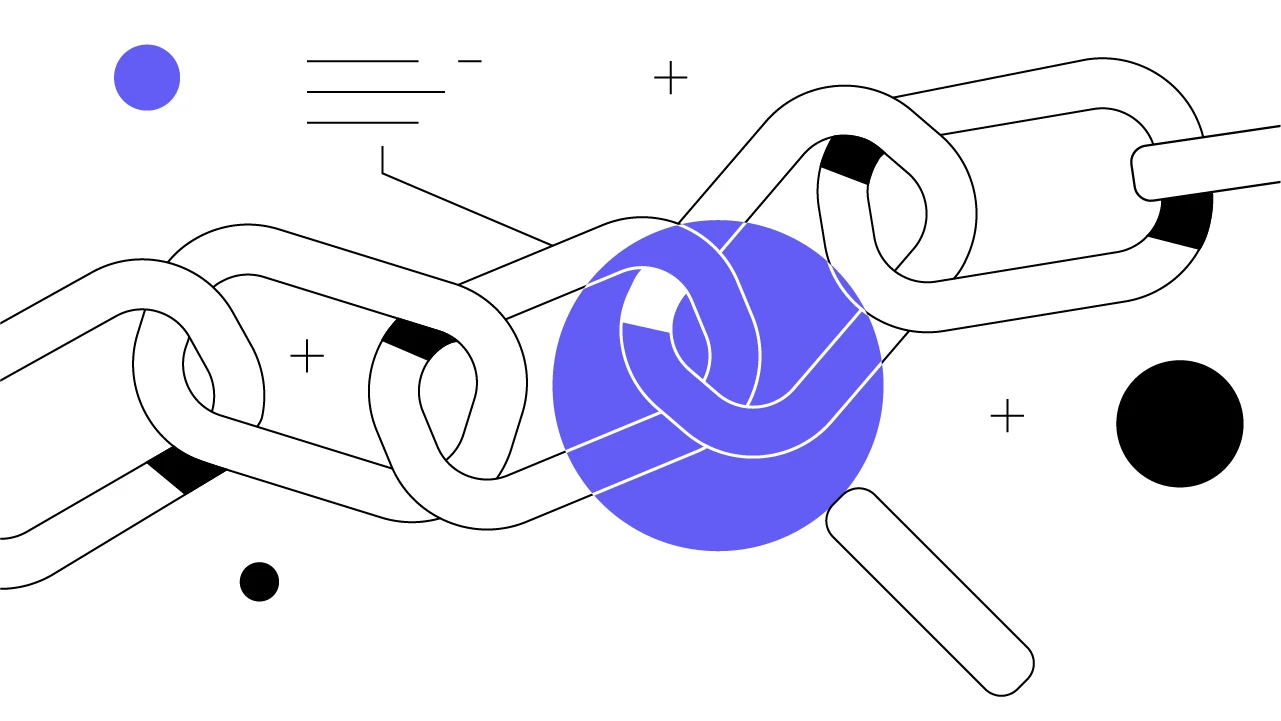
Project : Bitcoin Explorer

# Introduction

Course: INFO7500 Cryptocurrency / Smart Contact FALL 2024

Team: Akshatha Patil, Sumanayana Konda, Ruthwik Bommenahalli Gowda

In this CodeLab, we'll build a comprehensive Bitcoin Explorer that provides real-time insights into both on-chain and off-chain metrics. This project combines the power of Rust for efficient data ingestion, Node.js for a robust backend, and React for an interactive frontend. By the end of this tutorial, you'll have a full-stack application that continuously fetches Bitcoin data, stores it in a PostgreSQL database, and presents it through an intuitive web interface.



What you'll learn:

* How to ingest Bitcoin data using Rust
* Building a RESTful API with Node.js and Express
* Creating an interactive frontend with React and Chart.js
* Working with PostgreSQL for data storage
* Integrating on-chain and off-chain Bitcoin metrics

Why this stack?

* Rust: Offers high performance and safety for data ingestion
* Node.js: Provides a fast, scalable backend environment
* React: Enables the creation of a dynamic, responsive UI
* PostgreSQL: Offers robust, relational data storage

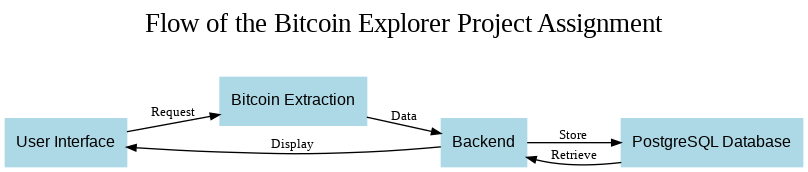
# Architecture

The project follows a three-tier architecture:

1. Frontend: React-based user interface
2. Backend: Node.js server with Express.js
3. Data Layer: PostgreSQL database and Rust-based Bitcoin data extractor

The application flow is as follows:

1. The Rust program extracts Bitcoin data from an API.
2. Extracted data is stored in the PostgreSQL database.
3. The Node.js backend serves this data through an API.
4. The React frontend fetches and displays the data.



