PeerPay: A Blockchain-Based Transaction Visualization System

Phase 2 Report: Implementation & Execution

Harsh Dayal
Registration Number: 22BCE10564

Course: IBM blockchain
Institution: VIT

April 19, 2025

Project Repository: https://github.com/Kaos599/PeerPay

Contents

1 Project Overview

Following the proposal outlined in Phase 1, I proceeded with the implementation and execution of the PeerPay project in this phase. PeerPay is my full-stack web application designed to demonstrate fundamental blockchain concepts through a simple peer-to-peer transaction system. My application allows users to view blocks containing transaction data and submit new transactions, which are then added to a persistent, cryptographically linked chain managed by a Node.js backend and stored in MongoDB. I built the frontend using modern React technologies for an interactive user experience.

This Phase 2 report documents the development process, including:

- The definition of the blockchain data structure and implementation of the backend logic.
- Integration with MongoDB for data persistence.
- Creation of the RESTful APIs using Node.js/Express.
- Development of the React frontend for user interaction.
- Initial testing and validation of the core functionality implemented.

2 System Architecture

I designed the system with distinct frontend, backend, and database components.

2.1 Components Description

1. **Frontend (Client Side):** Built with React and Vite, using Chakra UI for components and TanStack Query for API interaction and state management (including polling). React Hook Form and Zod handle transaction input and validation.
2. **Backend (Server Side):** Developed using Node.js and Express.js, it provides RESTful APIs, manages the blockchain logic (block creation, SHA-256 hashing via the ‘crypto‘ module), and communicates with the database.
3. **Database:** MongoDB stores the blockchain data (blocks), accessed via the Mongoose ODM from the backend.
4. **Blockchain Structure:** A custom structure defined by the Mongoose schema, linking blocks via ‘previousHash‘ and ensuring integrity via each block’s ‘hash‘.

2.2 Architecture Diagram

Figure 1 illustrates the interaction between these components.

3 Implementation Steps

I followed these steps to build the application:

3.1 Backend Development (Node.js + Express + MongoDB)

- Set up the Node.js project structure and installed dependencies (Express, Mongoose, CORS, etc.).
- Established a connection to my MongoDB Atlas database using Mongoose.
- Defined the ‘blockSchema‘ in Mongoose, specifying fields (‘index‘, ‘timestamp‘, ‘data‘, ‘previousHash‘, ‘hash‘) and types. Added the ‘calculateHash‘ instance method using ‘crypto.createHash(’sha256’)‘.

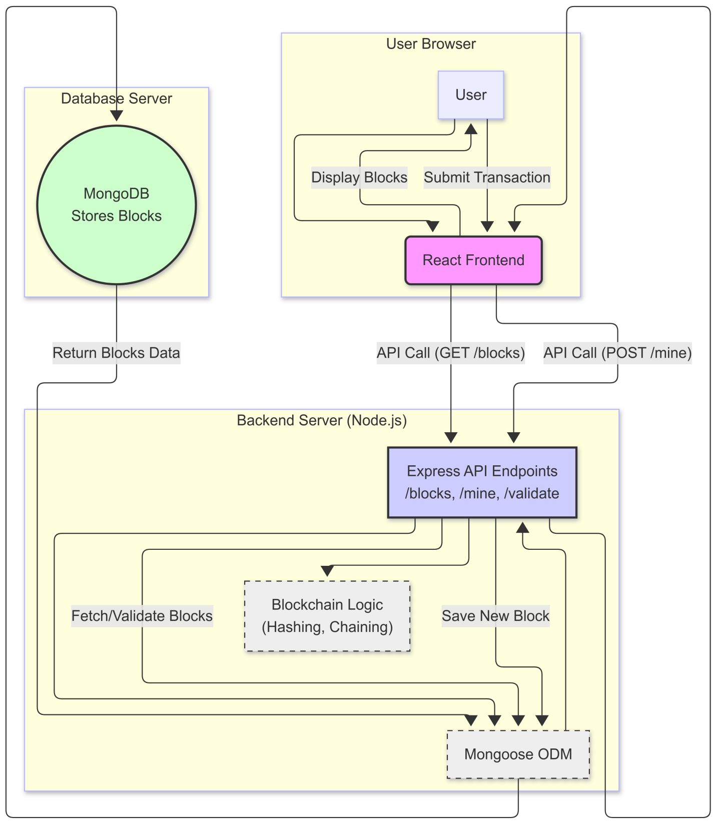


Figure 1: PeerPay System Architecture Implemented in Phase 2.

- Implemented the ‘ensureGenesisBlock‘ function to initialize the blockchain in the database if empty.
- Developed the core API endpoints as described in Table 1.
- Configured CORS middleware for frontend access and ‘express.json‘ for request body parsing.

