ChainStorm: A blockchain prototype built using Scala and Akka

•••

By :-
Parv Patel

What is ChainStorm?

ChainStorm is a blockchain technology that ensures secure, tamper-proof, and decentralized data storage. Unlike traditional blockchain implementations that rely on synchronous and resource-intensive mechanisms, ChainStorm improves efficiency by using actor-based concurrency with Akka, enabling asynchronous communication between nodes, miners, and clients. This approach enhances scalability by efficiently managing multiple tasks in parallel, reducing delays, and improving transaction throughput.

Use Cases

- **Tokenized Asset Management** 🎟️

Use Case: Convert real-world assets (land, stocks, art) into digital tokens

- **Secure Financial Transactions** 💰

Use Case: Peer-to-peer (P2P) payments and microtransactions

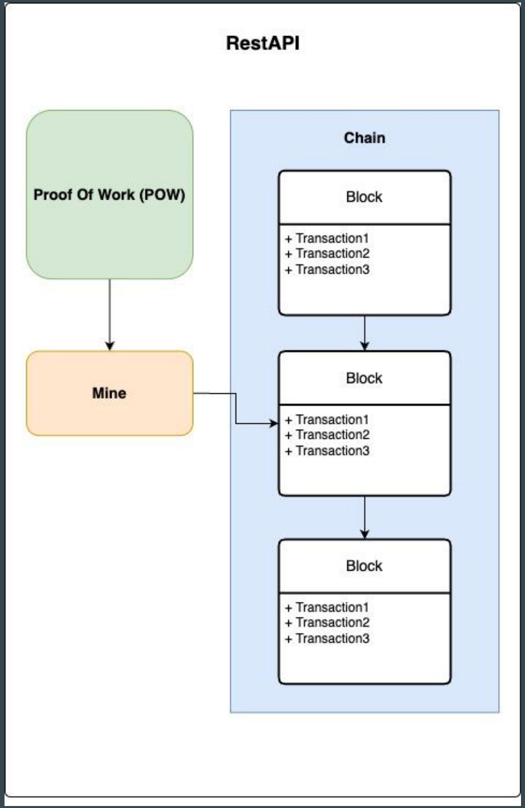
- **Decentralized Voting System** 🗳️

Use Case: Secure, transparent, and tamper-proof voting

Methodology - Blockchain Data Structure

- **Implement a Block case class containing:**
  - index: Position in the chain.
  - previousHash: Ensures chain integrity.
  - timestamp: Creation time.
  - data: Transactions or information stored.
  - hash: Unique fingerprint of the block.

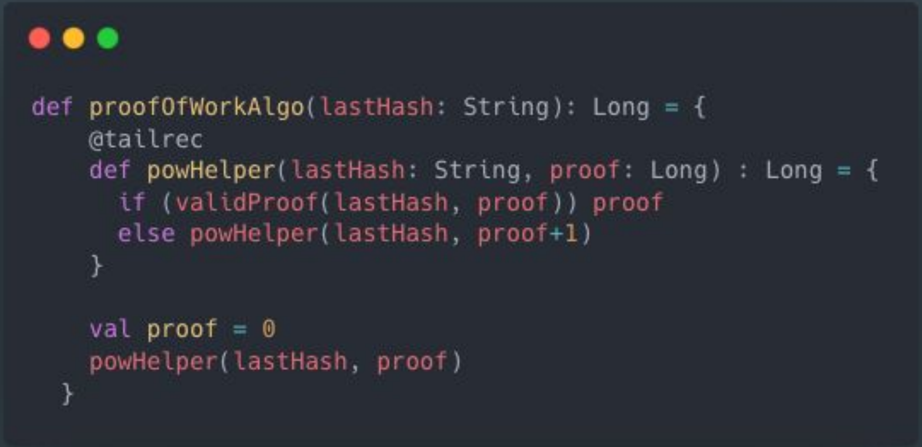
- **Use SHA-256 hashing to generate block hashes.**

