Read the Docs Template Documentation

Release 1.0

Read the Docs

Aug 13, 2020

Contents

CHAPTER 1

Overview

Horizen Sidechain SDK allows developers to quickly spin-up their own blockchain, customize business logic depending on use case, maintain interoperability with the mainchain native token (which acts as the medium of exchange between the whole ecosystem).

Sidechain SDK offers out-of-the-box support for the common features you'd expect from a Blockchain, but can also be easily customised and extended by developers to create a Blockchain that is tailored to their precise needs.

1.1 Tutorials - start here

For the new Sidechain developer, from installation to creating your own decentralized applications.

1.2 how-to

Practical step-by-step guides for the more experienced developer, covering several important topics.

1.3 key-topics

Explanation and analysis of some key concepts in Sidechain SDK.

1.4 Reference

Technical reference material, for classes, methods, APIs, commands.

CHAPTER 2

Join us online

Horizen Sidechain SDK is supported by a friendly and very knowledgeable community.

Join our Discord Server, and check the #sidechains channel

Our StackOverflow is for **questions** around Sidechain SDK development.

CHAPTER 3

Why Horizen Sidechains?

The first decentralized and fully customizable sidechain protocol in the industry that solves the biggest problems in applying blockchain solutions to real-world use cases.

- **A Novel Construction**

A revolutionary system of blockchains with decoupled consensus linked through common Cross-Chain Transfer Protocol (CCTP) — is indefinitely scalable, fully configurable to meet heterogeneous needs, and inclusive of embedded incentives for endogenous growth.

- **Scalability and Flexibility**

Zendoo uses a modular protocol that stresses functionality over design choice. Any type of rules can be deployed as a sidechain with this framework – whether it's a blockchain or other types of computing systems. This modularization enables massive scalability, application design freedom, and flexibility such that any component can be changed over time.

- **Decentralization**

Zendoo is decentralized in all its components. Decentralization provides resilience and reliability to the network. The Zendoo sidechain platform is fueled by a well-adopted cryptocurrency, ZEN, and supported by the largest node infrastructure in the industry. Furthermore, Zendoo doesn't rely on third parties for backward transfers, removing the need for trusted parties and honesty.

- **Privacy and Auditability**

Zendoo allows the verification of sidechains by the mainchain, without knowing the internal structure of the sidechain. Zendoo SDK provides a set of tools that will enable the creation of auditable and privacy-preserving blockchain applications, a requirement for many real-world applications.

- **Easy Deployment with the Sidechain SDK**

Zendoo comes with an SDK that includes all necessary components required for building a blockchain in a single toolbox. This allows developers to focus only on the specific features of their blockchain instead of low-level tasks, making the deployment of a complete blockchain much easier and faster.

3.1 Tutorials

The pages in this section of the documentation are aimed at the newcomer to the Horizen Sidechain SDK. They're designed to help you get started quickly, and show how easy it is to work with the sidechain SDK as a developer who wants to customize it and get it working according to their own requirements.

These tutorials take you step-by-step through some key aspects of this work. They're not intended to explain the topics in depth, or provide *reference material*, but they will leave you with a good idea of what is possible to achieve in just a few steps, and how to go about it.

Once you're familiar with the basics presented in these tutorials, you'll find the more in-depth coverage of the same topics in the How-to section.

The tutorials follow a logical progression, starting from installation of Horizen Sidechain SDK and the creation of a brand new project, and build on each other, so it's recommended to work through them in the order presented here.

3.1.1 Before you start

This tutorial offers Java developers all the needed information to build a complete blockchain application on the Horizen Sidechain system.

Apart from Java competency, this tutorial assumes that the reader has a high-level understanding of how blockchain-based distributed software works.

So, concepts such as Transactions, UTXO's, Blocks, Validation, Confirmation, Consensus, Unique chain, and chain forks, Hash Function, Private/Public key, and Signing should be known and understood, as well as the concept of a network of nodes and node communication.

If the above words are new to you, you can start by exploring the Horizen Academy website's material (link). Also, the original whitepaper by Satoshi Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System" (link), can be a good starting point. Direct experience with an existing blockchain software is also a very useful prerequisite. For that, you can install the Horizen "zend" software from (Github), and explore its rpc command interface and "regtest" mode.

Why a Sidechain?

The success of Bitcoin and of many of its successors, has led to the attempt to build more and more applications that do not require to trust a third party, not even the author of the software, to be confident that data is stored and processed according to what expected and declared. These distributed applications keep the same concept of an append-only ledger, that replaces the usual application database, that is stored and updated by the applications nodes, which also communicate to check and agree on the legitimacy of transactions, accept them and apply the relevant database updates. The success of this approach requires, among other things, that the overall system includes a robust logic to reward the app actors, so that a good enough amount of decentralization is maintained, such that any attempt of malicious behaviours bear an overwhelmingly anti-economical cost. Today, the only way to guarantee this day one, is to code the logic and data of a new application in the software that runs an existing, established blockchain supporting a traded coin. That way, the robustness of the blockchain extends to the new app, that can immediately make use of the availability of existing miners, nodes, and the coin itself.

Unfortunately, the above approach bears a scalability challenge. Blockchain's already suffer from scalability issues in their limited ability to process large volumes of transaction/time, and to accommodate sustained transaction peaks, that restrict the possibility of integrating a large number of new applications. Besides, each application logic needs to be coded in the node software, that is run by each node participating in the blockchain validation process, and this

also has an impact on scalability: the software cannot be changed and updated each time we want to add a new application, and cannot grow indefinitely.

Several attempts have been made to address these limitations; perhaps the most relevant is the idea of equipping each blockchain node with a virtual machine able to run short programs written in a specific, ad-hoc software language, e.g. Ethereum. This approach solves partially the logic scalability issue, as you don't need to change the node software each time you want to add a new application, but it brings no solution to the limited transaction throughput. Besides, the virtual machine approach typically limits the length and complexity of the application that can be supported.

The Horizen ecosystem offers a solution to the need of implementing blockchain-based distributed and decentralized applications, with all the advantages of the availability of a token that is publicly tradable, and that can be used both to rewards blockchain actors, and to support the business needs of the application itself, while solving both the scalability issues identified above. The approach is detailed in the (Zendoo whitepaper) the Horizen main blockchain, "mainchain", offers the ability to declare the existence of a sidechain, through a specific transaction, and then the possibility of sending and receiving ZEN's (the Horizen token) to and from that sidechain. There is no need to change the mainchain software each time a developer wants to implement a new application: each application will run on its own, purpose-built blockchain (a "sidechain"). This set of features, now implemented in testnet, is called "Cross-Chain Transfer Protocol", and is documented in chapter 4 of this tutorial. The Cross-Chain transfer protocol does not impose particular requirements on the sidechain architecture, as long as it supports the sidechain side of the ZEN exchange protocol.

The Horizen Sidechain SDK, offers all the basic components to build a sidechain that fully supports communication with the Horizen Mainchain. This codebase implements not only the Cross-Chain Transfer Protocol, but it also includes all the other elements needed to run a blockchain; in particular, it ships with a Proof of Stake consensus, that offers yet another scalability advantage, this time connected to the power and environmental cost of traditional Proof of Work consensus: we can scale the application logic, we can scale the number of transactions, without a big increase of wasted electricity. The architectural and protocol choices implemented by the SDK are introduced in the Zendoo whitepaper, as the "Latus" construction.

To facilitate the sidechain developers' work, the Horizen Sidechain SDK includes an example of a Sidechain Application, "SimpleApp", that just puts together all the standard components provided by the SDK, to run a basic sidechain able to receive ZEN coins from the mainchain, exchange them in sidechain, and send them back to mainchain. The SimpleApp does not add any new logic, it only configures and uses available classes and objects. Chapter 8 of this tutorial offers a detailed overview of the example, and it's a great place to start exploring the code. The next step to develop a new sidechain application, is to implement new data and logic in a sidechain node. The "Car Registry" example included in the SDK, shows how the basic components can be extended to deliver the needed functionalities. The process is documented in Chapter 9, as a step by step guide to build a custom sidechain. When that flow is clear, you'll be ready to bootstrap and run your fully distributed, decentralized blockchain, supporting your data, logic, and handling ZEN coins!

3.1.2 Installing the Sidechain SDK

We'll get started by setting up our environment.

Supported Platforms

Sidechains-SDK is available and tested on Linux and Windows (64bit).

Requirements

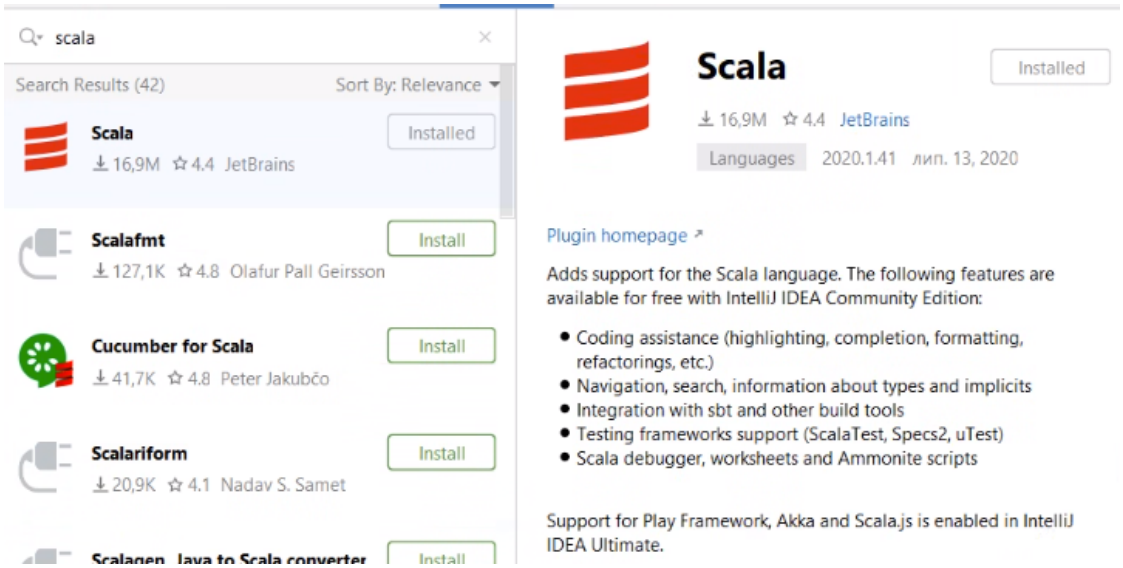
Horizen Sidechain SDK requires Java 8 or newer (Java 11 recommended), Scala 2.12.10+ or newer, and the latest version of zend_oo.

Installing on Linux OS & Windows OS:

1. Install Java JDK version 11 (link)
2. Install Scala 2.12.10+ (link)
3. Install Git (link)
4. Clone the Sidechains-SDK git repository

```
git clone git@github.com:HorizenOfficial/Sidechains-SDK.git
```

5. As IDE, please install and use IntelliJ IDEA Community Edition (link) In the IDE, please also install the Intellij Scala plugin: in the Settings->Plugins tab, select it from the marketplace:



6. In the IDE, you can now go to File and Open the root directory of the project repository, "Sidechains-SDK". The pom.xml file, the Maven's Project Object Model XML file that contains all the project configuration details should be automatically imported by the IDE. Otherwise, you can just open it.
7. Keep reading this tutorial, and start playing with the code. You will find some sidechain examples in the "examples/simpleapp" directory, that you can customize, start from there! When you are ready to run your standalone sidechain, you can install Maven (link).
8. To produce your specific sidechain jar files, you can change directory to the repository root and run the "mvn package" command.

Sidechain SDK Components:

As a result of step 8, three jar files will be generated:

- **sdk/target/Sidechains-SDK-0.2.0.jar** - The main SDK jar file that contains all the necessary classes and components

- **tools/sctool/target/Sidechains-SDK-ScBootstrappingTools-0.2.0.jar** - An executable bootstrap tool. It is used to create the configuration of the new Sidechain. You can find all available commands and examples of usage here

```
examples/simpleapp/mc_sc_workflow_example.md;
```

- **examples/simpleapp/target/Sidechains-SDK-simpleapp-0.2.0.jar** - A sidechain application example. You can find more details in the examples/simpleapp/readme.md file.

Sidechain Setup Configuration

Check the following link

3.1.3 Internal representation of a Blockchain

Being a distributed architecture, the sidechain software is meant to be delivered as a software application that will be compiled/installed by potentially many different independent, connected computers. In blockchain jargon, these computers are called "Nodes," and the term "node" is also generally used to name the blockchain software itself. So, the output of the Sidechain SDK, when customized by a developer, is a "Node" that implements core functionalities, and the added logic.

