

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

The screenshot shows a user interface for a 'Payments App'. At the top, it says 'Hello, User' with a profile icon. Below that, it displays 'Your Balance \$5000'. Under the heading 'Users', there is a search bar labeled 'Search users...'. A list of three users is shown: 'User 1', 'User 2', and 'User 3', each preceded by a small circular icon with a letter (U1, U2, U3). To the right of each user name is a black rectangular button with the text 'Send Money' in white.

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 Copy
```



Please keep a MongoDB url handy before you proceed. This will be your primary database for this assignment
1. Create a free one here - <https://www.mongodb.com/>
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

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

Copy

index.js

Frontend

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

```
function App() {  
  
  return (  
    <div>  
      Hello world  
    </div>  
  )  
}  
  
export default App
```

Copy

App.jsx

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

Simple solution

```
// backend/db.js
const mongoose = require('mongoose');

// Create a Schema for Users
const userSchema = new mongoose.Schema({
    username: String,
    password: String,
    firstName: String,
    lastName: String
});

// Create a model from the schema
const User = mongoose.model('User', userSchema);

module.exports = {
    User
};
```

Copy

Elegant Solution

```
// backend/db.js
const mongoose = require('mongoose');

// Create a Schema for Users
const userSchema = new mongoose.Schema({
    username: {
        type: String,
        required: true,
        unique: true,
        trim: true,
        lowercase: true,
        minLength: 3,
        maxLength: 30
    },
    password: {
        type: String,
        required: true,
        minLength: 6
    },
    firstName: {
        type: String,
        required: true,
        trim: true,
        maxLength: 50
    },
    lastName: {
        type: String,
        required: true,
        trim: true,
        maxLength: 50
    }
});
```

Copy

```
});

// Create a model from the schema
const User = mongoose.model('User', userSchema);

module.exports = {
  User
};
```

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

```
// backend/api/index.js
const express = require('express');

const router = express.Router();

module.exports = router;
```

Copy

Step 2

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

▼ Solution

```
// backend/index.js
const express = require("express");
const rootRouter = require("./routes/index");

const app = express();

app.use("/api/v1", rootRouter);
```

Copy

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

```
// backend/routes/user.js
const express = require('express');

const router = express.Router();

module.exports = router;
```

Copy

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

```
// backend/routes/index.js
const express = require('express');
```

Copy

```
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

Look at <https://www.npmjs.com/package/cors>

▼ Solution

```
// backend/index.js
const express = require('express');
const cors = require("cors");

app.use(cors());

const app = express();

module.exports = router;
```

Copy

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`

▼ Hint

<https://medium.com/@mmajdanski/express-body-parser-and-why-may-not-need-it-335803cd048c>

▼ Solution

```
// backend/index.js
const express = require('express');
const cors = require("cors");
const rootRouter = require("./routes/index");

const app = express();

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

app.use("/api/v1", rootRouter);
```

Copy

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
```

Copy

4. Export JWT_SECRET

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

▼ Solution

```
//backend/config.js
module.exports = {
  JWT_SECRET: "your-jwt-secret"
}
```

Copy

5. Listen on port **3000**

Make the express app listen on PORT 3000 of your machine

▼ Solution

```
// backend/index.js  
... Existing code  
  
app.listen(3000);
```

Copy

Step 6 - Add backend auth routes

In the user router (`backend/routes/user`), add 3 new routes.

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"  
}
```

Copy



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",
```

Copy

```
    firstName: "name",
    lastName: "name",
    password: "123456"
}
```

Response:

Status code - 200

```
{
  message: "User created successfully",
  token: "jwt"
}
```

[Copy](#)

Status code - 411

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

[Copy](#)

▼ Solution

```
const zod = require("zod");
const { User } = require("../db");
const jwt = require("jsonwebtoken");
const { JWT_SECRET } = require("../config");

const signupBody = zod.object({
  username: zod.string().email(),
  firstName: zod.string(),
  lastName: zod.string(),
  password: zod.string()
})

router.post("/signup", async (req, res) => {
  const { success } = signupBody.safeParse(req.body)
  if (!success) {
    return res.status(411).json({
      message: "Email already taken / Incorrect inputs"
    })
  }

  const existingUser = await User.findOne({
    username: req.body.username
  })

  if (existingUser) {
    return res.status(411).json({
      message: "Email already taken/Incorrect inputs"
    })
  }
}
```

[Copy](#)

```
const user = await User.create({
  username: req.body.username,
  password: req.body.password,
  firstName: req.body.firstName,
  lastName: req.body.lastName,
})
const userId = user._id;

const token = jwt.sign({
  userId
}, JWT_SECRET);

res.json({
  message: "User created successfully",
  token: token
})
})
```

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"
}
```

Copy

Response:

Status code - 200

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

Copy

Status code - 411

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

Copy

▼ Solution

Copy

```
const signinBody = zod.object({
    username: zod.string().email(),
    password: zod.string()
})

router.post("/signin", async (req, res) => {
    const { success } = signinBody.safeParse(req.body)
    if (!success) {
        return res.status(411).json({
            message: "Incorrect inputs"
        })
    }

    const user = await User.findOne({
        username: req.body.username,
        password: req.body.password
    });

    if (user) {
        const token = jwt.sign({
            userId: user._id
        }, JWT_SECRET);

        res.json({
            token: token
        })
        return;
    }

    res.status(411).json({
        message: "Error while logging in"
    })
})
```

