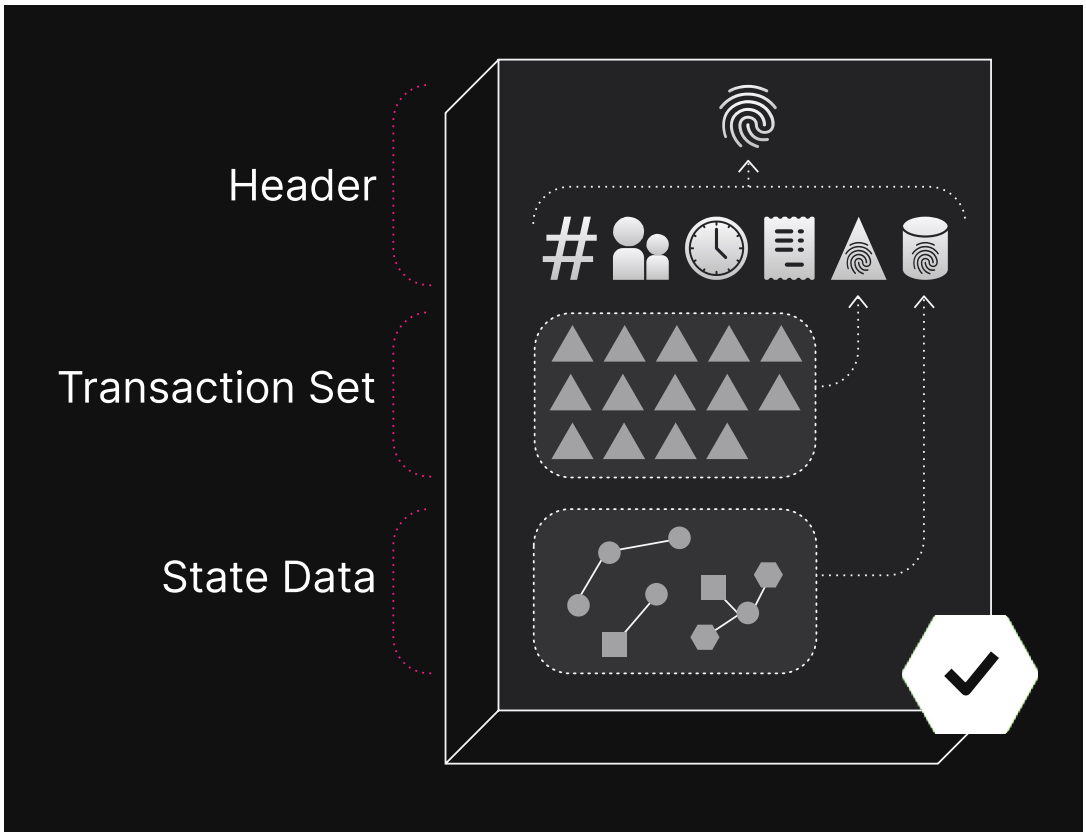
Welcome to Peter Parser (Draft)

In these files I will detail how the XRPL ledger is structured, how addresses are constructed and how various transaction types are serialized

Ledger Structure

The ledger structure is described at a high level in the XRPL docs here: https://xrpl.org/docs/concepts/ledgers/ledger-structure



Each ledger version has 3 parts:

- Header
- Transaction Set
- State Data

Header

Lets get the ledger header for ledger version 38129. Why 38129? Well this is the first ledger version I could find that contains a transaction, and since ledgers 0-32569 were lost (https://web.archive.org/web/20171211225452/https://forum.ripple.com/viewtopic.f=2&t=3613) and it is more interesting to look at one with a transaction than

not (more detail below), lets look at that one :)

First get the ledger header data:

In [1]:
```
from xrpl.clients import JsonRpcClient
import xrpl
import json
# I needed to add the next 2 lines to stop the error "RuntimeError:
import nest_asyncio
nest_asyncio.apply()

# Connect to the API
client = JsonRpcClient("https://s1.ripple.com:51234/")

# Create a ledger object model
ledger = xrpl.models.requests.Ledger(ledger_index="38129", binary=F

# Request the ledger object
ledger_result = client.request(ledger).result

# Print the ledger object
print(json.dumps(ledger_result, indent=2))
```


```
{
  "ledger_hash": "E6DB7365949BF9814D76BCC730B01818EB9136A89DB224F3F9
F5AAE4569D758E",
  "ledger_index": 38129,
  "validated": true,
  "ledger": {
    "account_hash": "2C23D15B6B549123FB351E4B5CDE81C564318EB845449CD
43C3EA7953C4DB452",
    "close_flags": 0,
    "close_time": 410424200,
    "close_time_human": "2013-Jan-02 06:43:20.000000000 UTC",
    "close_time_resolution": 10,
    "close_time_iso": "2013-01-02T06:43:20Z",
    "ledger_hash": "E6DB7365949BF9814D76BCC730B01818EB9136A89DB224F3
F9F5AAE4569D758E",
    "parent_close_time": 410424200,
    "parent_hash": "3401E5B2E5D3A53EB0891088A5F2D9364BBB6CE5B37A337D
2C0660DAF9C4175E",
    "total_coins": "99999999999996310",
    "transaction_hash": "DB83BF807416C5B3499A73130F843CF615AB8E797D7
9FE7D330ADF1BFA93951A",
    "ledger_index": 38129,
    "closed": true
  }
}
```