A Node consists of 4 main elements: "**History**," "**State**," "**Wallet**," and "**Memory pool**." Before we get to know these 4 elements we need to know what a "box" is.

Concept of a BOX

A box generalizes the concept of Bitcoin's UTXOs. A box is a cryptographic object that can be created with some secret keys. This box can be open (spent) by the owner of those secret keys. Once opened by the owner of the secret keys the box may not be opened again.

Node Main elements & intro to a "NodeView"

- **History** * "History" is a blockchain ledger, that is typically a list of Sidechain blocks that were received by the Node, and that have been verified against Consensus rules, and accepted.
- **State** * "State" is a snapshot of all boxes that haven't been opened yet. It represents the state at the current chain tip.
- **Wallet** * The "Wallet" has two main functionalities:
  1. It holds the Secret keys that belong to that specific Node.
  2. It keeps track of objects that are of interest to this specific node, e.g. received coins (output boxes whose secret keys are known by the node) and views of them (e.g. balances).
- **Memory Pool** * The "Memory pool" is a list of transactions that are known to the node but have not made it to a Sidechain block yet.

Altogether these 4 objects represent a "NodeView."

NodeViewHelper

All communication between NodeView objects is controlled by NodeViewHolder, which also provides a layer of communication within the application for local data processing of Blocks, Transactions, Secrets, etc.

In terms of customization, the History object is the only one that is fully controlled by the core and that in almost all circumstances does not need to be extended. It contains a ready-made implementation of the Latus consensus and of the Cross-Chain Transfer Protocol.

The core logic of State, Wallet and Memory Pool can instead be extended by sidechain developers:

- The "State" is a set of objects that are the result of processing all the previous blocks. These objects are needed to validate the next block, to allow the Node to efficiently verify, before applying a block, that all the defined rules have been respected by it. The "State" can be extended to keep track of new objects that can be useful to enforce additional rules that can be implemented in the application state interface.
- The "Wallet" can be extended through the ApplicationWallet interface, e.g. to change box ownership rules.
- The logic to accept transactions in "Memory Pool" can be also extended, e.g. transaction incompatibility rules to address possible custom data conflicts.

As mentioned before, the "Box" is an important element, as it is designed as an object that contains some data, e.g. an amount of ZEN coins, or data of a custom object (such as a car's plate as we'll see in Section 9), associated with some conditions (called "Proposition") that protect them from being spent other than by a party (or parties) able to satisfy that proposition. Usually, the ability to satisfy a Proposition is given by knowledge of some data ("called "Secret"), that can be used to produce a "Proof" that satisfies the Proposition and opens the Box, so that it can be spent.

If we translate the above into bitcoin-like terminology, a UTXO is a Box, a locking script of an output is a Proposition, e.g. a P2PK unlocking script, the signature is the proof, and its associated private key is the Secret.

Box Unique ID & Transactions

Each Box should have a unique id, which is deterministically determined using the box data as input. Since we may have several boxes locked by the same proposition, and representing the same data inside, we can avoid conflicts by using NoncedBox, which inherits Box and contains some Nonce data. Nonce data is a value that is deterministically assigned to the box depending on the Transaction that includes it, and the index of the Box inside the Transaction outputs list. This way we can guarantee that two boxes with the same data (proposition, amount and other custom fields) will have different nonces, so will have different unique box ids.

A Transaction is a sequence of inputs and outputs. Each input consists of a reference to the Box being opened, and a Proof that satisfies the condition of its Proposition. Each output is a new Box instance. Block is the only chain modifier, and it's made of header ("BlockHeader") and data ("BlockData"), similarly to the bitcoin block structure.

3.1.4 The Cross-Chain Transfer Protocol

The Cross-Chain Transfer Protocol ("CCTP") defines the communication between the mainchain and sidechain(s). It is a 2-way peg protocol that allows sending coins from mainchain to a sidechain, and vice versa.

At a high level, it defines two basic operations:

- **Forward Transfer**
- **Backward Transfer**

While all Sidechains know and follow the mainchain, which is an established and stable reality, mainchain needs to be made aware of the existence of every sidechain. So, Sidechains first must be declared in the mainchain.

We can declare a new Sidechain by using the following RPC command:

```
sc_create withdrawalEpochLength "address" amount "verification key" "vrfPublickKey"
↪"genSysConstant"
```

The command specifies where the first forward transfer coins are sent, as well as the epoch length, that defines the frequency, in blocks, of the backward transfers submissions (see the "backward transfers" paragraph below). The sc_create command also includes the cryptographic key to receive coins back from a Sidechain. The verification key guarantees that the received coins were processed according to a matching proving system. As a consequence of the sidechain declaration command, a unique sidechain id will be assigned to that sidechain, that from that moment on can be used for every operation related to that specific sidechain:

```
{
   "txid": "9e4676274f1ff9b3164de6e0d6492c4dfc1d564b0243a36208c6b7fe848f9d21",
   "scid": "2f7ed2e07ad78e52f43aafb85e242497f5a1da3539ecf37832a0a31ed54072c3",
}
```

Forward Transfer

A forward transfer sends coins from the mainchain to a sidechain. The Horizen Mainchain supports a "Forward Transfer" transaction type, that specifies the sidechain destination (*sidechain id* and *receiver address*) and the amounts of ZEN coins to be sent. From a mainchain perspective, the transferred coins are destroyed, they are only represented in the total balance of that particular sidechain. On the Sidechain side, the SDK provides all the functionalities that support Forward Transfers, so that a transferred amount is "converted" into a new Sidechain Box.

Backward Transfer

A backward transfer moves coins back from a sidechain to a mainchain destination. A Backward Transfer is initiated by a **Withdrawal Request** which is a sidechain transaction issued by the coin owners. The request specifies the mainchain destination, and the amount. More precisely, the withdrawal request owner will create a WithdrawalRequestBox that destroys the specified amount of coins in a sidechain. This is not enough to move those coins back to the mainchain though: we need to wait the end of the withdrawal epoch, when all the coins specified in that epoch's Withdrawal Requests are listed in a single Certificate, that is the propagated to the mainchain. The Certificate includes a succinct cryptographic proof that the rules associated with the declared verifying key have been respected. Certificates are processed by the mainchain consensus, which recreates the coins as specified by the certificate, only checking that the proof verifies, and that the coins received by a sidechain are not more than the amount that was sent to it.

Summary

The Cross-Chain Transfer Protocol assumes that proofs are generated with a specific proving system, but does not limit the logic of the computation that is proven by the proving system (the "circuit"). So, sidechain developers could implement the proving system that they want and need, to prove the legitimacy of backward transfers. The examples provided with the SDK implement a sample proving system, that proves that the certificate was signed by a minimum number of certifiers, whose key identities were declared at sidechain creation time. This is just a demo circuit; production sidechains require robust circuits (see the Latus recursive model in the (Zendoo paper).

3.1.5 Latus Consensus

As we have just seen, the Cross-Chain Transfer Protocol does not impose any requirements on the Sidechain architectural design other than the need to support the protocol itself. Having said that, the Horizen Sidechain SDK does offer a ready made implementation of the Latus consensus, which is a Proof of Stake ("PoS") consensus based on the Ouroboros Praos protocol.

Consensus Epochs & Forging

In Latus, the chain is split into "consensus epochs"; each epoch is made of a predefined number of time slots. Each slot is assigned to slot leaders, which are then authorized to generate ("forge") a block during that slot. So the protocol operates in a synchronous environment where each slot spans over a specific amount of time (e.g. 20 seconds). Slot leaders of a particular consensus epoch are chosen randomly before the epoch begins from the set of all sidechain forging stakeholders. The forging stake is a subset of all the coins managed by a sidechain. In fact each sidechain participant who wants to be a Forger, must have some forging stake - i.e. a set of "ForgerBoxes" assigned to him. ForgerBox is a particular kind of Box that contains an amount of coins locked for forging, and some specific data used by the forger to prove its block producing eligibility associated with that stake amount. The total amount of coins staked in ForgerBoxes is the total Forging Stake amount. The possibility of being a slot leader increases with the percentage of forging stake owned. It's possible to have more than one slot leader per slot; if more than one block is propagated, only one will be accepted by each node; the consensus rules will make sure that conflicting chains will eventually converge to a winning chain. Conversely, a consensus epoch could have empty slots, if their slot leader (or leaders) have not created and propagate blocks for them.

A slot leader eligible for a certain slot, that decides to create and propagate a new Sidechain Block for that slot, is called a "forger". A forger proves its eligibility for a slot by including in the block a cryptographic proof, in such a way that any node can validate, besides the validity of each transaction, also that the "slot leader" selection rule for that specific slot and consensus epoch was respected.

Forgers are also entitled and incentivized to include sidechain transactions and mainchain synchronization data into Sidechain Blocks. A limited amount of mainchain block data is added to sidechain blocks, in such a way that all the mainchain transactions that refer to a particular sidechain are included in that sidechain, that a reference to each mainchain block is present in all sidechains, and that information is stored in a sidechain so that any sidechain node is able to validate the mainchain block references without the need for a direct connection to the mainchain itself. Please note, the forger will need its own direct connection to mainchain nodes, to have a source of mainchain blocks data. The connection between the mainchain and sidechain nodes is established via a websocket interface provided by the mainchain node.

The Latus consensus, including mainchain block synchronization, forging logic and functionality, is implemented out-of-the-box by the core SDK, and developers do not need to make any changes to this. The forging process can be fully managed through the API interface provided by the SDK, see ("the api reference") .

Default Latus consensus parameters

- Seconds in one slot - 120, i.e. one block could be generated in two minutes
- Number of slots in one consensus Epoch - 720, i.e. new nonce is generated (and thus forging stake holder could check slot leader possability) every 720 * 120 = 86400 seconds, i.e. 24 hours.
- BlockSize Limit 2MB

3.1.6 Node communication

Communication between a user and a sidechain node is supported out of the box via HTTP POST requests API methods. Custom applications could extend them to add new, remove existing and and/or replace core behaviours.

The API configuration can be found in the sidechain configuration file.

For example see the restApi section of the following file for the SimpleApp:

```
examples/simpleapp/src/main/resources/sc_settings.conf
```

The available options are:

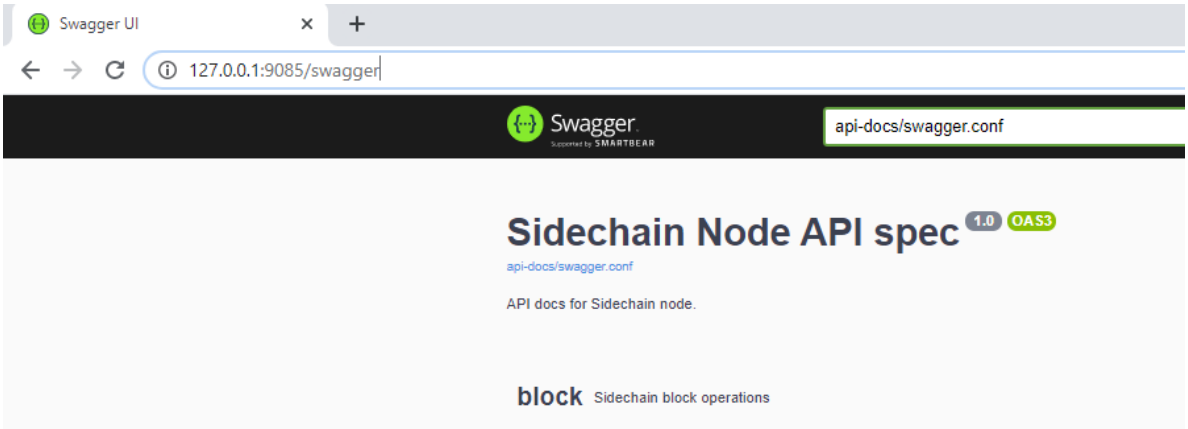
bindAddress – "IP:port" address for sending HTTP request, e.g. "127.0.0.1:9085"

api-key-hash – Authentication header must be a string that hashes to the field "api-key-hash" specified in each SC node conf file. Auth header could be empty If no api-key-hash is specified

timeout – Timeout on API requests in seconds

Note: There are many ways to send API requests to a Sidechain node (in fact any REST client could be used):

- Postman Collaboration Platform for API Development
- Embedded swagger client: Sending HTTP requests via a swagger client which is already embedded in the Sidechain Node. So you could run in your browser "IP:port" as defined in your configuration file, and select any of the commands shown there. For example:



Default standard API

Base API is organized in the following 5 groups:

- Block – Sidechain block operations like find best blockId, find blockId by block height etc. Also here you could find forging related commands like starting/stopping automatically forging, get information about forging like last epoch and slot index. Automatic forging gets current time to convert it into appropriate slot/epoch index, thus if by some reason a Sidechain node skip's the correct timeslot for whole consensus epoch when forging in automatic mode will always fail. Also, a Sidechain will be considered as deceased, as described before, i.e. communication between Sidechain and mainchain is no longer possible. However forging a block with manual set epoch/slot index is possible by API call /block/generate, it could be useful in case if Sidechain is run in isolated mode.
- Transaction – Sidechain transaction operations like find all transactions, create a transaction, without sending into memory pool, send transaction into memory pool, etc.
- Wallet – Sidechain wallet operations. Wallet operation could take optional parameter boxType for example in /wallet/balance API request. Box type could take as parameter RegularBox, ForgerBox etc., i.e. you could type here class name for required box type (in case of custom box type you oblige to use fully qualified class name). If box type is not matter then just omit that parameter, i.e. in case of /wallet/balance just use an empty body.
- Node –Sidechain node operations like connect to the node, see all connections, etc.
- Mainchain– Sidechain mainchain operations like get the best MC header included in Sidechain.