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

▼ Solution

Copy

```
// backend/routes/user.js
const express = require('express');

const router = express.Router();
const zod = require("zod");
const { User } = require("../db");
const jwt = require("jsonwebtoken");
const { JWT_SECRET } = require("../config");

const signupBody = zod.object({
```

```
username: zod.string().email(),
firstName: zod.string(),
lastName: zod.string(),
password: zod.string()
})

router.post("/signup", async (req, res) => {
  const { success } = signupBody.safeParse(req.body)
  if (!success) {
    return res.status(411).json({
      message: "Email already taken / Incorrect inputs"
    })
  }

  const existingUser = await User.findOne({
    username: req.body.username
  })

  if (existingUser) {
    return res.status(411).json({
      message: "Email already taken/Incorrect inputs"
    })
  }

  const user = await User.create({
    username: req.body.username,
    password: req.body.password,
    firstName: req.body.firstName,
    lastName: req.body.lastName,
  })
  const userId = user._id;

  const token = jwt.sign({
    userId
  }, JWT_SECRET);

  res.json({
    message: "User created successfully",
    token: token
  })
}

const signinBody = zod.object({
  username: zod.string().email(),
  password: zod.string()
})

router.post("/signin", async (req, res) => {
  const { success } = signinBody.safeParse(req.body)
  if (!success) {
    return res.status(411).json({
      message: "Email already taken / Incorrect inputs"
    })
  }

  const user = await User.findOne({
```

```
        username: req.body.username,
        password: req.body.password
    });

    if (user) {
        const token = jwt.sign({
            userId: user._id
        }, JWT_SECRET);

        res.json({
            token: token
        })
        return;
    }

    res.status(411).json({
        message: "Error while logging in"
    })
}

module.exports = router;
```

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

1. Checks the headers for an Authorization header (Bearer <token>)
2. Verifies that the token is valid
3. Puts the `userId` in the request object if the token checks out.
4. If not, return a 403 status back to the user

Header -

Copy

`Authorization: Bearer <actual token>`

▼ Solution

```
const { JWT_SECRET } = require("./config");
const jwt = require("jsonwebtoken");

const authMiddleware = (req, res, next) => {
    const authHeader = req.headers.authorization;

    if (!authHeader || !authHeader.startsWith('Bearer ')) {
        return res.status(403).json({});
    }

    const token = authHeader.split(' ')[1];

    try {
        const decoded = jwt.verify(token, JWT_SECRET);

        req.userId = decoded.userId;

        next();
    } catch (err) {
        return res.status(403).json({});
    }
};

module.exports = {
    authMiddleware
}
```

Copy

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_last_name",  
}
```

Copy

Response:

Status code - 200

```
{  
  message: "Updated successfully"  
}
```

Copy

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

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

Copy

▼ Solution

```
const { authMiddleware } = require("../middleware");
```

Copy

```

// other auth routes

const updateBody = zod.object({
  password: zod.string().optional(),
  firstName: zod.string().optional(),
  lastName: zod.string().optional(),
})

router.put("/", authMiddleware, async (req, res) => {
  const { success } = updateBody.safeParse(req.body)
  if (!success) {
    res.status(411).json({
      message: "Error while updating information"
    })
  }

  await User.updateOne({ _id: req.userId }, req.body);

  res.json({
    message: "Updated successfully"
  })
})

```

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"
    }
  ]
}
```

[Copy](#)

▼ Hints

<https://stackoverflow.com/questions/7382207/mongooses-find-method-with-or-condition-does-not-work-properly>

<https://stackoverflow.com/questions/3305561/how-to-query-mongodb-with-like>

▼ Solution

```
router.get("/bulk", async (req, res) => {
  const filter = req.query.filter || "";

  const users = await User.find({
    $or: [
      {
        firstName: {
          "$regex": filter
        }
      },
      {
        lastName: {
          "$regex": filter
        }
      }
    ]
  })

  res.json({
    user: users.map(user => ({
      username: user.username,
      firstName: user.firstName,
      lastName: user.lastName,
      _id: user._id
    }))
  })
})
```

Copy

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  
}
```

Copy

In the real world, you shouldn't store `floats` for balances in the database. You usually store an integer which represents the INR value with decimal places (for eg, if someone has 33.33 rs in their account, you store 3333 in the database).

There is a certain precision that you need to support (which for India is 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

```
const accountSchema = new mongoose.Schema({  
  userId: {  
    type: mongoose.Schema.Types.ObjectId, // Reference to User  
    ref: 'User',  
    required: true  
  },  
  balance: {  
    type: Number,  
    required: true  
  }  
});  
  
const Account = mongoose.model('Account', accountSchema);  
  
module.exports = {  
  Account  
}
```

Copy

▼ By the end of it, db.js should look like this

```
// backend/db.js  
const mongoose = require('mongoose');
```

Copy

```
mongoose.connect("mongodb://localhost:27017/paytm")

// Create a Schema for Users
const userSchema = new mongoose.Schema({
    username: {
        type: String,
        required: true,
        unique: true,
        trim: true,
        lowercase: true,
        minLength: 3,
        maxLength: 30
    },
    password: {
        type: String,
        required: true,
        minLength: 6
    },
    firstName: {
        type: String,
        required: true,
        trim: true,
        maxLength: 50
    },
    lastName: {
        type: String,
        required: true,
        trim: true,
        maxLength: 50
    }
});

const accountSchema = new mongoose.Schema({
    userId: {
        type: mongoose.Schema.Types.ObjectId, // Reference to User
        ref: 'User',
        required: true
    },
    balance: {
        type: Number,
        required: true
    }
});

const Account = mongoose.model('Account', accountSchema);
const User = mongoose.model('User', userSchema);

module.exports = {
    User,
    Account,
};
```

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');

const transferFunds = async (fromAccountId, toAccountId, amount) => {
    // Decrement the balance of the fromAccount
    await Account.findByIdAndUpdate(fromAccountId, { $inc: { balance: -amount } })

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

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

Copy

▼ Answer

1. What if the database crashes right after the first request (only the balance is decreased for one user, and not for the second user)
2. What if the Node.js crashes after the first update?

It would lead to a **database inconsistency**. Amount would get debited from the first user, and not credited into the other users account.

If a failure ever happens, the first txn should rollback.

This is what is called a **transaction** in a database. We need to implement a **transaction** on the next set of endpoints that allow users to transfer INR

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

```
router.post("/signup", async (req, res) => {
  const { success } = signupBody.safeParse(req.body)
  if (!success) {
    return res.status(411).json({
      message: "Email already taken / Incorrect inputs"
    })
  }

  const existingUser = await User.findOne({
    username: req.body.username
  })

  if (existingUser) {
    return res.status(411).json({
      message: "Email already taken/Incorrect inputs"
    })
  }

  const user = await User.create({
    username: req.body.username,
    password: req.body.password,
    firstName: req.body.firstName,
    lastName: req.body.lastName,
  })
  const userId = user._id;

