BSMD

Release 0.0.1

David Lopez & Bilal Farooq

Sep 07, 2020

LAYERS:

Blockchain has the potential to render the transaction of information more secure and transparent. Nowadays, transportation data are shared across multiple entities using heterogeneous mediums, from paper collected data to smartphone. Most of this data are stored in central servers that are susceptible to hacks. In some cases shady actors who may have access to such sources, share the mobility data with unwanted third parties. A multi-layered Blockchain framework for Smart Mobility Data-market (BSMD) is presented for addressing the associated privacy, security, management, and scalability challenges.

Each participant shares their encrypted data to the blockchain network and can transact information with other participants as long as both parties agree to the transaction rules issued by the owner of the data. Data ownership, transparency, auditability and access control are the core principles of the proposed blockchain for smart mobility data-market. For a description of the framework read the paper.

CHAPTER

ONE

PREREQUISITES

To start using the BSMD you must have at least one *Iroha* node running. *Hyperledger Iroha* is a straightforward distributed ledger technology (DLT), inspired by Japanese Kaizen principle — eliminate excessiveness (muri). Iroha has essential functionality for your asset, information and identity management needs, at the same time being an efficient and trustworthy crash fault-tolerant tool for your enterprise needs[1]. Click here to build and install an Iroha network

1.1 Contract

1.1.1 Broker

Defines a Broker class

class layers.contract.broker.**Broker**(*private_key*, *name*, *ip*, *public_info*)

Brokers look for users (passive nodes) in the BSMD and arrange transactions. Brokers are created in the public domain and can get the public details of all passive nodes

Parameters

- **private_key** (*str*) – Private key of the broker. This is not save in the class is just used to generate a public_key. In other functions the private_key must be used to sign transactions. You can generate private keys with IrohaCrypto.private_key()
- **name** (*str*) – name of the user (lower case)
- **ip** (*str*) – ip of one node hosting the blockchain
- **public_info** (*json*) – public information of the broker, e.g., type of node: broker

create_account(*private_key*)

Create a broker account in the BSMD. Brokers are automatically created in the public domain. This function works in two steps

1. The broker account in created in the public domain
2. Set the public details of the broker

Example

```
>>> import json
>>> from admin.administrator import Domain
>>> x = { "address": "123 Street, City", "type": "broker" }
```

(continues on next page)

[1] https://github.com/hyperledger/iroha.

(continued from previous page)

```
>>> public_info = json.dumps(x)
>>> broker = Broker('private_key','broker', '123.456.789', public_info)
>>> broker.create_account('private_key')
```

Parameters **private_key** (*str*) – The private key of the user

get_details_from(*user*, *private_key*)

Consult all details of the node. Broker can only consult details in the 'public' domain

Example

```
>>> import json
>>> from admin.administrator import Domain
>>> from layers.identification.identification import User
>>> x = { "gender": 30, "address": "123 Tennis" }
>>> user_info = json.dumps(x)
>>> x = { "address": "123 Street, City", "type": "broker" }
>>> broker_info = json.dumps(x)
>>> domain = Domain('name', 'default_role')
>>> user = User('private_key','David', domain, user_info)
>>> broker = Broker('private_key','broker', '123.456.789', broker_info)
>>> user_public_details = broker.get_details_from(user, 'private_key')
>>> print(user_public_details)
{
    "node@domain":{
        "Type":"user"
    }
}
```

Parameters

- **private_key** (*str*) – Key to sign the transaction
- **user** (User) – The user the broker want to consult

Returns solicited details of the user

Return type json

1.2 Communication

The communication module is use to establish p2p communications for transferring information. The *p2p_com* module can be used to securely chat or transferring files during a period of time. The *Federated_hook* module is used for p2p communication in a federated learning environment.

1.2.1 p2p_com

This module is for p2p communication between two nodes. The communication is done via sockets, for now the messages are not encrypted. For external networks you may need to open ports.

