V 1.0

Module Specification

Berzeck
NULS.IO

NULS 2.0 Platform – Module Specification

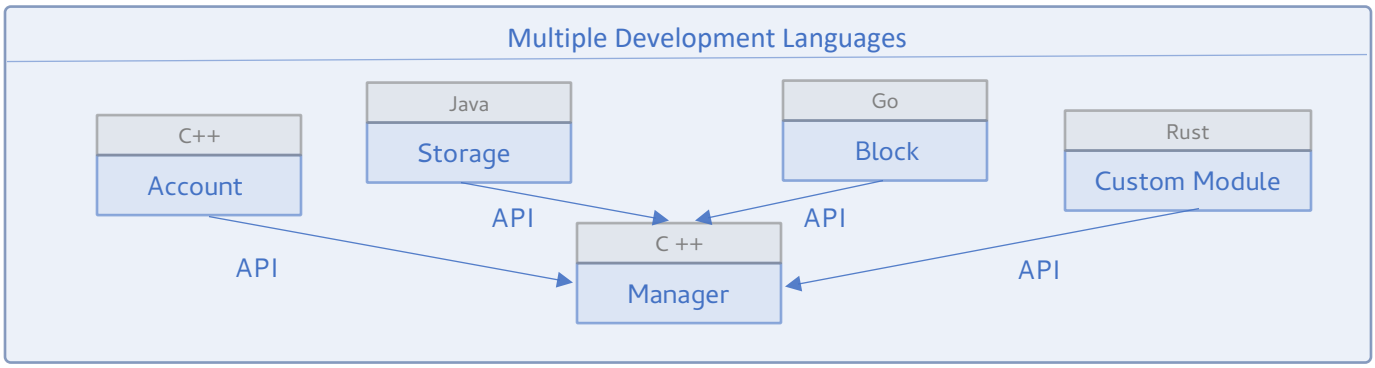
Table Of Contents

1] Design Goals

1.1] Development Flexibility

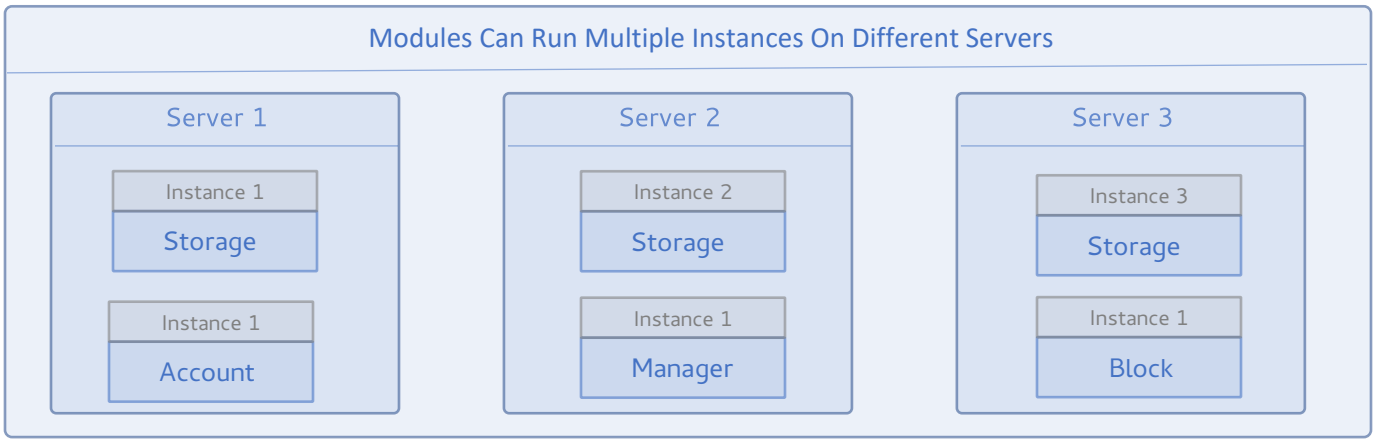
The platform is language agnostic so modules can be developed using different programming languages. The decision is made by developers themselves, and the only requirements are that modules use the same transport mechanism (WebSockets/JSon) and that they send their API to the Manager module.

This will allow developers to focus on a single module instead of learning the entire infrastructure. By focusing on just the functionality they are interested in, the learning curve will be a lot lower, and they can become productive in a shorter time span.



