
Horizen Docs

Release 1.0.1

unknown

May 03, 2023

CONTENTS

1	Overview	1
1.1	Tutorials - start here	1
1.2	how-to	1
1.3	key-topics	1
1.4	Reference	1
1.5	Guides	1
2	Join us online	3
3	Why Horizen Sidechains?	5
3.1	Tutorials	5
3.1.1	Before you start	6
3.1.2	Installing the Horizen Sidechain SDK	7
3.1.3	Internal Representation of a Blockchain	10
3.1.4	The Cross-Chain Transfer Protocol	11
3.1.5	Latus Consensus	16
3.1.6	Node communication	18
3.1.7	Base App	20
3.1.8	Sidechains SDK extension	30
3.1.9	Car Registry Tutorial	43
3.2	Reference	72
3.2.1	Sidechain Node API spec	72
3.3	Guides	112
	HTTP Routing Table	113

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.

1.5 Guides

General documentation and guides

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.

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 information needed 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.

You should be comfortable with concepts like transactions, UTXO's, blocks, validation, confirmation, consensus, unique chains, chain forks, hash functions, private/public keys and signing, along with 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 useful. 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 increasingly frequent attempts to build applications that do not require a trusted third party to ensure that data is stored and processed securely and correctly. These applications keep the concept of a distributed, append-only ledger in place of the traditional application database. This ledger is stored on, and updated by, the application's nodes, which use a consensus mechanism to reach agreement on the legitimacy of transactions, which they then accept and update the ledger. The success of this approach requires, among other things, that the overall system includes an incentive system to adequately reward the app node operators, so that a high degree of decentralization is maintained. The degree of decentralization is such that any attempt at malicious behaviour carries an overwhelmingly uneconomical cost. Today, the only way to guarantee this from day one is to develop a new application in the environment that provides a well-distributed, established, and robust blockchain supporting a traded coin. That way, the robustness of the blockchain extends to the new app, that can immediately make use of the established infrastructure of miners, nodes, and the coin itself.

Unfortunately, the above approach bears a scalability challenge. Blockchains have traditionally offered very limited ability to provide high transaction rates and to accommodate sustained transaction peaks. This severely restricts the number of applications that can be deployed on a blockchain. Additionally, each application needs to be coded in the software run by each node participating in the blockchain validation process, which also has an impact on scalability: the node's software must be 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 partially solves 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 an innovative solution to anyone implementing a blockchain-based distributed and decentralized applications. The environment provides a token that is publicly tradable, and that can be used to reward blockchain actors and support the application's business needs, while solving both of the scalability issues identified above. This approach is detailed in the ([Zendoo whitepaper](#)). The main Horizen blockchain (mainchain), offers the ability to declare the existence of a sidechain through a specific transaction, and once the integration with the mainchain is completed, sending and receiving ZENs (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 conforms to the API requirements of 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 mainchain. This codebase implements not only the Cross-Chain Transfer Protocol, but also includes all the other elements needed to run a blockchain; in particular, it ships with a Proof of Stake consensus protocol that offers yet another scalability advantage, this time related to the electrical power required by traditional Proof of Work consensus protocols: we can scale the application logic AND the number of transactions, both without a large increase in the amount of electrical power needed. The architectural and protocol choices implemented by the SDK are introduced in the Zendoo whitepaper, as the "Latus" construction.

To facilitate the sidechain developer's work, the SDK includes an example of a Sidechain Application, "SimpleApp", that 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 the sidechain, and send them back to the 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 in developing a new sidechain application is to implement new data and logic in a sidechain node. The "Lambo 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 Horizen Sidechain SDK

We'll get started by setting up our environment.

Supported Platforms

The Sidechain SDK is available and tested on 64-bit versions of Linux and Windows.

Requirements

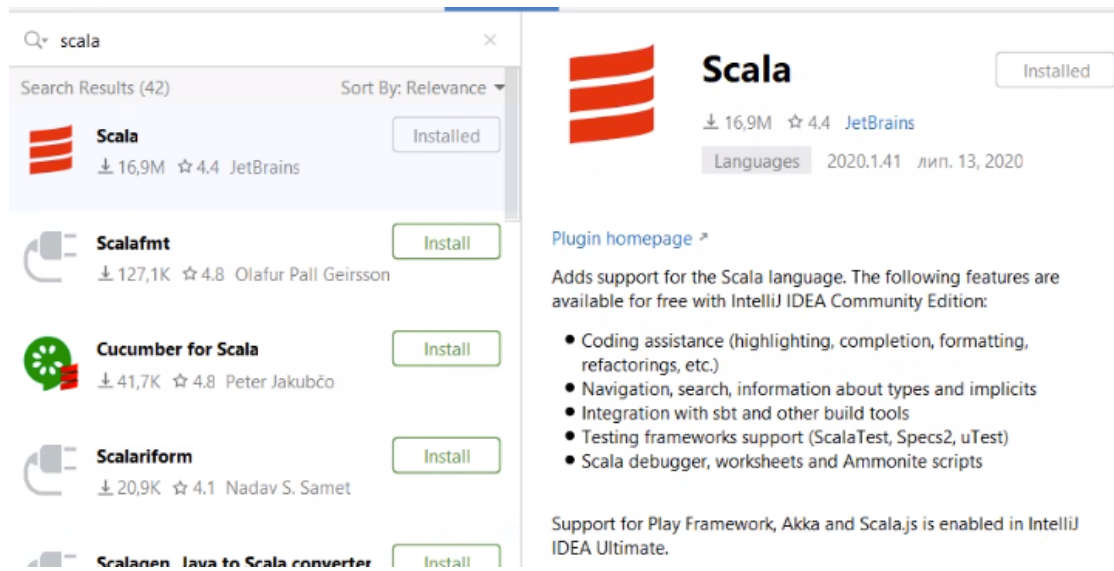
The Sidechain SDK requires Java 11 or newer, Scala 2.12.10+ or newer, and the latest version of [zen](#).

Installing on Windows

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:



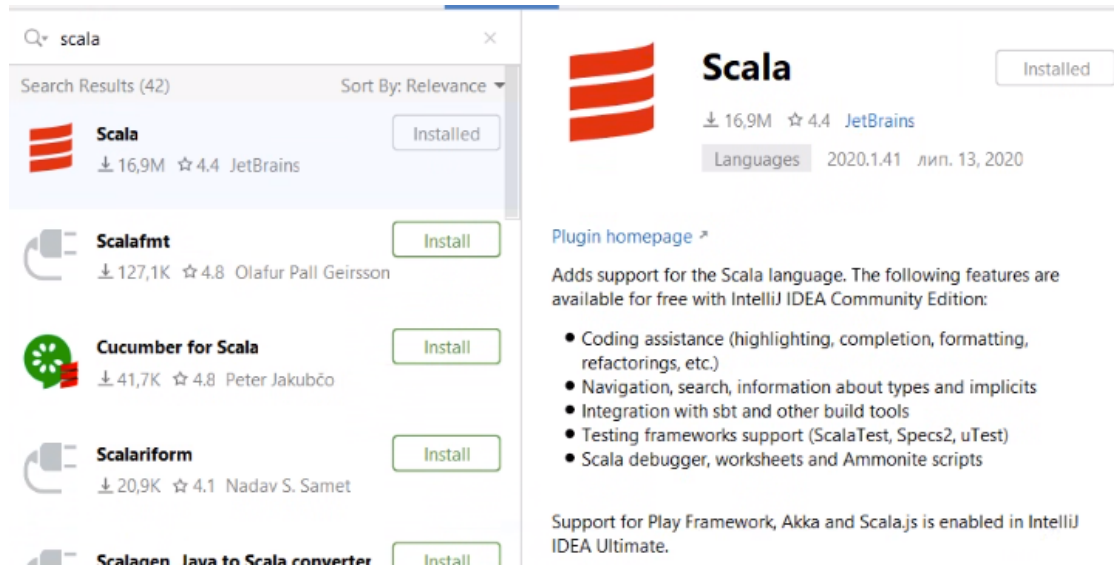
6. In the IDE, you can now go to File and Open the root directory of the project repository, “Sidechains-SDK”. The pom.xml file - the Maven Project Object Model XML file that contains all the project configuration details - should be automatically imported by the IDE. Otherwise, you can just open it.
7. Keep reading this tutorial, and start playing with the code. You will find a sidechain example in the “examples/simpleapp” directory ([link](#)); you can study the code and experiment with it while reading this documentation.
8. While fiddling with the code, you might also want to see a sidechain in action, understand its configuration files, look at its interaction with mainchain and its user interface. Best way to do that is to install a local mainchain and sidechain example node ([link](#))
9. When you are comfortable with the SDK core functionalities, you can tackle Chapter 8 and 9, and learn how to extend the software to add your own data and logic. Here the “Lambo Registry” example ([link](#)) will complement your reading, and show you how to create your own blockchain-based dApp.

Installing on Linux

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 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 a sidechain example in the “examples/simpleapp” directory ([link](#)); you can study the code and experiment with it while reading this documentation.
8. While fiddling with the code, you might also want to see a sidechain in action, understand its configuration files, look at its interaction with mainchain and its user interface. Best way to do that is to install a local mainchain and sidechain example node ([link](#))
9. When you are comfortable with the SDK core functionalities, you can tackle Chapter 8 and 9, and learn how to extend the software to add your own data and logic. Here the “Lambo Registry” example ([link](#)) will complement your reading, and show you how to create your own blockchain-based dApp.

3.1.3 Internal Representation of a Blockchain

The sidechain software is a distributed architecture and 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 four main elements: history, state, wallet, and memory pool. We need to know what a “box” is before we get to know these four elements.

Concept of a Box

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

Node Main Elements & Intro to a “NodeView”

- **History** - is a blockchain ledger that is typically a list of sidechain blocks that were received by the node, verified against consensus rules, and accepted.
- **State** - is a snapshot of all boxes that haven’t been opened yet. It represents the state at the current chain tip.
- **Wallet** - has two main functionalities:
 - It holds the secret keys that belong to that specific node.
 - 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** - is a list of transactions that are known to the node but have not made it to a sidechain block yet.

Together these four 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 the set of objects that result from 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 object that contains some data, e.g. an amount of ZEN, or data of a custom object (such as a car’s plate as we’ll see in Section 9), associated with some conditions (called a “proposition”) that protects it from being spent by anyone 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 a “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 assigned 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 nonced and custom field hash. 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. Box id is the hash of value, proposition, nonce, and custom field hash. 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. Transactions also have unique id based on hash of concatenation of message, proof and custom data. Custom data can be overridden in custom transaction class. The same way for customDataMessageToSign. This method can supply the base transaction class with a message to be signed by the transaction.

3.1.4 The Cross-Chain Transfer Protocol

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

At a high level, it defines three basic operations:

- **Forward Transfer**
- **Backward Transfer**
- **Ceased Sidechain Withdrawal**

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

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

```
sc_create {
  "version": version,
  "withdrawalEpochLength": withdrawalEpochLength,
  "fromaddress": mc_address,
  "changeaddress": mc_address,
  "toaddress": sc_address,
  "amount": creation_amount,
  "minconf": conf,
  "fee": fee,
  "wCertVk": vk,
  'customData': custom_data,
  "constant": constant,
  'wCeasedVk': cswVk,
```

(continues on next page)

(continued from previous page)

```

'vFieldElementCertificateFieldConfig': feCfg,
'vBitVectorCertificateFieldConfig': bvCfg,
'forwardTransferScFee': ftScFee,
'mainchainBackwardTransferScFee': mbtrScFee,
'mainchainBackwardTransferRequestDataLength': mbtrRequestDataLength
}

```

Parameters to the command must be passed in JSON format. The command must specify the destination address where the first forward transfer coins are sent (“toaddress”), its amount (“amount”), as well as the epoch length (“withdrawalEpochLength”). It is the epoch length that defines the frequency, in blocks, of the backward transfers’ submissions (see the “backward transfers” paragraph below). Otherwise sidechain can be declared as non ceasing. In this case sidechain not oblige to send certificates in determined period of time, but can decide itself when to submit it. The `sc_create` command also includes the cryptographic key to receive coins back from the sidechain (“wCertVk”). The verification key guarantees that the received coins were processed according to a matching proving system. Besides these parameters, `sc_create` has some optional ones, here is the complete set of parameters:

- **version** - (numeric, required) The version of the sidechain. Recommended to use version 1. For non ceasing sidechain and for circuit with key rotation must be 2.
- **withdrawalEpochLength** - (numeric, optional, default=100) length of the withdrawal epochs. The minimum valid value in regtest is: 2, the maximum (for any network type) is: 4032. For non ceasing sidechain should be 0.
- **fromaddress** - (string, optional) The MC taddr to send the funds from. If omitted funds are taken from all available UTXO.
- **changeaddress** - (string, optional) The MC taddr to send the change to, if any. If not set, “fromaddress” is used. If the latter is not set too, a newly generated address will be used.
- **toaddress** - (string, required) The receiver PublicKey25519Proposition in the SC.
- **amount** - (numeric, required) Funds to be sent to the newly created Sidechain. Value expressed in ZEN.
- **minconf** - (numeric, optional, default=1) Only use funds confirmed at least this many times.
- **fee** - (numeric, optional) The fee amount to attach to this transaction in ZEN. If not specified it is automatically computed using a fixed fee rate (default is 1zat per byte).
- **wCertVk** - (string, required) It is an arbitrary byte string of even length expressed in hexadecimal format. Required to verify a WCert SC proof. Its size must be 9216 bytes max.
- **customData** - (string, optional) An arbitrary byte string of even length expressed in hexadecimal format. A max limit of 1024 bytes will be checked.
- **constant** - (string, optional) It is an arbitrary byte string of even length expressed in hexadecimal format. Used as public input for WCert proof verification. Its size must be 32 bytes.
- **wCeasedVk** - (string, optional) It is an arbitrary byte string of even length expressed in hexadecimal format. Used to verify a Ceased sidechain withdrawal proof for given SC. Its size must be 9216 bytes max. Not supported in version 2.
- **vFieldElementCertificateFieldConfig** - (array, optional) An array whose entries are sizes (in bits). Any certificate should have as many custom FieldElements with the corresponding size.
- **vBitVectorCertificateFieldConfig** - (array, optional) An array whose entries are bitVectorSizeBits and max-CompressedSizeBytes pairs. Any certificate should have as many custom BitVectorCertificateField with the corresponding sizes.
- **forwardTransferScFee** - (numeric, optional, default=0) The amount of fee in ZEN due to sidechain actors when creating a FT

- **mainchainBackwardTransferScFee** - (numeric, optional, default=0) The amount of fee in ZEN due to sidechain actors when creating a MBTR
- **mainchainBackwardTransferRequestDataLength** - (numeric, optional, default=0) The expected size (max=16) of the request data vector (made of field elements) in a MBTR

As a consequence of the sidechain declaration command, a unique sidechain id will be assigned to that sidechain, and from that moment on that id can be used for every operation related to that specific sidechain:

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

From the Mainchain perspective a non ceasing sidechain doesn't have withdrawal epoch length and can switch the withdrawal epoch in any time. Meanwhile the non ceasing sidechain has a virtual withdrawal epoch length. This parameter motivates sidechain to schedule certificate submission after the specified period. The virtual withdrawal epoch length can not be less than 10 for the regtest network and less than 100 for the mainnet and the testnet networks.

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 amount of ZEN to be sent. Forward Transfer can be done by using following RPC command:

```
sc_send <outputs> [params]
```

The input arguments have the following structure:

- **1. outputs** - (string, required) A json array of json objects representing the amounts to send:

```
[{
  "scid": id,
  "toaddress":sc_addr,
  "amount":amount,
  "mcReturnAddress":mc_addr
}, ..., ]
```

Where:

- **scid** - (string, required) The uint256 side chain ID
- **toaddress** - (string, required) The receiver PublicKey25519Proposition in the SC
- **amount** - (numeric, required) Value expressed in ZEN
- **mcReturnAddress** - (string, required) The Horizen mainchain address where to send the backward transfer in case Forward Transfer is rejected by the sidechain

And:

- **2. params** - (string, optional) A json object with the command parameters:

```
{
  "fromaddress":taddr
  "changeaddress":taddr
  "minconf":conf
  "fee":fee
}
```

Where:

- **fromaddress** - (string, optional) The taddr to send the funds from. If omitted funds are taken from all available UTXO
- **changeaddress** - (string, optional) The taddr to send the change to, if any. If not set, “fromaddress” is used. If the latter is not set too, a newly generated address will be used
- **minconf** - (numeric, optional, default=1) Only use funds confirmed at least this many times.
- **fee** - (numeric, optional) The fee amount to attach to this transaction in ZEN. If not specified it is automatically computed using a fixed fee rate (default is 1zat per byte)

This command specifies the SC destination where the forward transfer coins are sent (“toaddress”), the amount (“amount”) and the MC address where to send a backward transfer in case Forward Transfer is rejected by the sidechai (“mcReturnAddress”).

From the mainchain’s 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 the mainchain destination. A Backward Transfer is initiated by a **Withdrawal Request** which is a sidechain transaction issued by the coin’s owner. The request specifies the mainchain destination address and the amount. More precisely, the withdrawal request owner will create a Withdrawal-RequestBox that destroys the specified amount of coins in the sidechain. This is not enough to move those coins back to the mainchain though: we need to wait until 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 then 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 match the amount that was sent to it.

As an optional step, on MC side it is possible to explicitly request a Backward Transfer from the SC which should be included in one of the next certificates via the following RPC command:

```
sc_request_transfer <outputs> [params]
```

The input arguments have the following structure:

- **1. outputs** - (string, required) A json array of json objects representing the request to send:

```
[{
  "scid": id,
  "vScRequestData": req_data,
  "mcDestinationAddress": mc_addr,
  "scFee": amount,
}, ..., ]
```

Where:

- **scid** - (string, required) The uint256 side chain ID
- **vScRequestData** - (array, required) It is an arbitrary array of byte strings of even length expressed in hexadecimal format representing a SC reference (for instance an Utxo ID) for which a backward transfer is being requested. The size of each string must be 32 bytes.
- **mcDestinationAddress** - (string, required) The Horizen mainchain address where to send the backward transfer

- **scFee** - (numeric, required) The amount in ZEN representing the value spent by the sender that will be gained by a SC forger

And:

- **2. params** - (string, optional) A json object with the command parameters:

```
{
  "fromaddress":taddr
  "changeaddress":taddr
  "minconf":conf
  "fee":fee
}
```

Where:

- **fromaddress** - (string, optional) The taddr to send the funds from. If omitted funds are taken from all available UTXO
- **changeaddress** - (string, optional) The taddr to send the change to, if any. If not set, “fromaddress” is used. If the latter is not set too, a newly generated address will be used
- **minconf** - (numeric, optional, default=1) Only use funds confirmed at least this many times.
- **fee** - (numeric, optional) The fee amount to attach to this transaction in ZEN. If not specified it is automatically computed using a fixed fee rate (default is 1zat per byte)

Ceased Sidechain Withdrawal

The funds of a ceased sidechain can be withdrawn back to the mainchain with a Ceased Sidechain Withdrawal request. This request can be performed right after the sidechain ceasing.

This feature is optional. In order to enable the CSW for a sidechain, it is necessary to provide a specific key to be used by the mainchain to verify the validity of a Ceased Sidechain Withdrawal. This key should be provided using the *wCeasedVk* parameter in *sc_create* command. In addition, the CSW requires 2 custom *FieldElementCertificateField* of 255 bits size, so the parameter *vFieldElementCertificateFieldConfig* in *sc_create* command should be set to [255, 255].

To create a CSW request, a nullifier and a Ceased Sidechain Withdrawal proof should be generated on the sidechain side. Nullifier can be generated by API command *nullifier* (CSW API group). Proof generation can be done with *generateCswProof* command. Command *cswInfo* shows csw related data for specified box id.

Mainchain request can be performed through a raw transaction with the following structure:

```
sc_csws = [{
  "amount": sc_csw_amount,
  "senderAddress": csw_mc_address,
  "scId": scid,
  "epoch": 0,
  "nullifier": nullifier,
  "activeCertData": actCertData,
  "ceasingCumScTxCommTree": ceasingCumScTxCommTree,
  "scProof": sc_proof1
}]
```

Circuit with key rotation

Circuit with key rotation is needed to replace compromised signers and masters keys of certificate submitters with new keys. These changes occur in-chain for 2 reasons:

- every node must keep knowledge about the recent set of public keys. And if we keep this information off-chain we can easily lose it.
- we need to be sure that all the nodes use exactly the same source of data for signing or verifying certificate, creating the snark proof, etc.

Every key rotation transaction is validated according to a set of rules, then all key rotations within certificate submission epoch are aggregated, included to certificate, submitted to Mainchain. Starting from the next epoch previous keys are invalidated, and new keys are activated. Key rotation can be performed with `createKeyRotationTransaction` API command(transaction group). Caller should be authenticated to use it.

Parameters for request are following:

- **keyType** of type Integer;
- **keyIndex** of type Integer, must not be less than zero;
- **newKey** of type String, this is a required parameter;
- **signingKeySignature** of type String, this is a required parameter;
- **masterKeySignature** of type String, this is a required parameter;
- **newKeySignature** of type String, this is a required parameter;
- **format** of type Boolean, can be nullable;
- **automaticSend** of type Boolean, can be nullable;
- **fee** of type Long, can be nullable;

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 any proving system 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’s architecture other than conforming to 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”, where each epoch comprises 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 propagated blocks for them.

Forger Stake can be delegated to another node with `makeForgerStake` command. This command contains a list of transactions, each of them specifies the `publicKey` of a sender, `blockSignPublicKey` and `vrfPubKey` of receiving node and coin amount. Also, Forger Stake can be delegated by `spendForgingStake` command. In this case, Forger Stake coins must be specified in `forgerOutputs`, coins specified in `regularOutputs` section will be moved from Forger Stakes and can be used as regular coins. Creation Forger Stakes from regular coins can be done with `createCoreTransaction`. As with `spendForgingStake`, Forger coins must be specified in `forgerOutputs` section.

A slot leader eligible for a certain slot that creates and propagates 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 their 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 possibility) every $720 * 120 = 86400$ seconds, i.e. 24 hours.
- BlockSize Limit 5MB

Seconds in one slot value must be specified in `SidechainApp` arguments. The minimum valid value is 10, the maximum is 300. Please note, that in case seconds in one slot has value 10, consensus epoch length shorten to 2 hours. At least one block of epoch must be generated during that time.

