Horizen Sidechain SDK

Release 1.0

Nov 23, 2020

Contents

CHAPTER 1

Overview

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

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

1.1 Tutorials - start here

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

1.2 how-to

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

1.3 key-topics

Explanation and analysis of some key concepts in Sidechain SDK.

1.4 Reference

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

CHAPTER 2

Join us online

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

Join our Discord Server, and check the #sidechains channel

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

CHAPTER 3

Why Horizen Sidechains?

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

- **A Novel Construction**

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

- **Scalability and Flexibility**

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

- **Decentralization**

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

- **Privacy and Auditability**

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

- **Easy Deployment with the Sidechain SDK**

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

3.1 Tutorials

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

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

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

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

3.1.1 Before you start

This tutorial offers Java developers all the 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 key,s 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

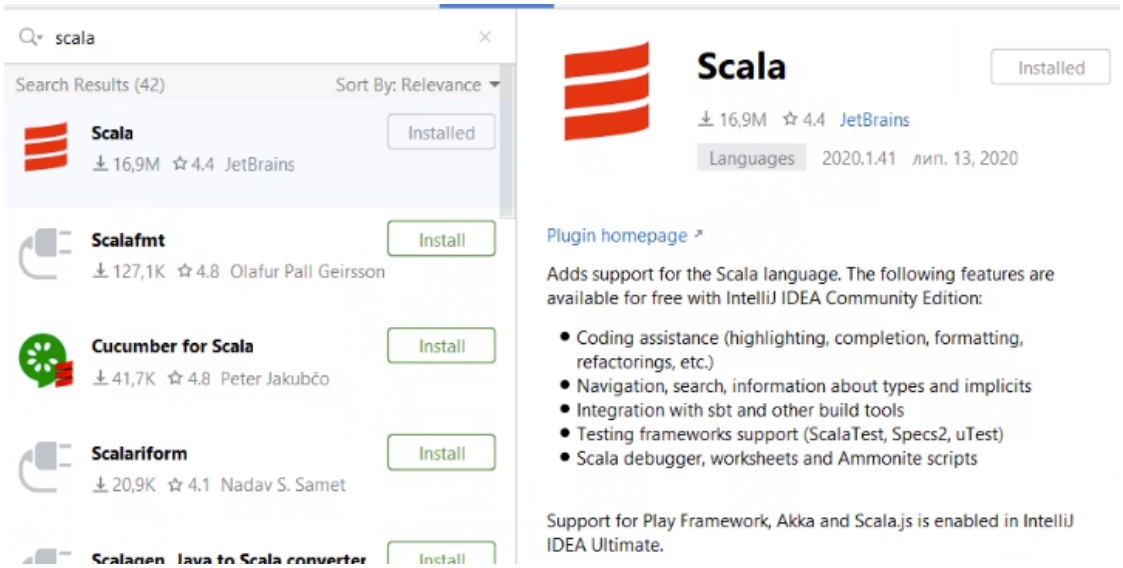
The Sidechain SDK requires Java 8 or newer (Java 11 recommended), Scala 2.12.10+ or newer, and the latest version of zend_oo.

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:



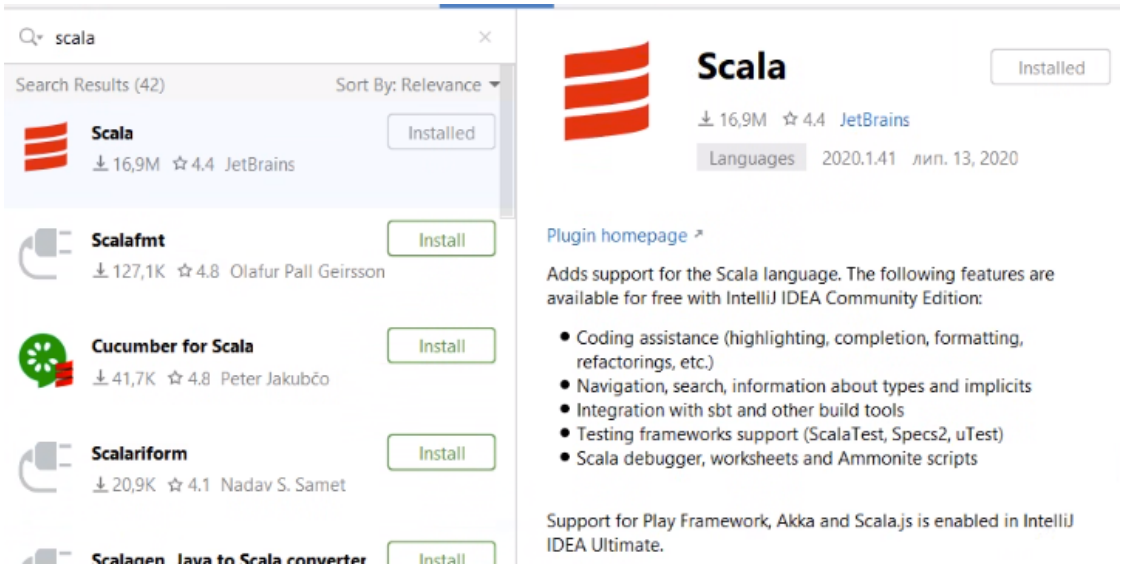
6. In the IDE, you can now go to File and Open the root directory of the project repository, "Sidechains-SDK". The pom.xml file - the Maven Project Object Model XML file that contains all the project configuration details - should be automatically imported by the IDE. Otherwise, you can just open it.
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 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 NoncedBox, which inherits Box and contains some Nonce data. Nonce data is a value that is deterministically assigned to the box depending on the Transaction that includes it, and the index of the Box inside the Transaction outputs list. This way we can guarantee that two boxes with the same data (proposition, amount and other custom fields) will have different nonces, so will have different unique box ids.