Assume you have two nodes. The ip:port of node1 is 123.456.789:5555 and while the ip:port of node2 is 987.654.321:5555. To stat a p2p communication do the following:

On node1 run

```
>>> receiver = Receiver('123.456.789', '4444')
>>> sender = Sender('987.654.321', '5555')
>>> treads = [receiver.start(), sender.start()]
```

On node2 run

```
>>> receiver = Receiver('987.654.321', '5555')
>>> sender = Sender('123.456.789', '4444')
>>> treads = [receiver.start(), sender.start()]
```

This code was taken from https://www.webcodegeeks.com/python/python-network-programming-tutorial/

class layers.communication.p2p_com.**Receiver**(*my_host*, *my_port*)

This class will receive messages

Parameters

- **my_host** (*str*) – My local ip address
- **my_port** (*str*) – My local port

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

class layers.communication.p2p_com.**Sender**(*my_friends_host*, *my_friends_port*)

This class is for p2p communication between two nodes. The communication is done via sockets, for now the messages are not encrypted. This class will receive messages

Parameters

- **my_friends_host** (*str*) – Ip address of the node you want to send a message
- **my_friends_port** (*str*) – Port of the node you want to send a message

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

layers.communication.p2p_com.**main**()

Main can be use to test the p2p service. Open two consoles try it.

1.2.2 Federated_hook

We use the bellow code for data transactions of large variables in the BSMD. In particular we use the socket implementation of coMind for transferring weights and we add a second layer to record all transactions in the BSMD This code was taken from comind.org.

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

class layers.communication.federated_hook.**_FederatedHook**(*is_chief*, *name*, *private_ip*, *public_ip*, *private_key*, *list_of_workers*, *domain*, *ip*, *wait_time=30*, *interval_steps=100*)

Provides a hook to implement federated averaging with tensorflow.

In a typical synchronous training environment, gradients will be averaged each step and then applied to the variables in one shot, after which replicas can fetch the new variables and continue. In a federated average training environment, model variables will be averaged every 'interval_steps' steps, and then the replicas will fetch the new variables and continue training locally. In the interval between two average operations, there is no data transfer, which can accelerate training.

The hook has two different ways of working depending if it is the chief worker or not.

The chief starts creating a socket that will act as server. Then it stays waiting _wait_time seconds and accepting connections of all those workers that want to join the training, and distributes a task index to each of them. This task index is not always necessary. In our demos we use it to tell each worker which part of the data-set it has to use for the training and it could have other applications.

Remember if you training is not going to be performed in a LAN you will need to do some port forwarding, the authors recommend you to have a look to this article they wrote about it.

Once the training is going to start sends it's weights to the other workers, so that they all start with the same initial ones. After each batch is trained, it checks if _interval_steps has been completed, and if so, it gathers the weights of all the workers and its own, averages them and sends the average to all those workers.

Workers open a socket connection with the chief and wait to get their worker number. Once the training is going to start they wait for the chief to send them its weights. After each training round they check if _interval_steps has been completed, and if so, they send their weights to the chief and wait for it's response, the averaged weights with which they will continue training.

Parameters

- **is_chief** (*bool*) – whether it is going to act as chief or not.
- **name** (*str*) – name of the node in the BSMD
- **private_ip** (*str*) – complete local ip in which the chief is going to serve its socket. Example: 172.134.65.123:7777
- **public_ip** (*str*) – ip to which the workers are going to connect.
- **private_key** (*str*) – private key of the node for signing the transactions
- **list_of_workers** (*list[str]*) – list of all the nodes that are willing to participate. In theory the chief node knows the list as he creates the domain and accounts for the participants
- **domain** (*str*) – name of the domain
- **ip** (*str*) – ip address for connecting to the BSMD
- **wait_time** (*int,optional*) – how long the chief should wait at the beginning for the workers to connect.
- **interval_steps** (*int,optional*) – number of steps between two "average op", which specifies how frequent a model synchronization is performed

_assign_vars(*local_vars*)
Utility to refresh local variables.

Parameters **local_vars** – List of local variables

Returns The ops to assign value of global vars to local vars.

_create_placeholders()
Creates the placeholders that we will use to inject the weights into the graph

_get_np_array(*connection_socket*)
Routine to receive a list of numpy arrays.

Parameters **connection_socket** – a socket with a connection already established.

_get_task_index()
Chief distributes task index number to workers that connect to it and lets them know how many workers are there in total.

Returns task index corresponding to this worker and the total workers.

static _receiving_subroutine(*connection_socket*)
Subroutine inside _get_np_array to receive a list of numpy arrays. If the sending was not correctly received it sends back an error message to the sender in order to try it again.

Parameters **connection_socket** – a socket with a connection already established.