Fee redistribution

At the end of the epoch, all fees are redistributed between all forgers. Each forger receives defined percentage of fee of each block it generated. All others fees of all blocks of the Withdrawal epoch must be spreaded equally between block forgers of the epoch. The remaining coins (after fee spreading) will be sent to first forgers of the epoch (the number of forgers is equal to the remained Satoshi), one Satoshi for each forger.

Pretty Good Decentralization ===

Pretty Good Decentralization (PGD) - mechanism to spread certificate signing between different nodes in the sidechain. It unloads submitter moving particular certificate signing calculation to other nodes. Each node can posses one or more schnorr key for signing. Keys must be specified in the configuration file.

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/or replace core behaviours.

The API configuration can be found in the sidechain node's configuration file.

For example, review the restApi section of the following file for the SimpleApp:

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

The available options are:

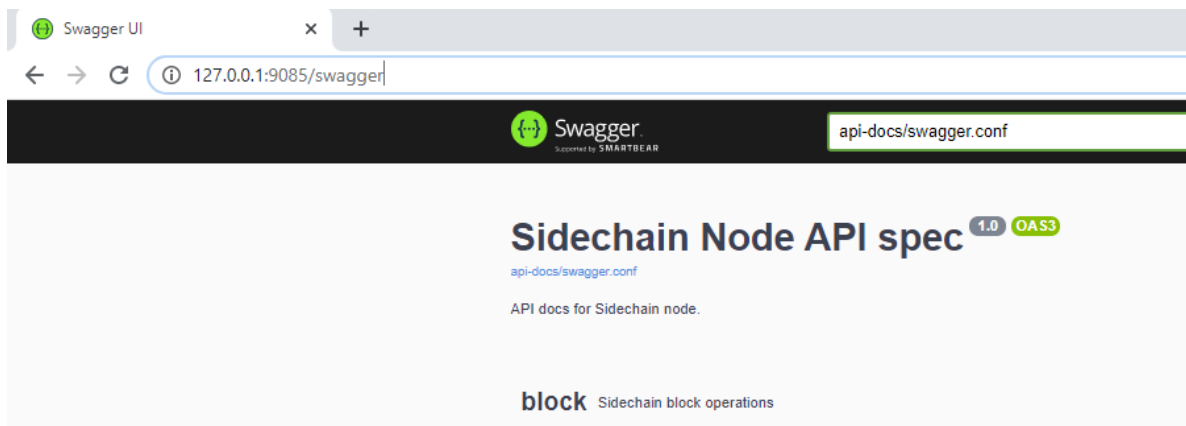
bindAddress – “IP:port” address for sending HTTP request, e.g. “127.0.0.1:9085”

apiKeyHash – Authentication header must be a standard HTTP Basic Authentication where the password hashes to the field “apiKeyHash” specified in each sidechain node's .conf file. The authentication header could be empty if no apiKeyHash is specified

timeout – Timeout in seconds on API requests

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 “IP:port”, as defined in your configuration file, in your browser and select any of the commands shown there. For example:



Default standard API

Base API is organized into the following 5 groups:

- **Block** – Sidechain block operations, e.g. find a block by its blockId, find a blockId by block height, etc. Also here you could find forging-related commands like the ones to automatically start/stop forging, get information about forging like last epoch and slot index. Automatic forging gets current time and converts it into appropriate slot/epoch index. Thus, if for some reason a sidechain node skips the correct timeslot for an entire consensus epoch when forging in automatic mode, it will always fail. A sidechain where this occurs will be considered deceased, and communication between the sidechain and mainchain is no longer possible. However, forging a block with a manually set epoch/slot index is possible by API call `/block/generate`, which could be useful if the sidechain is run in isolated mode.
- **Transaction** – Sidechain transaction operations like find all transactions, create a transaction without sending it into the memory pool, send transaction into memory pool, etc.
- **Wallet** – Sidechain wallet operations. Wallet operations could take `boxType` as an optional parameter, 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 are required to use the fully-qualified class name). If box type is not relevant, you can simply 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 mainchain header included in sidechain.
- **Submitter** – Certificate submitter operations like current status of certificate generation, managing operation of submitting, and signing of a certificate.
- **Csw** – Ceased Sidechain Withdrawal operations like CSW proof generation or managing nullifiers.

API authentication

We support the Basic Authentication inside our REST interface. In order to enable it you should define an api key hash inside the config file section **restApi.apiKeyHash**. Api key hash should be the **BCrypt** Hash of the password used in the Basic Auth.

It's possible to calculate this Hash using the **ScBootstrapping tool** with the command **encodeString**.

```
encodeString:{"string": "a8q38j2f0239jf2o1f20f"}
```

Then, in the HTTP request you need to add the Basic Authentication header.

Example:

HTTP request:

```
"Authorization": "Basic a8q38j2f0239jf2o1f20f"
```

Config file:

```
restApi {
  "apiKeyHash": "2y$12$vgalLEzU1jiLYI766CIeVOi1A9QwFBqYgjbAsD.2t8Z7SFP6ff4Eq"
}
```

If you want to add authentication to your custom endpoints you just need to wrap your code between the `withBasicAuth` directive.

Example:

```
your_custom_endpoint() = {  
    withBasicAuth {  
        <custom endpoint implementation>  
    }  
}
```

3.1.7 Base App

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

Secret / Proof / Proposition

The SDK uses its own terminology for private key / public key / signed message:

- **Secret** - Private key
- **Proposition** - Public key, used in boxes as a locker
- **Proof** - Signed message

The SDK ships with the following implementations for Secret / Proof / Proposition

- **Curve 25519**, currently used for Sidechain signing needs, e.g. to sign a transaction. This technology will not be used in the
 - PrivateKey25519
 - PublicKey25519Proposition
 - Signature25519
- **Verifiable Random Function based on [ginger-lib](#)**, used to assign and prove eligibility of block forgers.
 - VrfSecretKey
 - VrfPublicKey
 - VrfProof
- **Schnorr based on [ginger-lib](#)**.
 - SchnorrSecret
 - SchnorrProposition
 - SchnorrProof

Boxes

Data in a sidechain is meant to be represented as a Box. That data is kept “closed” by a Proposition, and can be opened (i.e. “spent”) only with the Proposition’s Secret(s). The Sidechain SDK offers two different Box types: Coin Box and non-Coin Box.

A Coin Box contains ZEN. A Non-Coin Box does not contain ZEN and represents a unique entity that can be transferred between different owners. Examples of a Coin box are ZenBox and ForgingBox. A Coin Box can add custom data to an object that represents coins, i.e. an object that 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.

A Box represents an entity in the blockchain, and all operations, such as create/open, are performed on it. Any Box contains a BoxData, which holds all the properties of that specific entity, such as value, proposition address, and any custom data.

Every Box has its own unique `boxId` (not be confused with box type id, which is used for serialization). That `boxId` 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 the custom data into the `customFieldsHash()` function.

The following Coin-Box types are provided by the SDK:

- **RegularBox** – contains ZEN coins
- **ForgerBox** – contains ZEN coins that are staked for forging eligibility. A higher amount of ZEN in a ForgerBox offers higher chances of being selected to forge blocks (please check “Proof of Stake” consensus for more information on this).
- **WithdrawalRequestBox** – contain ZEN coins ready to be transferred back to mainchain. The actual transfer will be finalized by backward transfers that will be included in a certificate posted to the mainchain, after the end of the epoch.

An SDK developer can declare custom Boxes; please refer to the SDK extension section for details.

Transactions

There are two basic transactions: [MC2SCAggregatedTransaction](#) and [SidechainCoreTransaction](#).

An `MC2SCAggregatedTransaction` is the implementation in a sidechain of Forward Transfers to that specific sidechain, i.e. mainchain transactions that send coins to addresses of that specific sidechain. When a Forger is going to produce a sidechain block, and a new mainchain block appears, the forger will mention that mainchain block as a reference that contains that 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 send coins inside a sidechain, create forging stakes, or perform withdrawal requests (i.e. send coins back to the mainchain).

All custom transactions inherited from `SidechainTransaction`. `SidechainNoncedTransaction` - a class that helps to deal with output Boxes nonces. `AbstractRegularTransaction` is a class that helps to deal with ZenBoxes. These classes can be extended to support custom logic operations. For example, if we think about a real estate sidechain, we can tokenize private property as a specific Box using `AbstractRegularTransaction`. Please refer to the SDK extensions for more details.

Serialization

Because the SDK is based on Scorex, it implements the Scorex pattern for data serialization: any application custom object that needs to be serialized, like Box, BoxData, Secret, Proof, Transaction, must implement the [Scorex BytesSerializable](#) interface.

This interface defines two methods:

- `byte[] bytes()` - returns a bytearray representing the object
- `Serializer serializer()` - returns the class responsible to parse and write the object through Scorex Reader and Writer, which are wrappers on byte streams

The SDK provides basic serializer interfaces for its objects (for example [BoxDataSerializer](#) for BoxData, [TransactionSerializer](#) for Transactions), ready to be extended when writing specific custom serializers. All other serializers must implement the ScorexSerializer interface.

This interface defines two abstract methods: - `serialize(T object, Writer writer)` - writes object to the Writer - `T parse(Reader reader)` - parse bytes from the Reader and returns an object

All serialization and parsing logic must be placed to these methods.

We also need to instruct the dependency injection system on what appropriate serializer must be used for each object: this must be performed inside the AppModule configure() method, by adding key-value maps: the key is the specific type-id of each object (each object type must declare a unique type id), and the value is the serializer instance to be used for that object. There are separate maps for each class of object (one for Boxes, one for BoxData, one for Transactions and so on). Please refer to the SDK extension section for more information.

SidechainNodeView

SidechainNodeView is the access point to the current node state; that includes NodeWallet, NodeHistory, NodeState, NodememoryPool, as well as application data. When defining custom API end points, you can extend a specific class and have access to SidechainNodeView.

Memory Pool

The Memory Pool is the node's mechanism for storing transactions that haven't been included in a block yet. It acts as a sort of transactions' "waiting room". It has a customizable size that can be changed in the configuration file. In case of a full Memory Pool the transaction with the lower **fee rate** is removed. It's also possible to define a minimum amount of fee rate that a transaction should have in order to be included in the Memory Pool. By default the transactions are sorted by fee rate.

The fee rate of a transaction is calculated by the following formula:

Note: $\text{transaction_fee_rate} = \text{transaction_fee} * 1000 / \text{transaction_size}$

These parameter can be found in the configuration file in the section mempool

```
mempool {
    #unit is MB
    maxSize = <MEMPOOL MAX SIZE> (Default is 300 MB)
    #unit is ZENToshi per kb
    minFeeRate = <MEMPOOL_MIN_FEE_RATE> (Default is 0)
}
```

There is also the possibility to define a max fee threshold that blocks the possibility to launch transactions from the node wallet that have fee > max fee. This property can be set in the configuration file in the section wallet.

```
wallet {
    #Long
    maxTxFee = 10000000 (Default value)
}
```

Node wallet

It contains the private keys known to the node.

State

It contains information about the node's current state, i.e. the information that the node stores and updates to be able to operate. As an example, to validate transactions a node needs to know which are the outputs that haven't been spent yet.

History

Provide access to history, i.e. to the previous blocks (on the active chain, and on forked ones).

Network layer

The network layer is made of two distinct parts: communication between nodes and communication between the node and node users. The interconnection among nodes is structured as a peer-to-peer network. Over the network, the SDK handles the handshake, blockchain synchronization, and transaction transmission. The communication between a node and its users is available through http end points.

Physical storage

The SDK introduces the unified physical storage interface, and this default implementation is based on the [LevelDB key-value storage](#). Sidechain developers can decide to use the default solution or provide a custom implementation. For example, the developer could decide to use encrypted storage, a Key Value store, a relational database or even a cloud solution. When using a custom implementation, please make sure that the [Storage](#) test passes.

User-specific settings

A 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, by modifying the configuration file. The file is written in [HOCON notation](#), that is JSON made more human-editable. The configuration file consists of the SDK's required fields and the application's custom fields, if needed. Sidechain developers can use the [io.horizen.settings.SettingsReader](#) utility class to extract sidechain-specific data and the config object itself to get custom parts.

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

    public SidechainSettings getSidechainSettings()
```

(continues on next page)

(continued from previous page)

```

    public Config getConfig()
}

```

In the above class, `userConfigPath` is the path to the user defined configuration file. The optional parameter `applicationConfigPath` is a path to a configuration file that can be defined by the developer to set default values or values that are not meant to be modified by the user. The two getters (`getSidechainSettings` and `getConfig`) return the two merged configurations.

SidechainApp class

The starting point of the SDK for each sidechain is the [SidechainApp class](#). Every sidechain application should create an instance of `SidechainApp`, passing all the required parameters, and then call its `run()` method to start the sidechain node:

```

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,
        Storage walletCswDataStorage,
        Storage stateUtxoMerkleTreeStorage,
        Storage stateForgerBoxStorage,
        Storage backupStorage,

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

        // Application specific logic handler for the node safe stop
        SidechainAppStopper applicationStopper,

        // Application specific configs for core forks activation
        ForkConfigurator forkConfigurator

        int consensusSecondsInSlot = 120
    )
}

```

(continues on next page)

(continued from previous page)

```
public void run()
}
```

The SidechainApp instance can be instantiated directly or through the [Guice DI library](#).

Direct instantiation:

All the required dependencies are passed inside the constructor:

```
SidechainApp app = new SidechainApp(...);
app.run();
```

Guice instantiation:

You can define a Guice module which declares all the bindings, then use that module to create a guice injector, and call its getInstance() method to obtain the app instance:

```
Injector injector = Guice.createInjector(new MyAppModule());
SidechainApp app = injector.getInstance(SidechainApp.class);
sidechainApp.run();
```

The Guice module class (MyAppModule in the example above) must extend the class com.google.inject.AbstractModule, and define the bindings inside its config() method. A binding definition could be done in the following ways:

```
bind( <injected_classType> )
    .annotatedWith(Names.named( <identifier>))
    .toInstance(<custom class instance>);
```

injected_classType and identifier must belong to the binding types defined in the SDK. In the following list, you can find all the bindings that can be declared, with a brief description and example of binding declaration code:

- SideChain settings

Must be an instance of io.horizen.SidechainSettings, defining the sidechain configuration parameters.

```
bind(SidechainSettings.class)
    .annotatedWith(Names.named("SidechainSettings"))
    .toInstance(..);
```

- Custom box serializers

Serializers to be used for custom boxes, in the form HashMap<CustomboxId, BoxSerializer>. Use new HashMap<>() ; if no custom serializers are required.

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

- Custom secrets serializers

Serializers to be used for custom secrets, in the form HashMap<SecretId, SecretSerializer>. Use new HashMap<>() ; if no custom serializers are required.

```
bind(new TypeLiteral<HashMap<Byte, SecretSerializer<Secret>>>>() {}))
    .annotatedWith(Names.named("CustomSecretSerializers"))
    .toInstance(..);
```

- Custom transaction serializers

Serializers to be used for custom transaction, in the form `HashMap<CustomTransactionId, TransactionSerializer>`. Use new `HashMap<>()`; if no custom serializers are required.

```
bind(new TypeLiteral<HashMap<Byte, TransactionSerializer<BoxTransaction<Proposition,
↳Box<Proposition>>>>>() {}))
.annotatedWith(Names.named("CustomTransactionSerializers"))
.toInstance(..);
```

- Application Wallet

Class defining custom application wallet logic. Must be an instance of a class implementing the `io.horizen.utxo.wallet.ApplicationWallet` interface.

```
bind(ApplicationWallet.class)
.annotatedWith(Names.named("ApplicationWallet"))
.toInstance(..);
```

- Application state

Class defining custom application state logic. Must be an instance of a class implementing the `io.horizen.utxo.state.ApplicationState` interface.

```
bind(ApplicationState.class)
.annotatedWith(Names.named("ApplicationState"))
.toInstance(..);
```

- Secret storage

Class for defining Secret storage, i.e. a place where secret keys are stored. Must be an instance of a class implementing the `io.horizen.storage.Storage` interface.

```
bind(Storage.class)
.annotatedWith(Names.named("SecretStorage"))
.toInstance(..);
```

- WalletBoxStorage

Internal storage used for the wallet. Must be an instance of a class implementing the `io.horizen.storage.Storage` interface.

```
bind(Storage.class)
.annotatedWith(Names.named("WalletBoxStorage"))
.toInstance(..);
```

- WalletTransactionStorage

Internal storage used for transactions. Must be an instance of a class implementing this interface: `io.horizen.storage.Storage`

```
bind(Storage.class)
    .annotatedWith(Names.named("WalletTransactionStorage"))
    .toInstance(..);
```

- WalletForgingBoxesInfoStorage

Internal storage used for forging boxes. Must be an instance of a class implementing the `io.horizen.storage.Storage` interface.

```
bind(Storage.class)
  .annotatedWith(Names.named("WalletForgingBoxesInfoStorage"))
  .toInstance(..);
```

- StateStorage

Internal storage used to save the current State, e.g. store information about boxes currently still closed, perform rollbacks in case of forks, etc. Must be an instance of a class implementing the `io.horizen.storage.Storage` interface.

```
bind(Storage.class)
  .annotatedWith(Names.named("StateStorage"))
  .toInstance(..);
```

- StateForgerBoxStorage

Internal storage used to save the Forger boxes. Must be an instance of a class implementing the `io.horizen.storage.Storage` interface.

```
bind(Storage.class)
  .annotatedWith(Names.named("StateForgerBoxStorage"))
  .toInstance(..);
```

- HistoryStorage

Internal storage used to store all the History data, including blocks of all forks. Must be an instance of a class implementing the `io.horizen.storage.Storage` interface.

```
bind(Storage.class)
  .annotatedWith(Names.named("HistoryStorage"))
  .toInstance(..);
```

- ConsensusStorage

Internal storage to save consensus data. Must be an instance of a class implementing the `io.horizen.storage.Storage` interface.

```
bind(Storage.class)
  .annotatedWith(Names.named("ConsensusStorage"))
  .toInstance(..);
```

- CswDataStorage

Internal storage to save data for recovering coins from the ceased Sidechain. Must be an instance of a class implementing the `io.horizen.storage.Storage` interface.

```
bind(Storage.class)
  .annotatedWith(Names.named("WalletCswDataStorage"))
  .toInstance(..);
```

- UtxoMerkleTreeStorage

Internal storage to save UTXO Merkle Tree data. Must be an instance of a class implementing the `io.horizen.storage.Storage` interface.

```
bind(Storage.class)
  .annotatedWith(Names.named("StateUtxoMerkleTreeStorage"))
  .toInstance(..);
```

- BackupStorage

Storage containing the non coin-boxes saved during the backup procedure and restored during the restore procedure (See [Backup and restore procedure](#)). If you don't want to have any restore logic you can leave this empty. Must be an instance of a class implementing the `io.horizen.storage.Storage` interface.

```
bind(Storage.class)
  .annotatedWith(Names.named("BackupStorage"))
  .toInstance(..);
```

- Custom API extensions

Used to add new custom endpoints to the http API.

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

- Forbidden standard API

Used to disable some of the standard http API endpoints. Each pair on the passed list represents a path to be disabled (the key is the basepath, the value the subpath).

```
bind(new TypeLiteral<List<Pair<String, String>>> () {})
  .annotatedWith(Names.named("RejectedApiPaths"))
  .toInstance(...);
```

- Sidechain Application stopper

It is a customized class instance which implements the public interface *SidechainAppStopper* and must provide an implementation of the method 'void stopAll()'. Such a method is called by the SDK when the node stop procedure is initiated. Such a procedure can be explicitly triggered via the API 'node/stop' or can be triggered when the JVM is shutting down, for instance when a SIGINT is received. In the custom implementation for instance, custom storages should be closed or any resources should be properly released. An example is provided in the "SimpleApp" with the `SimpleAppStopper.java` class.

```
bind(SidechainAppStopper.class)
  .annotatedWith(Names.named("ApplicationStopper"))
  .toInstance(applicationStopper);
```

- Fork configurator

SDK may introduce the backward incompatible changes that will lead to the hard fork for the already running sidechains. Every sidechain application should use *ForkConfigurator* to specify the activation points for regtest, testnet and mainnet networks.