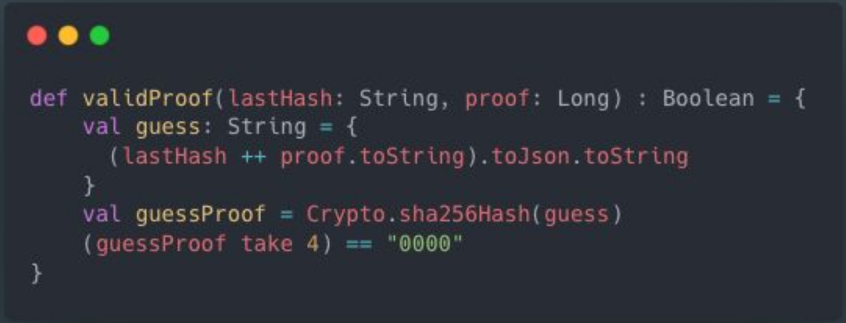


Methodology - Actors

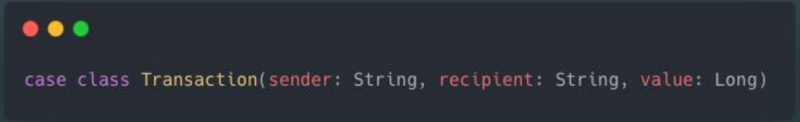
- **Blockchain Actor:** Stores, Validates and listens for new blocks.
- **Node Actor:** Simulates multiple nodes maintaining copies of the blockchain. Broadcasts and Synchronizes chain state with other nodes.
- **User/Client Actor:** Sends new transactions to the miner.
- **Miner Actor:** Mines new block for our blockchain.

Proof Of Work Algorithm

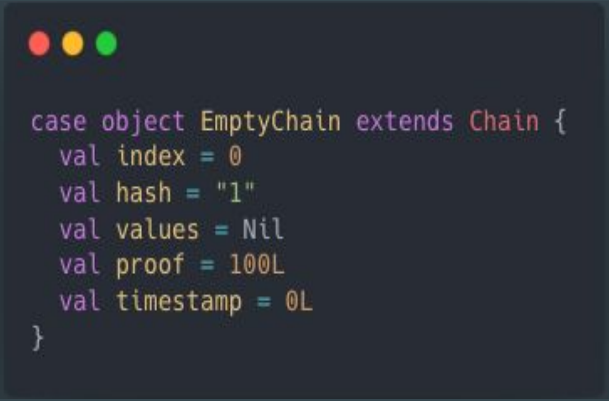

```scala
def proofOfWorkAlgo(lastHash: String): Long = {
    @tailrec
    def powHelper(lastHash: String, proof: Long) : Long = {
      if (validProof(lastHash, proof)) proof
      else powHelper(lastHash, proof+1)
    }

    val proof = 0
    powHelper(lastHash, proof)
  }
```


```scala
def validProof(lastHash: String, proof: Long) : Boolean = {
    val guess: String = {
      (lastHash ++ proof.toString).toJson.toString
    }
    val guessProof = Crypto.sha256Hash(guess)
    (guessProof take 4) == "0000"
}
```

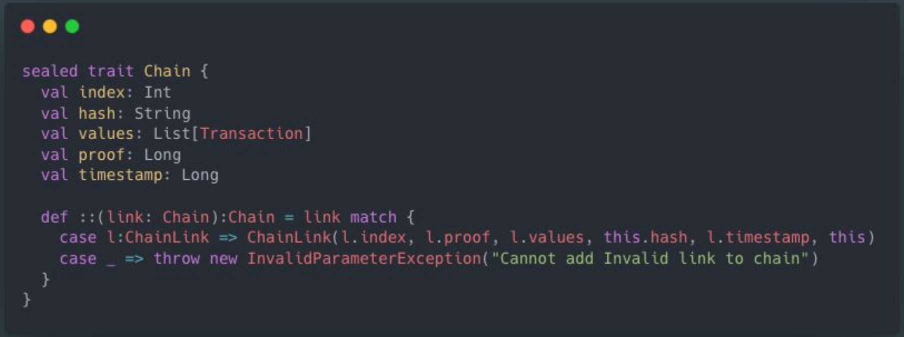
Transaction & Chain



```scala
case class Transaction(sender: String, recipient: String, value: Long)
```