1.2] Deployment Flexibility

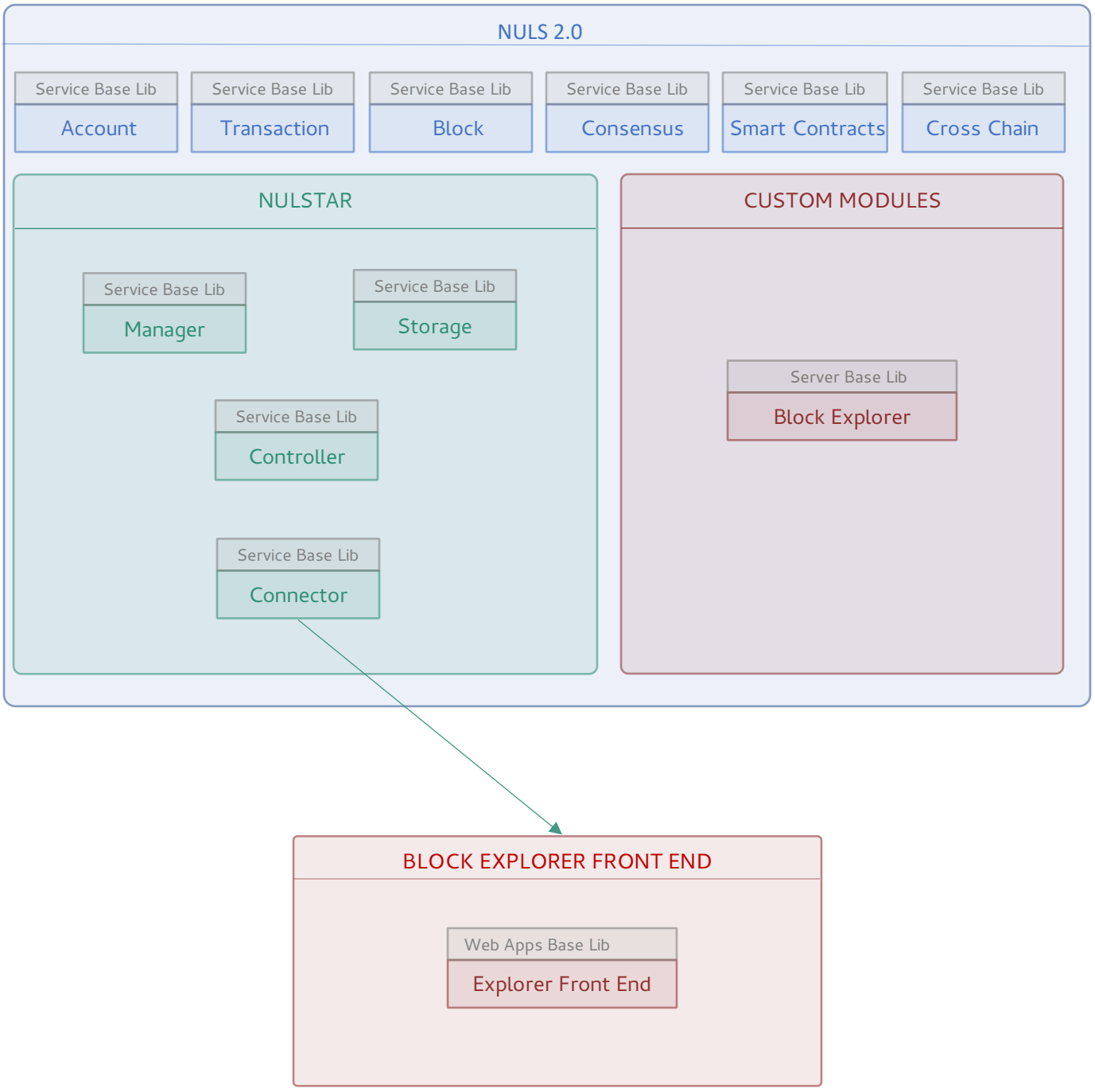
Every module is effectively a separate application, so it is possible to put most of them in a separate environment according to specific needs of each company. For example, if a module in a specific application is performance intensive then it can be deployed in several servers running concurrently.



1.3] Standard framework to build all type of complex systems

The individual components will serve as a base to build all type of systems around NULS like exchanges, social platforms, auction sites, betting sites, explorers, etc. NULS stops being just a blockchain implementation and becomes a development platform on which several complex applications will be built on.

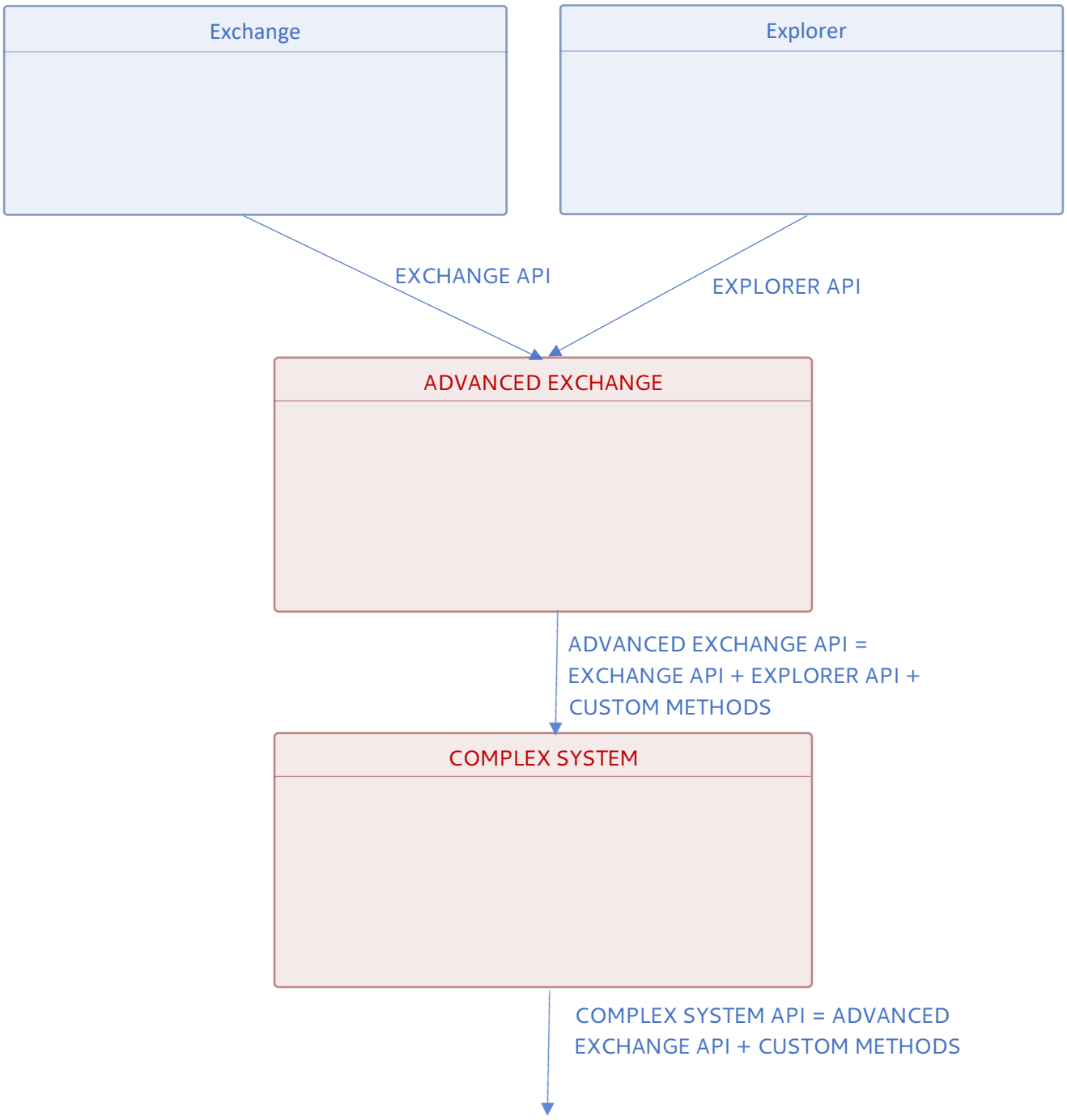
For example, to build a block explorer only one module needs to be attached to the standard modules (Described in section 2) which will handle specific functions and expose the specific API to external applications. If no custom API is required then just the front end needs to be developed.