```
bind(ForkConfigurator.class)
  .annotatedWith(Names.named("ForkConfiguration"))
  .toInstance(forkConfigurator);
```

- Seconds in slots parameter

It's integer parameter that defines slot duration. The minimum valid value is 10, the maximum is 300. (See `latu_params`)

```
bind(Integer.class)
  .annotatedWith(Names.named("ConsensusSecondsInSlot"))
  .toInstance(consensusSecondsInSlot);
```

SidechainApp arguments can be split into several groups:

1. Settings

- An instance of SidechainSettings can be retrieved by a custom application via SettingsReader, as seen above.

2. Custom objects serializers

- Developers will most likely want to add their custom data and business logic. For example, an application for tokenization of real-estate properties will want to create custom Box and BoxData types. These custom objects will have to be managed by the SDK, so that they can be sent through the network or stored on the disk. The SDK then need to know how to serialize them to bytes and how to deserialize them. This information is coded by the Sidechain developers, who must specify custom objects serializers and add them to the Serializer map. This will be better described in chapter 8.1, “Sidechain SDK extension, Data serialization”.

3. Application node extension of State and Wallet logic

- As seen above, the state is a snapshot of all unspent boxes on the blockchain at a given moment. So when a new block arrives, the ApplicationState validates the block, e.g. to prevent the spending of non-existing boxes, or to discard transactions with inconsistencies in their input/output balance. Developers can extend this validation process by introducing additional logic in ApplicationState and ApplicationWallet.

4. API extension - [link](#)

5. Node communication - [link](#)

6. Core forks management.

The SDK repository includes in its “examples” folder, the “SimpleApp” sidechain; it’s an application that does not introduce any custom logic: no custom boxes or transactions, no custom API, an empty ApplicationState and ApplicationWallet. “SimpleApp” shows the basic SDK functionalities, that are immediately available to the developer, and it’s the fastest way to get started with our SDK.

Remote Key Manager configuration

Remote Key Manager (or Secure Enclave) is a remotely hosted server that exposes REST HTTP API. When configured, the Sidechain nodes will communicate with it for operations such as creating and verifying signatures or messages. This functionality is used to sign Withdrawal Certificate with private parts of the signing keys, that can now be stored remotely and not specified in the Node configuration file.

- Example configuration:

```
remoteKeysManager {
    enabled = true
    address = "https://$host:$port"
}
```

By default, Remote Key Manager is disabled

- With Remote Key Manager configured, signers secrets configuration is not needed:

```
withdrawalEpochCertificate {
    ...
    signersSecrets = []
    ...
}
```

Key management inside Secure Enclave is based on AWS Key Management Service.

3.1.8 Sidechains SDK extension

To build a distributed, blockchain application, a developer typically needs to do more than just receive, transfer, and send coins back to the mainchain, as you can do with the basic components provided out-of-the-box by the SDK. Usually, there is the need to define some custom data, that the sidechain users can process and exchange according to some defined logic. In this chapter, we'll see what are the steps that should be taken to code a sidechain which implements custom data and logic. In the next one, we'll look in detail at a specific, customized sidechain example.

Custom box creation

The first step of the development process of a distributed app implemented as a sidechain, is the representation of the needed data. In the SDK, application data are modeled as “Boxes”.

Every custom box should at least implement the `io.horizen.utxo.box.Box` interface. The methods defined in the interface are the following:

- `long nonce()` The nonce guarantees that two boxes having the same properties and values, produce different and unique ids.
- `long value()` If the box type is a Coin-Box, this value is required and will contain the coin value of the Box. In the case of a Non-Coin box, this value is still required, and could have a customized meaning chosen by the developer, or no meaning, i.e. not used. In the latter case, by convention is generally set to 1.
- `Proposition proposition()` should return the proposition that locks this box. The proposition that is used in the SDK examples is `io.horizen.proposition.PublicKey25519Proposition`; it's based on [Curve 25519](#), a fast and secure elliptic curve used by Horizen mainchain. A developer may want to define and use custom propositions.
- `byte[] id()` should return a unique identifier of each box instance.
- `byte[] bytes()` should return the byte representation of this box.
- `BoxSerializer serializer()` should return the serializer of the box (see below).
- `byte boxTypeId()` should return the unique identifier of the box type: each box type must have a unique identifier inside the whole sidechain application.
- `String typeName()` should return the name of class
- `boolean isCustom()`

should return true for all custom boxes

As a common design rule, you usually do not implement the `Box` interface directly, but extend instead the abstract class `io.horizen.utxo.box.AbstractBox`, which already provides default implementations of some useful methods like `id()`, `equals()`, `hashCode()`, `typeName()` and `isCustom()`. This class requires the definition of another object: a class extending `io.horizen.utxo.box.AbstractBoxData`, where you should put all the properties of the box, including the proposition. You can think of the `AbstractBoxData` as an inner container of all the fields of your box. This data object must be passed in the constructor of `AbstractBox`, along with the nonce. The important methods of `AbstractBoxData` that need to be implemented are:

- `byte[] customFieldsHash()` Must return a hash of all custom data values, otherwise those data will not be “protected,” i.e., some malicious actor can change custom data during transaction creation.
- `Box getBox(long nonce)` creates a new `Box` containing this `BoxData` for a given nonce.
- `BoxDataSerializer serializer()` should return the serializer of this box data (see below)

BoxSerializer and BoxDataSerializer

Each box must define its own serializer and return it from the `serializer()` method. The serializer is responsible to convert the box into bytes, and parse it back later. It should implement the `io.horizen.utxo.box.BoxSerializer` interface, which defines two methods:

- `void serialize(Box box, scorex.util.serialization.Writer writer)` writes the box content into a Scorex writer
- `Box parse(scorex.util.serialization.Reader reader)` perform the opposite operation (reads a Scorex reader and re-create the Box)

Also any instance of `AbstractBoxData` needs to have its own serializer: if you declare a `boxData`, you should define one in a similar way. In this case the interface to be implemented is `io.horizen.utxo.box.data.BoxDataSerializer`

Specific actions for extension of Coin-box

A Coin Box is a Box that has a value in ZEN. The creation process is the same just described, with only one extra action: a *Coin box class* needs to implement the `CoinsBox<P extends PublicKey25519Proposition>` interface, without the implementation of any additional function (i.e. it's a mixin interface).

Transaction extension

A transaction is the basic way to implement the application logic, by processing input Boxes that get unlocked and opened (or “spent”), and create new ones. All custom transactions inherited from `SidechainTransaction`. `SidechainNoncedTransaction` - class that helps to deal with output boxes nonces. `AbstractRegularTransaction` class helps to deal with ZenBoxes. To define a new custom transaction, you have to extend the `io.horizen.utxo.transaction.SidechainNoncedTransaction` class or `io.horizen.utxo.transaction.SidechainTransaction`. The most relevant methods of this class are detailed below:

- `public List<BoxUnlocker<Proposition>> unlockers()`

Defines the list of Boxes that are opened when the transaction is executed, together with the information (Proof) needed to open them. Each element of the returned list is an instance of `BoxUnlocker`, which is an interface with two methods:

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

The two methods define the id of the closed box to be opened and the proof that unlocks the proposition for that box. When a box is unlocked and opened, it is spent or “burnt”, i.e. it stops existing; as such, it will be removed from the wallet and the blockchain state. As a reminder, a value inside a box cannot be “updated”: the process requires to spend the box and create a new one with the updated values.

- `public List<Box<Proposition>> newBoxes()`

This function returns the list of new boxes which will be created by the current transaction. As a good practice, you should use the `Collections.unmodifiableList()` method to wrap the returned list into a not updatable Collection:

```
@Override
public List<Box<Proposition>> newBoxes() {
    List<Box<Proposition>> newBoxes = ..... //new boxes are created here
```

(continues on next page)

(continued from previous page)

```
//....  
return Collections.unmodifiableList(newBoxes);  
}
```

- `public long fee()` Returns the fee to be paid to execute this transaction.
- `public byte transactionTypeId()` Returns the type of this transaction. Each custom transaction must have its own unique type.
- `public boolean transactionSemanticValidity()` Confirms if a transaction is semantically valid, e.g. checks that `fee > 0`, `timestamp > 0`, etc. This function is not aware of the state of the sidechain, so it can't check, for instance, if the input is a valid Box.

SidechainNoncedTransaction has already implementation of `newBoxes` function. But it requires an implementation of abstract function `getOutputData` that provides list of output data of the transaction. `AbstractRegularTransaction` requires the implementation of `getCustomOutputData` for retrieving output custom data of the transaction. The output of other data in `AbstractRegularTransaction` is already collected in the `getOutputData` function, which also uses `getCustomOutputData`.

Apart from the semantic check, the Sidechain will need to make also sure that all transactions are compliant with the application logic and syntax. Such checks need to be implemented in the `validate()` method of the `custom ApplicationState` class.

Transactions that process Coins

A key element of sidechains is the ability to trade ZEN.

ZEN are represented as Coin boxes, that can be spent and created.

Transactions handling coin boxes will generally perform some basic, standard operations, such as:

- select and collect a list of coin boxes in input which sum up to a value that is equal or higher than the amount to be spent plus fee
- create a coin box with the change
- check that the sum of the input boxes + fee is equal to the sum of the output coin boxes.

Inside the `Lambo-registry` demo application, you can find an example of implementation of a transaction that handles regular coin boxes and implements the basic operations just mentioned: [io.horizen.lambo.car.transaction.AbstractRegularTransaction](#). Please note that, in a decentralized environment, transactions generally require the payment of a fee, so that their inclusion in a block can be rewarded and so incentivised. So, even if a transaction is not meant to process coin boxes, it still needs to handle coins to pay its fee.

Custom Proof / Proposition creation

A proposition is a locker for a box, and a proof is an unlocker for a box. How a box is locked and unlocked can be changed by the developer. For example, a custom box might require to be opened by two or more independent private keys. This kind of customization is achieved by defining custom Proposition and Proof.

- **Creating custom Proposition** You can create a custom proposition by implementing the `ProofOfKnowledgeProposition<S extends Secret>` interface. The generic parameter `S` represents the kind of private key used to unlock the proposition, e.g. you could use `PrivateKey25519`. Let's see how you could declare a new kind of Proposition that accepts two different public keys, and that can be opened by just one of two corresponding private keys:

```

public final class MultiProposition implements ProofOfKnowledgeProposition
↳<PrivateKey25519> {

    // Specify json attribute name for the firstPublicKeyBytes field.
    @JsonProperty("firstPublicKey")
    private final byte[] firstPublicKeyBytes;

    // Specify json attribute name for the secondPublicKeyBytes field.
    @JsonProperty("secondPublicKey")
    private final byte[] secondPublicKeyBytes;

    public MultiProposition(byte[] firstPublicKeyBytes, byte[]
↳secondPublicKeyBytes) {
        if(firstPublicKeyBytes.length != KEY_LENGTH)
            throw new IllegalArgumentException(String.format("Incorrect
↳firstPublicKeyBytes length, %d expected, %d found", KEY_LENGTH,
↳firstPublicKeyBytes.length));

        if(secondPublicKeyBytes.length != KEY_LENGTH)
            throw new IllegalArgumentException(String.format("Incorrect
↳secondPublicKeyBytes length, %d expected, %d found", KEY_LENGTH,
↳secondPublicKeyBytes.length));

        this.firstPublicKeyBytes = Arrays.copyOf(firstPublicKeyBytes, KEY_LENGTH);
        this.secondPublicKeyBytes = Arrays.copyOf(secondPublicKeyBytes, KEY_LENGTH);
    }

    public byte[] getFirstPublicKeyBytes() { return firstPublicKeyBytes;}
    public byte[] getScndPublicKeyBytes() { return secondPublicKeyBytes;}

    //other required methods for serialization omitted here:
    //byte[] bytes()
    //PropositionSerializer serializer();
}

```

- **Creating custom Proof interface** You can create a custom proof by implementing `Proof<P extends Proposition>`, where *P* is the Proposition class that this Proof can open. You also need to implement the boolean `isValid(P proposition, byte[] messageToVerify);` function; it checks and states whether Proof is valid for a given Proposition or not. For example, the Proof to open the “two public keys” Proposition shown above could be coded this way:

```

public class MultiSpendingProof extends Proof<MultiProposition> {

    protected final byte[] signatureBytes;

    public MultiSpendingProof(byte[] signatureBytes) {
        this.signatureBytes = Arrays.copyOf(signatureBytes, signatureBytes.
↳length);
    }

    @Override
    public boolean isValid(MultiProposition proposition, byte[] message) {
        return (
            Ed25519.verify(signatureBytes, message, proposition.
↳getFirstPublicKeyBytes()) ||
            Ed25519.verify(signatureBytes, message, proposition.
↳getSecondPublicKeyBytes())

```

(continues on next page)

(continued from previous page)

```

        );
    }

    //other required methods for serialization omitted here:
    //byte[] bytes();
    //ProofSerializer serializer();
    //byte proofTypeId();
}

```

Application State

If we consider the representation of a blockchain in a node as a finite state machine, then the application state can be seen as the state of all the “registers” of the machine at the present moment. The present moment starts when the most recent block is received (or forged!) by the node, and ends when a new one is received/forged. A new block updates the state, so it needs to be checked for both semantic and contextual validity; if ok, the state needs to be updated according to what is in the block. A customized blockchain will likely include custom data and transactions. The `ApplicationState` interface needs to be extended to code the rules that state validity of blocks and transactions, and the actions to be performed when a block modifies the state (“onApplyChanges”), and when it is removed (“onRollback”, blocks can be reverted!):

`ApplicationState`:

```

interface ApplicationState {
    void validate(SidechainStateReader stateReader, SidechainBlock block) throws
    ↳ IllegalArgumentException;

    void validate(SidechainStateReader stateReader, BoxTransaction<Proposition, Box
    ↳ <Proposition>> transaction) throws IllegalArgumentException;

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

    Try<ApplicationState> onRollback(byte[] blockId);

    boolean checkStoragesVersion(byte[] blockId);

    Try<ApplicationState> onBackupRestore(BoxIterator i);
}

```

An example might help to understand the purpose of these methods. Let’s assume, as we’ll see in the next chapter, that our sidechain can represent a physical car as a token, that is coded as a “CarBox”. Each CarBox token should represent a unique car, and that will mean having a unique VIN (Vehicle Identification Number): the sidechain developer will make `ApplicationState` store the list of all seen VINs, and reject transactions that create CarBox tokens with any preexisting VINs.

Then, the developer could implement the needed custom state checks in the following way:

```

public boolean validate(SidechainStateReader stateReader, BoxTransaction
    ↳ <Proposition, Box<Proposition>> transaction)

```

- Custom checks on transactions should be performed here. If the function throws exception, then the transaction is considered invalid. This method is called either before including a transaction inside the memory pool or before accepting a new block from the network.

```
void validate(SidechainStateReader stateReader, SidechainBlock block) throws ↳
↳IllegalArgumentException
```

- Custom block validation should happen here. If the function throws exception, then the block will not be accepted by the sidechain node. Note that each transaction contained in the block had been already validated by the previous method, so here you should include only block-related checks (e.g. check that two different transactions in the same block don't declare the same VIN car)

```
public boolean validate(SidechainStateReader stateReader, BoxTransaction
↳<Proposition, Box<Proposition>> transaction)
```

- Any specific action to be performed after applying the block to the State should be defined here.

```
public Try<ApplicationState> onApplyChanges(SidechainStateReader stateReader, ↳
↳byte[] version, List<Box<Proposition>> newBoxes, List<byte[]> ↳
↳boxIdsToRemove)
```

- Any specific action after a rollback of the state (for example, in case of fork/reverted block) should be defined here.

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

- This method checks that all the storages of the application which get updated by the sdk via the “onApplyChange” call above, have the version corresponding to the blockId passed as input parameter. This is useful when checking the alignment of sdk and application storages versions at node restart.

```
public boolean checkStoragesVersion(byte[] blockId)
```

- This method is used during the restore procedure. It can be useful if you want to perform some operations based on the restored boxes.

```
Try<ApplicationState> onBackupRestore(BoxIterator i);
```

Application Wallet

Every sidechain node has a local wallet associated to it, in a similar way as the mainchain Zend node wallet. The wallet stores the user secret info and related balances. It is initialized with the genesis account key and the ZEN amount transferred by the sidechain creation transaction. New private keys can be added by calling the http endpoint `/wallet/createPrivateKey25519`. The local wallet data is updated when a new block is added to the sidechain, and when blocks are reverted.

Developers can extend Wallet logic by defining a class that implements the interface [ApplicationWallet](#). The interface methods are listed below:

```
interface ApplicationWallet {
    void onAddSecret(Secret secret);

    void onRemoveSecret(Proposition proposition);

    void onChangeBoxes(byte[] version, List<Box<Proposition>> boxesToBeAdded, List
↳<byte[]> boxIdsToRemove);

    void onRollback(byte[] version);

    boolean checkStoragesVersion(byte[] blockId);
```

(continues on next page)

(continued from previous page)

```
void onBackupRestore(BoxIterator i);
}
```

As an example, the `onChangeBoxes` method gets called every time new blocks are added or removed from the chain; it can be used to implement for instance the update to a local storage of values that are modified by the opening and/or creation of specific box types. Similarly to `ApplicationState`, the `checkStoragesVersion` method is useful when checking the alignment of sdk and application wallet storages versions at node restart.

Sidechain Application Stopper

A user application should define a class that implements the interface `SidechainAppStopper`. The interface is listed below:

```
interface SidechainAppStopper {
    void stopAll();
}
```

The `stopAll()` method gets called when the node stop procedure is initiated. Such a procedure can be explicitly triggered via the API 'node/stop' or can be triggered when the JVM is shutting down, for instance when a SIGINT is received. In the custom implementation for instance, custom storages should be closed and any resources should be properly released. An example is provided in the "SimpleApp" with the `SimpleAppStopper.java` class.

Custom API creation

A user application can extend the default standard API (see chapter 6) and add custom API endpoints. For example if your application defines a custom transaction, you may want to add an endpoint that creates one.

To add custom API you have to create a class which extends the `io.horizen.api.http.ApplicationApiGroup` abstract class, and implements the following methods:

- `public String basePath()` returns the base path of this group of endpoints (the first part of the URL)
- `public List<Route> getRoutes()` returns a list of `Route` objects: each one is an instance of a `akka.HttpRoute` object and defines a specific endpoint url and its logic. To simplify the development, the `ApplicationApiGroup` abstract class provides a method (`bindPostRequest`) that builds a `akka.Route` that responds to a specific http request with an (optional) json body as input. This method receives the following parameters:
 - the endpoint path
 - the function to process the request
 - the class that represents the input data received by the HTTP request call

Example:

```
public List<Route> getRoutes() {
    List<Route> routes = new ArrayList<>();
    routes.add(bindPostRequest("createCar", this::createCar,
    ↪ CreateCarBoxRequest.class));
    routes.add(bindPostRequest("createCarSellOrder",
    ↪ this::createCarSellOrder, CreateCarSellOrderRequest.class));
    routes.add(bindPostRequest("acceptCarSellOrder",
    ↪ this::acceptCarSellOrder, SpendCarSellOrderRequest.class));
    routes.add(bindPostRequest("cancelCarSellOrder",
    ↪ this::cancelCarSellOrder, SpendCarSellOrderRequest.class));
}
```

(continues on next page)

(continued from previous page)

```

    return routes;
}

```

Let's look in more details at the 3 parameters of the `bindPostRequest` method.

- The endpoint path: defines the endpoint path, that appended to the `basePath` will represent the http endpoint url.

For example, if your API group has a `basePath` = “carApi”, and you define a route with endpoint path “createCar”, the overall url will be:

`http://<node_host>:<api_port>/carApi/createCar`

- The function to process the 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`.
- * `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. endpoints that can be called without a JSON body in the request

The format of the `ApiResponse` to be returned will be described later in this chapter.

- The class that represents the body in the HTTP request

This needs to be a java bean, defining some private fields and getter and setter methods for each field. Each field in the json input will be mapped to the corresponding field by name-matching.

For example to handle the following json body :

```

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

```

you should code a request class like this one:

```

public 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;
    }
}

```

(continues on next page)

(continued from previous page)

```
}

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

Submitting transaction can be operated with TransactionSubmitProvider

```
trait TransactionSubmitProvider {
    @throws(classOf[IllegalArgumentException])
    def submitTransaction(tx: BoxTransaction[Proposition, Box[Proposition]]): Unit

    def asyncSubmitTransaction(tx: BoxTransaction[Proposition, Box[Proposition]],
                               callback: (Boolean, Option[Throwable]) => Unit): Unit
}
```

For example

```
val transactionSubmitProvider: TransactionSubmitProviderImpl = new
↳ TransactionSubmitProviderImpl(sidechainTransactionActorRef)

val tryRes: Try[Unit] = Try {
    transactionSubmitProvider.submitTransaction(transaction)
}
tryRes match {
    case Success(_) => // expected behavior
    case Failure(exception) => fail("Transaction expected to be submitted successfully.
↳ ", exception)
}
```

asyncSubmitTransaction allows after submitting transaction apply callback function.

```
val transactionSubmitProvider: TransactionSubmitProviderImpl = new
↳ TransactionSubmitProviderImpl(sidechainTransactionActorRef)

def callback(res: Boolean, errorOpt: Option[Throwable]): Unit = synchronized {
    // Some operations executed after submitting transaction
}

// Start submission operation ...
transactionSubmitProvider.asyncSubmitTransaction(transaction, callback)
```

Also there are available providers for retrieving NodeView and Secret submission

```
trait NodeViewProvider {
    def getNodeView(view: SidechainNodeView => Unit)
}
```

```
public interface SecretSubmitHelper {
    void submitSecret(Secret secret) throws IllegalArgumentException;
}
```

API response classes

The function that processes the request must return an object of type `io.horizen.api.http.ApiResponse`. In most cases, we can have two different responses: either the operation is successful, or an error has occurred during the API request processing.

For a successful response, you have to: - define an object implementing the `SuccessResponse` interface - add the annotation `@JsonView(Views.Default.class)` on top of the class, to allow the automatic conversion of the object into a json format. - add some getters representing the values you want to return.

For example, if a string should be returned, then the following response class can be defined:

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

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

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

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

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

If an error is returned, then the response will implement the `ErrorResponse` interface. The `ErrorResponse` interface has the following default functions implemented:

``public String code()`` – error code

``public String description()`` – error description

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

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

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

Custom api group injection:

Finally, you have to instruct the SDK to use your `ApiGroup`. This can be done with Guice, by binding the `“Custom-ApiGroups”` field:

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

Backup and restore procedure

This mechanism was introduced to make it possible to bootstrap a sidechain starting from a “snapshot” taken from an another sidechain of the same kind. This can be useful in the unfortunately case of a sidechain that get ceased. With this procedure you are able to make a backup of your unspent non-coin boxes contained in your ceased sidechain and start a new sidechain that contains these boxes.

Important notes:

- This procedure allows to backup and restore only NON-COIN boxes.
- These restored boxes are not propagated over the network. This means also that, in a re-bootstrapped sidechain, every single node must have these boxes inside it’s own data directory.
- The nodes must include the backup inside their data directory BEFORE they are started for the first time.

Backup procedure

The SDK contains a Class called `SidechainAppBackup` that can be referenced from the application level to perform a backup.

```
class SidechainBackup @Inject()
    (@Named("CustomBoxSerializers") val customBoxSerializers: JHashMap[JByte,
↳BoxSerializer[SidechainTypes#SCB]],
    @Named("BackupStorage") val backUpStorage: Storage,
    @Named("BackUpper") val backUpper : BoxBackupInterface
    ) extends ScorexLogging
{
    def createBackup(stateStoragePath: String, sidechainBlockIdToRollback: String,
↳copyStateStorage: Boolean): Unit = {
        ...
    }
}
```

It requires that the application level injects the following objects:

- `CustomBoxSerializer`: Map containing a serializer for every kind of new boxes added.
- `BackupStorage`: The Storage that will contains the backup.
- `BackUpper`: A class that implements the interface `BoxBackupInterface` that will be called to perform the backup.

The method `createBackup` starts the backup procedure by calling the function `BackUpper.backup`. It takes as parameters:

- `stateStoragePath`: File path to the `SidechainStateStorage`.
- `sidechainBlockIdToRollback`: Sidechain block id used for the storage rollback.
- `copyStateStorage`: If True performs a copy of the `SidechainStateStorage` before rollback it in order to avoid its permanent corruption (this is a one way procedure, the Storage can’t be used anymore after the rollback).

The `BoxBackupInterface` interface declares a method `backup` that should be implemented by the extending class.

```
public interface BoxBackupInterface {
    void backup(BoxIterator source, BackupStorage db) throws Exception;
}
```

The `backup` method receives an iterator over the Storage that will be taken as a source for the backup (typically it would be the `SidechainStateStorage`), and the Storage used to store the backup. You can use the method `nextBox` from this iterator to retrieve the next non-coin box from the Storage.

