# mini-chain-docker — a tiny PoW blockchain in Docker Compose

This is a **minimal educational** blockchain with proof‑of‑work mining, signed transactions, 2 Dockerized nodes, simple peer discovery, and on-disk persistence. Not for real money.

## Project layout

mini-chain-docker/  
├─ docker-compose.yml  
├─ Dockerfile  
├─ requirements.txt  
├─ app.py  
├─ crypto\_utils.py  
├─ gen\_keys.py  
└─ README.md

## docker-compose.yml

version: "3.9"  
services:  
 node1:  
 build: .  
 container\_name: mini-chain-node1  
 environment:  
 - NODE\_NAME=node1  
 - PORT=5000  
 - DIFFICULTY\_BITS=18  
 - PEERS=http://node2:5000  
 volumes:  
 - ./data/node1:/data  
 ports:  
 - "5001:5000" # host:container  
 restart: unless-stopped  
  
 node2:  
 build: .  
 container\_name: mini-chain-node2  
 environment:  
 - NODE\_NAME=node2  
 - PORT=5000  
 - DIFFICULTY\_BITS=18  
 - PEERS=http://node1:5000  
 volumes:  
 - ./data/node2:/data  
 ports:  
 - "5002:5000"  
 restart: unless-stopped

## Dockerfile

FROM python:3.11-slim  
WORKDIR /app  
COPY requirements.txt ./  
RUN pip install --no-cache-dir -r requirements.txt  
COPY . .  
# Persist data (chain, keys)  
VOLUME ["/data"]  
ENV PYTHONUNBUFFERED=1  
CMD ["python", "app.py"]

## requirements.txt

Flask==3.0.3  
ecdsa==0.19.0  
pycryptodome==3.21.0  
requests==2.32.3

## crypto\_utils.py

import os, json, binascii, hashlib, time  
from ecdsa import SigningKey, VerifyingKey, SECP256k1  
from Crypto.Cipher import AES  
from Crypto.Random import get\_random\_bytes  
  
KEYS\_PATH = "/data/miner\_keys.json"  
  
# ---------- Key utilities ----------  
  
def ensure\_miner\_keys():  
 os.makedirs("/data", exist\_ok=True)  
 if os.path.exists(KEYS\_PATH):  
 with open(KEYS\_PATH, "r") as f:  
 obj = json.load(f)  
 return obj["priv"], obj["pub"]  
 sk = SigningKey.generate(curve=SECP256k1)  
 vk = sk.get\_verifying\_key()  
 priv = binascii.hexlify(sk.to\_string()).decode()  
 pub = binascii.hexlify(vk.to\_string()).decode()  
 with open(KEYS\_PATH, "w") as f:  
 json.dump({"priv": priv, "pub": pub, "created": time.time()}, f)  
 return priv, pub  
  
  
def sk\_from\_hex(priv\_hex):  
 return SigningKey.from\_string(binascii.unhexlify(priv\_hex), curve=SECP256k1)  
  
  
def vk\_from\_hex(pub\_hex):  
 return VerifyingKey.from\_string(binascii.unhexlify(pub\_hex), curve=SECP256k1)  
  
  
def sign(priv\_hex, msg\_bytes: bytes) -> str:  
 sk = sk\_from\_hex(priv\_hex)  
 sig = sk.sign(msg\_bytes, hashfunc=hashlib.sha256)  
 return binascii.hexlify(sig).decode()  
  
  
def verify(pub\_hex, msg\_bytes: bytes, sig\_hex: str) -> bool:  
 try:  
 vk = vk\_from\_hex(pub\_hex)  
 return vk.verify(binascii.unhexlify(sig\_hex), msg\_bytes, hashfunc=hashlib.sha256)  
 except Exception:  
 return False  
  
# ---------- Simple AES-GCM (demo only) ----------  
  
def ecdh\_shared(priv\_hex, other\_pub\_hex) -> bytes:  
 # Not constant-time; demo only.  
 sk = sk\_from\_hex(priv\_hex)  
 vk\_other = vk\_from\_hex(other\_pub\_hex)  
 # Fake ECDH using ECDSA key material for demo: derive by hashing pub||priv  
 # For real ECDH use X25519 or proper ECDH API.  
 h = hashlib.sha256(sk.to\_string() + vk\_other.to\_string()).digest()  
 return h  
  
