inanalytics.io

last modification: **2026/04/20**

InAnalytics Data Platform

Real-time analytics, reporting, simulations, forecasting and data automation

Turn your industrial and business data

into cost savings, operational control

and deep visibility across your processes

InDriver Manual

InAnalytics Data Platform 1
Real-time analytics, reporting, simulations, forecasting and data automation 1
Turn your industrial and business data 1
into cost savings, operational control 1
and deep visibility across your processes 1
Innovative Data Analytics 9
Introduction to InDriver 10
Getting started - download, installation, requirements 10
Installation 11
Launching InDriver 12
License 12
Setup 16
InDriver first steps 21
Features and examples 24
InDriver JS Tasks 25
Recommended coding rules (AI) 28

InDriver API **35**
InDriver.configuration 35
InDriver.currentPath 36
InDriver.debug 37
InDriver.debugVariables 39
InDriver.driverName 39
InDriver.getCpuUsage 40
InDriver.hooks 41
InDriver.hookTs 41
InDriver.import 42
InDriver.installExtensions 44
InDriver.installHook 46
InDriver.isHook 47
InDriver.isPrivateMessage 47
InDriver.loadScript 48
InDriver.messageData 48
InDriver.messageSender 49
InDriver.messageTags 49
InDriver.messageTs 50
InDriver.msleep 51
InDriver.onHook 51
InDriver.restartTask 52
InDriver.sendMessage 53
InDriver.setFlag 54
InDriver.shutdown 55
InDriver.sleep 56
InDriver.sqlExecute 56
InDriver.sqlExecuteAll 59
InDriver.sqlIsSucceeded 60
InDriver.sqlLastError 60
InDriver.sqlReopen 61
InDriver.taskName 62
InDriver.uninstallHook 62
InDriver.updateStatistics 63
HttpServerApi **65**
HttpServerApi.debugMode 65
HttpServerApi.listen 66
address 66
port 67
HttpServerApi.route 68
HttpServerApi.setCorsHeaders 72
CryptographicHashAPI **75**

Supported algorithms 75
CryptographicHashApi.addData 76
CryptographicHashApi.addDataUtf8 77
CryptographicHashApi.algorithm 77
CryptographicHashApi.algorithms 78
CryptographicHashApi.hash 79
CryptographicHashApi.hashHex 79
CryptographicHashApi.hashHexUtf8 80
CryptographicHashApi.hashUtf8 81
CryptographicHashApi.isSupported 82
CryptographicHashApi.reset 82
CryptographicHashApi.result 83
CryptographicHashApi.resultHex 84
CryptographicHashApi.setAlgorithm 84
OPCUPClientApi 87
OPCUAClientApi.browse 87
OPCUAClientApi.connect 89
OPCUAClientApi.disconnect 91
OPCUAClientApi.isConnected 92
OPCUAClientApi.lastError 93
OPCUAClientApi.readMultipleNodes 93
OPCUAClientApi.runIterate 95
OPCUAClientApi.writeMultipleNodes 96
OPCUPServerApi 99
OPCUAServerApi.addNodes 99
OPCUAServerApi.lastError 100
OPCUAServerApi.readMultipleNodes 101
OPCUAServerApi.start 102
OPCUAServerApi.stop 104
OPCUAServerApi.writeMultipleNodes 104
RestApi 107
RestApi.begin 107
RestApi.commit 108
RestApi.commitWait 109
RestApi.defineRequest 110
RestApi.getData 113
RestApi.getRawHeader 113
RestApi.getRawHeaderPairs 114
RestApi.isSucceeded 115
RestApi.sendRequest 116
RestApi.wait 119
ModbusApi 124
ModbusApi.begin 124

ModbusApi.commit 125
ModbusApi.commitWait 126
ModbusApi.connectDevice 126
ModbusApi.getAllData 129
ModbusApi.getDeviceData 130
ModbusApi.getDeviceRequestData 131
ModbusApi.getDeviceRequestValue 131
ModbusApi.isLastTransactionCompleted 133
ModbusApi.isSucceeded 134
ModbusApi.readDevice 134
ModbusApi.wait 136
ModbusApi.writeDevice 137
DeviceAPI 141
DeviceAPI.begin 141
DeviceAPI.commit 142
DeviceAPI.commitWait 142
DeviceAPI.debugModeEnabled 143
DeviceAPI.execute 144
DeviceAPI.getDeviceData 145
DeviceAPI.getStatistics 146
DeviceAPI.waitForDevices 146
MbusAPI 148
MbusAPI.exec 148
MbusAPI.receive 150
ModbusAPI 151
ModbusAPI.exec 151
ModbusAPI.receive 152
Device libraries 153
AR252 library 154
CMK03 library 155
Multical Modbus library 155
MacREJ5R library 156
M-Bus device libraries 157
SMTPApi 162
SmtpApi.begin 163
SmtpApi.commit 163
SmtpApi.configure 164
SmtpApi.send 166
SmtpApi.lastError 168
TsApi 171
Standard TSAPI database model 171
tsapiinstall 174

tsapiuninstall 174
tsapiversion 175
tsapicreatetable 176
tsapicreateaggregationtable 177
tsapideletetable 177
tsapiinsert 178
tsapiinsertvalues 179
tsapiinsertagg 181
tsapiselectdistinctsources 182
tsapiselect 183
tsapiselectfirst 185
tsapiselectlast 185
tsapiselectlastn 186
tsapiselectprevious 188
tsapiselectagg 189
tsapiselectaggdata 190
tsapiselectaggwheretsin 191
tsapiselectvariablefromagg 193
Time Series JSON Data Aggregation 196
TsApi.aggregate 197
TsApi.defineAggregator 198
TsApi.setAggregatorDebugMode 202
TsApi.setAggregatorStepSize 203
ProcessApi 207
ProcessApi.start 207
ProcessApi.close 208
ProcessApi.closeAll 209
ProcessApi.kill 209
ProcessApi.killAll 209
ProcessApi.waitForStarted 210
ProcessApi.waitForFinished 210
ProcessApi.waitAllForStarted 211
ProcessApi.waitAllForFinished 211
ProcessApi.setWorkingDirectory 211
ProcessApi.workingDirectory 212
ProcessApi.program 212
ProcessApi.pid 212
ProcessApi.state 213
ProcessApi.status 213
ProcessApi.remove 214
PdfApi 216
PdfApi.load 216
PdfApi.isLoaded 217

PdfApi.pageText	218
PdfApi.readText	219
PdfApi.pageLabel	220
PdfApi.allPageLabels	221
PdfApi.setCodec	221
PdfApi.pageCount	222
PdfApi.setPassword	223
PdfApi.password	223
PdfApi.status	224
PdfApi.metadata	225
PdfApi.asImageBase64	225
PdfApi.savePageAsImage	226
PdfApi.close	227
FileApi	**230**
FileApi.addFileSystemWatcherPath	230
FileApi.appendLine	230
FileApi.close	231
FileApi.closeAll	231
FileApi.copy	232
FileApi.dirGoTo	232
FileApi.dirGoToApplicationDirectory	232
FileApi.dirGoToPath	233
FileApi.dirGoUp	233
FileApi.dirList	233
FileApi.dirLoadFiles	234
FileApi.dirSaveFiles	234
FileApi.fileExists	234
FileApi.fileSize	235
FileApi.isOpen	235
FileApi.open	235
FileApi.readAll	236
FileApi.readAllBase64	236
FileApi.readLines	236
FileApi.removeFile	237
FileApi.removeFileSystemWatcherPath	237
FileApi.write	237
SerialPortApi	**240**
SerialPortApi.availablePorts	240
SerialPortApi.close	240
SerialPortApi.closeAll	241
SerialPortApi.flush	241
SerialPortApi.isOpen	241

SerialPortApi.open 242
SerialPortApi.read 243
SerialPortApi.readAll 243
SerialPortApi.readLine 243
SerialPortApi.setTimeout 244
SerialPortApi.write 244
TcpSocketApi 247
TcpSocketApi.acceptRead 247
TcpSocketApi.connect 248
TcpSocketApi.disconnect 250
TcpSocketApi.disconnectAll 250
TcpSocketApi.isBusy 251
TcpSocketApi.write 252
TcpSocketApi.writeAndWait 253
TcpSocketApi.onMessage behavior 255
TcpServerApi 257
TcpSocketApi.close 257
TcpServerApi.listen 259
TcpServerApi.write 260
TcpServerApi.onMessage behavior 262
UdpSocketApi 265
UdpSocketApi.acceptRead 265
UdpSocketApi.bind 266
UdpSocketApi.isBusy 267
UdpSocketApi.write 268
UdpSocketApi.writeAndWait 270
XmlReaderApi 273
XmlReaderApi.addData 273
XmlReaderApi.addExtraNamespaceDeclaration 274
XmlReaderApi.atEnd 274
XmlReaderApi.attribute 275
XmlReaderApi.characterOffset 275
XmlReaderApi.clear 276
XmlReaderApi.close 276
XmlReaderApi.columnNumber 277
XmlReaderApi.documentEncoding 277
XmlReaderApi.documentVersion 277
XmlReaderApi.dtdName 278
XmlReaderApi.dtdPublicId 278
XmlReaderApi.dtdSystemId 278
XmlReaderApi.entityDeclarations 279
XmlReaderApi.entityExpansionLimit 279

XmlReaderApi.hasError 280
XmlReaderApi.hasStandaloneDeclaration 280
XmlReaderApi.isCDATA 281
XmlReaderApi.isCharacters 281
XmlReaderApi.isComment 281
XmlReaderApi.isDTD 282
XmlReaderApi.isEndDocument 282
XmlReaderApi.isEndElement 282
XmlReaderApi.isEntityReference 283
XmlReaderApi.isProcessingInstruction 283
XmlReaderApi.isStandaloneDocument 284
XmlReaderApi.isStartDocument 284
XmlReaderApi.isStartElement 284
XmlReaderApi.isWhitespace 285
XmlReaderApi.lastError 285
XmlReaderApi.lineNumber 285
XmlReaderApi.name 286
XmlReaderApi.namespaceProcessing 286
XmlReaderApi.namespaceUri 287
XmlReaderApi.open 287
XmlReaderApi.prefix 288
XmlReaderApi.processingInstructionData 288
XmlReaderApi.processingInstructionTarget 289
XmlReaderApi.qualifiedName 289
XmlReaderApi.readElementText 289
XmlReaderApi.readNext 290
XmlReaderApi.readNextStartElement 291
XmlReaderApi.setEntityExpansionLimit 291
XmlReaderApi.setNamespaceProcessing 292
XmlReaderApi.skipCurrentElement 293
XmlReaderApi.text 293
XmlReaderApi.tokenString 293
XmlReaderApi.tokenType 294

Innovative Data Analytics

www.inanalytics.io

Innovative Data Analytics is a recently established IT company located in Kraków, Poland, founded by programmer and automation engineer Andrzej Jarosz.

Leveraging his experience from his previous project, ANT Solutions, established in 2006, and maintaining a leading position in MES (Manufacturing Execution Systems), Andrzej has embarked on a new challenge with analytics.io.

The core idea driving Andrzej is to provide a smart data automation engine for engineers, developers, data analysts, and more, to build powerful systems using just a few lines of simple code.

The project is dedicated to both non-commercial users, who can use the software completely free, and commercial projects.

contact@inanalytics.io
www.linkedin.com/in/andrzej-jarosz-6b6ba96

Introduction to InDriver

InDriver is a versatile automation engine that executes multiple JavaScript tasks simultaneously, **facilitating easy solution creation within minutes, even without programming experience.**

Utilize the specialized API to support technologies such as REST API, SQL, Modbus, TCP Server, Socket, Serial Port, Files, JSON, and more, to create seamless custom data integration in just a few lines of code.

Distinguished from typical SaaS, **InDriver is an application** for straightforward installation on local machines or in the cloud, **providing unlimited data processing capabilities without cost constraints**.

InStudio allows remote configuration of InDrivers across multiple computers, enabling distributed solutions, with numerous copy-paste examples for quick integration.

Getting started - download, installation, requirements

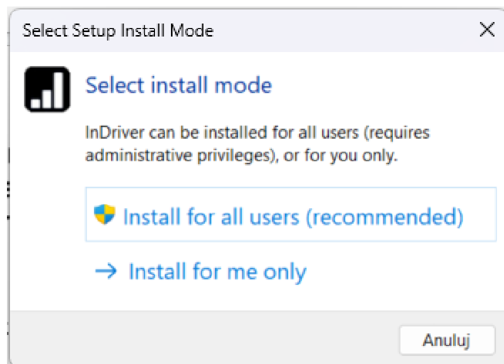
InDriver can be freely downloaded from https://www.inanalytics.io/downloads.
InDriver is absolutely free for non-commercial use.
For commercial use, users are obligated to purchase an annual plan.

InDriver is provided as a zip archive containing the InSetup.exe installer, which installs both InDriver and InStudio.
Currently, there is an available version for **Windows operating systems**, such as:
Windows 10 and **Windows Server 2016 and later**.

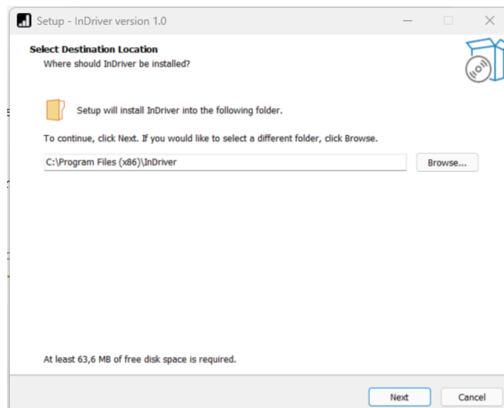
Installation

Unpack **InSetup.zip** and run **InSetup.exe**

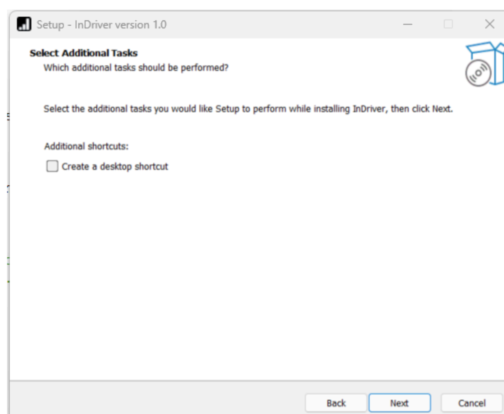
1. Install for all users



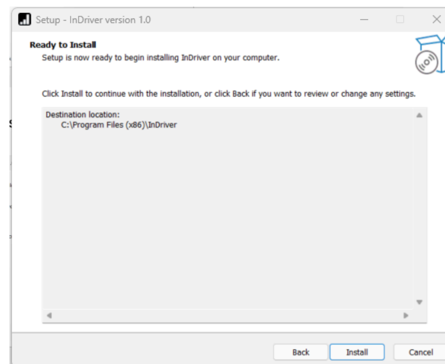
2. Select folder



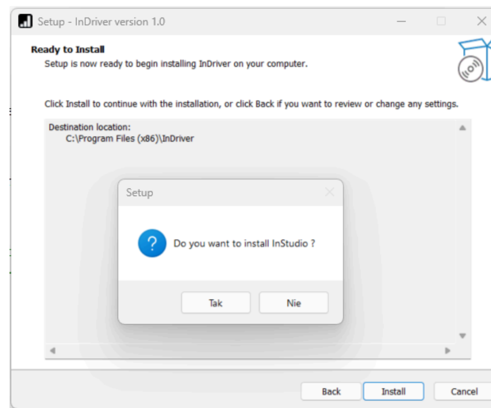
3. Create a desktop shortcut



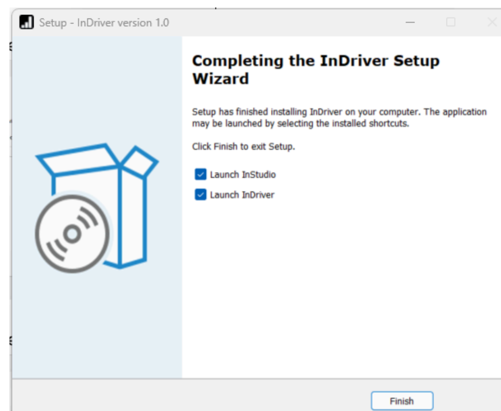
4. Confirm



5. Select to install InStudio



6. Installation complete



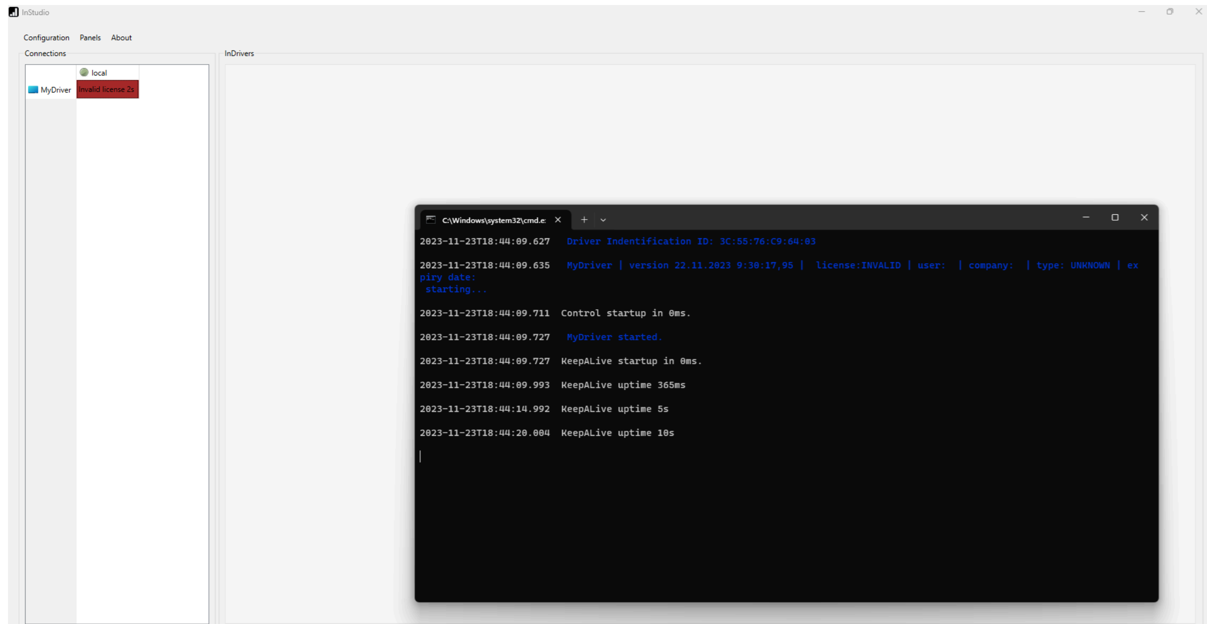
Launch InStudio and InDriver.

Innovative
Data
Analytics

inanalytics.io

Launching InDriver

After the installation is completed, and both InDriver and InStudio are launched, InDriver starts as a console application, while InStudio starts as a window application, as shown in the screenshot below:



To start InDriver, run either indriver.exe or the recommended indriver.bat script.
The indriver.bat script includes a loop that automatically restarts indriver.exe if it crashes unexpectedly, improving stability during execution.