The output above should show the following json:

```
{
  "ledger_hash":
"E6DB7365949BF9814D76BCC730B01818EB9136A89DB224F3F9F5AAE4569D75
# sha512-half hash of ledger header, more detail below
  "ledger_index": 38129, # the ledger number, starting at 0
```

```
increasing by 1 each new ledger every ~3 seconds for XRPL
  "validated": true, # shows if the ledger version is part
of the XRPL completed ledgers
  "ledger": {
    "account_hash":
"2C23D15B6B549123FB351E4B5CDE81C564318EB845449CD43C3EA7953C4DB4
# sha512-half of account state data, more detail below
    "close_flags": 0, #
    "close_time": 410424200, # time that the ledger closed
in ripple epoch (seconds since 01/01/2000 00:00:00 UTC
    "close_time_human": "2013-Jan-02 06:43:20.000000000
UTC", # close time in readable format
    "close_time_resolution": 10, #
    "close_time_iso": "2013-01-02T06:43:20Z", # close time
in different format
    "ledger_hash":
"E6DB7365949BF9814D76BCC730B01818EB9136A89DB224F3F9F5AAE4569D75
# same as above
    "parent_close_time": 410424200, # close time of previous
ledger version in ripple epoch time
    "parent_hash":
"3401E5B2E5D3A53EB0891088A5F2D9364BBB6CE5B37A337D2C0660DAF9C417
# parent ledger hash, in this case its the "ledger_hash" for
"ledger_index"=38128
    "total_coins": "99999999999996310", # amount of XRP
drops remaining, total XRP for all accts minus any fees
    "transaction_hash":
"DB83BF807416C5B3499A73130F843CF615AB8E797D79FE7D330ADF1BFA9395
# sha512-half of the Merkle Patricia Trie root of
transaction data contained in this ledger version
    "ledger_index": 38129, # same as above
    "closed": true #
  }
}
```

This is the ledger header for ledger version 38129. It's immutable, transparent and secure.

There is something slightly weird (at least I think) about this ledger, that the close_time and parent_close_time are the same, this is not usually the case, I don't know why this is the case here!

Ledger Hash

Let's calculate the ledger_hash:

In [2]:
```
import hashlib

def Sha512Hash(Password):
    HashedPassword=hashlib.sha512(Password.encode('utf-8')).hexdige
    print(HashedPassword)

# Copy the necessary values from the above output and convert hex v
```

```
data = bytes.fromhex(
    "4C575200"          # HashPrefix::ledgerMaster, this is static
    "000094F1"          # ledger_index, 38129
    "016345785D89F196"  # total_coins, 99999999999996310
    "3401E5B2E5D3A53EB0891088A5F2D9364BBB6CE5B37A337D2C0660DAF9C417
    "DB83BF807416C5B3499A73130F843CF615AB8E797D79FE7D330ADF1BFA9395
    "2C23D15B6B549123FB351E4B5CDE81C564318EB845449CD43C3EA7953C4DB4
    "18769388"          # parent_close_time, 410424200
    "18769388"          # close_time, 410424200
    "0A"                # close_time_resolution, 10
    "00"                # close_flags, 0
)

# Compute SHA-512
sha512_hash = hashlib.sha512(data).digest()

# Take the first 32 bytes (SHA-512Half)
ledger_hash = sha512_hash[:32].hex().upper()

print("Ledger Hash:", ledger_hash)
```

```
Ledger Hash: E6DB7365949BF9814D76BCC730B01818EB9136A89DB224F3F9F5AAE
4569D758E
```

So we calculated the ledger hash E6DB7365949BF9814D76BCC730B01818EB9136A89DB224F3F9F5AAE4569D7 which is equal to the ledger_hash from the ledger header. All good so far :)

Most of the fields make sense, but what are the transaction_hash and account_hash? How do we construct these?

Transaction Hash

The transaction_hash is a sha512-half of the Merkle Patricia Trie of the transaction data from the ledger version.

HOW IS THIS DONE? Maybe do a pic?