Table 1: Backend API Endpoints Implemented.

Method	Path	Description
GET	‘/blocks‘	Retrieves all blocks from the database, sorted by index.
POST	‘/mine‘	Accepts transaction data (sender, recipient, amount), creates a new block, calculates its hash, links it to the previous block, and saves it to the database.
GET	‘/validate‘	Fetches the entire chain and performs integrity checks (hash recalculation, previous hash links, index sequence).

3.2 Frontend Development (React + Vite + Chakra UI)

- Initialized the React project using Vite and installed necessary packages (Chakra UI, TanStack Query, React Hook Form, Zod, Framer Motion, Lucide React).

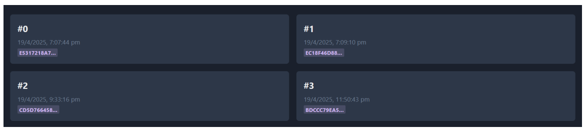
- Set up Chakra UI provider and theme (including light/dark mode).
- Created reusable components: ‘Header‘, ‘Footer‘, ‘BlockCard‘ (for displaying block details), ‘TransactionForm‘.
- Implemented API service functions (‘src/services/blockchain.js‘) using ‘fetch‘ or ‘axios‘ to call the backend endpoints.
- Utilized TanStack Query (‘useQuery‘) for fetching ‘/blocks‘, enabling ‘refetchInterval‘ for polling.
- Used TanStack Query (‘useMutation‘) to handle the ‘POST /mine‘ requests triggered by form submission.
- Built the ‘TransactionForm‘ using React Hook Form, integrated Zod for validation schema.
- Implemented ‘useToast‘ from Chakra UI for user feedback on transaction status.
- Added basic animations using Framer Motion.

3.3 Testing and Validation

- **Unit/Component Testing (Conceptual):** Ensured individual components rendered correctly and form validation logic worked as expected during development.
- **API Testing:** Used Postman to directly test responses and behavior of the ‘/blocks‘, ‘/mine‘, and ‘/validate‘ endpoints.
- **Integration Testing:** Tested the end-to-end flow: submitting a transaction via the UI, verifying the block appears in the list, checking the database, and confirming the ‘/validate‘ endpoint reports a valid chain.
- **Validation Checks:** Manually triggered the ‘/validate‘ endpoint after adding blocks and simulated data tampering in the database to confirm the validation logic caught inconsistencies.

4 Screenshots (Placeholders)

The following figures represent key UI elements and outputs.


#0
19/4/2025, 7:07:44 pm
E5317218A7...

#1
19/4/2025, 7:09:10 pm
EC18F46D88...

#2
19/4/2025, 9:33:16 pm
CD5D766458...

#3
19/4/2025, 11:50:43 pm
BDCCC79EA5...

Figure 2: Blockchain Visualization UI.

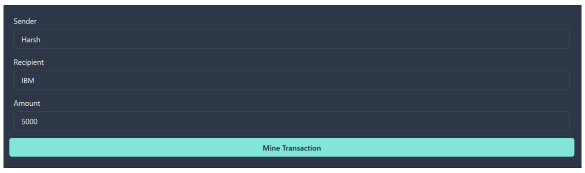

Sender
Harsh
Recipient
IBM
Amount
5000

Figure 3: Add Transaction Form UI.

5 Code Snippets

Key code segments from the implementation are shown below.

Transaction mined!

Figure 4: Success Notification.

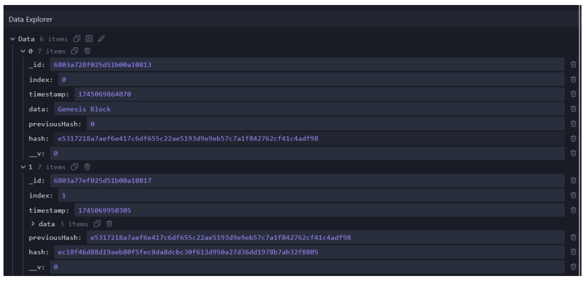

```
Data Explorer
Data 6 items
  0 7 items
    _id: 6803a728f025d51b00a10813
    index: 0
    timestamp: 1745069864070
    data: Genesis Block
    previousHash: 0
    hash: e5317218a7aef6e417c6df655c22ae5193d9e9eb57c7a1f842762cf41c4adf98
    __v: 0
  1 7 items
    _id: 6803a77ef025d51b00a10817
    index: 1
    timestamp: 1745069950305
    > data 3 items
    previousHash: e5317218a7aef6e417c6df655c22ae5193d9e9eb57c7a1f842762cf41c4adf98
    hash: ec18f46d88d19aeb80f5fec8da8dcbc30f613d950a27d36dd1978b7ab32f8805
    __v: 0
```

Figure 5: Backend Log Example (Optional).