A Transaction is a sequence of inputs and outputs. Each input consists of a reference to the Box being opened, and a Proof that satisfies the condition of its Proposition. Each output is a new Box instance.

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 two basic operations:

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

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

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

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

Forward Transfer

A forward transfer sends coins from the mainchain to a sidechain. The Horizen Mainchain supports a "Forward Transfer" transaction type that specifies the sidechain destination (*sidechain id* and *receiver address*) and the amount of ZEN to be sent. From a 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 WithdrawalRequestBox 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.

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.

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 possability) every 720 * 120 = 86400 seconds, i.e. 24 hours.
- BlockSize Limit 2MB

3.1.6 Node communication

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

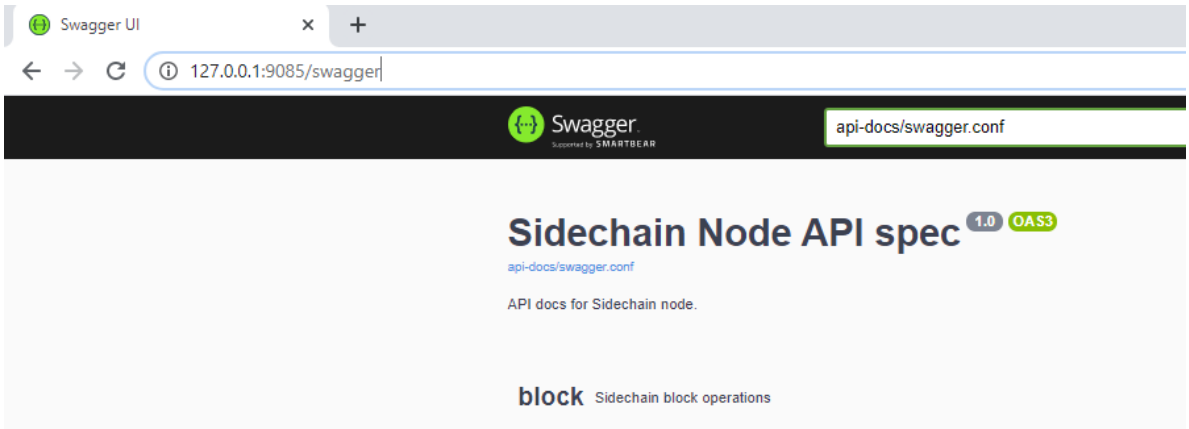
bindAddress – "IP:port" address for sending HTTP request, e.g. "127.0.0.1:9085"

api-key-hash – Authentication header must be a string that hashes to the field "api-key-hash" specified in each sidechain node's .conf file. The authentication header could be empty if no api-key-hash 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.

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 RegularBox 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). The SidechainCoreTransaction can be extended to support custom logic operations. For example, if we think about a real-estate sidechain, we can tokenize some private property as a specific Box using SidechainCoreTransaction. 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.

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".

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 IODB Library. 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 com.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()

    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,

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

        public void run()
}
```

The SidechainApp instance can be instantiated directly or through 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 com.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 box data serializers

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

```
bind(new TypeLiteral<HashMap<Byte,NoncedBoxDataSerializer<NoncedBoxData<Proposition,␣
↪NoncedBox<Proposition>>>>>(){})
.annotatedWith(Names.named("CustomBoxDataSerializers"))
.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 proposition serializers

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

```
bind(new TypeLiteral<HashMap<Byte, ProofSerializer<Proof<Proposition>>>>() {})
.annotatedWith(Names.named("CustomProofSerializers"))
.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 com.horizen.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 com.horizen.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 com.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 com.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: com.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 com.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 com.horizen.storage.Storage interface.