First we will get the transaction data from the network:

In [3]:
```
from xrpl.clients import JsonRpcClient
import xrpl
import json
# I needed to add the next 2 lines to stop the error "RuntimeError:
import nest_asyncio
nest_asyncio.apply()

# Connect to the API
client = JsonRpcClient("https://s1.ripple.com:51234/")

# Create a ledger object model
ledger = xrpl.models.requests.Ledger(ledger_index="38129", binary=F

# Request the ledger object
```

```
ledger_result = client.request(ledger).result

# Print the ledger object
print(json.dumps(ledger_result, indent=2))
```


```
{
  "ledger_hash": "E6DB7365949BF9814D76BCC730B01818EB9136A89DB224F3F9
F5AAE4569D758E",
  "ledger_index": 38129,
  "validated": true,
  "ledger": {
    "account_hash": "2C23D15B6B549123FB351E4B5CDE81C564318EB845449CD
43C3EA7953C4DB452",
    "close_flags": 0,
    "close_time": 410424200,
    "close_time_human": "2013-Jan-02 06:43:20.000000000 UTC",
    "close_time_resolution": 10,
    "close_time_iso": "2013-01-02T06:43:20Z",
    "ledger_hash": "E6DB7365949BF9814D76BCC730B01818EB9136A89DB224F3
F9F5AAE4569D758E",
    "parent_close_time": 410424200,
    "parent_hash": "3401E5B2E5D3A53EB0891088A5F2D9364BBB6CE5B37A337D
2C0660DAF9C4175E",
    "total_coins": "99999999999996310",
    "transaction_hash": "DB83BF807416C5B3499A73130F843CF615AB8E797D7
9FE7D330ADF1BFA93951A",
    "ledger_index": 38129,
    "closed": true,
    "transactions": [
      {
        "validated": true,
        "ledger_index": 38129,
        "close_time_iso": "2013-01-02T06:43:20Z",
        "ledger_hash": "E6DB7365949BF9814D76BCC730B01818EB9136A89DB2
24F3F9F5AAE4569D758E",
        "hash": "3B1A4E1C9BB6A7208EB146BCDB86ECEA6068ED01466D933528C
A2B4C64F753EF",
        "tx_json": {
          "Account": "r3kmLJN5D28dHuH8vZNUZpMC43pEHpaocV",
          "DeliverMax": "10000000000",
          "Destination": "rLQBHVhFnaC5gLEkgr6HgBJJ3bgeZHg9cj",
          "Fee": "10",
          "Flags": 0,
          "Sequence": 62,
          "SigningPubKey": "034AADB09CFF4A4804073701EC53C3510CDC9591
7C2BB0150FB742D0C66E6CEE9E",
          "TransactionType": "Payment",
          "TxnSignature": "3045022022EB32AECEF7C644C891C19F87966DF9C
62B1F34BABA6BE774325E4BB8E2DD62022100A51437898C28C2B297112DF8131F2BB
39EA5FE613487DDD611525F1796264639"
        },
        "meta": {
          "AffectedNodes": [
            {
              "CreatedNode": {
                "LedgerEntryType": "AccountRoot",
```

```
                "LedgerIndex": "4C6ACBD635B0F07101F7FA25871B0925F883
615546215217275584 5CE691C49E",
                "NewFields": {
                  "Account": "rLQBHVhFnaC5gLEkgr6HgBJJ3bgeZHg9cj",
                  "Balance": "10000000000",
                  "Sequence": 1
                }
              }
            },
            {
              "ModifiedNode": {
                "FinalFields": {
                  "Account": "r3kmLJN5D28dHuH8vZNUZpMC43pEHpaocV",
                  "Balance": "981481999380",
                  "Flags": 0,
                  "OwnerCount": 0,
                  "Sequence": 63
                },
                "LedgerEntryType": "AccountRoot",
                "LedgerIndex": "B33FDD5CF3445E1A7F2BE9B06336BEBD73A5
E3EE885D3EF93F7E3E2992E46F1A",
                "PreviousFields": {
                  "Balance": "991481999390",
                  "Sequence": 62
                },
                "PreviousTxnID": "2485FDC606352F1B0785DA5DE96FB9DBAF
43EB60ECBB01B7F6FA970F512CDA5F",
                "PreviousTxnLgrSeq": 31317
              }
            }
          ],
          "TransactionIndex": 0,
          "TransactionResult": "tesSUCCESS",
          "delivered_amount": "unavailable"
        }
      }
    ]
  }
}
```


Now we need to serialize this data into two parts: transaction_blob, and metadata_blob.

Transaction Blob

Take the transaction data from the above output and serialize it:

In [4]:
```
import xrpl
import json
tx_json = """{
          "Account": "r3kmLJN5D28dHuH8vZNUZpMC43pEHpaocV",
          "DeliverMax": "10000000000",
          "Destination": "rLQBHVhFnaC5gLEkgr6HgBJJ3bgeZHg9cj",
          "Fee": "10",
          "Flags": 0,
```

```
          "Sequence": 62,
          "SigningPubKey": "034AADB09CFF4A4804073701EC53C3510CDC959
          "TransactionType": "Payment",
          "TxnSignature": "3045022022EB32AECEF7C644C891C19F87966DF9
        }"""

transaction_1 = """{
          "Account": "r3kmLJN5D28dHuH8vZNUZpMC43pEHpaocV",
          "Amount": "10000000000",
          "Destination": "rLQBHVhFnaC5gLEkgr6HgBJJ3bgeZHg9cj",
          "Fee": "10",
          "Flags": 0,
          "Sequence": 62,
          "SigningPubKey": "034AADB09CFF4A4804073701EC53C3510CDC959
          "TransactionType": "Payment",
          "TxnSignature": "3045022022EB32AECEF7C644C891C19F87966DF9
        }"""


#test = xrpl.transaction.transaction_json_to_binary_codec_form(json
#print(json.dumps(test, indent=2), type(test))
'''print(xrpl.models.transactions.Payment.from_blob("12000022000000
#payment_from_dict=xrpl.models.transactions.Payment.from_dict(json.
#print("Transaction blob: ",payment_from_dict.blob())



from xrpl.core.binarycodec import encode, decode
print("Encoded: ", encode(json.loads(transaction_1)))
```

```
Encoded:  1200002200000000240000003E6140000002540BE40068400000000000
000A7321034AADB09CFF4A4804073701EC53C3510CDC95917C2BB0150FB742D0C66E
6CEE9E74473045022022EB32AECEF7C644C891C19F87966DF9C62B1F34BABA6BE774
325E4BB8E2DD62022100A51437898C28C2B297112DF8131F2BB39EA5FE613487DDD6
11525F17962646398114550FC62003E785DC231A1058A05E56E3F09CF4E68314D4CC
8AB5B21D86A82C3E9E8D0ECF2404B77FECBA
```

In [17]:
```
test_decode=xrpl.core.binarycodec.decode("120000220000000240000003
print("Transaction Decode: ", json.dumps(test_decode, indent=2))
tx_string = transaction_1="""{
          "Account": "r3kmLJN5D28dHuH8vZNUZpMC43pEHpaocV",
          "Amount": "10000000000",
          "Destination": "rLQBHVhFnaC5gLEkgr6HgBJJ3bgeZHg9cj",
          "Fee": "10",
          "Flags": 0,
          "Sequence": 62,
          "SigningPubKey": "034AADB09CFF4A4804073701EC53C3510CDC959
          "TransactionType": "Payment",
          "TxnSignature": "3045022022EB32AECEF7C644C891C19F87966DF9
        }"""
print("Transaction encode: ", xrpl.core.binarycodec.encode(test_dec
print(xrpl.core.binarycodec.encode(json.loads(tx_string)))
```

```
Transaction Decode:  {
  "TransactionType": "Payment",
  "Flags": 0,
  "Sequence": 62,
  "Amount": "10000000000",
  "Fee": "10",
  "SigningPubKey": "034AADB09CFF4A4804073701EC53C3510CDC95917C2BB015
0FB742D0C66E6CEE9E",
  "TxnSignature": "3045022022EB32AECEF7C644C891C19F87966DF9C62B1F34B
ABA6BE774325E4BB8E2DD62022100A51437898C28C2B297112DF8131F2BB39EA5FE6
13487DDD611525F1796264639",
  "Account": "r3kmLJN5D28dHuH8vZNUZpMC43pEHpaocV",
  "Destination": "rLQBHVhFnaC5gLEkgr6HgBJJ3bgeZHg9cj"
}
Transaction encode:  1200002200000000240000003E6140000002540BE400684
00000000000000A7321034AADB09CFF4A4804073701EC53C3510CDC95917C2BB0150
FB742D0C66E6CEE9E74473045022022EB32AECEF7C644C891C19F87966DF9C62B1F3
4BABA6BE774325E4BB8E2DD62022100A51437898C28C2B297112DF8131F2BB39EA5F
E613487DDD611525F17962646398114550FC62003E785DC231A1058A05E56E3F09CF
4E68314D4CC8AB5B21D86A82C3E9E8D0ECF2404B77FECBA
1200002200000000240000003E6140000002540BE4006840000000000000000A732103
4AADB09CFF4A4804073701EC53C3510CDC95917C2BB0150FB742D0C66E6CEE9E7447
3045022022EB32AECEF7C644C891C19F87966DF9C62B1F34BABA6BE774325E4BB8E2
DD62022100A51437898C28C2B297112DF8131F2BB39EA5FE613487DDD611525F1796
2646398114550FC62003E785DC231A1058A05E56E3F09CF4E68314D4CC8AB5B21D86
A82C3E9E8D0ECF2404B77FECBA
```