1.4] Interconnected ecosystem

The systems built around the platform will be able to connect between each other, allowing the ecosystem to improve at lot faster and at lower costs than other cryptocurrencies. This will be possible thanks to sharing the same message protocol and also some simple API conventions.

For example, if an explorer and an exchange exist then it would be a lot easier to build an application that offers functionality of both of these systems, just subscribe to the relevant functions and present the result in a frontend page; also this system could inherit the API of both base systems automatically and add its own functions allowing it to be used as base for other more complex systems and so on.



1.5] Complete module decoupling

The architecture guarantees that there are no hard dependencies between modules, so it is very easy to add new services runtime to the platform. If a module can't find any service that can provide what it needs then it can gracefully report to other components about this situation without crashing the whole application.

1.6] Near zero down time

Due to its architecture, it is a lot easier to provide continuous uptime service. If a module crashes then it can gracefully be deployed again automatically using standard cloud tools. The messages that had to be delivered to it, are just put on hold for few seconds until another instance is deployed. If a module is critical then multiple working units in parallel could be deployed and the Manager would act as load balancer.

Even upgrades could be done without restarting the whole infrastructure because each module can be replaced individually and the upgrades can be done in steps. Also, two versions of the same module could be deployed so that an enterprise can create a rule where it uses the new version just 5% of time to test real dynamics. This is done without risking the whole set of incoming commands, and of course if the upgrade fails then the previous version is restored gracefully and automatically.

1.7] Improved security and quality assurance

As the modules are compact and decoupled pieces of software, code reviews and tests can be done in an efficient manner. Every change just affects the integrity of the module being developed and the rest of infrastructure is not affected. It also allows for the flexibility to give different levels of testing to each module.

For example, some critical modules can be audited by an external party without the need to review all platform. As these services may be executed in different servers, it is more difficult to compromise the whole platform

1.8] Publish - Subscription development pattern

Most applications in crypto world are designed and developed using traditional techniques that are not well suited when internal states are changing constantly and you need to update your application accordingly. For example, when a node is downloading transactions there is a function which returns how many transactions have been processed. This function needs to be polled constantly to update the application, but polling is one of the most inefficient ways to transfer information.

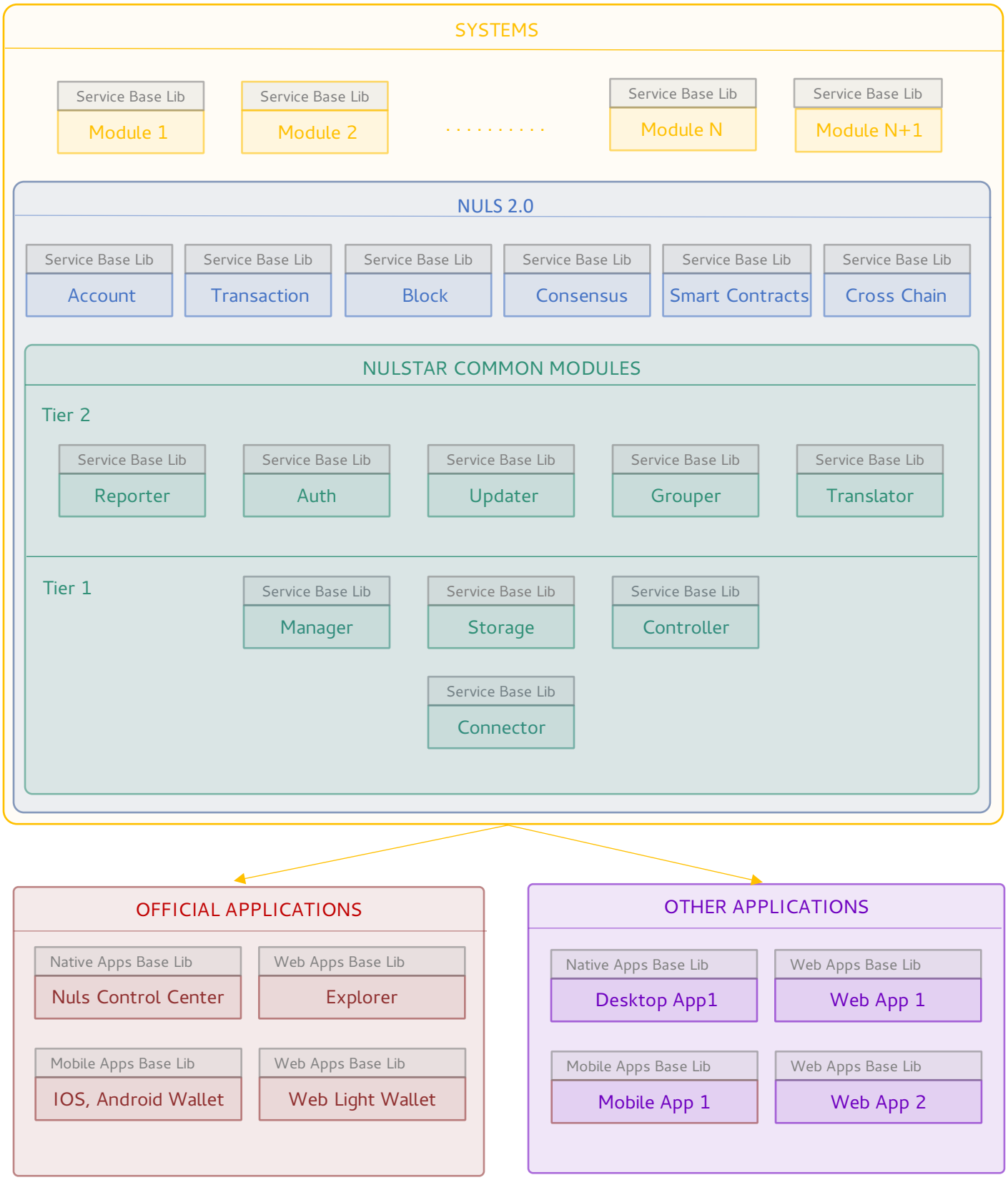
With Publish - Subscription development pattern, this problem could be eliminated altogether because the function will update constantly at specific periods or events defined by the user of the function. When the application no longer needs the information it just 'unsubscribes' from the function.

