# Credits Documentation

*Release 1.2.0*

David Smith

Mar 29, 2017

# Contents

Here you will find all the information necessary to begin building and deploying your own distributed ledger network with Credits. Start with *Getting started* manual.

CHAPTER 1

# Getting started

To start building a distributed ledger, you will need to install Credits Core framework, lay out your dapp scaffold and bootstrap your development network.

The Credits Core framework is written in Python and going forwards we will be using standard Python tools to work with it.

## Installation

Before installing the framework please make sure these external system dependencies are met:

- Python3.5+
- python pip
- libffi
- libzmq

To install the framework run:

```
pip install git+ssh://git@github.com/CryptoCredits/credits-core.git
```

You may want to use virtualenv to contain all installed python packages instead of putting it into your global system space.

The framework contains `credits` CLI tool that you will use to template your dapps and bootstrap your local dev network.

## Dapp scaffold

Once Credits Core is installed you can run:

```
credits dapp create --name <dapp_name>
```

This will create a `./<dapp_name>` folder and a scaffold of the Credits dapp inside it. You will use it to develop your distributed ledger.

Your dapp is effectively a python package, and it needs to be made discoverable for Credits Core framework to allow it to be later included into the ledger. To achieve this run:

```
pip install -e ./<dapp_name>
```

The `network.yaml` file is created as part of the dapp scaffold. It is used in both the dapp configuration, and as a base for the next step – network bootstrap.

## Network bootstrap

You need a distributed ledger network to both run your dapp and integrate your client applications. To bootstrap a distributed ledger network run:

```
credits local create ./<dapp_name>/network.yaml
```

This will roll out configuration for a single node network in the `./node00` folder. It contains all you will need to run the Credits Core network node.

You can also create multi-node network if needed by adding param `--count <number_of_nodes>`.

## Network run

To run the network node you will need to start the nodes you have just created. You do this with:

```
credits local run ./node00/*.yaml
```

To start the node you need to provide it with both the network config, which is copied to the node folder and the node config itself. Both are located in the node folder as `network.yaml` and `node.yaml`.

This starts node process in the foreground and outputs the node logs into stdout.

By default `loglevel` in node config is set to `WARNING`, you may want to make it more verbose with `--log-level <LOG_LEVEL>` option.

If you need to run a network of several nodes - you will need to run each node separately.

CHAPTER 2

# Credits Core blockchain

## Terminology

Before discussing the details of Credits Core framework architecture it's worth defining certain basic terms. Some of the terms are so widely used and abused across the industry it's worth stating so to avoid misconceptions.

### Blockchain

In the limited technical sense implied here the term *blockchain* refers to the data structure comprising of cryptographically linked data blocks. It implies an ability to reliably verify the contents of the blocks but does not imply a distribution of any kind.

The blockchain structure assumes and enforces append-only ability and intentionally gives no way to edit data cryptographically locked in the blocks.

Blockchain does not guarantee or prevent the replacement of parts or all blockchain with other blocks with different contents in case the consensus mechanics allows that, but that lies with the domain of DLT and consensus (see next).

### Distributed Ledger Technology

Distributed Ledger Technology (DLT) is a system that allows reliable replication of data between multiple nodes according to certain consensus algorithm. Usually it assumes the full and equal distribution of data between all nodes and without a central authority, but this really depends on the consensus algorithm used.

DLT depends on blockchain to operate since without ability to securely and independently verify the data the DLT degenerates into a simple master-master replication and can no longer guarantee the integrity of data being replicated.

### Consensus

An algorithm defining the way nodes of the DLT agree on the contents of the next block. Certain algorithms of consensus do allow for replacing of parts of blockchain starting from a certain block with different versions of history (forking), while others don't.

Consensus may require a certain arbitrary cryptographic challenge to be continuously solved (Proof of Work), provide a proof of possession of certain cryptographic keys (Proof of Stake) or have any other mechanics, but generally, it has to be deterministic and independently verifiable, so likely will rely on strong cryptography.

#### Proof of Work consensus

Proof of work is the most often mentioned mechanism for achieving consensus. Proof of Work requires that a contributor does a deterministically difficult amount of work that is then easy to check. Bitcoin, for example, does this by making miners hash until they get the longest string of zeroes. This artificially slows down block creation and makes it computationally and thus financially expensive in the Bitcoin network. Anyone can mine blocks but given the current normalised difficulty, it takes a ridiculously long time for non-specialized hardware to mine a valid block. This process is the only way invented by now to reliably implement a public permissionless consensus where anybody can participate.