Important notes:

In order to maintain a consistency between what the Mainchain knows about the Sidechain and what the Sidechain contains itself, if you want to perform a Backup of a ceased Sidechain, you should rollback the Storage to the version that contains the Mainchain block calculated by the following formula:

$$\text{Genesis_MC_block_height} + (\text{current_epoch} - 2) * \text{withdrawalEpochLength} - 1$$

This can be easily done by calling the endpoint `/backup/getSidechainBlockIdForBackup` and pass the block id obtained to the method `createBackup`.

Restore procedure

The restore procedure is automatically invoked when a Sidechain node starts from an empty blockchain. Before the application of the genesis block, the node is able to detect if there is a Backup Storage to restore into its data directory; in such a case, it performs several iterations over it in order to populate the other storages. The Backup Storage is scanned 4 times:

- First scan used to populate the `SidechainStateStorage` with all the boxes found in the `BackupStorage`.
- Second scan used to add the boxes owned by a node wallet proposition inside the `WalletBoxStorage`. In this way you will be able to see the restored boxes inside your wallet (only if you have the corresponding proposition imported in your wallet) and spend them.
- Third scan performed in the application level (`ApplicationState`). This is useful if you have some custom storages that you want to populate with the information taken from these boxes. You should override the method `public Try<ApplicationState> onBackupRestore(BoxIterator boxIterator)` inside the `ApplicationState`.
- Fourth scan performed in the application level (`ApplicationWallet`). This is useful if you want to perform some operations in your wallet based on the information taken from these boxes. You should override the method `public void onBackupRestore(BoxIterator i)` inside the `ApplicationWallet`.

Important notes:

The Backup Storage must be present inside your node data directory before starts the node for the first time.

The procedure fails if just a single coin box is found inside the Backup Storage.

If you own some of the restored boxes and you want to see them inside your wallet, you should add your secrets inside the config file of your node (before start the node for the first time). You can add your secrets inside the section `Wallet.genesisSecrets` by appending “00” at the beginning of your secret in case it is a `PrivateKey25519`, “03” if it is a `VrfPrivateKey` or “04” if it is a `SchnorrPrivateKey`.

Logging

The SDK logging system is based on the Log4J library. To fire additional log messages in your application code, just declare a log4j Logger in your class and use it:

```
public class MyCustomClass {

    Logger logger = LogManager.getLogger(MyCustomClass.class);

    public MyCustomClass() {
```

(continues on next page)

(continued from previous page)

```
        logger.debug("This is an example  debug message inside the constructor");
    }
}
```

Note: do not add any log4j library in your application pom file, as it is already loaded as a nested dependency of the SDK.

You can rely on the default logging configuration (based on Log4J library) and change just a few parameters inside the application configuration file, or override it completely with a custom one.

Default log configuration

By default, a predefined `log4j2.xml` configuration is used. It redirects all logging messages to the system console and to a filesystem log file, rotated and gzipped when it reaches 50MB size (only the latest 10 are then retained).

The following dynamic parameters are taken from the application configuration file, and can be changed there at any time:

```
scorex {
    ...

    logDir = /tmp/scorex/data/log

    logInfo {
        logFileName = "debug.log"
        logFileLevel = "warn"
        logConsoleLevel = "debug"
    }

    ...
}
```

- `scorex.logDir`: base folder where the log files are generated, injected in the `log4j.xml` in the placeholder `${sys:logDir}`
- `scorex.logInfo.logFileName` : log filename, injected in the `log4j.xml` in the placeholder `${sys:logFileName}`
- `scorex.logInfo.logFileLevel` : log level used for the file appender, injected in the `log4j.xml` in the placeholder `${sys:logFileLevel}`
- `scorex.logInfo.logConsoleLevel` : log level used for the console appender, injected in the `log4j.xml` in the placeholder `${sys:logConsoleLevel}`

Customized log configuration

If you add a custom `log4j2.xml` in your application's classpath, it will override the default one.

Same placeholders described before are available also here.

Fork Configuration

In the new versions of the SDK the backward incompatible changes may be introduced. For example, the changes to the consensus protocol or another kind of core transaction. That will lead to the hard fork. So, already running sidechains must be very careful and may have a unified mechanism to upgrade the nodes and activate such changes. For this every sidechain application should use *ForkConfigurator* to specify the activation points for regtest, testnet and mainnet networks. SDK implements consensus, which is measured in the fixed time epochs, but an epoch may have variant number of blocks associated with it. That's why forks activation points (caused by consensus changes) are specified in the consensus epoch numbers, rather than blockchain heights. Every sidechain network knows what is the current epoch and able to choose any epoch in the future for the fork activation while upgrading SDK version.

In the following example the first fork activates in regtest at epoch 10, in testnet at epoch 20, and in mainnet at epoch 30. The next fork is activated always not earlier than the previous one (but can be activated at the same time).

```
public class MyAppForkConfigurator extends ForkConfigurator {
    @Override
    public ForkConsensusEpochNumber getSidechainFork1() {
        return new ForkConsensusEpochNumber(/*regtest*/ 10, /*testnet*/ 20, /
↪ *mainnet*/30);
    }

    @Override
    public ForkConsensusEpochNumber getSidechainFork2() {
        return new ForkConsensusEpochNumber(/*regtest*/ 20, /*testnet*/ 20, /
↪ *mainnet*/30);
    }
}
```

3.1.9 Car Registry Tutorial

Car Registry App High-Level Overview

The [Lambo-Registry app](#) is a demo dApp implemented as a sidechain, that makes use of custom data and logic. It was developed to serve as a practical example of how the SDK can be extended. From a functional point of view, the application acts as a repository of existing cars and their owners, and offers to its users the possibility to sell and buy cars. It is a demo application, so it does not include all the needed checks and functionalities that a production application would need; for instance, users are now able to register a car by broadcasting a simple “Car Declaration” transaction. We could think that, in a real-world scenario, the ability to declare the existence of a new car in the sidechain, might be instead subject to the inclusion in the transaction of a certificate signed by the Department of Motor Vehicles, that guarantees that the car exists and it's owned by a user with a specified public key.

To sum up, the Lambo-Registry applications just accepts transactions that create cars, and then provides the following functionalities:

1. It stores information that identifies a specific car, such as vehicle identification number (VIN), model, production year, colour.
2. It allows car owners to be able to prove their ownership of the cars anonymously.
3. It gives the possibility to sell a car in exchange for ZEN.

User stories:

As usual, the first step of software development is the analysis. Let's list the functional requirements of our dApp as some simple user requests ("R"), and then the associated design decisions ("D"):

R: I want to add my car to the Car Registry App.

D: We'll introduce a transaction that creates a "Car Entry Box", with all the vehicle's identification information (VIN, manufacturer, model, year, registration number). The proposition associated to this box is the public key of the owner of the car. When a Car Box is created, the sidechain should verify that the vehicle identification information are unique to this sidechain.

R: I want to sell my car.

D: We'll introduce a "Car Sell Order Box" that includes the vehicle's information and its price in ZEN. Cars can exist in the sidechain either as a "Car Entry Box" or as a "Car Sell Order Box", but not both at the same time. The Car Sell Order Box will contain also the public key of the prospective buyer, so we assume that some kind of negotiation/agreement between the seller and the buyer took place off-chain. When a sell order is created, the sidechain will have to verify that there is no other active sell order for the same vehicle.

R: I want to buy a car.

D: To buy a car, the user will have to create a new transaction that accepts a sell order. That sell order must specify the user's public key. The transaction will create a new Car Entry Box, closed by the new owner's public key as proposition. The transaction will also transfer the correct amount of ZEN coins from the buyer to the seller.

R: I've changed my mind, and don't want to sell my car any more.

D: If the sell order is still active, it can be recalled by its creator. The car owner will create a new transaction containing the Car Sell Order as input, and a Car Entry Box closed by his public key as output.

R: I want to see all the cars I own, and the ones that have been offered to me.

D: This piece of information will be managed by ApplicationWallet. We can use the SDK standard endpoint "wallet/allBlocks" and filter by box type.

We can now start the development process, by addressing the data representation.

Boxes

When designing a new application, the preliminary step is to identify the needed custom boxes and their respective properties. Boxes are the basic objects that describe the state of our application. The Lambo-registry example implements the following custom boxes:

- **CarBox** A Box that represents a car instance. The following properties were selected to describe a car:
 - vehicle identification number (vin)
 - year of production
 - model
 - color
- **CarSellOrderBox** A Box that represents the intention to sell a car to someone. It has the same properties of a car, a price (in ZEN), and it is closed by a special proposition which can be opened either by the seller (to remove the car from sale) or the buyer (to complete the purchase).

Let's have a closer look at the code that defines a CarBox:


```

@JsonView(Views.Default.class)
@JsonIgnoreProperties({"carId", "value"})
public final class CarBox extends AbstractBox<PublicKey25519Proposition,
↳CarBoxData, CarBox> {

    public CarBox(CarBoxData boxData, long nonce) {
        super(boxData, nonce);
    }

    @Override
    public BoxSerializer serializer() {
        return CarBoxSerializer.getSerializer();
    }

    @Override
    public byte boxTypeId() {
        return CarBoxId.id();
    }

    CarBoxData getBoxData() {
        return boxData;
    }

    // Set car attributes getters, that is used to automatically construct
↳JSON view:
    public String getVin() {
        return boxData.getVin();
    }

    public int getYear() {
        return boxData.getYear();
    }

    public String getModel() {
        return boxData.getModel();
    }

    public String getColor() {
        return boxData.getColor();
    }

    public byte[] getCarId() {
        return Bytes.concat(
            getVin().getBytes(),
            Ints.toByteArray(getYear()),
            getModel().getBytes(),
            getColor().getBytes()
        );
    }
}

```

Let's start from the top declaration:

```

@JsonView(Views.Default.class)
@JsonIgnoreProperties({"carId", "value"})
public final class CarBox extends AbstractBox
↳<PublicKey25519Proposition, CarBoxData, CarBox> {

```

Our class extends the *AbstractBox* default class, is locked by a standard *PublicKey25519Proposition* and keeps all its properties into an object of type *CarBoxData*. The annotation *@JsonView* instructs the SDK to use a default viewer to convert an instance of this class into JSON format when a *CarBox* is included in the result of an http API endpoint. With that, there is no need to write the conversion code: all the properties associated to getter methods of the class are automatically converted to json attributes. For example, since our class has a getter method “*getModel()*”, the json will contain the attribute “model” with its value. We can specify some properties that must be excluded from the json output with the *@JsonIgnoreProperties* annotation.

The constructor of boxes extending *AbstractBox* is very simple, it just calls the superclass with two parameters: the *BoxData* and the nonce.

```
public CarBox(CarBoxData boxData, long nonce) {  
    super(boxData, nonce);  
}
```

The *BoxData* is a container of all the properties of our *Box*, we’ll have a look at it later. The nonce is a random number that allows the generation of different hash values also if the inner properties of two boxes have the same values.

```
@Override  
public byte boxTypeId() {  
    return CarBoxId.id();  
}
```

The method *boxTypeId()* returns the id of this box type: every custom box needs to have a unique type id inside the application. Note that the ids of custom boxes can overlap with the ids of the standard boxes (e.g. you can re-use the id type 1 that is already used for standard coin boxes).

The next method is used for serialization and deserialization of our *Box*: it defines the serializer to be used to generate a byte array from the box and to obtain the box back from the byte array:

```
@Override  
public BoxSerializer serializer() {  
    return CarBoxSerializer.getSerializer();  
}
```

The last methods of the class are just the getters of the box properties. In particular *getCarId()* is an example of a property that is the result of operations performed on other stored properties.

There are three more classes related to our *CarBox*: the *boxdata* and the *serializers*. Let’s have a closer look at them.

BoxData

BoxData allows us to group all the box properties and their serialization and deserialization logic in a single container object. Although its use is not mandatory (you can define field properties directly inside the *Box*), it is required if you choose to extend the base class *AbstractBox*, as we did for the *CarBox*, and it is in any case a good practice.

```
@JsonView(Views.Default.class)  
public final class CarBoxData extends AbstractBoxData  
    <PublicKey25519Proposition, CarBox, CarBoxData> {  
  
    // In CarRegistry example we defined 4 main car attributes:  
    private final String vin;    // Vehicle Identification Number  
    private final int year;     // Car manufacture year
```

(continues on next page)

(continued from previous page)

```

private final String model; // Car Model
private final String color; // Car color

public CarBoxData(PublicKey25519Proposition proposition, String_
↪vin,
                int year, String model, String color) {
    super(proposition, 1);
    this.vin = vin;
    this.year = year;
    this.model = model;
    this.color = color;
}

public String getVin() {
    return vin;
}

public int getYear() {
    return year;
}

public String getModel() {
    return model;
}

public String getColor() {
    return color;
}

@Override
public CarBox getBox(long nonce) {
    return new CarBox(this, nonce);
}

@Override
public byte[] customFieldsHash() {
    return Blake2b256.hash(
        Bytes.concat(
            vin.getBytes(),
            Ints.toByteArray(year),
            model.getBytes(),
            color.getBytes()));
}

@Override
public BoxDataSerializer serializer() {
    return CarBoxDataSerializer.getSerializer();
}

@Override
public String toString() {
    return "CarBoxData{" +
        "vin=" + vin +
        ", proposition=" + proposition() +
        ", model=" + model +
        ", color=" + color +
        ", year=" + year +

```

(continues on next page)

(continued from previous page)

```
        '}';  
    }  
}
```

Let's look in detail at the code above, starting from the beginning:

```
@JsonView(Views.Default.class)  
public final class CarBoxData extends AbstractBoxData  
    <PublicKey25519Proposition, CarBox, CarBoxData> {
```

Also this time, we have a basic class we can extend: `AbstractBoxData`.

```
public CarBoxData(PublicKey25519Proposition proposition, String vin,  
                 int year, String model, String color) {  
    super(proposition, 1);  
    this.vin = vin;  
    this.year = year;  
    this.model = model;  
    this.color = color;  
}
```

The constructor receives all the box properties, and the proposition that locks it. The proposition is passed up to the superclass constructor, which also receives a long number representing the ZEN value of the box. For boxes that don't handle coins (like this one) we can just pass a constant value 1.

```
@Override  
public CarBox getBox(long nonce) {  
    return new CarBox(this, nonce);  
}
```

The `getBox(long nonce)` is a helper method used to generate a new box from the content of this boxdata.

```
@Override  
public byte[] customFieldsHash() {  
    return Blake2b256.hash(  
        Bytes.concat(  
            vin.getBytes(),  
            Ints.toByteArray(year),  
            model.getBytes(),  
            color.getBytes());  
    }  
}
```

The method `customFieldsHash()` is used by the sidechain to generate a unique hash for each box instance: it needs to be defined in a way such that different property values of a boxdata always produce a different hash value. To achieve this, the code uses a scorex helper class (`scorex.crypto.hash.Blake2b256`) that generates a hash from a bytearray; the bytearray is the concatenation of all the properties values.

Boxdata, as Box, has some methods to define its serializer, and a unique type id:

```
@Override  
public BoxDataSerializer serializer() {  
    return CarBoxDataSerializer.getSerializer();  
}  
  
@Override  
public byte boxDataTypeId() {
```

(continues on next page)

(continued from previous page)

```

    return CarBoxDataId.id();
}

```

As expected, the class includes all the getters of every custom property (*getModel()*, *getColor()* etc..). Also, the *toString()* method is redefined to print out the content of boxdata in a more user-friendly format:

```

@Override
public String toString() {
    return "CarBoxData{" +
        "vin=" + vin +
        ", proposition=" + proposition() +
        ", model=" + model +
        ", color=" + color +
        ", year=" + year +
        '}';
}

```

BoxSerializer and BoxDataSerializer

Serializers are companion classes that are invoked by the SDK every time a Scorex reader and writer needs to deserialize or serialize a Box. We define one serializer/deserializer both for box and for boxdata. As you can see in the code below, since the “heavy” byte handling happens inside boxdata, their logic is very simple: they just call the right methods already defined in the associated (Box or BoxData) objects.

```

public final class CarBoxSerializer implements BoxSerializer<CarBox> {

    private static final CarBoxSerializer serializer = new
↳CarBoxSerializer();

    private CarBoxSerializer() {
        super();
    }

    public static CarBoxSerializer getSerializer() {
        return serializer;
    }

    @Override
    public void serialize(CarBox box, Writer writer) {
        writer.putLong(box.nonce());
        CarBoxDataSerializer.getSerializer().serialize(box.getBoxData(),
↳writer);
    }

    @Override
    public CarBox parse(Reader reader) {
        long nonce = reader.getLong();
        CarBoxData boxData = CarBoxDataSerializer.getSerializer().
↳parse(reader);

        return new CarBox(boxData, nonce);
    }
}

```

```

public final class CarBoxDataSerializer implements BoxDataSerializer
↳<CarBoxData> {

    private static final CarBoxDataSerializer serializer = new
↳CarBoxDataSerializer();

    private CarBoxDataSerializer() {
        super();
    }

    public static CarBoxDataSerializer getSerializer() {
        return serializer;
    }

    @Override
    public void serialize(CarBoxData boxData, Writer writer) {
        PublicKey25519PropositionSerializer.getSerializer().
↳serialize(boxData.proposition(), writer);
        byte[] vinBytes = boxData.getVin().getBytes(StandardCharsets.UTF_8);
        writer.putInt(vinBytes.length);
        writer.putBytes(vinBytes);
        writer.putInt(boxData.getYear());
        byte[] modelBytes = boxData.getModel().getBytes(StandardCharsets.UTF_
↳8);
        writer.putInt(modelBytes.length);
        writer.putBytes(modelBytes);
        byte[] colorBytes = boxData.getColor().getBytes(StandardCharsets.UTF_
↳8);
        writer.putInt(colorBytes.length);
        writer.putBytes(colorBytes);
    }

    @Override
    public CarBoxData parse(Reader reader) {
        PublicKey25519Proposition proposition =
↳PublicKey25519PropositionSerializer.getSerializer().parse(reader);
        int vinBytesLength = reader.getInt();
        String vin = new String(reader.getBytes(vinBytesLength),
↳StandardCharsets.UTF_8);
        int year = reader.getInt();
        int modelBytesLength = reader.getInt();
        String model = new String(reader.getBytes(modelBytesLength),
↳StandardCharsets.UTF_8);
        int colorBytesLength = reader.getInt();
        String color = new String(reader.getBytes(colorBytesLength),
↳StandardCharsets.UTF_8);
        return new CarBoxData(proposition, vin, year, model, color);
    }
}

```

Transactions

If Boxes are the objects that describe the state of our application, transactions are the actions that can describe the application state. They typically do that by opening (and therefore removing) some boxes (“input”), and creating new ones (“output”).

Our Car Registry application defines the following custom transactions:

- **CarDeclarationTransaction** - a transaction that declares a new car (by creating a new CarBox).
- **SellCarTransaction** - it creates a sell order for a car: a CarBox is “spent”, and a CarSellOrderBox containing all the data of the car to be sold is created.
- **BuyCarTransaction** - this transaction is used either by the buyer to accept the sell order, or by the seller to cancel it. It opens a CarSellOrderBox, and creates a CarBox (if it’s a sell order cancellation, the new CarBox will be assigned to the original owner).

Let’s look at the code of the last one, BuyCarTransaction, that is slightly more complicated than the other two:

```
public final class BuyCarTransaction extends AbstractRegularTransaction {

    private final CarBuyOrderInfo carBuyOrderInfo;

    public final static byte BUY_CAR_TRANSACTION_VERSION = 1;

    private byte version;

    public BuyCarTransaction(List<byte[]> inputZenBoxIds,
                             List<Signature25519> inputZenBoxProofs,
                             List<ZenBoxData> outputZenBoxesData,
                             CarBuyOrderInfo carBuyOrderInfo,
                             long fee,
                             byte version) {
        super(inputZenBoxIds, inputZenBoxProofs, outputZenBoxesData, fee);
        this.carBuyOrderInfo = carBuyOrderInfo;
        this.version = version;
    }

    // Specify the unique custom transaction id.
    @Override
    public byte transactionTypeId() {
        return BuyCarTransactionId.id();
    }

    @Override
    protected List<BoxData<Proposition, Box<Proposition>>>
↳getCustomOutputData() {
        ArrayList<BoxData<Proposition, Box<Proposition>>> customOutputData =
↳new ArrayList<>();
        customOutputData.add((BoxData) carBuyOrderInfo.
↳getNewOwnerCarBoxData());
        if(!carBuyOrderInfo.isSpentByOwner())
            customOutputData.add((BoxData) carBuyOrderInfo.
↳getPaymentBoxData());

        return customOutputData;
    }

    @Override
```

(continues on next page)

(continued from previous page)

```

public byte[] customDataMessageToSign() {
    return new byte[0];
}

@Override
public byte[] customFieldsData() {
    return carBuyOrderInfo.getNewOwnerCarBoxData().bytes();
}

@Override
public byte version() {
    return version;
}

// Override unlockers to contains ZenBoxes from the parent class_
→ appended with CarSellOrderBox entry.
@Override
public List<BoxUnlocker<Proposition>> unlockers() {
    // Get Regular unlockers from base class.
    List<BoxUnlocker<Proposition>> unlockers = super.unlockers();

    BoxUnlocker<Proposition> unlocker = new BoxUnlocker<Proposition>() {
        @Override
        public byte[] closedBoxId() {
            return carBuyOrderInfo.getCarSellOrderBoxToOpen().id();
        }

        @Override
        public Proof boxKey() {
            return carBuyOrderInfo.getCarSellOrderSpendingProof();
        }
    };
    // Append with the CarSellOrderBox unlocker entry.
    unlockers.add(unlocker);

    return unlockers;
}

// Define object serialization, that should serialize both parent class_
→ entries and CarBuyOrderInfo as well
void serialize(Writer writer) {
    writer.put(version());
    writer.putLong(fee());

    writer.putInt(inputZenBoxIds.size());
    for(byte[] id: inputZenBoxIds)
        writer.putBytes(id);

    zenBoxProofsSerializer.serialize(inputZenBoxProofs, writer);
    zenBoxDataListSerializer.serialize(outputZenBoxesData, writer);
    CarBuyOrderInfoSerializer.getSerializer().serialize(carBuyOrderInfo,
→ writer);
}

static BuyCarTransaction parse(Reader reader) {
    byte version = reader.getByte();
    long fee = reader.getLong();

```

(continues on next page)

(continued from previous page)

```

        int inputBytesIdsLength = reader.getInt();
        int idLength = NodeViewModifier$.MODULE$.ModifierIdSize();
        List<byte[]> inputZenBoxIds = new ArrayList<>();
        while(inputBytesIdsLength-- > 0)
            inputZenBoxIds.add(reader.getBytes(idLength));

        List<Signature25519> inputZenBoxProofs = zenBoxProofsSerializer.
↪parse(reader);
        List<ZenBoxData> outputZenBoxesData = zenBoxDataListSerializer.
↪parse(reader);
        CarBuyOrderInfo carBuyOrderInfo = CarBuyOrderInfoSerializer.
↪getSerializer().parse(reader);

        return new BuyCarTransaction(inputZenBoxIds, inputZenBoxProofs, ↪
↪outputZenBoxesData,
            carBuyOrderInfo, fee, version);
    }

    // Set specific Serializer for BuyCarTransaction class.
    @Override
    public TransactionSerializer serializer() {
        return BuyCarTransactionSerializer.getSerializer();
    }
}

```

Let's start from the top declaration:

```

public final class BuyCarTransaction extends ↪
↪AbstractRegularTransaction {

```

Our class extends the *AbstractRegularTransaction* default class, an abstract class designed to handle regular coin boxes. Since blockchain transactions usually require the payment of a fee (including the three custom transactions of our Car Registry application), and to pay a fee you need to handle coin boxes, usually custom transactions will extend this abstract class.

```

public BuyCarTransaction(List<byte[]> inputZenBoxIds,
                        List<Signature25519> inputZenBoxProofs,
                        List<ZenBoxData> outputZenBoxesData,
                        CarBuyOrderInfo carBuyOrderInfo,
                        long fee,
                        byte version) {
    ↪super(inputZenBoxIds, inputZenBoxProofs, outputZenBoxesData, ↪
↪fee);
    this.carBuyOrderInfo = carBuyOrderInfo;
    this.version = version;
}

```

The constructor receives all the parameters related to regular boxes handling (box ids to be opened, proofs to open them, regular boxes to be created, fee to be paid), and pass them up to the superclass. Moreover, it receives all other parameters specifically related to the custom boxes; in our example, the transaction needs info about the sell order that it needs to open, and it finds in the *CarBuyOrderInfo* object.

```

@Override
public List<BoxUnlocker<Proposition>> unlockers() {
    // Get Regular unlockers from base class.
}

```

(continues on next page)

(continued from previous page)

```

List<BoxUnlocker<Proposition>> unlockers = super.unlockers();

BoxUnlocker<Proposition> unlocker = new BoxUnlocker<Proposition>() {
    @Override
    public byte[] closedBoxId() {
        return carBuyOrderInfo.getCarSellOrderBoxToOpen().id();
    }

    @Override
    public Proof boxKey() {
        return carBuyOrderInfo.getCarSellOrderSpendingProof();
    }
};
unlockers.add(unlocker);
return unlockers;
}

```

The *unlockers()* method must return a list of *BoxUnlocker*'s, that contains the boxes which will be opened by this transaction, and the proofs to open them. The list returned from the superclass (in the first line of the method) contains the unlockers for the coin boxes, and it is combined with the unlocker for the *CarSellOrderBox*. As you can see we have used an inline declaration for the new unlocker, since it is a very simple object that has only two methods, one returning the box id to open and the other one the proof to open it.

```

@Override
public byte transactionTypeId() {
    return BuyCarTransactionId.id();
}

```

Just like with boxes, also each transaction type must have a unique id, returned by the method *transactionTypeId()*.