1.9] Dedicated user interfaces

The user interfaces will be applications completely decoupled from the core architecture, and the only thing binding them is the transport protocol and an API. This is very important because user interfaces have very short development cycles, whereas the core has a long development cycle; multiple developers and designers can work in parallel without the need to coordinate with each other or even with the core development team. This property will allow the development of multiple interfaces to fulfill different needs since companies may need powerful and complex interfaces, while others light and novice-friendly ones, some may prefer web applications while others native ones.

2] Architecture High Level Conceptual Design

FIG 2.1



2.1] Overview

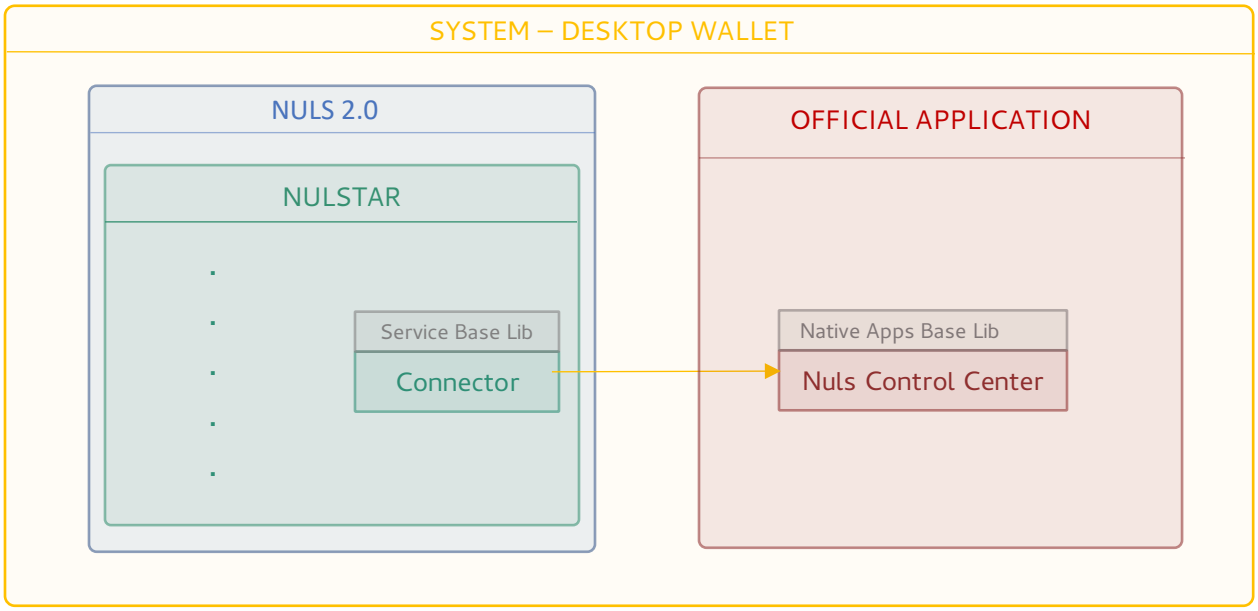
As shown in Fig. 2.1, all kind of complex systems could be built reusing NULS 2.0 components. All these systems will expose their APIs through the Connector module that interchanges messages using Json packets over WebSockets.

A configuration file called Modules.cfg should exist in every server instance where modules are deployed, it contains specific parameters and other directives that modules may require.

In turn, modules can be composed by smaller components developed as plugins; for example, Consensus module can have multiple components, one for each consensus mechanism like PoC, PoS, PoW; specific components should be activated when starting the application using Modules.cfg configuration file.

External applications like GUI, explorer, light wallet and other applications should establish connection to the Connector module to request any information they need to operate normally. Each one of these applications plus NULS 2.0 modules could be grouped in a single package for the convenience of the end user.

For example, the desktop wallet is just NULS 2.0 components plus NULS Control Center clustered in one single convenient package:



2.2] Base libraries

Base libraries (depicted with gray boxes in Fig 2.1) provide the basic standard functionality that allow modules and external applications to integrate to the platform easily and seamlessly. These libraries should be ported to multiple languages to facilitate module and application development for interested parties.

These libraries are:

- Service Base Library:

  All modules should inherit this library, which is in charge of providing the common tools to interact with the rest of the infrastructure seamlessly, like connection routines, sending/receiving Json packages, etc.

- Native Apps Base Library:
  This library allows the development of external native desktop applications including GUI interfaces, abstracting the platform implementation details from the developer.

- Mobile Apps Base Library:
  Designed for mobile and tablet platforms

- Web Apps Base Library:
  Provide the necessary methods to build web frontend to all kind of applications.

2.2] Nulstar

Nulstar is formed by a group of modules that can be used to develop any kind of complex systems including NULS 2.0. All these systems will expose their APIs through the Connector module that interchanges messages using Json format over WebSockets.

Tier 1 consists of the minimum set of modules required to operate the underlying infrastructure.

Manager, Storage, Controller, Connector

Tier 2 represents those modules, which are optional, that provide many common functions that several systems require in a systematic and standard way.