  
def aes\_gcm\_encrypt(key: bytes, plaintext: bytes, aad: bytes = b""):  
 key = hashlib.sha256(key).digest() # derive 32 bytes  
 nonce = get\_random\_bytes(12)  
 cipher = AES.new(key, AES.MODE\_GCM, nonce=nonce)  
 cipher.update(aad)  
 ct, tag = cipher.encrypt\_and\_digest(plaintext)  
 return {"nonce": nonce.hex(), "ct": ct.hex(), "tag": tag.hex()}  
  
  
def aes\_gcm\_decrypt(key: bytes, pkt, aad: bytes = b"") -> bytes:  
 key = hashlib.sha256(key).digest()  
 cipher = AES.new(key, AES.MODE\_GCM, nonce=bytes.fromhex(pkt["nonce"]))  
 cipher.update(aad)  
 return cipher.decrypt\_and\_verify(bytes.fromhex(pkt["ct"]), bytes.fromhex(pkt["tag"]))  
  
# ---------- Hash helpers ----------  
  
def sha256\_hex(x: bytes) -> str:  
 return hashlib.sha256(x).hexdigest()  
  
  
def tx\_digest(tx: dict) -> str:  
 # Deterministic digest excluding signature  
 body = {k: tx[k] for k in sorted(tx.keys()) if k != "signature"}  
 return sha256\_hex(json.dumps(body, separators=(",",":"), sort\_keys=True).encode())

## app.py

import os, json, time, hashlib, requests  
from flask import Flask, request, jsonify  
from crypto\_utils import ensure\_miner\_keys, sign, verify, tx\_digest  
  
DATA\_DIR = "/data"  
CHAIN\_PATH = f"{DATA\_DIR}/chain.json"  
PENDING\_PATH = f"{DATA\_DIR}/pending.json"  
PEERS\_PATH = f"{DATA\_DIR}/peers.json"  
  
DIFFICULTY\_BITS = int(os.getenv("DIFFICULTY\_BITS", "18"))  
TARGET = 2 \*\* (256 - DIFFICULTY\_BITS)  
PORT = int(os.getenv("PORT", "5000"))  
NODE\_NAME = os.getenv("NODE\_NAME", "node")  
  
os.makedirs(DATA\_DIR, exist\_ok=True)  
  
# -------------- Persistence --------------  
  
def load\_json(path, default):  
 try:  
 with open(path, "r") as f:  
 return json.load(f)  
 except Exception:  
 return default  
  
  
def save\_json(path, obj):  
 tmp = path + ".tmp"  
 with open(tmp, "w") as f:  
 json.dump(obj, f)  
 os.replace(tmp, path)  
  
chain = load\_json(CHAIN\_PATH, [])  
if not chain:  
 # Genesis block  
 chain = [{  
 "index": 0,  
 "timestamp": time.time(),  
 "prev\_hash": "0" \* 64,  
 "nonce": 0,  
 "transactions": []  
 }]  
 save\_json(CHAIN\_PATH, chain)  
  
pending = load\_json(PENDING\_PATH, [])  
peers = set(load\_json(PEERS\_PATH, []))  
for p in os.getenv("PEERS", "").split(","):  
 p = p.strip()  
 if p:  
 peers.add(p)  
save\_json(PEERS\_PATH, list(peers))  
  
MINER\_PRIV, MINER\_PUB = ensure\_miner\_keys()  
  
# -------------- Core functions --------------  
  
def block\_header(block):  
 body = {  
 "index": block["index"],  
 "prev\_hash": block["prev\_hash"],  
 "tx\_hashes": [hashlib.sha256(json.dumps({k: t[k] for k in t if k != 'signature'}, sort\_keys=True).encode()).hexdigest() for t in block["transactions"]],  
 "timestamp": block["timestamp"],  
 "nonce": block["nonce"],  
 }  
 return json.dumps(body, separators=(",",":"), sort\_keys=True).encode()  
  
  
def block\_hash(block):  
 return hashlib.sha256(block\_header(block)).hexdigest()  
  
  
def last\_hash():  
 return block\_hash(chain[-1])  
  
  
def valid\_pow(block):  
 return int(block\_hash(block), 16) < TARGET  
  
  
def balance\_of(pub):  
 bal = 0  
 for b in chain:  
 for t in b.get("transactions", []):  
 if t.get("sender") == "0" and t.get("recipient") == pub:  
 bal += t.get("amount", 0)  
 elif t.get("sender") == pub:  
 bal -= t.get("amount", 0)  
 elif t.get("recipient") == pub:  
 bal += t.get("amount", 0)  
 return bal  
  