5.1 Backend: Mongoose Block Schema

```
const blockSchema = new mongoose.Schema({
    index: { type: Number, required: true, unique: true },
    timestamp: { type: Number, required: true },
    data: { type: Object, required: true }, // { sender, recipient, amount }
    previousHash: { type: String, required: true },
    hash: { type: String, required: true, unique: true }
});

blockSchema.methods.calculateHash = function() {
    return crypto.createHash('sha256').update(
        this.index + this.previousHash + this.timestamp + JSON.stringify(this.data)
    ).digest('hex');
};

const BlockModel = mongoose.model('Block', blockSchema);
```

Listing 1: Mongoose Schema for Blocks (backend/server.js)

5.2 Backend: Add Block API Endpoint ('/mine')

```
app.post('/mine', async (req, res) => {
    const transactionData = req.body.data;
    // Basic validation ... (omitted for brevity)

    try {
        const latestBlock = await BlockModel.findOne().sort({ index: -1 });
        if (!latestBlock) { /* Error handling ... */ }

        const newIndex = latestBlock.index + 1;
        const newTimestamp = Date.now();
        const newPreviousHash = latestBlock.hash;

        const newBlock = new BlockModel({
            index: newIndex,
            timestamp: newTimestamp,
            data: transactionData,
            previousHash: newPreviousHash,
            hash: '' // Placeholder
        });
        newBlock.hash = newBlock.calculateHash(); // Calculate the hash
        await newBlock.save(); // Save to DB

        console.log('Transaction Block added:', newBlock);
        res.status(201).send(newBlock);
    } catch (error) { /* Error handling ... */ }
});
```

Listing 2: POST /mine Endpoint Logic (backend/server.js)

6 Challenges Faced and Solutions

During implementation, I encountered several challenges, which I addressed as detailed in Table 2.

Table 2: Implementation Challenges and Solutions.

Challenge	**Solution Applied**
Real-time UI Updates	Utilized TanStack Query's polling mechanism ('refetchInterval') on the 'GET /blocks' query to automatically refresh the block list display on the frontend.
Asynchronous Operations Handling	Consistently employed 'async/await' syntax for handling Promises returned by database operations (Mongoose) and API calls ('fetch'/'axios'), and managed loading/error states effectively using TanStack Query hooks.
CORS Issues	Configured the 'cors' middleware in the Express backend application to explicitly allow requests originating from the frontend's development server address (e.g., 'http://localhost:5173').
Form State Management & Validation	Integrated React Hook Form for efficient form state handling and Zod for defining a clear schema for transaction data, enabling robust client-side validation before submission.
Database Initialization & Connection Management	Implemented error handling for the initial MongoDB connection and developed the 'ensureGenesisBlock' function to guarantee the blockchain's starting point upon successful connection.

7 Evaluation Criteria Mapping

I evaluated the outcome of this phase against the initial criteria, summarized in Table 3.

8 Current Output and Status

As of the end of Phase 2:

- The PeerPay application allows users to view the blockchain fetched from the backend.
- Users can submit new transactions through a validated form I created.
- New blocks are successfully created, hashed, linked, and persisted in MongoDB.
- The frontend UI updates via polling to show the latest state of the chain.
- My backend provides an API for block retrieval, addition, and validation.
- The system demonstrates the intended blockchain concepts.

Table 3: Evaluation Against Project Criteria.

Criteria	Outcome Achieved in Phase 2
Correctness	The system functions correctly: transactions are added, blocks are created and linked with valid hashes, data persists in MongoDB, and the chain validation endpoint confirms integrity. The UI displays data accurately.
Completeness	All core features defined in the proposal were implemented: block visualization, transaction submission with validation, backend API endpoints for core operations, and database persistence.
Code Quality	I maintained a modular structure in both frontend (components, services, hooks) and backend (routes, schema). I applied modern JavaScript practices (ES6+, async/await) and utilized established libraries effectively.
Frontend UI/UX	I delivered a modern, responsive UI using Chakra UI, providing clear block visualization, intuitive form interaction, appropriate user feedback (toasts), and aesthetic features like theme switching.

9 Conclusion

Phase 2 successfully translated the project proposal into a working application. I implemented the core functionalities of PeerPay, establishing the backend API, database persistence, and the interactive React frontend. My system effectively demonstrates fundamental blockchain principles using a modern technology stack. The challenges encountered were managed through appropriate technical solutions, resulting in a functional prototype ready for final testing, refinement, and reporting in Phase 3.