Reporter, Auth, Updater, Grouper, Translator

2.3] NULS 2.0

NULS 2.0 consists of the modules provided by Nulstar Tier 1 plus the specific ones that implements the NULS blockchain protocol, in a later phase Tier 2 modules will also be provided.

The list of NULS 2.0 specific modules is:

Account, Transaction, Block, Consensus, Smart Contracts, Cross Chain

3] Base Libraries Detailed Design

3.1] Service Base Library

Every module that needs to be integrated to the platform should inherit this library. This library serves as the most basic piece of software that developers need to use for developing modules in any language which this library is ported to, as it provides several common functions. Developers may also choose to perform these functions manually.

Its functions are:

- Provide simple basic information about the module.
- Start at least on WebSocket server to receive incoming connections.
- Provide functionality to add more WebSocket servers as required.
- Encode packets to be sent to Json specification.
- Decode and process packets received from Json specification.
- Manage a queue of pending requests awaiting responses.
- Log critical events to hard disk.
- Collect all methods with their respective parameters that will conform the module's API.
- Ability to add meta information each one of the API's methods including a description, parameter validation information and event/period limitations.
- Connect to Manager module and provide the module's API, metainformation and connection information.
- Connect to Storage module for logging purposes.
- Send non-critical events logging information to Storage module.
- Establish and negotiate WebSocket connections to other modules as required using the connection information returned from the Manager module.

3.1.1] Provide simple basic information about the module

Five functions that return strings are responsible for this task:

- fName() .- The name of the module.

- fVersion() .- Version of the module. It is composed by four sections: Major, minor, bug, build. Major number is reserved when the module suffered a massive overhaul or refactoring. Changes in minor version are performed when module offers new features. Bug version changes are given for minor features or for fixing issues. And last, the build version should be increased for a new compilation. The convention used states that if a module increases the major version then it is not backwards compatible so special care must be taken when upgrading, the module is compatible otherwise.

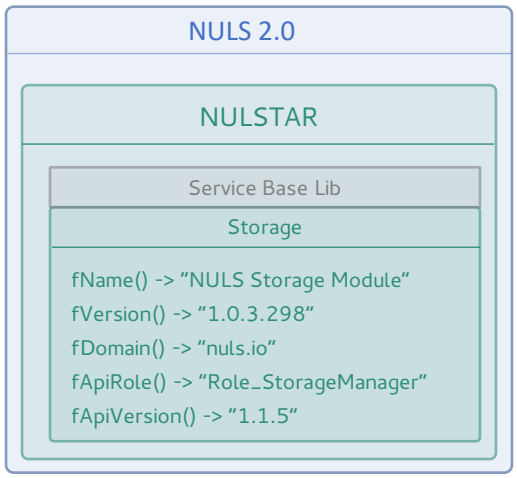
- fDomain() .- In order to avoid clashes with third party module names and Roles, a domain must be given that serves as a prefix, its preferable to be a unique string so an owned web domain name is suggested.

- fApiRole() .- A string the represents the specific Role this module is able to offer. The convention states that the string should start with "Role_", similar Roles must offer similar API's otherwise

Manager module uses the domain to specify which module to use. More than one module can run concurrently offering the same Role, in this case Manager component is used as a load balancer.

- fApiVersion .- API version number comprised of three sections: Major, Minor, Bug. Major number is reserved when the API changes are not backward compatible. Minor version is changed when the API adds new methods or modify existing ones without breaking compatibility. Bug version is used for minor changes, like method description or for changes that are backward and forward compatible with other modules which has different Bug version number but equal Major and Minor version numbers.

  Example:

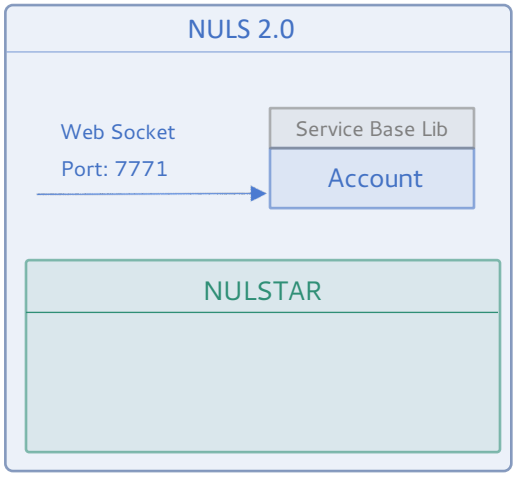


3.1.2] Start at least on WebSocket server to receive incoming connections.

By default a module needs to open a WebSocket server to process incoming connections using the port, the log level and the encryption type specified in the command line parameters when executing it (Refer to section 4), if the port is not available then it must try with the next port number and log this occurrence to Storage module, if this module is not available then it must be logged to hard disk as a critical issue.

The WebSocket port should only receive connection requests by modules inside the architecture, for external applications another WebSocket server should be opened by Connector module to accept external requests.

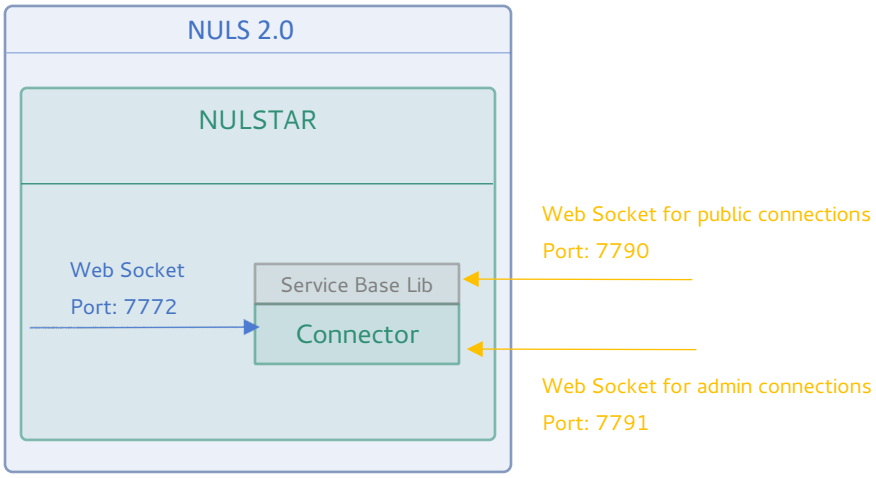
Example: The Account module starts a Web Socket server at startup accepting incoming connections at port 7771.