# How to run the Application?

## Prerequisites

* Rust (latest stable version)
* Node.js (v14 or later)
* PostgreSQL (v12 or later)
* npm or yarn

## Setup

1. Clone the repository:

| git clone https://github.com/your-repo/bitcoin-explorer.git  cd bitcoin-explorer |
| --- |

1. Set up the database:

| psql -c "CREATE DATABASE bitcoin\_explorer" |
| --- |

1. Set up environment variables:

Create a .env file in the root directory with the following content:

| DATABASE\_URL=postgres://username:password@localhost/bitcoin\_explorer |
| --- |

1. Install Dependencies

| # For Rust ingestion  cd ingestion  cargo build  # For Node.js backend  cd ../backend  npm install  # For React frontend  cd ../frontend  npm install |
| --- |

1. Run the application:

| # Start the Rust ingestion (in ingestion directory)  cargo run  # Start the Node.js backend (in backend directory)  npm start  # Start the React frontend (in frontend directory)  npm start |
| --- |

Visit http://localhost:3000 to view the application.

# Bitcoin Extraction

The Rust component is responsible for fetching Bitcoin data and storing it in our PostgreSQL database. We chose Rust for its performance and safety features, making it ideal for continuous data ingestion.

## Key Components

1. API Client: Uses reqwest to fetch data from Bitcoin APIs.
2. Database Connection: Utilizes tokio-postgres for asynchronous database operations.
3. Data Models: Defines structures for Bitcoin block and price data.

## Code Explanation

| use reqwest::Error as ReqwestError;  use serde::{Deserialize};  use tokio\_postgres::{Client, NoTls, Error as PgError};  #[derive(Deserialize, Debug)]  struct BlockchainApiResponse {  height: u64,  hash: String,  time: String,  // ... other fields  }  async fn fetch\_bitcoin\_data(url: &str) -> Result<BlockchainApiResponse, ReqwestError> {  let response = reqwest::get(url).await?.json::<BlockchainApiResponse>().await?;  Ok(response)  }  async fn insert\_bitcoin\_data(client: &Client, block: &BlockchainApiResponse) -> Result<(), PgError> {  client.execute(  "INSERT INTO bitcoin\_details (height, hash, time) VALUES ($1, $2, $3)",  &[&(block.height as i64), &block.hash, &block.time],  ).await?;  Ok(())  }  #[tokio::main]  async fn main() -> Result<(), Box<dyn std::error::Error>> {  // ... database connection setup  loop {  let block\_data = fetch\_bitcoin\_data("https://api.example.com/bitcoin").await?;  insert\_bitcoin\_data(&client, &block\_data).await?;  tokio::time::sleep(tokio::time::Duration::from\_secs(60)).await;  }  } |
| --- |

This code fetches Bitcoin data every 60 seconds and inserts it into our database. We use tokio for asynchronous operations, allowing efficient handling of I/O-bound tasks.

# Backend : NodeJS

The Node.js backend serves as an intermediary between our database and frontend. It provides a RESTful API to fetch and serve Bitcoin data.

## Key Components

1. Express Server: Handles HTTP requests and routes.
2. PostgreSQL Client: Connects to and queries our database.
3. API Routes: Defines endpoints for fetching Bitcoin data.

| const express = require('express');  const { Pool } = require('pg');  const cors = require('cors');  const app = express();  const pool = new Pool({  connectionString: process.env.DATABASE\_URL,  });  app.use(cors());  app.get('/api/historical', async (req, res) => {  try {  const result = await pool.query('SELECT \* FROM bitcoin\_details ORDER BY time DESC LIMIT 100');  res.json(result.rows);  } catch (err) {  console.error(err);  res.status(500).json({ error: 'Internal server error' });  }  });  app.listen(3000, () => console.log('Server running on port 3000')); |
| --- |

This Express server provides an endpoint to fetch historical Bitcoin data. We use a connection pool for efficient database connections and implement error handling for robustness.

# Frontend : React

The React frontend presents our Bitcoin data in an interactive, user-friendly interface. We use Chart.js for data visualization and styled-components for styling.

## Key Components

1. React Hooks: Manage state and side effects.
2. Chart.js: Renders interactive charts for Bitcoin metrics.
3. Styled Components: Provides scoped, component-based styling.