Returns

static _send_np_array(*arrays_to_send*, *connection_socket*, *iteration*, *tot_workers*, *sender*, *private_key*, *receiver*, *domain*, *ip*, *list_participants=None*)
Send weights to nodes via a socket. Also write the transaction in the BSMD

Parameters

- **arrays_to_send** – weight to be send
- **connection_socket** –
- **iteration** (*int*) – iteration number in the federated process
- **tot_workers** (*int*) – total number of node in the federated process
- **sender** (*str*) – name of the node sending the information
- **private_key** (*str*) – private key of the node sending the transaction
- **receiver** (*str*) – name of the receiver
- **optional list_participants** (*array,*) – list of participants in the federated process. This variable is just need in the first loop

_start_socket_server()
Creates a socket with ssl protection that will act as server.

Returns ssl secured socket that will act as server.

_start_socket_worker()
Creates a socket with ssl protection that will act as client.

Returns ssl secured socket that will work as client.

after_create_session(*session*, *coord*)

If chief: Once the training is going to start sends it's weights to the other workers, so that they all start with the same initial ones. Once it has send the weights to all the workers it sends them a signal to start training.

Workers: Wait for the chief to send them its weights and inject them into the graph.

Parameters

- **session** –
- **coord** –

after_run(*run_context*, *run_values*)

Both chief and workers, check if they should average their weights in

this round. Is this is the case:

If chief: Tries to gather the weights of all the workers, but ignores those that lost connection at some point. It averages them and then send them back to the workers. Finally in injects the averaged weights to its own graph.

Workers: Send their weights to the chief. Wait for the chief to send them the averaged weights and inject them into their graph.

before_run(*run_context*)
Session before_run

begin()
Session begin

end(*session*)
Session end

1.3 Identification

1.3.1 User

Defines a User class.

class layers.identification.user.**User**(*private_key*, *name*, *domain*, *ip*, *public_info*)
Create an User object. This object can be use to create a passive node in the BSMD

Example

```
>>> import json
>>> from utils.administrator import Domain
>>> x = { "age": 30, "city": "New York" }
>>> public = Domain('public', 'default_role')
>>> account_information = json.dumps(x)
>>> me = User('private_key', 'My Name', public, '123.456.789', account_
↪information)
>>> print(me.name)
My name
```

Parameters

- **private_key** (*str*) – private key of the user. This is not save in the class is just used to generate a public_key. In other functions the private_key must be used to sign transactions. You can generate private keys with IrohaCrypto.private_key()
- **name** (*str*) – name of the user (lower case)

- **domain** (Domain) – domain where the user will live
- **ip** (*str*) – ip of one node hosting the blockchain
- **public_info** (*json*) – public information of the user. If domain is public this field can't be null

create_account (*private_key*)

Create a personal account in the BSMD. In the public domain all your public information is automatically populated

Example

```
>>> import json
>>> from utils.administrator import Domain
>>> x = { "age": 30, "city": "New York" }
>>> account_information = json.dumps(x)
>>> public = Domain('public', 'default_role')
>>> user = User('private_key','David', public, '123.456.789', account_
↪information)
>>> user.create_account('private_key')
```

Parameters **private_key** (*str*) – The private key of the user

create_domain (*domain*, *private_key*)

Creates a domain for personal use. You can create a domain for a particular process, e.g., Federated Learning

Example

```
>>> import json
>>> from utils.administrator import Domain
>>> x = { "age": 30, "city": "New York" }
>>> account_information = json.dumps(x)
>>> domain = Domain('name', 'default_role')
>>> user = User('private_key', 'My Name', 'My domain', '123.456.789', account_
↪information)
>>> user.create_domain(domain, 'private_key')
```

Parameters

- **domain** (Domain) – domain to be created
- **private_key** (*str*) – key to sign the transaction

get_a_detail (*detail_key*, *private_key*)

Consult a detail of the user

Example

```
>>> import json
>>> from utils.administrator import Domain
>>> x = { "age": 30, "city": "New York" }
>>> account_information = json.dumps(x)
>>> domain = Domain('name', 'default_role')
>>> user = User('private_key','David',domain, account_information)
>>> details = user.get_a_detail('private_key', 'age')
>>> print(details)
```

(continues on next page)

(continued from previous page)

```
{
    "user@domainA":{
        "Age":"35"
    }
}
```

Parameters

- **private_key** (*str*) – key to sign the transaction
- **detail_key** (*str*) – name of the detail to be consulted