3.1.3] Provide functionality to add more WebSocket servers as required.

The implementation should have a provision to add more WebSocket servers for special cases controlled by the module.

Example: The Connector module needs to open two more Web Socket servers that accepts incoming connections from external applications, one should be enabled just for privileged users and other for the rest.



3.1.4] Encode packets to be sent to Json specification.

Eight types of Messages are currently defined that must be implemented as classes or structures (depending on implementation language):

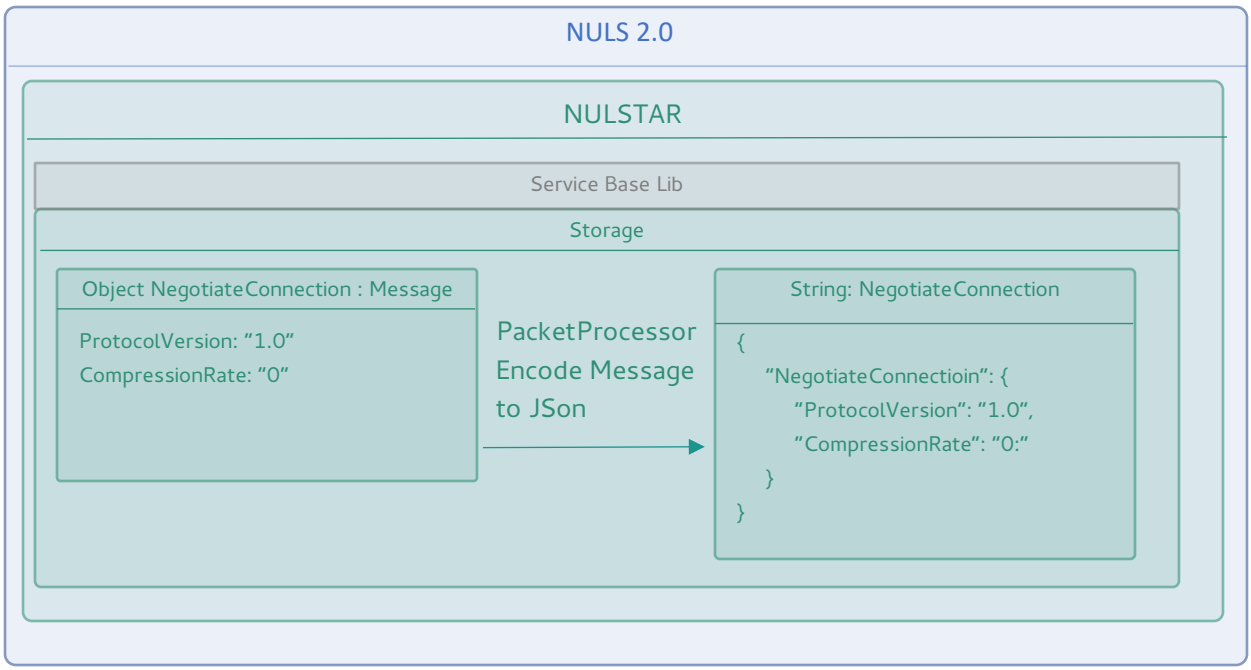
- NegotiateConnection
- NegotiateConnectionResponse
- Request

- Unsubscribe
- Response
- Ack
- RegisterCompoundMethod
- UnregisterCompoundMethod.

Please refer to Nulstar - Documentation - Message Protocol document for specific field details.

When a module needs to send a Message, it must instance the respective class or structure and fill the values with appropriate values, the resulting object must be passed to another called PacketProcessor that is in charge of getting values and encoding them to the selected format (JSon in this case). Several formats should be added in the future for different use cases like XML or binary.

Example: The object that holds NegotiateConnection data is passed to PacketProccesor which encodes the values to Json format



3.1.5] Decode and process packets received from Json specification.

Packets received should be decoded and processed depending on the Message type:

- NegotiateConnection: The module should send back a Message of type NegotiateConnectionResponse, the filed NegotiationStatus should be set to "1" if the protocol is compatible and if it is able to send compressed packets with the specified compression level using **zlib** algorithm, "0" otherwise. This message type could be extended in the future with user authentication mechanism.

- NegotiateConnectionResponse: When the module receives a Message of this type, it should check the NegotiateConnection field, and send a Log Request to Storage module (See section 3.1.12 for further details). If negotiation is successful then the connection must be tagged as such so further requests/responses could be processed, otherwise 2 more tries with 10 second interval must be performed, if these tries fail then it must report to the Manager that the module can't operate normally.

- Request: This message should be decoded and the module must process the methods specified in the RequestMethods array field with their respective parameters. A Response Message should be crafted with the required specific information. If one method fails then the Response should be considered unsuccessful as a whole even if some methods processed successfully. If Event or Period intervals are declared, methods should be processed again resulting in a new Response Message at each Event/Period.

- Response: How to process this message is defined by each module, if no more Responses are expected then PendingMessages queue should eliminate the respective entry (See section 3.1.6 for further details)

- Notification: How to process this message is defined by each module.

- RegisterCompoundMethod: The module should be able to create a new virtual method with the specified name and parameter aliases, then an API Registration Request should be sent to the Manager module to add the virtual function (Check 3.1.10 for information about crafting such request)

- UnregisterRegisterCompoundMethod: The module should be able to remove the specified virtual method, then an API Registration Request should be sent to the Manager module to add the virtual function (Check 3.1.10 for information about crafting such request).

3.1.6] Manage a queue of pending requests awaiting responses

A queue of requests awaiting response should be created to keep track on which Request messages didn't receive a proper response, the queue must record the request id, a reference to the caller object and time of last response (this is the case when the request is sent with Event or Period more than zero so more Response Messages are expected). When no more Response Messages are expected then the appropriate record should be eliminated from the queue.