```
bind(Storage.class)
.annotatedWith(Names.named("StateStorage"))
.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 com.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 com.horizen.storage.Storage interface.

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

SidechainApp arguments can be split into 4 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 be 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

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.

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 is to 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 `com.horizen.box.NoncedBox` 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 com.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.

As a common design rule, you usually do not implement the NoncedBox interface directly, but extend instead the abstract class com.horizen.box.AbstractNoncedBox, which already provides default implementations of some useful methods like `id()`, `equals()` and `hashCode()`. This class requires the definition of another object: a class extending com.horizen.box.AbstractNoncedBox, where you should put all the properties of the box, including the proposition. You can think of the AbstractNoncedBoxData as an inner container of all the fields of your box. This data object must be passed in the constructor of AbstractNoncedBox, along with the nonce. The important methods of AbstractNoncedBoxData 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.
- `NoncedBoxDataSerializer serializer()` should return the serializer of this box data (see below)

BoxSerializer and NoncedBoxDataSerializer

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 com.horizen.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 AbstractNoncedBoxData need's to have its own serializer: if you declare a box-Data, you should define one in a similar way. In this case the interface to be implemented is com.horizen.box.data.NoncedBoxDataSerializer

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. To define a new custom transaction, you have to extend the com.horizen.transaction.BoxTransaction class. 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 the process requires to spend the box and create a new one with the updated values.

- `public List<NoncedBox<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<NoncedBox<Proposition>> newBoxes() {
  List<NoncedBox<Proposition>> newBoxes =  .....  //new boxes are created here
  //....
  return Collections.unmodifiableList(newBoxes);
}
```

- `public long fee()` Returns the fee to be paid to execute this transaction.
- `public long timestamp()` Returns the timestamp of the transaction creation. As a good practice, timestamp should be created outside transaction, passed in the transaction's constructor, and returned here.
- `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. check 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.

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.
```

(continues on next page)

(continued from previous page)

```
  @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[] getScondPublicKeyBytes() { 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()
            );
      }

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

(continues on next page)

(continued from previous page)

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

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

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

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

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 returns false, 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.

```
public boolean validate(SidechainStateReader stateReader, SidechainBlock␣
↪block)
```

- Custom block validation should happen here. If the function returns false, 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)
```

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);
}
```

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.

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 com.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.Http Route object and defines a specific endpoint url and its logic. To simplify the developement, 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));
      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(){
```

(continues on next page)

(continued from previous page)

```
    return someBytes;
  }

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

  public int getNumber(){
    return number;
  }

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

API response classes

The function that processes the request must return an object of type com.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 CustomSuccessResponce implements SuccessResponse{
  private final String response;

  public CustomSuccessResponce (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":
  {
```

(continues on next page)

(continued from previous page)

```
  "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);
```

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 AbstractNoncedBox
↪<PublicKey25519Proposition, CarBoxData, CarBox> {

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

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

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

(continues on next page)

(continued from previous page)

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

    public static CarBox parseBytes(byte[] bytes) {
        long nonce = Longs.fromByteArray(Arrays.copyOf(bytes, Longs.BYTES));
        CarBoxData boxData = CarBoxDataSerializer.getSerializer().
↪parseBytes(Arrays.copyOfRange(bytes, Longs.BYTES, bytes.length));
        return new CarBox(boxData, nonce);
    }

    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 AbstractNoncedBox
 ↪<PublicKey25519Proposition, CarBoxData, CarBox> {
```

Our class extends the *AbstractNoncedBox* 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 AbstractNoncedBox 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 three methods are used for serialization and deserialization of our Box: they define the serializer to be used, and the methods used to generate a byte array from the box and to obtain the box back from the byte array (note that they delegate the byte handling logic to the CarBoxData):

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

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

public static CarBox parseBytes(byte[] bytes) {
    long nonce = Longs.fromByteArray(Arrays.copyOf(bytes, Longs.
↪BYTES));
    CarBoxData boxData = CarBoxDataSerializer.getSerializer().
↪parseBytes(Arrays.copyOfRange(bytes, Longs.BYTES, bytes.length));
    return new CarBox(boxData, nonce);
}
```

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 AbstractNoncedBox, as we did for the CarBox, and it is in any case a good practice.