  // --- Create new account ----

  await Account.create({
    userId,
    balance: 1 + Math.random() * 10000
  })

  // ---
```

```
const token = jwt.sign({
  userId
}, JWT_SECRET);

res.json({
  message: "User created successfully",
  token: token
})
})
```

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

▼ Solution

```
// backend/routes/account.js
const express = require('express');

const router = express.Router();

module.exports = router;
```

Copy

2. Route requests to it

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

▼ Solution

```
// backend/user/index.js
const express = require('express');
const userRouter = require("./user");
const accountRouter = require("./account");

const router = express.Router();

router.use("/user", userRouter);
router.use("/account", accountRouter);

module.exports = router;
```

Copy

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
}
```

Copy

▼ Solution

```
router.get("/balance", authMiddleware, async (req, res) => {
  const account = await Account.findOne({
    userId: req.userId
  });

  res.json({
    balance: account.balance
});
```

Copy

```
});  
});
```

2. An endpoint for user to transfer money to another account

Method: POST

Route: /api/v1/account/transfer

Body

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

Copy

Response:

Status code - 200

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

Copy

Status code - 400

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

Copy

Status code - 400

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

Copy

▼ Bad Solution (doesn't use transactions)

```
router.post("/transfer", authMiddleware, async (req, res) => {      Copy  
  const { amount, to } = req.body;  
  
  const account = await Account.findOne({
```

```

        userId: req.userId
    });

    if (account.balance < amount) {
        return res.status(400).json({
            message: "Insufficient balance"
        })
    }

    const toAccount = await Account.findOne({
        userId: to
    });

    if (!toAccount) {
        return res.status(400).json({
            message: "Invalid account"
        })
    }

    await Account.updateOne({
        userId: req.userId
    }, {
        $inc: {
            balance: -amount
        }
    })

    await Account.updateOne({
        userId: to
    }, {
        $inc: {
            balance: amount
        }
    })

    res.json({
        message: "Transfer successful"
    })
}
);

```

▼ Good solution (uses txns in db)

```

router.post("/transfer", authMiddleware, async (req, res) => {      Copy
    const session = await mongoose.startSession();

    session.startTransaction();
    const { amount, to } = req.body;

    // Fetch the accounts within the transaction
    const account = await Account.findOne({ userId: req.userId });

    if (!account || account.balance < amount) {
        await session.abortTransaction();
        return res.status(400).json({

```

```

        message: "Insufficient balance"
    });
}

const toAccount = await Account.findOne({ userId: to }).session();

if (!toAccount) {
    await session.abortTransaction();
    return res.status(400).json({
        message: "Invalid account"
    });
}

// Perform the transfer
await Account.updateOne({ userId: req.userId }, { $inc: { balance: -amount } });
await Account.updateOne({ userId: to }, { $inc: { balance: amount } });

// Commit the transaction
await session.commitTransaction();
res.json({
    message: "Transfer successful"
});
}
);

```

▼ Problems you might run into If you run into the problem mentioned above, feel free to proceed with the bad solution

<https://stackoverflow.com/questions/51461952/mongodb-v4-0-transaction-mongoerror-transaction-numbers-are-only-allowed-on-a>

Final Solution

▼ Finally, the account.js file should look like this

```
// backend/routes/account.js
const express = require('express');
const { authMiddleware } = require('../middleware');
const { Account } = require('../db');

const router = express.Router();

router.get("/balance", authMiddleware, async (req, res) => {
    const account = await Account.findOne({
        userId: req.userId
    });

    res.json({
        balance: account.balance
    })
});
```

Copy

```

router.post("/transfer", authMiddleware, async (req, res) => {
  const session = await mongoose.startSession();

  session.startTransaction();
  const { amount, to } = req.body;

  // Fetch the accounts within the transaction
  const account = await Account.findOne({ userId: req.userId });

  if (!account || account.balance < amount) {
    await session.abortTransaction();
    return res.status(400).json({
      message: "Insufficient balance"
    });
  }

  const toAccount = await Account.findOne({ userId: to }).session(session);

  if (!toAccount) {
    await session.abortTransaction();
    return res.status(400).json({
      message: "Invalid account"
    });
  }

  // Perform the transfer
  await Account.updateOne({ userId: req.userId }, { $inc: { balance: -amount } });
  await Account.updateOne({ userId: to }, { $inc: { balance: amount } });

  // Commit the transaction
  await session.commitTransaction();

  res.json({
    message: "Transfer successful"
  });
});

module.exports = router;

```

Experiment to ensure transactions are working as expected -

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

▼ Code

```
// backend/routes/account.js
const express = require('express');
const { authMiddleware } = require('../middleware');
```

[Copy](#)

```
const { Account } = require('../db');
const { default: mongoose } = require('mongoose');

const router = express.Router();

router.get("/balance", authMiddleware, async (req, res) => {
  const account = await Account.findOne({
    userId: req.userId
  });

  res.json({
    balance: account.balance
  })
});

async function transfer(req) {
  const session = await mongoose.startSession();

  session.startTransaction();
  const { amount, to } = req.body;

  // Fetch the accounts within the transaction
  const account = await Account.findOne({ userId: req.userId });

  if (!account || account.balance < amount) {
    await session.abortTransaction();
    console.log("Insufficient balance")
    return;
  }

  const toAccount = await Account.findOne({ userId: to }).session(session);

  if (!toAccount) {
    await session.abortTransaction();
    console.log("Invalid account")
    return;
  }

  // Perform the transfer
  await Account.updateOne({ userId: req.userId }, { $inc: { balance: -amount } });
  await Account.updateOne({ userId: to }, { $inc: { balance: amount } });

  // Commit the transaction
  await session.commitTransaction();
  console.log("done")
}

transfer({
  userId: "65ac44e10ab2ec750ca666a5",
  body: {
    to: "65ac44e40ab2ec750ca666aa",
    amount: 100
  }
})

transfer({
```

```
        userId: "65ac44e10ab2ec750ca666a5",
        body: {
          to: "65ac44e40ab2ec750ca666aa",
          amount: 100
        }
      })
module.exports = router;
```