3.1.7] Log critical events to hard disk

When a module is unable to perform its functions properly and is unable to send a Log request to Storage Module then it should append a Critical error type to a file named Error.log with the following format:

Date|Time|Time Zone|EventLevel|Id|Critical error description

3.1.8] Collect all methods with their respective parameters that will conform the API

The base class should be able to collect the module methods that will conform the API using introspection techniques, each method should be tagged with one of the three specified types:

- Public: The method should be exposed to internal modules and external applications as well.
- Admin: Means that the method should be exposed to internal modules and external applications as well but only through the Admin Web Socket server specified in the Connector module.
- Protected: The method should be exposed only to internal modules.

3.1.9] Ability to add metadata information for each one of the API's methods including a description, parameter validation information and event/period limitations

Optionally modules should send three more types of method metadata:

- Method Description: This provides the description and/or help text of the method.
- Parameter validation: Ranges for numbers and regular expressions for strings could be sent to Manager module so validation checks are performed even before a Request Message is received.
- Event/Period limitations: The minimum Period in seconds and number of Events that the method should send back Response Messages.

3.1.10] Connect to Manager module and provide the module's API, metainformation and connection information

When starting a module, a Request Message invoking RegisterAPI should be sent to Manger module with information about each method and their respective metadata, also a string array with required Roles with their minimum API version should be attached.

Example:

```
{
  "Request": {
    "RequestID": "sdj8jcf8899ekffEFefee",
    "RequestInternalID": "348022847492",
    "RequestDate": "2018-11-05",
    "RequestTime": "03:00:00",
    "RequestTimeZone": "-4",
    "SubscriptionEventCounter": "0",
    "SubscriptionPeriod": "0",
    "ResponseMaxSize: "0",
    "RequestMethods": [
      {
```

```
"RegisterAPI": {
    "ServiceAPIVersion": "0.1.0",
    "ServiceDomain": "nuls.io",
    "ServiceName": "NULS Connector",
    "ServiceRole": "Role_ConnectionManager",
    "ServiceVersion": "0.0.1.1",
    "RoleDependencies": [ { "Role_StorageManager", "0.1.0" } ],
    "ServiceIP": "130.34.32.44",
    "ServicePort": "7775",
    "Methods": [
     {
      "MethodDescription": "Sets the maximum number of client connections that should be accepted.\n
           Parameters:\n  maxconnections [0- ]: Maximum connections allowed. 0 means no limit.",
      "MethodMinEvent": "0",
      "MethodMinPeriod": "0",
      "MethodName": "setmaxconnections",
      "MethodScope": "admin",
      "Parameters": [
       {
        "ParameterName": "lMaxConnections",
        "ParameterType": "int",
        "ParameterRange": "1, 50",
        "ParameterRangeMinIncluded": "1",
        "ParameterRangeMaxIncluded": "1",
        "ParameterRegularExpValidator": ""
       }
      ]
     },
    {
      ......
      ......
      ......
      ......
```

3.1.11] Connect to Storage module for logging purposes

This step should be performed automatically when registering the API if RoleDependencies includes the Storage module Role as a dependency ("Role_StorageManager")

3.1.12] Send non-critical events logging information to Storage module

When a module wants to log an event, a Request of type LogEvent must be sent to Storage module.

There are five levels of log events:

- CriticalError = 1
- ImportantError = 2
- Warning = 3
- Info = 4
- Everything = 5

Example:

```
{
  "Request": {
    "RequestID": "AEsdj8jcf88d3fEFefee",
    "RequestInternalID": "348022847492",
    "RequestDate": "2018-11-05",
    "RequestTime": "03:00:00",
    "RequestTimeZone": "-4",
    "SubscriptionEventCounter": "0",
    "SubscriptionPeriod": "0",
    "ResponseMaxSize: "0",
    "RequestMethods": [
      {
        "LogEvent": {
          "LogEventLevel": "4",
           "LogSourceMessageType": "Request"
          "LogSourceMessageID": ""
          "LogSourceModule": "NULS Connector",
          "LogComments": "Incoming connection accepted from IP: 190.190.1.1!"
        }
      }
    ]
  }
}
```

3.1.13] Establish and negotiate WebSocket connections to other modules as required using the connection information returned from the Manager module

After registering the API, connection information from modules that offer the API Roles needed is received. (See section 4.2 for further details). The module should establish connections with these other modules.

3.2] The Controller – Nulstar executable

This module represents the main executable in the system, therefore the libraries it depends on must be present in the root directory of the application. As it is possible to install modules on other instances or servers, a copy of this module should be present on all these instances so it can manage all modules seamlessly, the other instances should communicate with the main instance so all modules can be controlled from the main instance, the main instance needs to be in the same server where Service Manager module is deployed.

It is advisable for this module to be registered as a service in the host operating system so it can automatically be started.

Its functions are:

- Load parameters from Nulstar.ncf
- Read configuration file from each deployed module
- Start modules passing the correct parameters and environment variables to find libraries
- Offer an API to start and stop modules on demand
- When starting modules, it should compare the hash of the executable and libraries against a list downloaded and signed from official repositories
- Connect to the Controller main instance so it can receive commands

3.2.1] Load parameters from Nulstar.ncf

Nulstar.ncf is the Controller's configuration file where several parameters can be found. The format used is Qt's settings file standard format.

```
[Network]
AllowedNetworks=127.0.0.1/32
CommPort=7770
MainControllerURL=127.0.0.1:7770

[Output]
LogLevel=4

[Security]
SslMode=0
HashFile=
HashFileAlgorithm=SHA256
```

AllowedNetworks: Specifies which networks are allowed to connect to this module using CIDR notation.

CommPort: Which port will this module use to listen for incoming connections.

MainControllerIP: If the main controller is deployed on another server, it's IP must be configured in this parameter, otherwise it must be the loopback address: 127.0.0.1

LogLevel: The type of events that must be logged, there are five levels: 1 Critical, 2 Important, 3 Warning, 4 Information, 5 Everything

SslMode: If this value is zero then no encryption will be used, otherwise a certificate must be issued to every module to open encrypted WebSockets channels.