Metadata Blob

In [6]:
```
test_decode=xrpl.core.binarycodec.decode("201C00000000F8E3110061564
print(json.dumps(test_decode, indent=2))
print(xrpl.core.binarycodec.encode(test_decode))
```

```
{
  "TransactionIndex": 0,
  "AffectedNodes": [
    {
      "CreatedNode": {
        "LedgerEntryType": "AccountRoot",
        "LedgerIndex": "4C6ACBD635B0F07101F7FA25871B0925F88361554621
52172755845CE691C49E",
        "NewFields": {
          "Sequence": 1,
          "Balance": "10000000000",
          "Account": "rLQBHVhFnaC5gLEkgr6HgBJJ3bgeZHg9cj"
        }
      }
    },
    {
      "ModifiedNode": {
        "LedgerEntryType": "AccountRoot",
        "PreviousTxnLgrSeq": 31317,
        "PreviousTxnID": "2485FDC606352F1B0785DA5DE96FB9DBAF43EB60EC
BB01B7F6FA970F512CDA5F",
        "LedgerIndex": "B33FDD5CF3445E1A7F2BE9B06336BEBD73A5E3EE885D
3EF93F7E3E2992E46F1A",
        "PreviousFields": {
          "Sequence": 62,
          "Balance": "991481999390"
        },
        "FinalFields": {
          "Flags": 0,
          "Sequence": 63,
          "OwnerCount": 0,
          "Balance": "981481999380",
          "Account": "r3kmLJN5D28dHuH8vZNUZpMC43pEHpaocV"
        }
      }
    }
  ],
  "TransactionResult": "tesSUCCESS"
}
201C00000000F8E3110061564C6ACBD635B0F07101F7FA25871B0925F88361554621
52172755845CE691C49EE824000000016240000002540BE4008114D4CC8AB5B21D86
A82C3E9E8D0ECF2404B77FECBAE1E1E51100612500007A55552485FDC606352F1B07
85DA5DE96FB9DBAF43EB60ECBB01B7F6FA970F512CDA5F56B33FDD5CF3445E1A7F2B
E9B06336BEBD73A5E3EE885D3EF93F7E3E2992E46F1AE6240000003E62400000E6D8
EEB01EE1E722000000002400000003F2D0000000062400000E484E2CC148114550FC6
2003E785DC231A1058A05E56E3F09CF4E6E1E1F1031000
```


In [10]:
```
test_meta = """{"AffectedNodes": [
{
  "CreatedNode": {
    "LedgerEntryType": "AccountRoot",
    "LedgerIndex": "4C6ACBD635B0F07101F7FA25871B0925F88361554621521
    "NewFields": {
      "Account": "rLQBHVhFnaC5gLEkgr6HgBJJ3bgeZHg9cj",
      "Balance": "10000000000",
      "Sequence": 1
```

```
    }
  }
},
{
  "ModifiedNode": {
    "FinalFields": {
      "Account": "r3kmLJN5D28dHuH8vZNUZpMC43pEHpaocV",
      "Balance": "981481999380",
      "Flags": 0,
      "OwnerCount": 0,
      "Sequence": 63
    },
    "LedgerEntryType": "AccountRoot",
    "LedgerIndex": "B33FDD5CF3445E1A7F2BE9B06336BEBD73A5E3EE885D3EF
    "PreviousFields": {
      "Balance": "991481999390",
      "Sequence": 62
    },
    "PreviousTxnID": "2485FDC606352F1B0785DA5DE96FB9DBAF43EB60ECBB0
    "PreviousTxnLgrSeq": 31317
  }
}
],
"TransactionIndex": 0,
"TransactionResult": "tesSUCCESS",
"delivered_amount": "unavailable"
}"""

test_meta2 = """{"AffectedNodes": [
{
  "CreatedNode": {
    "LedgerEntryType": "AccountRoot",
    "LedgerIndex": "4C6ACBD635B0F07101F7FA25871B0925F88361554621521
    "NewFields": {
      "Account": "rLQBHVhFnaC5gLEkgr6HgBJJ3bgeZHg9cj",
      "Balance": "10000000000",
      "Sequence": 1
    }
  }
},
{
  "ModifiedNode": {
    "FinalFields": {
      "Account": "r3kmLJN5D28dHuH8vZNUZpMC43pEHpaocV",
      "Balance": "981481999380",
      "Flags": 0,
      "OwnerCount": 0,
      "Sequence": 63
    },
    "LedgerEntryType": "AccountRoot",
    "LedgerIndex": "B33FDD5CF3445E1A7F2BE9B06336BEBD73A5E3EE885D3EF
    "PreviousFields": {
      "Balance": "991481999390",
      "Sequence": 62
    },
    "PreviousTxnID": "2485FDC606352F1B0785DA5DE96FB9DBAF43EB60ECBB0
```

```
    "PreviousTxnLgrSeq": 31317
  }
}
],
"TransactionIndex": 0,
"TransactionResult": "tesSUCCESS"
}"""

test_meta_json = json.loads(test_meta2)
print(json.dumps(test_meta_json, indent=2))
#print(xrpl.core.binarycodec.encode(test_decode))
print(xrpl.core.binarycodec.encode(test_meta_json))
```

```
{
  "AffectedNodes": [
    {
      "CreatedNode": {
        "LedgerEntryType": "AccountRoot",
        "LedgerIndex": "4C6ACBD635B0F07101F7FA25871B0925F88361554621
52172755845CE691C49E",
        "NewFields": {
          "Account": "rLQBHVhFnaC5gLEkgr6HgBJJ3bgeZHg9cj",
          "Balance": "10000000000",
          "Sequence": 1
        }
      }
    },
    {
      "ModifiedNode": {
        "FinalFields": {
          "Account": "r3kmLJN5D28dHuH8vZNUZpMC43pEHpaocV",
          "Balance": "981481999380",
          "Flags": 0,
          "OwnerCount": 0,
          "Sequence": 63
        },
        "LedgerEntryType": "AccountRoot",
        "LedgerIndex": "B33FDD5CF3445E1A7F2BE9B06336BEBD73A5E3EE885D
3EF93F7E3E2992E46F1A",
        "PreviousFields": {
          "Balance": "991481999390",
          "Sequence": 62
        },
        "PreviousTxnID": "2485FDC606352F1B0785DA5DE96FB9DBAF43EB60EC
BB01B7F6FA970F512CDA5F",
        "PreviousTxnLgrSeq": 31317
      }
    }
  ],
  "TransactionIndex": 0,
  "TransactionResult": "tesSUCCESS"
}
201C00000000F8E3110061564C6ACBD635B0F07101F7FA25871B0925F88361554621
52172755845CE691C49EE824000000016240000002540BE4008114D4CC8AB5B21D86
A82C3E9E8D0ECF2404B77FECBAE1E1E51100612500007A55552485FDC606352F1B07
85DA5DE96FB9DBAF43EB60ECBB01B7F6FA970F512CDA5F56B33FDD5CF3445E1A7F2B
E9B06336BEBD73A5E3EE885D3EF93F7E3E2992E46F1AE6240000003E62400000E6D8
EEB01EE1E722000000002400000003F2D0000000062400000E484E2CC148114550FC6
2003E785DC231A1058A05E56E3F09CF4E6E1E1F1031000
```


