friedger fix: return root 76b4e9f · last month

200 lines (170 loc) · 6.79 KB

Code Blame Raw

```typescript
import { BitcoinRPC } from './rpc';
import { TxProofResult, BitcoinRPCConfig } from './types';
import { calculateWTXID, calculateMerkleRoot, getMerkleProof, hash256, MerkleProofStep, calcu

const DEBUG = process.env.DEBUG === 'true';

function debug(...args: any[]) {
  if (DEBUG) {
    console.log('\x1b[33m[PROOF]\x1b[0m', ...args);
  }
}

function extractBlockHeader(blockHex: string): string {
  return blockHex.substring(0, 160);
}

function getTxHashes(block: any): Buffer[] {
  debug('Getting tx hashes from:', block.tx);
  const hashes = block.tx.map((tx: any) => {
    const hash = Buffer.from(tx.txid, 'hex').reverse();
    debug('Converted hash:', tx.txid, 'to:', hash.toString('hex'));
    return hash;
  });
  return hashes;
}

function verifyMerkleProof(txid: string, proofSteps: MerkleProofStep[], merkleRoot: string):
  debug('\n=== Starting Merkle Proof Verification ===');
  debug('Transaction ID:', txid);
  debug('Expected Merkle Root:', merkleRoot);
  debug('Proof steps:', proofSteps.map(step => ({
    position: step.position,
    hash: step.data.toString('hex')
  })));

  // Convert txid to internal byte order (reverse)
  let currentHash = Buffer.from(txid, 'hex').reverse();
  debug('Starting hash (internal order):', currentHash.toString('hex'));

  for (const step of proofSteps) {
    debug('\nProof Step:');
    debug('Current hash:', currentHash.toString('hex'));
    debug('Proof element:', step.data.toString('hex'));
    debug('Position:', step.position);

    const combined = step.position === 'left' ?
      Buffer.concat([step.data, currentHash]) :
      Buffer.concat([currentHash, step.data]);

    debug('Concatenated:', combined.toString('hex'));
    currentHash = hash256(combined);
    debug('After hash256:', currentHash.toString('hex'));
  }

  // Convert final hash back to display order (reverse)
  const calculatedRoot = currentHash.reverse().toString('hex');
  debug('\n=== Final Results ===');
  debug('Calculated root:', calculatedRoot);
  debug('Expected root:', merkleRoot);

  if (calculatedRoot !== merkleRoot) {
    throw new Error(`Merkle proof verification failed: Expected ${merkleRoot} but got ${calcu
  }

  return true;
}

export async function bitcoinTxProof(
  txid: string,
  blockHeight: number,
  rpcConfig: BitcoinRPCConfig
): Promise<TxProofResult> {
  const rpc = new BitcoinRPC(rpcConfig);

  debug('Getting block hash for height:', blockHeight);
  const blockHash = await rpc.call('getblockhash', [blockHeight]);

  debug('Getting block data...');
  const block = await rpc.call('getblock', [blockHash, 2]);
  debug('Block merkle root:', block.merkleroot);
  debug('Block transactions:', JSON.stringify(block.tx, null, 2));

  const rawBlock = await rpc.call('getblock', [blockHash, 0]);
  const blockHeader = extractBlockHeader(rawBlock);

  // Get transaction hashes and create merkle proof
  const txHashes = getTxHashes(block);
  debug('Transaction hashes (internal order):', txHashes.map(h => h.toString('hex')));

  const txIndex = block.tx.findIndex((tx: any) => tx.txid === txid);
  debug('Transaction index:', txIndex);

  if (txIndex === -1) {
    throw new Error('Transaction not found in block');
  }