HashFile: If empty then modules should start without checking hashes; if not, each hash of each file should be compared to the hash list (detailed in3.2.5).

HashFileAlgorithm: The hash algorithm that will be used for the HashFile. SHA256 is the default option.

3.2.2] Read configuration file from each deployed module

Modules are deployed following a simple convention:

Modules/<Namespace>/<Module Name>/<Version>

For example, the Manager module is deployed in the following directory:

Modules/Nulstar/Manager/1.0.0

Nuls 2.0 specific modules should be deployed in the following directory:

Modules/Nuls/<Module Name>/<Version>

In the root directory of each module a file called Module.ncf should exist where is detailed the startup parameters for each module.

For example, for Manager module, the figure below shows its parameter configuration

Module Configuration Example

```
[Core]
Language=CPP
Managed=1

[Libraries]
Nulstar=1.1.3
Qt=5.12.2

[Network]
AllowedNetworks=127.0.0.1/32
CommPort=7771

[Output]
LogLevel=4

[Security]
SslMode=0
```

Language: The development language on which this module was developed. It helps locating the correct libraries to link with the module.

Managed: If this module is managed by the Controller module, is this parameter is false then the module will need to be started and stopped manually

Nulstar: Which version of Nulstar libraries it is using

Qt: Version of Qt libraries the module is linked to

AllowedNetworks: Specifies which networks are allowed to connect to this module using CIDR notation.

CommPort: Which port will this module use to listen for incoming connections.

LogLevel: The type of events that must be logged, there are five levels: 1 Critical, 2 Important, 3 Warning, 4 Information, 5 Everything

SslMode: If this value is zero then no encryption will be used, otherwise a certificate must be issued to every module to open encrypted WebSockets channels.

3.2.3] Start modules passing the correct parameters and environment variables to find libraries

The first module that the Controller needs lo start is the Manager which must be located in the same server instance as the Controller and it must store internally the URL on which the Manager will accept incoming connections.

Manager and other Nulstar modules are located in Modules/Nulstar/ directory.

Libraries' location follows a simple convention:

Libraries/<Language>/<Library Group Name>/<Version>

Examples:

Libraries/CPP/Nulstar/1.0.2/

Libraries/CPP/Nulstar/1.3.1/

Libraries/CPP/Qt/5.12.1/

Libraries/Java/Lib1/1.0.2/

Startup parameters should be sent lowercase using 'getopt' convention (one hyphen for short names and two hyphens for long names).

Startup parameters are grouped in three categories:

- Environment variables: To locate successfully correct versions of libraries; these variables must be operating system specific. Information on how to build the correct PATHs must be extracted from [Core] and [Libraries] of Module.ncf file of each module (detailed in 3.2.1])

- Manager URL: Must be sent to modules so they can register their API and connect to dependences.

- Other specific parameters: The rest of parameters located in Modules.ncf must also be sent on startup.

Example: Executing Manager module in Linux

LD_LIBRARY_PATH=$(pwd)/Libraries/CPP/Qt/5.12.1/;$(pwd)/Libraries/CPP/Nulstar/1.0.2/ $(pwd)/Modules/Nulstar/Manager --managerurl 127.0.0.1:7771 --loglevel 4 --sslmode 0 --commport 7771 --allowednetworks 127.0.0.1/32

3.2.4] Offer an API to start and stop modules on demand

The Controller must register its API as a normal module and must offer four basic functions that should be exposed using admin privileges.

- startallmodules
- startmodule <Module Name>
- stopallmodules
- stopmodule <Module Name>
- registermodules

3.2.5] When starting modules, it should compare the hash of the executable and libraries against a list downloaded and signed from official repositories

In order to improve security, as explained in 3.2.1] there is an option to check hashes of each module and library before the Controller starts up the corresponding modules, below an example of what HashFile looks like:

HashFile Example

```
[Libraries]
NNetwork_CPP_1.1.0=2ccde07e9eaadc1b06a2fcd276a2364baf04772d5c0567d8a3d333cb3082b6d1
NNetwork_CPP_1.2.0=99196929374e067a05b4f97f3e61beb4d987e5c8f4546119a036f6d86e2e929a
NNCore_Java_1.3.1=36a7da47692860955aed337b0dd56e4a134f9a08711785d5fd8a67bf50da2092

[Modules]
Nulstar/Manager=b9f2c2bb05237a710856625483c51a712ced4fc10c0c13b3d432a815db2f6a26
Nulstar/Connector=391ff8a395bad8b267ab9604b2827507a140ab940b7f905a3bc8b6465be05135
Nuls/Block=ff6956640981b7a22bcb13beb3d37d7d9a70551657fd75c7a3a3c67b897197b5

[Security]
Hash=9d19be92a55355e5bcd2bb3fb760a7bc94404c45623e4ef2aca1a07453ab4177
```

The Hash parameter is obtained concatenating all modules, libraries and a key that is embedded in Nulstar executable, this is made to assure this file has not been edited manually.

The Controller needs to hash files before executing them and if hashes differ this must be logged as a Critical event and module should not start.

3.2.6] Connect to the Controller main instance so it can receive commands

When deploying modules on different servers, instances of Controller must be deployed too so modules can be managed seamlessly.

Copies of Controller do not need to register their API to the Manager as the main instance is responsible for this but they must send the list of modules and associated versions to the main controller therefore when connecting to it, they must send a Request message with the respective information.

```
{
  ……
  ”MessageType”: ”Request”,
  ”MessageData”: {
       ”RequestAck”: ”0”,
       ”SubscriptionEventCounter”: ”0”,
       ”SubscriptionPeriod”: ”0”,
       ”SubscriptionRange”: ”0”,
       ”ResponseMaxSize: ”0”,
       ”RequestMethods”: {
            ”RegisterModules”: {
                 ”Block”: ”1.3.4”,
                 ”Transaction”: ”2.3.4”,
            },
        }
   }
}
```