#### Proof of Stake consensus

Proof of stake is far more like a traditional weighted voting model. Everyone locks up some value as a promise of their good intentions inside the system and then there are fixed voting rounds where each person votes using the weight of the value locked up. In an example both Alice and Bob stake 50 value into the system, they both have equal votes but neither have the majority. Both together can vote and provide the majority for confirming a block. Anyone can propose a block but only those with stake can vote.

This model does work very well in case the permissioned DLT, i.e. where participants of consensus have to be given an explicit prior permission to join. In this case, their stake contribution can be verified as part of the permission granting process.

### Blockchain forking

Applied to the blockchain, the event of forking is an occasion where starting from a certain block the chain of blocks splits into two or more chains, descending from the same parent but further producing different blocks according to different rules.

Some DLT consensus algorithms allow that, others don't.

## Credits Core

As any blockchain - the Credits blockchain consists of Blocks. And every block contains transactions that happened since the previous block created.

Unlike other blockchains, the Credits blockchain also contains States, which are snapshots of the contents of the blockchain created after each block is appended to the blockchain.

### Blockchain State

While still being a blockchain, Credits Core blockchain starts with the state and not a block.

The State, or state of the world, is simply just a key-value map of data. Credits Core has one global state object, that is comprised of many different *models*. The Dapp models are effectively data models and are used to define the data structure that is supposed to go into specific substate, and also provide an FQDN that will be used to reference that part of the state.

There can be multiple different models in the global state, and it can contain arbitrary values, all that is important is that every key in the state is a string. An example of a traditional state as shown:

```
{
    "works.credits.loans": {
        "Bob": 200,
        "Dave": 200
    },
    "works.credits.balances": {
        "Alice": 100,
        "Carol": 100
    }
}
```

In the example above there is a `works.credits.balances` model and a `works.credits.loans` model. Each model has to have an FQDN key and may have entries inside it. You can define as many models as you need inside the global state.

Inside the model you can have an arbitrary number of key-value pairs. Usually, Core addresses are used as the keys in a model but this doesn't always need to be the case, as the keys nature depends on application's business logic. The model is ordered by insertion order and the whole global state is hashed to provide a state hash.

The initial state of the world may be referenced as Genesis State, but normally it's called just `state 0`.

## Blockchain Block

Essentially a block is just a collection of transactions. Blocks are formed by taking valid unconfirmed transactions from the internal transactions pool and making a block that contains these transactions. Once the block is formed it is distributed in the network and nodes can decide to vote and commit to this block. A block also contains information on the previous state of the world that it is built on. By referencing the previous state, a node can take the block, check that it is starting in the same place as the creator of the block, apply the transactions and calculate the next state. Assuming all nodes are following the same logic – all nodes will compute identical next state. If that is not the case – network partitioning will occur, and this edge case is discussed *a little later*.

## Blockchain structure

Building from states and blocks the chain is formed. Because Credits blockchain has intermediate states it's not a direct link from block to block, instead, a block is formed from the current state, and then the application of that block to current state forms the next state.

Imagine starting at the following state 0:

```
{
    "works.credits.balances": {
        "Alice": 100,
        "Bob": 0
    }
}
```

And there is a transaction that moves 50 credits from `Alice` to `Bob`. This transaction can apply to state 0, so it is formed into a block that builds upon state 0.

```
+-----------+
|           |
|  State 0  |
|           |
+-----+-----+
      |
      |
+-----v-----+
|           |
|  Block 0  |
|           |
+-----------+
```

The block is then distributed between the nodes and references the state it is built on. Once the network agrees to make this block the next one in the chain each node applies transactions in this block to state 0 to produce the next state.

```
+-----------+      +-----------+
|           |      |           |
|  State 0  |   +-->  State 1  |
|           |   |  |           |
+-----+-----+   |  +-----------+
      |         |
      |         |
+-----v-----+   |
|           |   |
|  Block 0  +---+
|           |
+-----------+
```

The new state 1 looks like the following:

```
{
    "balance": {
        "Alice": 50,
        "Bob": 50
    }
}
```