The last three methods of the class are related to the serialization handling. The approach is very similar to what we saw for boxes: the methods *bytes()* and *parseBytes(byte[] bytes)* perform a “two-way conversion” into and from an array of bytes, while the *serializer()* method returns the serializer helper to operate with Scorex reader's and writer's.

As we did with the *CarBox*, also here we have chosen to code the low level “byte handling” logic inside the two methods *serialize()* and *parse(Reader reader)*, keeping a very simple implementation for the serializer:

```

public final class BuyCarTransactionSerializer implements
↳ TransactionSerializer<BuyCarTransaction> {

    private static final BuyCarTransactionSerializer serializer = new
↳ BuyCarTransactionSerializer();

    private BuyCarTransactionSerializer() {
        super();
    }

    public static BuyCarTransactionSerializer getSerializer() {
        return serializer;
    }

    @Override
    public void serialize(BuyCarTransaction transaction, Writer writer) {
        transaction.serialize(writer);
    }

    @Override

```

(continues on next page)

(continued from previous page)

```

public BuyCarTransaction parse(Reader reader) {
    return BuyCarTransaction.parse(reader);
}
}

```

One of the parameters of the class constructor is `CarBuyOrderInfo`, an object that contains the needed info about the sell order we are handling. Let's take a look at its implementation:

```

public final class CarBuyOrderInfo implements Serializable {
    final CarSellOrderBox carSellOrderBoxToOpen; // Sell order box
    ↪to be spent in BuyCarTransaction
    final SellOrderSpendingProof proof; // Proof to unlock
    ↪the box above

    public CarBuyOrderInfo(CarSellOrderBox carSellOrderBoxToOpen,
    ↪SellOrderSpendingProof proof) {
        this.carSellOrderBoxToOpen = carSellOrderBoxToOpen;
        this.proof = proof;
    }

    public CarSellOrderBox getCarSellOrderBoxToOpen() {
        return carSellOrderBoxToOpen;
    }

    public SellOrderSpendingProof getCarSellOrderSpendingProof() {
        return proof;
    }

    // Recreates output CarBoxData with the same attributes
    ↪specified in CarSellOrder.
    // Specifies the new owner depends on proof provided:
    // 1) if the proof is from the seller then the owner remain the
    ↪same
    // 2) if the proof is from the buyer then it will become the new
    ↪owner
    public CarBoxData getNewOwnerCarBoxData() {
        PublicKey25519Proposition proposition;
        if(proof.isSeller()) {
            proposition = new
    ↪PublicKey25519Proposition(carSellOrderBoxToOpen.proposition().
    ↪getOwnerPublicKeyBytes());
        } else {
            proposition = new
    ↪PublicKey25519Proposition(carSellOrderBoxToOpen.proposition().
    ↪getBuyerPublicKeyBytes());
        }

        return new CarBoxData(
            proposition,
            carSellOrderBoxToOpen.getVin(),
            carSellOrderBoxToOpen.getYear(),
            carSellOrderBoxToOpen.getModel(),
            carSellOrderBoxToOpen.getColor()
        );
    }
}

```

(continues on next page)

(continued from previous page)

```

// Check if proof is provided by Sell order owner.
public boolean isSpentByOwner() {
    return proof.isSeller();
}

// Coins to be paid to the owner of Sell order in case if Buyer
↳spent the Sell order.
public ZenBoxData getPaymentBoxData() {
    return new ZenBoxData(
        new PublicKey25519Proposition(carSellOrderBoxToOpen.
↳proposition().getOwnerPublicKeyBytes()),
        carSellOrderBoxToOpen.getPrice()
    );
}

@Override
public byte[] bytes() {
    return serializer().toBytes(this);
}

@Override
public ScorexSerializer<BytesSerializable> serializer() {
    return (ScorexSerializer) CarBuyOrderInfoSerializer.
↳getSerializer();
}
}

```

If you look at the code above, you can see that this object is not much more than a container of the information that needs to be processed: the *CarSellOrderBox* that should be opened, and the proof to open it. It then includes their getters, and a couple of “utility” methods: *getNewOwnerCarBoxData()* and *getPaymentBoxData()*. The first one, *getNewOwnerCarBoxData()*, creates a new *CarBox* with the same properties of the sold car, and “assigns” it (by locking it with the right proposition) to either the buyer or the seller, depending on who opened the order.

```

public CarBoxData getNewOwnerCarBoxData() {
    PublicKey25519Proposition proposition;
    if(proof.isSeller()) {
        proposition = new
↳PublicKey25519Proposition(carSellOrderBoxToOpen.proposition().
↳getOwnerPublicKeyBytes());
    } else {
        proposition = new
↳PublicKey25519Proposition(carSellOrderBoxToOpen.proposition().
↳getBuyerPublicKeyBytes());
    }
    return new CarBoxData(
        proposition,
        carSellOrderBoxToOpen.getVin(),
        carSellOrderBoxToOpen.getYear(),
        carSellOrderBoxToOpen.getModel(),
        carSellOrderBoxToOpen.getColor()
    );
}

```

The second one, *getPaymentBoxData()*, creates a coin box with the payment of the order price to the seller (it will be used only if the buyer accepts the order):

```

public ZenBoxData getPaymentBoxData() {
    return new ZenBoxData(
        new PublicKey25519Proposition(carSellOrderBoxToOpen.
    ↪proposition().getOwnerPublicKeyBytes()),
        carSellOrderBoxToOpen.getPrice()
    );
}

```

Also this time we have the methods to serialize and deserialize the object: since the `CarBuyOrderInfo` is a property of our transaction and the transaction can be serialized, we need to be able to serialize and deserialize it as well.

Now that we have seen how a transaction is built, you may wonder how it can be created and submitted to the sidechain. This could be achieved in several ways, depending on the needs of our application, e.g. by using an RPC command, a code defined trigger, an offline wallet that creates the byte-array of the transaction and sends it through the default API method `'transaction/sendTransaction'`, ... One of the most common ways to support the creation of a custom transaction is by extending the default API endpoints, and add a new custom local wallet endpoint to let the user create it via HTTP. We will look into that at the end of this chapter.

Custom proof and proposition

A proposition is a box locker, and a proof is its unlocker. The SDK offers default Propositions and Proofs, and a developer can define custom ones.

Inside the Lambo Registry application, you can find a custom proposition: `SellOrderProposition`. It requires two public keys, while the corresponding proof (`SellOrderSpendingProof`) is able to unlock it by supplying only one of those two keys.

Let's look at them, starting with the `SellOrderProposition`:

```

@JsonView(Views.Default.class)
public final class SellOrderProposition implements ↵
    ↪ProofOfKnowledgeProposition<PrivateKey25519> {
    static final int KEY_LENGTH = Ed25519.publicKeyLength();

    // Specify json attribute name for the ownerPublicKeyBytes field.
    @JsonProperty("ownerPublicKey")
    private final byte[] ownerPublicKeyBytes;

    // Specify json attribute name for the buyerPublicKeyBytes field.
    @JsonProperty("buyerPublicKey")
    private final byte[] buyerPublicKeyBytes;

    public SellOrderProposition(byte[] ownerPublicKeyBytes, byte[] ↵
    ↪buyerPublicKeyBytes) {
        if(ownerPublicKeyBytes.length != KEY_LENGTH)
            throw new IllegalArgumentException(String.format("Incorrect ↵
    ↪ownerPublicKeyBytes length, %d expected, %d found", KEY_LENGTH, ↵
    ↪ownerPublicKeyBytes.length));

        if(buyerPublicKeyBytes.length != KEY_LENGTH)
            throw new IllegalArgumentException(String.format("Incorrect ↵
    ↪buyerPublicKeyBytes length, %d expected, %d found", KEY_LENGTH, ↵
    ↪buyerPublicKeyBytes.length));

        this.ownerPublicKeyBytes = Arrays.copyOf(ownerPublicKeyBytes, KEY_
    ↪LENGTH);
    }

```

(continues on next page)

(continued from previous page)

```

        this.buyerPublicKeyBytes = Arrays.copyOf(buyerPublicKeyBytes, KEY_
↪LENGTH);
    }

    @Override
    public byte[] pubKeyBytes() {
        return Arrays.copyOf(ownerPublicKeyBytes, KEY_LENGTH);
    }

    public byte[] getOwnerPublicKeyBytes() {
        return pubKeyBytes();
    }

    public byte[] getBuyerPublicKeyBytes() {
        return Arrays.copyOf(buyerPublicKeyBytes, KEY_LENGTH);
    }

    @Override
    public PropositionSerializer serializer() {
        return SellOrderPropositionSerializer.getSerializer();
    }

    @Override
    public int hashCode() {
        int result = Arrays.hashCode(ownerPublicKeyBytes);
        result = 31 * result + Arrays.hashCode(buyerPublicKeyBytes);
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (obj == null)
            return false;
        if (!(obj instanceof SellOrderProposition))
            return false;
        if (obj == this)
            return true;
        SellOrderProposition that = (SellOrderProposition) obj;
        return Arrays.equals(ownerPublicKeyBytes, that.ownerPublicKeyBytes)
            && Arrays.equals(buyerPublicKeyBytes, that.
↪buyerPublicKeyBytes);
    }
}

```

As you can see from the code above, a custom proposition can have a number of private fields; in our case the *ownerPublicKeyBytes* and *buyerPublicKeyBytes* properties, which also have *getOwnerPublicKeyBytes()* and *getBuyerPublicKeyBytes()* as getter methods.

A custom proposition must:

- **implement the `ProofOfKnowledgeProposition` interface**, and define its “pubKeyBytes” method, that returns a byte representation of the public key of this proposition:

```

@Override
public byte[] pubKeyBytes() {
    return Arrays.copyOf(ownerPublicKeyBytes, KEY_LENGTH);
}

```

(continues on next page)

(continued from previous page)

}

- provide the usual method and class for serialization and deserialization:

- serializer()
- implement SellOrderPropositionSerializer:

```
public final class SellOrderPropositionSerializer implements
↳PropositionSerializer<SellOrderProposition> {

    private static final SellOrderPropositionSerializer serializer = new
↳SellOrderPropositionSerializer();

    private SellOrderPropositionSerializer() {
        super();
    }

    public static SellOrderPropositionSerializer getSerializer() {
        return serializer;
    }

    @Override
    public void serialize(SellOrderProposition proposition, Writer writer) {
        writer.putBytes(proposition.getOwnerPublicKeyBytes());
        writer.putBytes(proposition.getBuyerPublicKeyBytes());
    }

    @Override
    public SellOrderProposition parse(Reader reader) {
        byte[] ownerPublicKeyBytes = reader.getBytes(SellOrderProposition.KEY_
↳LENGTH);
        byte[] buyerPublicKeyBytes = reader.getBytes(SellOrderProposition.KEY_
↳LENGTH);

        return new SellOrderProposition(ownerPublicKeyBytes,
↳buyerPublicKeyBytes);
    }
}
```

- implement the hashCode() and equals() methods, used to compare the proposition with other ones:

```
@Override
public int hashCode() {
    int result = Arrays.hashCode(ownerPublicKeyBytes);
    result = 31 * result + Arrays.hashCode(buyerPublicKeyBytes);
    return result;
}

@Override
public boolean equals(Object obj) {
    if (obj == null)
        return false;
    if (!(obj instanceof SellOrderProposition))
        return false;
    if (obj == this)
        return true;
}
```

(continues on next page)

(continued from previous page)

```

        SellOrderProposition that = (SellOrderProposition) obj;
        return Arrays.equals(ownerPublicKeyBytes, that.ownerPublicKeyBytes)
            && Arrays.equals(buyerPublicKeyBytes, that.buyerPublicKeyBytes);
    }

```

Now we can analyse the corresponding proof class, `SellOrderSpendingProof`:

```

public final class SellOrderSpendingProof extends AbstractSignature25519
    <PrivateKey25519, SellOrderProposition> {
    // To distinguish who opened the CarSellOrderBox: seller or buyer
    private final boolean isSeller;
    private final byte[] signatureBytes;

    public static final int SIGNATURE_LENGTH = Ed25519.
    <signatureLength();

    public SellOrderSpendingProof(byte[] signatureBytes, boolean isSeller) {
        <super(signatureBytes);
        if (signatureBytes.length != SIGNATURE_LENGTH)
            throw new IllegalArgumentException(String.format(
            <"Incorrect signature length, %d expected, %d found", SIGNATURE_
            <LENGTH,
                signatureBytes.length));
        this.isSeller = isSeller;
        this.signatureBytes = signatureBytes;
    }

    public boolean isSeller() {
        return isSeller;
    }

    public byte[] signatureBytes() {
        return Arrays.copyOf(signatureBytes, SIGNATURE_LENGTH);
    }

    // Depends on isSeller flag value check the signature against 
    <seller or buyer public key specified in SellOrderProposition.
    @Override
    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());
        }
    }

    @Override
    public byte[] bytes() {
        return Bytes.concat(
            new byte[] { (isSeller ? (byte)1 : (byte)0) },
            signatureBytes

```

(continues on next page)

(continued from previous page)

```

    );
}

public static SellOrderSpendingProof parseBytes(byte[] bytes) {
    int offset = 0;

    boolean isSeller = bytes[offset] != 0;
    offset += 1;

    byte[] signatureBytes = Arrays.copyOfRange(bytes, offset,
    ↪offset + SIGNATURE_LENGTH);

    return new SellOrderSpendingProof(signatureBytes, isSeller);
}

@Override
public ProofSerializer serializer() {
    return SellOrderSpendingProofSerializer.getSerializer();
}

@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    SellOrderSpendingProof that = (SellOrderSpendingProof) o;
    return Arrays.equals(signatureBytes, that.signatureBytes) &&
    ↪isSeller == that.isSeller;
}

@Override
public int hashCode() {
    int result = Objects.hash(signatureBytes.length);
    result = 31 * result + Arrays.hashCode(signatureBytes);
    result = 31 * result + (isSeller ? 1 : 0);
    return result;
}
}

```

The most important method here is *isValid*: it receives a proposition and a `byte[]` message, and checks that the signature contained in this proof is valid against them. The signature was passed in the constructor. If this method returns true, any box locked with the proposition can be opened with this proof.

```

@Override
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()
        );
    }
}

```

(continues on next page)

(continued from previous page)

```
    );  
}  
}
```

You should be familiar with all the other methods. *proofTypeId* returns a unique identifier of this proof type:

```
@Override  
public byte proofTypeId() {  
    return CarRegistryProofsIdsEnum.SellOrderSpendingProofId.id();  
}
```

Then we have the methods that compare the proof with other ones:

```
@Override  
public boolean equals(Object o) {  
    if (this == o) return true;  
    if (o == null || getClass() != o.getClass()) return false;  
    SellOrderSpendingProof that = (SellOrderSpendingProof) o;  
    return Arrays.equals(signatureBytes, that.signatureBytes) &&  
        ↪ isSeller == that.isSeller;  
}  
  
@Override  
public int hashCode() {  
    int result = Objects.hash(signatureBytes.length);  
    result = 31 * result + Arrays.hashCode(signatureBytes);  
    result = 31 * result + (isSeller ? 1 : 0);  
    return result;  
}
```

and the methods to serialize and deserialize it;

```
@Override  
public ProofSerializer serializer() {  
    return SellOrderSpendingProofSerializer.getSerializer();  
}
```

```
@Override  
public void serialize(SellOrderSpendingProof boxData, Writer writer)  
    ↪ {  
    writer.put(boxData.isSeller() ? (byte)1 : (byte)0);  
    writer.putBytes(boxData.signatureBytes());  
}  
  
@Override  
public SellOrderSpendingProof parse(Reader reader) {  
    boolean isSeller = reader.getBytes() != 0;  
    byte[] signatureBytes = reader.getBytes(SellOrderSpendingProof.  
    ↪ SIGNATURE_LENGTH);  
  
    return new SellOrderSpendingProof(signatureBytes, isSeller);  
}
```

Please note: the relationship between proposition, proofs and boxes is already defined by the generics used when declaring them. For example, the *SellOrderProposition* (first row below) is also part of the declaration of the related proof and custom box (*CarSellOrderBox*) that gets locked by it:

public final class **SellOrderProposition** implements ProofOfKnowledgeProposition<PrivateKey25519>

public final class SellOrderSpendingProof extends AbstractSignature25519<PrivateKey25519, **SellOrderProposition**>

public final class CarSellOrderBox extends AbstractBox<**SellOrderProposition**, CarSellOrderBoxData, CarSellOrderBox>

This way, some design errors can be identified already at compile time.

Application state

By implementing the *io.horizen.utxo.state.ApplicationState* interface with a custom class, developers can:

- define specific rules to validate transactions (before they are accepted in the mempool and later when included in a block)
- define specific rules to validate blocks (before they are appended to the blockchain)
- be notified when a new block is added to the blockchain (“*onApplyChanges*”), receiving all the boxes created and removed by its transactions, or when a block revert happens (“*onRollback*”).

The methods of the interface are the following ones:

```
public interface ApplicationState {

    void validate(SidechainStateReader stateReader, SidechainBlock block)
    ↪throws IllegalArgumentException;

    void validate(SidechainStateReader stateReader, BoxTransaction
    ↪<Proposition, Box<Proposition>> transaction) throws
    ↪IllegalArgumentException;

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

    Try<ApplicationState> onRollback(byte[] blockId);

    boolean checkStoragesVersion(byte[] blockId);

}
```

Please note how the block revert notification is implemented: a `byte[]` representing a version id is passed every time *onApplyChanges* is called. If a rollback happens, the same version id is passed by the *onRollback* method: all versions after that one have to be discarded.

The method *checkStoragesVersion* is called by the SDK in order to check the alignment of SDK and any application custom storages versions (if any) at node restart.

Most methods have a *SidechainStateReader* parameter. It’s a utility class you can use to access the closed boxes of the sidechain, i.e. all the boxes that haven’t been spent yet. Here its interface definition:

```
public interface SidechainStateReader {

    Optional<Box> getClosedBox(byte[] boxId);

}
```

Now let’s see how the application State is used in our Lambo Registry app, starting from the *onApplyChanges* method:

```

@Override
public Try<ApplicationState> onApplyChanges(SidechainStateReader stateReader,
                                             byte[] blockId,
                                             List<Box<Proposition>> newBoxes,
                                             List<byte[]> boxIdsToRemove) {
    //we update the Car info database. The data from it will be used during
    //validation.

    //collect the vin to be added: the ones declared in new boxes
    Set<String> vinToAdd = carInfoDbService.extractVinFromBoxes(newBoxes);
    //collect the vin to be removed: the ones contained in the removed boxes
    //that are not present in the previous list
    Set<String> vinToRemove = new HashSet<>();
    for (byte[] boxId : boxIdsToRemove) {
        stateReader.getClosedBox(boxId).ifPresent(box -> {
            if (box instanceof CarBox){
                String vin = ((CarBox)box).getVin();
                if (!vinToAdd.contains(vin)){
                    vinToRemove.add(vin);
                }
            } else if (box instanceof CarSellOrderBox){
                String vin = ((CarSellOrderBox)box).getVin();
                if (!vinToAdd.contains(vin)){
                    vinToRemove.add(vin);
                }
            }
        });
    }
    carInfoDbService.updateVin(blockId, vinToAdd, vinToRemove);
    return new Success<>(this);
}

```

As you can see this method is used to update a list containing all the VIN (vehicle identification numbers) that appear in our blockchain. To do that, it inspects the two types of boxes that contain a VIN (CarBox and CarSellOrderBox), and adds each VIN to the list if the box has been created, or remove it if the box has been spent. Since this method is called every time a new block is appended to the chain, we can be sure the list is always updated.