```

```
    let merkleProof: MerkleProofStep[] = [];
    let merkleProofHex = '';
    if (block.tx.length === 1) {
      debug('Single transaction block - no merkle proof needed');
    } else {
      debug('Calculating merkle proof...');
      merkleProof = getMerkleProof(txHashes, txIndex);
      debug('Merkle proof:', merkleProof.map(step => ({
        position: step.position,
        hash: step.data.toString('hex')
      })));

      if (!verifyMerkleProof(txid, merkleProof, block.merkleroot)) {
        throw new Error('Merkle proof verification failed');
      }
```

```
    let root = "";
    if (block.tx.length > 1 && txIndex !== 0) {
      debug('Calculating witness merkle data...');
      try {
        debug('Fetching raw transactions for witness merkle tree...');
        const txs = await Promise.all(block.tx.map(async (tx: any) => {
          const rawTxData = await rpc.call('getrawtransaction', [tx.txid, true, blockHash]);
          const hasWitness = rawTxData.hex.includes('0001') || rawTxData.vin.some((input: any)
          debug(`Transaction ${tx.txid}:`, {
            hasWitness,
            vinLength: rawTxData.vin.length,
            hasWitnessData: rawTxData.vin.some((input: any) => input.txinwitness)
          });
          return rawTxData.hex;
        }));

        debug('Calculating witness merkle proof...');
        const { proof, root: calculatedRoot } = calculateWitnessMerkleProof(txs, txIndex);
        root = calculatedRoot.toString('hex');
        debug('Witness merkle root:', root);
        debug('Witness proof steps:', proof.map(step => ({
          position: step.position,
          hash: step.data.toString('hex')
        })));

        if (proof.length === 0) {
```