A new transaction is formed and posted to the blockchain, this transaction moves the remaining 50 from `Alice` to `Bob`. Another new block is formed looking like such:

```
+-----------+      +-----------+
|           |      |           |
|  State 0  |   +-->  State 1  |
|           |   |  |           |
+-----+-----+   |  +-----+-----+
      |         |        |
      |         |        |
+-----v-----+   |  +-----v-----+
|           |   |  |           |
|  Block 0  +---+  |  Block 1  |
|           |      |           |
+-----------+      +-----------+
```

The process continues and block 1 will be applied to state 1, forming the next full state.

```
+-----------+      +-----------+      +-----------+
|           |      |           |      |           |
|  State 0  |   +-->  State 1  |   +-->  State 2  |
|           |   |  |           |   |  |           |
+-----+-----+   |  +-----+-----+   |  +-----------+
      |         |        |         |
      |         |         |         |
+-----v-----+   |   +-----v-----+   |
|           |   |   |           |   |
|  Block 0  +---+   |  Block 1  +---+
|           |       |           |
+-----------+       +-----------+
```

Leaving it with a final state of:

```
{
    "balance": {
        "Alice": 0,
        "Bob": 100
    }
}
```

From here onwards other transactions can happen, further mutating global state and adding new blocks to the chain. The process will run indefinitely as long as there is a quorum of nodes in the network to agree on blocks and new valid transactions are coming in.

## Credits Core consensus

Credits Core consensus is a leaderless two-phase commit algorithm with variable *voting power*.

This means that each and every DLT network participant is equal in its rights to gather transactions from the unconfirmed transactions pool and form a block, and they are free to vote on the blocks that make the most sense according to current block validation rules. Also, votes may have different weights though according to voting power distribution for a given network.

Essentially Credits Core consensus is a variant of Proof of Stake algorithm.

### Consensus example sequence

Assume a network of three nodes, A B and C. Network starts at height 0, no blocks exist yet and the current state of the network is state 0.

1. Node A receives a valid transaction from a client through HTTP gateway.
2. Node A verifies the transaction and onboards it, adding it to the unconfirmed transactions pool.
3. Node A recognises new transaction in the pool and tries to form a block
4. Node A forms a block proposal and sends it out to other nodes in the network.
5. Nodes B and C receive the block proposal and verify the proposed block for validity.
6. Nodes B and C confirm block validity and start voting on the block. This is the process of *voting*, phase one of the consensus algorithm. At this point only one block proposal exists, so all votes are given to this block.
7. Votes are exchanged and if one block reaches the quorum of *Voting Power* backing – it is now a *voted block*. The phase one is done.
8. When the node receives enough votes on a block and thus finds out that the block was *voted*, the node goes into phase two of the consensus algorithm – *committing* to the block.
9. Once the selected block has received quorum Voting Power backing - it is considered committed to be the next block in the chain. Phase two is done.

10. Every node receiving the block with enough committing VP attached to it adds this block to the chain and persists it on whatever storage is used with that particular node.
11. The process can start over from scratch if there are new transactions in the pool.

In a more complex real life situations, there will be multiple transactions forming block proposals on different nodes and nodes will exchange and vote on proposals until one of the proposals reaches the quorum.

The consensus algorithm depends on several things:

1. There has to be a required quorum defined. The current default is at least 51% of all Voting Power present in the system.
2. There has to be voting power available in the system, and whoever is executing the votes using it – has to have access to the corresponding private keys. This is discussed in more details in *Voting* section.
3. There has to be a connection between nodes that will allow to propagate the blocks and votes.

Given these conditions are met - the consensus algorithm will function correctly.

## Voting and Voting Power

Voting in Credits Core consensus is strongly tied to Voting Power (VP). VP is one or more arbitrary integer values assigned addresses on the blockchain. These values are stored in the blockchain *State* and represent the weight of votes allocated to each of these addresses.

```
{
    "credits.voting.model.voting": {
        "19joM8wBG7bAmBypMj23DBmmjGmqfxL4Bj": 100,
        "14RixTSeLtit5GJoDKdEJ23ob74vcvcFvv": 100
    }
}
```

Example above is a subset of the blockchain state showing two addresses with `100` of voting power assigned to each. The VP figure itself assigned to each address is of little importance, what matters is the relative weight of VP of each address to the total VP declared.