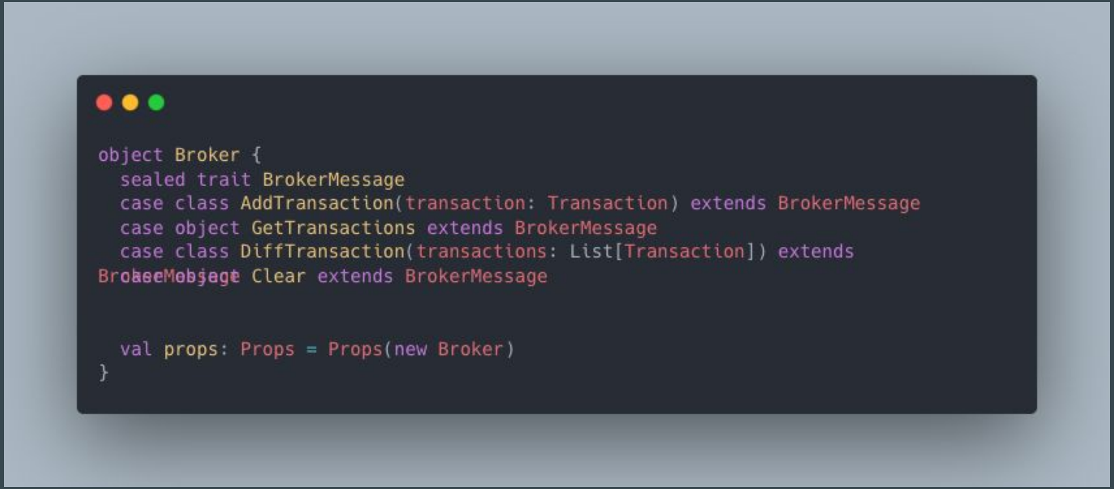


```scala
case object EmptyChain extends Chain {
  val index = 0
  val hash = "1"
  val values = Nil
  val proof = 100L
  val timestamp = 0L
}
```



```scala
sealed trait Chain {
  val index: Int
  val hash: String
  val values: List[Transaction]
  val proof: Long
  val timestamp: Long

  def ::(link: Chain):Chain = link match {
    case l:ChainLink => ChainLink(l.index, l.proof, l.values, this.hash, l.timestamp, this)
    case _ => throw new InvalidParameterException("Cannot add Invalid link to chain")
  }
}
```

Broker Actor

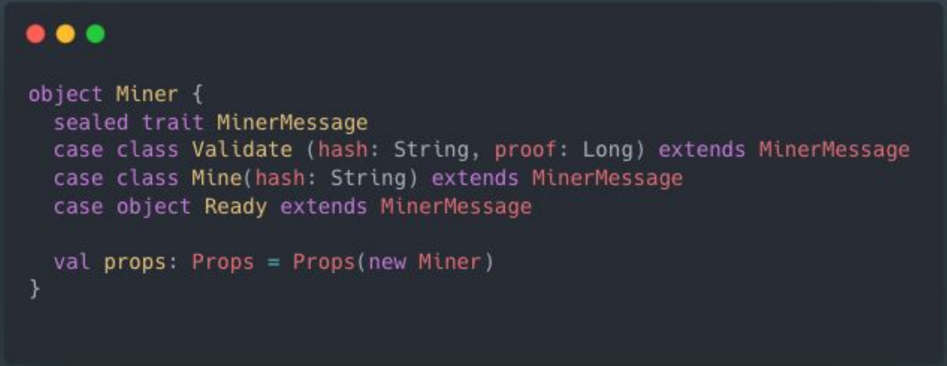


```
object Broker {
  sealed trait BrokerMessage
  case class AddTransaction(transaction: Transaction) extends BrokerMessage
  case object GetTransactions extends BrokerMessage
  case class DiffTransaction(transactions: List[Transaction]) extends
BrokerMessage
  case object Clear extends BrokerMessage


  val props: Props = Props(new Broker)
}
```

The **Broker** in Chainstorm acts as a transaction manager. It's responsible for **temporarily storing transactions** , helping nodes **synchronize transaction pools** , and **clearing them** after mining.