Returns solicited details of the user

Return type json

get_a_detail_written_by(*user*, *detail_key*, *private_key*)

Consult a detail of the node writen by other node

Example

```
>>> import json
>>> from utils.administrator import Domain
>>> x = { "age": 30, "city": "New York" }
>>> account_information = json.dumps(x)
>>> x = { "age": 34, "city": "Mexico" }
>>> account_information_juan = json.dumps(x)
>>> domain = Domain('name', 'default_role')
>>> user = User('private_key','David',domain, account_information)
>>> juan = User('private_key_juan','Juan',domain, account_information_juan)
>>> details = user.get_a_detail_written_by(juan, 'FederatingParam',  'private_
↪key')
>>> print(details)
{
    "user@domainC":{
        "FederatingParam":"35.242553"
    }
}
```

Parameters

- **private_key** (*str*) – key to sign the transaction
- **user** (User) – user who write information on your identification
- **detail_key** (*str*) – name of the detail to be consulted

Returns solicited details of the user

Return type json

get_all_details(*private_key*)

Consult all details of the user in all the domains

Example

```
>>> import json
>>> from utils.administrator import Domain
>>> x = { "age": 30, "city": "New York" }
```

(continues on next page)

(continued from previous page)

```
>>> account_information = json.dumps(x)
>>> domain = Domain('name', 'default_role')
>>> user = User('private_key','David',domain, account_information)
>>> details = user.get_all_details('private_key')
>>> print(details)
```

Parameters **private_key** (*str*) – Key to sign the transaction

Returns solicited details of the user

Return type json

{

“user@domainA”:{ “Age”:”35”, “Name”:”Quetzalcoatl”

}, “user@domainB”:{

“Location”:”35.3333535,-45.2141556464”, “Status”:”valid”

}, “user@domainC”:{

“FederatingParam”:”35.242553”, “Loop”:”3”

}

}

get_all_details_written_by(*user*, *private_key*)

Consult all details writen by some other node

Example

```
>>> import json
>>> from utils.administrator import Domain
>>> x = { "age": 30, "city": "New York" }
>>> account_information = json.dumps(x)
>>> x = { "age": 34, "city": "Mexico" }
>>> account_information_juan = json.dumps(x)
>>> domain = Domain('name', 'default_role')
>>> user = User('private_key','David',domain, account_information)
>>> juan = User('private_key_juan','Juan',domain, account_information_juan)
>>> details = user.get_all_details_written_by(juan, 'private_key')
>>> print(details)
{
    "user@domain":{
        "FederatingParam":"35.242553",
        "Loop":"3"
    },
    "user@domain":{
        "sa_param":"44",
        "Loop":"3"
    }
}
```

Parameters

- **private_key** (*str*) – key to sign the transaction
- **user** (User) – user who write information on your identification

Returns solicited details of the user

Return type json

get_balance(*private_key*)

Get the balance of my account. Use the private key of the user to get his current balance. The function will return a dictionary with the id of the asset, the account id and the balance.

Example

```
>>> import json
>>> x = { "age": 30, "city": "New York" }
>>> account_information = json.dumps(x)
>>> user = User('private_key', 'My Name', 'My domain', '123.456.789', account_
↪information)
>>> balance = user.get_balance('private_key')
>>> print(balance)
{asset_id: "fedcoin#federated",
account_id: "generator@federated",
balance: "1000"}
```

Parameters **private_key** (*str*) – key to sign the transaction

Returns A a dictionary with the id of the asset, the account id and the balance

Return type dict

grants_access_set_details_to(*user*, *private_key*)

Grant permission to a node to set details on your identification

Example

```
>>> import json
>>> from utils.administrator import Domain
>>> x = { "age": 30, "city": "New York" }
>>> account_information = json.dumps(x)
>>> x = { "age": 34, "city": "Mexico" }
>>> account_information_juan = json.dumps(x)
>>> domain = Domain('name', 'default_role')
>>> user = User('private_key','David',domain, account_information)
>>> juan = User('private_key_juan','Juan',domain, account_information_juan)
>>> user.grants_access_set_details_to(juan, 'private_key')
```

Parameters

- **user** (User) – User you want to grant permissions to set detail on your behalf
- **private_key** (*str*) – Key to sign the transaction

revoke_access_set_details_to(*user*, *private_key*)