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

    private final String vin;   // Vehicle Identification Number
    private final int year;     // Car manufacture year
    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, 0);
            this.vin = vin;
            this.year = year;
            this.model = model;
            this.color = 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 NoncedBoxDataSerializer serializer() {
            return CarBoxDataSerializer.getSerializer();
    }

    @Override
    public byte boxDataTypeId() {
            return CarBoxDataId.id();
    }

    @Override
    public byte[] bytes() {
            return Bytes.concat(
                    proposition().bytes(),
                    Ints.toByteArray(vin.getBytes().length),
                    vin.getBytes(),
                    Ints.toByteArray(year),
                    Ints.toByteArray(model.getBytes().length),
                    model.getBytes(),
                    Ints.toByteArray(color.getBytes().length),
                    color.getBytes()
            );
```

(continues on next page)

(continued from previous page)

```
    }

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

            PublicKey25519Proposition proposition =␣
↪PublicKey25519PropositionSerializer.getSerializer()
            .parseBytes(Arrays.copyOf(bytes,␣
↪PublicKey25519Proposition.getLength()));
            offset += PublicKey25519Proposition.getLength();

            int size = Ints.fromByteArray(Arrays.copyOfRange(bytes,␣
↪offset, offset + Ints.BYTES));
            offset += Ints.BYTES;

            String vin = new String(Arrays.copyOfRange(bytes, offset,
↪ offset + size));
            offset += size;

            int year = Ints.fromByteArray(Arrays.copyOfRange(bytes,␣
↪offset, offset + Ints.BYTES));
            offset += Ints.BYTES;

            size = Ints.fromByteArray(Arrays.copyOfRange(bytes,␣
↪offset, offset + Ints.BYTES));
            offset += Ints.BYTES;

            String model = new String(Arrays.copyOfRange(bytes,␣
↪offset, offset + size));
            offset += size;

            size = Ints.fromByteArray(Arrays.copyOfRange(bytes,␣
↪offset, offset + Ints.BYTES));
            offset += Ints.BYTES;

            String color = new String(Arrays.copyOfRange(bytes,␣
↪offset, offset + size));

            return new CarBoxData(proposition, vin, year, model,␣
↪color);
    }

    public String getVin() {
            return vin;
    }

    public int getYear() {
            return year;
    }

    public String getModel() {
            return model;
    }

    public String getColor() {
            return color;
    }
```

(continues on next page)

(continued from previous page)

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

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

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

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

```
public CarBoxData(PublicKey25519Proposition proposition, String vin,
                  int year, String model, String color) {
   super(proposition, 0);
   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 0 constant value.

```
@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 NoncedBoxDataSerializer serializer() {
    return CarBoxDataSerializer.getSerializer();
}

@Override
public byte boxDataTypeId() {
    return CarBoxDataId.id();
}
```

Two very important methods are *bytes()* and *parseBytes()*: they contain the logic to serialize and deserialize properties and proposition. The code is quite verbose but simple: *bytes()* returns a byte array that is the concatenation of all the properties values, while *parseBytes()* reads it and writes the values back. Note that for variable-length fields like strings, the field length needs to be first known and serialized, and made part of the bytearray, so that *parseBytes()* can then read the correct length of bytes of that field. You can see it in the code that serializes the car model string:

```
return Bytes.concat(
            ....
            Ints.toByteArray(model.getBytes().length),
            model.getBytes(),
            ....
    );
```

and this is the code in *parseBytes()* that reads the bytearray and writes back the car model:

```
size = Ints.fromByteArray(Arrays.copyOfRange(bytes, offset, offset +␣
↪Ints.BYTES));
offset += Ints.BYTES;
String model = new String(Arrays.copyOfRange(bytes, offset, offset +␣
↪size));
```

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.putBytes(box.bytes());
    }

    @Override
    public CarBox parse(Reader reader) {
            return CarBox.parseBytes(reader.getBytes(reader.
↪remaining()));
    }
    }
```

```
public final class CarBoxDataSerializer implements␣
↪NoncedBoxDataSerializer<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) {
        writer.putBytes(boxData.bytes());
    }

    @Override
    public CarBoxData parse(Reader reader) {
        return CarBoxData.parseBytes(reader.getBytes(reader.
↪remaining()));
    }
}
```

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;
    private List<NoncedBox<Proposition>> newBoxes;

    public BuyCarTransaction(List<byte[]> inputRegularBoxIds,
                         List<Signature25519> inputRegularBoxProofs,
                         List<RegularBoxData> outputRegularBoxesData,
                         CarBuyOrderInfo carBuyOrderInfo,
                         long fee,
                         long timestamp) {
            super(inputRegularBoxIds,
                  inputRegularBoxProofs,
                  outputRegularBoxesData,
                  fee, timestamp);
            this.carBuyOrderInfo = carBuyOrderInfo;
    }

    @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();
            }
            };
            unlockers.add(unlocker);
            return unlockers;
    }


    @Override
    public List<NoncedBox<Proposition>> newBoxes() {
            if(newBoxes == null) {
                // Get new boxes from base class.
```