## Code Explanation

| import React, { useState, useEffect } from 'react';  import styled from 'styled-components';  import { Line } from 'react-chartjs-2';  function App() {  const [bitcoinData, setBitcoinData] = useState([]);  useEffect(() => {  fetch('http://localhost:3000/api/historical')  .then(response => response.json())  .then(data => setBitcoinData(data));  }, []);  const chartData = {  labels: bitcoinData.map(d => new Date(d.time).toLocaleDateString()),  datasets: [{  label: 'Bitcoin Price (USD)',  data: bitcoinData.map(d => d.price),  borderColor: 'rgb(75, 192, 192)',  }]  };  return (  <div>  <h1>Bitcoin Explorer</h1>  <Line data={chartData} />  </div>  );  }  export default App; |
| --- |

This React component fetches Bitcoin data on mount and renders a line chart of Bitcoin prices. We use the useEffect hook for data fetching and useState for state management.

# Metrics

We track several key Bitcoin metrics:

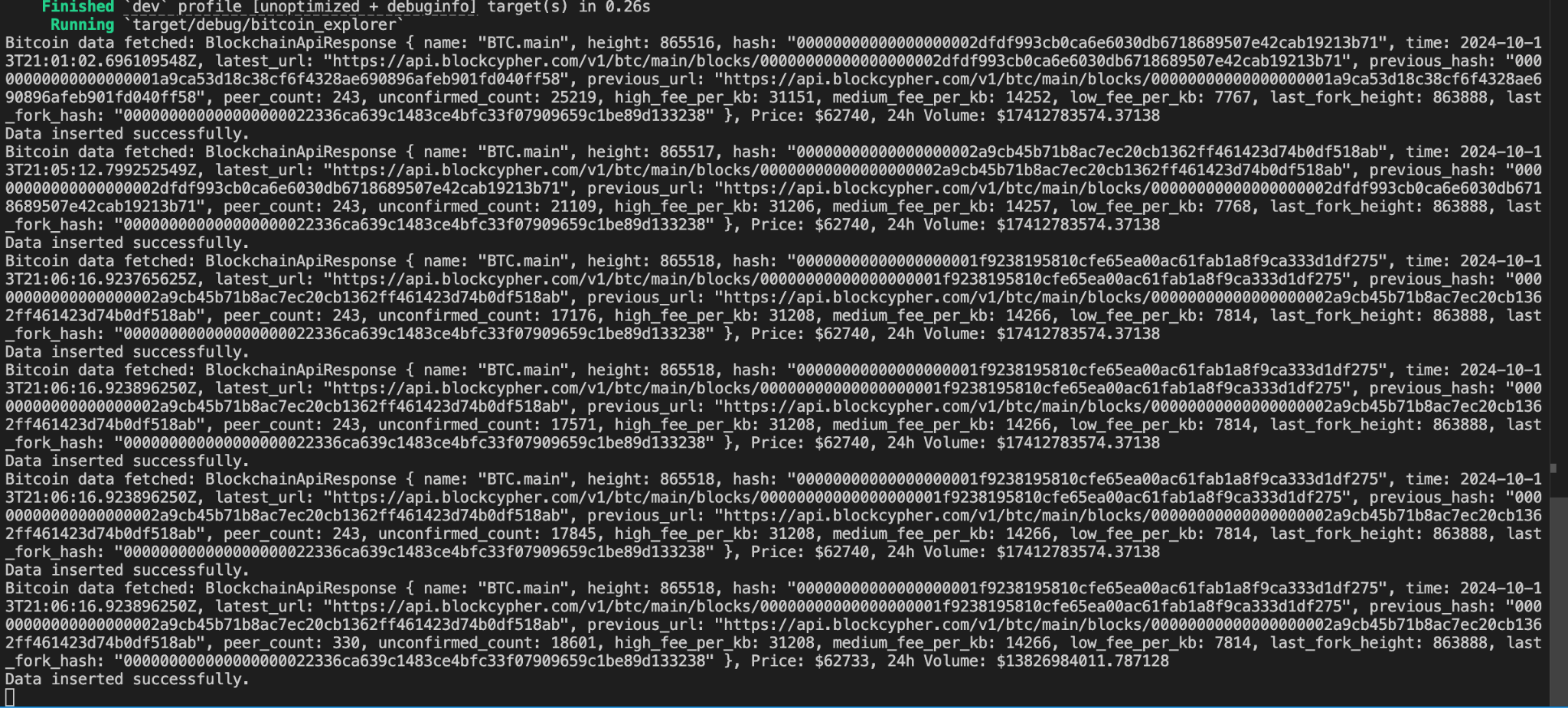
1. Block Height: Indicates the current length of the blockchain, reflecting network growth.
2. Price: An off-chain metric crucial for understanding Bitcoin's market value.
3. Transaction Fees: Reflects network congestion and user prioritization.
4. Peer Count: Indicates network health and decentralization level.
5. Unconfirmed Transactions: Shows current network activity and potential congestion.