In [ ]:

Check this is the same as the on-chain data:

In [12]:
```
from xrpl.clients import JsonRpcClient
import xrpl
import json
# I needed to add the next 2 lines to stop the error "RuntimeError:
import nest_asyncio
```

```
nest_asyncio.apply()

# Connect to the API
client = JsonRpcClient("https://s1.ripple.com:51234/")

# Create a ledger object model
ledger = xrpl.models.requests.Ledger(ledger_index="38129", binary=T

# Request the ledger object
ledger_result = client.request(ledger).result

# Print the ledger object
print(json.dumps(ledger_result, indent=2))
```


```
{
  "ledger_hash": "E6DB7365949BF9814D76BCC730B01818EB9136A89DB224F3F9
F5AAE4569D758E",
  "ledger_index": 38129,
  "validated": true,
  "ledger": {
    "ledger_data": "000094F1016345785D89F1963401E5B2E5D3A53EB0891088
A5F2D9364BBB6CE5B37A337D2C0660DAF9C4175EDB83BF807416C5B3499A73130F84
3CF615AB8E797D79FE7D330ADF1BFA93951A2C23D15B6B549123FB351E4B5CDE81C5
64318EB845449CD43C3EA7953C4DB45218769388187693880A00",
    "closed": true,
    "transactions": [
      {
        "meta_blob": "201C00000000F8E3110061564C6ACBD635B0F07101F7FA
25871B0925F883615546215217275584 5CE691C49EE824000000016240000002540B
E4008114D4CC8AB5B21D86A82C3E9E8D0ECF2404B77FECBAE1E1E51100612500007A
55552485FDC606352F1B0785DA5DE96FB9DBAF43EB60ECBB01B7F6FA970F512CDA5F
56B33FDD5CF3445E1A7F2BE9B06336BEBD73A5E3EE885D3EF93F7E3E2992E46F1AE6
240000003E62400000E6D8EEB01EE1E722000000002400000 03F2D00000000624000
00E484E2CC148114550FC62003E785DC231A1058A05E56E3F09CF4E6E1E1F103100
0",
        "tx_blob": "120000220000000024000000 3E6140000002540BE4006840
0000000000000A7321034AADB09CFF4A4804073701EC53C3510CDC95917C2BB0150F
B742D0C66E6CEE9E74473045022022EB32AECEF7C644C891C19F87966DF9C62B1F34
BABA6BE774325E4BB8E2DD62022100A51437898C28C2B297112DF8131F2BB39EA5FE
613487DDD611525F17962646398114550FC62003E785DC231A1058A05E56E3F09CF4
E68314D4CC8AB5B21D86A82C3E9E8D0ECF2404B77FECBA",
        "hash": "3B1A4E1C9BB6A7208EB146BCDB86ECEA6068ED01466D933528C
A2B4C64F753EF"
      }
    ]
  }
}
```


Now we have all of the ingredients for the transaction hash, lets calculate it

We're aiming for
DB83BF807416C5B3499A73130F843CF615AB8E797D79FE7D330ADF1BFA9395

In [3]:
```
#This calculates the "key" or "index" this is defines where the tra

import hashlib
```

```
tx_blob = "1200002200000000240000003E6140000002540BE400684000000000
meta_blob = "201C00000000F8E3110061564C6ACBD635B0F07101F7FA25871B09

data = bytes.fromhex(
    "54584E00"+
    tx_blob
)


ledger_hash = hashlib.sha512(data).digest()[:32].hex().upper()
print(ledger_hash)
```

```
3B1A4E1C9BB6A7208EB146BCDB86ECEA6068ED01466D933528CA2B4C64F753EF
```