To use this voting power, i.e. cast a vote one must have in possession the private key corresponding to the address VP is assigned to to be able to prove the ownership. By default, every node of the Credits DLT network holds a private key to one of the VP addresses and casts votes with it, and VP itself is split equally between all parties.

However it is totally possible practically to create new addresses and assign VP to them, imbalance the system by giving some addresses more VP than others or strip out VP completely from some addresses (set to `0` or delete address).

In the edge case when VP is completely absent, so the VP model contains no addresses with none-zero VP values – DLT will stall and won't be able to progress, i.e. confirm new blocks.

So in the simplest default case VP is distributed equally between addresses attached to each node of the DLT network.

```
+------------+      +------------+
|            |      |            |
|  Node 0    <------>  Node 1    |
|  VP 100    |      |  VP 100    |
+-----+------+      +-+----------+
      |               |
      +----+        +---+
           |        |
         +-v--------v-+
         |            |
        |  Node 2   |
        |  VP 100   |
        +-----------+
```

In this example each node has VP value of `100` and votes for blocks using this VP.

A more advanced case:

```
+-----------+      +-----------+
|           |      |           |
|  Node 0   <------>  Node 1   |
|  VP 100   |      |  VP 100   |
+-----+-----+      +-+-------+-+
      |              |       |
      +---+       +--+       |
          |       |          |
        +-v-------v-+      +-v---------+
        |           |      |           |
        |  Node 2   <------>  Node 2   |
        |  VP 100   |      |  VP    0  |
        +-----------+      +-----------+
```

In here three nodes still have same `100` VP, while one other node has no VP at all. This node cannot vote on the blocks going into the blockchain but has full visibility of the blockchain contents and can confirm it's validity.

### Partitioning

Forking is not possible in Credits Core, but the Credits network can go into the partitioned state.

Partitioning is a situation where the quorum cannot be reached in a part of the network because either the network connectivity is impaired and nodes cannot propagate the votes on new blocks or the nodes cannot agree on the rules of the network and do not cast votes on blocks that they assume invalid.

In both of these scenarios the Voting Power required to choose and commit to new block becomes unavailable to part or the whole network, and the affected part of the network stalls, i.e. cannot continue to grow the chain.

Since not enough VP is available to progress the chain and vote on blocks – no minority chain will form, the forking or chain reorganisation will not happen at any circumstance and any data that was voted and committed to the chain before the partitioning will be not affected.

In this case, if only a minority of the network's VP is affected and the majority is still both in agreement and has enough VP available to form a quorum – the majority of the network will continue to operate normally.

The partitioning situation will likely to require manual intervention to identify and address the root of the problem, whether it is a connectivity issue, consensus disagreement or anything else.

CHAPTER 3

# Credits Core Dapp

The Credits Dapp is a distributed ledger-based application developed using the Credits Core framework. Dapp is a unit of business logic, and it's supposed to be designed with specific real-life use case in mind.

Credits Dapp is a software module designed to run as part of the Credits Core node, it's not a standalone application in itself. However, it's not meant to be uploaded into the blockchain as a smart contract code. Instead, it is provided as a part of network setup and configuration before Core node is started. Once the node is started the Dapp runs as integral part of the Core, inside Core's runtime.

Credits Dapp is a Python module and has to be discoverable to the Core using standard Python importing tools. Credits Dapp is must provide *state* definition, transforms that describe how that state is to be mutated and optionally unit tests that ensure the validity of the above.

Credits Dapps can be mixed together, inherited, extended and reused just as any other software libraries. Several basic dapps are readily available as built-ins and loadable modules provided by Credits.

## Models

Credits Dapp Model is a data model used to populate the relevant state. A model carries an FQDN that defines the name for relevant state. Also model defines defaults that will be used for new keys added to this state.

Model should be extended from the base class `credits.core.model` and produce a *Marshallable* object. Here is an example of a fully valid model:

```python
class BalanceModel(Model):
    fqdn = 'works.credits.balance'
    default_factory = int
```

This model is describing balances and enforces its values to be an integer.

## Tranforms

Transform is a standard unit of work in Credits Core. It's used to modify, transform the state and produce the new state. Each transform has to have `verify()` and `apply()` methods that verify the validity of the transform, it's data and the state it's applied against and apply actual changes to the state.