(continues on next page)

(continued from previous page)

```
                newBoxes = new ArrayList<>(super.newBoxes());

                // Set CarBox with specific owner depends on proof. See␣
↪CarBuyOrderInfo.getNewOwnerCarBoxData() definition.
                long nonce = getNewBoxNonce(carBuyOrderInfo.
↪getNewOwnerCarBoxData().proposition(), newBoxes.size());
            newBoxes.add((NoncedBox) new CarBox(carBuyOrderInfo.
↪getNewOwnerCarBoxData(), nonce));

            // If Sell Order was opened by the buyer -> add payment box for␣
↪Car previous owner.
            if (!carBuyOrderInfo.isSpentByOwner()) {
                    RegularBoxData paymentBoxData = carBuyOrderInfo.
↪getPaymentBoxData();
                    nonce = getNewBoxNonce(paymentBoxData.proposition(),␣
↪newBoxes.size());
                    newBoxes.add((NoncedBox) new RegularBox(paymentBoxData,␣
↪nonce));
            }
            }
            return Collections.unmodifiableList(newBoxes);
    }

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

    // Define object serialization, that should serialize both parent class␣
↪entries and CarBuyOrderInfo as well
    @Override
    public byte[] bytes() {
            ByteArrayOutputStream inputsIdsStream = new␣
↪ByteArrayOutputStream();
            for(byte[] id: inputRegularBoxIds)
            inputsIdsStream.write(id, 0, id.length);

            byte[] inputRegularBoxIdsBytes = inputsIdsStream.toByteArray();
            byte[] inputRegularBoxProofsBytes = regularBoxProofsSerializer.
↪toBytes(inputRegularBoxProofs);
            byte[] outputRegularBoxesDataBytes =␣
↪regularBoxDataListSerializer.toBytes(outputRegularBoxesData);
            byte[] carBuyOrderInfoBytes = carBuyOrderInfo.bytes();

            return Bytes.concat(
            Longs.toByteArray(fee()),                                   // 8␣
↪bytes
            Longs.toByteArray(timestamp()),                             // 8␣
↪bytes
            Ints.toByteArray(inputRegularBoxIdsBytes.length),           // 4␣
↪bytes
            inputRegularBoxIdsBytes,                                    //␣
↪depends on previous value (>=4 bytes)
            Ints.toByteArray(inputRegularBoxProofsBytes.length),        // 4␣
↪bytes
            inputRegularBoxProofsBytes,                                 //␣
↪depends on previous value (>=4 bytes)
```

(continues on next page)

(continued from previous page)

```
            Ints.toByteArray(outputRegularBoxesDataBytes.length),    // 4␣
↪bytes
            outputRegularBoxesDataBytes,                             //␣
↪depends on previous value (>=4 bytes)
            Ints.toByteArray(carBuyOrderInfoBytes.length),           // 4␣
↪bytes
            carBuyOrderInfoBytes                                     //␣
↪depends on previous value (>=4 bytes)
            );
    }

    // Define object deserialization similar to 'toBytes()' representation.
    public static BuyCarTransaction parseBytes(byte[] bytes) {
            int offset = 0;

            long fee = BytesUtils.getLong(bytes, offset);
            offset += 8;

            long timestamp = BytesUtils.getLong(bytes, offset);
            offset += 8;

            int batchSize = BytesUtils.getInt(bytes, offset);
            offset += 4;

            ArrayList<byte[]> inputRegularBoxIds = new ArrayList<>();
            int idLength = NodeViewModifier$.MODULE$.ModifierIdSize();
            while(batchSize > 0) {
            inputRegularBoxIds.add(Arrays.copyOfRange(bytes, offset, offset␣
↪+ idLength));
            offset += idLength;
            batchSize -= idLength;
            }

            batchSize = BytesUtils.getInt(bytes, offset);
            offset += 4;

            List<Signature25519> inputRegularBoxProofs =␣
↪regularBoxProofsSerializer.parseBytes(Arrays.copyOfRange(bytes, offset,␣
↪offset + batchSize));
            offset += batchSize;

            batchSize = BytesUtils.getInt(bytes, offset);
            offset += 4;

            List<RegularBoxData> outputRegularBoxesData =␣
↪regularBoxDataListSerializer.parseBytes(Arrays.copyOfRange(bytes, offset,␣
↪offset + batchSize));
            offset += batchSize;

            batchSize = BytesUtils.getInt(bytes, offset);
            offset += 4;

            CarBuyOrderInfo carBuyOrderInfo = CarBuyOrderInfo.
↪parseBytes(Arrays.copyOfRange(bytes, offset, offset + batchSize));
            return new BuyCarTransaction(inputRegularBoxIds,␣
↪inputRegularBoxProofs, outputRegularBoxesData, carBuyOrderInfo, fee,␣
↪timestamp);
```