In [4]:

```
# Now lets calculate the leaf node has, based on this: https://gith
from binascii  import hexlify, unhexlify
import math

# Create a variable length prefix for a xrpl serialized vl field
def make_vl_bytes(l):
    if type(l) == float:
        l = ceil(l)
    if type(l) != int:
        return False
    if l <= 192:
        return bytes([l])
    elif l <= 12480:
        b1 = math.floor((l - 193) / 256 + 193)
        return bytes([b1, l - 193 - 256 * (b1 - 193)])
    elif l <= 918744:
        b1 = math.floor((l - 12481) / 65536 + 241)
        b2 = math.floor((l - 12481 - 65536 * (b1 - 241)) / 256)
        return bytes([b1, b2, l - 12481 - 65536 * (b1 - 241) - 256
    else:
        return err("Cannot generate vl for length = " + str(l) + ",

def hash_txn(txn):
    if type(txn) == str:
        txn = unhexlify(txn)
    return sha512h(b'TXN\x00' + txn)

def sha512h(x):
    m = hashlib.sha512()
    m.update(x)
    return m.digest()[:32]

# Hash the txn and meta data as a leaf node in the shamap
def hash_txn_and_meta(txn, meta):
    if type(txn) == str:
        txn = unhexlify(txn)

    if type(meta) == str:
        meta = unhexlify(meta)

    vl1 = make_vl_bytes(len(txn))
    vl2 = make_vl_bytes(len(meta))
```

```
    if vl1 == False or vl2 == False:
        return False

    payload = b'SND\x00' + vl1 + txn + vl2 + meta + hash_txn(txn)
    #print("payload1", hexlify(payload), payload.hex(), sha512h(pay
    return sha512h(payload)

print("hash: ", hash_txn_and_meta(tx_blob, meta_blob).hex().upper()
```

```
hash:  D42EE1686B347D14144A2398049A29E69BC3CF76140965EB1DAFC6BC351CA
683
```

In [17]:
```
print(b'MIN\x00' + unhexlify("3B1A4E1C9BB6A7208EB146BCDB86ECEA6068E

data = bytes(32)
print(data)

print(hashlib.sha512(b'MIN\x00' + data + data + data + b'\xd4.\xe1h
```

```
b'MIN\x00;\x1aN\x1c\x9b\xb6\xa7 \x8e\xb1F\xbc\xdb\x86\xec\xea`h\xed\
x01Fm\x935(\xca+Ld\xf7S\xef\xd4.\xe1hk4}\x14\x14J#\x98\x04\x9a)\xe6\
x9b\xc3\xcfv\x14\te\xeb\x1d\xaf\xc6\xbc5\x1c\xa6\x83'
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
DB83BF807416C5B3499A73130F843CF615AB8E797D79FE7D330ADF1BFA93951A
```

Account State

Now lets do the same for the account state

In [ ]:

In [ ]:
```
Now we have all of the inputs, lets calculate the transaction hash:
```

In [ ]:

In [ ]:

XRPL Accounts

What exactly is an XRP account? Why is it secure?

1. There are 2 types of accepted addresses on XRPL based on 2 cryptographic curves ed25519 and secp256k1 (the same one as bitcoin). Ed25519 is Edwards Curve Digital Signature algorithm, SECP256k1 is Elliptic Curve Digital Signature Algorithm.
2. Both are both Elliptic Curve approaches to Public Key cryptography. This simply means the following: a.
3. SECP256K1 a. Generate a key pair “keypair_secp256k1 =

xrpl.core.keypairs.derive_keypair(seed=)" b. OUTPUT: ('','') "keypair: ('03038CAB8D9A21904487416F64F960A3765A82F105F7EBC3054B94BD0 '008860E6AB41B773E05117D11919AD0A3963F6AE9658593E8292346078F c. This represents two things: (this is a bit mathsy but this is as deep as this course will go into maths!) i. first the public key is the x coordinates of a point P=(x,y) on the curve y^2=x^3+7 modulo 2**256−2*32−977, this is a 66 byte hexadecimal string prefixed with 02 if the corresponding y coordinate is even, and 03 if the corresponding y coordinate is odd. ii. second is a multiplier point for the elliptic curve prefixed with 00. iii. To validate the point do this