- **AddTransaction:** To add Transaction to the chain.
- **GetTransaction:** Fetches the pending Transaction.
- **Clear:** Clears the pending Transaction List.

Miner Actor

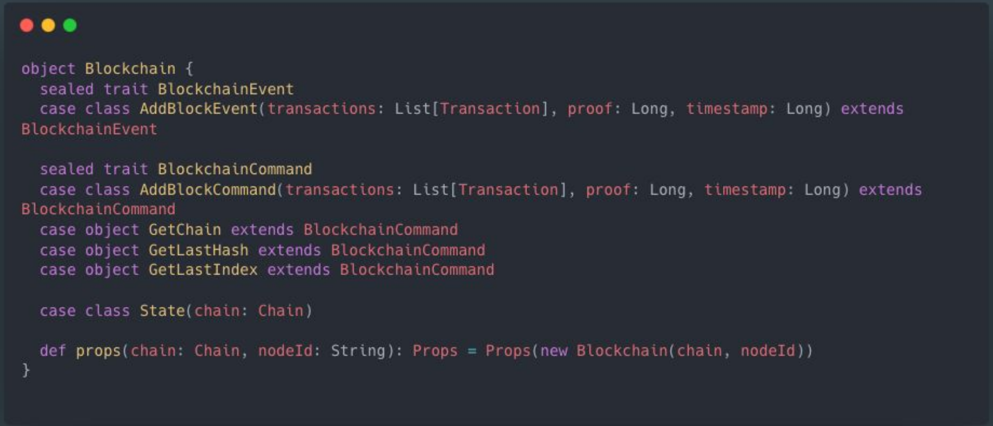


```scala
object Miner {
  sealed trait MinerMessage
  case class Validate (hash: String, proof: Long) extends MinerMessage
  case class Mine(hash: String) extends MinerMessage
  case object Ready extends MinerMessage

  val props: Props = Props(new Miner)
}
```

the **Miner actor** is responsible for simulating the mining process. It receives hashes to mine, validates proofs, and signals when it's ready to mine again.

- **Validate:** A message sent to check if the given proof-of-work is valid for the hash.
- **Mine:** Tells the miner to start the mining process for a given hash.
- **Ready:** Indicates that the miner is idle and ready to mine another block to coordinate with Node Actor.

Blockchain Actor

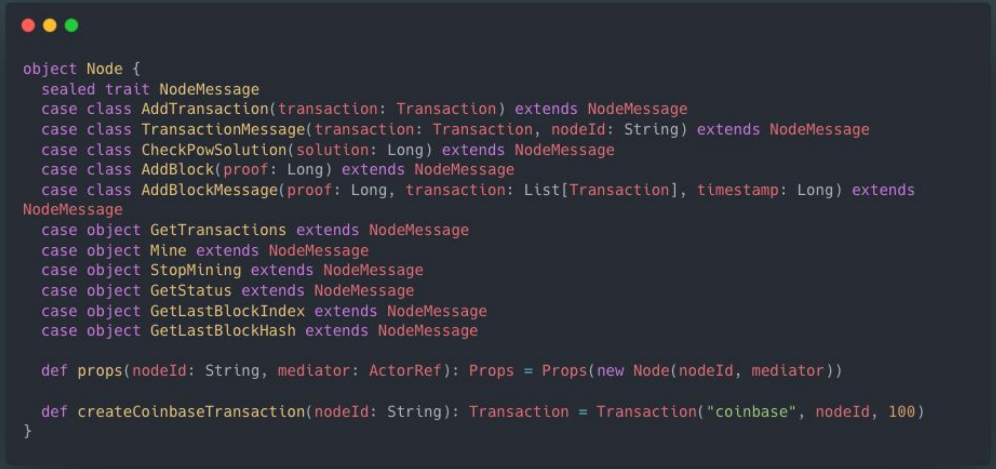


```
object Blockchain {
  sealed trait BlockchainEvent
  case class AddBlockEvent(transactions: List[Transaction], proof: Long, timestamp: Long) extends
BlockchainEvent

  sealed trait BlockchainCommand
  case class AddBlockCommand(transactions: List[Transaction], proof: Long, timestamp: Long) extends
BlockchainCommand
  case object GetChain extends BlockchainCommand
  case object GetLastHash extends BlockchainCommand
  case object GetLastIndex extends BlockchainCommand

  case class State(chain: Chain)

  def props(chain: Chain, nodeId: String): Props = Props(new Blockchain(chain, nodeId))
}
```