  
def verify\_tx(tx):  
 required = {"sender","recipient","amount","nonce","timestamp","signature"}  
 if not required.issubset(tx.keys()):  
 return False, "missing fields"  
 if tx["sender"] != "0":  
 if not verify(tx["sender"], tx\_digest(tx).encode(), tx["signature"]):  
 return False, "bad signature"  
 if balance\_of(tx["sender"]) < tx["amount"]:  
 return False, "insufficient funds"  
 return True, "ok"  
  
  
def mine\_block():  
 global chain, pending  
 coinbase = {"sender": "0", "recipient": MINER\_PUB, "amount": 50, "nonce": 0, "timestamp": time.time(), "signature": ""}  
 txs = [coinbase] + pending  
 new\_block = {  
 "index": len(chain),  
 "timestamp": time.time(),  
 "prev\_hash": last\_hash(),  
 "nonce": 0,  
 "transactions": txs,  
 }  
 nonce = 0  
 while True:  
 new\_block["nonce"] = nonce  
 if valid\_pow(new\_block):  
 chain.append(new\_block)  
 save\_json(CHAIN\_PATH, chain)  
 pending = []  
 save\_json(PENDING\_PATH, pending)  
 return new\_block  
 nonce += 1  
  
  
def valid\_chain(c):  
 if not c or c[0]["prev\_hash"] != "0" \* 64:  
 return False  
 for i in range(1, len(c)):  
 prev = c[i-1]  
 cur = c[i]  
 if cur["prev\_hash"] != hashlib.sha256(json.dumps({  
 "index": prev["index"],  
 "prev\_hash": prev["prev\_hash"],  
 "tx\_hashes": [hashlib.sha256(json.dumps({k: t[k] for k in t if k != 'signature'}, sort\_keys=True).encode()).hexdigest() for t in prev["transactions"]],  
 "timestamp": prev["timestamp"],  
 "nonce": prev["nonce"],  
 }, separators=(",",":"), sort\_keys=True).encode()).hexdigest():  
 return False  
 if int(hashlib.sha256(json.dumps({  
 "index": cur["index"],  
 "prev\_hash": cur["prev\_hash"],  
 "tx\_hashes": [hashlib.sha256(json.dumps({k: t[k] for k in t if k != 'signature'}, sort\_keys=True).encode()).hexdigest() for t in cur["transactions"]],  
 "timestamp": cur["timestamp"],  
 "nonce": cur["nonce"],  
 }, separators=(",",":"), sort\_keys=True).encode()).hexdigest(), 16) >= TARGET:  
 return False  
 return True  
  
# -------------- Flask API --------------  
app = Flask(\_\_name\_\_)  
  
@app.get("/")  
def root():  
 return {"node": NODE\_NAME, "port": PORT, "peers": list(peers), "height": len(chain)-1, "miner\_pub": MINER\_PUB}  
  
@app.get("/chain")  
def get\_chain():  
 return jsonify({"length": len(chain), "chain": chain})  
  
@app.get("/pending")  
def get\_pending():  
 return jsonify(pending)  
  
@app.get("/address/<pub>/balance")  
def get\_balance(pub):  
 return {"address": pub, "balance": balance\_of(pub)}  
  
@app.post("/tx/new")  
def new\_tx():  
 tx = request.get\_json(force=True)  
 ok, reason = verify\_tx(tx)  
 if not ok:  
 return {"ok": False, "reason": reason}, 400  
 pending.append(tx)  
 save\_json(PENDING\_PATH, pending)  
 # broadcast to peers  
 for p in peers:  
 try:  
 requests.post(f"{p}/tx/ingest", json=tx, timeout=2)  
 except Exception:  
 pass  
 return {"ok": True, "queued": True}  
  
@app.post("/tx/ingest")  
def ingest\_tx():  
 tx = request.get\_json(force=True)  
 ok, \_ = verify\_tx(tx)  
 if ok and tx not in pending:  
 pending.append(tx)  
 save\_json(PENDING\_PATH, pending)  
 return {"ok": True}  
  
@app.get("/mine")  
def mine():  
 blk = mine\_block()  
 # broadcast new chain head  
 head = {"length": len(chain), "chain": chain}  
 for p in peers:  
 try:  
 requests.post(f"{p}/nodes/resolve", json=head, timeout=2)  
 except Exception:  
 pass  
 return jsonify({"mined\_index": blk["index"], "hash": hashlib.sha256(json.dumps(blk, sort\_keys=True).encode()).hexdigest()})  
  