▼ Error

```
balance: 2099.5399690668773,
  __v: 0
)
/Users/harkiratsingh/Projects/100x/week-8.2/backend/node_modules/mongodb/lib/cmap/connection.js:205
  callback(new error_1.MongoServerError(document));
^

MongoServerError: WriteConflict error: this operation conflicted with another operation. Please retry your operation or multi-document transaction.
    at Connection.onMessage (/Users/harkiratsingh/Projects/100x/week-8.2/backend/node_modules/mongodb/lib/cmap/connection.js:205:26)
    at MessageStream.<anonymous> (/Users/harkiratsingh/Projects/100x/week-8.2/backend/node_modules/mongodb/lib/cmap/connection.js:64:60)
    at MessageStream.emit (node:events:519:28)
    at processIncomingData (/Users/harkiratsingh/Projects/100x/week-8.2/backend/node_modules/mongodb/lib/cmap/message_stream.js:117:16)
    at MessageStream._write (/Users/harkiratsingh/Projects/100x/week-8.2/backend/node_modules/mongodb/lib/cmap/message_stream.js:33:9)
    at writeOrBuffer (node:internalstreams/writable:564:12)
    at _write (node:internalstreams/writable:493:10)
    at Writable.write (node:internalstreams/writable:502:10)
    at Socket.ondata (node:internalstreams/readable:1007:22)
    at Socket.emit (node:events:519:28)
    at operationTime: Timestamp { low: 1, high: 1705839061, unsigned: true },
ok: 0,
code: 112,
```

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

You are using the Lightweight API Client, sign in or create an account to work with collections, environments and unlock all free features in Postman.

History New Import PUT http://lo... POST http://lo... GET http://lo... GET http://lo... POST http://lo... POST http://lo... POST http://lo... POST http://lo... GET http://lo... + ...

HTTP <http://localhost:3000/api/v1/account/balance>

GET http://localhost:3000/api/v1/account/balance

Params Authorization Headers (9) Body Pre-request Script Tests Settings Cookies

Headers (8 hidden)

Key Value

Authorization Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9eyJc2VySW...

Body Cookies Headers (8) Test Results

Pretty Raw Preview Visualize JSON

1 "balance": 2099.5399690668773

2

3

4

Body Cookies Headers (8) Test Results

Pretty Raw Preview Visualize JSON

200 OK 14 ms 297 B Save Response

Console Not connected to a Postman account

Make transfer

You are using the Lightweight API Client, sign in or create an account to work with collections, environments and unlock all free features in Postman.

History New Import < UT http://lo... POST http://lo... GET http://lo... GET http://lo... POST http://lo... POST http://lo... POST http://lo... GET http://lo... POST http://lo... GET http://lo... + ...

HTTP <http://localhost:3000/api/v1/account/transfer>

POST http://localhost:3000/api/v1/account/transfer

Params Authorization Headers (9) Body Pre-request Script Tests Settings Cookies

Body

none form-data x-www-form-urlencoded raw binary JSON Beautify

1 "to": "65ac44e10ab2ec750ca666a5",

2 "amount": "100"

3

4

Body Cookies Headers (8) Test Results

Pretty Raw Preview Visualize JSON

1 "message": "Transfer successful"

2

3

4

Body Cookies Headers (8) Test Results

Pretty Raw Preview Visualize JSON

200 OK 54 ms 300 B Save Response

Console Not connected to a Postman account

Get balance again (notice it went down)

The screenshot shows the Postman interface. On the left, the 'History' sidebar lists various API calls made today, including multiple GET and POST requests to the 'account' endpoint. The main panel displays a specific GET request to `http://localhost:3000/api/v1/account/balance`. The 'Body' tab of the request details shows a JSON response with a single key-value pair: `"balance": 1999.5399690668773`. The status bar at the bottom indicates a 200 OK response with 21 ms latency and 297 B size.

Mongo should look something like this -

The screenshot shows the MongoDB Compass interface connected to the 'paytm' database on port 27017. The 'paytm.accounts' collection is selected. The interface shows three documents with the following data:

- Document 1: `_id: ObjectId('65ac44df0ab2ec750ca666a2')`, `userId: ObjectId('65ac44df0ab2ec750ca666a0')`, `balance: 1736.5103102866142`, `__v: 0`
- Document 2: `_id: ObjectId('65ac44e10ab2ec750ca666a7')`, `userId: ObjectId('65ac44e10ab2ec750ca666a5')`, `balance: 513.0107076284458`, `__v: 0`
- Document 3: `_id: ObjectId('65ac44e40ab2ec750ca666ac')`, `userId: ObjectId('65ac44e40ab2ec750ca666aa')`, `balance: 1999.5399690668773`, `__v: 0`

Step 1 - Add routing to the react app

Import `react-router-dom` into your project and add the following routes -

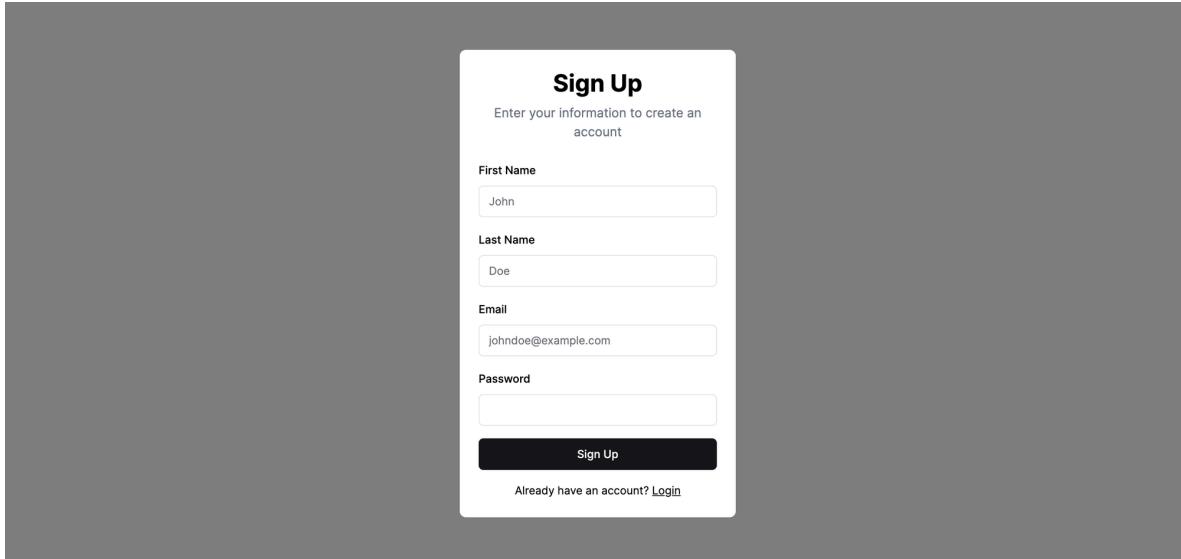
1. `/signup` - The signup page
2. `/signin` - The signin page
3. `/dashboard` - Balances and see other users on the platform.
4. `/send` - Send money to other users