The Blockchain actor is the **heart of Chainstorm** . It stores the current chain, handles commands like adding blocks, and responds to queries about the chain's state.

- **AddBlockEvent:** Defines events that are used for persisting or logging state changes.
- **AddBlockCommand:** Command sent when a new block is mined and should be added to the chain.
- **GetChain:** Returns the entire blockchain.
- **GetLastHash:** Returns the hash of the last block.
- **GetLastIndex:** Returns the index/height of the last block.

Node Actor



```
object Node {
  sealed trait NodeMessage
  case class AddTransaction(transaction: Transaction) extends NodeMessage
  case class TransactionMessage(transaction: Transaction, nodeId: String) extends NodeMessage
  case class CheckPowSolution(solution: Long) extends NodeMessage
  case class AddBlock(proof: Long) extends NodeMessage
  case class AddBlockMessage(proof: Long, transaction: List[Transaction], timestamp: Long) extends
NodeMessage
  case object GetTransactions extends NodeMessage
  case object Mine extends NodeMessage
  case object StopMining extends NodeMessage
  case object GetStatus extends NodeMessage
  case object GetLastBlockIndex extends NodeMessage
  case object GetLastBlockHash extends NodeMessage

  def props(nodeId: String, mediator: ActorRef): Props = Props(new Node(nodeId, mediator))

  def createCoinbaseTransaction(nodeId: String): Transaction = Transaction("coinbase", nodeId, 100)
}
```

the **Node actor** serves as the **main controller** for handling transactions, mining requests, and communication with the Broker, Miner, and Blockchain actors.

- **AddTransaction:** Add a new transaction to the node's local transaction pool.
- **AddBlock:** Command to add a new block once a valid proof has been found.
- **GetTransaction:** Indicates that the miner is idle and ready to mine another block to coordinate with Node Actor.
- **Mine | StopMining:** Tells the node to start or stop mining.
- **CheckPowSolution:** Used to check if the given proof is valid (sent from the miner).

Milestones/sprints

Week 1(18 March - 22 March): Implement Blockchain Data Structure

Week 2 (23 March - 29 March): Develop Blockchain Actor Model

Week 3 (30 March - 4 April): Implement Miner & Proof-of-Work

Week 4 (6th April - 12 April): Implement Node & Client Actors

Week 5 (13 April - 19 April): Testing & Debugging

Week 6: Deployment

What will you program in Scala?

Scala: Blockchain model and Actors

Network (RestAPI): Akka-HTTP in Scala

Model Representation (Front-end) : Web API in from Server

Goals of the project

- Fully functional Blockchain model. (satisfied)
- Actor based Concurrency (Satisfied)
- Decentralized Network (Partially)
- Blocks are mined, validated, and securely stored.

Acceptance Criteria

- ✅ Transactions can be created, added, and propagated to nodes
- ✅ Broker actor maintains an accurate transaction pool
- ✅ Miner actor can compute a valid proof of work
- ✅ Blockchain actor can add new blocks with valid proofs
- ✅ Nodes can request and mine new blocks

Thank You