(continues on next page)

(continued from previous page)

```
    }

    // 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[]> inputRegularBoxIds,
                         List<Signature25519> inputRegularBoxProofs,
                         List<RegularBoxData> outputRegularBoxesData,
                         CarBuyOrderInfo carBuyOrderInfo,
                         long fee,
                         long timestamp) {
    super(inputRegularBoxIds,
          inputRegularBoxProofs,
          outputRegularBoxesData,
          fee, timestamp);
    this.carBuyOrderInfo = carBuyOrderInfo;
}
```

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 timestamp), 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.
    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 *bytes()* and *ParseBytes(byte[] bytes)*, 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) {
            writer.putBytes(transaction.bytes());
    }

    @Override
    public BuyCarTransaction parse(Reader reader) {
            return BuyCarTransaction.parseBytes(reader.getBytes(reader.
↪remaining()));
    }
    }
```

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 {
    private final CarSellOrderBox carSellOrderBoxToOpen;  // Sell␣
↪order box to be spent in BuyCarTransaction
    private final SellOrderSpendingProof proof;           // Proof␣
↪to unlock the box above

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

(continues on next page)

(continued from previous page)

```
            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()
            );
    }

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

    // CarBuyOrderInfo minimal bytes representation.
    public byte[] bytes() {
            byte[] carSellOrderBoxToOpenBytes =␣
↪CarSellOrderBoxSerializer.getSerializer().
↪toBytes(carSellOrderBoxToOpen);
```

(continues on next page)

(continued from previous page)

```
            byte[] proofBytes = SellOrderSpendingProofSerializer.
↪getSerializer().toBytes(proof);

            return Bytes.concat(
            Ints.toByteArray(carSellOrderBoxToOpenBytes.length),
            carSellOrderBoxToOpenBytes,
            Ints.toByteArray(proofBytes.length),
            proofBytes
            );
    }

    // Define object deserialization similar to 'toBytes()'␣
↪representation.
    public static CarBuyOrderInfo parseBytes(byte[] bytes) {
            int offset = 0;

            int batchSize = BytesUtils.getInt(bytes, offset);
            offset += 4;

            CarSellOrderBox carSellOrderBoxToOpen =␣
↪CarSellOrderBoxSerializer.getSerializer().parseBytes(Arrays.
↪copyOfRange(bytes, offset, offset + batchSize));
            offset += batchSize;

            batchSize = BytesUtils.getInt(bytes, offset);
            offset += 4;

            SellOrderSpendingProof proof =␣
↪SellOrderSpendingProofSerializer.getSerializer().parseBytes(Arrays.
↪copyOfRange(bytes, offset, offset + batchSize));

            return new CarBuyOrderInfo(carSellOrderBoxToOpen, proof);
    }
    }
```

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(),
```

(continues on next page)

(continued from previous page)

```
            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 RegularBoxData getPaymentBoxData() {
    return new RegularBoxData(
            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> {
    private static final int KEY_LENGTH = Ed25519.keyLength();

    // 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));
```

(continues on next page)

(continued from previous page)

```
            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);

            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 byte[] bytes() {
            return Bytes.concat(
            ownerPublicKeyBytes,
            buyerPublicKeyBytes
            );
    }

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

            byte[] ownerPublicKeyBytes = Arrays.copyOfRange(bytes, offset,␣
↪offset + KEY_LENGTH);
            offset += KEY_LENGTH;

            byte[] buyerPublicKeyBytes = Arrays.copyOfRange(bytes, offset,␣
↪offset + KEY_LENGTH);

            return new SellOrderProposition(ownerPublicKeyBytes,␣
↪buyerPublicKeyBytes);

    }

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

    @Override
    public int hashCode() {
```

(continues on next page)

(continued from previous page)

```
            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);
}
```

- **provide the usual methods for serialization and deserialization:**
  - byte[] bytes()
  - parseBytes(byte[] bytes)
  - serializer()
- **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)
```

(continues on next page)

(continued from previous page)

```
        return true;
    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> {

    private final boolean isSeller;

    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;
    }

    public boolean isSeller() {
            return isSeller;
    }

    @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 proofTypeId() {
            return CarRegistryProofsIdsEnum.SellOrderSpendingProofId.
↪id();
    }

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

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();
}
```

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 AbstractNoncedBox**<SellOrderProposition**, CarSellOrderBoxData, CarSellOrderBox>

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

Application state

By implementing the *com.horizen.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 {

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

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

    boolean validate(SidechainStateReader stateReader, SidechainBlock block);

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

    }
```

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.