3.1.7 Base App

Sidechain SDK provides to the developers an out of the box implementation of the Latus Consensus Protocol and the Crosschain Transfer Protocol. Additionally to this, the SDK provides basic transactions, network layer, data storage and node configuration, as well as entry points for any custom extension.

Secret / Proof / Proposition

- **Sidechain SDK** uses its own terms for secret key / public key / signed message and provides various types of them.
- **Secret** - Private key
- **Proposition** - Public key, used in boxes as a locker
- **Proof** - Signed message
- SDK provides the following implementations for Secret / Proof / Proposition
  - **Curve 25519**
    - PrivateKey25519
    - PublicKey25519Proposition
    - Signature25519
  - **VRF based on ginger-lib**
    - VrfSecretKey
    - VrfPublicKey
    - VrfProof
  - **Schnorr based on ginger-lib**
    - SchnorrSecret
    - SchnorrPropostion
    - SchnorrProof

Boxes

Data in a sidechain is meant to be represented as a Box, that we can see as data kept "closed" by a Proposition, that can be open only with the Proposition's Secret(s). The Sidechain SDK offers two different Box types: Coin Box and non-Coin Box. A Non-Coin box represents a unique entity that can be transferred between different owners. A Coin box is a box that contains ZEN, examples of a Coin box are RegularBox and ForgingBox. A Coin Box can add custom data to an object that represents some coins, i.e., that it holds an intrinsic defined value. For example, a developer would extend a Coin Box to manage a time lock on a UTXO, e.g., to implement smart contract logic. In particular, any box can be split into two parts: Box and BoxData (box data is included in the Box). The Box itself represents the entity in the blockchain, i.e., all operations such as create/open are performed on boxes. Box data contains information about the entity like value, proposition address, and any custom data.

Every Box has its unique boxId (not be confused with box type id, which is used for serialization). That box id is calculated for each Box by the following function in the SDK core:

```
public final byte[] id() {
   if(id == null) {
       id = Blake2b256.hash(Bytes.concat(
               this instanceof CoinsBox ? coinsBoxFlag : nonCoinsBoxFlag,
               Longs.toByteArray(value()),
               proposition().bytes(),
               Longs.toByteArray(nonce()),
               boxData.customFieldsHash()));
   }
   return id;
}
```

Note: The id is used during transaction verification, so it is important to add custom data into customFieldsHash() function.

The following Coin-Box types are provided by SDK:

- **RegularBox** – contains ZEN coins
- **ForgerBox** – contains ZEN coins are used for forging
- **WithdrawalRequestBox** – contain ZEN coins are used to backward transfer, i.e. move coins back to the mainchain.

An SDK developer can declare custom Boxes, please refer to SDK extension section.

Transactions

There are two basic transactions: MC2SCAggregatedTransaction and SidechainCoreTransaction. An MC2SCAggregatedTransaction is the implementation of Forward Transfer and can only be added as a part of the mainchain block reference data during synchronization with the mainchain. When a Forger is going to produce a sidechain block, and a new mainchain block appears, the forger will recreate that mainchain block as a reference that will contain sidechain related data. If a Forward Transfer exists in the mainchain block, it will be included into the MC2SCAggregatedTransaction and added as a part of the reference. The SidechainCoreTransaction is the transaction, which can be created by anyone to send coins inside a sidechain, create forging stakes or perform withdrawal requests (send coins back to the MC). The SidechainCoreTransaction can be extended to support custom logic operations. For example, if we think about real-estate sidechain, we can tokenize some private property as a specific Box using SidechainCoreTransaction. Please refer to SDK extensions for more details.

Serialization

Because the SDK is based on Scorex we implement the Scorex way of data serialization.

- Any serialized data like Box/BoxData/Secret/Proof/Transaction implements Scorex BytesSerializable interface/trait.
- BytesSerializable declare functions byte[] bytes() and Serializer serializer().
- Serializer itself works with Reader/Writer, which are wrappers on byte stream.
- Scorex Reader and Writer also implements functionality like reading/parsing data of integer/long/string etc.
- Serialization and parsing itself implemented in data class by implementation `byte[] bytes()` (required by BytesSerializable interface) and implementation static function for parsing bytes `public static Data parseBytes(byte[] bytes)`

- Also, for correct parse purposes, special bytes such as a **unique id** of data type are put at the beginning of the byte stream (it is done automatically). Thus any serialized data shall provide a unique id. Specific serializers shall be set for those unique ids during the dependency injection setting as well as custom Serializer shall be put into Custom Serializers Map, which are defined at AppModule. Please refer to the SDK extension section for more information

SidechainNodeView

SidechainNodeView is a provider to current Node state including NodeWallet, NodeHistory, NodeState, NodememoryPool and application data as well. SidechainNodeView is accessible during custom API implementation.

Memory Pool

A mempool is a node's mechanism for storing information on unconfirmed transactions. It acts as a sort of waiting room for transactions that have not yet been included in a block

Node wallet

Contains available private keys, required for generating correct proofs

State

Contains information about current node state

History

Provide access to history, i.e. blocks not only from active chain but from forks as well.

Network layer

The network layer can be divided into communication between Nodes and communication between the node and user. Node interconnection is organized as a peer-to-peer network. Over the network, the SDK handles the handshake, blockchain synchronization, and transaction transmission.

Physical storage

Physical storage. The SDK introduces the unified physical storage interface, this default implementation is based on the LevelDB library. Sidechain developers can decide to use the default solution or to provide the custom one. For example, the developer could decide to use encrypted storage, a Key Value store, a relational database or even a cloud solution. In case of your own implementation, please make sure that Storage test passes for your custom storage.

User specific settings

The user can define custom configuration options, such as a specific path to the node data storage, wallet seed, node name and API server address/port. To do this, he should write into the configuration file in a HOCON notation. The configuration file consists of the SDK required fields and application custom fields if needed. Sidechain developers can use com.horizen.settings.SettingsReader utility class to extract Sidechain specific data and Config object itself to get custom parts.

```
class SettingsReader {
    public SettingsReader (String userConfigPath, Optional<String>␣
↪applicationConfigPath)

    public SidechainSettings getSidechainSettings()

    public Config getConfig()
}
```

Moreover, if a specific sidechain contains general application settings that should be controlled only by the developer, it is possible to define basic application config that can be passed as an argument to SettingsReader.

SidechainApp class

The starting point of the SDK for each sidechain is the SidechainApp class. Every sidechain application should create an instance of SidechainApp with passing all required parameters and then execute the sidechain node flow:

```
class SidechainApp {
        public SidechainApp(
                // Settings:
                SidechainSettings sidechainSettings,

                // Custom objects serializers:
                HashMap<> customBoxSerializers,
                HashMap<> customBoxDataSerializers,
                HashMap<> customSecretSerializers,
                HashMap<> customTransactionSerializers,

                // Application Node logic extensions:
                ApplicationWallet applicationWallet,
                ApplicationState applicationState,

                // Physical storages:
                Storage secretStorage,
                Storage walletBoxStorage,
                Storage walletTransactionStorage,
                Storage stateStorage,
                Storage historyStorage,
                Storage walletForgingBoxesInfoStorage,
                Storage consensusStorage,

                // Custom API calls and Core API endpoints to disable:
                List<ApplicationApiGroup> customApiGroups,
                List<Pair<String, String>> rejectedApiPaths
        )

        public void run()
}
```

The SidechainApp instance can be instantiated directly or through Guice DI library. Binding by Guice could be done in the following ways:

```
bind(injected_classType)
        .annotatedWith(Names.named("Injected_parameter_name"))
        .toInstance(injected_variable_name);
```

or

```
bind(new TypeLiteral<injected_classType>() {})
       .annotatedWith(Names.named("Injected_parameter_name"))
       .toInstance(injected_variable_name);
```

In the following table, we describe used injections and their description. While injected injected_classType and "Injected_parameter_name" shall be used as it described in table, injected_variable_name could be differrent

We can split SidechainApp arguments into 4 groups:

1. **Settings**
    - The instance of SidechainSettings is retrieved by custom application via SettingsReader, as was described above.
2. **Custom objects serializers**
    - Developers will want to add their custom business logic. For example, tokenization of real-estate properties will be required to create custom Box and BoxData types. These custom objects must be somehow managed by SDK to be sent through the network or stored to the disk. In both cases, SDK should know how to serialize a custom object to bytes and how to restore it. To maintain this, sidechain developers should specify custom objects serializers and add them to custom...Serializer map following the specific rules (Data Serialization Section)
3. **Application node extension of State and Wallet logic**
    - As was said above, State is a snapshot of all closed boxes of the blockchain at some moment. So when the next block arrives, the ApplicationState validates the block to prevent the spending of non-existing boxes or transaction inputs and outputs coin balances inconsistency. Developers can extend State by introducing additional logic in ApplicationState and ApplicationWallet. See appropriate sections.
4. **API extension** - link
5. **Node communication** link

Inside the SDK, we implemented a SimpleApp example designed to demonstrate the basic SDK functionalities. It is the fastest way to get started with our SDK. SimpleApp has no custom logic: no custom boxes and transactions, no custom API, and an empty ApplicationState and ApplicationWallet.

The SimpleApp requires a single argument to start: the path to the user configuration file. Under the hood, it has to parse its config file using SettingsReader, and then initialize and run SidechainApp.

3.1.8 Sidechains SDK extension

Data serialization

Any data like **Box/BoxData/Secret/Proposition/Proof/Transaction** shall provide a way to serialize itself to bytes and provide a way to parse it from bytes. Serialization is performed via a special Serializer class. Any custom data needs to define its own Serializer and definition of parsing/serializing and needs to declare those Serializers for the SDK. Thus SDK will be able to use proper Serializer for custom data. The steps to describe serialization/parsing for some CustomData are the following:

- Implement BytesSerializable interface for *CustomData*, i.e. `functions byte[] bytes()` and `Serializer serializer()` (which shall return CustomDataSerializer), also implement `public static CustomData parseBytes(byte[] bytes)` function for parsing from bytes
- Create `CustomDataSerializer` and implement `ScorexSerializer interface`, i.e. functions `void serialize(CustomData customData, Writer writer)` and `CustomData parse(Reader reader);`

- Provide a unique id for that data type by implementing a special function. List of data type and appropriate functions is next:

Data type / Base class	Function to be overridden
interface Box	byte boxTypeId()
interface NoncedBoxData	byte boxDataTypeId()
interface Proof	byte proofTypeId()
interface Secret	byte secretTypeId()
abstract class BoxTransaction	byte transactionTypeId()

- In your AppModule class (i.e. class which extends `AbstractModule`, in SimpleApp it is `SimpleAppModule`) define Custom Serializer map, for example for boxes it could be `Map<Byte, BoxSerializer<Box<Proposition>>> customBoxSerializers = new HashMap<>();` where key is data type id and value is CustomSerializer for those data type id.
- Add your custom serializer into the map, for example it could be something `like customBoxSerializers.put((byte)MY_CUSTOM_BOX_ID, (BoxSerializer) CustomBoxSerializer.getSerializer());`
- Bind map with custom serializers to your application in the app model class:

```
TypeLiteral<HashMap<Byte, Common serializer type>() {})
      .annotatedWith(Names.named(Bound property name))
      .toInstance(Created map with custom serializers);
```

Where **Common serializer type** and **Bound property name** can have the following values

Bound property name	Common serializer type
CustomBoxSerializers	BoxSerializer<Box<Proposition>>>
CustomBoxDataSerializers	NoncedBoxDataSerializer<NoncedBoxData <Proposition, NoncedBox<Proposition>>>
CustomSecretSerializers	SecretSerializer<Secret>>
CustomProofSerializers	ProofSerializer<Proof<Proposition>>
CustomTransactionSerializers	TransactionSerializer<BoxTransaction <Proposition, Box<Proposition>>>

Example:

```
bind(new TypeLiteral<HashMap<Byte, BoxSerializer<Box<Proposition>>>>() {})
     .annotatedWith(Names.named("CustomBoxSerializers"))
     .toInstance(customBoxSerializers);
```

Where

- **BoxSerializer<Box<Proposition>>>** – common serializer type
- **"CustomBoxSerializers"** – bound property name
- **customBoxSerializers** – created map with all defined custom serializers.