▼ Solution

```
function App() {
  return (
    <>
      <BrowserRouter>
        <Routes>
          <Route path="/signup" element={<Signup />} />
          <Route path="/signin" element={<Signin />} />
          <Route path="/dashboard" element={<Dashboard />} />
          <Route path="/send" element={<SendMoney />} />
        </Routes>
      </BrowserRouter>
    </>
  )
}
```

Copy

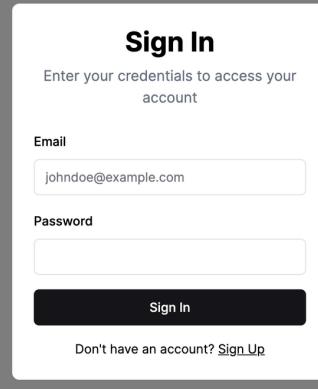
Step 2 - Create and hook up Signup page



If the user signup is successful, take the user to [/dashboard](#)

If not, show them an error message

Step 3 - Create the signin page



If the signin in successful, take the user to [/dashboard](#)

Step 4 - Dashboard page

Your Balance \$5000**Users**

Search users...

U1 User 1

Send Money

U2 User 2

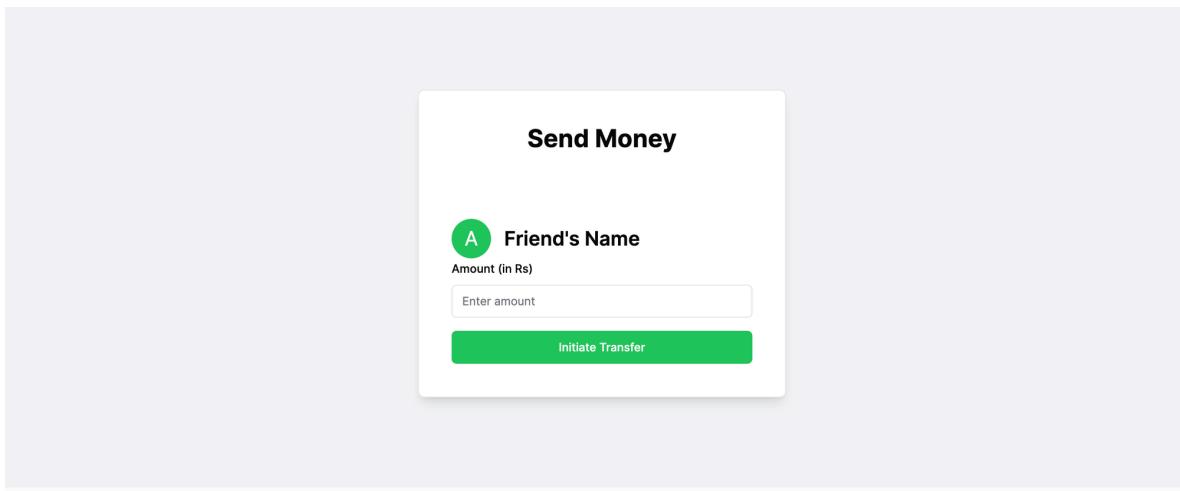
Send Money

U3 User 3

Send Money

Show the user their balance, and a list of users that exist in the database

Clicking on **Send money** should open a modal that lets the user send money

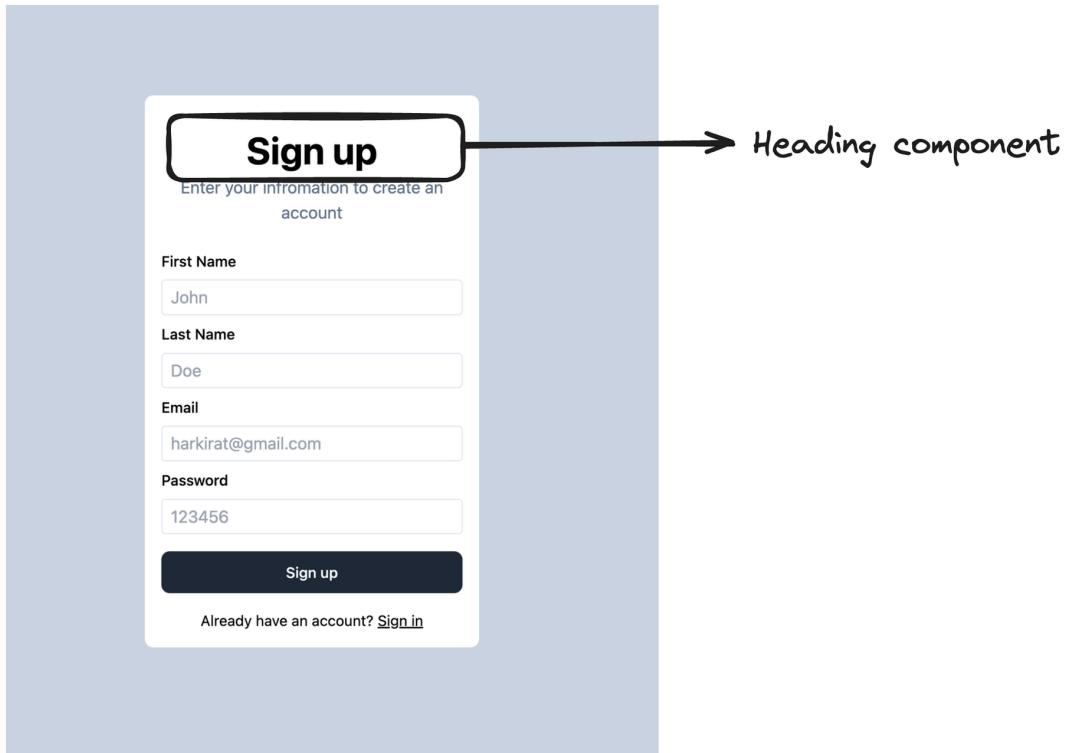


Step 5 - Auth Components

Full Signup component

You can break down the app into a bunch of components. The code only contains the styles of the component, not any onclick functionality.