Revoke permission to a node to set details on your identification

Example

```
>>> import json
>>> from utils.administrator import Domain
>>> x = { "age": 30, "city": "New York" }
>>> account_information = json.dumps(x)
>>> x = { "age": 34, "city": "Mexico" }
```

(continues on next page)

(continued from previous page)

```
>>> account_information_juan = json.dumps(x)
>>> domain = Domain('name', 'default_role')
>>> user = User('private_key','David',domain, account_information)
>>> juan = User('private_key_juan','Juan',domain, account_information_juan)
>>> user.revoke_access_set_details_to(juan, 'private_key')
```

Parameters

- **user** (User) – User you want to revoke permissions to set details on your behalf
- **private_key** (*str*) – Key to sign the transaction

set_detail(*detail_key*, *detail_value*, *private_key*)

Set a detail in my account. The details can be stored in JSON format with limit of 4096 characters per detail

Example

```
>>> import json
>>> from utils.administrator import Domain
>>> x = { "gender": 30, "address": "123 Tennis" }
>>> account_information = json.dumps(x)
>>> domain = Domain('name', 'default_role')
>>> user = User('private_key','David', domain, account_information)
>>> user.set_detail('personal information', account_information, 'private_key
↪')
```

Parameters

- **detail_key** (*str*) – Name of the detail we want to set
- **detail_value** (*json*) – Value of the detail
- **private_key** (*str*) – Key to sign the transaction

set_detail_to(*user*, *detail_key*, *detail_value*, *private_key*)

Set a detail to a node. The details can be stored in JSON format with limit of 4096 characters per detail. You must have the permission from the node to set information on his identification

Example

```
>>> import json
>>> from utils.administrator import Domain
>>> x = { "age": 30, "city": "New York" }
>>> account_information = json.dumps(x)
>>> x = { "age": 34, "city": "Mexico" }
>>> account_information_juan = json.dumps(x)
>>> domain = Domain('name', 'default_role')
>>> user = User('private_key','David',domain, account_information)
>>> juan = User('private_key_juan','Juan',domain, account_information_juan)
>>> user.set_detail_to(juan, 'Job', 'Bartender', 'private_key')
```

Parameters

- **user** (User) – user you want to set the details
- **detail_key** (*str*) – Name of the detail we want to set

- **detail_value** (*str*) – Value of the detail
- **private_key** (*str*) – key to sign the transaction

transfer_assets_to(*user*, *asset_name*, *quantity*, *description*, *private_key*)
Transfer assets from one account to another. Both users must be in the same domain.

Example

```
>>> import json
>>> from utils.administrator import Domain
>>> x = { "age": 30, "city": "New York" }
>>> account_information = json.dumps(x)
>>> x = { "age": 34, "city": "Mexico" }
>>> account_information_dante = json.dumps(x)
>>> domain = Domain('name', 'default_role')
>>> user = User('private_key','David',domain, account_information)
>>> dante = User('dante_private_key','Dante',domain, account_information_
↪dante)
>>> user.transfer_assets_to(dante, 'coin', '2', 'Shut up and take my money')
```

Parameters

- **user** (User) – User you want to transfer the assets
- **asset_name** (*str*) – Name of the asset to be transferred
- **quantity** (*float*) – Number of assets we want to transfer
- **description** (*str*) – Small message to the receiver of assets
- **private_key** (*str*) – Key to sign the transaction

1.4 Incentive

1.4.1 Asset

Defines an Asset class.

class layers.incentive.asset.**Asset**(*name*, *domain*, *precision*)
Assets are created in domains and are use as cryptocurrencies

Example

```
>>> from admin.administrator import Domain
>>> public = Domain('public', 'default_role')
>>> Asset('coin', public, 3)
```

Parameters

- **name** (*str*) – name of the asset
- **domain** (Domain) – Domain object
- **precision** (*int*) – precession of the asset, e.g. precession = 3, the asset has 3 decimal points

1.5 Utils

The *Iroha* module have functions to post transactions in the BSMD and functions to node details manipulation.

The *Administrator*. module have functions to perform task such as domain, node and peer creation. This module is intended for the maintenance of the BSMD.