Custom box creation

a) SDK Box extension Overview

To build a real application, a developer will need more to do more than receive, transfer, and send coins back. A distributed app, built on a sidechain, will typically have to define some custom data that the sidechain users will be

able to exchange according to a defined logic. The creation of new Boxes requires the definition of four new classes. We will use the name Custom Box as a definition for some abstract custom Box:

Class type	Class description
Custom Box Data class	– Contains all custom data definitions plus proposition for Box – Provide required information for serialization of Box Data – Define the way for creation new Custom Box from current Custom Box Data
Custom Box Data Serializer Singleton	– Define the way how to parse bytes from Reader into Custom Box Data object – Define the way how to put boxData object into Writer Parsing function used in a Serializer class can be put in that class as well. However, it can be defined somewhere else
Custom Box	Representation new entity in Sidechain, contains appropriate Custom Box Data class
Custom Box Serializer Singleton	– Define the way how to parse bytes from Reader into Box object – Define the way how to put boxData object into Writer Parsing function used in a Serializer class can be put in that class as well. However, it can be defined somewhere else

Custom Box Data class creation

The SDK provides base class for any Box Data class:

```
AbstractNoncedBoxData<P extends Proposition, B extends AbstractNoncedBox<P, BD, B>,␣
↪BD extends AbstractNoncedBoxData<P, B, BD>>
```

where

`P extends Proposition` – Proposition type for the box, for common purposes `PublicKey25519Proposition` can be used as it used in regular boxes

`BD extends AbstractNoncedBoxData<P, B, BD>` – Definition of type for Box Data which contains all custom data for a new custom box

`B extends AbstractNoncedBox<P, BD, B>` – Definition of type for Box itself, required for description inside of new Custom Box data

That base class provides the following data by default:

```
proposition of type P long value
```

If the box type is a Coin-Box then this value is required and will contain data such as coin value. In the case of a Non-Coin box it will be used in custom logic only. As a common practice for non-Coin box you can set it always equal to 1

So the creation of new Custom Box Data will be created in the following way:

```
public class CustomBoxData extends AbstractNoncedBoxData<PublicKey25519Proposition,␣
↪CustomBox, CustomBoxData>
```

The new custom box data class requires the following:

1. Custom data definition

- Custom data itself
- Hash of all added custom data shall be returned in "public byte[] customFieldsHash() "function, otherwise, custom data will not be "protected," i.e., some malicious actor can change custom data during transaction creation.

2. Serialization definition

- Serialization to bytes shall be provided by Custom Box Data by overriding and implementing the method `public byte[] bytes()` this function serializes the proposition, value and any added custom data.

- Additionally definition of Custom Box Data id for serialization by overriding `public byte boxDataTypeId()` function, please check the serialization section for more information about using ids.
- Override `public NoncedBoxDataSerializer serializer()` function with proper **Custom Box Data serializer**. Parsing Custom Box Data from bytes can be defined in that class as well, please refer to the serialization section for more information about it

3. Custom Box creation

- Any Box Data class shall provide the way how to create a new Box for a given nonce. For that purpose override the function

```
public CustomBox getBox(long nonce)
```

Custom Box Data Serializer class creation

The SDK provides a base class for Custom Box Data Serializer `NoncedBoxDataSerializer<D extends NoncedBoxData>` where **D** is type of serialized Custom Box Data

So creation of a Custom Box Data Serializer can be done in following way:

```
public class CustomBoxDataSerializer implements NoncedBoxDataSerializer<CustomBoxData>
```

The new Custom Box Data Serializer require's the following:

1. Definition of function for writing Custom Box Data into the Scorex Writer by implementation of the following method.

```
public void serialize(CustomBoxData boxData, Writer writer)
```

2. Definition of function for reading Custom Box Data from Scorex *Reader* by implementation of the function

```
public CustomBoxData parse(Reader reader)
```

3. Class shall be converted to singleton, for example it can be done in following way:

```
private static final CustomBoxDataSerializer serializer = new␣
↪CustomBoxDataSerializer();

private CustomBoxDataSerializer() {
super();
}

public static CustomBoxDataSerializer getSerializer() {
return serializer;
}
```

Custom Box class creation

The SDK provides a base class for creation of a Custom Box:

```
public class CustomBox extends AbstractNoncedBox<PublicKey25519Proposition,␣
↪CustomBoxData, CustomBoxBox>
```

As parameters for **AbstractNoncedBox** three template parameters shall be provided:

```
P extends Proposition
```

- Proposition type for the box, for common purposes. `PublicKey25519Proposition` could be used as it used in regular boxes

```
BD extends AbstractNoncedBoxData<P, B, BD>
```

- Definition of type for Box Data which contains all custom data for a new custom box

```
B extends AbstractNoncedBox<P, BD, B>
```

- Definition of type for Box itself, required for description inside of new Custom Box data.

The Custom Box itself requires implementation of following functionality:

1. Serialization definition

- The box itself provides the way to be serialized into bytes, thus function `public byte[] bytes()` shall be implemented
- Method for creation of a new Car Box object from bytes `public static CarBox parseBytes(byte[] bytes)`
- Providing box type id by implementation of function `public byte boxTypeId()` which return custom box type id. Finally, proper serializer for the Custom Box shall be returned by implementing function `public BoxSerializer serializer()`

Custom Box Serializer Class

SDK provide base class for Custom Box Serializer BoxSerializer<B extends Box> where B is type of serialized Custom Box So the creation of Custom Box Serializer can be done in the following way:

```
public class CustomBoxSerializer implements NoncedBoxSerializer<CustomBox>
```

The new Custom Box Serializer requires the following:

1. Definition of function for writing *Custom Box* into the *Scorex Writer* by implementation of the following.

```
public void serialize(CustomBox box, Writer writer)
```

2. Definition of function for reading *Custom Box* from *Scorex Reader* by implementation of the following method

```
public CustomBox parse(Reader reader)
```

3. Class shall be converted to singleton, for example it could be done in following way:

```
private static final CustomBoxSerializer serializer = new CustomBoxSerializer();
private CustomBoxSerializer() {
 super();
}
public static CustomBoxSerializer getSerializer() {
 return serializer;
}
```

Specific actions for extension of Coin-box

A Coin box is created and extended as a usual non-coin box, only one additional action is required: *Coin box class* shall also implement interface `CoinsBox<P extends PublicKey25519Proposition>` interface without any additional function implementations, i.e., it is a mixin interface.

Transaction extension

A transaction in the SDK is represented by the following class.

```
public abstract class BoxTransaction<P extends Proposition, B extends Box<P>>
```

This class provides access to data such as which boxes will be created, unlockers for input boxes, fees, etc. SDK developer could add custom transaction check by implementing *custom ApplicationState*

Any custom transaction shall implement three important functions: `public boolean transactionSemanticValidity()` – this function defines is transaction semantically valid or not, i.e. verify stateless (without context) transaction correctness. Non zero fee and positive timestamp are examples of such verification.

`public List<BoxUnlocker<Proposition>> unlockers()` – SDK core does box opening verification by checking proofs against input box ids. However, information about closed boxes and proofs for that box shall be returned separately by each transaction. For such purposes each transaction shall return a list of unlockers which are implements following interface:

```
public interface BoxUnlocker<P extends Proposition>
{
  byte[] closedBoxId();
  Proof<P> boxKey();
}
```

Where *closedBoxId* is the id of the closed box and *boxKey* is correct proof for that box.

`public List<NoncedBox<Proposition>> newBoxes()` – function returns list of new boxes which shall be created by current transaction. Be aware due to some internal implementation of SDK that function must be implemented in the following way:

```
@Override
public List<NoncedBox<Proposition>> newBoxes() {
  if(newBoxes == null) {
  //new boxes are created here, newBoxes shall be updated by those new boxes
      }
  }
  return Collections.unmodifiableList(newBoxes);
}
```

Custom Proof / Proposition creation

A proposition is a locker for a box, and Proof is an unlocker for a box. For some reason, a way how the box is locked/unlocked can be changed by the SDK developer. For example, a special box can be opened by two or more independent private keys. For such reason, custom Proof / Proposition can be created.

- Creating custom Proposition For creating a custom Proposition `ProofOfKnowledgeProposition<S extends Secret>` interface shall be implemented. Generic parameter is just a marker for the type of private key, for example, *PrivateKey25519* It could be used. Inside the Proposition, we could put two different public keys, which are used for locking the box.

- Creating custom Proof interface `Proof<P extends Proposition>` shall be implemented where *P* is an appropriate Proposition class. `Function boolean isValid(P proposition, byte[] messageToVerify);` shall be implemented. That function defines whether Proof is valid for a given proposition and Proof or not. For example, in the case of Proposition with two different public keys, we could try to verify the message using public keys in Proposition one by one and return true if Proof had been created by one of the expected private keys.

ApplicationState and Wallet

ApplicationState:

```
interface ApplicationState {
boolean validate(SidechainStateReader stateReader, SidechainBlock block);

boolean validate(SidechainStateReader stateReader, BoxTransaction<Proposition, Box
↪<Proposition>> transaction);

Try<ApplicationState> onApplyChanges(SidechainStateReader stateReader, byte[] version,
↪ List<Box<Proposition>> newBoxes, List<byte[]> boxIdsToRemove);

Try<ApplicationState> onRollback(byte[] version);
}
```

For example, the custom application may have the possibility to tokenize cars by the creation of Box entries - let us call them CarBox. Each CarBox token should represent a unique car by having a unique *VIN* (Vehicle Identification Number). To do this, Sidechain developer may define ApplicationState to store the list of actual VINs and reject transactions with CarBox tokens with VIN already existing.

The next custom state checks could be done here:

- `public boolean validate(SidechainStateReader stateReader, SidechainBlock block)` – any custom block validation could be done here. If the function returns false, then the block will not be accepted by the Sidechain Node.
- `public boolean validate(SidechainStateReader stateReader, BoxTransaction<Proposition, Box<Proposition>> transaction)` – any custom checks for the transaction could be done here if the function returns false then transaction is assumed as invalid and for example will not be included in a memory pool.
- `public Try<ApplicationState> onApplyChanges(SidechainStateReader stateReader, byte[] version, List<Box<Proposition>> newBoxes, List<byte[]> boxIdsToRemove)` – any specific action after block applying in State could be defined here.
- `public Try<ApplicationState> onRollback(byte[] version)` – any specific action after rollback of State (for example in case of fork/invalid block) could be defined here

Application Wallet

The Wallet by default keeps user secret info and related balances. The actual data is updated when a new block is applied to the chain or when some blocks are reverted. Developers can specify custom secret types that will be processed by Wallet. The developer may extend the logic using ApplicationWallet

```
interface ApplicationWallet {
  void onAddSecret(Secret secret);
  void onRemoveSecret(Proposition proposition);
  void onChangeBoxes(byte[] version, List<Box<Proposition>> boxesToUpdate, List
↪<byte[]> boxIdsToRemove);
```

(continues on next page)

(continued from previous page)

```
  void onRollback(byte[] version);
}
```

For example, a developer needs to have some event-based data, like an auction slot that belongs to him, and will start in 10 blocks and expire in 100 blocks. So in ApplicationWallet, he will additionally keep this event-based info and will react when a new block is going to be applied (onChangeBoxes method execution) to activate or deactivate that slot in ApplicationWallet.