All the 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, staring from the *onApplyChanges* method:

```
@Override
public Try<ApplicationState> onApplyChanges(SidechainStateReader stateReader,
                                            byte[] version,
```

(continues on next page)

(continued from previous page)

```
                                                    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(version, 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
public boolean validate(SidechainStateReader stateReader, BoxTransaction
↪<Proposition, Box<Proposition>> transaction) {
    // 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())){
                return false;
            }
        }
    }
    return true;
}
```

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
public boolean validate(SidechainStateReader stateReader, SidechainBlock␣
↪block) {
    //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)){
                    return false;
                }else{
                    vinList.add(currentVin);
                }
            }
        }
    }
    return true;
    }
```

Finally, 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[] version) {
    carInfoDbService.rollback(version);
    return new Success<>(this);
}
```

Application wallet

The interface *com.horizen.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[] version, List<Box<Proposition>> boxesToUpdate,␣
↪List<byte[]> boxIdsToRemove);
    void onRollback(byte[] version);
    }
```

The Lambo registry example does not implement the interface *ApplicationWallet* because its wallet has basic requirements. You may need to use interface *com.horizen.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.

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 *com.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));
    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;
```

(continues on next page)

(continued from previous page)

```
    }

    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>> regularBoxes = view.getNodeWallet().
↪boxesOfType(RegularBox.class, boxIdsToExclude);
```

(continues on next page)

(continued from previous page)

```
        int index = 0;
        while (amountToPay > 0 && index < regularBoxes.size()) {
            paymentBoxes.add(regularBoxes.get(index));
            amountToPay -= regularBoxes.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<RegularBoxData> regularOutputs = new ArrayList<>();
        if (change > 0) {
            regularOutputs.add(new␣
↪RegularBoxData((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());
        }

        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,
```

(continues on next page)

(continued from previous page)

```
                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 javabean, 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 {
    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>> regularBoxes = view.getNodeWallet().
↪boxesOfType(RegularBox.class, boxIdsToExclude);
int index = 0;
while (amountToPay > 0 && index < regularBoxes.size()) {
    paymentBoxes.add(regularBoxes.get(index));
    amountToPay -= regularBoxes.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<RegularBoxData> regularOutputs = new ArrayList<>();
if (change > 0) {
    regularOutputs.add(new RegularBoxData((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();
```

(continues on next page)

(continued from previous page)

```
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);
```

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`

Find Block by ID

Parameters

Name	Type	Required	Description
blockId	String	yes	Find block by ID

query boolean active return only active versions

query boolean built return only built versions

Example request:

Bash

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

Example response:

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

{
   "result":{
      "blockHex":"string",
      "block":{
         "id":"string",
         "parentId":"string",
         "timestamp":0,
         "mainchainBlocks":[
            {
               "header":{
                  "mainchainHeaderBytes":"string",
                  "version":0,
                  "hashPrevBlock":"string",
                  "hashMerkleRoot":"string",
                  "hashReserved":"string",
                  "hashSCMerkleRootsMap":"string",
                  "time":0,
                  "bits":0,
                  "nonce":"string",
                  "solution":"string"
               },
               "sidechainRelatedAggregatedTransaction":{
                  "id":"string",
                  "fee":0,
                  "timestamp":0,
                  "mc2scTransactionsMerkleRootHash":"string",
                  "newBoxes":[
                     {
                        "id":"string",
                        "proposition":{
                           "publicKey":"string"
```

(continues on next page)

(continued from previous page)


```
                        },
                        "value":0,
                        "nonce":0,
                        "activeFromWithdrawalEpoch":0,
                        "typeId":0
                     }
                  ]
               },
               "merkleRoots":[
                  {
                     "key":"string",
                     "value":"string"
                  }
               ]
            }
         ],
         "sidechainTransactions":[
            {

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


`POST /block/findLastIds`

Returns an array with the ids of the last x blocks

Parameters

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

Example request:

Bash

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

Example response:

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

`POST /block/findIdByHeight`

Return a sidechain block Id by its height in a blockchain

Parameters

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

Example request:

Bash

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

Example response:


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

`POST /block/best`

Return here best sidechain block id and height in active chain

No Parameters

Example request:

Bash

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

Example response:

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


(continues on next page)

(continued from previous page)

```
      "id": "055c15d9a6c9ae299493d241705a2bcfdfbc72a19f04394a26aa53b39f6ee2a6
↪"
    },
    "height": 371
  }
}
```