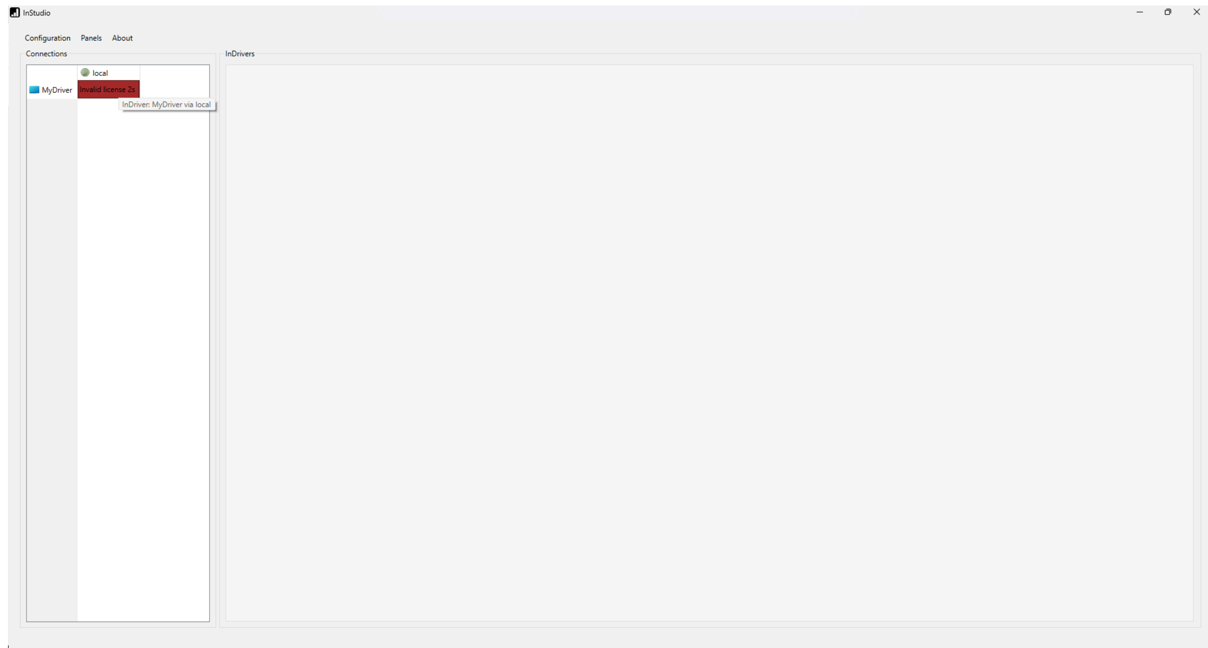
InDriver also accepts several command-line arguments:

- **–mode [batch]**
  When running in **batch mode**, the InDriver process can be terminated programmatically using the InDriver.shutdown() API function. This mode is intended for scenarios where InDriver executes a script and then exits automatically upon completion.
- **–dir [path]**
  Specifies the **working directory** containing the driver.cfg configuration file.Launch

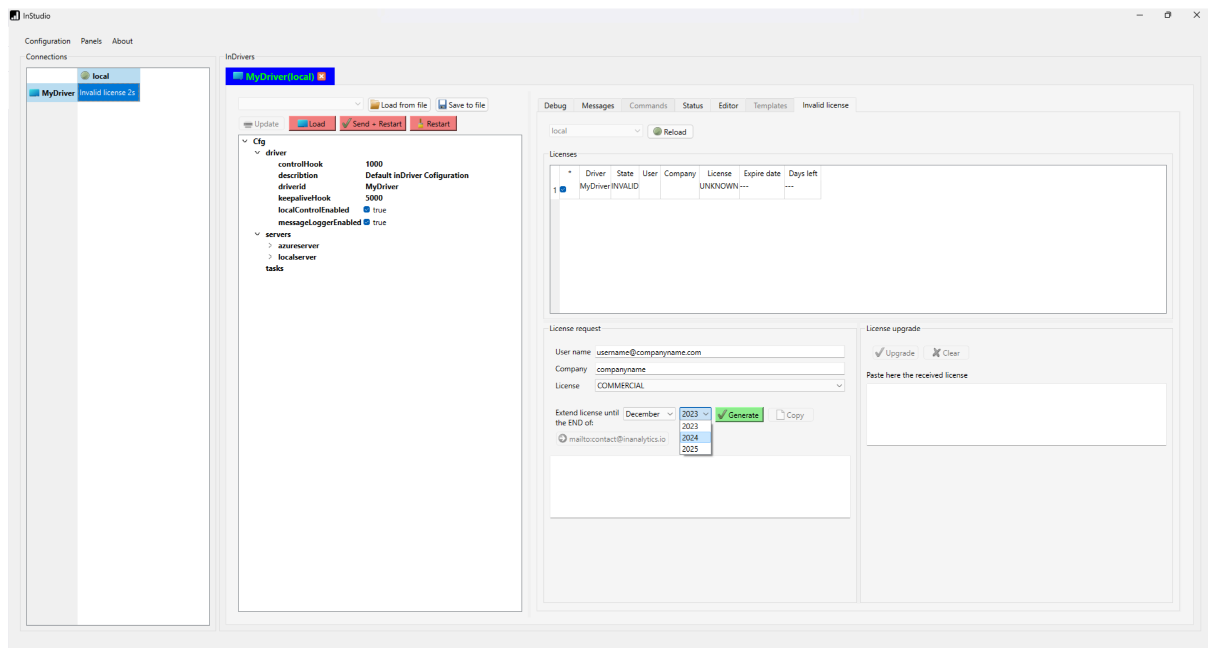
License

The system requires a license, even for non-commercial use, when a free license is provided. To set up the license, please follow these steps:

1. Press MyDriver in the Connections table.



2. Fill out the License Request Form, providing the following details: User name/email address, Company, License type (Commercial/Free), and license period. Commercial licenses can be purchased for one, two, or three years with one-month accuracy. Free licenses can be obtained for a fixed one-year period.



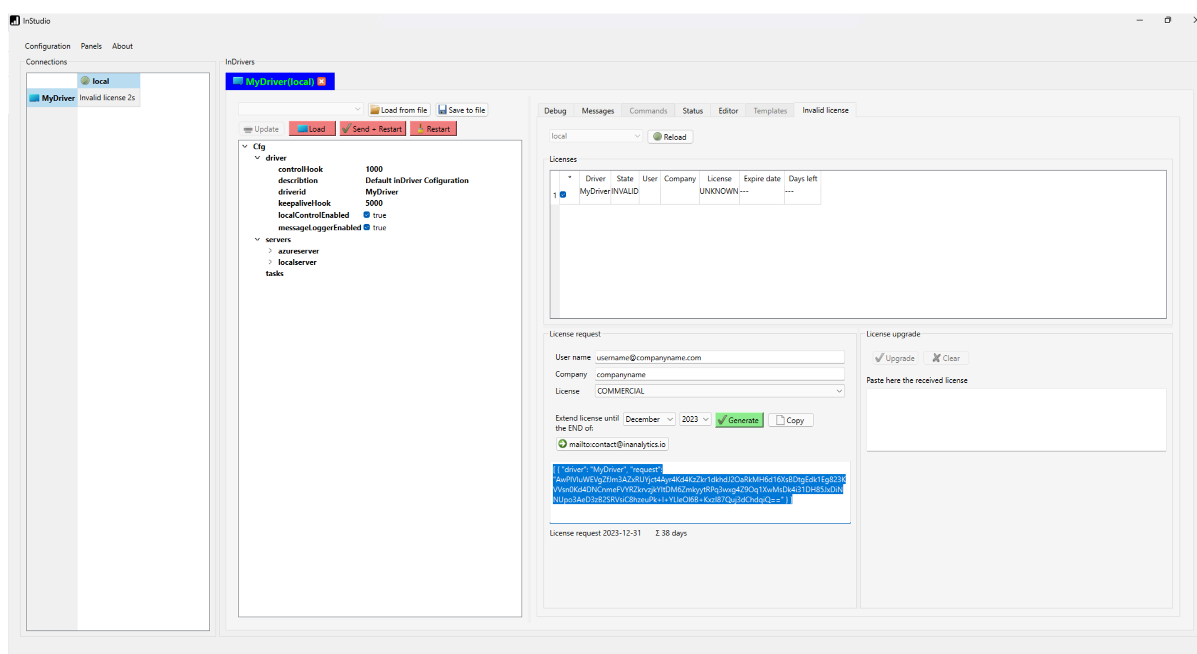
3. Press 'Generate,' copy the generated license request, and send it to contact@inanalytics.io. Alternatively, you can click the 'mailto:contact@inalaytic.io' button, and your default mail app will open.

To expedite the license request, please provide us with your organization's data and describe your project. If you are requesting a free license, please explain why your project is non-commercial.

We prioritize all license requests with maximum urgency and commit to responding within 24 hours.

Remark:

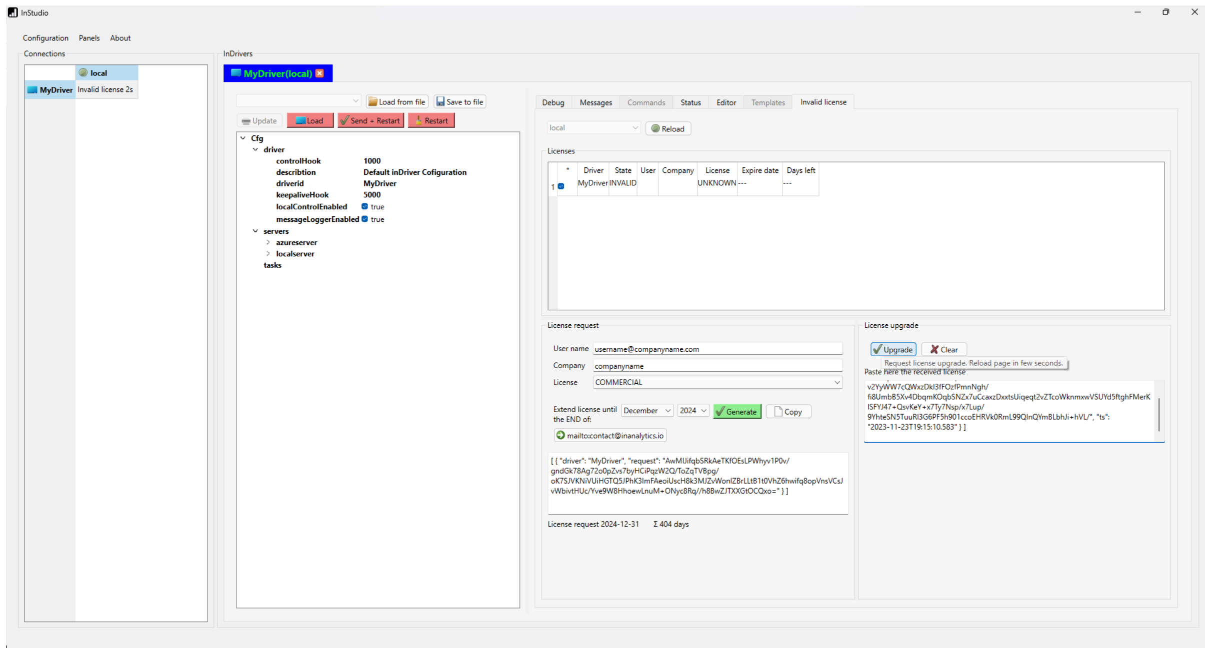
- For Free licenses, Innovative Data Analytics may request confirmation if the project being realized with a free license is genuinely non-commercial. You may be asked to fill out a form with details related to your organization and project. This procedure is applicable every time the license is granted or prolonged.
- For commercial requests, the license will be granted after the purchase is finalized.



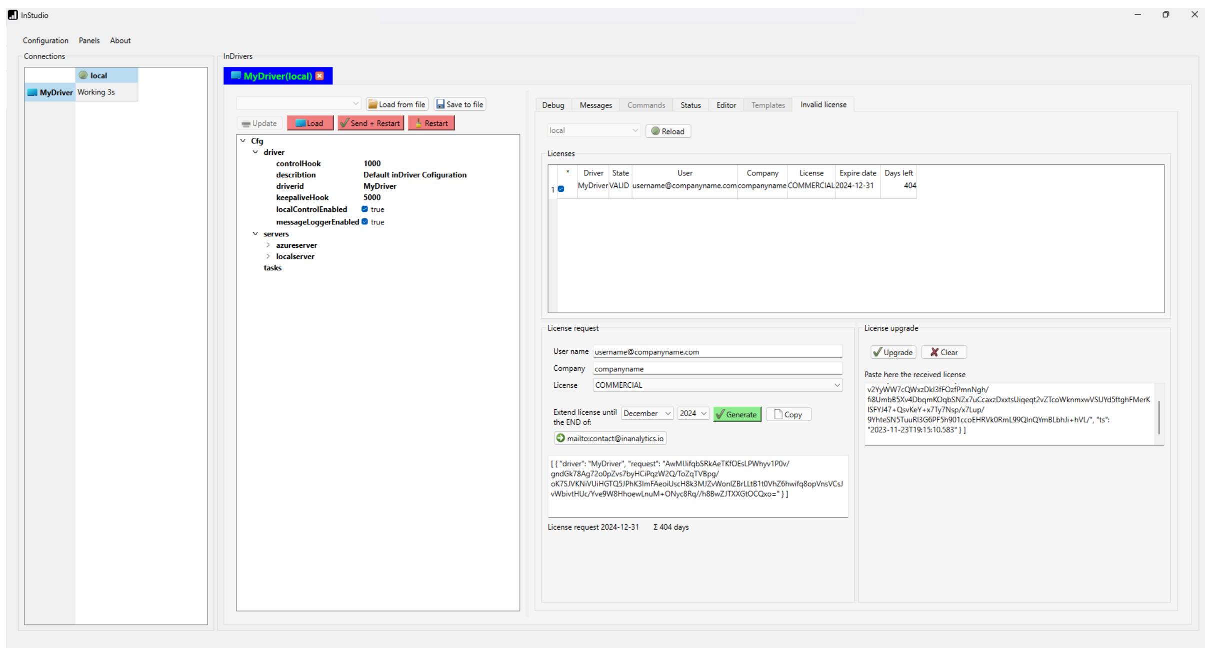
Innovative
Data
Analytics

inanalytics.io

4. When you receive the license key, please copy it into the 'License Upgrade' text editor, then press the 'Upgrade' button.



5. Wait a few seconds. In the 'Connection' table on the left, 'MyDriver' should switch from 'Invalid license' to 'Working'. You can also press the 'Reload' button. After InDriver restarts, you should see your license.



Setup

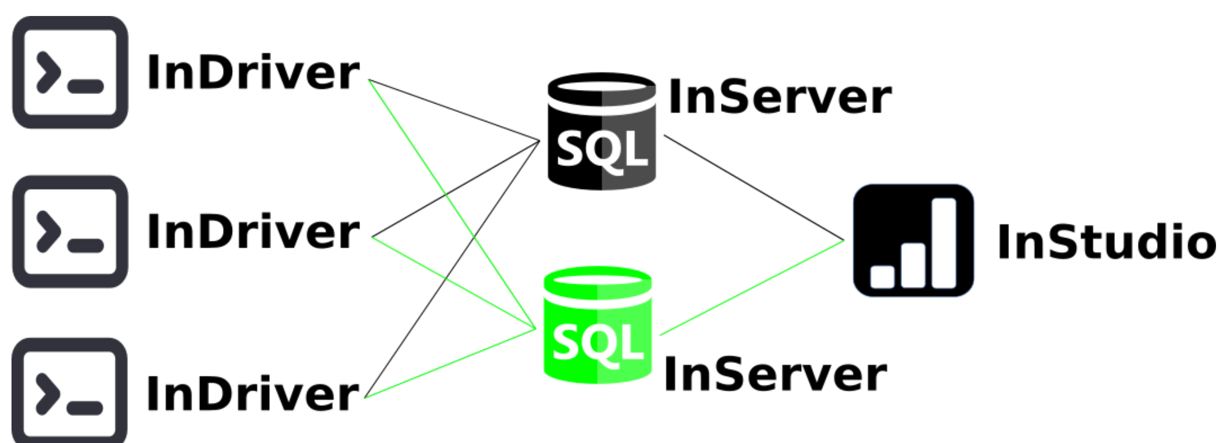
InStudio Setup allows you to configure SQL Servers, and user data, install InServer, and manage other APIs included in the InDriver package, as well as manage licenses.

InServer

InStudio communicates with InDriver, enabling their configuration, management, and programming of tasks using either a local machine connection or a database connection.

A local machine connection provides direct access to InDriver installed on the same machine. On the other hand, a database connection not only facilitates communication on the local machine but also enables the construction of a distributed system. In a distributed system, one or more InDrivers may run on various machines and be managed from a single InStudio.

Database communication becomes particularly valuable for system redundancy when configuring more than one database server. The schema below illustrates the database connection between InDrivers and InStudio.



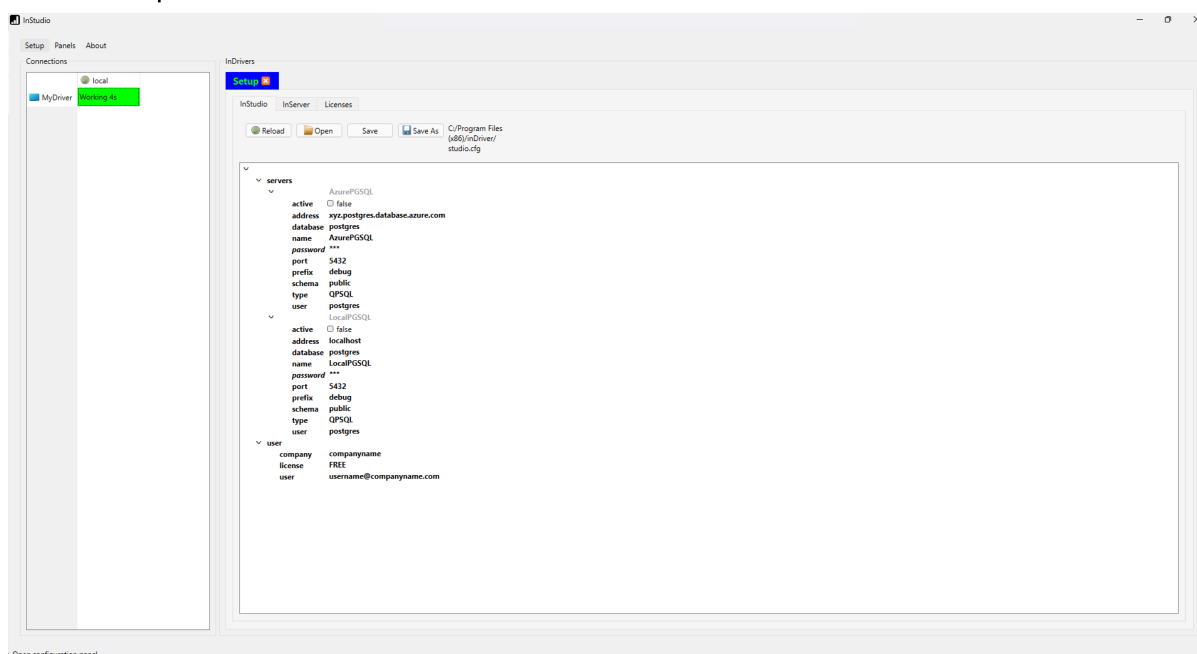
One or more SQL Servers can serve as gateways by installing InServerAPI on them, a process easily facilitated through InStudio.

Configure SQL Servers

To configure SQL servers, open the 'Setup' menu and navigate to the 'InStudio' tab. By default, two servers are provided. Fill in the correct details:

- 'Address': SQL database address
- 'Database': Database name
- 'Password': Password for the specified user (It is safe - the system uses encryption to store passwords).
- 'User': Database user name
- 'Type': Database type (choose from the following options):
    - QDB2: IBM DB2 (version 7.1 and above)
    - QIBASE: Borland InterBase / Firebird
    - QMYSQL / MARIADB: MySQL or MariaDB (version 5.6 and above)
    - QOCI: Oracle Call Interface Driver (version 12.1 and above)
    - QODBC: Open Database Connectivity (ODBC) - Microsoft SQL Server and other ODBC-compliant databases
    - QPSQL: PostgreSQL (versions 7.3 and above)
    - QSQLITE: SQLite version 3
    - QMIMER: Mimer SQL (version 11 and above)

Enter the 'Name' for the configured server in InStudio, remember to set 'Active' to true, and press the 'Save' button.

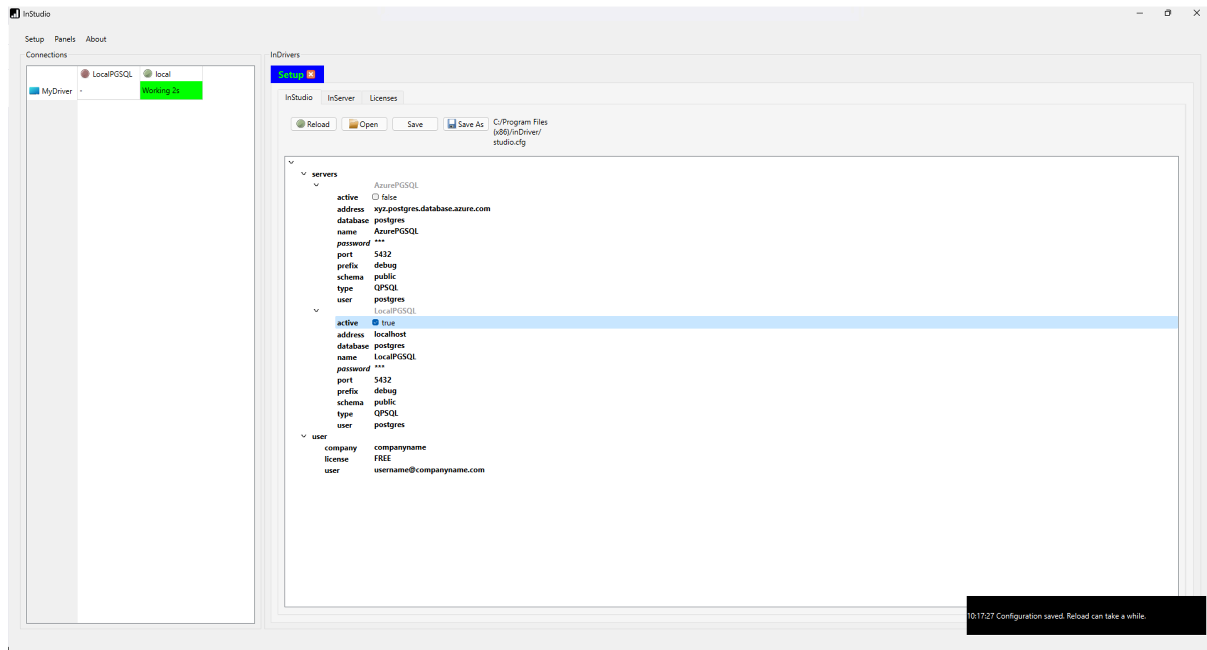


If valid data is provided, after a few seconds, the connection table on the left should display configured servers in the horizontal header, indicated in green for successful connections or red if the database connection failed.