1. Heading component

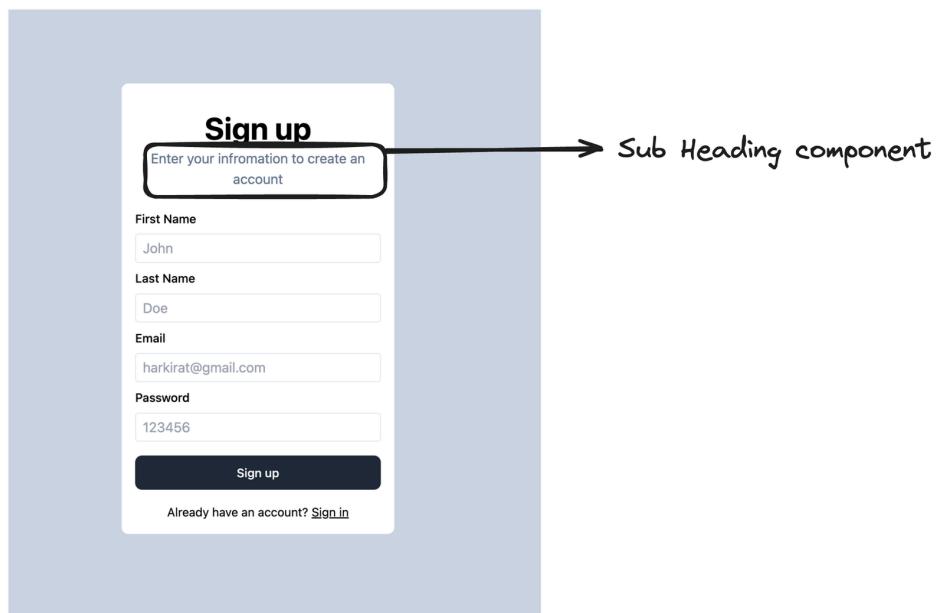


▼ Code

```
export function Heading({label}) {  
  return <div className="font-bold text-4xl pt-6">  
    {label}  
  </div>  
}
```

Copy

2. Sub Heading component

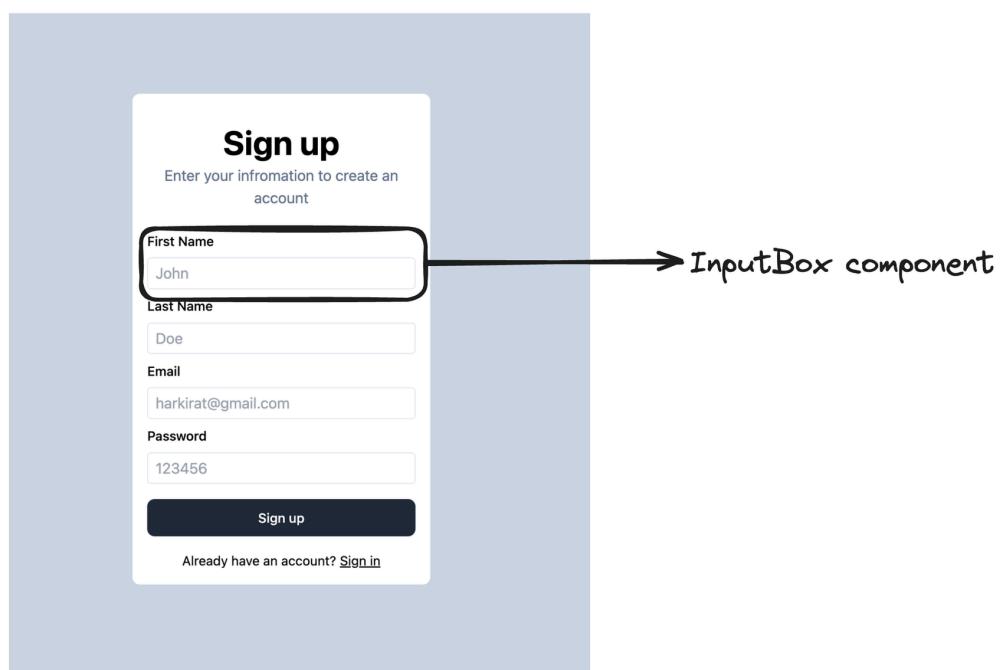


▼ Code

```
export function SubHeading({label}) {  
  return <div className="text-slate-500 text-md pt-1 px-4 pb-4">  
    {label}  
  </div>  
}
```