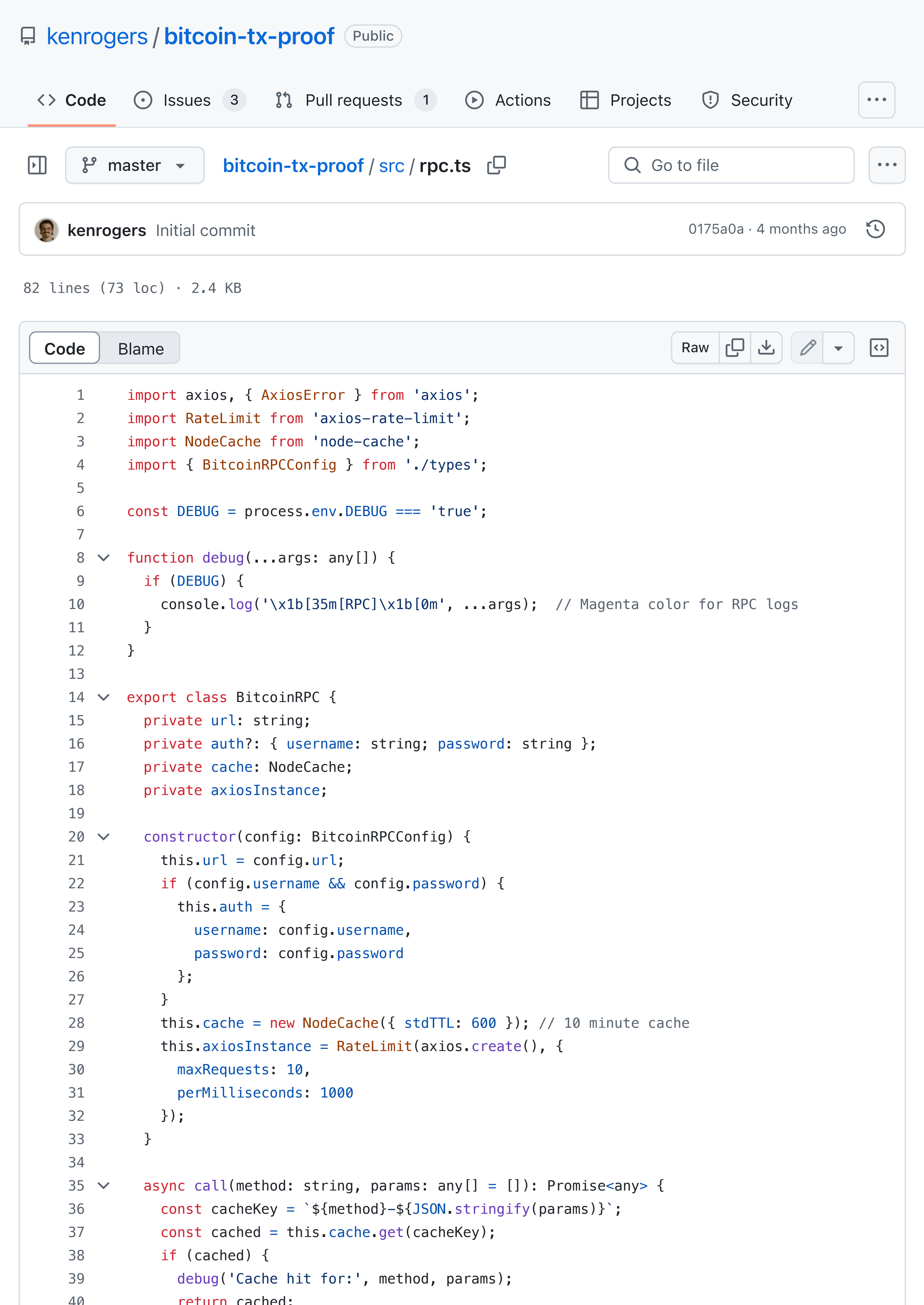
```
        debug('No witness proof needed (single tx or no witness data)');
        witnessMerkleProof = '';
      } else {
        // Convert proof to hex string
        witnessMerkleProof = proof
          .map(step => step.data.toString('hex'))
          .join('');
        witnessMerkleProofArray = proof.map(step => new Uint8Array(step.data));
        debug('Witness merkle proof hex:', witnessMerkleProof);
      }
    } catch (error) {
      debug('Error in witness calculations:', error);
      if (error instanceof Error) {
        debug('Error stack:', error.stack);
      }
    }
  } else {
    debug('Skipping witness merkle proof:',
      block.tx.length === 1 ? 'single-tx block' :
      txIndex === 0 ? 'coinbase transaction' :
      'unknown reason');
  }

  const result: TxProofResult = {
    blockHeight,
    transaction: rawTx.hex,
    blockHeader,
    txIndex,
    merkleProofDepth: Math.ceil(Math.log2(Math.max(block.tx.length, 2))),
    witnessMerkleRoot: root,
    witnessMerkleProof,
    witnessMerkleProofArray,
    witnessReservedValue: '0000000000000000000000000000000000000000000000000000000000000000',
    coinbaseTransaction: coinbaseTx.hex,
    coinbaseMerkleProof: merkleProofHex,
    coinbaseMerkleProofArray: merkleProof.map(step => new Uint8Array(step.data))
  };

  debug('Final result:', {
    ...result,
    transaction: result.transaction.substring(0, 64) + '...',
    coinbaseTransaction: result.coinbaseTransaction.substring(0, 64) + '...'
  });

  return result;
}

export { BitcoinRPCConfig, TxProofResult };
```