These metrics provide a comprehensive view of both the Bitcoin blockchain's state and its broader ecosystem.

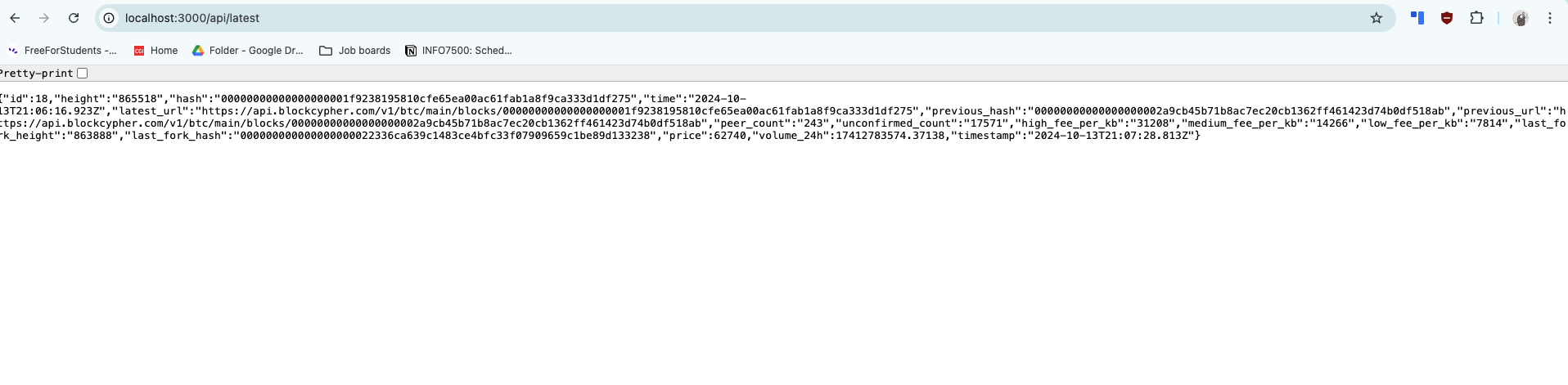
# Results

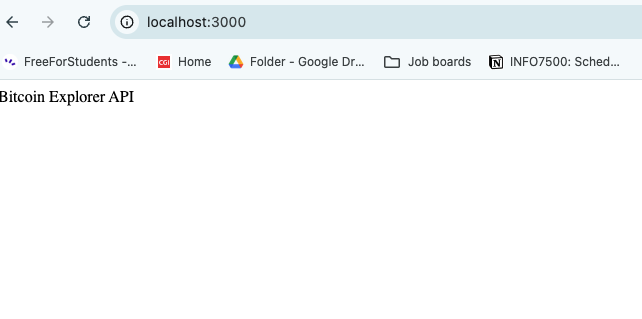
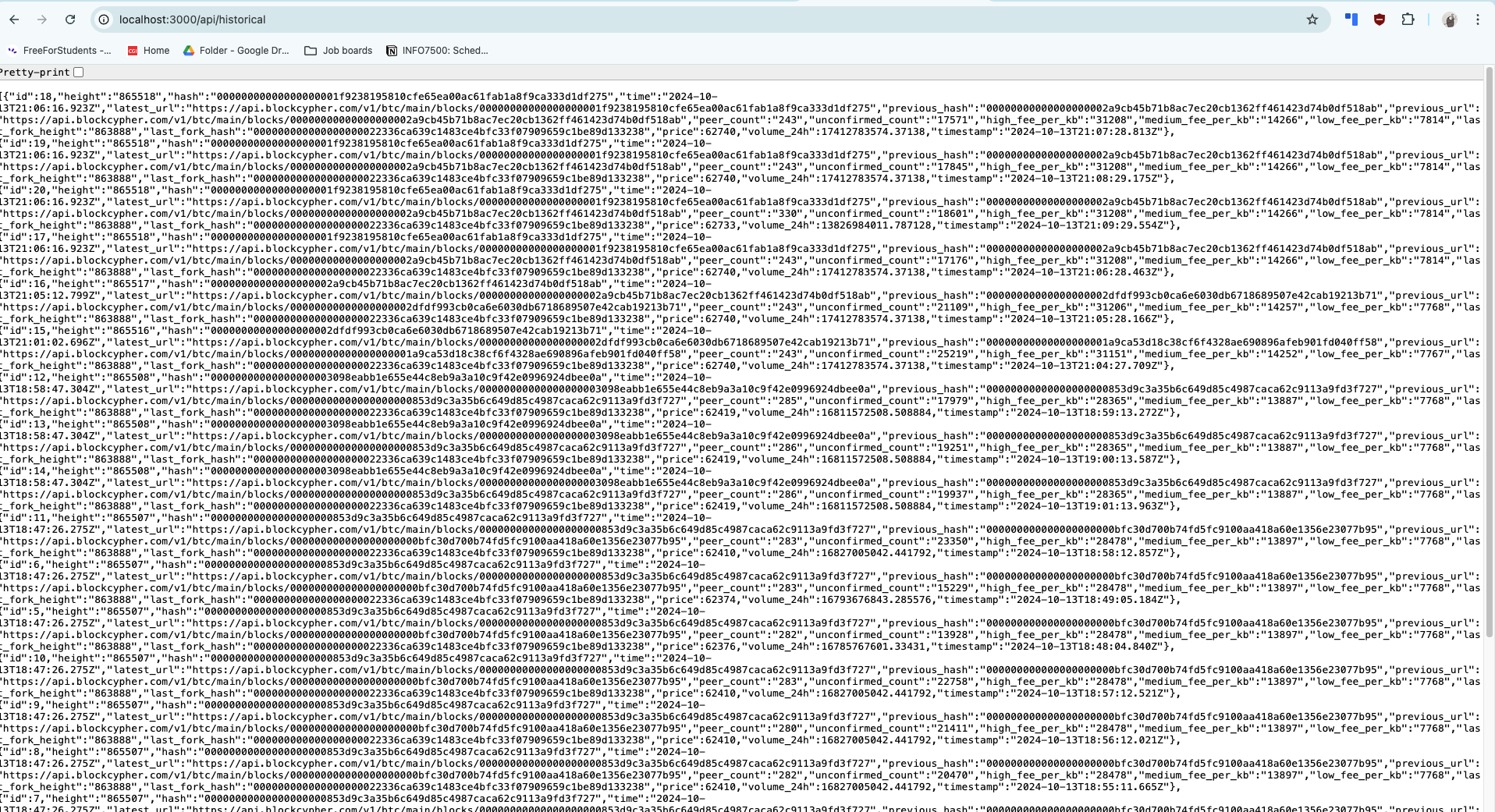
Our Bitcoin Explorer provides real-time insights into the Bitcoin network, allowing users to track both on-chain and off-chain metrics through an intuitive interface. This project demonstrates the power of combining Rust's performance, Node.js's scalability, and React's interactivity in building a full-stack blockchain explorer.

Bitcoin Extraction:

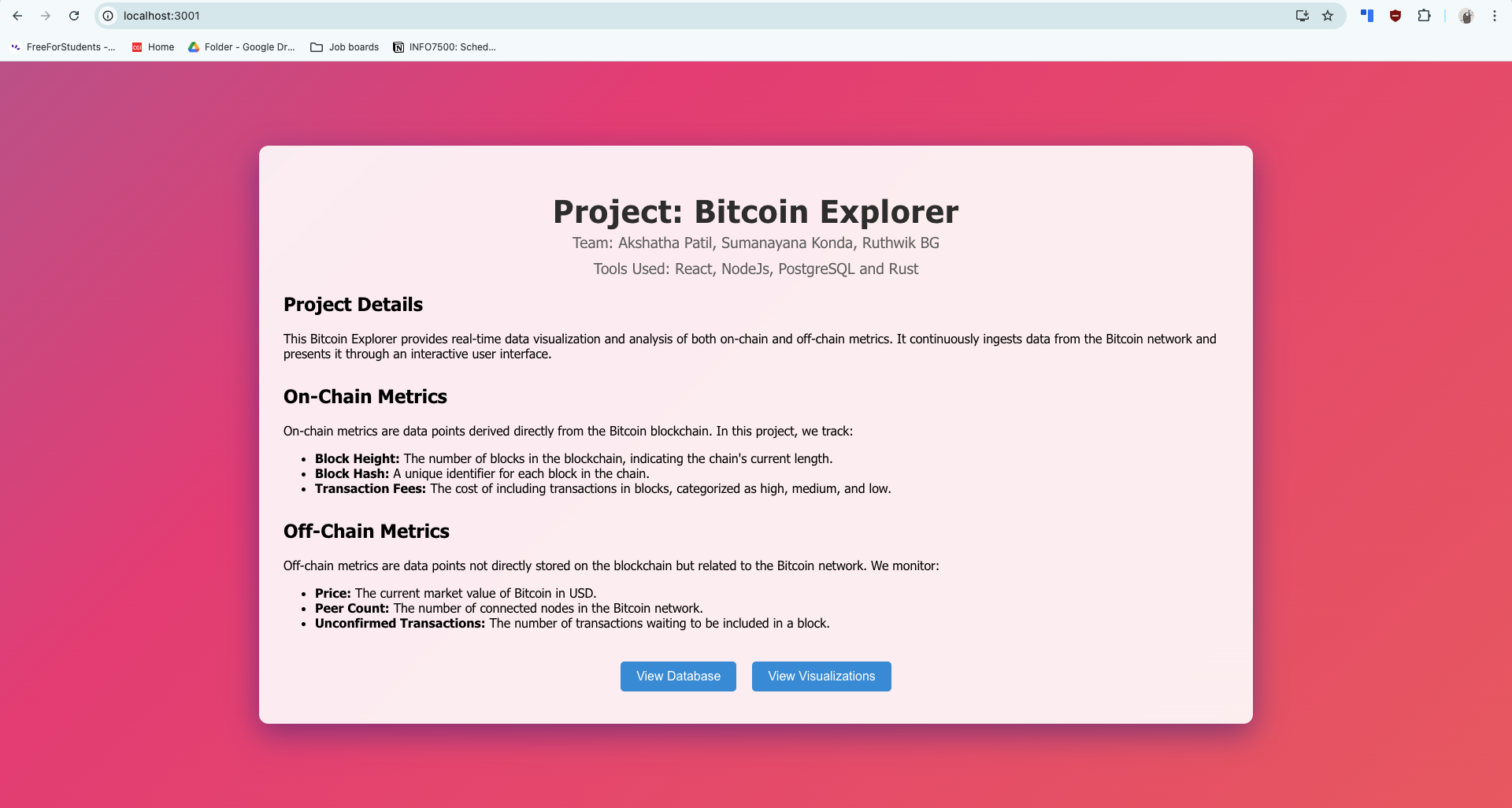