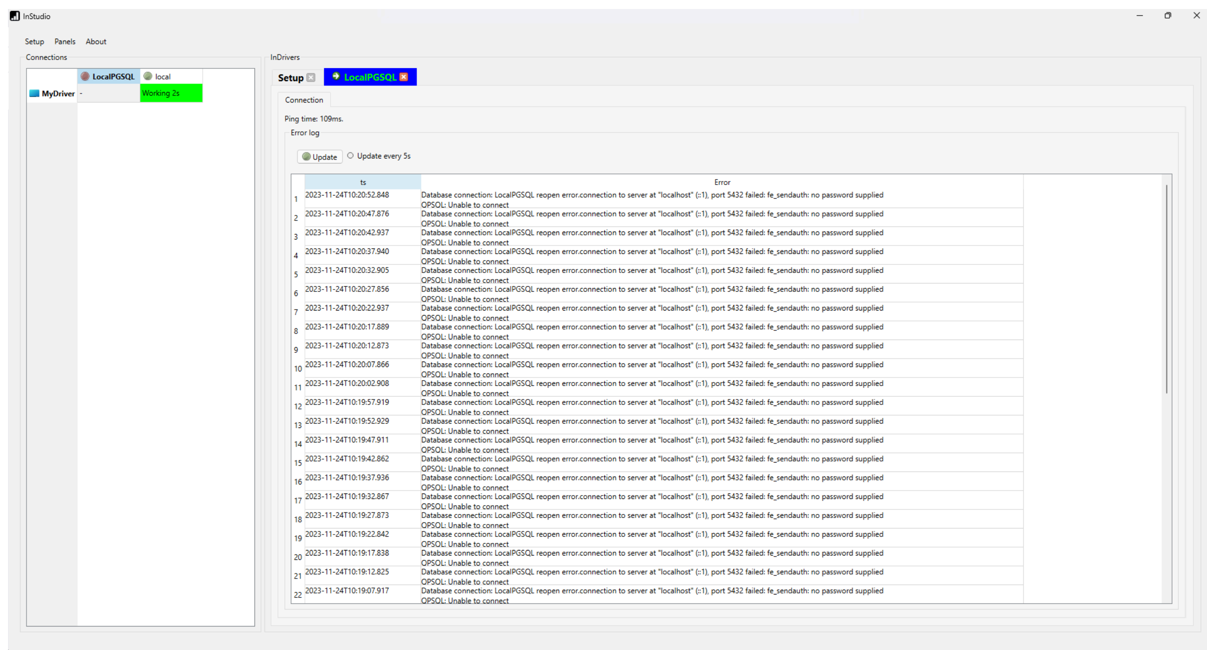
Innovative Data Analytics

inanalytics.io

The screenshot below illustrates a situation where the LocalPGSQL connection has failed, as indicated by the red status in the Connections table on the left.



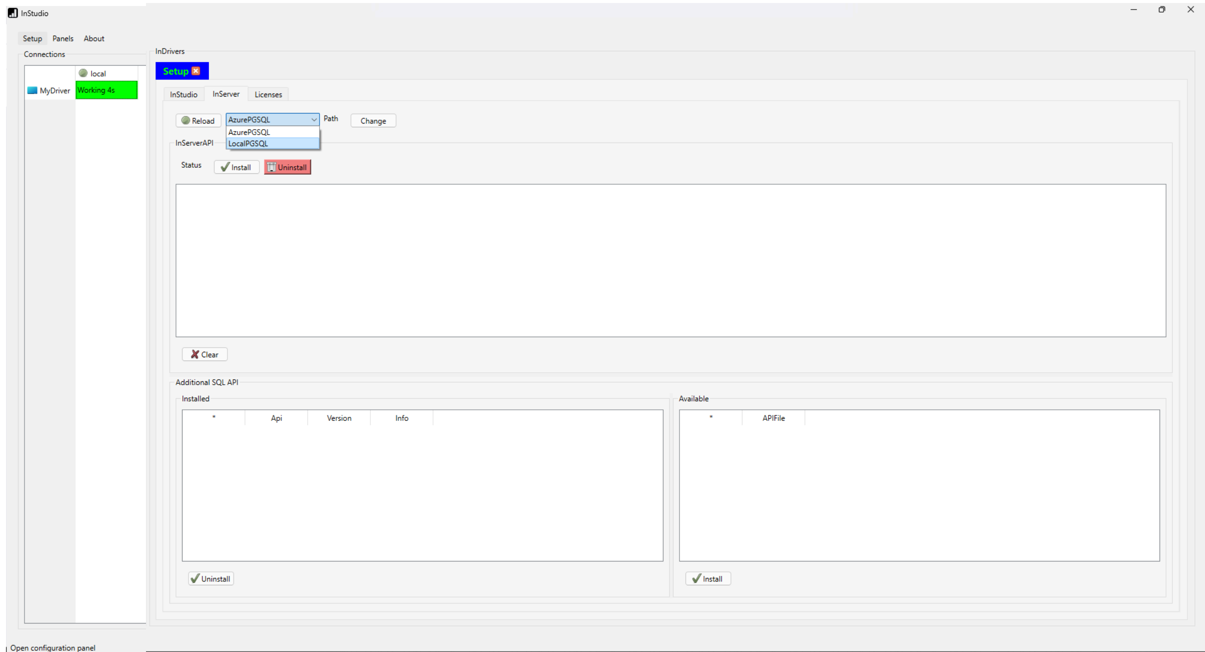
To debug the reason for any error, click on the server name, which opens a server window with a table of errors. Click 'Refresh,' and the last 100 errors will be updated.



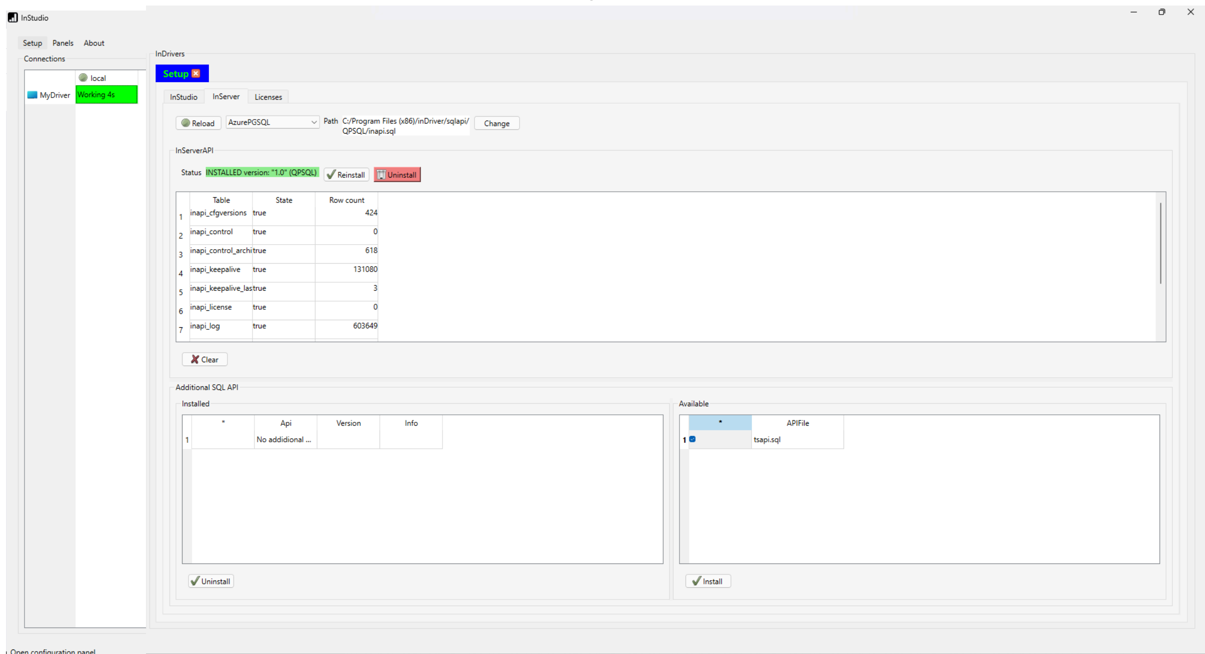
Install InServerAPI

When SQL Server(s) are configured correctly, the next step is to install InServerAPI. InServerAPI is a set of SQL functions and tables used by InDrivers and InStudio to communicate with each other, and store logs, configurations, messages, etc.

To install InServerAPI, open the 'InServer' tab, select the previously configured SQL server from the combo box, and press 'Install.'



If the installation is successful, the installed SQL tables should be listed in the table, and the status should be indicated in green.



Install the Additional API(s)

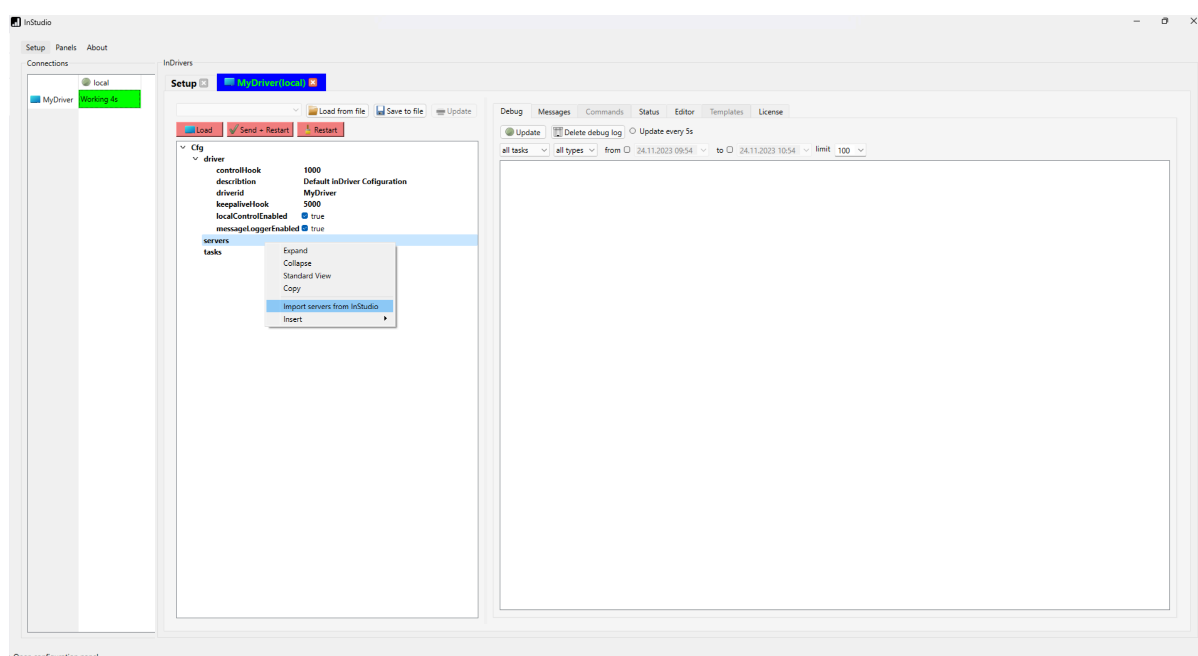
This step is not obligatory. The additional API may provide a set of SQL functions and tables to support the programming of various solutions.

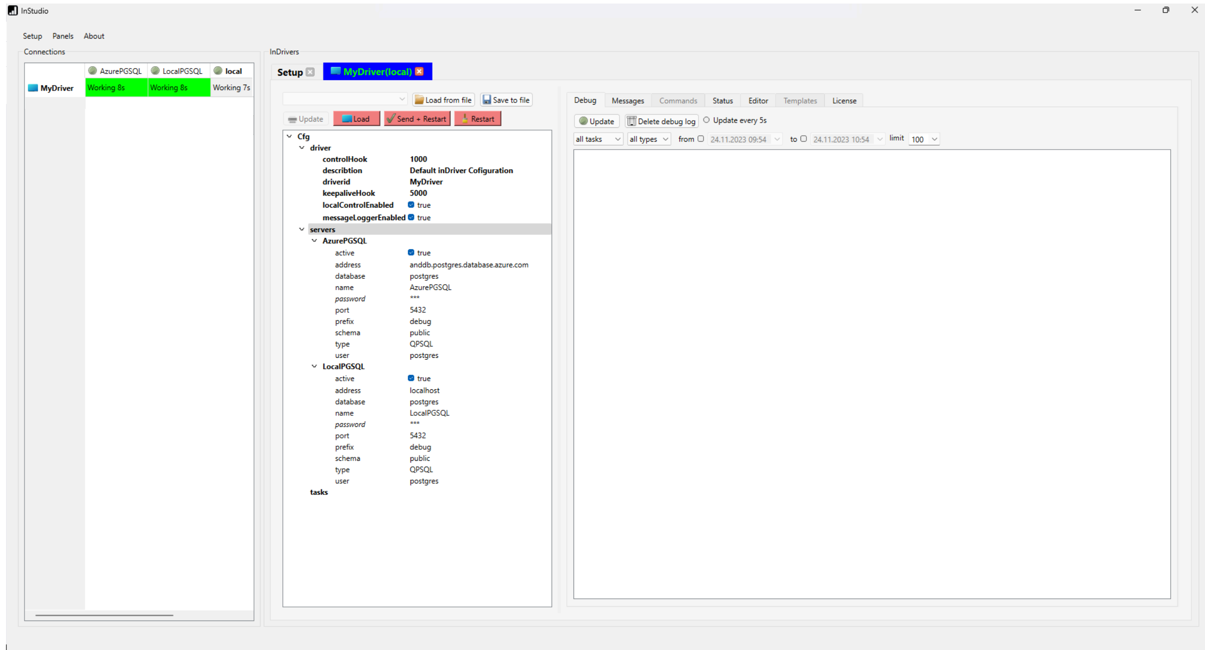
Currently, **TsApi (tsapi.sql) is available**, which includes functions dedicated to **JSON Time Series Processing**. To install an additional API on the previously selected server, select the API by checking it and pressing the 'Install' button. If the installation is successful, the 'Installed API' table should display the installed API.

InDriver first steps

Configuring SQL Servers

To begin configuring InDriver, click on the selected row in the column corresponding to the server or local connector.
If you prefer to use a database connection between InDriver and InStudio (with InServerAPI previously installed), simply press 'Import servers from InStudio.' All previously configured servers will be copied to InDriver's configuration. After this, press the 'Send+Restart' button. InDriver will restart and should be connected to the servers.

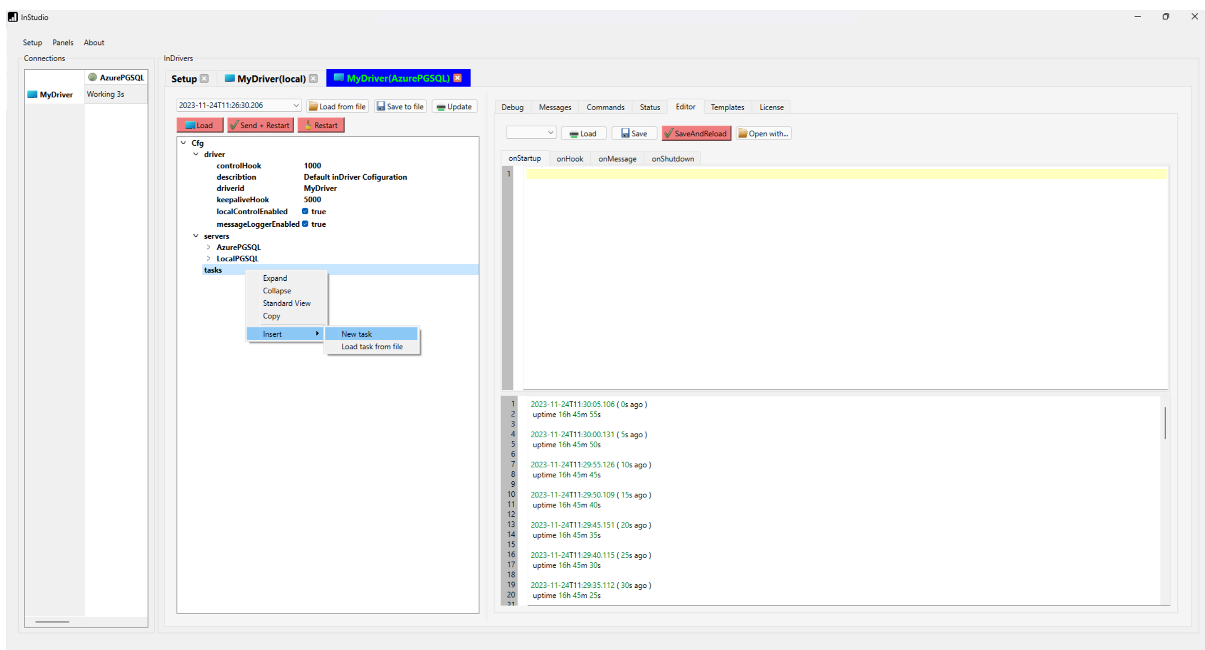




It is possible to add more SQL server connections for InDriver, not only the servers that play the role of InServers. InDriver tasks can easily interact with databases. Once the database source is configured, the InDriver.sqlExecute() function may be used by providing the configured server name.

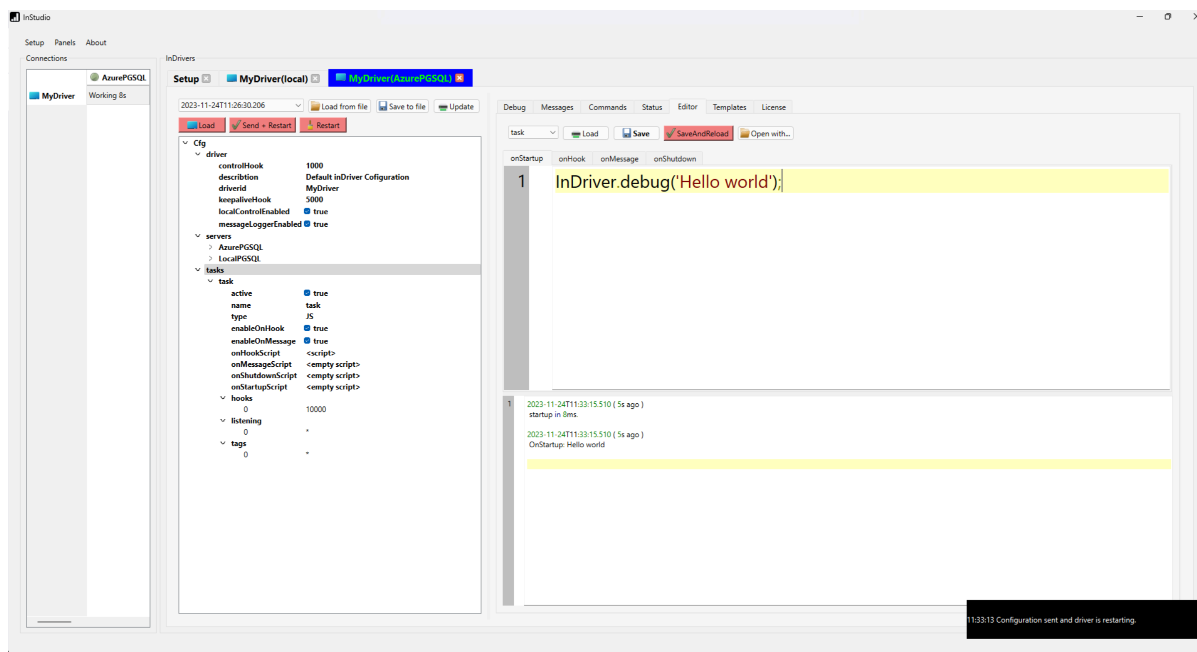
Configuring first task

To add a new task, click 'Insert' in the 'Tasks' context menu.



First ‘Hello world’ script

Open the task, select the 'Editor' tab, and write InDriver.debug('Hello world'); in the 'onStartup' script. Then, press the 'Save And Reload' button. Done! Your first JS task is being reloaded, and in a few seconds, you will see the debug log 'Hello world!' Great job!