The list is then used in the *validate* method. To validate a single transaction, we check that the VIN is not already in the list:

```

@Override
void validate(SidechainStateReader stateReader, BoxTransaction<Proposition,
Box<Proposition>> transaction) throws IllegalArgumentException {
    // we go through all CarDeclarationTransactions and verify that each
    //CarBox represents a unique Car.
    if (CarDeclarationTransaction.class.isInstance(transaction)){
        Set<String> vinList = carInfoDbService.
        extractVinFromBoxes(transaction.newBoxes());
        for (String vin : vinList) {
            if (! carInfoDbService.validateVin(vin, Optional.empty())){
                throw new IllegalArgumentException("Vin has been used before.
");
            }
        }
    }
}

```

To validate an entire block, we need an additional check, to be sure that in the same block two different transactions don't declare the same VIN:

```
@Override
void validate(SidechainStateReader stateReader, SidechainBlock block) throws
↳IllegalArgumentException {
    //We check that there are no multiple transactions declaring the same
↳VIN inside the block
    Set<String> vinList = new HashSet<>();
    for (BoxTransaction<Proposition, Box<Proposition>> t : JavaConverters.
↳seqAsJavaList(block.transactions())){
        if (CarDeclarationTransaction.class.isInstance(t)){
            for (String currentVin : carInfoDbService.extractVinFromBoxes(t.
↳newBoxes())){
                if (vinList.contains(currentVin)){
                    throw new IllegalArgumentException("Vin has been used in
↳another transaction.");
                }else{
                    vinList.add(currentVin);
                }
            }
        }
    }
}
```

The *rollback* method, which is very simple and delegates all the logic to the service used to store our list:

```
@Override
public Try<ApplicationState> onRollback(byte[] blockId) {
    carInfoDbService.rollback(blockId);
    return new Success<>(this);
}
```

Finally, the *checkStoragesVersion* method, which is also very simple and just check the version of *carInfoDbService* storage against the input parameter:

```
@Override
public boolean checkStoragesVersion(byte[] blockId)
{
    byte[] ver = carInfoDbService.lastVersionID().orElse(new
↳ByteArrayWrapper(NULL_VERSION)).data();
    return Arrays.equals(blockId, ver);
}
```

Application wallet

The interface *io.horizen.utxo.wallet.ApplicationWallet* is another extension point that allows an application to be notified each time a secret or box is added or removed from the sidechain node local wallet.

```
public interface ApplicationWallet {
    void onAddSecret(Secret secret);
    void onRemoveSecret(Proposition proposition);
    void onChangeBoxes(byte[] blockId, List<Box<Proposition>> boxesToUpdate,
↳List<byte[]> boxIdsToRemove);
    void onRollback(byte[] blockId);
    boolean checkStoragesVersion(byte[] blockId);
}
```

The Lambo registry example does not implement the interface *ApplicationWallet* because its wallet has basic requirements. You may need to use interface *io.horizen.utxo.wallet.ApplicationWallet* depending on your app requirements. For example, if the app needs to maintain a separate wallet balance or counter of a specific kind of custom boxes associated to locally stored keys, you could put the code that updates those records inside the *onChangeBoxes* method.

Application Stopper

The interface *io.horizen.SidechainAppStopper* allows an application to be called when the node stop procedure is initiated:

```
public interface SidechainAppStopper {
    void stopAll();
}
```

Such a procedure can be explicitly triggered via the API ‘node/stop’ or can be triggered when the JVM is shutting down, for instance when a SIGINT is received. In the Lambo registry implementation of the method ‘void stopAll()’, the *carInfoDbService* storage is closed:

```
@Override
public void stopAll() {
    carInfoDbService.close()
}
```

API extension

An application can extend the standard API endpoints and define custom ones. As an example, the Lambo Registry application adds four endpoints, one for each added transaction:

- createCar
- createCarSellOrder
- acceptCarSellOrder
- cancelCarSellOrder

These new endpoints do not broadcast the transaction directly, but only produce a signed hex version of it; to execute the transaction, the user will later have to post it to the standard endpoint */transaction/sendTransaction*. This approach is just a design choice, so it’s not a mandatory requirement. Before looking at the code, please note that all these endpoints need to interact with the local wallet to unlock boxes and sign the transactions.

So, the first step to add endpoints is to extend the *io.horizen.api.http.ApplicationApiGroup* class, and implement its two methods:

```
@Override
public String basePath() {
    return "carApi";
}

@Override
public List<Route> getRoutes() {
    List<Route> routes = new ArrayList<>();
    routes.add(bindPostRequest("createCar", this::createCar,
    ↪ CreateCarBoxRequest.class));
    routes.add(bindPostRequest("createCarSellOrder",
    ↪ this::createCarSellOrder, CreateCarSellOrderRequest.class));
    routes.add(bindPostRequest("acceptCarSellOrder",
    ↪ this::acceptCarSellOrder, SpendCarSellOrderRequest.class)); (continues on next page)
```

(continued from previous page)

```

    routes.add(bindPostRequest("cancelCarSellOrder",
    ↪this::cancelCarSellOrder, SpendCarSellOrderRequest.class));
    return routes;
}

```

The first method defines the first part of our endpoint urls.

The second method returns the list of the new routes. The SDK uses the [Akka Http Routing library](#), and the type of each array element returned by this method must be an Akka Route. In most cases (including the Lambo registry example) you don't have to know much more about Akka routes, as you can just use the provided *bindPostRequest* method to build a route element. The *bindPostRequest* method returns an Akka route that responds to an HTTP POST request, and receives three parameters:

- a String, representing the request path
- the method implementing the logic
- a class representing the request class

We can see all this in the first endpoint defined in the Lambo registry: "createCar".

This is the class associated to its request (CreateCarBoxRequest - the third parameter):

```

public class CreateCarBoxRequest {
    public String vin;
    public int year;
    public String model;
    public String color;
    public String proposition;
    public long fee;

    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;
    }
}

```

As you can see the class is just a javabean that will map the fields of the input json into the request body. You have to provide the setter of each property, to allow the SDK engine to populate the fields with the request data.

Now let's check out the method implementing the endpoint logic (i.e. the second parameter of the *bindPostRequest* method):

```
private ApiResponse createCar(SidechainNodeView view, CreateCarBoxRequest
    ent) {
    try {
        // Parse the proposition of the Car owner.
        PublicKey25519Proposition carOwnershipProposition =
    PublicKey25519PropositionSerializer.getSerializer()
        .parseBytes(BytesUtils.fromHexString(ent.proposition));

        //check that the vin is unique (both in local veichle store and in
    mempool)
        if (! carInfoDBService.validateVin(ent.vin, Optional.of(view.
    getNodeMemoryPool()))){
            throw new IllegalStateException("Vehicle identification number
    already present in blockchain");
        }

        CarBoxData carBoxData = new CarBoxData(carOwnershipProposition, ent.
    vin, ent.year, ent.model, ent.color);

        // Try to collect regular boxes to pay fee
        List<Box<Proposition>> paymentBoxes = new ArrayList<>();
        long amountToPay = ent.fee;

        // Avoid to add boxes that are already spent in some Transaction
    that is present in node Mempool.
        List<byte[]> boxIdsToExclude = boxesFromMempool(view.
    getNodeMemoryPool());
        List<Box<Proposition>> ZenBoxes = view.getNodeWallet().
    boxesOfType(ZenBox.class, boxIdsToExclude);
        int index = 0;
        while (amountToPay > 0 && index < ZenBoxes.size()) {
            paymentBoxes.add(ZenBoxes.get(index));
            amountToPay -= ZenBoxes.get(index).value();
            index++;
        }

        if (amountToPay > 0) {
            throw new IllegalStateException("Not enough coins to pay the fee.
    ");
        }

        // Set change if exists
        long change = Math.abs(amountToPay);
        List<ZenBoxData> regularOutputs = new ArrayList<>();
        if (change > 0) {
            regularOutputs.add(new ZenBoxData((PublicKey25519Proposition)
    paymentBoxes.get(0).proposition(), change));
        }

        // Create fake proofs to be able to create transaction to be signed.
        List<byte[]> inputIds = new ArrayList<>();
        for (Box b : paymentBoxes) {
            inputIds.add(b.id());
        }
    }
```

(continues on next page)

(continued from previous page)

```

        List fakeProofs = Collections.nCopies(inputIds.size(), null);
        Long timestamp = System.currentTimeMillis();

        CarDeclarationTransaction unsignedTransaction = new
↳CarDeclarationTransaction(
            inputIds,
            fakeProofs,
            regularOutputs,
            carBoxData,
            ent.fee,
            timestamp);

        // Get the Tx message to be signed.
        byte[] messageToSign = unsignedTransaction.messageToSign();

        // Create real signatures.
        List<Signature25519> proofs = new ArrayList<>();
        for (Box<Proposition> box : paymentBoxes) {
            proofs.add((Signature25519) view.getNodeWallet().
↳secretByPublicKey(box.proposition()).get().sign(messageToSign));
        }

        // Create the transaction with real proofs.
        CarDeclarationTransaction signedTransaction = new
↳CarDeclarationTransaction(
            inputIds,
            proofs,
            regularOutputs,
            carBoxData,
            ent.fee,
            timestamp);

        return new TxResponse(ByteUtils.
↳toHexString(sidechainTransactionsCompanion.toBytes((BoxTransaction)
↳signedTransaction)));
    }
    catch (Exception e) {
        return new CarResponseError("0102", "Error during Car declaration.",
↳Some.apply(e));
    }
}

```

Please note that:

- the method receives two parameters: the first one is *SidechainNodeView*, an utility class that gives access to a snapshot of the current blockchain state and the current wallet. It can be used, for example, to find a closed box owned by the user, that is a box that can be spent in the transaction. The second parameter is the “request class” previously introduced.
- the method must return a class implementing the *ApiResponse* interface, or its sub-interface *SuccessResponse* if the method executes without errors. It can be any javabeen, but it must include the *@JsonView* annotation, to instruct the SDK engine to serialize it to json, and must expose the data to be returned in public fields. The response class in the Lambo registry example has only one field (*transactionBytes*), which is a String containing the HEX representation of the created transaction:

```

@JsonView(Views.Default.class)
static class TxResponse implements SuccessResponse {

```

(continues on next page)

(continued from previous page)

```

    public String transactionBytes;

    public TxResponse(String transactionBytes) {
        this.transactionBytes = transactionBytes;
    }
}

```

If we now look into the method logic, we can see that, at first, it parses the input data and constructs the objects from it (carOwnershipProposition and carBoxData). It also performs a security check that returns an error if the user tries to declare a car with a Vehicle Identification Number which already exists:

```

// Parse the proposition of the Car owner.
PublicKey25519Proposition carOwnershipProposition =
    ↪PublicKey25519PropositionSerializer.getSerializer()
    .parseBytes(BytesUtils.fromHexString(ent.proposition));

//check that the vin is unique (both in local veichle store and in mempool)
if (! carInfoDBService.validateVin(ent.vin, Optional.of(view.
    ↪getNodeMemoryPool()))){
    throw new IllegalStateException("Vehicle identification number already
    ↪present in blockchain");
}

CarBoxData carBoxData = new CarBoxData(carOwnershipProposition, ent.vin, ent.
    ↪year, ent.model, ent.color);

```

One more note about the Vehicle Identification Number check: a similar check is also performed in the applicationState as part of the consensus validation, to discard invalid transactions. As a general design rule, all checks on data correctness must be performed in both points. This way, transactions are verified by the endpoint. The endpoint will only allow valid transactions on the network. If a user tries to bypass the creation endpoint by broadcasting the binary transaction hex directly, the consensus check will not accept invalid transactions.

After this check, the code builds two lists: *paymentBoxes*, a list of coins used to pay the fee, and *regularOutputs*, the output boxes. We start this second list with the change (if any) of the fee payment.

```

// Try to collect regular boxes to pay fee
List<Box<Proposition>> paymentBoxes = new ArrayList<>();
long amountToPay = ent.fee;

// Avoid to add boxes that are already spent by transactions in the node
    ↪Mempool.
List<byte[]> boxIdsToExclude = boxesFromMempool(view.getNodeMemoryPool());
List<Box<Proposition>> ZenBoxes = view.getNodeWallet().boxesOfType(ZenBox.
    ↪class, boxIdsToExclude);
int index = 0;
while (amountToPay > 0 && index < ZenBoxes.size()) {
    paymentBoxes.add(ZenBoxes.get(index));
    amountToPay -= ZenBoxes.get(index).value();
    index++;
}

if (amountToPay > 0) {
    throw new IllegalStateException("Not enough coins to pay the fee.");
}

// Set change if exists

```

(continues on next page)

(continued from previous page)

```

long change = Math.abs(amountToPay);
List<ZenBoxData> regularOutputs = new ArrayList<>();
if (change > 0) {
    regularOutputs.add(new ZenBoxData((PublicKey25519Proposition)
        paymentBoxes.get(0).proposition(), change));
}

```

Now everything is ready to build and sign the transaction. To generate signature proofs, we need the transaction bytes. But to obtain the transaction bytes, we need to create it with the needed proofs. To cut this dependency loop, transactions are built in the following way:

1. Create fake/empty proofs,
2. Create transaction by using those dummy proofs
3. Receive Tx message to be signed from transaction at step 2 (we can do it because proofs are not part of the message that needs to be signed)
4. Create real proof by using Tx message to be signed
5. Create the real transaction with real proofs

In the code:

```

// Create fake proofs to be able to create transaction to be
↳signed.
List<byte[]> inputIds = new ArrayList<>();
for (Box b : paymentBoxes) {
    inputIds.add(b.id());
}

List fakeProofs = Collections.nCopies(inputIds.size(), null);
Long timestamp = System.currentTimeMillis();

CarDeclarationTransaction unsignedTransaction = new
↳CarDeclarationTransaction(
    inputIds,
    fakeProofs,
    regularOutputs,
    carBoxData,
    ent.fee,
    timestamp);

// Get the Tx message to be signed.
byte[] messageToSign = unsignedTransaction.messageToSign();

// Create real signatures.
List<Signature25519> proofs = new ArrayList<>();
for (Box<Proposition> box : paymentBoxes) {
    proofs.add((Signature25519) view.getNodeWallet()
        .secretByPublicKey(box.proposition())
        .get()
        .sign(messageToSign));
}

// Create the transaction with real proofs.
CarDeclarationTransaction signedTransaction = new
↳CarDeclarationTransaction(

```

(continues on next page)

(continued from previous page)

```
inputIds,  
proofs,  
regularOutputs,  
carBoxData,  
ent.fee,  
timestamp);
```

Finally, the response construction:

```
return new TxResponse(  
    ByteUtils.toHexString(sidechainTransactionsCompanion.  
        ↳toBytes((BoxTransaction) signedTransaction))  
);
```

As a result, this endpoint will be exposed by this url: `/carApi/createCar` and will be invoked with a post http request. Input and output data will be represented in json format.

The structure of the others endpoints is similar, it's a good exercise to check them out and see how they were implemented.

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`

Returns the block with the specified ID, together with its height in the blockchain

Parameters

Name	Type	Required	Description
blockId	String	yes	Block ID

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:

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 block ids

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":[
      "055c15d9a6c9ae299493d241705a2bcfdabc72a19f04394a26aa53b39f6ee2a6",
      "ae6bcf104b7a7cccf83dfa23494760fb8d9a4d5cc3de82443de8b82bb86669d1",
      "9120b0f8518d1944d4b0e8fac8990acc7dcb792ea660414906a03f346407160c",
      "e5b0e97df9502c9510e4862041754b62931c9dc0a4fa873b3a0d75561dcbe712",
      "6a080e3ee665980bf647b450749b04177fe272537808bb4aec70417f9994bd04",
      "97d1956ecb1199fe03171b0923dff4031850e33db56dd1afc3b5384350315d80",
      "2c3a4a91989110218a827f8baefa3a8e5baf33e7e16d32b2bdace94553478dde",
      "cf82fba3e75ac89ca7e8d1c29458b2d5eff9d807407d3265c14251da2c70b3b1",
      "d61da61b2c877f717fa50563a42cbad4420486bfa3b1f05d888528d69d8258d8",
      "921f9406d8edd03d2f5b65aa6f89e452720c7ef07244ee06f3ad19d2c49e45d8"
    ]
  }
}
```

POST /block/findIdByHeight

Returns a sidechain block Id by its height in the blockchain

Parameters

Name	Type	Required	Description
height	int	yes	Retrieves block ID by its 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*Returns 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":
        ↪ "ed59dd9a4200a09783fe1dc9f095e7d41cbbc3cbc2c5ffb3363a150b242d7b72",
        "timestamp": 1644419532,
        "forgingStakeInfo": {
          "blockSignPublicKey": {
            "publicKey":
            ↪ "a5b10622d70f094b7276e04608d97c7c699c8700164f78e16fe5e8082f4bb2ac"
          },
          "vrfPublicKey": {
            "publicKey":
            ↪ "076a9191a89fee51439600b0455db357a9899694d1cdad6a3c71bf65e6cce53280"
          },
          "stakeAmount": 60000000000
        },
        "forgingStakeMerklePath": "00",
        "vrfProof": {
          "vrfProof":
          ↪ "47d0f05a139f375e238c38d0440628cbd20640ebd4739bba1bd391c6ab94033100954ea33aeb55608ff17636905d8"
          ↪ ""
        },
        "sidechainTransactionsMerkleRootHash":
        ↪ "0000000000000000000000000000000000000000000000000000000000000000",
        "mainchainMerkleRootHash":
        ↪ "0000000000000000000000000000000000000000000000000000000000000000",
        "ommersMerkleRootHash":
        ↪ "0000000000000000000000000000000000000000000000000000000000000000",
        "ommersCumulativeScore": 0,
        "feePaymentsHash":
        ↪ "0000000000000000000000000000000000000000000000000000000000000000",
        "signature": {
          "signature":
          ↪ "589128b800466a912e1571ac2ab2e952dc5d589046abad051cd5d41fce6eee6ba868d1be259691dfcafd0b440d6d8"
          ↪ ""
        },
        "id":
        ↪ "d7f2763c95381c87fb01b6b33da46539f4ef3853e7661b6da4ae5ed26c6e59cb"
      },
      "sidechainTransactions": [],
      "mainchainBlockReferencesData": [],

```

(continues on next page)

(continued from previous page)

```

    "mainchainHeaders": [],
    "ommers": [],
    "timestamp": 1644419532,
    "parentId":
    ↪ "ed59dd9a4200a09783fe1dc9f095e7d41cbbc3cbc2c5fffb3363a150b242d7b72",
      "id": "d7f2763c95381c87fb01b6b33da46539f4ef3853e7661b6da4ae5ed26c6e59cb
    ↪ "
    },
    "height": 4
  }
}

```

POST /block/getFeePayments

Returns the list of ZenBoxes that represents the forgers fee payments paid after the given block was applied.

Parameters

Name	Type	Required	Description
blockId	String	yes	Block ID

Example request:

Bash

```
curl -X POST "http://127.0.0.1:9086/block/getFeePayments" -H "accept: application/json" -H "Content-Type: application/json" -d '{"blockId":"0...6"}'
```

Example response:

```

{
  "result" : {
    "feePayments" : [ {
      "nonce" : -9087003896463582454,
      "id" :
    ↪ "7fe62e862d531d6598c57905754f17875a68cc7848723fa3c42fcc483e7f5a0e",
      "isCustom" : false,
      "value" : 941,
      "typeName" : "ZenBox",
      "proposition" : {
        "publicKey" :
    ↪ "a5b10622d70f094b7276e04608d97c7c699c8700164f78e16fe5e8082f4bb2ac"
      }
    }, {
      "nonce" : 1359254115016647210,
      "id" :
    ↪ "e23aea7695d1956530771b3f6446790c688c59e77fc41f57730888d5f04c3508",
      "isCustom" : false,
      "value" : 260,
      "typeName" : "ZenBox",
      "proposition" : {
        "publicKey" :
    ↪ "05e47de1dd136b1d92a65758db781e1145677865a8bb3412dcac3ab65e8d071c"
      }
    } ]
  }
}

```


POST /block/findBlockInfoById

Returns *SidechainBlockInfo* for a single block and if it is in the active chain.