kenrogers Initial commit 0175a0a · 4 months ago

138 lines (115 loc) · 4.23 KB

Code Blame Raw

```typescript
import { Transaction } from 'bitcoinjs-lib';
import crypto from 'crypto';

const DEBUG = process.env.DEBUG === 'true';

function debug(...args: any[]) {
  if (DEBUG) {
    console.log('\x1b[34m[MERKLE]\x1b[0m', ...args);
  }
}

export function sha256(buffer: Buffer): Buffer {
  return crypto.createHash('sha256').update(buffer).digest();
}

export function hash256(buffer: Buffer): Buffer {
  return sha256(sha256(buffer));
}

export function calculateWTXID(txHex: string): Buffer {
  const tx = Transaction.fromHex(txHex);
  if (!tx.hasWitnesses()) {
    return Buffer.from(tx.getId(), 'hex').reverse();
  }
  return Buffer.from(tx.getHash(true).toString('hex'), 'hex').reverse();
}

export function calculateMerkleRoot(hashes: Buffer[]): Buffer {
  debug('Calculating merkle root for', hashes.length, 'hashes');
  if (hashes.length === 0) return Buffer.alloc(32, 0);
  if (hashes.length === 1) {
    debug('Single hash, returning:', hashes[0].toString('hex'));
    return hashes[0];
  }

  const newHashes: Buffer[] = [];
  for (let i = 0; i < hashes.length; i += 2) {
    const left = hashes[i];
    const right = i + 1 < hashes.length ? hashes[i + 1] : left;
    debug('Concatenating:', left.toString('hex'), '+', right.toString('hex'));
    debug('Concatenating:', left.toString('hex'), '+', right.toString('hex'));
    const combined = Buffer.concat([left, right]);
    const newHash = hash256(combined);
    debug('New hash:', newHash.toString('hex'));
    newHashes.push(newHash);
  }

  return calculateMerkleRoot(newHashes);
}

export interface MerkleProofStep {
  position: 'left' | 'right';
  data: Buffer;
}

export function getMerkleProof(hashes: Buffer[], index: number): MerkleProofStep[] {
  debug('Generating merkle proof for index', index, 'in', hashes.length, 'hashes');
  if (hashes.length === 0 || hashes.length === 1) {
    return [];
  }

  const proof: MerkleProofStep[] = [];
  let currentIndex = index;
  let currentLevel = [...hashes];

  while (currentLevel.length > 1) {
    debug('Current level size:', currentLevel.length, 'Current index:', currentIndex);
    const isRightNode = currentIndex % 2 === 1;
    const pairIndex = isRightNode ? currentIndex - 1 : currentIndex + 1;
    const position = isRightNode ? 'left' : 'right';

    if (pairIndex < currentLevel.length) {
      debug('Adding proof step:', position, currentLevel[pairIndex].toString('hex'));
      proof.push({
        position,
        data: currentLevel[pairIndex]
      });
    } else {
      // If there's no pair (odd number of nodes), duplicate the current node
      debug('Adding duplicate proof step:', position, currentLevel[currentIndex].toString('he
      proof.push({
        position,
        data: currentLevel[currentIndex]
      });
    }

    // Calculate next level
    const newLevel: Buffer[] = [];
    for (let i = 0; i < currentLevel.length; i += 2) {
      const left = currentLevel[i];
      const right = i + 1 < currentLevel.length ? currentLevel[i + 1] : left;
      const combined = Buffer.concat([left, right]);
      newLevel.push(hash256(combined));
    }

    currentLevel = newLevel;
    currentIndex = Math.floor(currentIndex / 2);
  }

  debug('Generated proof steps:', proof.map(step => ({
    position: step.position,
    hash: step.data.toString('hex')
  })));
  return proof;
}

export function verifyMerkleProof(txHash: Buffer, proof: MerkleProofStep[], root: Buffer): bo
  debug('Verifying merkle proof');
  debug('Starting hash:', txHash.toString('hex'));
  debug('Expected root:', root.toString('hex'));

  let currentHash = txHash;

  for (const step of proof) {
    debug('Proof step:', step.position, step.data.toString('hex'));
    const combined = step.position === 'left' ?
      Buffer.concat([step.data, currentHash]) :
      Buffer.concat([currentHash, step.data]);

    debug('Combined:', combined.toString('hex'));
    currentHash = hash256(combined);
    debug('After hash:', currentHash.toString('hex'));
  }

  debug('Final hash:', currentHash.toString('hex'));
  debug('Expected root:', root.toString('hex'));
  return currentHash.equals(root);
}

export function calculateWitnessMerkleProof(txs: string[], index: number): {
  proof: MerkleProofStep[],
  root: Buffer
} {
  const wtxids = txs.map(tx => calculateWTXID(tx));
  const proof = getMerkleProof(wtxids, index);
  const root = calculateMerkleRoot(wtxids);

  return { proof, root };
}
```