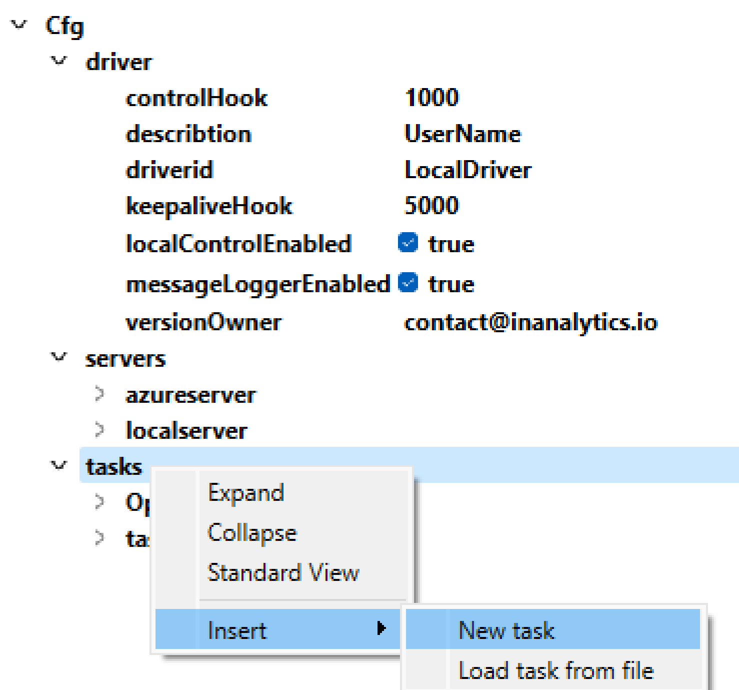
Features and examples

To be done soon :)

InDriver JS Tasks

InDriver can simultaneously execute multiple tasks. To insert a new task, click 'Insert' on the tasks content menu and select either 'New task' or 'Load task from file.' The second option enables loading saved task configurations, including example tasks with names starting from '@...' provided with InDriver in the default folder 'SavedTasks'. InDriver configuration is stored as a JSON object in the 'driver.cfg' file.

Inserting a new task:



Task configuration:

Key	Value
task	
active	true
enableOnHook	true
enableOnMessage	true
hooks	
0	10000
listening	
0	*
name	task
onHookScript	<script>
onMessageScript	<empty script>
onShutdownScript	<empty script>
onStartupScript	<empty script>
tags	
0	*
type	JS
UserData	
Number	123
String	Abc

"Default Task configuration items are displayed with bold font and consist of:

- ‘**active**’: true when the task will be started or false when is inactive
- ‘**enableonHook**’: true when execution of onHook function is enabled or false when disabled
- ‘**enableonMessage**’: true when execution of onMessage function is enabled or false when disabled
- ‘**hooks**’: is an array with defined Hook intervals. Intervals are in milliseconds (ms). InDriver can have multiple Hook intervals defined.
- ‘**listening**’: is an array with the names of other tasks being listened to. The default value is ‘*’, indicating that this task listens to messages sent from all other tasks."
- ‘**name**’ is the name of the task. Each task should have a unique name. When more tags have equal names, the first one will be started only.
- ‘**onHookScritp**’: is a JS code executed every Hook interval defined in ‘**hooks**’ item ‘**onMessageScript**’: is a JS code executed whenever another task executes InDriver.sendMessage() function with tags defined in ‘**listening**’ item
- ‘**onShutdown()**’: is a JS code executed once then the task is being shutdown
- ‘**onStartup()**’: is a JS code executed once when the task is starting
- ‘**tags**’: is an array used for filtering messages coming into this task. The default value is ‘*’, meaning that messages with any tags will be listened to by this task. Tags are employed to identify various messages; for example, messages with data intended for logging into the database can be marked with the 'archive' tag.
- that will be added to all messages sent from this Task
- ‘**type**’: User-defined JS Tasks are of type ‘**JS**’. InDriver uses some other internal tasks that are not configurable by the user. In user Tasks this parameter should be set as ‘**JS**’

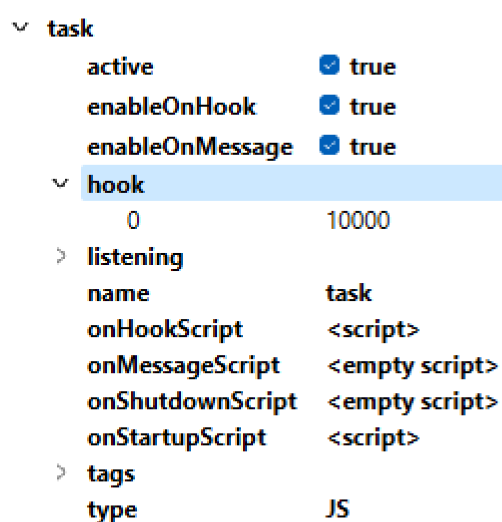
Additionally, the user can add other JSON Objects, Variables, or Arrays that will store user data. In this example ‘**userData**’ object is added with Number and String values.

Each InDriver Task functions as a distinct JavaScript Engine, executing four ‘pre-defined functions:

- onStartup()
- onShutdown()
- onHook()
- onMessage()

onStartup() and onShutdown() are called once when the Task starts and stops, respectively.

The onHook() function is continuously called at intervals defined in the Task configuration, or it can be declared using InDriver.installHook(hook) and removed using InDriver.uninstallHook(hook).



The above image shows the Task default configuration set.

Hooks are intervals defined in milliseconds (ms); for example, a hook of 10000 (equals 10s), executing every 10s synchronized with the computer clock where InDriver is running.

Recommended coding rules (AI)

When generating InDriver code, follow these rules:

1. Use only documented InDriver API.
When generating an InDriver task, use only functions, modules, signatures, and behavior explicitly documented in the InDriver Manual. Do not invent APIs, helper methods, field names, or runtime behavior.

2. Generate complete importable JSON, not loose code snippets.
Unless the user explicitly asks only for a script fragment, always return a complete JSON object representing exactly one InDriver task, ready to save as a `.cfg` file and import with *Load task from file*.

3. Match the InDriver task configuration model.
The generated JSON must follow the InDriver task configuration structure and include the standard task properties used by InDriver tasks:
`active`, `enableOnHook`, `enableOnMessage`, `hooks`, `listening`, `name`, `onHookScript`, `onMessageScript`, `onShutdownScript`, `onStartupScript`, `tags`, and `type`.
The type for user tasks must be `"JS"`. Additional user configuration may be added as extra JSON properties when needed.

4. Put lifecycle code into script fields, not wrapper functions.
InDriver evaluates the task as separate lifecycle script blocks. Therefore, code must be written directly into:

- `onStartupScript`
- `onHookScript`
- `onMessageScript`
- `onShutdownScript`

Do not wrap the task logic inside `function onStartup() {}`, `function onHook() {}`, etc.

5. Follow the official coding pattern.

- Initialization → `onStartupScript`
- Periodic logic → `onHookScript`
- Imports → `onStartupScript`
- Hooks → installed in startup when required

After each SQL execution:

- check `InDriver.sqlIsSucceeded()`
- log `InDriver.sqlLastError()` on failure

Prefer structured JSON in logs and messages.

6. Respect hook-based execution model.
If the task is periodic:

- define hook intervals in the `hooks` array
- implement logic in `onHookScript`

If multiple hooks are used:

- distinguish them using `InDriver.isHook(...)`

7. Keep the output strictly machine-importable.
Return only the raw JSON file content unless the user explicitly asks for explanation.
Do not include:

- markdown
- comments outside JSON
- explanations
- pseudo-JSON

All script fields must contain valid JSON strings with proper escaping.

8. Preserve example-compatible naming and style.
Use camelCase naming consistent with InDriver examples:

- `enableOnHook`
- `enableOnMessage`
- `onStartupScript`

- `onHookScript`
- `onMessageScript`
- `onShutdownScript`

9. Include optional configuration inside the task JSON.
If the task requires parameters (e.g. connection names, thresholds, table names, signals), include them as additional JSON properties such as:

- `userData`
- `config`

Access them using:
`InDriver.configuration(...)`

10. Generate production-ready task content.
The generated task must be:

- clean
- minimal
- deterministic
- directly usable after import

Do not include placeholders, mock logic, or invented behavior.

Execution Model & Script Structure Rules

11. Do not validate task configuration at runtime.
If the script is executing, the task is already properly loaded and configured in InDriver.
Do not add checks for task existence, configuration presence, or initialization state.

12. Do not use top-level `return` statements in lifecycle scripts.
The following are NOT function bodies:

- `onStartupScript`
- `onHookScript`
- `onMessageScript`
- `onShutdownScript`

Using `return` directly at the top level is invalid and must be avoided.

13. Use internal functions for control flow.
If early exit or structured logic is required, wrap code inside a function and invoke it.

Recommended pattern:

```
function run() {
  if (condition) {
    return;
  }

  // main logic
}

run();
```

14. Keep lifecycle scripts as execution containers.
Top-level script content should:

- define helper functions
- call one main function (e.g. `init()`, `run()`, `process()`)

Avoid placing complex logic directly at the top level.

15. Use early returns only inside functions.
Early return statements are allowed only inside user-defined functions, never at the script top level.

16. Maintain clean and deterministic execution structure.
Each lifecycle block should follow a consistent pattern:

- define functions
- call main function

This ensures:

- readability
- maintainability
- predictable execution

17. Separate initialization and processing logic.

- `onStartupScript` → initialization only
- `onHookScript` → periodic processing

Do not place heavy processing logic in `onStartupScript` unless explicitly required.

18. Follow production-grade structure.
Generated code must reflect real-world engineering practices:

- clear separation of concerns
- minimal side effects
- predictable behavior
- safe execution within InDriver runtime

InDriver Task JSON Template

```
{
 "active": true,
 "enableOnHook": true,
 "enableOnMessage": false,
 "hooks": [
   60000
 ],
 "listening": [
   "*"
 ],
 "name": "TaskName",
 "onHookScript": "",
 "onMessageScript": "",
 "onShutdownScript": "",
 "onStartupScript": "",
 "tags": [
   "*"
 ],
 "type": "JS",
 "userData": {
 }
}
```

Field Description (for AI usage)

- **active**
 Boolean flag indicating whether the task is enabled and should be executed by InDriver.
- **enableOnHook**
 Enables execution of the `onHookScript`. Must be `true` for periodic tasks.
- **enableOnMessage**
 Enables execution of the `onMessageScript` when messages are received from other tasks.
- **hooks**
 Array of hook intervals in milliseconds. Defines how often `onHookScript` is executed.
 Example: `60000` = every 60 seconds.
- **listening**
 Array of task names that this task listens to for incoming messages.
 `"*"` means listening to all tasks.
- **name**
 Unique name of the task. Must be unique within the InDriver configuration.
- **onStartupScript**
 JavaScript code executed once when the task starts.
 Used for:
  - API imports (`InDriver.import(...)`)
  - initialization
  - installing hooks (`InDriver.installHook(...)`)
  - configuration setup
- **onHookScript**
 JavaScript code executed periodically based on defined `hooks`.
 This is where the main logic of the task should be implemented.
- **onMessageScript**
 JavaScript code executed when a message is received via `InDriver.sendMessage(...)`.
 Used for inter-task communication.
- **onShutdownScript**
 JavaScript code executed once when the task is shutting down.
 Used for cleanup logic (optional).
- **tags**
 Array of tags assigned to messages sent by this task.
 Used for routing messages between tasks.
- **type**
 Must always be `"JS"` for user-defined JavaScript tasks.
- **userData**
 Optional object for custom configuration parameters.
 Should be used to store task-specific settings such as:
  - connection names
  - thresholds

- table names
- signal names

These values can be accessed using `InDriver.configuration([...])`.

Important Rule for AI

When generating an InDriver task:

- Always produce a **complete JSON object matching this structure**
- Place all logic directly inside:
  - `onStartupScript`
  - `onHookScript`
  - `onMessageScript`
  - `onShutdownScript`
- Do **not** wrap lifecycle code in functions like `function onStartup() {}`
- Ensure the JSON is **valid, clean, and ready to import as a `.cfg` file**

InDriver API

The InDriver object is the default global object available in every JavaScript task.

It provides functions for:

- loading additional API modules,
- scheduling periodic execution with hooks,
- exchanging messages between tasks,
- executing SQL,
- debugging,
- reading task configuration,

InDriver.configuration

Signature

```
InDriver.configuration(): Object
InDriver.configuration(path: String[]): Object
```

Description

Returns task configuration as a JSON value.

Two forms are available:

- InDriver.configuration() returns the full configuration of the current task
- InDriver.configuration(path) returns a nested fragment selected by path

Arguments

path

- type: String[]
- optional in the overloaded form
- path to nested configuration object/property

Examples:

```
[]

["UserData"]

["UserData", "Number"]
```

```
["Sql", "Connections", "Main"]
```

Returns

- Object
- Array
- primitive JSON value
- undefined if the path cannot be resolved

Behavior

- full configuration is returned when called without arguments
- nested lookup is performed when path is provided
- the returned value is a JSON value, not a mutable live reference to internal runtime state

Example

```
let fullCfg = InDriver.configuration();

let userData = InDriver.configuration(["UserData"]);

let numberValue = InDriver.configuration(["UserData", "Number"]);
```

Notes for AI

- Always treat the returned value as JSON
- Do not assume a schema unless the task configuration explicitly defines it

---

InDriver.currentPath

Signature

```
InDriver.currentPath(): String
```

Description

Returns the directory containing the current InDriver.exe runtime.

Returns

- String

Typical use

- loading scripts
- locating local files relative to runtime directory
- debugging deployment environment

Example

```
InDriver.debug(InDriver.currentPath());
```

InDriver.debug

Signature

```
InDriver.debug(message: String|Object, msgType: String = "debug", cut: Boolean = true): void
```

Description

Outputs a debug message to the Debug console.

This function is the primary mechanism for:

- logging runtime information,
- debugging task execution,
- reporting errors and system states.

If message is not a string, it is internally converted to JSON representation.

Arguments

message

- type: String | Object | Array | Number | Boolean
- content to log
- objects and arrays are serialized to JSON

msgType

- type: String
- allowed values:
  - "debug" (default)
  - "info"
  - "warning"
  - "critical"

    ○ "fatal"

Maps internally to logging severity levels.

cut

- type: Boolean
- default: true

Behavior:

- true → message truncated (~2000 chars)
- false → full message logged

Behavior

- Writes message to Debug console
- Serializes non-string values
- Applies severity level based on msgType

Returns

- void

Examples

```
InDriver.debug("Task started");
InDriver.debug("Connection failed", "critical");
InDriver.debug({ step: "import", ok: true }, "info", false);
```

Edge cases

- Very large objects → truncated if cut=true
- Invalid msgType → fallback to "debug"

Rules

- Use "critical" for execution failures
- Prefer structured JSON logs when debugging data pipelines
- For large objects, use cut = false only when full output is needed

---

InDriver.debugVariables

Signature

```
InDriver.debugVariables(): String
```

Description

Returns a textual list of variables defined in the global scope of the current JavaScript task and also prints them to the Debug console.

Useful for:

- debugging variable scope
- inspecting runtime state

Returns

- String — textual representation of variables

Example

```
var i = 10;
var j = "variable";
var array = [1, 2, 3, j];

InDriver.debugVariables();
```

Behavior

- Scans global JS scope
- Outputs variable names and values

Edge cases

- Large objects → output may be truncated
- Only global scope is included

InDriver.driverName

Signature

```
InDriver.driverName(): String
```

Description

Returns the name of the current InDriver instance.

Returns

- String

Example

```
const name = InDriver.driverName();
InDriver.debug("Driver name: " + name);
```

InDriver.getCpuUsage

Signature

```
InDriver.getCpuUsage(ms: Number = 1000): Number
```

Description

Measures and returns CPU usage.

Arguments

ms

- type: Number
- optional
- default: 1000
- measurement window in milliseconds

Returns

- Number

Behavior

- waits/measures over the provided interval
- longer intervals may provide more stable estimates

Example

```
let cpu = InDriver.getCpuUsage();
InDriver.debug("CPU usage: " + cpu + "%");
```

Rules

- This may be blocking for the specified interval
- Avoid calling too frequently in high-frequency hooks

InDriver.hooks

Signature

```
InDriver.hooks(): String
```

Description

Returns currently installed hook intervals.

Returns

- String

Behavior

The value is returned as a string representation of installed hooks. In practice it should be treated as serialized data rather than a native JS array unless the runtime documentation explicitly states otherwise.

Example

```
InDriver.debug("Hooks: " + InDriver.hooks());
```

Rules

- Do not assume this returns Number[]
- Treat it as text/serialized output unless verified otherwise in runtime examples

---

InDriver.hookTs

Signature

```
InDriver.hookTs(): Date
InDriver.hookTs(hook: Number): Date
```

Description

Returns DateTime of the currently executed hook.

Arguments

- hook (optional)
  - type: Number
  - hook interval in milliseconds

Behavior

- Without argument → current hook timestamp
- With argument → timestamp of specific hook

Example

```
const ts = InDriver.hookTs();

if (InDriver.isHook(10000)) {
  const ts = InDriver.hookTs(10000);
}
```

Rules

- Intended for use inside onHook
- Outside hook context, behavior may be undefined or implementation-dependent

---

InDriver.import

Signature

```
InDriver.import(apiName: String): Boolean
```

Description

Creates a JavaScript object providing access to additional API.

Arguments

- apiName: name of API module

Available APIs

- ModbusApi
- RestApi
- TsApi
- ProcessApi
- FileApi
- SerialPortApi
- TcpSocketApi
- TcpServerApi
- UdpSocketApi
- PdfApi
- XmlApi
- OPCUAClientApi
- OPCUAServerApi
- OpenCVApi
- SMTPApi
- HttpServerApi

Behavior

- Loads API module
- Creates global object (e.g. RestApi)
- Returns success flag

Returns

- Boolean
- true if module was loaded successfully
- false if module name is invalid or unsupported

Example

```
InDriver.import("ModbusApi");
Modbus.connectDevice(...);
```

Edge cases

- Invalid name → false

Notes for AI

- Always import a module before its first use
- Prefer importing in onStartup()

InDriver.installExtensions

Signature

```
InDriver.installExtensions(extension: String): void
```

Description

Installs additional JavaScript engine extensions provided by QJSEngine.

This function enables selected non-standard JS features in the current task runtime.

Supported extension names are mapped internally as follows:

- "TranslationExtension" → QJSEngine::TranslationExtension
- "ConsoleExtension" → QJSEngine::ConsoleExtension
- "GarbageCollectionExtension" → QJSEngine::GarbageCollectionExtension
- "AllExtensions" → QJSEngine::AllExtensions

If an unknown extension name is provided, the function does **not** throw an exception and does **not** return a status flag. Instead, it logs a debug message indicating that the extension is unknown and lists supported values.

Arguments

extension

- type: String
- name of extension to install

Allowed values

- "TranslationExtension"
- "ConsoleExtension"
- "GarbageCollectionExtension"
- "AllExtensions"

Returns

- void

Behavior

- installs the selected extension into the current JS engine
- affects the current task runtime
- if extension is invalid:
  - no extension is installed
  - a debug message is written

Internal fallback behavior for invalid value

Equivalent runtime behavior:

```
InDriver.debug(
 "InDriver.installExtensions('" + extension +
 "') - unknown extension, available: ConsoleExtension, GarbageCollectionExtension,
TranslationExtension, AllExtensions"
);
```

Examples

```
InDriver.installExtensions("ConsoleExtension");
InDriver.installExtensions("AllExtensions");
```

Invalid example

```
InDriver.installExtensions("MyCustomExtension");
```

Expected effect:

- no extension installed
- debug log with list of valid names

Notes for developer

Because the function returns void, there is no direct success/failure flag.
If you need to validate input, you should ensure the extension name is one of the supported values before calling it.

Rules

- Use only documented values
- Do not invent extension names
- Prefer "ConsoleExtension" or "AllExtensions" only when the task explicitly needs extra JS engine functionality

---

InDriver.installHook

Signature

```
InDriver.installHook(hook: Number): void
```

Description

Registers periodic execution interval for onHook.

Arguments

- hook
  - type: Number
  - interval in milliseconds

Examples:

- 1000 → 1 second
- 10000 → 10 seconds
- 60000 → 1 minute
- 3600000 → 1 hour

Behavior

- multiple hooks may be installed for one task
- hooks are synchronized with system clock
- once installed, onHook may be executed for that interval

Returns

- **void**

Example

```
InDriver.installHook(10000);
InDriver.installHook(60000);
InDriver.installHook(3600000);
```

Rules

- Prefer installing hooks in onStartup()
- When multiple hooks exist, combine with InDriver.isHook(...) inside onHook()

InDriver.isHook

Signature

```
InDriver.isHook(hook: Number): Boolean
```

Description

Checks whether the current onHook execution was triggered by the specified hook interval.