Copy

3. InputBox component



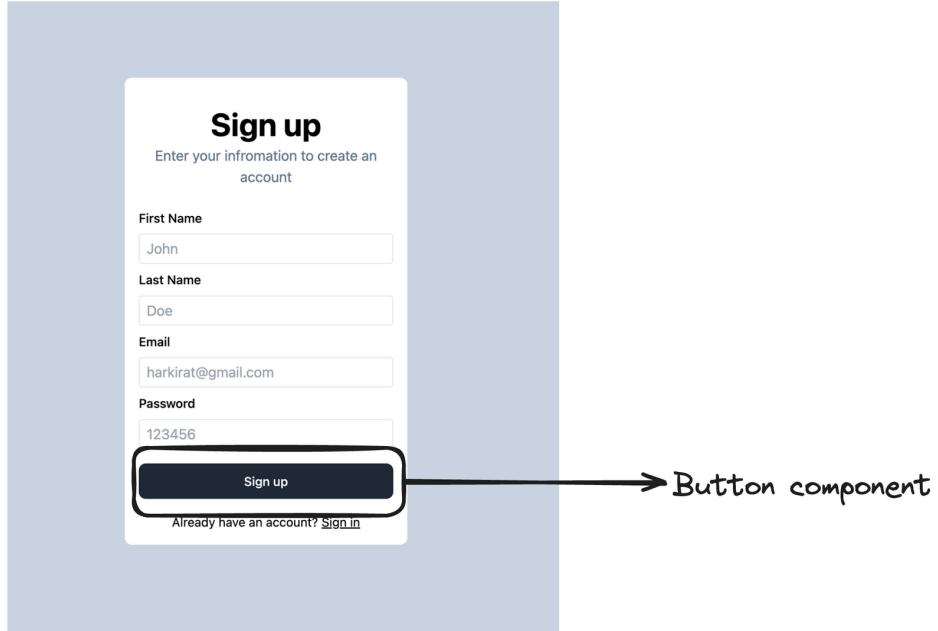
▼ Code

```
export function InputBox({label, placeholder}) {  
  return <div>
```

Copy

```
<div className="text-sm font-medium text-left py-2">
  {label}
</div>
<input placeholder={placeholder} className="w-full px-2 py-2" type="text"/>
</div>
}
```

4. Button Component



▼ Code

```
export function Button({label, onClick}) {
  return <button onClick={onClick} type="button" className="w-full h-10 rounded-md font-medium text-white bg-blue-700 border border-blue-700 transition duration-300 ease-in-out">{label}</button>
}
```

Copy

This section was blindly copied from <https://flowbite.com/docs/components/buttons/>

5. BottomWarning

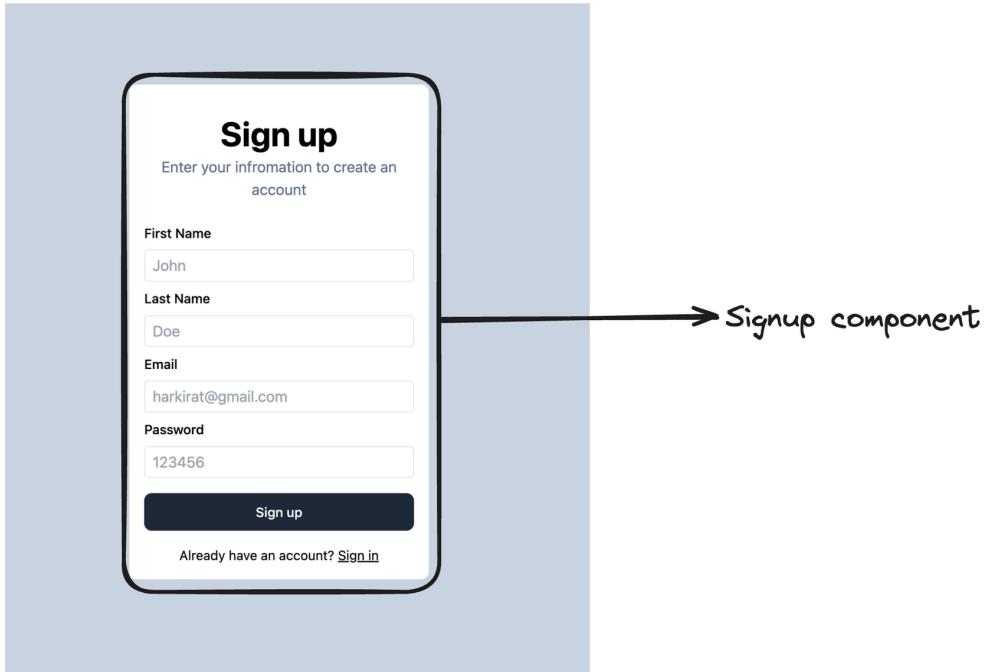


▼ Code

```
import { Link } from "react-router-dom"                                Copy

export function BottomWarning({label, buttonText, to}) {
  return <div className="py-2 text-sm flex justify-center">
    <div>
      {label}
    </div>
    <Link className="pointer underline pl-1 cursor-pointer" to={to}>
      {buttonText}
    </Link>
  </div>
}
```

Full Signup component



▼ Code

```
import { BottomWarning } from "../components/BottomWarning"          Copy
import { Button } from "../components/Button"
import { Heading } from "../components/Heading"
import { InputBox } from "../components/InputBox"
import { SubHeading } from "../components/SubHeading"

export const Signup = () => {
  return <div className="bg-slate-300 h-screen flex justify-cei
<div className="flex flex-col justify-center">
  <div className="rounded-lg bg-white w-80 text-center p-2 h
    <Heading label={"Sign up"} />
    <SubHeading label={"Enter your information to create an ac
      <InputBox placeholder="John" label={"First Name"} />
      <InputBox placeholder="Doe" label={"Last Name"} />
      <InputBox placeholder="harkirat@gmail.com" label={"Email"} />
      <InputBox placeholder="123456" label={"Password"} />
    <div className="pt-4">
      <Button label={"Sign up"} />
    </div>
    <BottomWarning label={"Already have an account?"} button
  </div>
</div>
</div>
}
```

Step 6 - Signin-ed Comonents

1. AppBar



▼ Code

```
export const AppBar = () => {
  return <div className="shadow h-14 flex justify-between">
    <div className="flex flex-col justify-center h-full ml-4">
      PayTM App
    </div>
    <div className="flex">
      <div className="flex flex-col justify-center h-full">
        Hello
      </div>
      <div className="rounded-full h-12 w-12 bg-slate-200 flex flex-col justify-center h-full">
        U
      </div>
    </div>
  </div>
}
```

Copy

2. Balance



▼ Code

```
export const Balance = ({ value }) => {
  return <div className="flex">
    <div className="font-bold text-lg">
      Your balance
    </div>
    <div className="font-semibold ml-4 text-lg">
      Rs {value}
    </div>
  </div>
}
```