Transform has to be an *Applicable*, *Marshallable* and *Hashable* object. Transforms are constructed with all necessary values required to modify state during the `apply()` method call and are used as a standard unit of work inside the Credits Core to modify the *state*.

When a Transaction with Transform inside is onboarded to the Node, it is verified against the current *Blockchain State*. The `verify()` method should perform checks against the state to check if this Transform will be applicable either now or sometime in the future. If a Transform fails verification at any point it will be flushed from the Node. Note that if a Transform attempts to modify state during its validation, changes made to the state will be disposed of.

When a Transform is applied it will be given the current state and expected to modify and return a new state. During the transaction application, a Transform may perform any verification that has to be performed "upon application". If this verification fails, the apply should fail and return an erroneous result. However, failure of `apply` doesn't cause the transaction to be discarded. It stays in the unconfirmed pool until it's either gets confirmed or it's `verify` method also fails. Only when `verify` fails transaction is discarded and forgotten.

### Example transform

Key-Value storage use case is probably the simplest one possible on the distributed ledger. In this case, following transform can be used:

```python
MODEL_KV = "credits.kv.model.kv"

class KVTransform(transform.Transform):
    fqdn = "credits.kv.transform.KVTransform"
    required_models = {
        MODEL_KV
    }

    def __init__(self, author, key, value):
        self.author = author
        self.key = key
        self.value = value

    @property
    def required_keys(self):
        return {
            self.key,
        }

    @property
    def required_authorizations(self):
        return {
            self.author,
        }

    @classmethod
    def unmarshall(cls, registry, payload):
        return cls(
            author=payload["author"],
            key=payload["key"],
            value=payload["value"],
        )

    def marshall(self):
        return {
            "fqdn": KVTransform.fqdn,
            "author": self.author,
            "key": self.key,
            "value": self.value,
        }

    def verify(self, state):
        """
        Ensure that the value given is in-fact valid JSON.
        """
        json.dumps(self.value)  # failures here are ok

    def apply(self, state):
        """
        Set the value against its key in the Key Value Model.
        """
        state[MODEL_KV][self.key] = self.value # This function mutates state so␣
↪there is not need to return it
```

This is an example of a fully functional Key-Value transform. It can store arbitrary values in the blockchain against arbitrary keys. The only verification done is to make sure the value is JSONifiable.

This and few other transforms are readily available in `credits/core/builtin.py` as built-in transforms.

## Reusable dapps

A Credits Dapp is essentially a Python package, and thus it can be simply imported and reused as any other regular code library. Several simplest transforms are available within the Core itself as built-ins, while several more advanced libraries are accessible as additional modules.

### Built-ins and third party

To use external modules you will need to define them as `requirements` in *network config* and also define specific transforms and models from those modules that you want to be imported into your dapp.

Apart from straight reuse you can also extend and reuse the code from external modules in your own transforms. To do that just `import` it as a regular python library. You can import from `credits.core.builtin` without any external dependencies. These are available transforms:

- KVTransform
- ACLTransform
- BalanceAdjustTransform
- BalanceTransferTransform

## Core network configuration

To run a network Core nodes you will need a `network.yaml` configuration file. It's automatically generated by `credits dapp create` CLI command, however during development you probably will need to edit `network.requirements`, `network.transforms`, `network.models` and `network.initial_state`.

Sample config:

```
network.name: sample-finance-dapp_network
network.uuid: b8805bde-755b-4349-9d48-a217dbf3b24d
network.initial_state: {}
network.enable_manhole: True
network.hash_provider: sha256
network.requirements:
  - git+ssh://git@github.com/CryptoCredits/credits-finance.git
  - git+ssh://git@github.com/CryptoCredits/credits-admin.git
network.transforms:
  - credits.core.KVTransform
  - finance.tokens.transforms.TokenTransferTransform
  - finance.tokens.transforms.TokenCreditTransform
  - finance.tokens.transforms.TokenDebitTransform
network.models:
  - finance.tokens.models.TokenModel
```

- `network.name` – name of this network. Must be alphanumeric lowercase.
- `network.uuid` – UUID of this network.
- `network.initial_state` – state 0 of the network. Any predefined seed balances, rood admin keys etc will go here.
- `network.enable_manhole` – debug feature. Is likely to be removed in final release.
- `network.hash_provider` – default hash provider to be used in this network.
- `network.requirements` – list of `pip` requirements to be installed and made discoverable by this network. Supports all the features as `python pip requirements.txt` file has.
- `network.transforms` – list of transforms to be loaded into the Core and made usable.
- `network.models` – list of models to be loaded into the Core and made usable.