Arguments

hook

- type: Number
- hook interval in milliseconds

Returns

- Boolean

Example

```
if (InDriver.isHook(10000)) {
   let ts = InDriver.hookTs(10000);
}
```

Typical use

- differentiating behavior when one task has multiple installed hook intervals

---

InDriver.isPrivateMessage

Signature

```
InDriver.isPrivateMessage(): Boolean
```

Description

Returns true if the current message was sent by the same task that is now processing it.

Returns

- Boolean

Equivalent logic

```
InDriver.taskName() === InDriver.messageSender()
```

Example

```
if (InDriver.isPrivateMessage()) {
  InDriver.debug("Received private/self message");
}
```

InDriver.loadScript

Signature

```
InDriver.loadScript(scriptPath: String): Boolean
```

Description

Loads an external JavaScript file.

Arguments

path

- type: String
- file path
- may be relative to current runtime directory or absolute

Returns

- Boolean

Example

```
InDriver.loadScript("scriptslibrary/script.js");
InDriver.loadScript("c:/program files/InDriver/scriptslibrary/script.js");
```

InDriver.messageData

Signature

```
InDriver.messageData(): String
```

Description

Returns payload data of the currently processed message.

Returns

- String

Example

```
const msgData = InDriver.messageData();
```

Rules

- If structured data was sent, parse it explicitly, e.g. JSON.parse(msgData)

InDriver.messageSender

Signature

```
InDriver.messageSender(): String
```

Description

Returns the name of the task that sent the current message.

Returns

- String

Example

```
const sender = InDriver.messageSender();
```

InDriver.messageTags

Signature

```
InDriver.messageTags(): String
```

Description

Returns tags of the current message.

Returns

- String

Behavior

The underlying C++ API returns QString, so tags should be treated as string/serialized tag representation rather than guaranteed String[].

Example

```
const tags = InDriver.messageTags();

InDriver.debug("Tags: " + tags);
```

Rules

- Do not assume native JS array unless verified by runtime examples
- If your project standardizes tags as comma-separated or JSON text, parse accordingly

---

InDriver.messageTs

Signature

```
InDriver.messageTs(): Date
```

Description

Returns timestamp of the current message.

Returns

- Date

Example

```
const msgTs = InDriver.messageTs();
```

InDriver.msleep

Signature

```
InDriver.msleep(msecs: Number): void
```

Description

Sleeps/blocks the current task for the specified number of milliseconds.

Arguments

msecs

- type: Number
- milliseconds

Returns

- void

Example

```
InDriver.msleep(500);
```

Rules

- This is blocking
- Avoid inside high-frequency hooks unless explicitly intended

---

InDriver.onHook

Signature

```
InDriver.onHook(h: Number): void

InDriver.onHook(h: Number[]): void
```

Description

Triggers onHook immediately, simulating one or more hook events.

Main purpose:

- debugging
- forcing execution without waiting for real hook timing

Arguments

h

- type: Number or Number[]
- one hook interval or a list of hook intervals in milliseconds

Returns

- void

Example

```
InDriver.onHook(10000);

InDriver.onHook([1000, 10000]);
```

Notes

- Typically useful in onStartup() during tests/debugging
- Not a replacement for real installed hooks

InDriver.restartTask

Signature

```
InDriver.restartTask(name: String = ""): void
```

Description

Stops and restarts a task.

Arguments

name

- type: String
- optional
- default: ""

Behavior:

- empty string → restart current task
- non-empty string → restart named task

Returns

- void

Typical use

- retry logic
- recovery from transient failure
- task refresh after dynamic reconfiguration

Example

InDriver.restartTask();

InDriver.restartTask("ArchiveTask");

InDriver.sendMessage

Signature

```
InDriver.sendMessage(ts: Date, tags: String, data: String): void

InDriver.sendMessage(ts: Date, tags: String, data: JsonValue): void
```

Description

Sends a message to listening tasks.

Arguments

ts

- type: Date
- message timestamp

tags

- type: String
- tag or tag expression used for message routing/filtering

data

- type: String | JsonValue
- message payload

Returns

- void

Behavior

- message is routed to tasks that listen for matching senders/tags
- if data is a JSON value, it is serialized by the runtime

Examples

```
let ts = new Date();

let rows = InDriver.sqlExecute("LocalDatabase", ["select * from public.table"]);



InDriver.sendMessage(ts, "archive", JSON.stringify(rows));

InDriver.sendMessage(ts, "archive", rows);
```

Rukes

- Use Date or QDateTime-compatible timestamp values
- Prefer JSON payloads for structured inter-task communication
- tags in the current C++ signature is a single QString, not String[]

InDriver.setFlag

Signature

```
InDriver.setFlag(flag: String, info: String = ""): void
```

Description

Sets a status flag for the current task.

Arguments

flag

- type: String
- status/flag name

info

- type: String
- optional
- additional status information

Returns

- void

Typical use

- task state visualization
- alarm/busy/ok states
- operator feedback in UI

Example

```
InDriver.setFlag("busy", "calculations in progress");

InDriver.setFlag("ok", "archive completed");
```

InDriver.shutdown

Signature

```
InDriver.shutdown(): void
```

Description

Stops all running tasks and gracefully exits the InDriver application.

Returns

- void

Behavior

- only effective when InDriver runs in batch mode (-mode batch)
- in other modes, calling this function has no effect

Example

```
InDriver.shutdown();
```

Rules

- Use only for batch/automated workflows
- Do not assume it works in normal interactive mode

InDriver.sleep

Signature

```
InDriver.sleep(secs: Number): void
```

Description

Sleeps/blocks the current task for the specified number of seconds.

Arguments

secs

- type: Number
- seconds

Returns

- void

Example

```
InDriver.sleep(2);
```

Notes

- Blocking call
- Prefer sparing use in scheduled tasks

InDriver.sqlExecute

Signature

```
InDriver.sqlExecute(connectionName: String, sql: String[]): JsonArray

InDriver.sqlExecute(connectionName: String, sql: String[], bindings: Object): JsonArray
```

Description

Executes an SQL query on the selected configured server and returns result rows as JSON array.

Arguments

connectionName

- type: String
- configured SQL connection name

sql

- type: String[]
- SQL text represented as string list
- typically used as a concatenated SQL template

Examples:

["select * from public.table"]

["insert into ", "public.table", " (a) values (1)"]

bindings

- type: Object
- optional
- named SQL bindings for parameterized queries

Example:

{ data: json, ts: "2026-01-01T00:00:00Z" }

Returns

- JsonArray

Each row is returned as JSON object:

[

```
{ "column1": "value1", "column2": 123 }

]
```

Behavior

- executes query against selected server
- stores execution state for sqlIsSucceeded() and sqlLastError()
- with bindings, named placeholders such as :data can be used

Examples

```
let rows = InDriver.sqlExecute(

  "localserver",

  ["select * from public.table"]

);




let json = JSON.stringify(list);

InDriver.sqlExecute(

  "azureserver",

  ["insert into ", "<schema>", ".shelly (source, ts, data) values ('Shelly', :ts, :data);"],

  { ts: new Date().toISOString(), data: json }

);
```

Critical usage pattern

```
let rows = InDriver.sqlExecute("localserver", ["select * from public.table"]);


if (!InDriver.sqlIsSucceeded()) {

  InDriver.debug(InDriver.sqlLastError(), "critical");

  return;

}
```

Rules

- Always check sqlIsSucceeded() after execution
- Use bindings instead of string concatenation whenever possible
- Do not assume result array is non-empty

InDriver.sqlExecuteAll

Signature

```
InDriver.sqlExecuteAll(sql: String[]): Boolean
```

Description

Executes the provided SQL on all configured database servers.

Arguments

sql

- type: String[]
- SQL text/template

Returns

- Boolean

Behavior

- returns true only if execution succeeded on all servers
- if one server fails, return value is false
- some servers may still have executed successfully even when the overall result is false

Example

```
let ok = InDriver.sqlExecuteAll([

  "insert into public.table (ts, task) values (now(), '",

  InDriver.taskName(),

  "');"
```

```
]);
```

Rules

- Use carefully for distributed writes
- On failure, expect partial success across servers

InDriver.sqlIsSucceeded

Signature

```
InDriver.sqlIsSucceeded(): Boolean
```

Description

Returns execution status of the last sqlExecute(...) or sqlExecuteAll(...).

Returns

- Boolean

Example

```
let rows = InDriver.sqlExecute("localserver", ["select * from public.table"]);


if (!InDriver.sqlIsSucceeded()) {
  InDriver.debug(InDriver.sqlLastError());
}
```

InDriver.sqlLastError

Signature

```
InDriver.sqlLastError(): String
```

Description

Returns the error text from the last failed SQL execution.

Returns

- String
- empty string if no SQL error is currently stored

Example

```
if (!InDriver.sqlIsSucceeded()) {
  InDriver.debug(InDriver.sqlLastError(), "critical");
}
```

InDriver.sqlReopen

Signature

```
InDriver.sqlReopen(connectionName: String): Boolean
```

Description

Attempts to reopen the specified SQL connection.

Arguments

connectionName

- type: String
- configured SQL connection name

Returns

- Boolean

Typical use

- recovery after lost DB connection
- reconnect logic after communication failures

Example

```
if (!InDriver.sqlReopen("localserver")) {
  InDriver.debug("Could not reopen DB connection", "critical");
```

```
}
```

InDriver.taskName

Signature

```
InDriver.taskName(): String
```

Description

Returns the name of the current task.

Returns

- String

Example

```
const taskName = InDriver.taskName();
```

InDriver.uninstallHook

Signature

```
InDriver.uninstallHook(h: Number): void
```

Description

Removes a previously installed hook interval.

Arguments

h

- type: Number
- hook interval in milliseconds

Returns

- void

Example

```
InDriver.uninstallHook(60000);
```

Typical use

- stopping a periodic activity dynamically
- changing task scheduling at runtime

InDriver.updateStatistics

Signature

```
InDriver.updateStatistics(stat: String): void
```

Description

Updates runtime statistics/debug information for the task.

Arguments

stat

- type: String
- statistics payload, typically JSON text

Returns

- void

Example

```
InDriver.updateStatistics(JSON.stringify({
  rowsProcessed: 1000,
  elapsedMs: 250
}));
```

Rules

- Prefer valid JSON text
- Use for metrics/status, not ordinary logging

HttpServerApi

HttpServerApi allows an InDriver task to start an embedded HTTP server, register endpoints, dispatch requests to JavaScript handler functions, and build HTTP responses from JavaScript return values. It is intended for lightweight REST endpoints, operator panels, local integration endpoints, and exposing task data over HTTP. The API is stateful: registered routes, debug mode, and CORS headers are stored inside the HttpServerApi instance.

HttpServerApi.debugMode

Signature

```
HttpServerApi.debugMode(mode: Boolean = true): void
```

Description

Enables or disables debug mode for the embedded HTTP server.

When debug mode is enabled:

- incoming requests are serialized and logged to the Debug console before the JS handler is called,
- internal error details may be included in responses generated by the safe error helpers.

When disabled:

- request logging is not performed,
- client-facing errors are more generic, while details remain internal.

Arguments

mode

- type: Boolean
- optional
- default: true

Returns

- void

Behavior

The function only updates internal flag mDebugMode. It does not start or stop the server and does not affect already registered routes except changing runtime logging/error verbosity.

Example

```
HttpServerApi.debugMode(true);
```

Rules

- Use true during development and testing.
- Use false in production unless verbose request logging is required.

HttpServerApi.listen

Signature

```
HttpServerApi.listen(httpServerCfg: String): Number

HttpServerApi.listen(httpServerCfg: Object = {}): Number
```

Description

Starts the HTTP server and binds it to the configured address and port.

There are two overloads:

- JSON string
- JSON object

The object overload is internally converted to JSON text and then processed by the string overload.

Arguments

httpServerCfg

- type: String | Object
- optional in object overload
- configuration with the following supported keys:

address

- type: String
- optional
- supported values, case-insensitive:

  - "Broadcast"
  - "LocalHost"
  - "LocalHostIPv6"
  - "Any"
  - "AnyIPv6"
  - "AnyIPv4"

If omitted:

- defaults to LocalHost

If invalid:

- falls back to LocalHost silently.

port

- type: Number
- optional
- TCP port to bind

If omitted:

- defaults to 8080 in current source code

Manual text previously suggested random port may be chosen when omitted, but current implementation sets default port to 8080. The code should be treated as source of truth here.

Returns

- Number
- opened server port
- returns 0 if binding fails

Behavior

The server listens using QHttpServer::listen(address, port). After the bind attempt, a debug line is emitted showing the final address, returned port, and a textual classification such as:

- Any (IPv4)
- Any (IPv6)
- LocalHost (IPv4)
- LocalHost (IPv6)
- Loopback
- Multicast
- Specific IP

Examples

```
let port = HttpServerApi.listen({
  address: "Any",
  port: 8080
});
let port = HttpServerApi.listen('{"address":"LocalHost","port":9000}');
```

Edge cases

- invalid or unsupported address string → defaults to LocalHost
- occupied port or bind failure → returns 0
- calling listen does not automatically define routes; routes must be registered separately

Rules

- Treat 0 as failure.
- Use "Any" to expose the server beyond localhost.
- Use "LocalHost" for local-only endpoints.

HttpServerApi.route

Signature

```
HttpServerApi.route(path: String, method: String, functionName: String): Boolean
```

Description

Registers an HTTP endpoint and binds it to a JavaScript function in global scope.

Incoming requests matching the normalized path and parsed HTTP method will invoke the specified JS function. The handler function is looked up dynamically by name in the global JS object at request time, not at registration time.

Arguments

path

- type: String

- route pattern

Behavior:

- trimmed
- if empty → becomes "/"
- if missing leading slash → leading slash is added automatically

Examples:

"/orders"

"orders"   // normalized to "/orders"

""   // normalized to "/"

method

- type: String
- HTTP method, case-insensitive

Supported values:

- "GET"
- "POST"
- "PUT"
- "DELETE"
- "PATCH"
- "OPTIONS"

If invalid:

- registration fails and returns false.

functionName

- type: String
- name of JavaScript function in global scope

Requirements:

- must not be empty after trimming
- must refer to a callable global JS function at request time

If missing/blank:

- registration fails and returns false

If route exists physically already:

- handler mapping is updated and function returns true without creating a second physical route.

Returns

- Boolean
- true if route registration or handler update succeeded
- false if:
  - JS engine is unavailable
  - function name is empty
  - HTTP method is invalid
  - underlying QHttpServer route creation fails

Request object passed to handler

The JS function is called with one argument: req.

req contains fields documented in the manual:

- query
- params
- body
- headers
- method
- url
- path
- remoteAddress
- remotePort

Runtime behavior during request handling

When a request hits a registered route:

1. If JS engine is missing → 500 Internal Server Error
2. If the script is paused → 503 Service Unavailable
3. If no handler name is configured for the endpoint → 404 Not Found
4. If the named JS function does not exist or is not callable → 404 Not Found
5. Otherwise:
   - engine property scriptName is set to the function name
   - request is converted to JS object
   - if debug mode is enabled, request JSON is logged
   - the JS function is called
   - its result is converted into an HTTP response

Response handling from handler return value

Full response object

If handler returns object:

```
{
 status: 200,
 contentType: "application/json",
 body: { ok: true }
}
```

then:

- status is used as HTTP status
- contentType is used as response content type
- body is serialized accordingly
- CORS headers are appended if configured

Simplified return values

If handler returns:

- Object or Array → JSON response, application/json
- String → text/plain; charset=utf-8
- Boolean → "true" / "false" as text
- Number → text
- Null → "null" as text
- other values → converted via toString() and sent as text/plain

Examples

```
HttpServerApi.route("/orders", "post", "createOrder");

function createOrder(req) {
  return {
     status: 200,
     contentType: "application/json",
     body: { ok: true }
  };
}
```

```
HttpServerApi.route("status", "get", "getStatus");


function getStatus(req) {

  return { message: "System OK" };

}
```

Edge cases

- empty handler name → false
- unsupported method → false
- duplicate route same path+method → existing physical route reused, handler mapping updated
- handler removed later from global scope → requests start returning 404

Rules

- Always define the JS handler function in global scope before using the route in production code.
- Prefer explicit full-response object when status code or content type matters.
- Prefer normalized paths with leading /.

HttpServerApi.setCorsHeaders

Signature

```
HttpServerApi.setCorsHeaders(headers: Object): void
```

Description

Replaces the current set of custom CORS/response headers with a new JSON object.

Headers stored here are appended to every response generated by HttpServerApi, including normal responses and error responses. The function does not merge incrementally; it replaces the stored header object.

Arguments

headers

- type: Object

- JSON object where:
    - key = header name
    - value = header value converted to string

Typical examples:

- "Access-Control-Allow-Origin": "*"
- "Access-Control-Allow-Methods": "GET, POST, PUT, DELETE, OPTIONS"
- "Access-Control-Allow-Headers": "Content-Type"

Returns

- void

Behavior

- overwrites mHeaders
- all stored headers are added to subsequent responses

Example

```
HttpServerApi.setCorsHeaders({
  "Access-Control-Allow-Origin": "*",
  "Access-Control-Allow-Methods": "GET, POST, PUT, DELETE, OPTIONS",
  "Access-Control-Allow-Headers": "Content-Type"
});
```

Edge cases

- empty object → effectively clears previously configured extra headers
- non-string values are converted to string on response output

Rules

- Call once during startup unless you intentionally want to replace headers dynamically.
- Use before listen() or before handling traffic for predictable behavior.

Recommended usage pattern

onStartup:

```
  InDriver.import("HttpServerApi");
  HttpServerApi.debugMode(true);
  HttpServerApi.setCorsHeaders({
     "Access-Control-Allow-Origin": "*",
     "Access-Control-Allow-Methods": "GET, POST, PUT, DELETE, OPTIONS",
     "Access-Control-Allow-Headers": "Content-Type"
  });

  HttpServerApi.route("/status", "get", "getStatus");
  HttpServerApi.route("/orders", "post", "createOrder");

  let port = HttpServerApi.listen({
     address: "Any",
     port: 8080
  });

  InDriver.debug("HTTP server started on port: " + port);


function getStatus(req) {
  return {
     status: 200,
     contentType: "application/json",
     body: { ok: true, path: req.path }
  };
}

function createOrder(req) {
  return { received: req.body };
}
```

CryptographicHashAPI

`CryptographicHashApi` allows an InDriver task to calculate cryptographic hashes for text and raw binary data.

The API supports two usage modes:

- **stateful hashing** — build a hash incrementally using `addData()` / `addDataUtf8()` and finalize with `result()` or `resultHex()`
- **one-shot hashing** — compute a hash in a single call using `hash*()` methods

The API is stateful: the selected algorithm and the internal hashing context are stored inside the instance.

Default algorithm: `SHA256`

Supported algorithms

The following algorithms are available:

Classic

- `MD4`
- `MD5`
- `SHA1`

SHA-2

- `SHA224`
- `SHA256`
- `SHA384`
- `SHA512`

SHA-3

- `SHA3_224`
- `SHA3_256`
- `SHA3_384`
- `SHA3_512`

BLAKE2

- `BLAKE2B_160`
- `BLAKE2B_256`

- `BLAKE2B_384`
- `BLAKE2B_512`
- `BLAKE2S_128`
- `BLAKE2S_160`
- `BLAKE2S_224`
- `BLAKE2S_256`

Algorithm names are:

- case-insensitive
- trimmed
- allow `_` and `-` interchangeably for some variants (e.g. `SHA3-256`)

CryptographicHashApi.addData

Signature

```
CryptographicHashApi.addData(text: String): Boolean
```

Description

Adds text data to the current incremental hash.

The string is converted to UTF-8 before hashing.

Arguments

`text`

type: `String`

Text to append to the current hash input.

Returns

`Boolean`

- `true` if data was added
- `false` if internal state is unavailable

Example

```
CryptographicHashApi.reset();
CryptographicHashApi.addData("Hello ");
CryptographicHashApi.addData("World");
```

Rules

- Use for normal string input.
- Use `addDataUtf8()` when exact byte control is required.

CryptographicHashApi.addDataUtf8

Signature

```
CryptographicHashApi.addDataUtf8(data: QByteArray): Boolean
```

Description

Adds raw byte data to the current incremental hash.

No encoding or conversion is performed.

Arguments

`data`

type: `QByteArray`

Raw bytes.

Returns

`Boolean`

- `true` if data was added
- `false` if internal state is unavailable

Example

```
let bytes = InDriver.toUtf8("abc");
CryptographicHashApi.addDataUtf8(bytes);
```

Rules

- Use when hashing binary data or pre-encoded UTF-8.

CryptographicHashApi.algorithm

Signature

```
CryptographicHashApi.algorithm(): String
```

Description

Returns the currently selected algorithm.

Returns

`String`

Normalized algorithm name.

Example

```
let algo = CryptographicHashApi.algorithm(); // "SHA256"
```

Rules

- Useful for diagnostics and logging.