Custom API creation

Steps to extend the API:

1. Create a class (e.g. MyCustomApi) which extends the ApplicationApiGroup abstract class (you could create multiple classes, for example to group functions by functionality).
2. In a class where all dependencies are declared (e.g. SimpleAppModule in our Simple App example) we need to create the following variable: `List<ApplicationApiGroup> customApiGroups = new ArrayList<>();`
3. Create a new instance of the class MyCustomApi, and then add it to *customApiGroups*

At this point, MyCustomApi will be included in the API route, but we still need to declare the HTTP address. To do that:

1. Override the basepath() method -

```
public String basePath() {
 return "myCustomAPI";
}
```

Where "myCustomAPI" is part of the HTTP path for that API group

2. Define HTTP request classes – i.e. the json body in the HTTP request will be converted to that request class. For example, if as "request" we want to use byte array data with some integer value, we could define the following class:

```
public static class MyCustomRequest {
 byte[] someBytes;
 int number;

public byte[] getSomeBytes(){
 return someBytes;
}

public void setSomeBytes(String bytesInHex){
 someBytes = BytesUtils.fromHexString(bytesInHex);
}

public int getNumber(){
 return number;
}

public void setNumber(int number){
this.number = number;
}
}
```

Setters are defined to expect data from JSON. So, for the given MyCustomRequest we could use next JSON:

```
{
"number": "342",
"someBytes":
↪"a5b10622d70f094b7276e04608d97c7c699c8700164f78e16fe5e8082f4bb2ac"
}
```

And it will be converted to an instance of the *MyCustomRequest* class with vin = 342, and someBytes = bytes which are represented by hex string "a5b10622d70f094b7276e04608d97c7c699c8700164f78e16fe5e8082f4bb2ac"

3. Define a function to process the HTTP request: Currently we support three types of function's signature:

   - ApiResponse `custom_function_name(Custom_HTTP_request_type)` – a function that by default does not have access to *SidechainNodeView*. To have access to *SidechainNodeViewHolder*, this special call should be used: `getFunctionsApplierOnSidechainNodeView().applyFunctionOnSidechainNodeView(Function<SidechainNodeView, T> function)`
   - `ApiResponse custom_function_name(SidechainNodeView, Custom_HTTP_request_type)` – a function that offers by default access to SidechainNodeView
   - `ApiResponse custom_function_name(SidechainNodeView)` – a function to process empty HTTP requests, i.e. JSON body shall be empty

Inside those functions, all required action could be defined, and with them also function response results. Responses could be based on *SuccessResponse* or *ErrorResponse* interfaces. The JSON response will be formatted by using the defined getters.

4. Add response classes

As a result of an API request, the result shall be sent back via HTTP response. In a typical case, we could have two different responses: operation is successful, or some error had appeared during processing the API request. SDK provides following way to declare those API responses: For a successful response, implement SuccessResponse interface with data to be returned. That data shall be accessible via getters. Also, that class shall have the next annotation required for marshaling and correct conversion to JSON: `@JsonView(Views.Default.class)`. The developer can define here some other custom class for JSON marshaling. For example, if a string should be returned, then the following response class can be defined:

```
@JsonView(Views.Default.class)
class CustomSuccessResponce implements SuccessResponse{
private final String response;

public CustomSuccessResponce (String response) {
this.response = response;
}

public String getResponse() {
return response;
}
}
```

In such case API response will be represented in the following JSON format:

```
{"result": {"response" : "response from CustomSuccessResponse object"}}
```

In case of something going wrong and error shall be returned then response shall implement ErrorResponse interface which by default have next functions to be implemented:

`public String code()` – error code

`public String description()` – error description

`public Option<Throwable> exception()` – Caught exception during API processing

As a result next JSON will be returned in case of error:

```
{
"error": {
"code": "Defined error code",
"description": "Defined error description",
"Detail": "Exception stack trace"
}
}
```

5. Add defined route processing functions to route

   Override `public List<Route> getRoutes() function` by returning all defined routes, for example:

```
List<Route> routes = new ArrayList<>();
routes.add(bindPostRequest("getNSecrets",␣
↪this::getNSecretsFunction, GetSecretRequest.class));
routes.add(bindPostRequest("getNSecretOtherImplementation",␣
↪this::getNSecretOtherImplementationFunction,␣
↪GetSecretRequest.class));
routes.add(bindPostRequest("getAllSecretByEmptyHttpBody",␣
↪this::getAllSecretByEmptyHttpBodyFunction));
return routes;
```

   Where

   `getNSecrets`, `getNSecretOtherImplementation`, `getAllSecretByEmptyHttpBody` are defined API end points; `this::getNSecretsFunction`, `this::getNSecretOtherImplementationFunction`, `getAllSecretByEmptyHttpBodyFunction` binded functions;

`GetSecretRequest.class` – class for defining type of HTTP request

3.1.9 Car Registry Tutorial

Car Registry App high level overview

The Car Registry app is an example of a sidechain that implements specific custom data and logic. The purpose of the application is to manage a simplified service that keeps records of existing cars and their owners. It is simplified as sidechain users will be able to register cars by merely paying a transaction fee. In contrast, in a real-world scenario, the ability to create a car will be bound by the presentation of a certificate signed by the Department of Motor Vehicles or analogous authority, or some other consensus mechanism that guarantees that the car exists in the real world and it's owned by a user with a given public key. Accepting that cars will show up in sidechain in our example, we want to build an application that can store information that identifies a specific car, such as vehicle identification number, model, production year, color. We will also want car owners to prove their ownership of the cars without disclosing information about their identity. We also want users to sell and buy cars against ZEN coins.

User stories:

1 **Q: I want to add my car to a Car Registry Sidechain.**

A: Create a new Car Entry Box, which contains car identification information (Unique car identifier, VIN, manufacturer, model, year, registration number), and certificate. Proposition in this box is my public key in this Sidechain. When I create a box, Sidechain should check car identification information and certificate to be unique in this Sidechain.

2 **Q: I want to create sell order to sell my car using Car Registry Sidechain.**

A: I create a new Car Sell Order Box that contains the price in coins and information from the Car Entry Box. So cars can exist in the Sidechain as a Car Entry Box or as a Car Sell Order, but not at the same time. Also, this box contains the buyer's public key. When I create a sell order, Sidechain should check if there is no other active sell order with this Car Entry Box. The current Sell Order consists of the same information that consists of the Car Entry Box plus description.

3 **Q: I want to see all available Sell orders in Sidechain**

A: Have additional storage, which is managed by ApplicationState and stores all Car Sell Orders. All these orders can be retrieved using the new HTTP API call.

4 **Q: I want to accept a sell order and buy the car.**

A: By accepting sell order, I create a new transaction in the Sidechain, which creates a new Car Entry Box with my public key as Proposition and transfers coins amount from me to the previous car owner.

5 **Q: I want to cancel my Car Sell Order.**

A: I create a new transaction containing Car Sell Order as input and Car Entry Box with my public key as Proposition as output.

6. **Q: I want to see my car entry boxes and car sell orders related to me (both created by me and proposed to me).**

A: Implement new storage that will be managed by the application state to store this information. Implement a new HTTP API, that contains a new method to get this information.

So, the starting point of the development process is the data representation. A car is an example of a non-coin box because it represents some entity, but not money. Another example of a non-coin box is a car that is selling. We need another box for a selling car because a standard car box does not have additional data like sale price, seller proposition address, etc. For the money representation, standard Regular Box is used (Regular box is coin box), SDK provides that box. Besides new entities CarBox and CarSellOrder, we also need to define a way to create/destroy those new entities. For that purpose, new transactions shall be defined: transaction for creating a new car, a transaction that moves CarBox to CarSellOrder, transaction which declares car selling, i.e., moving CarSellOrder to the new CarBox. All created transactions are not automatically put into the memory pool, so a raw transaction in hex representation shall be put by /transaction/sendTransaction API request. In summary, we will add the next car boxes and transactions:

Entity name	Entity description	Entity fields
CarBox	Box which contains car box data, which could be stored and operated in Sidechain	boxData – contains car box data
CarBoxData	Description of the car by using defined properties	vin – vehicle identification number which contains unique identification number of the car year – vehicle year production model – car model color – car color
CarSellOrderBox	Box which contains car sell order data, which could be stored and operated in Sidechain.	boxData – contains car sell order data
CarSellOrderBoxData	Description of the car which is in sell status. That box data contains a special type of proposition SellOrderProposition. That proposition allows us to spent the box in two different ways: by seller and by buyer	vin – vehicle identification number which contains unique identification number of the car year – vehicle year production model – car model color – car color
CarSellOrderInfo	Information about car's selling as well as proof of a current car owner. Used in transaction processing.	carBoxToOpen – car box for start selling proof – proof for open initial car box price – selling price buyerProposition – current implementation expect to have the specific buyer which had been found off chain. Thus during creation of car sell order we already know buyer and shall put his future car proposition
CarBuyOrderInfo	Data required for buying a car or recall a car sell order. Used in transaction processing.	carSellOrderBoxToOpen – Car sell order box to be open proof – specific proof of type SellOrderSpendingProof for confirming buying of the car or recall car sell order

Special proposition and proof:

a) **SellOrderProposition** The standard proposition only contains one public key, i.e., only one specific secret key could open that proposition. However, for a sell order, we need a way to open and spend the box in two different ways, so we need to specify an additional proposition/proof. SellOrderProposition contains two public keys:

```
ownerPublicKeyBytes
```

and

```
buyerPublicKeyBytes
```

So the seller or buyer's private keys could open that proposition.

b) **SellOrderSpendingProof** The proof that allows us to open and spend

```
CarSellOrderBox
```

in two different ways: opened by the buyer and thus buy the car or opened by the seller and thus recall car sell order. Such proof creation requires two different API calls, but as a result, in both cases, we will have the same type of transaction with the same proof type.

Transactions:

AbstractRegularTransaction

Base custom transaction, all other custom transactions extend this base transaction.

Input parameters are:

`inputRegularBoxIds` - list of regular boxes for payments like fee and car buying `inputRegularBoxProofs` - appropriate list of proofs for box opening for each regular box in `inputRegularBoxIds` `outputRegularBoxesData` - list of output regular boxes, used as the change from paying a fee, as well as a new regular box for payment for the car. `fee` - transaction fee `timestamp` - transaction timestamp

Output boxes:

Regular Boxes created by change or car payment

CarDeclarationTransaction

Transaction for declaring a car in the Sidechain, this transaction extends `AbstractRegularTransaction` thus some base functionality already is implemented.

Input parameters are:

`inputRegularBoxIds` – list of regular boxes for payments like fee and car buying `inputRegularBoxProofs` – appropriate list of proofs for box opening for each regular box in inputRegularBoxIds `outputRegularBoxesData` – list of output regular boxes, used as change from paying a fee, as well as a new regular box for car payment. `fee` – transaction fee `timestamp` – transaction timestamp `outputCarBoxData` – box data which contains information about a new car.

Output boxes:

New CarBox with new declared car

SellCarTransaction

Transaction to initiate the selling process of the car.

Input parameters are:

`inputRegularBoxIds` - list of regular boxes for payments like fee and car buying `inputRegularBoxProofs` - appropriate list of proofs for box opening for each regular box in inputRegularBoxIds `outputRegularBoxesData` - list of output regular boxes, used as change from paying fee, as well as new regular box for payment for car. `fee` – transaction fee `timestamp` - transaction timestamp `carSellOrderInfo` - information about car selling, including such information as car description and specific proposition `SellOrderProposition`.

Output boxes:

CarSellOrderBox, which represents the car to be sold, that box could be opened by the initial car owner or specified buyer in case if a buyer buys that car.

BuyCarTransaction

This transaction allows us to buy a car or recall a car sell order.

Input parameters are:

`inputRegularBoxIds` - list of regular boxes for payments like fee and purchasing the car `inputRegularBoxProofs` - appropriate list of proofs for box opening for each regular box in inputRegularBoxIds `outputRegularBoxesData` - list of output regular boxes, used as change from paying fee, as well as a new regular box for payment for the car. `fee` - transaction fee `timestamp` - transaction timestamp `carBuyOrderInfo` - information for buy car or recall car sell order.

Output boxes:

Two possible outputs are possible. In the case of buying a car, new CarBox with a new owner, a new Regular box with a value declared in carBuyOrderInfo for the Car's former owner.

Car registry implementation

First of all, we need to define new boxes. As described before, a Car Box is a non-coin box as defined before we need Car Box Data class to describe custom data. So we need to define CarBox and CarBoxData as separate classes for setting proper way to serialization/deserialization.