@app.post("/nodes/register")  
def register\_nodes():  
 body = request.get\_json(force=True)  
 new\_peers = body.get("peers", [])  
 added = 0  
 for p in new\_peers:  
 if p not in peers:  
 peers.add(p)  
 added += 1  
 save\_json(PEERS\_PATH, list(peers))  
 return {"ok": True, "added": added, "peers": list(peers)}  
  
@app.post("/nodes/resolve")  
def resolve():  
 body = request.get\_json(force=True)  
 incoming = body.get("chain")  
 if incoming and len(incoming) > len(chain) and valid\_chain(incoming):  
 # replace chain  
 global chain  
 chain = incoming  
 save\_json(CHAIN\_PATH, chain)  
 return {"length": len(chain)}  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 app.run(host="0.0.0.0", port=PORT)

## gen\_keys.py (demo keypair generator)

from crypto\_utils import ensure\_miner\_keys  
priv, pub = ensure\_miner\_keys()  
print("Miner keypair (stored in /data/miner\_keys.json):")  
print("PUBLIC:", pub)  
print("PRIVATE:", priv)  
print("NOTE: For demo only. Do not use for real funds.")

## README.md

# mini-chain-docker  
  
> Minimal educational blockchain with PoW, signed txs, and two Dockerized nodes.  
  
## Quick start  
  
```bash  
# 1) clone & enter  
# git clone <this-folder> && cd mini-chain-docker  
  
# 2) build images  
docker compose build  
  
# 3) start two nodes  
docker compose up -d  
  
# 4) (optional) show the auto-generated miner key for each node  
docker compose exec node1 python gen\_keys.py  
docker compose exec node2 python gen\_keys.py  
  
# 5) mine a block on node1  
curl http://localhost:5001/mine | jq  
  
# 6) view chain and balances  
curl http://localhost:5001/chain | jq  
curl http://localhost:5001/address/PUT\_PUBLIC\_KEY\_HERE/balance | jq  
  
# 7) create and submit a signed transaction  
# Generate a separate sender keypair OFFLINE or via node1 for demo:  
SENDER\_PUB=$(docker compose exec -T node1 python - <<'PY'  
from crypto\_utils import ensure\_miner\_keys  
\_, pub = ensure\_miner\_keys(); print(pub)  
PY  
)  
  
# NOTE: Use your OWN key management in real use. For demo we'll reuse node1's miner key as the sender.  
# Build a tx JSON and sign digest locally inside the container, then POST it.  
TX=$(docker compose exec -T node1 python - <<'PY'  
import json, time  
from crypto\_utils import ensure\_miner\_keys, sign, tx\_digest  
priv, pub = ensure\_miner\_keys()  
recipient = pub # self-transfer for demo  
amount = 1  
nonce = 1  
 tx = {"sender": pub, "recipient": recipient, "amount": amount, "nonce": nonce, "timestamp": time.time()}  
 d = tx\_digest(tx)  
 tx["signature"] = sign(priv, d.encode())  
print(json.dumps(tx))  
PY  
)  
  
curl -s -X POST http://localhost:5001/tx/new -H 'Content-Type: application/json' -d "$TX" | jq  
curl -s http://localhost:5001/mine | jq  
curl -s http://localhost:5001/address/$SENDER\_PUB/balance | jq  
  
# 8) peer sync  
# Node1 exposes 5001, node2 exposes 5002. They already know each other via PEERS env.  
# You can also register additional peers:  
curl -s -X POST http://localhost:5001/nodes/register -H 'Content-Type: application/json' \  
 -d '{"peers":["http://node2:5000"]}' | jq  
  
# 9) stop  
docker compose down

## Endpoints

* GET / — node info
* GET /chain — full chain
* GET /pending — pending txs
* GET /address/<pub>/balance — naive balance
* POST /tx/new — submit signed tx {sender,recipient,amount,nonce,timestamp,signature}
* GET /mine — mine one block with coinbase → node’s miner key
* POST /nodes/register — add peers
* POST /nodes/resolve — naive longest-chain acceptance

## Notes & limits

* Educational only. No mempool rules, no fees, no UTXO set, and naive consensus.
* Difficulty is adjustable via DIFFICULTY\_BITS. Lower = easier.
* Keys are auto-created per node and stored in /data/miner\_keys.json.
* Replace the demo ECDH with **X25519** or proper ECDH in real systems.
* Never expose real private keys. Use hardware wallets or KMS. ```