CryptographicHashApi.algorithms

Signature

```
CryptographicHashApi.algorithms(): Array
```

Description

Returns the list of all supported algorithms.

Returns

`Array`

Array of strings.

Example

```
let list = CryptographicHashApi.algorithms();
```

Rules

- Use for dynamic configuration or UI selection.

CryptographicHashApi.hash

Signature

```
CryptographicHashApi.hash(text: String): QByteArray
```

Description

Computes a one-shot hash for a string.

The string is converted to UTF-8 and hashed immediately.

Arguments

`text`

type: `String`

Input text.

Returns

`QByteArray`

Binary hash.

Example

```
let hash = CryptographicHashApi.hash("abc");
```

Rules

- Use for simple one-off hashing.

CryptographicHashApi.hashHex

Signature

```
CryptographicHashApi.hashHex(text: String): String
```

Description

Computes a one-shot hash and returns it as hexadecimal string.

Arguments

`text`

type: `String`

Input text.

Returns

`String`

Hex-encoded hash.

Example

```
let hex = CryptographicHashApi.hashHex("abc");
```

Rules

- Preferred for logging, IDs, and JSON usage.

CryptographicHashApi.hashHexUtf8

Signature

```
CryptographicHashApi.hashHexUtf8(data: QByteArray): String
```

Description

Computes a one-shot hash for raw bytes and returns hex string.

Arguments

`data`

type: `QByteArray`

Input bytes.

Returns

`String`

Hex-encoded hash.

Example

```
let bytes = InDriver.toUtf8("abc");
let hex = CryptographicHashApi.hashHexUtf8(bytes);
```

Rules

- Use when working with binary data.

CryptographicHashApi.hashUtf8

Signature

```
CryptographicHashApi.hashUtf8(data: QByteArray): QByteArray
```

Description

Computes a one-shot hash for raw bytes.

Arguments

`data`

type: `QByteArray`

Input bytes.

Returns

`QByteArray`

Binary hash.

Example

```
let bytes = InDriver.toUtf8("abc");
let hash = CryptographicHashApi.hashUtf8(bytes);
```

Rules

- Use when binary output is required.

CryptographicHashApi.isSupported

Signature

```
CryptographicHashApi.isSupported(algorithm: String): Boolean
```

Description

Checks whether a given algorithm is supported.

Arguments

`algorithm`

type: `String`

Algorithm name.

Returns

`Boolean`

- `true` if supported
- `false` otherwise

Example

```
if (CryptographicHashApi.isSupported("SHA3-256")) {
  CryptographicHashApi.setAlgorithm("SHA3-256");
}
```

Rules

- Use before `setAlgorithm()` in dynamic scenarios.

CryptographicHashApi.reset

Signature

```
CryptographicHashApi.reset(): Boolean
```

Description

Clears current incremental hash state.

Returns

`Boolean`

Always `true`.

Behavior

- keeps current algorithm
- clears all previously added data

Example

```
CryptographicHashApi.reset();
```

Rules

- Use before starting a new incremental hash.

CryptographicHashApi.result

Signature

```
CryptographicHashApi.result(): QByteArray
```

Description

Returns binary hash for current incremental state.

Returns

`QByteArray`

Binary digest.

Example

```
let digest = CryptographicHashApi.result();
```

Rules

- Use when raw bytes are needed.

CryptographicHashApi.resultHex

Signature

```
CryptographicHashApi.resultHex(): String
```

Description

Returns current hash as hexadecimal string.

Returns

`String`

Hex-encoded digest.

Example

```
let hex = CryptographicHashApi.resultHex();
```

Rules

- Preferred for most scripting use cases.

CryptographicHashApi.setAlgorithm

Signature

```
CryptographicHashApi.setAlgorithm(algorithm: String): Boolean
```

Description

Sets the active hashing algorithm.

Arguments

`algorithm`

type: `String`

Algorithm name.

Returns

`Boolean`

- `true` if successful
- `false` if unsupported

Behavior

- updates current algorithm
- resets internal hash state

Example

```
CryptographicHashApi.setAlgorithm("SHA512");
```

Rules

- Call before hashing when non-default algorithm is needed.
- Changing algorithm clears current hash state.

Recommended usage pattern

One-shot hashing

```
InDriver.import("CryptographicHashApi");

let hex = CryptographicHashApi.hashHex("Hello World");
```

Incremental hashing

```
InDriver.import("CryptographicHashApi");

CryptographicHashApi.setAlgorithm("SHA256");
CryptographicHashApi.reset();

CryptographicHashApi.addData("part1");
```

```
CryptographicHashApi.addData("part2");

let hex = CryptographicHashApi.resultHex();
```

OPCUPClientApi

OPCUAClientApi provides OPC UA client access from JavaScript tasks. It supports:

- creating and reusing named client connections,
- browsing nodes,
- reading multiple node values,
- writing multiple node values,
- checking connection state,
- retrieving the last error.

The API stores client instances by clientName, so the same logical client can be reconfigured and reconnected by calling connect(...) again with the same name.

OPCUAClientApi.browse

Signature

```
OPCUAClientApi.browse(
  clientName: String,
  nodeIdStr: String = "",
  filter: String = "",
  limit: Number = 100
): Array
```

Description

Browses the OPC UA address space starting from a selected node and returns discovered nodes as a JSON array.

Arguments

clientName

- type: String
- name of previously created OPC UA client connection

nodeIdStr

- type: String
- optional
- start node identifier

Supported special values in current implementation:

- "ROOT"
- "Objects"
- "Types"
- "Views"

Behavior:

- empty string → defaults to Objects folder
- invalid or unparsable node string → also falls back to Objects folder

filter

- type: String
- optional
- filter expression interpreted by MyOPCUAClientBrowseFilter

From source, the filter object supports at least:

- namespace filtering,
- object name filtering,
- variable name filtering.

Because the exact JS filter grammar is implemented deeper in helper classes and is not fully documented in the visible snippet, this argument should be documented as an implementation-defined filter string rather than over-specified.

limit

- type: Number
- optional
- default: 100
- maximum number of nodes to browse

Returns

- Array
- JSON array of browsed nodes

Behavior

- clears previous lastError
- looks up client by name
- resolves nodeIdStr into a start node
- performs browse on existing client
- on error:
    - fills mLastError
    - logs fatal debug message

- returns current array result, often empty

Example

```
InDriver.import("OPCUAClientApi");

OPCUAClientApi.connect(

  "Client1",

  '{"EndpointUrl":"opc.tcp://localhost:4840","Timeout":5000}'

);

let nodes = OPCUAClientApi.browse("Client1", "Objects", "", 50);

InDriver.debug(nodes);
```

Edge cases

- unknown clientName → error: client not initialized
- invalid nodeIdStr → falls back to Objects
- browse failure → returns empty or partial array and sets lastError

Rules

- Always call connect(...) before browsing.
- If you need deterministic start point, pass explicit node id or one of the built-in root names.
- After unexpected empty results, check OPCUAClientApi.lastError().

OPCUAClientApi.connect

Signature

```
OPCUAClientApi.connect(clientName: String, cfg: Object): Boolean

OPCUAClientApi.connect(clientName: String, cfg: String = ""): Boolean
```

Description

Creates or reconfigures a named OPC UA client and attempts to connect it to a server.

Arguments

clientName

- type: String
- unique logical client identifier

cfg

- type: Object | String
- JSON configuration for the OPC UA connection

The object overload is internally serialized to JSON and passed to the string overload.

Known documented field:

- EndpointUrl
- Timeout

Example:

```
{
 "EndpointUrl": "opc.tcp://localhost:4840",
 "Timeout": 5000
}
```

Returns

- Boolean
- true if connection succeeds
- false if connection fails

Behavior

- clears previous lastError
- if client with same name already exists:
  - updates its configuration
  - attempts reconnect using new settings
- otherwise:
  - creates new MyOPCUAClient
  - connects runtime shutdown/interruption signals
  - stores it in map
- on connect failure:
  - mLastError is set to "OPC client <name> connection failed with error: ..."
  - fatal debug log is emitted

Example

```
InDriver.import("OPCUAClientApi");

let ok = OPCUAClientApi.connect(
  "Client1",
  '{"EndpointUrl":"opc.tcp://localhost:4840","Timeout":5000}'
);

if (!ok) {
  InDriver.debug(OPCUAClientApi.lastError(), "critical");
}
```

Edge cases

- malformed config string may still be accepted at API level, but connection can fail deeper in client setup
- reconnecting same clientName updates existing client instead of creating a duplicate

Notes for AI

- Reuse stable clientName values when a task owns one persistent connection.
- Check lastError() after false.

OPCUAClientApi.disconnect

Signature

```
OPCUAClientApi.disconnect(clientName: String): Boolean
```

Description

Disconnects a previously created OPC UA client.

Arguments

clientName

- type: String
- name of existing client

Returns

- Boolean

Behavior

- clears previous lastError
- looks up client by name
- disconnects client if it exists
- if client is missing:
  - sets mLastError to a “not initialized, use connect(...) first” message
  - logs fatal debug message
  - returns false

Example

```
let ok = OPCUAClientApi.disconnect("Client1");

if (!ok) {
  InDriver.debug(OPCUAClientApi.lastError(), "critical");
}
```

Edge cases

- disconnecting unknown client name → false
- disconnect failure from underlying client → false and lastError

---

OPCUAClientApi.isConnected

Signature

```
OPCUAClientApi.isConnected(clientName: String): Boolean
```

Description

Checks whether the named OPC UA client is currently connected.

Arguments

clientName

- type: String

Returns

- Boolean

Example

```
if (OPCUAClientApi.isConnected("Client1")) {
  InDriver.debug("Client1 connected");

}
```

Rules

- Use before read/write if connection stability is uncertain.
- If false, reconnect or inspect lastError().

OPCUAClientApi.lastError

Signature

```
OPCUAClientApi.lastError(): String
```

Description

Returns the last error message stored by the client API.

Returns

- String

Behavior

- returns empty string when no error is currently stored

Example

```
let err = OPCUAClientApi.lastError();

if (err) InDriver.debug(err, "critical");
```

OPCUAClientApi.readMultipleNodes

Signature

```
OPCUAClientApi.readMultipleNodes(clientName: String, nodeCfg: String): Array

OPCUAClientApi.readMultipleNodes(clientName: String, nodeCfg: Array): Array
```

Description

Reads values from multiple OPC UA nodes and returns results as JSON array.

Arguments

clientName

- type: String
- existing client name

nodeCfg

- type: String | Array
- JSON array describing nodes to read

Minimal documented example:

[

{ "nodeId": "ns=2;s=Temperature" }

]

Returns

- Array

Behavior

- array overload is serialized and passed to string overload
- clears previous lastError
- if client is missing:
  - sets not-initialized error
  - logs fatal
  - returns array result, usually empty
- if JSON parse fails:
  - stores parse error
  - logs fatal
  - returns empty array
- if underlying client read succeeds:
  - returns updated node array with read values
- otherwise:
  - returns current array and sets lastError

The source follows the same pattern as other OPC methods: parse input JSON, perform operation, enrich same structure, return resulting QJsonArray.

Example

```
let data = OPCUAClientApi.readMultipleNodes(
  "Client1",
  '[{"nodeId":"ns=2;s=Temperature"}]'
);
InDriver.debug(data);
```

Edge cases

- unknown client → failure
- invalid JSON → failure
- valid request but unreadable node → underlying read failure and lastError

Notes for AI

- Prefer array/object construction in JS and let runtime serialize when supported.
- Always check lastError() if result is unexpectedly empty.

OPCUAClientApi.runIterate

Signature

```
OPCUAClientApi.runIterate(clients: String[] = [], timeout: Number = 0): void
```

Description

Runs client iterate cycle for all clients or only selected clients.

Arguments

clients

- type: String[]
- optional
- empty list means iterate all registered clients

timeout

- type: Number
- optional
- default: 0

Returns

- void

Behavior

Iterates through internal client map and calls runIterateClient(timeout) on:

- every client if clients is empty,
- only named clients if clients contains specific keys.

Example

```
OPCUAClientApi.runIterate();

OPCUAClientApi.runIterate(["Client1", "Client2"], 10);
```

Notes

This is a low-level helper. For many standard task flows, explicit connect/read/write is enough.

OPCUAClientApi.writeMultipleNodes

Signature

```
OPCUAClientApi.writeMultipleNodes(clientName: String, nodeCfg: String): Boolean

OPCUAClientApi.writeMultipleNodes(clientName: String, nodeCfg: Array): Boolean
```

Description

Writes values to multiple OPC UA nodes.

Arguments

clientName

- type: String

nodeCfg

- type: String | Array
- JSON array of write definitions

Documented example:

```
[
 { "nodeId": "ns=2;s=Temperature", "value": 25 }
]
```

Returns

- Boolean

Behavior

- clears lastError
- parses request
- writes values via client
- on underlying write error:
    - client implementation may disconnect and reconnect automatically if lastError is not empty
    - API returns false
- missing client or invalid JSON also return false

Example

```
let ok = OPCUAClientApi.writeMultipleNodes(
  "Client1",
  '[{"nodeId":"ns=2;s=Temperature","value":25}]'
);

if (!ok) {
  InDriver.debug(OPCUAClientApi.lastError(), "critical");
}
```

Rules

- Prefer writing explicit typed values that match target node expectations.
- After failed writes, recheck connection state with isConnected().

OPC UA client pattern

onStartup:

```
  InDriver.import("OPCUAClientApi");

  let ok = OPCUAClientApi.connect(
```

```
    "Client1",
    '{"EndpointUrl":"opc.tcp://localhost:4840","Timeout":5000}'
);

if (!ok) {
    InDriver.debug(OPCUAClientApi.lastError(), "critical");
}
```

onHook:

```
let values = OPCUAClientApi.readMultipleNodes(
    "Client1",
    '[{"nodeId":"ns=2;s=Temperature"}]'
);

if (OPCUAClientApi.lastError()) {
    InDriver.debug(OPCUAClientApi.lastError(), "critical");
    return;
}

InDriver.debug(values);
```

OPCUPServerApi

OPCUAServerApi provides an embedded OPC UA server inside an InDriver task. It supports:

- starting/stopping the server,
- adding nodes,
- reading node values,
- writing node values,
- retrieving the last error.

The server is stored as a single internal mOPCServer instance. Most operations fail if start(...) has not been called first.

OPCUAServerApi.addNodes

Signature

```
OPCUAServerApi.addNodes(nodeCfg: String, folder: String = ""): Boolean

OPCUAServerApi.addNodes(nodeCfg: Array, folder: String = ""): Boolean
```

Description

Adds nodes to the running OPC UA server.

Arguments

nodeCfg

- type: String | Array
- JSON configuration of nodes to add

Documented example:

```
[
 { "nodeId": "ns=2;s=TemperatureSensor", "name": "Sensor" }
]
```

folder

- type: String
- optional
- destination parent folder
- current implementation uppercases the value before passing it deeper into server logic

Manual says default parent is "Objects", though the direct API default is empty string and underlying server logic determines how that is resolved.

Returns

- Boolean

Behavior

- fails if server not started
- parses JSON
- on parse error:
  - sets lastError
  - returns false
- on server-side add failure:
  - prefixes "addNodes function failed. " to underlying server error
  - logs fatal
  - returns false

Example

```
InDriver.import("OPCUAServerApi");

OPCUAServerApi.start('{"EndpointUrl":"opc.tcp://localhost:4840"}');

let ok = OPCUAServerApi.addNodes(
'[{"nodeId":"ns=2;s=TemperatureSensor","name":"Sensor"}]',
"Objects");
```

Edge cases

- calling before start() → false
- invalid JSON → false
- unsupported node schema → false with server error text

OPCUAServerApi.lastError

Signature

```
OPCUAServerApi.lastError(): String
```

Description

Returns the last stored server API error.

Returns

- String

Example

```
let err = OPCUAServerApi.lastError();

if (err) InDriver.debug(err, "critical");
```

OPCUAServerApi.readMultipleNodes

Signature

```
OPCUAServerApi.readMultipleNodes(nodeCfg: String): Array

OPCUAServerApi.readMultipleNodes(nodeCfg: Array): Array
```

Description

Reads values from multiple nodes exposed by the embedded OPC UA server.

Arguments

nodeCfg

- type: String | Array
- JSON array describing nodes to read

Documented example:

```
[
 { "nodeId": "ns=2;s=Temperature" }
]
```

Returns

- Array

Behavior

- if server exists and JSON parses correctly:
  - underlying server fills values into request structure

    - resulting array is returned
- on failure:
    - lastError is set
    - fatal debug log emitted
    - current parsed array is still returned, which may be empty or partially filled

Example

```
let data = OPCUAServerApi.readMultipleNodes(

  '[{"nodeId":"ns=2;s=Temperature"}]'

);

InDriver.debug(data);
```

Edge cases

- server not started → failure
- invalid JSON → failure
- unknown node → underlying read failure

Rules

- Treat result as structured array, not plain text.
- If values are missing, inspect lastError().

OPCUAServerApi.start

Signature

```
OPCUAServerApi.start(cfg: String = ""): void
```

Description

Starts the embedded OPC UA server using JSON configuration.

Arguments

cfg

- type: String
- optional
- JSON configuration

Documented example:

```
{
 "EndpointUrl": "opc.tcp://localhost:4840"
}
```

Returns

- void

Behavior

- clears lastError
- attempts to parse config JSON
- if JSON is invalid:
  - appends warning-like text to lastError
  - continues with default values anyway
- if server already exists:
  - updates its config
- otherwise:
  - creates new MyOPCUAServer
- then attempts startServer()
- writes debug line about success or failure
- connects interruption/shutdown signals afterward

Important implication

Because start() returns void, success must be inferred by:

- checking logs,
- checking lastError(),
- or attempting follow-up operations.

Example

```
InDriver.import("OPCUAServerApi");

OPCUAServerApi.start('{"EndpointUrl":"opc.tcp://localhost:4840"}');
```

Edge cases

- invalid JSON → defaults may still be used
- start failure does not throw; inspect logs and lastError()

Rules

- Always call before addNodes, readMultipleNodes, or writeMultipleNodes.
- If startup is critical, log lastError() after start.

OPCUAServerApi.stop

Signature

```
OPCUAServerApi.stop(): void
```

Description

Stops the embedded OPC UA server.

Returns

- void

Example

onShutdown:

```
OPCUAServerApi.stop();
```

Notes

- Safe cleanup step for tasks that own the server lifecycle.

OPCUAServerApi.writeMultipleNodes

Signature

```
OPCUAServerApi.writeMultipleNodes(nodeCfg: String): Boolean

OPCUAServerApi.writeMultipleNodes(nodeCfg: Array): Boolean
```

Description

Writes values to multiple nodes hosted by the embedded OPC UA server.

Arguments

nodeCfg

- type: String | Array
- JSON array containing node ids and values

Documented example:

[

{ "nodeId": "ns=2;s=Temperature", "value": 22.5 }

]

Returns

- Boolean

Behavior

- fails if server not started
- parses JSON request
- performs write via underlying server
- on failure:
  - stores detailed message in lastError
  - logs fatal debug
  - returns false

Example

```
let ok = OPCUAServerApi.writeMultipleNodes( '[{"nodeId":"ns=2;s=Temperature","value":22.5}]'
);

if (!ok) {
  InDriver.debug(OPCUAServerApi.lastError(), "critical");
}
```

Edge cases

- server not started → false
- invalid JSON → false
- node type/value mismatch → likely server-side write failure

OPC UA server pattern

onStartup:

```
  InDriver.import("OPCUAServerApi");

  OPCUAServerApi.start('{"EndpointUrl":"opc.tcp://localhost:4840"}');

  let ok = OPCUAServerApi.addNodes(
     '[{"nodeId":"ns=2;s=TemperatureSensor","name":"Sensor"}]',
     "Objects"
  );

  if (!ok) {
     InDriver.debug(OPCUAServerApi.lastError(), "critical");
  }
}
```

onHook:

```
  OPCUAServerApi.writeMultipleNodes(
     '[{"nodeId":"ns=2;s=Temperature","value":22.5}]'
  );
```

onShutdown:

```
  OPCUAServerApi.stop();
```

RestApi

RestApi provides HTTP client functionality for JavaScript tasks. It supports:

- defining named HTTP requests,
- sending requests immediately or batching them into one transaction,
- waiting for completion with timeout,
- reading returned body and response headers,
- checking overall success state of the last REST operation.

The API is stateful. It stores:

- request definitions,
- pending batched transactions,
- returned response bodies and headers,
- success state for the last transaction.

RestApi.begin

Signature

```
RestApi.begin(): void
```

Description

Starts a transaction block for collecting multiple REST requests before executing them together.

Behavior

After calling begin():

- previously stored transaction results are cleared,
- previous success state is reset,
- sendRequest(...) does not execute immediately,
- requests are queued internally until commit() or commitWait() is called.