4. (Optional we can just validate private key generates public key) Split the string into 2 parts 03038CAB8D9A21904487416F64F960A3765A82F105F7EBC3054B94BD05
5. (Optional, we can just validate the private generates the public key) In python work out y coordinate using the script uncompress_point.py. You'll need to add the address to the script. For the above address the uncompressed address is of the format : 04038CAB8D9A21904487416F64F960A3765A82F105F7EBC3054B94BD05
6. Add the private key and run the script multiply_point.py 0x38cab8d9a21904487416f64f960a3765a82f105f7ebc3054b94bd051b88c 0x3bbe072531fce2c0e51a74c1b2fda9d3088082564e48a9017ce4de7ef759c a. The first part is the x coordinate and should be the same as the in the public key. b. The prefix is 02 if the y coordinate is even and 03 if it is odd iv. Sign a message
7.
8. Ed25519 a.
9. Now we have a public/private key pair in SECP256k1 we can sign a message. Lets sign the message "XRP is cool" a. What is message signing? Well I have a private key that is secret only to me, and a public key that I give to anyone (I want to send me money). How does the payee know that the public key they have is actually for me? Well one way is they can send me a message that only they know like "Hi, I think you're really cool! Please sign this and send me the signature, then I'll send you the money I owe you :)", I can sign the message, and then you can prove (using the public key) that I have the private key for the public key you will pay to. b. Lets try by example i. I have a public and private key. I send you the public key because I want you to send me some XRP. That public key is: "03483543312967AAC0C5A805BF68F6A188B33863A55B0B7AA710F50333 ii. Send me a message "Hi, think you're really cool! Please sign this and send me the signature, then I'll send you the money I owe you :)" iii. Sign the message and get result "304402204B4266DB40B1AB33EC71654ED84AA89ED99383A742E7F7CA5

iv. Validate the message

10. Interesting facts: a. Both are 128 bits – this is how XRP works

In [ ]:

XRPL Transactions

There are various types of transactions that are possible on the XRPL ledger: https://xrpl.org/docs/references/protocol/transactions/types

Payment Transactions

Payment transaction send XRP

In [ ]:
```
###################################
# Step 1: Create and FUND a source wallet on the testnet
# 1.1 Create and fund a wallet: xrpl-py > Wallet Methods > generate
# 1.2 Define a client connection for JsonRPC: xrpl-py > Network Cli
# 1.3 (Optional) Save the output of the wallet secret object
# Result: Wallet on the testnet, with a balance, that we can contro
###################################
import xrpl

testnet_client = xrpl.clients.JsonRpcClient(url="https://s.altnet.r

## Run this bit the first time you run the script only
#source_wallet = xrpl.wallet.generate_faucet_wallet(client=testnet_
#print(source_wallet)
#print(source_wallet.seed)

source_wallet_seed = "sXXX"
source_algorithm = xrpl.constants.CryptoAlgorithm('ed25519')
source_wallet = xrpl.wallet.Wallet.from_seed(seed=source_wallet_see
print("source_wallet: ", source_wallet)



###################################
# Step 2: Create destination wallet
# 2.1 Create a wallet: xrpl-py > Wallet Methods > xrpl.wallet.Walle
# 2.2 Create a seed: xrpl-py > XRPL Core Codecs > XRPL Keypairs Cod
# 2.3 Create entropy: any 16 byte string
# 2.4 Create algorithm object: xrpl-py > XRPL Global Variables > xr
# 2.5 Get the sequence: for a new wallet sequence can be set to 0
# Result: Wallet that does not yet exist on the testnet that we can
###################################

destination_entropy = xrpl.utils.str_to_hex("xrpumpkingiscool")
destination_algorithm = xrpl.constants.CryptoAlgorithm('secp256k1')
destination_seed = xrpl.core.keypairs.generate_seed(entropy=destina
destination_wallet = xrpl.wallet.Wallet.from_seed(seed=destination_
print("destination_wallet: ", destination_wallet)
```

```
###################################
# Step 3: Make the payment transaction
# 3.1 Create the payment transaction: xrpl-py > XRPL Models > XRPL
# 3.1 Get the address for source wallet: Step 1 > source_wallet.cla
# 3.2 Get the amount: xrpl-py > Utilities > xrpl.utils.xrp_to_drops
# 3.3 Get the destination address: Step 2 > destination_wallet.clas
# Result: Payment transaction object
###################################

import json
transaction_amount = xrpl.utils.xrp_to_drops(1000)
payment_transaction=xrpl.models.transactions.Payment(account=source
print(payment_transaction)

###################################
# Step 4: Signe the transaction
# 4.1 Sign the transaction: xrpl-py > Transaction Methods > xrpl.tr
# Result: signed Payment transaction object
###################################
signed_transaction = xrpl.transaction.autofill_and_sign(transaction
print(signed_transaction)

###################################
# Step 5: Submit the transaction
# 5.1 Submit the transaction: xrpl-py > Transaction Methods > xrpl.
# Result: successful Payment transaction on the testnet that debite
###################################
submit_transaction = xrpl.transaction.submit(transaction=signed_tra
print(submit_transaction)

###################################
# Step 6: Validate on the ledger
# 6.1 Get the balance for the destination wallet: xrpl-py > Account
# Result: print the balance of the destination wallet
###################################
print("Balance of destination: ", xrpl.account.get_balance(address=
```