CHAPTER 4

# Interfaces

## Marshallable

The Marshallable interface provides the necessary methods to convert fully realised instances into more primitive maps and to reverse the process by converting these maps back into instances.

### FQDN

Marshallable objects must contain a Fully Qualified Domain Name (FQDN), this is used by the framework to resolve objects from serialised representations. The FQDN should be made as an attribute of the implementor as either a class level variable or a property, either are acceptable. This fqdn will be used during the unmarshalling process to locate an associated Class to instantiate.

```python
class Foo:
    fqdn = "fully.qualified.domain.name.Foo"

# OR

class Bar:
    @property
    def fqdn(self):
        return "fully.qualified.domain.name.Bar"
```

### marshall

Marshalling is the process of converting an instance into a map of primitive variables which can be then serialised. This allows libraries like `json` and `msgpack` to use their `dumps` methods to convert the output of `marshall` to a string or series of bytes suitable for transport over the network. The `fqdn` used by the framework to resolve the implementor's class and convert the output of `marshallable` back into an instance. Note that a Marshallable object should also marshall any of it's subcomponents.

```python
def marshall(self) -> dict:
    return {
        "fqdn": self.fqdn,
        "variable": 49,
        "subcomponent": self.subcomponent.marshall(),
    }
```

### unmarshall

Once a marshalled object has been received, the unmarshall method may be used to reverse the process. Like marshalling, unmarshalling should also be performed on all subcomponents to fully resolve the object.

```python
@classmethod
def unmarshall(cls, registry: Registry, payload: dict) -> cls:
    # Note that registry.unmarshall will call cls.unmarshall on the class that
    # resolves to the subcomponent's fqdn.

    return cls(
        variable=payload["variable"],
        subcomponent=registry.unmarshall(payload["subcomponent"]),
    )
```

## Applicable

The `Applicable` interface provides the necessary methods for objects to perform manipulations against the state.

### verify

The `verify` method should accept State and verify if the implementor is capable of manipulating state either immediately or in the future. If verification fails it should return a failed result with a reason why verification failed.

NOTE: States have default values that are returned when non-existant keys are requested. When checking for a key's membership, you should use explicit membership checking via `in` and `not in`. All other approaches are considered unsafe.

```python
def verify(self, state: dict) -> Result:
    if self.from_address not in state["foo.bar"]:
        # self.from_address does not exist in 'foo.bar', in this example we
        # consider this a failure to verify and return a suitable message.
        return None, "from_addresss %s does not exist in 'foo.bar'" % self.from_
↪address

    return state
```

### apply

The `apply` method should manipulate and return state. In the event of an error, return an erroneous Result.

```python
def apply(self, state: dict) -> Result:
    try:
        state["foo.bar"][self.from_address] -= self.amount
        state["foo.bar"][self.to_address] += self.amount
        return state

    except Exception as e:
        raise e
```

## Signable

The `Signable` interface provides a single method to take an unsigned object and return a signed version.

### sign

The `sign` method should accept a `credits.key.SigningKey` and sign some sort of challenge stored within the implementor. Then return a new instance of the implementor with both the signature and the associated `credits.key.VerifyingKey`. These additional variables can then, for example, be used in conjunction with the `Applicable.verify` method to check if the implementor has been signed.

```
def sign(self, signing_key: SigningKey) -> self:
    verifying_key = signing_key.get_verifying_key()
    signature = signing_key.sign(self.challenge)

    return self.__class__(
        address=self.address,
        nonce=self.nonce,
        challenge=self.challenge,
        verifying_key=verifying_key,
        signature=signature,
    )
```

## Hashable

The `Hashable` interface provides a single method to provide a cryptographic hash of the implementor.

