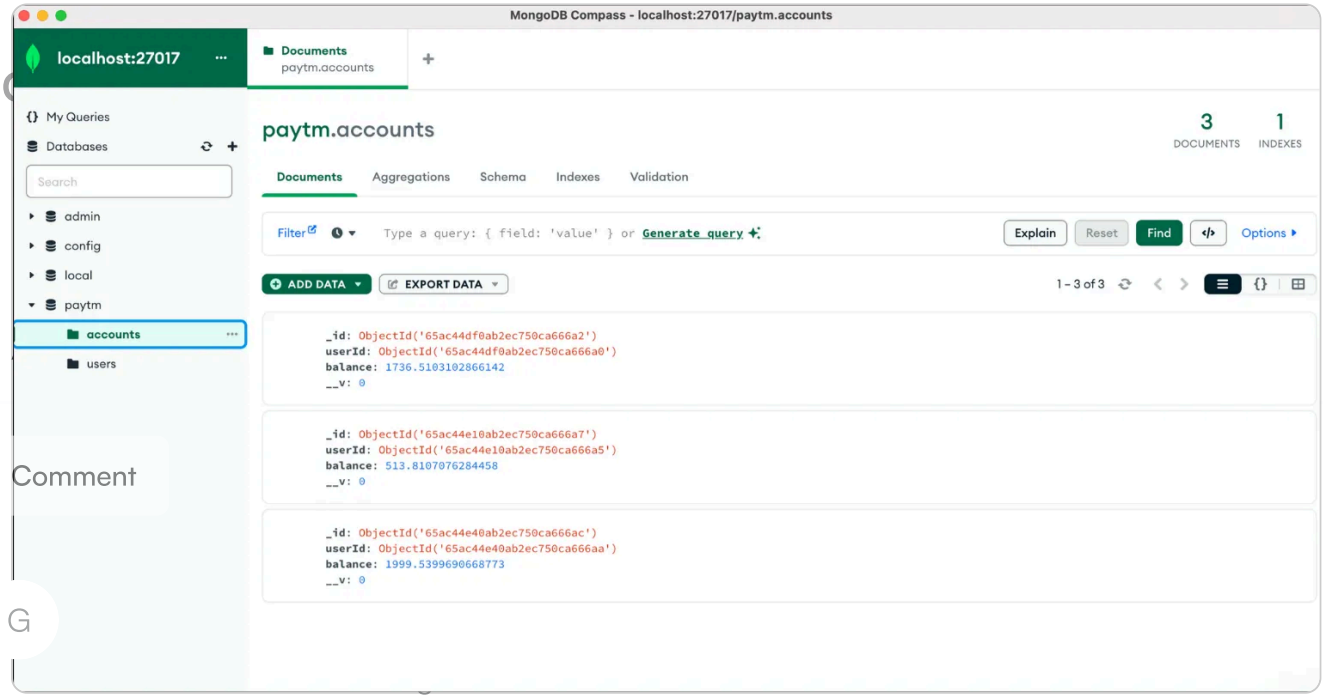
Make transfer



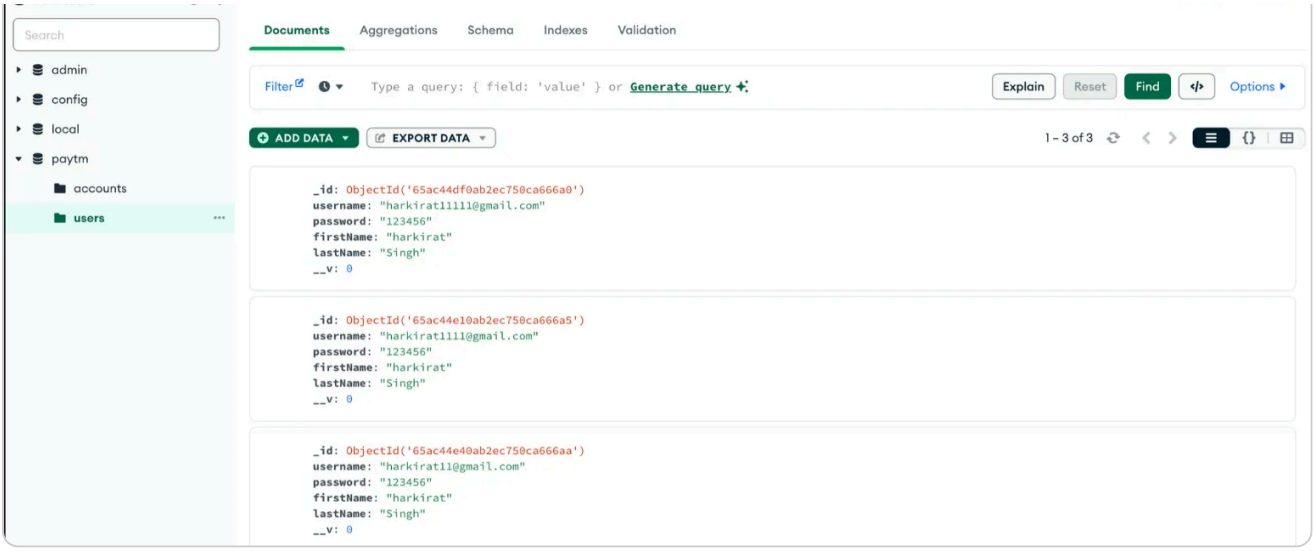
Get balance again (notice it went down)



Mongo should look something like this



3 0 1 Reply


Search
admin
config
local
paytm
accounts
users
Documents
Aggregations
Schema
Indexes
Validation
Filter
Type a query: { field: 'value' } or Generate query
Explain
Reset
Find
Options
ADD DATA
EXPORT DATA
1 - 3 of 3
_id: ObjectId('65ac44df0ab2ec750ca666a0')
username: "harkirat11111@gmail.com"
password: "123456"
firstName: "harkirat"
lastName: "Singh"
__v: 0
_id: ObjectId('65ac44e10ab2ec750ca666a5')
username: "harkirat1111@gmail.com"
password: "123456"
firstName: "harkirat"
lastName: "Singh"
__v: 0
_id: ObjectId('65ac44e40ab2ec750ca666aa')
username: "harkirat11@gmail.com"
password: "123456"
firstName: "harkirat"
lastName: "Singh"
__v: 0