Take Home Assessment:

Expected Completion Time: 4-6 hours
Submission Format: Github project link
Expected Language: Scala

High Level Description:

Create a self-contained project that implements a single-node blockchain solution for conventional currency-style transactions. This should consist of an RPC or REST API which can accept a ‘transaction’ denoting a value transfer between two accounts. The transaction must be cryptographically signed by the owner of the transaction’s source account, and should update an internal ledger corresponding to all balances associated with this currency. This should be roughly equivalent to a conventional banking transfer, consisting of a source, destination, and amount of currency being transferred.

This single-node blockchain service, upon receiving transactions, should attempt to bundle or group them together into ‘blocks’. Where a block is defined as a group of transactions with no conflicting or invalid state produced by applying all of them to the current ledger state. Each block must contain a hash reference to the prior block, and by applying each of them in order, the current ledger state can be reproduced as each block acts as a ‘diff’ from the previous state.

It should prevent double spends (a user ‘over-drawing’ their account, or sending their entire balance to two separate destinations) or other erroneous transactions, and respond to transaction submission requests with validation information. Valid blocks should also be signed by the single node running this service.

For reference to signature libraries and key pair management and generation, please refer to bouncycastle or spongeycastle https://www.bouncycastle.org/ – artifacts should be available through build. Any ECDSA signing mechanism is okay to use. Any alternative signing functions using a public / private key or hashing functions are acceptable. Primary focus or area of concern should be around designing an appropriate data model, appropriate REST API, validation criteria, hashing and signing functions, and overall architecture.

An ‘address’ corresponds to the unique data contained within a public key. For ECDSA, this can be represented in either compact or non-compact serialized form, either is acceptable.

Scope / Requirements Details:

- Any dependencies may be imported
- Scala language is required
- SBT build is required (no gradle or maven)

- Implement an API of some kind, RPC, REST, or any other which allows an external user to submit a transaction, and returns a clear success or failure along with potential errors associated with the failure.
- Add tests that prove at least the following cases:
    - Valid transaction - a valid signed transaction allows the movement of a specified balance from the source wallet to the destination wallet if the amount is less than or equal to the source balance.
    - Transaction signed by wrong wallet - a signed transaction signed by a different wallet than the source wallet is rejected and does not result in a balance transfer.
    - Invalid signed transaction - a signed transaction with a totally invalid signature is rejected and does not result in a balance transfer.
    - Overdraft - a valid signed transaction is rejected if the transaction amount exceeds the balance of the source wallet.
    - Repeating transaction - a valid signed transaction is rejected if it repeats a nonce that's already been accepted into a block.
- Should demonstrate accessibility to block data in some form (via API or otherwise), block data must contain references to all transactions within, and the references should allow query through a separate API
- Must be possible to query a transaction reference from the block API via a transaction API to access the data associated with a given transaction.
- Validations should be performed with respect to state, signature, and data formats.
- Transaction must demonstrate the ability to change a balance from a source address to a destination address
- Must be possible to access the balances / internal ledger state (API), balances must be consistent with transactions that have been sent and valid.

Data Model Details:

- Block:
    - Collection of or sequence of transactions
    - Monotonically increasing block sequence number
    - Reference to prior block hash
    - Signature attached corresponding to hash of the data of the block signed by the keypair of the current (single process) node.
    - Invalid to contain multiple conflicting transactions
- Transaction:
    - Source, destination addresses for moving funds
    - Amount being transferred
    - Signature from the source signing the hash of all data within the transaction corresponding to the correct source address private key.
    - Hash of the transaction should be what is being signed by the signature.
    - Nonce or ordinal associated with the number of sequential transactions used for the source address, monotonically increasing and used to prevent duplicate replay transactions.

- Address:
    - Corresponds to the data associated with a public key
    - Should have some notion of an associated balance (which is updated by blocks)
- Balances / Ledger State
    - Information about the total currency amount as of the latest block per account / address.

Project Submission:

Please submit the project as a public GitHub repository, containing a README explaining how to run the project (including tests,) the architectural design and decisions made, and how this project would be extended for production readiness.