### hash

Given a `credits.hash.HashProvider`, construct some sort of string hash it.

```
def hash(self, hash_provider: credits.hash.HashProvider) -> str:
    # A primitive example where we use variables from this class to construct a
    # challenge. Then hash and return the output.

    challenge = self.name + str(self.age) + str(self.value)
    return hash_provider.hash(challenge)
```

# CHAPTER 5

# FAQ

## Is Credits a cryptocurrency?

No. Credits is not a cryptocurrency and has nothing to do with cryptocurrencies. Credits is a software framework for the creation of applications based on Distributed Ledger Technology (DLT).

## Is Credits a distributed ledger or blockchain?

The **distributed ledger** vs **blockchain** comparison is a politically charged question so is not easy to answer in the conventional sense. However, within the scope of this documentation we are using these terms in the strictly technical sense, and technical definitions of both are given in terms page.

In the strict technical sense Credits Core is a DLT framework.

## Is there a public Credits token?

No. Since Credits is not a cryptocurrency there is no underpinning token or anything of a kind.

## Is there a public Credits' blockchain?

No. Credits is not a public permissionless blockchain in the sense of e.g. Bitcoin being called a public blockchain and is not supposed to be. Credits Core is a DLT framework designed to help developers of conventional end-user applications do their work, i.e. build applications, without having to worry much about implications of a distributed ledger.

A public but permissioned distributed ledger can be built with Credits, and this distinction is discussed later in this FAQ.

## Is there a Credits blockchain at all?

Yes, and more than one. But those blockchains are mostly private, so it is not possible to access them without explicit permission from their respective owners.

## So what is Credits Core?

Credits Core is a software framework for building DLT based applications. It is a developer tool designed to help developers build applications using Distributed Ledger Technology. It has an underlying blockchain data structure in strict technical definition but is not meant to be a public ledger like Bitcoin or Ethereum. It is designed primarily for building applications with private permissioned blockchains.

## What is a private blockchain?

Private blockchain is quite simply - a blockchain that is kept privately and not exposed to the world. Very much like a traditional database - the private blockchain is a blockchain-as-database, backend system accessed by the backend application developers, DevOps and other engineers, but not accessible to the public.

Private blockchain is by definition permissioned since once needs an explicit permission to access such blockchain. However not being private doesn't make a blockchain immediately permissionless.

## What is permissioned blockchain or distributed ledger?

Permissioned blockchain is a blockchain where one needs to obtain a permission to join a specific distributed ledger network and participate in its consensus.

This may be a private ledger, where one needs explicit permission to even see the contents of the specific blockchain or a public permissioned ledger where the blockchain content is available for reading without prior permission, but joining the network is still possible only after explicit permission from the distributed ledger owners/operators.

A permissioned ledger may be public, where the content of the blockchain is available for reading and verification, or private.

## Is Credits a Proof of Work or Proof of Stake?

Credits is a Proof of Stake system. For more details please refer to refer to *Credits Core consensus algorithm* description.

## Is Proof of Stake actually useful?

In a permissioned distributed ledger there is no need to impose an arbitrary artificially complicated task (proof of work) to ensure the alignment of participants' economic incentives with the goals of the network's existence. A permissioned system by definition is created by certain participants with a specific goal and is not meant to be used by the uninterested public. Thus the participants of the network are supposed to have an interest in network's existence and functioning through interests external to the network itself.

Stakes in a permissioned system are a mere reflection of stakeholders input into the creation and maintenance of the said system. Stakes can and should reflect the relationships between the stakeholders outside of the network and correspond to mutual duties and obligations.

In this sense Proof of Stake consensus algorithm is a much better reflection of the permissioned distributed ledger system then a Proof of Work.

## What about "nothing at stake" problem?

Nothing-at-stake is a conceptual problem and a possible attack vector in a Proof of Stake distributed ledgers that implies that uninterested operators that may hold "stake" in the Proof of Stake consensus not necessarily have the actual stake in the said system, i.e. have nothing to lose if the system is malfunctioning or subverted.

This is a problem in a permissionless Proof of Stake systems since the wide public is allowed to join the consensus without an obligation to have an explicit stake in the system. This is not so much of a problem in the permissioned Proof of Stake ledgers because all participants of the network are expected to have an external interest in network's existence and good standing. So in the permissioned system, the participants are expected to have stake prior to joining the network, otherwise, they should have no reasons and should be not allowed to participate, and so the nothing-at-stake problem does not apply.

## What about blockchain forking?

Forking is not possible in Credits consensus algorithm. For more details please refer to *consensus algorithm details*.