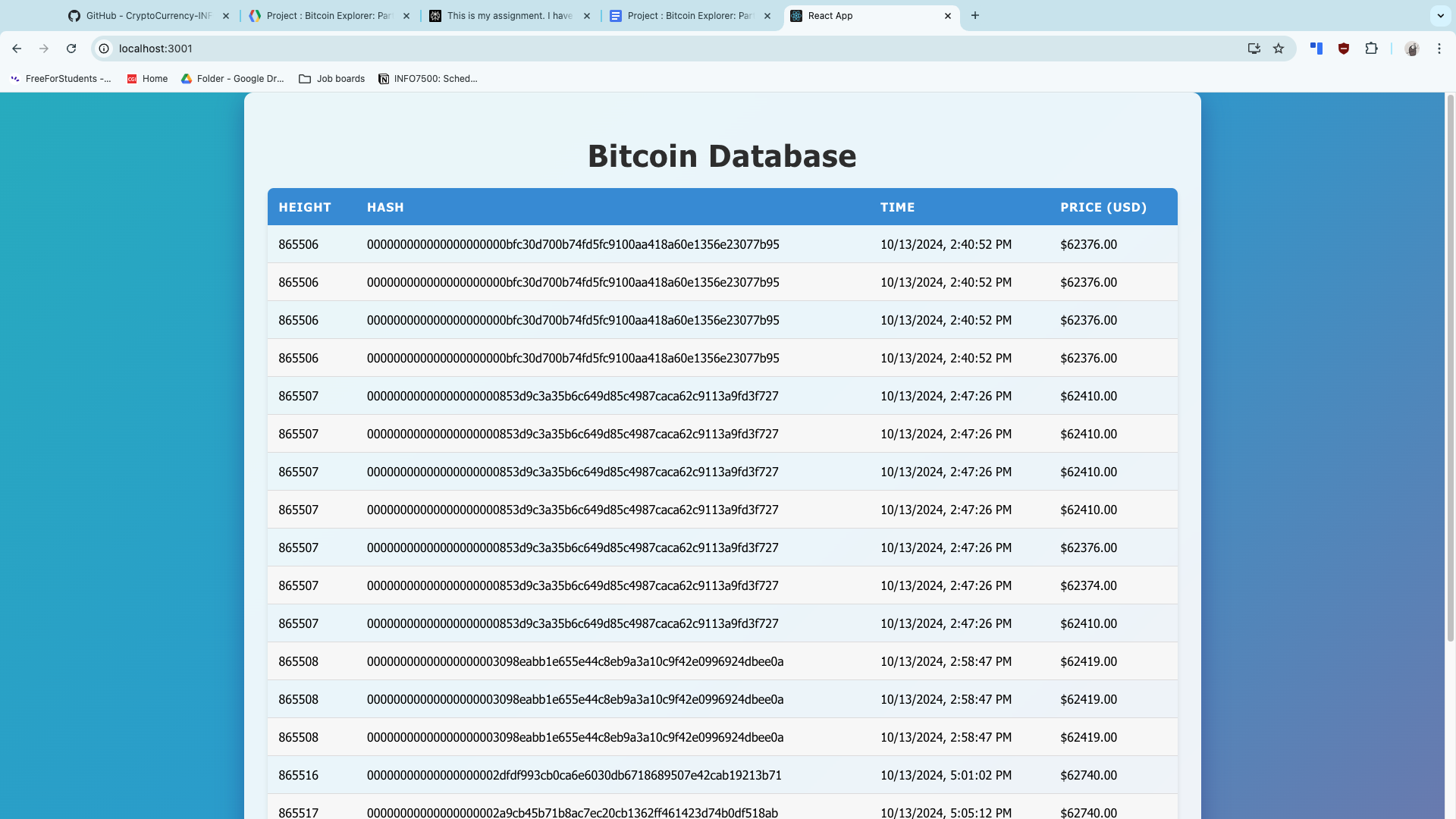
Backend:

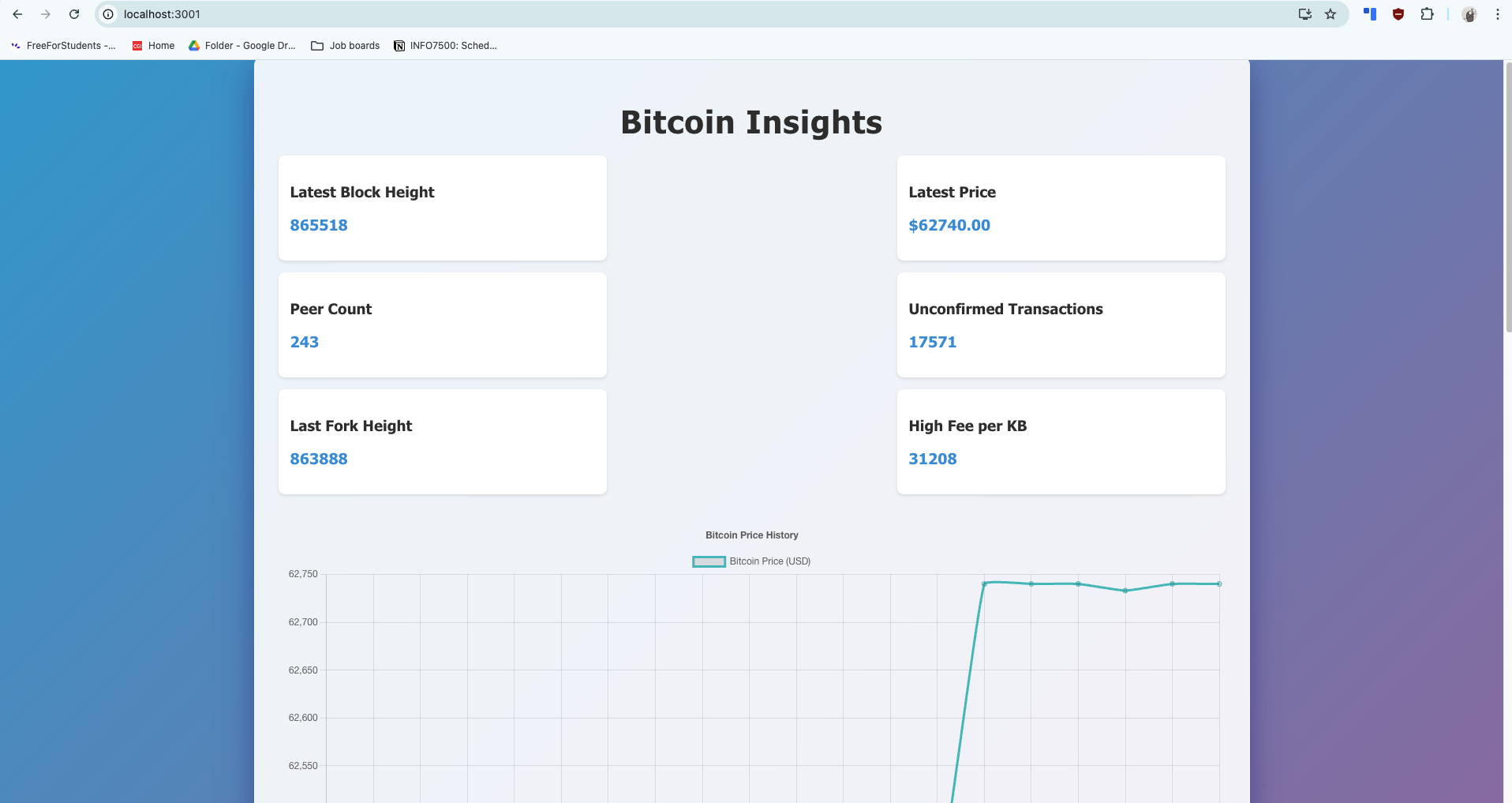


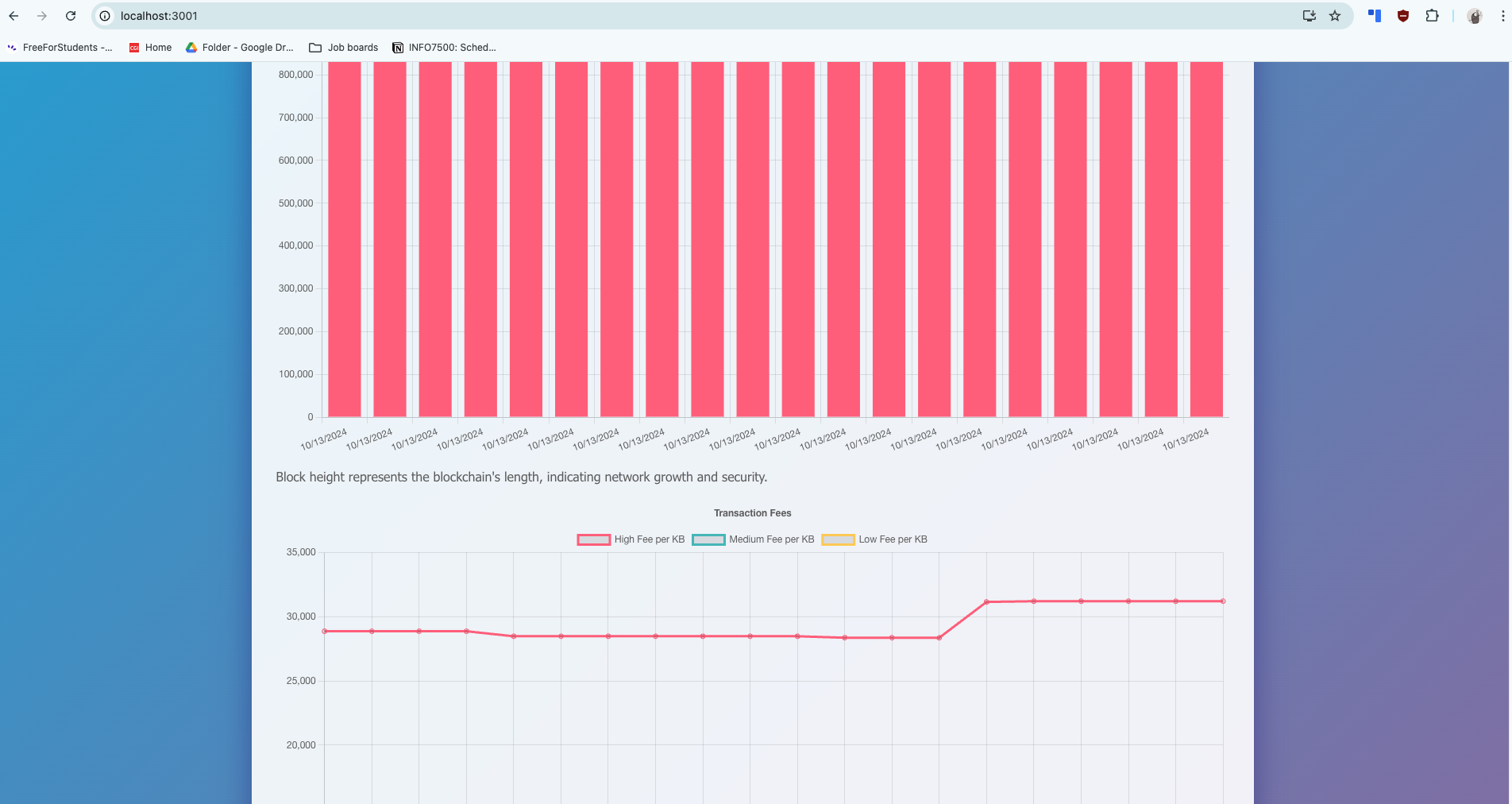


Frontend:









# References

## 

1. Bitcoin Developer Documentation: <https://developer.bitcoin.org/>
2. Rust Documentation: <https://doc.rust-lang.org/book/>
3. Node.js Documentation: <https://nodejs.org/en/docs/>
4. React Documentation: <https://reactjs.org/docs/getting-started.html>
5. Chart.js Documentation: <https://www.chartjs.org/docs/latest/>

### 