Parameters

Name	Type	Required	Description
blockId	String	yes	Block ID

Example request:

Bash

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

Example response:

```
{
  "result": {
    "blockInfo": {
      "height": 4,
      "score": 4,
      "parentId": "ed59dd9a4200a09783fe1dc9f095e7d41cbbc3cbc2c5ffb3363a150b242d7b72",
      "timestamp": 1644419532,
      "semanticValidity": "Valid",
      "mainchainHeaderBaseInfo": [],
      "mainchainReferenceDataHeaderHashes": [],
      "withdrawalEpochInfo": {
        "epoch": 0,
        "lastEpochIndex": 1
      },
      "vrfOutputOpt": {
        "bytes": "41b2c57d834fa5a479871022c7af3992c80ddbe5c205efcc1e56219bc2cc8c33"
      },
      "lastBlockInPreviousConsensusEpoch":
      ↪ "650fe8657567b3b7779d30858177b1ba24b7bef6be250b443fca4e1bb9293a"
    },
    "isInActiveChain": true
  }
}
```

POST /block/startForging

Starts forging

This endpoint needs authentication (See [API authentication](#))

No Parameters**Example request:**

Bash

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

Example response:

```
{
  "result": {}
}
```

POST /block/stopForging

Stops forging

This endpoint needs authentication (See [API authentication](#))

No Parameters**Example request:**

Bash

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

Example response:

```
{
  "result": {}
}
```

POST /block/generate

Tries 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

Returns 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" : 5,
    "bestSlotNumber" : 4,
    "forgingEnabled" : false
  }
}
```

Sidechain Transaction operations

POST /transaction/allTransactions

Returns all transactions in the memory pool.

Parameters

Name	Type	Re-quired	Description
format	boolean	no	Returns an array of transaction ids if format=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" : [ {
      "modifierTypeId" : 2,
      "id" :
      ↪ "c93924cbd905be02f4e4ed03aded33e8d2be77482e9566c32a8785238720ea80",
      "newBoxes" : [ {
        "nonce" : -7274338835981510232,
        "id" :
        ↪ "0226207ce3de087a519ed0722eb20d16833c8bddf425b649d8de794aed3c7d9b",
        "isCustom" : false,
        "value" : 2000000000,
        "typeName" : "ZenBox",

```

(continues on next page)

(continued from previous page)

```

        "proposition" : {
          "publicKey" :
↪ "51c2d4c69602f30c8935551a076dc478eb196531996bb4dde4e345e115d3771a"
        }
      } ],
      "fee" : 0,
      "version" : 1,
      "unlockers" : [ {
        "boxKey" : {
          "signature" :
↪ "3edf5dc31fb487d9a47f2098f873d6d6a4276acc42190edf3e308a5ba0335912a8b223fdd84bcbfacb9f66e0b77f0"
↪ "
        },
        "closedBoxId" :
↪ "a2a1f8ef8ed1d4056a32e8a588574f5d8bfa233fdd4060b231a2b8b69a5ac17c"
      } ] ],
      "isCustom" : false,
      "typeName" : "SidechainCoreTransaction"
    } ]
  }
}

```

POST /transaction/findById

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

Parameters

Name	Type	Description
transactionId	String	Finds by Transaction Id
blockHash	String	Searches in block referenced by blockHash (does not care about txIndex parameter)
transactionIndex	boolean	txIndex = true -> Searches in memory pool, if not found, searches in the whole blockchain
format	boolean	If format = true, returns a JSONObject, otherwise returns the transaction as byte serialization

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":true}'
```

Example response:

```

{
  "result" : {
    "transaction" : {
      "modifierTypeId" : 2,
      "id" :
↪ "a8addale352f9aefa0b157bba3578c46390b13b55dcac6fa59d87d0feeb15025",
      "newBoxes" : [ {

```

(continues on next page)

(continued from previous page)

```
        "nonce" : -633504681414039135,
        "id" :
    ↪ "7238937c1e947a4f9d02a76bcb551dc4d366b71b3bda83947ae90136d8dd8ceb",
        "isCustom" : false,
        "value" : 1500000000,
        "typeName" : "ZenBox",
        "proposition" : {
            "publicKey" :
    ↪ "76e1a22f78459ff60b6e7bba4067b05f341b1206792c1f1029e38906d809884b"
        }
    },
    "fee" : 0,
    "version" : 1,
    "unlockers" : [ {
        "boxKey" : {
            "signature" :
    ↪ "82c634e922e2c681cd1dbb5c4cd367ce48acf3037d707cd885ad3cc25ab15dbe84c2ff88b719a164e790019e9839c
    ↪ "
        },
        "closedBoxId" :
    ↪ "4cf6109023f18ba1364be15071b5d62983247e37d52fa1489f158039bbde7772"
    } ],
    "isCustom" : false,
    "typeName" : "SidechainCoreTransaction"
}
}
```

POST /transaction/decodeTransactionBytes

Returns 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" : {
      "modifierTypeId" : 2,
      "id" :
    ↪ "a8addale352f9aefa0b157bba3578c46390b13b55dcac6fa59d87d0feeb15025",
      "newBoxes" : [ {
        "nonce" : -633504681414039135,
        "id" :
    ↪ "7238937c1e947a4f9d02a76bcb551dc4d366b71b3bda83947ae90136d8dd8ceb",
    ↪ "7238937c1e947a4f9d02a76bcb551dc4d366b71b3bda83947ae90136d8dd8ceb",
      } ]
    }
  }
}
```

(continues on next page)

(continued from previous page)

```
        "isCustom" : false,
        "value" : 1500000000,
        "typeName" : "ZenBox",
        "proposition" : {
            "publicKey" :
↪ "76e1a22f78459ff60b6e7bba4067b05f341b1206792c1f1029e38906d809884b"
        }
    } ],
    "fee" : 0,
    "version" : 1,
    "unlockers" : [ {
        "boxKey" : {
            "signature" :
↪ "82c634e922e2c681cd1dbb5c4cd367ce48acf3037d707cd885ad3cc25ab15dbe84c2ff88b719a164e790019e9839c
↪ "
        },
        "closedBoxId" :
↪ "4cf6109023f18ba1364be15071b5d62983247e37d52fa1489f158039bbde7772"
    } ],
    "isCustom" : false,
    "typeName" : "SidechainCoreTransaction"
}
}
```

POST /transaction/createCoreTransaction

Creates and signs a Sidechain core transaction, specifying inputs and outputs. Returns the new transaction as a hex string if `format = false`, otherwise its JSON representation.

This endpoint needs authentication (See [API authentication](#))

Parameters**Example Value**

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

(continues on next page)

(continued from previous page)

```

    "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"}]}'

```

Example response:

```

{
  "result" : {
    "transactionBytes" :
    ↪ "0101b8a6d6b9070282750e3a53818ece2c116f6978401115c394dbe1ffdc184fb9ab9a8aca968927020194ed9aa99
    ↪ "
  }
}

```

POST /transaction/createCoreTransactionSimplified

Creates and signs a Sidechain core transaction, specifying inputs and outputs. Returns the new transaction as a hex string if format = false, otherwise its JSON representation.

This endpoint needs authentication (See [API authentication](#))

Parameters

Example Value

```

{
  "regularOutputs": [
    {
      "publicKey": "string",
      "value": 0
    }
  ],
  "withdrawalRequests": [
    {
      "publicKey": "string",
      "value": 0
    }
  ],
  "forgerOutputs": [
    {
      "publicKey": "string",
      "blockSignPublicKey": "string",
      "vrfPubKey": "string",
      "value": 0
    }
  ]
}

```

(continues on next page)

(continued from previous page)

```

    }
  ],
  "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":"","value":0}], "withdrawalRequests":[{"publicKey":"","value":0}], "forgerOutputs":[{"pub

```

Example response:

```

{
  "result" : {
    "transaction" : {
      "modifierTypeId" : 2,
      "id" :
      ↪ "b8a997743d5f4a7f7c43141fa5c156494ed7ddd2b52a26274a3eea0fe76aa6a6",
      "newBoxes" : [ {
        "nonce" : 1797084183504923750,
        "id" :
        ↪ "f1d9b9a010e069885a3b8235b80b2249da53ce8a45dab1169d0d27911f5dc3ee",
        "isCustom" : false,
        "value" : 10,
        "typeName" : "ZenBox",
        "proposition" : {
          "publicKey" :
          ↪ "94ed9aa9928393fde71280023e1afcc2578cfbcb68bff90b6d2785dbbd6c7beb"
        }
      }, {
        "nonce" : -1293113557566329474,
        "id" :
        ↪ "3a9383a555ad292f2cad30ce8d2c194b97fa0a35f9e8c5baa267b8c093a5cb58",
        "isCustom" : false,
        "value" : 999999989,
        "typeName" : "ZenBox",
        "proposition" : {
          "publicKey" :
          ↪ "a5b10622d70f094b7276e04608d97c7c699c8700164f78e16fe5e8082f4bb2ac"
        }
      } ],
      "fee" : 1,
      "version" : 1,
      "unlockers" : [ {
        "boxKey" : {
          "signature" :
          ↪ "508a896275014539c77705bcf276eaf613674d11c5d2211bd9ba9e2fb85f653e6efe49d0554297667932d312aed12
          ↪ "
        },
        "closedBoxId" :
        ↪ "82750e3a53818ece2c116f6978401115c394dbe1ffdc184fb9ab9a8aca968927"
      } ],
      "isCustom" : false,

```

(continues on next page)

(continued from previous page)

```

    "typeName" : "SidechainCoreTransaction"
  }
}

```

POST /transaction/sendCoinsToAddress

Creates and signs a regular transaction, specifying outputs and fee. Then validates and sends the transaction. Returns the id of the transaction

This endpoint needs authentication (See [API authentication](#))

Parameters

Example Value

```

{
  "outputs": [
    {
      "publicKey": "string",
      "value": 0
    }
  ],
  "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" :
    → "bc4cc8d2469f49f89d69f5b77f7a890e40ce3ac0e971bcabd6db6a78131fa2b5"
  }
}

```

POST /transaction/withdrawCoins

Creates and signs a regular transaction, specifying withdrawal outputs and fee. Then validates and sends the transaction. Returns the id of the transaction.

This endpoint needs authentication (See [API authentication](#))

Parameters

```

{
  "outputs": [
    {
      "mainchainAddress": "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/withdrawCoins" -H "accept: application/json" -H "Content-Type: application/json" -d '{"outputs":[{"mainchainAddress":"string","value":0}], "fee":0}'
```

Example response:

```
{
  "result" : {
    "transactionId" :
    ↪ "ccaa312d3eded27469a8241ccc885b19361687caded0bdf0511a20310ef46310"
  }
}
```

POST /transaction/makeForgerStake

Creates and signs a Sidechain core transaction, specifying forger stake outputs and fee. Then validates and sends the transaction. Returns the id of the transaction

This endpoint needs authentication (See [API authentication](#))

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" :
    ↪ "ccaa312d3eded27469a8241ccc885b19361687caded0bdf0511a20310ef46310"
  }
}
```

POST /transaction/spendForgingStake

Creates and signs sidechain core transaction, specifying inputs and outputs. Returns the new transaction as a hex string if format = false, otherwise its JSON representation.

This endpoint needs authentication (See [API authentication](#))

Parameters

Example Value

```
{
  "transactionInputs": [
    {
      "boxId": "string"
    }
  ],
  "regularOutputs": [
    {
      "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/spendForgingStake" -H "accept: application/json" -H "Content-Type: application/json" -d '{"transactionInputs":[{"boxId":"string"}],"regularOutputs":[{"publicKey":"string","value":0}],"forgerOutputs":[{"publicKey":"string","blockSignPublicKey":"string","vrfPubKey":"string","value":0}]' --data-binary ''
```

Example response:

```
{
  "result" : {
    "transaction" : {
      "modifierTypeId" : 2,
      "id" :
      ↪ "081b668f9e2e63c81be89a5e0ca4a7c9166cefde59b026072e3a89704919767b",
      "newBoxes" : [ {
        "nonce" : -219144346324825135,
        "id" :
        ↪ "431d60e42c2503fdfbe9d6d530d04a75c051de32905a68d88fc86830f3d13aae",
        "isCustom" : false,
        "value" : 10,
        "typeName" : "ZenBox",
        "proposition" : {
```

(continues on next page)

(continued from previous page)

```
        "publicKey" :
        ↪ "94ed9aa9928393fde71280023e1afcc2578cfbcb68bff90b6d2785dbbd6c7beb"
        },
        "fee" : 1,
        "version" : 1,
        "unlockers" : [ {
            "boxKey" : {
                "signature" :
                ↪ "66c56f5a9dacfc0e4df5dbb1b4bb95fbd66b3d6e3b626d9f83f72587410495e90dfe8f310a57e8c3bd89c315a54f9
                ↪ "
            },
            "closedBoxId" :
            ↪ "1076fded2f91d1231247764e05ecb44c605012dbbcac95e9ce0aced3619484d3"
        } ],
        "isCustom" : false,
        "typeName" : "SidechainCoreTransaction"
    }
}
```

POST /transaction/sendTransaction

Validates and sends a transaction, given its serialization as input. Then returns the id of the transaction.

This endpoint needs authentication (See [API authentication](#))

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" :
    ↪ "ccaa312d3eded27469a8241ccc885b19361687cadef0bdf0511a20310ef46310"
  }
}
```

POST /transaction/createKeyRotationTransaction

Creates and signs sidechain transaction for signers or masters certificate submitter key rotation.

Parameters

Name	Type	Description
keyType	int	Key type - 0 for signers key, 1 for masters key. Min = 0. Max = 1
keyIndex	int	Index of certificate submitter key
newKey	string	Value of new key
signingKeySignature	string	Signing key signature
masterKeySignature	string	Master key signature
newKeySignature	string	New key signature (if key type 0, then new signers key signature; if key type 1, then master key signature). Min = 0. Max = 1.
format	boolean	Optional field - true if format, false if non format
automatic-Send	boolean	Optional field - true if automatic, false if not automatic
fee	int	Optional field for transaction fee

Example request:

Bash

```
curl -X POST "http://127.0.0.1:9085/transaction/createKeyRotationTransaction" -H "accept: application/json" -d '{"keyType": 0, "keyIndex": 3, "newKey": "string", "signingKeySignature": "string", "masterKeySignature": "string", "newKeySignature": "string"}'
```

Example response:

```
{
  "result": {
    "transactionId":
    → "3c25254df2f57a524c65b5883550bb1a41130493c6440c60eb6256f4c142dbc9"
  }
}
```

Sidechain Wallet Operations**POST /wallet/allBoxes**

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

This endpoint needs authentication (See [API authentication](#))

Parameters

Name	Type	Required	Description
boxTypeClass	String	No	Type of boxes. If not specified, returns all existing boxes
excludeBoxIds	String Array	No	ID of boxes to exclude

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": [
      {
        "nonce": -673297840433871900,
        "id":
↪ "1076fded2f91d1231247764e05ecb44c605012dbbcac95e9ce0aced3619484d3",
        "vrfPubKey": {
          "publicKey":
↪ "53db9055f6eff032310ca618c2bf8edea98927c24a472ff69d0eb0fed6285e1c00"
        },
        "blockSignProposition": {
          "publicKey":
↪ "a5b10622d70f094b7276e04608d97c7c699c8700164f78e16fe5e8082f4bb2ac"
        },
        "isCustom": false,
        "value": 100000000000,
        "typeName": "ForgerBox",
        "proposition": {
          "publicKey":
↪ "a5b10622d70f094b7276e04608d97c7c699c8700164f78e16fe5e8082f4bb2ac"
        }
      },
      {
        "nonce": -3970212005742197000,
        "id":
↪ "82750e3a53818ece2c116f6978401115c394dbe1ffdc184fb9ab9a8aca968927",
        "isCustom": false,
        "value": 100000000000,
        "typeName": "ZenBox",
        "proposition": {
          "publicKey":
↪ "47286ba429e486767d35e79702206d1181894487f8d74550cbleec3b0bd9b5f3"
        }
      },
      {
        "nonce": -8654764026769776000,
        "id":
↪ "e40ab88ee303e914de929971b81ded0fdec66f9aed48736658fe2b440014d867",
        "isCustom": false,
        "value": 200000000000,
        "typeName": "ZenBox",
        "proposition": {
          "publicKey":
↪ "51c2d4c69602f30c8935551a076dc478eb196531996bb4dde4e345e115d3771a"
        }
      },
      {
        "nonce": -633504681414039200,
        "id":
↪ "7238937c1e947a4f9d02a76bcb551dc4d366b71b3bda83947ae90136d8dd8ceb",
        "isCustom": false,
        "value": 150000000000,
        "typeName": "ZenBox",
        "proposition": {
          "publicKey":
↪ "76e1a22f78459ff60b6e7bba4067b05f341b1206792e1f1029e38906d809884b"
        }
      }
    ]
  }
}

```

(continues on next page)

(continued from previous page)

```

    }
  }
]
}

```

POST /wallet/coinsBalance*Returns the global balance for all types of boxes***This endpoint needs authentication** (See [API authentication](#))**No Parameters****Example request:**

Bash

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

Example response:

```

{
  "result" : {
    "balance" : 1030000000000
  }
}

```

POST /wallet/balanceOfType*Returns the global balance for given type of boxes***This endpoint needs authentication** (See [API authentication](#))**Parameters**

Name	Type	Required	Description
boxType	String	Yes	Type of boxes.

Example request:

Bash

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

Example response:

```

{
  "result" : {
    "balance" : 1030000000000
  }
}

```

POST /wallet/createPrivateKey25519

Creates new secret and returns corresponding address (public key)

This endpoint needs authentication (See [API authentication](#))

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" :
      ↪ "aea4154c7d88e956d480b913e5c3277db994b6d8f23240e7d49f3997d4e12c24"
    }
  }
}
```

POST /wallet/createVrfSecret

Creates new Vrf secret and returns corresponding public key

This endpoint needs authentication (See [API authentication](#))

No Parameters

Example request:

Bash

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

Example response:

```
{
  "result" : {
    "proposition" : {
      "publicKey" :
      ↪ "4439cbfd50af1b846e5ef06889d3192ef7a459bdd4640dc6da506062de43113c80"
    }
  }
}
```

POST /wallet/allPublicKeys

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

This endpoint needs authentication (See [API authentication](#))

Parameters

Name	Type	Required	Description
proptype	String	No	Proposition Type. If not specified, returns all propositions.

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" :
      ↪ "b78cf604e40a1a76b3e4736f6126b3a46b2ba7abf90101078fa1d9f098972a1d00"
    }, {
      "publicKey" :
      ↪ "a5b10622d70f094b7276e04608d97c7c699c8700164f78e16fe5e8082f4bb2ac"
    }, {
      "publicKey" :
      ↪ "accb2fbee54955df172a19506c8c35630e4146090d1e583858a025eea207583b80"
    }, {
      "publicKey" :
      ↪ "cc0161709c7589f8f6a4db76f78060ffa32861071bf8e89cace82521a42ee42e00"
    }, {
      "publicKey" :
      ↪ "a6b218079d2f476a47a849e2bb6cfd01427a631e9a1c8467e2951f4e524fa92100"
    }, {
      "publicKey" :
      ↪ "8c7538228452600d368b119015a230e5ae1a3a2eef500e02a92c55643c868c3c00"
    }, {
      "publicKey" :
      ↪ "7a3d71650ce54add0a7f773b8d37621dddc250390a09b725c66fad3ce606570d80"
    }, {
      "publicKey" :
      ↪ "9b64fc8291e238761b3262c833404268d9f4077c5f253fa177b113753832500980"
    }, {
      "publicKey" :
      ↪ "3650bcc96e3533d9352f5826efd1f5723cc2594d5aeb7efba228a8d23944492f80"
    }, {
      "publicKey" :
      ↪ "47286ba429e486767d35e79702206d1181894487f8d74550cbleec3b0bd9b5f3"
    }, {
      "publicKey" :
      ↪ "94ed9aa9928393fde71280023e1afcc2578cfbcb68bff90b6d2785dbbd6c7beb"
    }, {
      "publicKey" :
      ↪ "51c2d4c69602f30c8935551a076dc478eb196531996bb4dde4e345e115d3771a"
    }, {
      "publicKey" :
      ↪ "aea4154c7d88e956d480b913e5c3277db994b6d8f23240e7d49f3997d4e12c24"
    }, {
      "publicKey" :
      ↪ "4439cbfd50af1b846e5ef06889d3192ef7a459bdd4640dc6da506062de43113c80"
    } ]
  }
}
```

POST /wallet/importSecret

Import a secret into the wallet

This endpoint needs authentication (See [API authentication](#))

Parameters

Name	Type	Required	Description
privKey	String	Yes	Secret to import inside the wallet

Example request:

Bash

```
curl -X POST "http://127.0.0.1:9086/wallet/importSecret" -H "accept: application/json" -H "Content-Type: application/json" -d '{"privKey":"string"}'
```

Example response:

```
{
  "result" : {
    "proposition" :
    ↪ "4439cbfd50af1b846e5ef06889d3192ef7a459bdd4640dc6da506062de43113c80"
  }
}
```

POST /wallet/exportSecret

Export a secret corresponding to a public key from the wallet

This endpoint needs authentication (See [API authentication](#))

Parameters

Name	Type	Required	Description
publickey	String	Yes	PublicKey to export

Example request:

Bash

```
curl -X POST "http://127.0.0.1:9086/wallet/exportSecret" -H "accept: application/json" -H "Content-Type: application/json" -d '{"publicKey":"string"}'
```

Example response:

```
{
  "result" : {
    "privKey" :
    ↪ "002b64a179846da0b13ed5b4354dbdeb85a500c60ccb12c01a0fded2bd5d8b58e58bb8302e2b46763c830099c6fd8"
    ↪ ""
  }
}
```

POST /wallet/importSecrets

Import all the secret from a file. The file must contain in each line: SECRET + " " + PUBLICKEYS

This endpoint needs authentication (See [API authentication](#))

Parameters

Name	Type	Required	Description
path	String	Yes	Path to the file to import

Example request:

Bash

```
curl -X POST "http://127.0.0.1:9086/wallet/importSecrets" -H "accept: application/json" -H "Content-Type: application/json" -d "{\"path\":\"string\"}"
```

Example response:

```
{
  "result" : {
    "successfullyAdded" : 3,
    "failedToAdd": 1,
    "summary": [
      {
        "lineNumber": 2,
        "description": "string"
      }
    ]
  }
}
```

POST /wallet/dumpSecrets

Dump all the wallet secrets to a file

This endpoint needs authentication (See [API authentication](#))

Parameters

Name	Type	Required	Description
path	String	Yes	Path where the file will be created

Example request:

Bash

```
curl -X POST "http://127.0.0.1:9086/wallet/dumpSecrets" -H "accept: application/json" -H "Content-Type: application/json" -d "{\"path\":\"string\"}"
```

Example response:

```
{
  "result" : {
    "status": "string"
  }
}
```

Sidechain Node operations

POST /node/allPeers

Returns the list of all sidechain node peers

This endpoint needs authentication (See [API authentication](#))

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" : "/127.0.0.1:8430",
      "lastSeen" : 1650012289802,
      "name" : "node1",
      "connectionType" : "Outgoing"
    }, {
      "address" : "/127.0.0.1:8431",
      "lastSeen" : 1650012291959,
      "name" : "node2",
      "connectionType" : "Outgoing"
    } ]
  }
}
```

POST /node/connect

Sends the request to connect to a sidechain node

This endpoint needs authentication (See [API authentication](#))

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" : "/127.0.0.1:8330"
  }
}
```

POST /node/connectedPeers

Returns the list of all connected sidechain node peers

This endpoint needs authentication (See [API authentication](#))

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" : "/127.0.0.1:8430",
      "lastSeen" : 1650012289802,
      "name" : "node1",
      "connectionType" : "Outgoing"
    }, {
      "address" : "/127.0.0.1:8431",
      "lastSeen" : 1650012291959,
      "name" : "node2",
      "connectionType" : "Outgoing"
    } ]
  }
}
```

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" : [ ]
  }
}
```

POST /node/stop

Initiates a graceful stop procedure for the sidechain node. Returns an empty object

This endpoint needs authentication (See [API authentication](#))

No Parameters

Example request:

Bash

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

Example response:

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

POST /node/storageVersions

Returns the list of all node storage versions

No Parameters

Example request:

Bash

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

Example response:

```
{
  "result": {
    "listOfVersions": {
      "additionalProp1": "string",
      "additionalProp2": "string",
      "additionalProp3": "string"
    }
  },
  "error": {
    "code": "string",
    "description": "string",
    "detail": "string"
  }
}
```

POST /node/sidechainId

Returns the sidechain id

No Parameters

Example request:

Bash

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

Example response:

```
{
  "result" : {
    "sidechainId" :
    ↪ "0a1c910e65d7feb6f1dd53342cc212584d24f0ce643dbba88312e5630714850b"
  }
}
```

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

Finds Mainchain Block reference by its hash or by its height

Parameters

Name	Type	Description
hash	String	Block hash
height	int	Block height
format	boolean	If false, returns block hex format, otherwise returns JSONObject format

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" : {
    "blockReferenceInfo" : {
      "mainchainHeaderSidechainBlockId" :
      ↪ "650fe8657567b3b7779d30858177b1ba24b7bef6be250b443fca4e1bb9293a",
      "mainchainReferenceDataSidechainBlockId" :
      ↪ "650fe8657567b3b7779d30858177b1ba24b7bef6be250b443fca4e1bb9293a",
      "hash" :
      ↪ "09eddf0dbf848a6e866afd0d5dff4b2d7da6641250fcf8424b8077dab39eded2",
      "parentHash" :
      ↪ "0c4a3c40b60a96874720ff48798c3d2ff62840cc46b6401e5973fa78a294e61e",
      "height" : 420
    }
  }
}
```

POST /mainchain/blockReferenceByHash

*Reference block by hash***Parameters**

Name	Type	Description
hash	String	Block hash
format	boolean	If false, returns block hex format, otherwise returns JSONObject format

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" : {
        "version" : 3,
        "hashPrevBlock" :
        ↪ "0c4a3c40b60a96874720ff48798c3d2ff62840cc46b6401e5973fa78a294e61e",
        "hashMerkleRoot" :
        ↪ "fb83cf64532ae6f32197456ecad508b9970ae81aac5b8163911e44ebfa298525",
        "hashScTxCommitment" :
        ↪ "29a5a7b37890d5f5b5e5b17c709edff5d624988b3396048390ad8d067f9e6130",
        "time" : 1644418188,
        "bits" : 537857783,
        "nonce" :
        ↪ "00004a6e2b6bfef30470d239aef5f10fbd474670f733f710c4907abc5a00002c",
        "solution" :
        ↪ "1a33713b030af4cf1f2da118b1641401af7422166515838d14fbb73fa6d4f9393f4aa977",
        "hash" :
        ↪ "09eddf0dbf848a6e866afd0d5dff4b2d7da6641250fcf8424b8077dab39eded2"
      },
      "data" : {
        "headerHash" :
        ↪ "09eddf0dbf848a6e866afd0d5dff4b2d7da6641250fcf8424b8077dab39eded2",
        "sidechainRelatedAggregatedTransaction" : {
          "modifierTypeId" : 2,
          "id" :
          ↪ "ceee25a97f6c6d232a7237567c534fa9344113f22a6d6358ae75921605220865",
          "newBoxes" : [ {
            "nonce" : 4237164552941399434,
            "id" :
            ↪ "85a29f2ed2095fc5c8fc764061ce6c7ac85a26e42235ed12b99b5e323106e040",
            "vrfPubKey" : {
              "publicKey" :
              ↪ "076a9191a89fee51439600b0455db357a9899694d1cdad6a3c71bf65e6cce53280"
            },
            "blockSignProposition" : {
              "publicKey" :
              ↪ "a5b10622d70f094b7276e04608d97c7c699c8700164f78e16fe5e8082f4bb2ac"
            },
            "isCustom" : false,
            "value" : 60000000000,

```

(continues on next page)

(continued from previous page)

[illegible]

Certificate Submitter operations

POST /submitter/isCertGenerationActive

Returns if certificate generation is in progress

No Parameters

Example request:

Bash

```
curl -X POST "http://127.0.0.1:9085/submitter/isCertGenerationActive" -H "accept: application/json"
```

Example response:

```
{
  "result" : {
    "state" : false
  }
}
```

POST /submitter/isCertificateSubmitterEnabled

Returns if certificate submitter is enabled

No Parameters

Example request:

Bash

```
curl -X POST "http://127.0.0.1:9085/submitter/isCertificateSubmitterEnabled" -H "accept: application/json"
```

Example response:

```
{
  "result" : {
    "enabled" : false
  }
}
```

POST /submitter/enableCertificateSubmitter

Enables automatic certificate submission

This endpoint needs authentication (See [API authentication](#))

No Parameters**Example request:**

Bash

```
curl -X POST "http://127.0.0.1:9085/submitter/enableCertificateSubmitter" -H "accept: application/json"
```

Example response:

```
{
  "result": { }
```

POST /submitter/disableCertificateSubmitter

Disables automatic certificate submission

This endpoint needs authentication (See [API authentication](#))

No Parameters**Example request:**

Bash

```
curl -X POST "http://127.0.0.1:9085/submitter/disableCertificateSubmitter" -H "accept: application/json"
```

Example response:

```
{
  "result": { }
```

POST /submitter/isCertificateSignerEnabled

Returns if certificate signing option is enabled

No Parameters**Example request:**

Bash

```
curl -X POST "http://127.0.0.1:9085/submitter/isCertificateSignerEnabled" -H "accept: application/json"
```

Example response:

```
{
  "result":{
    "enabled" : false
  }
}
```

POST /submitter/enableCertificateSigner

Enables automatic certificate signing

This endpoint needs authentication (See [API authentication](#))

No Parameters

Example request:

Bash

```
curl -X POST "http://127.0.0.1:9085/submitter/enableCertificateSigner" -H "accept: application/json"
```

Example response:

```
{
  "result": { }
}
```

POST /submitter/disableCertificateSigner

Disables automatic certificate signing

This endpoint needs authentication (See [API authentication](#))

No Parameters

Example request:

Bash

```
curl -X POST "http://127.0.0.1:9085/submitter/disableCertificateSigner" -H "accept: application/json"
```

Example response:

```
{
  "result": { }
}
```

POST /submitter/getKeyRotationMessageToSignForSigningKey

Accepts public key and returns hash of the public key.

Parameters

Name	Type	Description
schnorrPublicKey	string	Public key of certificate signer
withdrawalEpoch	int	Number of withdrawal epoch

Example request:

Bash

```
curl -X POST "http://127.0.0.1:9085/submitter/getKeyRotationMessageToSignForSigningKey" -H "accept: application/json" -d '{"schnorrPublicKey":"string","withdrawalEpoch": 100}'
```

Example response:

```
{
  "result":{
    "keyRotationMessageToSign" :
    ↪ "4a2cbb9ff049b2a973c02e23f5cba3e1ac46d8bc030b75868b6510c764f0fc01"
  }
}
```

POST /submitter/getKeyRotationMessageToSignForMasterKey

Accepts public key and returns hash of the public key.

Parameters

Name	Type	Description
schnorrPublicKey	string	Public key of certificate signer
withdrawalEpoch	int	Number of withdrawal epoch

Example request:

Bash

```
curl -X POST "http://127.0.0.1:9085/submitter/getKeyRotationMessageToSignForMasterKey" -H "accept: application/json" -d '{"schnorrPublicKey":"string","withdrawalEpoch": 100}'
```

Example response:

```
{
  "result":{
    "keyRotationMessageToSign" :
    ↪ "4a2cbb9ff049b2a973c02e23f5cba3e1ac46d8bc030b75868b6510c764f0fc01"
  }
}
```

POST /submitter/getCertifiersKeys

Accepts number of withdrawal epoch and returns signer keys of certificate signers.

Parameters

Name	Type	Description
withdrawalEpoch	int	Withdrawal epoch of certificate signer keys

Example request:

Bash

```
curl -X POST "http://127.0.0.1:9085/submitter/getCertifiersKeys" -H "accept: application/json" -d '{"withdrawalEpoch": 100}'
```

Example response:

```
{
  "result": {
    "certifiersKeys": {
      "signingKeys": [{
        "publicKey":
        ↪ "ec4166e9225e97e90dde76089dd4edbb5ab60fb5ea60230a256ca3d2e4c2162c80"
      }, {
        "publicKey":
        ↪ "3fd1d98e4d4331f31d28a4b652ac9c7b3ea5ac1b35e0ef113434307b79cd590c80"
      }, {
        "publicKey":
        ↪ "b2130ed9458ff6f917b717b4765b185e40f6139ee7546830ba8ddd1f73b37b2400"
      }, {
        "publicKey":
        ↪ "ce0b8c7c4345a7fec79424cfa519d732d68aef16c7c0e5146c5efc2d9454601980"
      }, {
        "publicKey":
        ↪ "08be76211383c6cd3bfa7c72d49d5a79c79efd04d297535cf0004e5cf1ba7e0b00"
      }, {
        "publicKey":
        ↪ "606efe3b31cdab05fee935f58da6c88f7554f9bc55f0c6c3c577889a168aad3480"
      }, {
        "publicKey":
        ↪ "f9b41abe48c176f928b39ad66520969fd66be40c47dad5964b622f2b6620590580"
      }
    ],
    "masterKeys": [{
      "publicKey":
      ↪ "9b59d065c3373a70eab20263f6511a29d4af3aa20b3d9600295dcd985381bd2580"
    }, {
      "publicKey":
      ↪ "6edd6574af4d49474b981a89c8ff783b1bf3db63b2c818459ea130b4fab6bc0c80"
    }, {
      "publicKey":
      ↪ "39077d62d10ca0a9639908d0e7b3d37787d84d1a6c81624371015064383da02000"
    }, {
      "publicKey":
      ↪ "efd7e4f58e039f23bad6b7b5dc06c8c7a3c8f90a9f94ce6ae4164bc6ecb8f10980"
    }, {
      "publicKey":
      ↪ "a62b704bcc08e4c2fc4dd2ae51e6812dfa6519fc57db77812a3123639b5e4a3380"
    }, {
      "publicKey":
      ↪ "b46172f71951fe8a421ac77847821ac9f65105962f1cd2761ed9b0cf9400561500"
    }, {
      "publicKey":
      ↪ "a477534cac7bad0c77f81f8b5da7aec9582cebcf95de57aa6fafbc3cd7deca2480"
    }
  ]
}
}
```

POST /submitter/getKeyRotationProof

Returns key rotation proof (key type, index of key, new key value and 2 signatures proving key rotation) if type of circuit is *NaiveThresholdSignatureCircuitWithKeyRotation*.

Parameters

Name	Type	Description
withdrawalEpoch	int	Number of withdrawal epoch
indexOfKey	int	Index of certificate submitter key. Min = 0
keyType	int	Key type - 0 for signers key, 1 for masters key. Min = 0. Max = 1

Example request:

Bash

```
curl -X POST "http://127.0.0.1:9085/submitter/getKeyRotationProof" -H "accept: application/json" -d "{\"withdrawalEpoch\": 100, \"indexOfKey\": 2, \"keyType\": 0}"
```

Example response:

```
{
  "result": {
    "keyType" : 0,
    "index" : 0,
    "newKey" :
    ↪ "2cddac0f51b4329ab6ee85ccaf4e3bbcb1b80639a96e41239de978bd99d245f0a00",
    "signingKeySignature" :
    ↪ "1d39072beb8480edeee6dabf16ee15526bfd2170680dcc4a23d656bbb9740d1d0977f58009c19943d7964314aa9c9",
    ↪ ",
    "masterKeySignature" :
    ↪ "1d39072beb8480edeee6dabf16ee15526bfd2170680dcc4a23d656bbb9740d1d0977f58009c19943d7964314aa9c9",
    ↪ "
  }
}
```

Ceased Sidechain Withdrawal operations**POST /csw/hasCeased***Returns current status of the Sidechain***No Parameters****Example request:**

Bash

```
curl -X POST "http://127.0.0.1:9085/csw/hasCeased" -H "accept: application/json"
```

Example response:

```
{
  "result":{
    "state":true
  }
}
```

POST /csw/isCSWEnabled

Returns if the Ceased Sidechain Withdrawal is enabled in the Sidechain

No Parameters**Example request:**

Bash

```
curl -X POST "http://127.0.0.1:9085/csw/isCSWEnabled" -H "accept: application/json"
```

Example response:

```
{
  "result" : {
    "cswEnabled" : true
  }
}
```

POST /csw/generateCswProof

Tries to generate csw proof and returns current status of this operation. Possible statuses are:

- *SidechainIsAlive* - Sidechain is still alive;
- *InvalidAddress* - Receiver address has invalid value: MC toaddress expected;
- *NoProofData* - Information for given box id is missed;
- *ProofGenerationStarted* - Started proof generation, was not started of present before;
- *ProofGenerationInProgress* - Proof generation was started before, still in process;
- *ProofCreationFinished* - Proof is ready.

This endpoint needs authentication (See [API authentication](#))

Parameters

Name	Type	Description
boxId	String	Coin box id in hex
receiverAddress	String	Horizen address

Example request:

Bash

```
curl -X POST "http://127.0.0.1:9085/csw/generateCswProof" -H "accept: application/json" -H "Content-Type: application/json" -d '{"boxId":"string","receiverAddress":"string"}'
```

Example response:

```
{
  "result" : {
    "state" : "ProofCreationFinished",
    "description" : "CSW proof generation is finished"
  }
}
```

POST /csw/cswInfo

Returns information about csw proof for given box id

Parameters

Name	Type	Description
boxId	String	Coin box id in hex

Example request:

Bash

```
curl -X POST "http://127.0.0.1:9085/csw/cwsInfo" -H "accept: application/json" -H "Content-Type: application/json" -d '{"boxId": "string"}'
```

Example response:

```
{
  "result" : {
    "cswInfo" : {
      "cswType" : "UtxoCswData",
      "amount" : 1000000000,
      "scId" :
        ↪ "1b22519c38b6415a96a3cf73e1227bae06a2a2cf73a935685e19c9a1a9480da7",
      "nullifier" :
        ↪ "e93952c5eab52b9648b5b711c11c134f05710a5e9de86c23608f690d31b7c300",
      "proofInfo" : {
        "status" : "Generated",
        "scProof" :
          ↪ "0201a5711a4337d74df34e3a44278a2b77bee9b56a1277afc8451cbcd2c0e5ad7a39000002dc3816c9c0dace2dd52",
          ↪ ",
        "receiverAddress" : "ztZd886uGxthiBhwSsQEmi72uDA6GNPeHua"
      },
      "activeCertData" :
        ↪ "5e17042bbba41298dc5c057176ce24d018552b6435d63d88a44740bbc767d833",
      "ceasingCumScTxCommTree" :
        ↪ "bcb01e73e4256d436ed404f196de7d923c46a89d351f27056fe757a144a4601b"
    }
  }
}
```

POST /csw/cswBoxIds

Returns list of available box ids for csw

No Parameters

Example request:

Bash

```
curl -X POST "http://127.0.0.1:9085/csw/cswBoxIds" -H "accept: application/json"
```

Example response:

```
{
  "result" : {
    "cswBoxIds" : [
      ↪ "e814e033de45a7d7770180ee0de8a5bef0ac83edc69b8665f496ef04e728a83c",
      ↪ "6dc9f828a5351c655e313307f118539abd496068d80d9e31d3dcf20fc48d11c7",
      ↪ "439a69030d1db3d1cd8c4d2a2badd02fdd9030066bald9df946ebbef95606d4b",
      ↪ "2630250338195552e9154ce051dc764c3bb5db24dlf3fffe3975b5105e07e098",
      ↪ "3f244c0dc44bbe48262327f4dc88b913d232bc134957dd2c739d08dd1ae69a43",
      ↪ "28b2a31d7e46a0e7978e6e55ffbec9f270da8d2734d9dc87cdd6fe3eb910a727",
      ↪ "ee17566ffd58d10c6d4b99ee66baed18090a11f37b99ec288caedd3c19ba69c6",
      ↪ "6ee25cfe5fcce8f858b6c74a4fa37ff05d768caa1988aebfcdb945bc0696ef70",
      ↪ "c78775169ebb30b80862c2daa90bce7637d2b15c282a713ff33d376007b62a57" ]
    }
  }
}
```

(continued from previous page)

```
}  
}
```

POST /csw/nullifier

Returns nullifier for given box id

Parameters

Name	Type	Description
boxId	String	Coin box id in hex

Example request:

Bash

```
curl -X POST "http://127.0.0.1:9085/csw/nullifier" -H "accept: application/json" -H "Content-Type: application/json"  
-d "{\"boxId\":\"string\"}"
```

Example response:

```
{  
  "result" : {  
    "nullifier" :  
    ↪ "e93952c5eab52b9648b5b711c11c134f05710a5e9de86c23608f690d31b7c300"  
  }  
}
```

Sidechain Backup operations**POST /backup/getSidechainBlockIdForBackup**

Returns the sidechain block id to use in the backup procedure.

The block id is calculated using the following formula:

- $Genesis_MC_block_height + (current_epoch - 2) * withdrawalEpochLength - 1$

This endpoint returns an error in case the sidechain current epoch is less than 3.

No Parameters**Example request:**

Bash

```
curl -X POST "http://127.0.0.1:9085/backup/getSidechainBlockIdForBackup" -H "accept: application/json"
```

Example response:

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

POST /backup/getRestoredBoxes

Returns the non-coin boxes restored by the restore procedure in a paginated way.

Parameters

Name	Type	Description
numberOfElements	int	Number of boxes to return. Max = 100
lastBoxId	string	Last box id received. It's optional and in case of empty or non value the endpoint starts to answer with the first box found.

Example request:

Bash

```
curl -X POST "http://127.0.0.1:9085/backup/getRestoredBoxes" -H "accept: application/json" -d "{\"numberOfElements\": 100, \"lastBoxId\": \"\"}"
```

Example response:

```
{
  "result": {
    "boxes": [ {
      "customUuid" :
      ↪ "71723462d695198c31e65136c9cc42c50b23c478f165c8957cb0509fd123cbb8",
      "customValue" : 866000000,
      "nonce" : 3509023985616518242,
      "id" :
      ↪ "84e2dd2f114a829422345fb0f27dfd836a803235f3e82418185aa68a0ba2f3b8",
      "typeName" : "CustomBox",
      "proposition" : {
        "publicKey" :
        ↪ "46630ae9f76aa3359a3007566aa83e661cdd0d024b484b85a6d9e0d2e4d51fb5"
      },
      "isCustom" : true
    }, {
      "customUuid" :
      ↪ "71723462d695198c31e65136c9cc42c50b23c478f165c8957cb0509fd123cbb8",
      "customValue" : 3561000000,
      "nonce" : 6626025734618418495,
      "id" :
      ↪ "8e5757b44d0199ee75ecd6a7cfb7c7deb8f675e14670f976abfe59f3521eae97",
      "typeName" : "CustomBox",
      "proposition" : {
        "publicKey" :
        ↪ "52ba271cc2d786c8197901679f1f7d47112c9dbee0dde082387a2295d7a8e074"
      },
      "isCustom" : true
    }, {
      "customUuid" :
      ↪ "71723462d695198c31e65136c9cc42c50b23c478f165c8957cb0509fd123cbb8",
      "customValue" : 4165000000,
      "nonce" : -7600613233944287562,
      "id" :
      ↪ "54f3d032494cbc4bccedc9a64d181b0e7e8ddd3d69f960db52b677995b6ecc2e",

```

(continues on next page)

(continued from previous page)

```
        "typeName" : "CustomBox",
        "proposition" : {
            "publicKey" :
↪ "7377afd5a3d6d134a0748e7f0f3d9b11b67295a98384ee364bb464bafefbf5dc9"
        },
        "isCustom" : true
    } ],
    "startingBoxId" :
↪ "e49866604b904546b7a83b04ff0fa131528de045bff8199af8cc47b9516cb512"
    }
}
```

3.3 Guides

[Zend to Zend_oo Block_Explorer guide pdf](#)

[Zend to Zend_oo Exchanges guide pdf](#)

HTTP ROUTING TABLE

/backup

POST /backup/getRestoredBoxes, [111](#)

POST /backup/getSidechainBlockIdForBackup,
[110](#)

/block

POST /block/best, [75](#)

POST /block/findBlockInfoById, [77](#)

POST /block/findById, [72](#)

POST /block/findIdByHeight, [74](#)

POST /block/findLastIds, [73](#)

POST /block/forgingInfo, [78](#)

POST /block/generate, [78](#)

POST /block/getFeePayments, [76](#)

POST /block/startForging, [77](#)

POST /block/stopForging, [78](#)

/csw

POST /csw/cswBoxIds, [109](#)

POST /csw/cswInfo, [108](#)

POST /csw/generateCswProof, [108](#)

POST /csw/hasCeased, [107](#)

POST /csw/isCSWEnabled, [107](#)

POST /csw/nullifier, [110](#)

/mainchain

POST /mainchain/bestBlockReferenceInfo,
[99](#)

POST /mainchain/blockReferenceByHash,
[100](#)

POST /mainchain/blockReferenceInfoBy,
[100](#)

POST /mainchain/genesisBlockReferenceInfo,
[99](#)

/node

POST /node/allPeers, [96](#)

POST /node/blacklistedPeers, [97](#)

POST /node/connect, [96](#)

POST /node/connectedPeers, [97](#)

POST /node/sidechainId, [98](#)

POST /node/stop, [97](#)

POST /node/storageVersions, [98](#)

/submitter

POST /submitter/disableCertificateSigner,
[104](#)

POST /submitter/disableCertificateSubmitter,
[103](#)

POST /submitter/enableCertificateSigner,
[104](#)

POST /submitter/enableCertificateSubmitter,
[103](#)

POST /submitter/getCertifiersKeys, [105](#)

POST /submitter/getKeyRotationMessageToSignForMaster,
[105](#)

POST /submitter/getKeyRotationMessageToSignForSigner,
[104](#)

POST /submitter/getKeyRotationProof, [106](#)

POST /submitter/isCertGenerationActive,
[102](#)

POST /submitter/isCertificateSignerEnabled,
[103](#)

POST /submitter/isCertificateSubmitterEnabled,
[102](#)

/transaction

POST /transaction/allTransactions, [79](#)

POST /transaction/createCoreTransaction,
[82](#)

POST /transaction/createCoreTransactionSimplified,
[83](#)

POST /transaction/createKeyRotationTransaction,
[88](#)

POST /transaction/decodeTransactionBytes,
[81](#)

POST /transaction/findById, [80](#)

POST /transaction/makeForgerStake, [86](#)

POST /transaction/sendCoinsToAddress,
[85](#)

POST /transaction/sendTransaction, [88](#)

POST /transaction/spendForgingStake, [87](#)

POST /transaction/withdrawCoins, [85](#)

/wallet

POST /wallet/allBoxes, [89](#)
POST /wallet/allPublicKeys, [92](#)
POST /wallet/balanceOfType, [91](#)
POST /wallet/coinsBalance, [91](#)
POST /wallet/createPrivateKey25519, [91](#)
POST /wallet/createVrfSecret, [92](#)
POST /wallet/dumpSecrets, [95](#)
POST /wallet/exportSecret, [94](#)
POST /wallet/importSecret, [93](#)
POST /wallet/importSecrets, [94](#)