Implementation of CarBoxData:

CarBoxData is implemented according description from `Custom Box Data Creation` section as `public class CarBoxData extends AbstractNoncedBoxData<PublicKey25519Proposition, CarBox, CarBoxData>` with custom data as:

```
private final BigInteger vin;
private final int year;
private final String model;
private final String color;
```

Few comments about implementation:

1. @JsonView(Views.Default.class) is used during class declaration. That annotation allows SDK core to do proper JSON serialization.
2. Serialization is implemented in public byte[] bytes() function as well as parsing implemented in public static CarBoxData parseBytes(byte[] bytes) function. SDK developer, as described before, shall include proposition and value into serialization/deserialization. The order doesn't matter.
3. CarBoxData shall have a value parameter as a Scorex limitation, but in our business logic, CarBoxData does not use that data at all because each car is unique and doesn't have any inherent value. Thus value is hidden, i.e., value is not present in the constructor parameter and just set by default to "1" in the class constructor.
4. `public byte[] customFieldsHash()` shall be implemented because we introduce some new custom data.

Implementation of CarBoxDataSerializer:

`CarBoxDataSerializer` is implemented according to the description from `Custom Box Data Serializer Creation` section as `public class CarBoxDataSerializer implements NoncedBoxDataSerializer<CarBoxData>`.

Implementation of CarBox:

`CarBox` is implemented according to description from `Custom Box Class creation` section as `public class CarBox extends AbstractNoncedBox<PublicKey25519Proposition, CarBoxData, CarBox>`

Few comments about implementation:

1. As a serialization part SDK developer shall include `long nonce` as a part of serialization, thus serialization is implemented in the following way:

```
public byte[] bytes()
{
    return Bytes.concat(
        Longs.toByteArray(nonce),
        CarBoxDataSerializer.getSerializer().toBytes(boxData)
    );
}
```

2. `CarBox` defines his own unique id by implementation of the function `public byte boxTypeId()`. Similar function is defined in `CarBoxData` but it is a different ids despite value returned in `CarBox` and `CarBoxData` is the same.

Implementation of CarBoxSerializer:

A CarBoxSerializer is implemented according to the description from the ("Custom Box Data Serializer Creation section") as

```
public class CarBoxSerializer implements BoxSerializer<CarBox>
```

Implementation of SellOrderProposition

A SellOrderProposition is implemented as

```
public final class SellOrderProposition implements ProofOfKnowledgeProposition
↪<PrivateKey25519>
```

A point to note is that the proposition contains two public keys, thus that proposition could be opened by two different keys.

Implementation of SellOrderPropositionSerializer

A SellOrderPropositionSerializer is implemented as

```
public final class SellOrderPropositionSerializer implements PropositionSerializer
↪<SellOrderProposition>
```

Implementation of SellOrderSpendingProof

A SellOrderSpendingProof is implemented as

```
extends AbstractSignature25519<PrivateKey25519, SellOrderProposition>
```

Implementation Comments: Information about proof type is defined by the result of method boolean isSeller(). For example an implementation of method isValid uses that flag:

```
public boolean isValid(SellOrderProposition proposition, byte[] message) {
 if(isSeller) {
  // Car seller wants to discard selling.
  return Ed25519.verify(signatureBytes, message, proposition.
↪getOwnerPublicKeyBytes());
 } else {
  // Specific buyer wants to buy the car.
  return Ed25519.verify(signatureBytes, message, proposition.
↪getBuyerPublicKeyBytes());
 }
}
```

Implementation of CarSellOrderBoxData

A CarSellOrderBoxData is implemented according to the description from the ("Custom Box Data class creation section") as

```
public class CarSellOrderData extends AbstractNoncedBoxData<SellOrderProposition,␣
↪CarSellOrderBox, CarSellOrderBoxData>
```

with custom data as:

```
private final String vin;
private final int year;
private final String model;
private final String color;
```

Few comments about implementation: Proposition and value shall be included in serialization as it done in CarBoxData Id of that box data could be different than in CarBoxData CarSellOrderBoxData uses custom proposition type, thus *proposition* field have *SellOrderProposition* type

Implementation of CarSellOrderBoxDataSerializer

A CarSellOrderDataSerializer is implemented according to the description from the ("Custom Box Data Serializer creation section") as

```
public class CarSellOrderBoxDataSerializer implements NoncedBoxDataSerializer
↪<CarSellOrderData>
```

Implementation of CarSellOrderBox

A CarSellorder is implemented according to description from the ("Custom Box Class creation section") as

```
public final class CarSellOrderBox extends AbstractNoncedBox<SellOrderProposition,␣
↪CarSellOrderBoxData, CarSellOrderBox>
```

AbstractRegularTransaction

AbstractRegularTransaction is implemented as

```
public abstract class AbstractRegularTransaction extends SidechainTransaction
↪<Proposition, NoncedBox<Proposition>>
```

Basic functionality is implemented for building required unlockers for input Regular boxes and returning a list of output Regular boxes according to input parameter *outputRegularBoxesData*. Also, basic transaction semantic validity is checked here.

CarDeclarationTransaction

CarDeclarationTransaction extends previously declared *AbstractRegularTransaction* in the following way: `public final class CarDeclarationTransaction extends AbstractRegularTransaction` newBoxes() – a new box with a newly created car shall be added as well. Thus that function shall be overridden as well for adding new CarBox additional to regular boxes.

SellCarTransaction

SellCarTransaction extends previously declared AbstractRegularTransaction in next way: `public final class SellCarTransaction extends AbstractRegularTransaction` Similar to *CarDeclarationTransaction, newBoxes()* function shall also return a new specific box. In our case that new box is *CarSellOrderBox*. Also due we have specific box to open (CarBox), we also need to add unlocker for CarBox, so unlocker for that CarBox had been added in `public List<BoxUnlocker<Proposition>> unlockers()`

BuyCarTransaction

Few comments about implementation: During the creation of unlockers in function *unlockers()*, we need to also create a specific unlocker for opening a car sell order. Another *newBoxes()* function has a bit specific implementation. That function forces to create a new RegularBox as payment for a car in case the car has been sold. Anyway, a new Car box also shall be created according to information in `carBuyOrderInfo`.

Extend API:

- Create a new class CarApi which extends ApplicationApiGroup class, add that new class to Route by it in SimpleAppModule, like described in Custom API manual. In our case it is done in `CarRegistryAppModule` by
  - Creating `customApiGroups` as a list of custom API Groups:
  - `List<ApplicationApiGroup> customApiGroups = new ArrayList<>()``;`
  - Adding created `CarApi` into `customApiGroups:` `customApiGroups.add(new CarApi());`
  - Binding that custom api group via dependency injection:

```
bind(new TypeLiteral<List<ApplicationApiGroup>> () {})
        .annotatedWith(Names.named("CustomApiGroups"))
        .toInstance(customApiGroups);
```

- Define Car creation transaction.
  - Defining request class/JSON request body As input for the transaction we expected: Regular box id as input for paying fee; Fee value; Proposition address which will be recognized as a Car Proposition; Vehicle identification number of car. So next request class shall be created:

```
public class CreateCarBoxRequest {
public String vin;
public int year;
public String model;
public String color;
public String proposition; // hex representation of public key proposition
public long fee;

// Setters to let Akka jackson JSON library to automatically deserialize the␣
↪request body.
    public void setVin(String vin) {
        this.vin = vin;
    }

    public void setYear(int year) {
        this.year = year;
    }

    public void setModel(String model) {
        this.model = model;
    }

    public void setColor(String color) {
        this.color = color;
    }

    public void setProposition(String proposition) {
        this.proposition = proposition;
    }

    public void setFee(long fee) {
        this.fee = fee;
    }
 }
```

Request class shall have appropriate setters and getters for all class members. Class members' names define a structure for related JSON structure according to Jackson library, so next JSON structure is expected to be set:

```
{
   "vin":"30124",
   "year":1984,
   "model": "Lamborghini"
   "color":"deep black"
   "carProposition":"a5b10622d70f094b7276e04608d97c7c699c8700164f78e16fe5e8082f4bb2ac
↪",
   "fee": 1,
   "boxId": "d59f80b39d24716b4c9a54cfed4bff8e6f76597a7b11761d0d8b7b27ddf8bd3c"
}
```

Few notes: setter's input parameter could have a different type than set class member. It allows us to make all necessary conversation in setters.

- Define response for Car creation transaction, the result of transaction shall be defined by implementing Success-

Response interface with class members which shall be returned as API response, all members shall have properly set getters, also response class shall have proper annotation `@JsonView(Views.Default.class)` thus jackson library is able correctly represent response class in JSON format. In our case, we expect to return transaction bytes, so response class is next:

```
@JsonView(Views.Default.class)
class TxResponse implements SuccessResponse {
public String transactionBytes;
    public TxResponse(String transactionBytes) {
        this.transactionBytes = transactionBytes;
    }
}
```

- Define Car creation transaction itself

```
private ApiResponse createCar(SidechainNodeView view, CreateCarBoxRequest ent)
```

As a first parameter we pass reference to SidechainNodeView, second reference is previously defined class on step 1 for representation of JSON request.

- Define request for Car sell order transaction CreateCarSellOrderRequest similar as it was done for Car creation transaction request
  - Define request class for Car sell order transaction CreateCarSellOrderRequest as it was done for Car creation transaction request:

```
public class CreateCarSellOrderRequest {
 public String carBoxId; // hex representation of box id
 public String buyerProposition; // hex representation of public key␣
↪proposition
 public long sellPrice;
 public long fee;

 // Setters to let Akka jackson JSON library to automatically deserialize the␣
↪request body.

 public void setCarBoxId(String carBoxId) {
     this.carBoxId = carBoxId;
 }

 public void setBuyerProposition(String buyerProposition) {
     this.buyerProposition = buyerProposition;
 }

 public void setSellPrice(long sellPrice) {
     this.sellPrice = sellPrice;
 }

 public void setFee(int fee) {
     this.fee = fee;
 }
}
```

- Define Car Sell order transaction itself – `private ApiResponse createCarSellOrder(SidechainNodeView view, CreateCarSellOrderRequest ent)` Required actions are similar as it was done to Create Car transaction. The main idea is a moving Car Box into CarSellOrderBox.
- Define Car sell order response – As a result of Car sell order we could still use TxResponse

- **Create AcceptCarSellorder transaction**
  - Specify request as

```
public class SpendCarSellOrderRequest {
 public String carSellOrderId; // hex representation of box id
 public long fee;
 // Setters to let Akka jackson JSON library to automatically deserialize␣
↪the request body.
 public void setCarSellOrderId(String carSellOrderId) {
 this.carSellOrderId = carSellOrderId;
 }

 public void setFee(long fee) {
 this.fee = fee;
 }
}
```

  - Specify acceptCarSellOrder transaction itself
  - As a result we still could use TxResponse class
  - **Important part is creation proof for BuyCarTransaction, because we accept car buying then we shall form proof**

```
SellOrderSpendingProof buyerProof = new SellOrderSpendingProof(
buyerSecretOption.get().sign(messageToSign).bytes(),
isSeller
);
```

Where *isSeller* is false.

- **Create cancelCarSellOrder transaction**
  - Specify cancel request as

```
public class SpendCarSellOrderRequest {
    public String carSellOrderId; // hex representation of box id
    public long fee;

    // Setters to let Akka jackson JSON library to automatically␣
↪deserialize the request body.

    public void setCarSellOrderId(String carSellOrderId) {
        this.carSellOrderId = carSellOrderId;
    }

    public void setFee(long fee) {
        this.fee = fee;
    }
}
```

  - Specify transaction itself. Because we recall our sell order then isSeller parameter during transaction creation is set to false.