Returns

- void

Example

```
RestApi.begin();

RestApi.sendRequest("Krakow");
```

```
RestApi.sendRequest("Chicago");

RestApi.commitWait();
```

Rules

- Use begin() when you want to execute multiple requests together.
- Use it before mixed workflows when synchronizing with other APIs like Modbus.

RestApi.commit

Signature

```
RestApi.commit(): void
```

Description

Executes all queued requests collected after begin(), but does **not** wait for their completion.

Behavior

- assigns a new internal transaction id
- calculates max timeout from queued requests
- calls each queued request
- stores id on replies to distinguish them from older responses
- finishes the “batch collection” phase by setting internal flag mRequestsBegun = false

Returns

- void

Important

After commit(), you usually need to call:

RestApi.wait();

or

RestApi.wait(timeout);

Example

```
RestApi.begin();

RestApi.sendRequest("Krakow");

RestApi.sendRequest("Chicago");

RestApi.commit();

RestApi.wait();
```

Rules

- commit() only starts execution.
- It does not guarantee data is ready yet.

RestApi.commitWait

Signature

```
RestApi.commitWait(): void
```

Description

Executes all queued requests and waits until all are finished or timeout occurs.

Behavior

Internally equivalent to:

RestApi.commit();

RestApi.wait();

Returns

- void

Example

```
RestApi.begin();

RestApi.sendRequest("Krakow");
```

```
RestApi.sendRequest("Chicago");

RestApi.commitWait();
```

Rules

- This is the preferred method for batched requests when the next code depends on returned data.

RestApi.defineRequest

Signature

```
RestApi.defineRequest(reqName: String, cfg: String): void

RestApi.defineRequest(reqName: String, cfg: Object): void
```

Description

Defines a named REST request configuration that can later be executed by sendRequest(reqName).

Arguments

reqName

- type: String
- logical request name

Examples:

"Krakow"

"Chicago"

"OrderSync"

cfg

- type: String | Object
- JSON configuration describing request

Supported keys from manual and source:

- url

- timeout
- type
- headers
- rawData

Supported type values

Current code supports only:

- "get"
- "post"
- "put"

If another type is given:

- internal request type becomes undefined
- request execution may fail to create a reply

timeout

- type: Number
- optional
- if missing or <= 0, runtime uses 3000 ms default.

headers

- type: Object
- optional
- may include standard headers or raw headers

Recognized standard header keys in source:

- ContentDispositionHeader
- ContentTypeHeader
- ContentLengthHeader
- CookieHeader
- SetCookieHeader
- ETagHeader
- IfMatchHeader
- IfNoneMatchHeader
- LastModifiedHeader
- LocationHeader
- ServerHeader
- UserAgentHeader

Any other key is sent as raw header.

rawData

- type: String | Object | Array
- optional
- used as request payload for POST/PUT
- if object/array, runtime serializes it to JSON text before sending.

Returns

- void

Behavior

- if cfg parses correctly, request definition is stored under reqName
- if JSON parsing fails:
  - debug critical message is logged,
  - definition is not stored,
  - no exception is thrown in the shown implementation.

Example

```
RestApi.defineRequest("Krakow", {
  url: "https://api.openweathermap.org/data/2.5/weather?appid=your_app_id&q=krakow",
  timeout: 5000,
  type: "get",
  headers: {
     ContentTypeHeader: "application/json"
  }
});
Example with payload
RestApi.defineRequest("CreateOrder", {
  url: "https://example.com/api/orders",
  timeout: 5000,
  type: "post",
  headers: {
     ContentTypeHeader: "application/json"
  },
  rawData: {
     id: 123,
     status: "new"
  }
});
```

Rules

- Always use lowercase get, post, put for predictable behavior.
- Prefer object overload in JS for readability.
- Use defineRequest(...) for reusable requests.

RestApi.getData

Signature

```
RestApi.getData(reqName: String): String
```

Description

Returns the response body stored for the named request from the most recent completed REST execution.

Arguments

reqName

- type: String
- request name used in defineRequest(...) / sendRequest(...)

Returns

- String
- response body text
- empty string if no result exists for that name.

Example

```
const data = RestApi.getData("Krakow");

InDriver.debug(data);
```

Rules

- Parse JSON explicitly if the endpoint returns JSON:

```
let obj = JSON.parse(RestApi.getData("Krakow"));
```

RestApi.getRawHeader

Signature

```
RestApi.getRawHeader(reqName: String, headerName: String): String
```

Description

Returns a single response header value for the named request.

Arguments

reqName

- type: String

headerName

- type: String
- exact header name as stored in raw response headers

Returns

- String
- header value
- empty string if request result or header is missing.

Example

```
let contentType = RestApi.getRawHeader("Krakow", "Content-Type");
```

RestApi.getRawHeaderPairs

Signature

```
RestApi.getRawHeaderPairs(reqName: String): String
```

Description

Returns all raw response headers for the named request as serialized JSON text.

Arguments

reqName

- type: String

Returns

- String
- serialized JSON object containing header-value pairs
- empty string if no response data exists.

Example

```
let headersJson = RestApi.getRawHeaderPairs("Krakow");

InDriver.debug(headersJson);
```

RestApi.isSucceeded

Signature

```
RestApi.isSucceeded(): Boolean
```

Description

Returns the success status of the last REST execution block.

Returns

- Boolean

Behavior

The value is backed by internal mTransactionSucceeded. It becomes true when wait(...) finishes before timeout, not necessarily only when every HTTP status code is “successful”. Network reply errors are logged as fatal, but isSucceeded() is not set directly from HTTP status codes.

Important nuance

isSucceeded() mainly indicates:

- the wait cycle completed before timeout

It does **not** mean:

- every endpoint returned HTTP 200
- every response body is semantically correct

Example

```
if (RestApi.isSucceeded()) {

  let data = RestApi.getData("Krakow");

  InDriver.debug(data);

}
```

Rules

- Treat isSucceeded() as “transaction completed in time”.
- If strict validation is required, inspect response body and headers too.

RestApi.sendRequest

Signature

```
RestApi.sendRequest(reqName: String): Boolean

RestApi.sendRequest(reqName: String, jsonPayload: Object): Boolean

RestApi.sendRequest(reqName: String, data: String): Boolean
```

Description

Sends a REST request using either:

- an already defined request name,
- a defined request name plus overridden payload/config.

Overloads

1. sendRequest(reqName)

Uses previously defined request configuration.

2. sendRequest(reqName, jsonPayload)

Serializes the object to JSON text and delegates to string overload.

3. sendRequest(reqName, data)

Uses string payload/config override.

Returns

- Boolean

Behavior for sendRequest(reqName)

- if reqName is found in request definitions:
    - request is executed or queued
    - returns true
- if not found:
    - returns false

Behavior for sendRequest(reqName, data)

- if reqName exists in request definitions:
    - creates request from stored definition, applies override via internal setCfg(...)
    - executes/queues it
    - returns true
- if reqName does not exist:
    - creates a temporary request using reqName and provided config string
    - executes/queues it
    - returns false anyway. This is a subtle but important runtime behavior.

Immediate vs batched execution

If begin() was **not** called:

- request is sent immediately

If begin() **was** called:

- request is added to internal transaction queue
- actual network call happens only at commit() / commitWait() time.

Payload behavior

For POST/PUT:

- payload is taken from rawData
- string stays as text
- object/array is serialized to JSON text.

Example: single immediate request

```
RestApi.defineRequest("Krakow", {
```

```
  url: "https://api.openweathermap.org/data/2.5/weather?appid=your_app_id&q=krakow",
  timeout: 5000,
  type: "get",
  headers: {
     ContentTypeHeader: "application/json"
  }
});


RestApi.sendRequest("Krakow");
RestApi.wait();


if (RestApi.isSucceeded()) {
  const data = RestApi.getData("Krakow");
  InDriver.debug(data);
}
```

Example: batched execution

```
RestApi.begin();
RestApi.sendRequest("Krakow");
RestApi.sendRequest("Chicago");
RestApi.commitWait();
```

Example: override payload

```
RestApi.defineRequest("CreateOrder", {
  url: "https://example.com/api/orders",
  timeout: 5000,
  type: "post",
```

```
  headers: {
    ContentTypeHeader: "application/json"
  }
});


RestApi.sendRequest("CreateOrder", {
  id: 123,
  status: "new"
});
```

Rules

- Prefer defineRequest(...) first, then sendRequest(name).
- Do not rely on true from the overload with payload when request name was not previously defined.
- After immediate request, call wait() before reading results.

RestApi.wait

Signature

```
RestApi.wait(): void

RestApi.wait(timeout: Number): void
```

Description

Waits until all replies from the last commit() finish or until timeout occurs.

Arguments

timeout

- type: Number
- optional
- timeout in milliseconds

If omitted:

- runtime uses mMaxTimeout, which is calculated from queued requests during commit().

Returns

- void

Behavior

If timeout occurs:

- debug critical message is emitted
- JS engine error is thrown:
  "RestApi call timeout <timeout>ms."

Example

```
RestApi.commit();

RestApi.wait();

RestApi.commit();

RestApi.wait(10000);
```

Important

This function does **not** return false on timeout.
Instead, it raises a JS error in the runtime.

Rules

- Use commitWait() when you want a simpler pattern.
- If you use commit() directly, always follow with wait() or wait(timeout) before reading results.

Response and request internals worth documenting

Supported HTTP methods

From current source:

- get
- post
- put only

Default timeout

If timeout <= 0 or missing:

- defaults to 3000 ms.

SSL errors

onsslErrors(...) deletes the reply object and ignores detailed SSL errors in the shown implementation.

Reply storage

On each finished reply, runtime stores:

- raw response headers
- full response body text
  under the request name.

Recommended usage patterns

Pattern 1 — one named request

onStartup:

```
InDriver.import("RestApi");
RestApi.defineRequest("Krakow", {
   url: "https://api.openweathermap.org/data/2.5/weather?appid=your_app_id&q=krakow",
   timeout: 5000,
   type: "get",
   headers: {
      ContentTypeHeader: "application/json"
   }
});

InDriver.installHook(60000);
```

}

onHook:

```
  RestApi.sendRequest("Krakow");
  RestApi.wait();

  if (RestApi.isSucceeded()) {
     const data = RestApi.getData("Krakow");
     InDriver.debug(data);
  }
```

Pattern 2 — parallel requests

```
RestApi.begin();
RestApi.sendRequest("Krakow");
RestApi.sendRequest("Chicago");
RestApi.commitWait();

if (RestApi.isSucceeded()) {
  const krakow = RestApi.getData("Krakow");
  const chicago = RestApi.getData("Chicago");
}
```

Pattern 3 — POST with payload

```
RestApi.defineRequest("CreateOrder", {
  url: "https://example.com/api/orders",
  timeout: 5000,
  type: "post",
  headers: {
     ContentTypeHeader: "application/json"
  }
});

RestApi.sendRequest("CreateOrder", {
  id: 123,
  status: "new"
});
RestApi.wait();
```

Rules

When generating code using RestApi, follow these rules:

1. Import RestApi in onStartup().
2. Use defineRequest(...) for reusable calls.
3. Only assume support for get, post, and put.
4. After immediate sendRequest(...), call wait().
5. After begin(), call commit() + wait() or commitWait().
6. Use getData(...) only after wait/commitWait has completed.
7. Treat isSucceeded() as timeout/transaction completion status, not a full semantic HTTP success guarantee.

ModbusApi

ModbusApi provides Modbus device communication for JavaScript tasks. It supports:

- creating named device connections,
- reading Modbus registers/coils,
- writing Modbus values,
- batching multiple requests in one transaction,
- waiting for completion,
- extracting structured transaction results,
- checking timeout/error state of the last transaction.

The API is stateful. It stores:

- named devices,
- queued transaction requests,
- results of the last transaction,
- success state of the last transaction.

ModbusApi.begin

Signature

```
ModbusApi.begin(): void
```

Description

Starts a Modbus transaction block.

After begin():

- previous transaction data is cleared,
- success flag is reset,
- subsequent readDevice(...) / writeDevice(...) calls are queued instead of being executed immediately.

Returns

- void

Example

```
ModbusApi.begin();
ModbusApi.readDevice("Moxa1", '{"name":"coils","type":"COILS","address":1,"size":8}');
```

```
ModbusApi.commitWait();
```

Rules

- Use begin() when grouping multiple reads/writes.
- Do not call getAllData() before commitWait() or wait().

ModbusApi.commit

Signature

```
ModbusApi.commit(): void
```

Description

Executes all queued Modbus requests collected after begin(), but does not wait for completion.

Behavior

- ends the transaction-collection phase,
- checks device liveness for all registered devices,
- dispatches queued requests to matching devices,
- updates internal mWaitingRequests and mMaxTimeOut,
- if a request references an unknown device, runtime throws JS error: "<deviceName> not found."

Returns

- void

Important

After commit(), you normally call:

ModbusApi.wait();

or

ModbusApi.wait(timeout);

Example

```
ModbusApi.begin();
```

```
ModbusApi.readDevice("Moxa1", '{"name":"coils","type":"COILS","address":1,"size":8}');
ModbusApi.commit();
ModbusApi.wait();
```

ModbusApi.commitWait

Signature

```
ModbusApi.commitWait(): void
```

Description

Executes all queued requests and waits for their completion or timeout.

Behavior

Internally equivalent to:

ModbusApi.commit();

ModbusApi.wait();

Returns

- void

Example

```
ModbusApi.begin();
ModbusApi.readDevice("Moxa1", '{"name":"coils","type":"COILS","address":1,"size":8}');
ModbusApi.commitWait();
```

Rules

- Preferred method for most data acquisition code because it ensures results are ready immediately after the call.

ModbusApi.connectDevice

Signature

```
ModbusApi.connectDevice(deviceName: String, cfg: String): void
```

Description

Creates a named Modbus device connection definition.

If the device name does not already exist, a new JSModbusDevice instance is created and stored.
If the device name already exists, the current code does nothing — it does not overwrite or reconfigure the existing device.

Arguments

deviceName

- type: String
- logical name of the device

cfg

- type: String
- JSON configuration string

Supported connection modes

TCP mode

Supported keys from manual:

- mode: "TCP"
- networkAddress
- networkPort (default 502)
- timeoutMs (default 3000)
- numberOfRetries (default 5)

Minimal TCP example:

```
{
 "mode": "TCP",
 "networkAddress": "192.168.0.22"
}
```

Full TCP example:

```
{
 "mode": "TCP",
 "networkAddress": "192.168.0.22",
 "networkPort": 502,
 "timeoutMs": 3000,
```

```
 "numberOfRetries": 3
}
```

RTU mode

Supported keys from manual:

- mode: "RTU"
- serialPortName
- parity (default 2, even)
- baudRate (default 9600)
- dataBits (default 8)
- stopBits (default 1)
- timeoutMs (default 3000)
- numberOfRetries (default 5)

Minimal RTU example:

```
{
 "mode": "RTU",
 "serialPortName": "COM1"
}
```

Full RTU example:

```
{
 "mode": "RTU",
 "serialPortName": "COM1",
 "baudRate": 9600,
 "parity": 2,
 "dataBits": 8,
 "stopBits": 1,
 "timeoutMs": 3000,
 "numberOfRetries": 3
}
```

Returns

- void

Behavior

- creates device only if deviceName is not already present
- connection lifecycle is maintained by runtime; the manual and source indicate automatic reconnect behavior after disconnect/failure.

Example

```
ModbusApi.connectDevice(
  "IOLogic",
  '{"mode":"TCP","networkAddress":"192.168.0.22"}'
);
```

Edge cases

- repeated connectDevice with same name does not reconfigure existing device in current source
- malformed config may create an invalid device object that later fails during requests

Rules

- Use stable unique deviceName.
- Call once in onStartup().
- Do not assume connectDevice immediately proves communication is healthy; health is exercised during requests.

ModbusApi.getAllData

Signature

```
ModbusApi.getAllData(): String
```

Description

Returns the data collected in the last Modbus transaction for all devices and all requests, serialized as JSON string.

Returns

- String

Behavior

Iterates over all requests in the last transaction and merges their result data into one JSON object.

Example output

```
{
 "Moxa1": {
  "Read": {
    "coilsA": {
```

```
      "1": true,
      "2": false
    }
  }
 }
}
```

Example

```
let dataJson = ModbusApi.getAllData();
let data = JSON.parse(dataJson);
```

Rules

- Parse returned string explicitly with JSON.parse(...) if structured access is needed.

ModbusApi.getDeviceData

Signature

```
ModbusApi.getDeviceData(deviceName: String): String
```

Description

Returns the last transaction data for one selected device, serialized as JSON string.

Arguments

deviceName

- type: String

Returns

- String

Behavior

Filters last transaction requests by deviceName and builds JSON only for that device.

Example

```
let deviceData = JSON.parse(ModbusApi.getDeviceData("Moxa1"));
```

ModbusApi.getDeviceRequestData

Signature

```
ModbusApi.getDeviceRequestData(deviceName: String, registerName: String): String
```

Description

Returns the last transaction data for one selected device and one selected request/register name, serialized as JSON string.

Arguments

deviceName

- type: String

registerName

- type: String
- request name defined in registerCfg.name

Returns

- String

Behavior

Filters last transaction requests by both device name and request/register name.

Example

```
let coilsA = JSON.parse(ModbusApi.getDeviceRequestData("Moxa1", "coilsA"));
```

ModbusApi.getDeviceRequestValue

Signature

```
ModbusApi.getDeviceRequestValue(
  deviceName: String,
  registerName: String,
  addresses: Number[]
```

```
): Number
```

Description

Returns a numeric value composed from selected addresses of the result for the specified device and request.

Arguments

deviceName

- type: String

registerName

- type: String

addresses

- type: Number[]
- Modbus addresses to extract from the selected request result

Returns

- Number (qint64)

Behavior

- searches the last transaction for matching device and register name
- computes a combined value from the selected addresses
- if addresses are invalid for the request range:
  - throws RangeError
  - error text includes expected valid address range.

Example

```
let value = ModbusApi.getDeviceRequestValue("Moxa1", "coilsA", [1, 2, 3]);
```

Edge cases

- invalid addresses → RangeError
- matching request not found → returns 0

Rules

- Use only after the corresponding request has completed.
- Make sure address numbers are inside the requested address span.

ModbusApi.isLastTransactionCompleted

Signature

```
ModbusApi.isLastTransactionCompleted(deviceName: String): Boolean

ModbusApi.isLastTransactionCompleted(): Boolean
```

Description

Checks whether the last Modbus transaction completed without request-level error/timeout.

Arguments

deviceName

- type: String
- optional in overloaded form
- if provided, limits the check to one device

Returns

- Boolean

Behavior

- with deviceName:
    - returns false if any request for that device has error/timeout
- without arguments:
    - returns false if any request in the whole transaction has error/timeout
- otherwise returns true

Example

```
if (!ModbusApi.isLastTransactionCompleted("Moxa1")) {
  InDriver.debug("Moxa1 transaction incomplete", "critical");
}
```

Important nuance

This function checks request completion/error state, while isSucceeded() reflects whether the whole wait cycle completed before timeout. Both can be useful and are not exactly the same thing.

ModbusApi.isSucceeded

Signature

```
ModbusApi.isSucceeded(): Boolean
```

Description

Returns whether the last Modbus transaction completed before the wait timeout.

Returns

- Boolean

Behavior

mTransactionSucceeded becomes true only when wait(...) finishes because all requests completed before the timer expired. If timeout occurs, it remains false.

Example

```
if (ModbusApi.isSucceeded()) {
  let data = ModbusApi.getAllData();
}
```

Rules

- Use this as a coarse transaction success flag.
- For per-device/per-request detail, also inspect isLastTransactionCompleted(...) and returned data.

ModbusApi.readDevice

Signature

```
ModbusApi.readDevice(deviceName: String, registerCfg: String): void
```

Description

Queues or executes a Modbus read request for the specified device.

Arguments

deviceName

- type: String
- name of previously connected device

registerCfg

- type: String
- JSON request definition

Supported keys from manual:

- name — request name
- deviceAddress — slave/device address, required in RTU mode, default 1
- type — one of:
  - "COILS"
  - "DISCRETEINPUTS"
  - "HOLDINGREGISTERS"
  - "INPUTREGISTERS"
- address — starting address
- size — number of coils/registers to read

Example:

```
{
 "name": "coils1",
 "deviceAddress": 1,
 "type": "COILS",
 "address": 1,
 "size": 8
}
```

Returns

- void

Behavior

- if inside a begun transaction (begin() already called):
  - request is appended to transaction queue only
- otherwise:
  - runtime automatically performs:
    - begin()
    - append request

- commitWait()
  so the call behaves like an immediate blocking read.

Example

```
ModbusApi.readDevice(
  "IOLogic",
  '{"name":"coils1","type":"COILS","address":1,"size":8}'
);
```

Rules

- Use inside begin()...commitWait() when batching multiple requests.
- Use direct call only for simple single-request reads.