kenrogers / bitcoin-tx-proof Public

Code · Issues 3 · Pull requests 1 · Actions · Projects · Security

master · bitcoin-tx-proof / src / rpc.ts · Go to file

kenrogers Initial commit · 0175a0a · 4 months ago

82 lines (73 loc) · 2.4 KB

Code · Blame · Raw

```
import axios, { AxiosError } from 'axios';
import RateLimit from 'axios-rate-limit';
import NodeCache from 'node-cache';
import { BitcoinRPCConfig } from './types';

const DEBUG = process.env.DEBUG === 'true';

function debug(...args: any[]) {
  if (DEBUG) {
    console.log('\x1b[35m[RPC]\x1b[0m', ...args);  // Magenta color for RPC logs
  }
}

export class BitcoinRPC {
  private url: string;
  private auth?: { username: string; password: string };
  private cache: NodeCache;
  private axiosInstance;

  constructor(config: BitcoinRPCConfig) {
    this.url = config.url;
    if (config.username && config.password) {
      this.auth = {
        username: config.username,
        password: config.password
      };
    }
    this.cache = new NodeCache({ stdTTL: 600 }); // 10 minute cache
    this.axiosInstance = RateLimit(axios.create(), {
      maxRequests: 10,
      perMilliseconds: 1000
    });
  }

  async call(method: string, params: any[] = []): Promise<any> {
    const cacheKey = `${method}-${JSON.stringify(params)}`;
    const cached = this.cache.get(cacheKey);
    if (cached) {
      debug('Cache hit for:', method, params);
      return cached;
    }

    debug('Making RPC call:', method, params);
    try {
      const response = await this.axiosInstance.post(this.url, {
        jsonrpc: '2.0',
        id: 'bitcointxproof',
        method,
        params
      }, {
        auth: this.auth,
        headers: { 'Content-Type': 'application/json' }
      });

      debug('RPC response:', response.data);

      if (response.data.error) {
        throw new Error(`RPC Error: ${response.data.error.message}`);
      }

      this.cache.set(cacheKey, response.data.result);
      return response.data.result;
    } catch (error) {
      if (axios.isAxiosError(error)) {
        const axiosError = error as AxiosError;
        if (axiosError.response) {
          debug('RPC Error Response:', {
            status: axiosError.response.status,
            statusText: axiosError.response.statusText,
            data: axiosError.response.data
          });
          throw new Error(`RPC Error (${axiosError.response.status}): ${JSON.stringify(axiosE
        } else if (axiosError.request) {
          debug('RPC Request Error:', axiosError.message);
          throw new Error(`RPC Request Failed: ${axiosError.message}`);
        }
      }
      debug('Unexpected RPC Error:', error);
      throw error;
    }
  }
}
```

kenrogers / bitcoin-tx-proof Public

Code · Issues 3 · Pull requests 1 · Actions · Projects · Security

master · bitcoin-tx-proof / src / types.ts · Go to file

friedger feat: add test for clarity-bitcoin-v5 (WIP) 5f4346f · last month

20 lines (19 loc) · 509 Bytes

Code · Blame · Raw

```ts
export interface BitcoinRPCConfig {
  url: string;
  username?: string;
  password?: string;
}

export interface TxProofResult {
  blockHeight: number;
  transaction: string;
  blockHeader: string;
  txIndex: number;
  merkleProofDepth: number;
  witnessMerkleRoot: string;
  witnessMerkleProof: string;
  witnessMerkleProofArray: Uint8Array<ArrayBuffer>[];
  witnessReservedValue: string;
  coinbaseTransaction: string;
  coinbaseMerkleProof: string;
  coinbaseMerkleProofArray: Uint8Array<ArrayBuffer>[];
}
```