Either way, you'll be able to find support and help from the numerous friendly members of the Horizen community, on our Discord channel `#sidechains`

3.2 Reference

3.2.1 Sidechain Node API spec

Sidechain Block operations

`POST /block/findById`

Find Block by ID

Parameters

Name	Type	Required	Description
blockId	String	yes	Find block by ID

query boolean active return only active versions

query boolean built return only built versions

Example request:

Bash

curl -X POST "http://127.0.0.1:9085/block/findById" -H "accept: application/json" -H "Content-Type: application/json" -d "{"blockId":"0...6"}"

Example response:

```
HTTP/1.1 200 OK
Vary: Accept
Content-Type: text/javascript

{
   "result":{
      "blockHex":"string",
      "block":{
         "id":"string",
         "parentId":"string",
         "timestamp":0,
         "mainchainBlocks":[
            {
               "header":{
                  "mainchainHeaderBytes":"string",
                  "version":0,
                  "hashPrevBlock":"string",
                  "hashMerkleRoot":"string",
                  "hashReserved":"string",
                  "hashSCMerkleRootsMap":"string",
                  "time":0,
                  "bits":0,
                  "nonce":"string",
                  "solution":"string"
               },
```

(continues on next page)

(continued from previous page)


```
                "sidechainRelatedAggregatedTransaction":{
                   "id":"string",
                   "fee":0,
                   "timestamp":0,
                   "mc2scTransactionsMerkleRootHash":"string",
                   "newBoxes":[
                      {
                         "id":"string",
                         "proposition":{
                            "publicKey":"string"
                         },
                         "value":0,
                         "nonce":0,
                         "activeFromWithdrawalEpoch":0,
                         "typeId":0
                      }
                   ]
                },
                "merkleRoots":[
                   {
                      "key":"string",
                      "value":"string"
                   }
                ]
             }
          ],
          "sidechainTransactions":[
             {

             }
          ],
          "forgerPublicKey":{
             "publicKey":"string"
          },
          "signature":{
             "signature":"string"
          }
       }
    },
    "error":{
       "code":"string",
       "description":"string",
       "detail":"string"
    }
}
```


`POST /block/findLastIds`

Returns an array with the ids of the last x blocks

Parameters

Name	Type	Required	Description
number	int	yes	Retrieves the last x number of blocks

Example request:

Bash

curl -X POST "http://127.0.0.1:9085/block/findLastIds" -H "accept: application/json" -H "Content-Type: application/json" -d "{"number":10}"

Example response:

```
{
   "result":{
      "lastBlockIds":[
         "055c15d9a6c9ae299493d241705a2bcfdfbc72a19f04394a26aa53b39f6ee2a6",
         "ae6bcf104b7a7cccf83dfa23494760fb8d9a4d5cc3de82443de8b82bb86669d1",
         "9120b0f8518d1944d4b0e8fac8990acc7dcb792ea660414906a03f346407160c",
         "e5b0e97df9502c9510e4862041754b62931c9dc0a4fa873b3a0d75561dcbe712",
         "6a080e3ee665980bf647b450749b04177fe272537808bb4aec70417f9994bd04",
         "97d1956ecb1199fe03171b0923dff4031850e33db56dd1afc3b5384350315d80",
         "2c3a4a91989110218a827f8baefa3a8e5baf33e7e16d32b2bdace94553478dde",
         "cf82fba3e75ac89ca7e8d1c29458b2d5eff9d807407d3265c14251da2c70b3b1",
         "d61da61b2c877f717fa50563a42cbad4420486bfa3b1f05d888528d69d8258d8",
         "921f9406d8edd03d2f5b65aa6f89e452720c7ef07244ee06f3ad19d2c49e45d8"
      ]
   }
}
```

`POST /block/findIdByHeight`

Return a sidechain block Id by its height in a blockchain

Parameters

Name	Type	Required	Description
height	int	yes	Retrieves block ID by it´s height

Example request:

Bash

curl -X POST "http://127.0.0.1:9086/block/findIdByHeight" -H "accept: application/json" -H "Content-Type: application/json" -d "{"height":100}"

Example response:

```
{
   "result":{
      "blockId":
↪"e8c92a6c217a7dced190b729a7815f0be6a011ea23a38e083e79298bb66620e7"
   }
}
```

`POST /block/best`

Return here best sidechain block id and height in active chain

No Parameters

Example request:

Bash

curl -X POST "http://127.0.0.1:9086/block/best" -H "accept: application/json"

Example response:


```
{
  "result": {
    "block": {
      "header": {
        "version": 1,
        "parentId":
↪"ae6bcf104b7a7cccf83dfa23494760fb8d9a4d5cc3de82443de8b82bb86669d1",
        "timestamp": 1595475730,
        "forgerBox": {
          "nonce": -8596034112114319000,
          "id":
↪"f290e648415642b051cf6075b5fcaa7609eddd9a919d144cc2062db632918d9e",
          "typeId": 3,
          "vrfPubKey": {
            "valid": true,
            "publicKey":
↪"d984ea8909760cb69d0a1a13848bd534e9ac28ec0ac20c3b05d557fa6512405185d799d1bab96068ad903a8f72e08
↪"
          },
          "blockSignProposition": {
            "publicKey":
↪"153623a54522cc0336068a305ac13f530f4fdc95ee105a7ee85939326b9996fb"
          },
          "value": 10000000000,
          "proposition": {
            "publicKey":
↪"153623a54522cc0336068a305ac13f530f4fdc95ee105a7ee85939326b9996fb"
          }
        },
        "forgerBoxMerklePath": "00000000",
        "vrfProof": {
          "vrfProof":
↪"6be4253461faa494c5b79befbd12a39d73bf80c8c0d4b004bb72b49d0203fee1880057100dec12d4fbaf49e304798
↪"
        },
        "sidechainTransactionsMerkleRootHash":
↪"0000000000000000000000000000000000000000000000000000000000000000",
        "mainchainMerkleRootHash":
↪"0000000000000000000000000000000000000000000000000000000000000000",
        "ommersMerkleRootHash":
↪"0000000000000000000000000000000000000000000000000000000000000000",
        "ommersCumulativeScore": 0,
        "signature": {
          "signature":
↪"2c5e2d784bdb46ab07a9958152605a363931fa2794c714169e054667ef615f176be20a8db5a8dc40f02daca3d6684
↪",
          "typeId": 1
        },
        "id":
↪"055c15d9a6c9ae299493d241705a2bcfdfbc72a19f04394a26aa53b39f6ee2a6"
      },
      "sidechainTransactions": [],
      "mainchainBlockReferencesData": [],
      "mainchainHeaders": [],
```


(continues on next page)

(continued from previous page)

```
      "ommers": [],
      "timestamp": 1595475730,
      "parentId":
↪"ae6bcf104b7a7cccf83dfa23494760fb8d9a4d5cc3de82443de8b82bb86669d1",
      "id": "055c15d9a6c9ae299493d241705a2bcfdfbc72a19f04394a26aa53b39f6ee2a6
↪"
    },
    "height": 371
  }
}
```

`POST /block/startForging`

Start forging

No Parameters

Example request:

Bash

curl -X POST "http://127.0.0.1:9086/block/startForging" -H "accept: application/json"

Example response:

```
{
  "result": {
    "result": "string"
  },
  "error": {
    "code": "string",
    "description": "string",
    "detail": "string"
  }
}
```

`POST /block/stopForging`

Stop forging

No Parameters

Example request:

Bash

curl -X POST "http://127.0.0.1:9086/block/stopForging" -H "accept: application/json"

Example response:

```
{
  "result": {
    "result": "string"
  },
  "error": {
    "code": "string",
    "description": "string",
    "detail": "string"
```

(continues on next page)

(continued from previous page)

```
  }
}
```

`POST /block/generate`

Try to generate new block by epoch and slot number Returns id of generated sidechain block

Parameters

Name	Type	Required	Description
epochNumber	int	yes	Epoch Number
slotNumber	int	yes	Slot Number

Example request:

Bash

curl -X POST "http://127.0.0.1:9086/block/generate" -H "accept: application/json" -H "Content-Type: application/json" -d "{"epochNumber":3,"slotNumber":45}"

Example response:

```
{
  "result": {
    "blockId":
↪"7f25d35aadae65062033757e5049e44728128b7405ff739070e91d753b419094"
  }
}
```

`POST /block/forgingInfo`

Get forging info

No Parameters

Example request:

Bash

curl -X POST "http://127.0.0.1:9086/block/forgingInfo" -H "accept: application/json"

Example response:

```
{
  "result": {
    "consensusSecondsInSlot": 120,
    "consensusSlotsInEpoch": 720,
    "bestEpochNumber": 3,
    "bestSlotNumber": 45
  }
}
```

Sidechain Transaction operations

`POST /transaction/allTransactions`

Find all transactions in the memory pool

Parameters

Name	Type	Required	Description
format	boolean	no	Returns an array of transaction ids if formatMemPool=false, otherwise a JSONObject for each transaction

Example request:

Bash

curl -X POST "http://127.0.0.1:9087/transaction/allTransactions" -H "accept: application/json" -H "Content-Type: application/json" -d "{"format":true}"

Example response:

```
{
  "result": {
    "transactions": []
  }
}
```

`POST /transaction/findById`

- *blockHash set -> Search in block referenced by blockHash (do not care about txIndex parameter)*
- *blockHash not set, txIndex = true -> Search in memory pool, if not found, search in the whole blockchain*
- *blockHash not set, txIndex = false -> Search in memory pool*

Parameters

Name	Type	Description
transactionId	String	Find by Transaction Id
blockHash	String	Search in block referenced by blockHash (do not care about txIndex parameter)
transactionIndex	boolean	txIndex = true -> Search in memory pool, if not found, search in the whole blockchain
format	boolean	

Example request:

Bash

curl -X POST "http://127.0.0.1:9087/transaction/findById" -H "accept: application/json" -H "Content-Type: application/json" -d "{"transactionId":"string","blockHash":"string","transactionIndex":false,"format":false}"

Example response:

```
{
  "result": {
    "transaction": {},
```

(continues on next page)

(continued from previous page)

```
    "transactionBytes": "string"
  },
  "error": {
    "code": "string",
    "description": "string",
    "detail": "string"
  }
}
```

`POST /transaction/decodeTransactionBytes`

Return a JSON representation of a transaction given its byte serialization

Parameters

Name	Type	Required	Description
transactionBytes	String	yes	byte String

Example request:

Bash

curl -X POST "http://127.0.0.1:9087/transaction/decodeTransactionBytes" -H "accept: application/json" -H "Content-Type: application/json" -d "{"transactionBytes":"string"}"

Example response:

```
{
  "result": {
    "transaction": {}
  },
  "error": {
    "code": "string",
    "description": "string",
    "detail": "string"
  }
}
```

`POST /transaction/createCoreTransaction`

Create and sign a Sidechain core transaction, specifying inputs and outputs. Return the new transaction as a hex string if format = false, otherwise its JSON representation.

Parameters

Example Value

```
{
  "transactionInputs": [
    {
      "boxId": "string"
    }
  ],
  "regularOutputs": [
    {
```

(continues on next page)

(continued from previous page)

```
      "publicKey": "string",
      "value": 0
    }
  ],
  "withdrawalRequests": [
    {
      "publicKey": "string",
      "value": 0
    }
  ],
  "forgerOutputs": [
    {
      "publicKey": "string",
      "blockSignPublicKey": "string",
      "vrfPubKey": "string",
      "value": 0
    }
  ],
  "format": false
}
```

Example request:

Bash

curl -X POST "http://127.0.0.1:9087/transaction/createCoreTransaction" -H "accept: application/json" -H "Content-Type: application/json" -d "{"transactionInputs":[{"boxId":"string"}],"regularOutputs":[{"publicKey":"string","value":0}],"withdrawalRequests":[{"publicKey":"string","value":

Example response:

```
{
  "result": {
    "transaction": {},
    "transactionBytes": "string"
  },
  "error": {
    "code": "string",
    "description": "string",
    "detail": "string"
  }
}
```

`POST /transaction/createCoreTransactionSimplified`

Create and sign a Sidechain core transaction, specifying inputs and outputs. Return the new transaction as a hex string if format = false, otherwise its JSON representation.