Copy

3. Users component



▼ Code

```
import { useState } from "react"
import { Button } from "./Button"

export const Users = () => {
  // Replace with backend call
  const [users, setUsers] = useState([
    {
      firstName: "Harkirat",
      lastName: "Singh",
      _id: 1
    }
  ]);

  return <>
    <div className="font-bold mt-6 text-lg">
      Users
    </div>
    <div className="my-2">
      <input type="text" placeholder="Search users..." className="form-control" />
    </div>
    <div>
      {users.map(user => <User user={user} />)}
    </div>
  </>
}
```

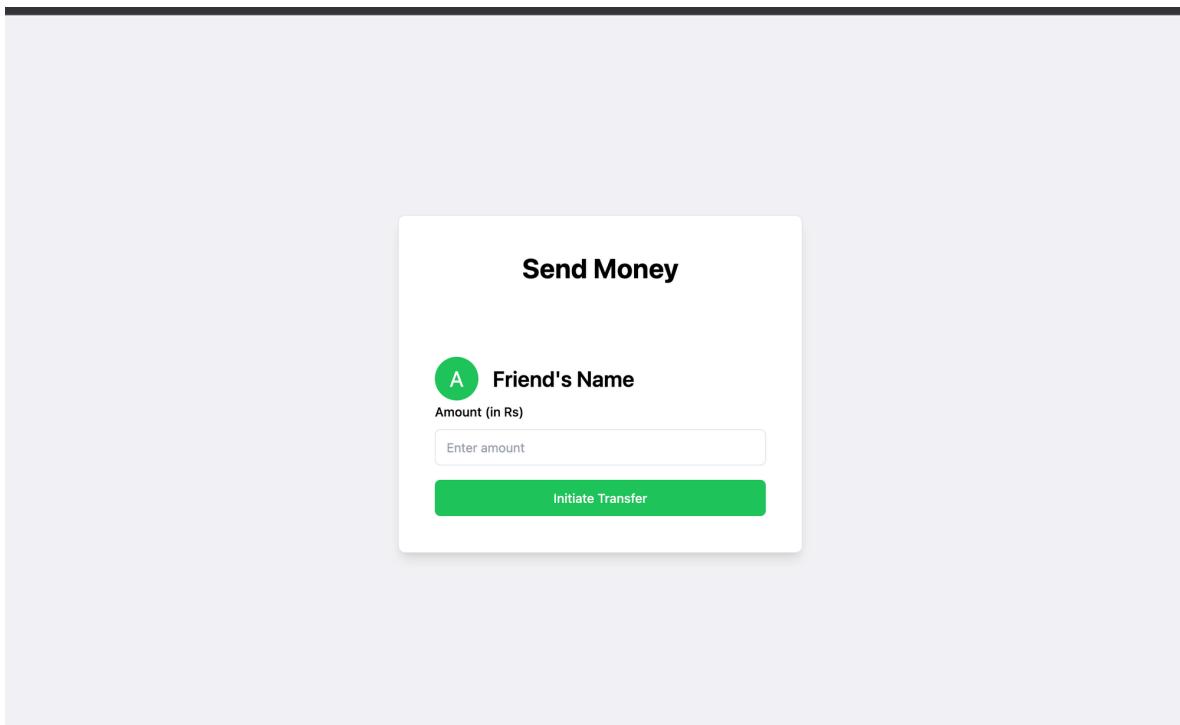
Copy

```
</>
}

function User({user}) {
  return <div className="flex justify-between">
    <div className="flex">
      <div className="rounded-full h-12 w-12 bg-slate-200" style={{borderRadius: "50%", width: "24px", height: "24px", display: "flex", alignItems: "center", justifyContent: "center", gap: "8px", padding: "2px 0", margin: "0 auto", width: "24px", height: "24px", border: "1px solid #ccc", border-radius: "50%}}>
        <img alt="User icon" style={{width: "16px", height: "16px", border: "1px solid #ccc", border-radius: "50%}}/>
        {user.firstName[0]}
      </div>
    </div>
    <div className="flex flex-col justify-center h-ful">
      <div>
        {user.firstName} {user.lastName}
      </div>
    </div>
  </div>

  <div className="flex flex-col justify-center h-ful">
    <Button label={"Send Money"} />
  </div>
</div>
}
```

4. SendMoney Component



▼ Code

```

export const SendMoney = () => {
  return <div class="flex justify-center h-screen bg-gray-100">
    <div className="h-full flex flex-col justify-center">
      <div>
        <div class="border h-min text-card-foreground max-w-md p-6">
          <div class="flex flex-col space-y-1.5 p-6">
            <h2 class="text-3xl font-bold text-center">Send Money</h2>
            <div class="p-6">
              <div class="flex items-center space-x-4">
                <div class="w-12 h-12 rounded-full bg-green-100" style="font-size: 1.5em; border-radius: 50%; text-align: center; width: 1.5em; height: 1.5em; margin-right: 0.5em;"></div>
                <span class="text-2xl text-white">A</span>
              </div>
              <h3 class="text-2xl font-semibold">Friend's Information</h3>
            </div>
            <div class="space-y-4">
              <div class="space-y-2">
                <label
                  class="text-sm font-medium leading-none italic" for="amount">
                  Amount (in Rs)
                </label>
                <input
                  type="number"
                  class="flex h-10 w-full rounded-md border border-gray-300 py-2 px-3" id="amount"
                  placeholder="Enter amount"
                />
              </div>
              <button class="justify-center rounded-md text-blue-600 py-2 px-4">
                Initiate Transfer
              </button>
            </div>
          </div>
        </div>
      </div>
    </div>
  </div>
}

```

Step 7 - Wiring up the backend calls

You can use

1. fetch or
2. axios

to wire up calls to the backend server.

The final code looks something like this -

<https://github.com/100xdevs-cohort-2/paytm/tree/complete-solution> (complete-solution branch on the repo)

The important bits here are -

1. Signup call - <https://github.com/100xdevs-cohort-2/paytm/blob/complete-solution/frontend/src/pages/Signup.jsx#L36>
2. Call to get all the users given the filter - <https://github.com/100xdevs-cohort-2/paytm/blob/complete-solution/frontend/src/components/Users.jsx#L13>
3. Call to transfer money b/w accounts - <https://github.com/100xdevs-cohort-2/paytm/blob/complete-solution/frontend/src/pages/SendMoney.jsx#L45>