`POST /block/startForging`

Start forging

No Parameters

Example request:

Bash

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

Example response:

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

`POST /block/stopForging`

Stop forging

No Parameters

Example request:

Bash

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

Example response:

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

POST /block/generate

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

Parameters

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

Example request:

Bash

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

Example response:

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

POST /block/forgingInfo

Get forging info

No Parameters

Example request:

Bash

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

Example response:

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

Sidechain Transaction operations

POST /transaction/allTransactions

Find all transactions in the memory pool

Parameters

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

Example request:

Bash

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

Example response:

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

POST /transaction/findById

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

Parameters

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

Example request:

Bash

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

Example response:

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

POST /transaction/decodeTransactionBytes

Return a JSON representation of a transaction given its byte serialization

Parameters

Name	Type	Required	Description
transactionBytes	String	yes	byte String

Example request:

Bash

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

Example response:

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

POST /transaction/createCoreTransaction

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

Parameters

Example Value

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

(continues on next page)

(continued from previous page)

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

Example request:

Bash

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

Example response:

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

`POST /transaction/createCoreTransactionSimplified`

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

Parameters

Example Value

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

(continues on next page)

(continued from previous page)

```
      "publicKey": "string",
      "blockSignPublicKey": "string",
      "vrfPubKey": "string",
      "value": 0
    }
  ],
  "fee": 0,
  "format": true
}
```

Example request:

Bash

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

Example response:

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

`POST /transaction/sendCoinsToAddress`

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

Parameters

Example Value

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

Example request:

Bash

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

Example response:

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

`POST /transaction/withdrawCoins`

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

Parameters

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

Example request:

Bash

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

Example response:

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

`POST /transaction/makeForgerStake`

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

Parameters

Example Value

```
{
  "outputs": [
    {
      "publicKey": "string",
      "blockSignPublicKey": "string",
```

(continues on next page)

(continued from previous page)

```
      "vrfPubKey": "string",
      "value": 0
    }
  ],
  "fee": 0
}
```

Example request:

Bash

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

Example response:

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

`POST /transaction/spendForgingStake`

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

Parameters

Example Value

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

Example request:

Bash

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

Example response:

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

`POST /transaction/sendTransaction`

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

Parameters

Name	Type	Description
transactionBytes	String	Signed Transaction Bytes

Example request:

Bash

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

Example response:

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

Sidechain Wallet Operations

`POST /wallet/allBoxes`

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

Parameters

Example Value


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


Example request:

Bash

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

Example response:


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

`POST /wallet/balance`

Return the global balance for all types of boxes

Parameters

Name	Type	Required	Description
boxType	String	No	Box type

Example request:

Bash

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

Example response:


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

`POST /wallet/createPrivateKey25519`

Create new secret and return corresponding address (public key)

No Parameters

Example request:

Bash

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

Example response:


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

`POST /wallet/createVrfSecret`

Create new Vrf secret and return corresponding public key

No Parameters

Example request:

Bash

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

Example response:


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


(continues on next page)

(continued from previous page)

```
    }
  }
}
```

`POST /wallet/allPublicKeys`

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

Parameters

Name	Type	Description
protoype	String	

Example request:

Bash

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

Example response:

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

Sidechain node operations

`POST /node/allPeers`

Returns the list of all sidechain node peers

No Parameters

Example request:

Bash

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

Example response:

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

`POST /node/connect`

Send the request to connect to a sidechain node

Parameters

Name	Type	Description
host	String	Node hostname
port	int	Node Port

Example request:

Bash

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

Example response:

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

`POST /node/connectedPeers`

Returns the list of all connected sidechain node peers

No Parameters

Example request:

Bash

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

Example response:


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

`POST /node/blacklistedPeers`

Returns the list of all blacklisted sidechain node peers

No Parameters

Example request:

Bash

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

Example response:


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


Sidechain Mainchain Operations

`POST /mainchain/bestBlockReferenceInfo`

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

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

No Parameters

Example request:

Bash

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

Example response:

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

`POST /mainchain/genesisBlockReferenceInfo`

Reference to Genesis Block

No Parameters

Example request:

Bash

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

Example response:

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

`POST /mainchain/blockReferenceInfoBy`

Parameters

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

Example request:

Bash

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

Example response:

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

`POST /mainchain/blockReferenceByHash`

Reference block by hash

Parameters

Name	Type	Description
hash	String	Block hash
format	boolean	

Example request:

Bash

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

Example response:

```
{
  "result": {
    "blockReference": {
      "header": {
        "mainchainHeaderBytes": "string",
```

(continues on next page)

(continued from previous page)


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

HTTP Routing Table

/block

/mainchain

/node

/transaction

/wallet