Parameters

Example Value

```
{
  "regularOutputs": [
    {
      "publicKey": "string",
      "value": 0
```

(continues on next page)

(continued from previous page)

```
    }
  ],
  "withdrawalRequests": [
    {
      "publicKey": "string",
      "value": 0
    }
  ],
  "forgerOutputs": [
    {
      "publicKey": "string",
      "blockSignPublicKey": "string",
      "vrfPubKey": "string",
      "value": 0
    }
  ],
  "fee": 0,
  "format": true
}
```

Example request:

Bash

curl -X POST "http://127.0.0.1:9087/transaction/createCoreTransactionSimplified" -H "accept: application/json" -H "Content-Type: application/json" -d "{"regularOutputs":[{"publicKey":"string","value":0}],"withdrawalRequests":[{"publicKey":"string","value":0}],"forgerOutputs":[{"publicKey":"st

Example response:

```
{
  "result": {
    "transaction": {},
    "transactionBytes": "string"
  },
  "error": {
    "code": "string",
    "description": "string",
    "detail": "string"
  }
}
```

`POST /transaction/sendCoinsToAddress`

Create and sign a regular transaction, specifying outputs and fee. Then validate and send the transaction. Then return the id of the transaction

Parameters

Example Value

```
{
  "outputs": [
    {
      "publicKey": "string",
      "value": 0
    }
```

(continues on next page)

(continued from previous page)

```
  ],
  "fee": 0
}
```

Example request:

Bash

curl -X POST "http://127.0.0.1:9087/transaction/sendCoinsToAddress" -H "accept: application/json" -H "Content-Type: application/json" -d "{"outputs":[{"publicKey":"string","value":0}],"fee":0}"

Example response:

```
{
  "result": {
    "transactionId": "string"
  },
  "error": {
    "code": "string",
    "description": "string",
    "detail": "string"
  }
}
```

`POST /transaction/withdrawCoins`

Create and sign a regular transaction, specifying withdrawal outputs and fee. Then validate and send the transaction. Then return the id of the transaction

Parameters

```
{
  "outputs": [
    {
      "publicKey": "string",
      "value": 0
    }
  ],
  "fee": 0
}
```

Example request:

Bash

curl -X POST "http://127.0.0.1:9087/transaction/withdrawCoins" -H "accept: application/json" -H "Content-Type: application/json" -d "{"outputs":[{"publicKey":"string","value":0}],"fee":0}"

Example response:

```
{
  "code": 0,
  "reason": "string",
  "detail": "string"
}
```

`POST /transaction/makeForgerStake`

Create and sign a Sidechain core transaction, specifying forger stake outputs and fee. Then validate and send the transaction. Then return the id of the transaction

Parameters

Example Value


```
{
  "outputs": [
    {
      "publicKey": "string",
      "blockSignPublicKey": "string",
      "vrfPubKey": "string",
      "value": 0
    }
  ],
  "fee": 0
}
```


Example request:

Bash

curl -X POST "http://127.0.0.1:9087/transaction/makeForgerStake" -H "accept: application/json" -H "Content-Type: application/json" -d "{"outputs":[{"publicKey":"string","blockSignPublicKey":"string","vrfPubKey":"string","value":0}],"fee":0}"

Example response:


```
{
  "result": {
    "transactionId": "string"
  },
  "error": {
    "code": "string",
    "description": "string",
    "detail": "string"
  }
}
```

`POST /transaction/spendForgingStake`

Create and sign sidechain core transaction, specifying inputs and outputs. Return the new transaction as a hex string if format = false, otherwise its JSON representation.

Parameters

Example Value


```
{
  "transactionInputs": [
    {
      "boxId": "string"
    }
  ],
  "regularOutputs": [
    {
      "publicKey": "string",
      "value": 0
    }
```


(continues on next page)

(continued from previous page)

```
  ],
  "forgerOutputs": [
    {
      "publicKey": "string",
      "blockSignPublicKey": "string",
      "vrfPubKey": "string",
      "value": 0
    }
  ],
  "format": false
}
```

Example request:

Bash

curl -X POST "http://127.0.0.1:9087/transaction/spendForgingStake" -H "accept: application/json" -H "Content-Type: application/json" -d "{"transactionInputs":[{"boxId":"string"}],"regularOutputs":[{"publicKey":"string","value":0}],"forgerOutputs":[{"publicKey":"string","blockSignPub

Example response:

```
{
  "result": {
    "transaction": {},
    "transactionBytes": "string"
  },
  "error": {
    "code": "string",
    "description": "string",
    "detail": "string"
  }
}
```

`POST /transaction/sendTransaction`

Validate and send a transaction, given its serialization as input. Then return the id of the transaction

Parameters

Name	Type	Description
transactionBytes	String	Signed Transaction Bytes

Example request:

Bash

curl -X POST "http://127.0.0.1:9087/transaction/sendTransaction" -H "accept: application/json" -H "Content-Type: application/json" -d "{"transactionBytes":"string"}"

Example response:

```
{
  "result": {
    "transactionId": "string"
  },
```

(continues on next page)

(continued from previous page)

```
  "error": {
    "code": "string",
    "description": "string",
    "detail": "string"
  }
}
```

Sidechain Wallet Operations

`POST /wallet/allBoxes`

Return all boxes, excluding those which ids are included in excludeBoxIds list

Parameters

Example Value

```
{
  "boxTypeClass": "string",
  "excludeBoxIds": [
    "string"
  ]
}
```

Example request:

Bash

curl -X POST "http://127.0.0.1:9086/wallet/allBoxes" -H "accept: application/json" -H "Content-Type: application/json" -d "{"boxTypeClass":"string","excludeBoxIds":["string"]}"

Example response:

```
{
  "result": {
    "boxes": [
      {
        "id": "string",
        "proposition": {
          "publicKey": "string"
        },
        "value": 0,
        "nonce": 0,
        "activeFromWithdrawalEpoch": 0,
        "typeId": 0
      }
    ]
  },
  "error": {
    "code": "string",
    "description": "string",
    "detail": "string"
  }
}
```

`POST /wallet/balance`

Return the global balance for all types of boxes

Parameters

Name	Type	Required	Description
boxType	String	No	Box type

Example request:

Bash

curl -X POST "http://127.0.0.1:9086/wallet/balance" -H "accept: application/json" -H "Content-Type: application/json" -d "{"boxType":"string"}"

Example response:

```
{
  "result": {
    "balance": 0
  },
  "error": {
    "code": "string",
    "description": "string",
    "detail": "string"
  }
}
```

`POST /wallet/createPrivateKey25519`

Create new secret and return corresponding address (public key)

No Parameters

Example request:

Bash

curl -X POST "http://127.0.0.1:9086/wallet/createPrivateKey25519" -H "accept: application/json"

Example response:

```
{
  "result": {
    "proposition": {
      "publicKey": "string"
    }
  },
  "error": {
    "code": "string",
    "description": "string",
    "detail": "string"
  }
}
```

`POST /wallet/createVrfSecret`

Create new Vrf secret and return corresponding public key

No Parameters

Example request:

Bash

curl -X POST "http://127.0.0.1:9086/wallet/createVrfSecret" -H "accept: application/json"

Example response:


```
{
  "result": {
    "proposition": {
      "valid": true,
      "publicKey":
↪"ef3df0e2ca6f34dc89c2c14e23aecd37370ec4739230a6ec640a1fc87857ee5e7f55f3784e5ddd3c8e733bcdefb67
↪"
    }
  }
}
```

`POST /wallet/allPublicKeys`

Returns the list of all wallet's propositions (public keys)

Parameters

Name	Type	Description
protoype	String	

Example request:

Bash

curl -X POST "http://127.0.0.1:9086/wallet/allPublicKeys" -H "accept: application/json" -H "Content-Type: application/json" -d "{}"

Example response:


```
{
  "result": {
    "propositions": [
      {
        "publicKey": "string"
      }
    ]
  },
  "error": {
    "code": "string",
    "description": "string",
    "detail": "string"
  }
}
```

Sidechain node operations

`POST /node/allPeers`

Returns the list of all sidechain node peers

No Parameters

Example request:

Bash

curl -X POST "http://127.0.0.1:9086/node/allPeers" -H "accept: application/json"

Example response:

```
{
  "result": {
    "peers": [
      {
        "address": "string",
        "lastSeen": 0,
        "name": "string",
        "connectionType": "string"
      }
    ]
  },
  "error": {
    "code": "string",
    "description": "string",
    "detail": "string"
  }
}
```

`POST /node/connect`

Send the request to connect to a sidechain node

Parameters

Name	Type	Description
host	String	Node hostname
port	int	Node Port

Example request:

Bash

curl -X POST "http://127.0.0.1:9086/node/connect" -H "accept: application/json" -H "Content-Type: application/json" -d "{"host":"string","port":0}"

Example response:

```
{
  "result": {
    "connectedTo": "string"
  },
  "error": {
    "code": "string",
```

(continues on next page)

(continued from previous page)

```
    "description": "string",
    "detail": "string"
  }
}
```

`POST /node/connectedPeers`

Returns the list of all connected sidechain node peers

No Parameters

Example request:

Bash

curl -X POST "http://127.0.0.1:9086/node/connectedPeers" -H "accept: application/json"

Example response:

```
{
  "result": {
    "peers": [
      {
        "address": "string",
        "lastSeen": 0,
        "name": "string",
        "connectionType": "string"
      }
    ]
  },
  "error": {
    "code": "string",
    "description": "string",
    "detail": "string"
  }
}
```

`POST /node/blacklistedPeers`

Returns the list of all blacklisted sidechain node peers

No Parameters

Example request:

Bash

curl -X POST "http://127.0.0.1:9086/node/blacklistedPeers" -H "accept: application/json"

Example response:

```
{
  "result": {
    "addresses": [
      "string"
    ]
  },
  "error": {
```

(continues on next page)

(continued from previous page)

```
    "code": "string",
    "description": "string",
    "detail": "string"
  }
}
```

Sidechain Mainchain Operations

`POST /mainchain/bestBlockReferenceInfo`

Returns the best MC block header which has already been included in a SC block. Returns:

- Mainchain block reference hash with the most height;
- Its height in mainchain;
- Sidechain block ID which contains this MC block reference.

No Parameters

Example request:

Bash

curl -X POST "http://127.0.0.1:9086/mainchain/bestBlockReferenceInfo" -H "accept: application/json"

Example response:

```
{
  "result": {
    "blockReferenceInfo": {
      "mainchainHeaderSidechainBlockId":
↪"a9fd0eee294ee95daad3b72e1f307b52d6b34591dc0c211e49238634c68ecac2",
      "mainchainReferenceDataSidechainBlockId":
↪"a9fd0eee294ee95daad3b72e1f307b52d6b34591dc0c211e49238634c68ecac2",
      "hash":
↪"0e9329f275d8e5081cb10b605a767841eed9d6b4a49e550114bde0ca96fd375c",
      "parentHash":
↪"00ecbbcb1beb5c262f4638d8ac9c9dd5f1e5474f8d97114a426f53d856eccd7a",
      "height": 255
    }
  }
}
```

`POST /mainchain/genesisBlockReferenceInfo`

Reference to Genesis Block

No Parameters

Example request:

Bash

curl -X POST "http://127.0.0.1:9086/mainchain/genesisBlockReferenceInfo" -H "accept: application/json"

Example response:

```
{
  "result": {
    "blockReferenceInfo": {
      "mainchainHeaderSidechainBlockId":
↪"5392e4e8f0f02b00600604d9e65d606418e9e4788552eb0a02629ea9bf6d2a74",
      "mainchainReferenceDataSidechainBlockId":
↪"5392e4e8f0f02b00600604d9e65d606418e9e4788552eb0a02629ea9bf6d2a74",
      "hash":
↪"0536ec69de7f5ec3c8161bc34a014ffe7cae112cab03770972e45fd15da2de82",
      "parentHash":
↪"06660749307d87444d627c3c8b7d795706ce42a62f2b1858043dd9892f8a20d5",
      "height": 221
    }
  }
}
```

`POST /mainchain/blockReferenceInfoBy`

Parameters

Name	Type	Description
hash	String	Block hash
height	int	Block height
format	boolean	

Example request:

Bash

curl -X POST “http://127.0.0.1:9086/mainchain/blockReferenceInfoBy” -H “accept: application/json” -H “Content-Type: application/json” -d “{“hash”:”string”,”height”:0,”format”:false}”

Example response:


```
{
  "result": {
    "blocReferencekInfo": {
      "hash": "string",
      "parentHash": "string",
      "height": 0,
      "sidechainBlockId": "string"
    },
    "blockHex": "string"
  },
  "error": {
    "code": "string",
    "description": "string",
    "detail": "string"
  }
}
```

`POST /mainchain/blockReferenceByHash`

Reference block by hash

Parameters

Name	Type	Description
hash	String	Block hash
format	boolean	

Example request:

Bash

curl -X POST "http://127.0.0.1:9086/mainchain/blockReferenceByHash" -H "accept: application/json" -H "Content-Type: application/json" -d "{"hash":"string","format":false}"

Example response:


```
{
  "result": {
    "blockReference": {
      "header": {
        "mainchainHeaderBytes": "string",
        "version": 0,
        "hashPrevBlock": "string",
        "hashMerkleRoot": "string",
        "hashReserved": "string",
        "hashSCMerkleRootsMap": "string",
        "time": 0,
        "bits": 0,
        "nonce": "string",
        "solution": "string"
      },
      "sidechainRelatedAggregatedTransaction": {
        "id": "string",
        "fee": 0,
        "timestamp": 0,
        "mc2scTransactionsMerkleRootHash": "string",
        "newBoxes": [
          {
            "id": "string",
            "proposition": {
              "publicKey": "string"
            },
            "value": 0,
            "nonce": 0,
            "activeFromWithdrawalEpoch": 0,
            "typeId": 0
          }
        ]
      },
      "merkleRoots": [
        {
          "key": "string",
          "value": "string"
        }
      ]
    },
    "blockHex": "string"
  },
  "error": {
    "code": "string",
    "description": "string",
```


(continues on next page)

(continued from previous page)

```
    "detail": "string"
  }
}
```

HTTP Routing Table

/block

/mainchain

/node

/transaction

/wallet