1.5.1 Iroha

Functions to post transactions in the iroha implementation of the BSMD

utils.iroha.**get_a_detail_written_by**(*name*, *writer*, *private_key*, *detail_key*, *domain*, *ip*)

This function can be use when the User object is no available. Consult a details of the node writen by other node

Example

```
>>> juan_detail = get_a_detail_written_by('David', 'Juan', 'private key of david',
↪ 'detail_key of Juan', 'domain',     'ip')
>>> print(juan_detail)
{
    "nodeA@domain":{
    "Age":"35"
}
```

Parameters

- **name** (*str*) – Name of the node consulting the information
- **writer** (*str*) – Name of the node who write the detail
- **private_key** (*str*) – Private key of the user
- **detail_key** (*str*) – Name of the detail we want to consult
- **domain** (*str*) – Name of the domain
- **ip** (*str*) – Address for connecting to the BSMD

Returns returns the detail writen by "the writer"

Return type json

utils.iroha.**set_detail_to_node**(*sender*, *receiver*, *private_key*, *detail_key*, *detail_value*, *domain*, *ip*)

This function can be use when the User object is no available. The sender must have permission to write in the details of the receiver.

In federated learning the details are in JSON format and contains the address (location) where the weight is stored if the weight is small enough it can be embedded to the block if needed)

Example

```
>>> set_detail_to_node('David', 'Juan', 'private key of david', 'detail key of␣
↪Juan', 'detail value', 'domain'     'ip')
```

Parameters

- **sender** (*str*) – Name of the node sending the information
- **receiver** (*str*) – Name of the node receiving the information

- **private_key** (*str*) – Private key of the user
- **detail_key** (*str*) – Name of the detail we want to set
- **detail_value** (*str*) – Value of the detail
- **domain** (*str*) – Name of the domain
- **ip** (*str*) – address for connecting to the BSMD

utils.iroha.**trace**(*func*)
A decorator for tracing methods' begin/end execution points

1.5.2 Administrator

Defines an Admin of the BSMD. This module also defines the domains

class utils.administrator.**Admin**(*ip*)
The administrator object of the BSMD. This object can create assets, add active nodes, add passive nodes and creates de public domain in the BSMD

Parameters **ip** (*str*) – ip address of one node hosting the Blockchain

Example

```
>>> Admin('123.456.789')
```

add_assets_to_user(*user*, *asset*, *asset_qty*)
The admin creates credit for a user. Users can buy credit in the BSMD to pay for services. This functions works in two step:

1. admin add assets to his own wallet
2. admin transfers the asset to the user

Example

```
>>> import json
>>> from layers.identification.user import User
>>> from layers.incentive.asset import Asset
>>> x = { "age": 30, "city": "Cartagena" }
>>> account_information = json.dumps(x)
>>> public = Domain('public', 'default_role')
>>> user = User('private_key','David',public, account_information)
>>> asset = Asset('coin', public, 3)
>>> asset.domain
>>> admin = Admin('123.456.789')
>>> admin.add_assets_to_user(user, asset, 330.2)
```

Parameters

- **user** (User) – User receiving the assets
- **asset** (Asset) – Asset to be transferred
- **asset_qty** (*float*) – Quantity of assets the node buy

create_asset(*asset*)
Creates an asset

Example

```
>>> from layers.incentive.asset import Asset
>>> public = Domain('public', 'default_role')
>>> asset = Asset('coin', public, 3)
>>> admin = Admin('123.456.789')
>>> admin.create_asset(asset)
```

Parameters **asset** (Asset) – Asset to be created

create_domain(*domain*)

Creates a domain

Example

```
>>> public = Domain('public', 'default_role')
>>> admin = Admin('123.456.789')
>>> admin.create_domain(public)
```

Parameters **domain** (Domain) – domain to be created

create_user_in_iroha(*user*)

Creates a personal account in a domain

Example

```
>>> import json
>>> from layers.identification.user import User
>>> x = { "age": 30, "city": "New York" }
>>> account_information = json.dumps(x)
>>> public = Domain('public', 'default_role')
>>> user = User('private_key','David',public, account_information)
>>> admin = Admin('123.456.789')
>>> admin.create_user_in_iroha(user)
```

Parameters **user** (User) – a user object

class utils.administrator.**Domain**(*name*, *default_role*)

The domain can be use to run distributed processes. Or to constantly share information of an specific type

Example

```
>>> Domain('public', 'default_role')
```

Parameters

- **name** (*str*) – name of the domain
- **default_role** (*str*) – default role in the domain

PYTHON MODULE INDEX

l

u

INDEX

Symbols

A

B

C

D

E

G