ModbusApi.wait

Signature

```
ModbusApi.wait(): void

ModbusApi.wait(maxTimeOut: Number): void
```

Description

Waits for completion of the last committed Modbus transaction.

Arguments

maxTimeOut

- type: Number
- optional
- timeout in milliseconds

If omitted:

- runtime uses internally computed mMaxTimeOut, based on device timeouts involved in the current transaction.

Returns

- void

Behavior

- starts event loop
- waits until either:
  - all requests finish
  - timeout occurs
- if finished before timeout:
  - mTransactionSucceeded = true
- if timed out:
  - logs critical timeout message for each request that has error/timeout
  - does not throw in shown code
  - mTransactionSucceeded remains false

Example

```
ModbusApi.commit();
ModbusApi.wait();
ModbusApi.commit();
ModbusApi.wait(10000);
```

Notes for AI

- Prefer commitWait() for simpler code.
- If you use commit(), always follow with wait() before reading data.

ModbusApi.writeDevice

Signature

```
ModbusApi.writeDevice(deviceName: String, registerCfg: String): void
```

Description

Queues or executes a Modbus write request for the specified device.

Arguments

deviceName

- type: String

registerCfg

- type: String

- JSON write definition

Supported keys from manual:

- name
- deviceAddress
- type
    - "COILS"
    - "DISCRETEINPUTS"
    - "HOLDINGREGISTERS"
    - "INPUTREGISTERS"
- address
- data — array of values to write

Example:

```
{
 "name": "coils1",
 "deviceAddress": 1,
 "type": "COILS",
 "address": 1,
 "data": [1, 1, 1]
}
```

Returns

- void

Behavior

- inside transaction block → request is queued
- outside transaction block → automatic immediate pattern:

    begin()

    append request

    commitWait()

Example

```
ModbusApi.writeDevice(
  "Moxa",
  '{"name":"coils1","type":"COILS","address":1,"data":[1,1,1]}'
);
```

Rules

- Prefer batching when writing multiple registers/devices in one cycle.

Recommended usage patterns

Pattern 1 — one immediate read

onHook()

```
ModbusApi.readDevice(
    "IOLogic",
    '{"name":"coils1","type":"COILS","address":1,"size":8}'
);

if (ModbusApi.isSucceeded()) {
    let data = JSON.parse(ModbusApi.getAllData());
    InDriver.debug(data);
}
```

Pattern 2 — batched reads

```
ModbusApi.begin();

ModbusApi.readDevice("Moxa1", '{"name":"coilsA","type":"COILS","address":1,"size":4}');
ModbusApi.readDevice("Moxa1", '{"name":"coilsB","type":"COILS","address":5,"size":4}');
ModbusApi.readDevice("Moxa2", '{"name":"coils","type":"COILS","address":1,"size":8}');

ModbusApi.commitWait();

if (ModbusApi.isSucceeded()) {
  let all = JSON.parse(ModbusApi.getAllData());
}
```

Pattern 3 — mixed transaction with RestApi

```
ModbusApi.begin();
RestApi.begin();

RestApi.sendRequest("Krakow");
ModbusApi.readDevice("IOLogic", '{"name":"coils1","type":"COILS","address":1,"size":8}');
```

```
ModbusApi.commit();
RestApi.commit();

ModbusApi.wait();
RestApi.wait();

const weatherData = RestApi.getData("Krakow");
const modbusData = ModbusApi.getAllData();
```

Notes for code generation

When generating code with ModbusApi, follow these rules:

1. Call InDriver.import("ModbusApi") in onStartup().
2. Call connectDevice(...) once during initialization.
3. Use begin()...commitWait() for multiple requests.
4. After reading returned JSON text, use JSON.parse(...) if structured access is needed.
5. Use isSucceeded() for overall transaction timeout success.
6. Use isLastTransactionCompleted(...) for finer completion/error checks.
7. Use getDeviceRequestValue(...) only when you know the requested addresses are within the register range.

DeviceAPI

DeviceAPI is a generic transaction layer for reading data from external devices connected through:

- serial ports (RTU)
- TCP sockets (TCP)
- serial-over-socket transport (RTUoverTCP)

It provides:

- grouped execution of many device requests,
- transport abstraction over SerialPortApi and TcpSocketApi,
- device result collection,
- timeout handling and reconnection,
- execution statistics.

DeviceAPI is intended to be used through derived classes such as:

- ModbusAPI
- MbusAPI

DeviceAPI.begin

Signature

```
deviceApi.begin(): void
```

Description

Starts a grouped device transaction.

Behavior

- clears queued device configurations
- clears device instances
- clears collected device data
- resets internal device index
- marks the API as being inside a grouped transaction.

Example

```
modbus.begin();
mbus.begin();
```

Rules

- Use begin() before queuing multiple device reads in the same cycle.
- Use one begin() per protocol group.

DeviceAPI.commit

Signature

```
deviceApi.commit(): void
```

Description

Starts execution of queued device requests without blocking until completion.

Behavior

- resets execution status tracking
- starts asynchronous execution by calling the internal execution loop once.

Example

```
modbus.begin();
modbus.execute(req1);
modbus.execute(req2);
modbus.commit();
```

Rules

- Use commit() when execution may continue in parallel with another API, for example mbus.commit().
- After commit(), call waitForDevices(...) or another wait step before reading results.

DeviceAPI.commitWait

Signature

```
deviceApi.commitWait(timeout: Number = 0): void
```

Description

Executes queued device requests and waits until they finish or the timeout is reached.

Arguments

timeout

- type: Number
- optional
- timeout in milliseconds
- 0 means no explicit external timeout limit

Behavior

- repeatedly runs the internal execution loop
- waits in short intervals
- finishes by checking connections and leaving grouped mode.

Example

```
ModbusApi.begin();
MacREJ5R_readVQTAsyncReq("BB_Pion2_Gaz", 1);
ModbusApi.commitWait();
```

Rules

- Use commitWait() when data is needed immediately after the call.
- Prefer this for simple one-protocol grouped reads.

DeviceAPI.debugModeEnabled

Signature

```
deviceApi.debugModeEnabled(enabled: Boolean): void
```

Description

Enables or disables verbose device debug logging.

Behavior

When enabled, the API logs:

- executed device configs,
- sent telegrams,

- received telegrams.

Example

```
modbus.debugModeEnabled(false);
mbus.debugModeEnabled(false);
```

Rules

- Use true during commissioning and troubleshooting.
- Use false in production unless detailed communication logs are needed.

DeviceAPI.execute

Signature

```
deviceApi.execute(deviceConfig: Object): void
```

Description

Queues a device request for execution.

Arguments

deviceConfig

- type: Object
- device request configuration

Required transport fields

Every request should provide:

- name
- commPort
- mode

Depending on protocol, it also needs:

- fnc for ModbusAPI
- device for MbusAPI

Behavior

- sets execStatus = "none"

- if no grouped transaction is active:
  - automatically starts begin()
  - adds the request
  - immediately starts commit()
- otherwise:
  - only appends the request to the current group.

Example

```
modbus.execute(ar252_ValuesModbusRequest("ar252_1", 1, "socket_Pion2_Com1",
"RTUoverTCP") );
```

Rules

- In grouped workflows, call begin() first and then multiple execute(...).
- Do not read results before commitWait() or waitForDevices(...).

DeviceAPI.getDeviceData

Signature

```
deviceApi.getDeviceData(device: String = ""): Object
```

Description

Returns collected device data.

Arguments

device

- type: String
- optional
- device name

Returns

Object

Behavior

- with device → returns data for one device
- without device → returns all device data from the current transaction.

Example

```
let modbusJSON = modbus.getDeviceData();
let mbusJSON = mbus.getDeviceData();
```

Rules

- This is the main result accessor after a transaction finishes.
- Device data is stored under device.name, and then usually under item, defaulting to "values".

DeviceAPI.getStatistics

Signature

```
deviceApi.getStatistics(): Object
```

Description

Returns execution statistics grouped by communication port and device name.

Returned fields

Typical statistics include:

- TotalCnt
- OKCnt
- NOKCnt
- ErrorRatio
- lastExecTime

Example

```
let stats = modbus.getStatistics();
InDriver.debug(stats);
```

DeviceAPI.waitForDevices

Signature

```
deviceApi.waitForDevices(otherDevices: Array = [], timeout: Number = 0): void
```

Description

Waits until the current API instance and optional additional API instances finish execution.

Arguments

otherDevices

- type: Array
- optional
- list of other DeviceAPI-compatible objects

timeout

- type: Number
- optional
- timeout in milliseconds

Behavior

- waits until all passed APIs finish
- then finalizes the current transaction and checks connections.

Example

```
modbus.commit();
mbus.commit();
modbus.waitForDevices([mbus]);
```

This matches your usage pattern.

Rules

- Use this when ModbusAPI and MbusAPI run in parallel in the same hook.
- Prefer this over separate waits when both protocol groups share one cycle.

DeviceAPI transport model

Description

DeviceAPI selects transport based on deviceConfig.mode:

- TCP and RTUoverTCP → TcpSocketApi
- RTU → SerialPortApi

Requests are sent through:

- write(...) in async mode
- writeAndWait(...) in sync mode.

On timeout:

- the device request gets timeout status
- statistics are updated
- transport is reopened:
    - TcpSocketApi.connect(port)
    - or SerialPortApi.open(port)

Rules

- Always provide valid commPort and mode.
- Use frameSendDelay when the device or converter requires frame spacing.
- Use timeout when devices may be slow.

MbusAPI

MbusAPI is a device-level API for M-Bus meters. It extends DeviceAPI and adds:

- M-Bus telegram handling,
- device-specific read flows,
- M-Bus meter parsers for selected supported devices.

Supported device types in the attached scripts:

- RelayPadPulsM4
- ItronCyble
- KamstrupMultical

MbusAPI.exec

Signature

```
mbusApi.exec(deviceConfig: Object): Boolean
```

Description

Creates the correct M-Bus device handler and starts the request sequence.

Arguments

deviceConfig

- type: Object
- M-Bus device configuration

Required fields

- name
- device
- commPort
- mode

Common M-Bus-specific fields:

- address
- deviceId
- frameSendDelay
- item

Supported device values

- "RelayPadPulsM4"
- "ItronCyble"
- "KamstrupMultical"

Returns

Boolean

- true → supported device type
- false → unsupported device type

Example

```
mbus.execute({
  name: "BB_LakPion2_Woda",
  device: "KamstrupMultical",
  fnc: "readData",
  item: "values",
  deviceId: 0,
  address: 1,
  frameSendDelay: 100,
  commPort: "socket_Pion2_Com2",
  mode: "TCP"
});
```

This matches your current usage.

Rules

- Use device to select the meter library.
- Use exact supported names.
- fnc is not the main dispatcher here; device is.

MbusAPI.receive

Signature

```
mbusApi.receive(data: Object): void
```

Description

Processes received transport data and forwards it to the correct device instance.

Behavior

- normalizes received bytes
- finds the device instance by request device name
- calls its specific parser/receive logic.

Rules

- Route incoming onMessage() payload to mbus.receive(data) after parsing JSON from InDriver.messageData().
- Use this as the main M-Bus response entry point.

M-Bus device flow

Description

The attached M-Bus device classes follow a common flow:

1. send() creates and sends initial telegram
2. if address is 0, device initialization telegram is sent first
3. receive() validates the reply
4. if the reply is correct:
    - accept(...) is called

- parsed values are appended to device data.

Rules

- For many M-Bus devices, first communication may include initialization when address === 0.
- Use deviceId when initialization by secondary addressing is required.

ModbusAPI

ModbusAPI is a device-level API for Modbus communication. It extends DeviceAPI and supports:

- Modbus RTU request generation
- Modbus TCP transport conversion
- parsing read and write responses
- standard Modbus function groups.

ModbusAPI.exec

Signature

```
modbusApi.exec(deviceConfig: Object): Boolean
```

Description

Creates the correct Modbus request handler and starts the request.

Required fields

- name
- fnc
- slaveAddress
- startAddress
- commPort
- mode

Additional fields depend on function:

- quantity
- value
- state

- frameSendDelay
- item

Supported fnc values

- ReadCoils
- ReadDiscreteInputs
- ReadHoldingRegisters
- ReadInputRegisters
- WriteSingleCoil
- WriteSingleRegister
- WriteMultipleCoils
- WriteMultipleRegisters

Returns

Boolean

- true → supported function
- false → unsupported function

Example

```
modbus.execute({
  name: "BB_LakierniaPion2_Cieplo",
  item: "values",
  fnc: "ReadHoldingRegisters",
  slaveAddress: 94,
  startAddress: 0,
  quantity: 54,
  frameSendDelay: 20,
  commPort: "socket_Pion2_Com1",
  mode: "RTUoverTCP"
});
```

This matches the helper-generated requests used in your script.

Rules

- Use exact fnc names.
- Use helper request builders when available.

ModbusAPI.receive

Signature

```
modbusApi.receive(data: Object): void
```

Description

Processes received Modbus data and forwards it to the correct request instance.

Behavior

- normalizes received bytes
- if mode is TCP, TCP frame is converted/processed before Modbus parsing
- if mode is RTU, RTU frame is validated directly
- only valid responses are forwarded to the request instance.

Rules

- Use this as the central Modbus response entry point in onMessage().
- Do not parse raw transport bytes directly in application logic if ModbusAPI is already used.

ModbusAPI transport behavior

Description

For mode: "TCP", request frames are converted from RTU format into Modbus TCP format before sending. For RTU responses, CRC validation is performed; for TCP responses, the MBAP header is stripped before Modbus response parsing.

Rules

- Use the same logical request object for RTU and TCP where possible.
- Change only mode and commPort to adapt transport.

Device libraries

The attached device libraries build reusable device-specific logic on top of ModbusAPI or MbusAPI. They follow a consistent pattern:

- request builder function

- parser function
- optional older direct ModbusApi.readDevice(...) helper variants.

This is the recommended level for AI-generated code, because protocol details stay separate from business logic.

AR252 library

Description

The apar.js helper provides support for the AR252 device. It includes:

- ar252_ValuesModbusRequest(...)
- ar252_ParseValuesModbus(...)

Returned values

The parser returns:

- rH
- t
- aH
- dPC

Example

```
modbus.execute(

  ar252_ValuesModbusRequest("ar252_1", 1, "socket_Pion2_Com1", "RTUoverTCP")

);



let modbusJSON = modbus.getDeviceData();

if ("ar252_1" in modbusJSON) {

  let d = ar252_ParseValuesModbus(modbusJSON.ar252_1.values);

}
```

Rules

- Use the helper request builder instead of manually writing register definitions.
- Use the parser function on modbus.getDeviceData()[deviceName].values.

CMK03 library

Description

The common.js helper contains support for CMK03. It provides request builders such as:

- cmk03_TValuesModbusRequest(...)
- cmk03_QValuesModbusRequest(...)
- cmk03_VValuesModbusRequest(...)
- cmk03_VbValuesModbusRequest(...)

and parsers such as:

- cmk03_ParseTValuesModbus(...)
- cmk03_ParseQValuesModbus(...)
- cmk03_ParseVValuesModbus(...)

Returned values

Typical parsed values:

- T
- Q
- V

Rules

- Use these helpers when CMK register ordering matters.
- Do not replace them with generic float parsing unless device byte order is known.

Multical Modbus library

Description

The kamstrup.js helper includes:

- multical_KeyValuesModbusRequest(...)
- multical_ParseKeyValuesModbus(...)

Returned values

The parser returns a normalized object containing fields such as:

- V1
- ActualPower
- T1
- T2
- dT
- E1
- Vol1

Example

```
modbus.execute(
  multical_KeyValuesModbusRequest(
     "BB_LakierniaPion2_Cieplo",
     94,
     "socket_Pion2_Com1",
     "RTUoverTCP"
  )
);
```

This matches your script.

MacREJ5R library

Description

The plum.js helper provides support for MacREJ5R. It includes:

- macrej5_QTValuesModbusRequest(...)
- macrej5_VValuesModbusRequest(...)
- macrej5_ParseQTValuesModbus(...)
- macrej5_ParseVValuesModbus(...)
- combined async helpers like MacREJ5R_readVQTAsyncReq(...) and MacREJ5R_readVQTAsyncProcess(...)

Returned values

Typical parsed values:

- Q
- T
- V

Example

ModbusApi.begin();

MacREJ5R_readVQTAsyncReq("BB_Pion2_Gaz", 1);

ModbusApi.commitWait();

let data = JSON.stringify(MacREJ5R_readVQTAsyncProcess("BB_Pion2_Gaz"));

This matches your current pattern.

Rules

- Use the async helpers when working with the built-in low-level ModbusApi.
- Use request builder + parser helpers when working with new ModbusAPI() grouped transactions.

M-Bus device libraries

Description

The mbusAPI.js file contains three device classes:

- RelayPadPulsM4
- ItronCyble
- KamstrupMultical

Returned values

- RelayPadPulsM4 → { v: volume }
- ItronCyble → { v: volume }
- KamstrupMultical → { summary, fields } where summary contains normalized values such as:
  - volume_m3
  - flow_m3_h
  - t1_C
  - t2_C
  - dT_C
  - power_kW
  - timestamp
  - energy_GJ

Example

```
mbus.execute({
  name: "BB_LakPion2_Woda",
  device: "KamstrupMultical",
  fnc: "readData",
  item: "values",
  deviceId: 0,
  address: 1,
  frameSendDelay: 100,
  commPort: "socket_Pion2_Com2",
  mode: "TCP"
});
```

Rules

- For Kamstrup M-Bus, prefer values.summary for business logic.
- Use values.fields only when detailed raw quantities are needed.

Recommended usage patterns

Pattern 1 — startup initialization

onStartup:

```

  InDriver.import("SerialPortApi");
  InDriver.import("TcpSocketApi");
  InDriver.import("ModbusApi");

  InDriver.loadScript("deviceLibrary/deviceAPI.js");
  InDriver.loadScript("deviceLibrary/modbusAPI.js");
  InDriver.loadScript("deviceLibrary/mbusAPI.js");
  InDriver.loadScript("deviceLibrary/kamstrup.js");
  InDriver.loadScript("deviceLibrary/plum.js");
  InDriver.loadScript("deviceLibrary/apar.js");

  TcpSocketApi.connect("socket_Pion2_Com1", {
     address: "10.4.107.186",
     port: 4001,
     timeout: 5000,
     mode: "Buffer"
  });

  TcpSocketApi.connect("socket_Pion2_Com2", {
```

```
    address: "10.4.107.186",
    port: 4002,
    timeout: 5000,
    mode: "Buffer"
});

var modbus = new ModbusAPI();
var mbus = new MbusAPI();

modbus.debugModeEnabled(false);
mbus.debugModeEnabled(false);
```

Pattern 2 — grouped Modbus and M-Bus execution

onHook:

```
modbus.begin();
mbus.begin();

modbus.execute(
    multical_KeyValuesModbusRequest(
        "BB_LakierniaPion2_Cieplo",
        94,
        "socket_Pion2_Com1",
        "RTUoverTCP"
    )
);

modbus.execute(
    ar252_ValuesModbusRequest("ar252_1", 1, "socket_Pion2_Com1", "RTUoverTCP")
);

mbus.execute({
    name: "BB_LakPion2_Woda",
    device: "KamstrupMultical",
    fnc: "readData",
    item: "values",
    deviceId: 0,
    address: 1,
    frameSendDelay: 100,
    commPort: "socket_Pion2_Com2",
```

```
    mode: "TCP"
  });

  modbus.commit();
  mbus.commit();
  modbus.waitForDevices([mbus], 5000);

  let modbusJSON = modbus.getDeviceData();
  let mbusJSON = mbus.getDeviceData();
```

Pattern 3 — insert parsed data into TSAPI storage

```
if ("ar252_1" in modbusJSON) {

  let d = ar252_ParseValuesModbus(modbusJSON.ar252_1.values);

  InDriver.sqlExecute(

    "LocalPGSQL",

    `select tsapiinsert('public', 'measurements', 'BB_ar252_1', '${ts.toISOString()}',
'${JSON.stringify(d)}')`

  );

}
```

Rules for code generation

When generating code with DeviceAPI, MbusAPI, ModbusAPI, and the device helper libraries, follow these rules:

1. Import and load everything explicitly

```
InDriver.import("SerialPortApi");
InDriver.import("TcpSocketApi");
InDriver.import("ModbusApi");
```

```
InDriver.loadScript("deviceLibrary/deviceAPI.js");
InDriver.loadScript("deviceLibrary/modbusAPI.js");
InDriver.loadScript("deviceLibrary/mbusAPI.js");
```

2. Use grouped execution for many devices

begin → execute → commit/commitWait → getDeviceData

3. For parallel Modbus + M-Bus cycles

```
modbus.commit();
mbus.commit();
modbus.waitForDevices([mbus]);
```

4. Always provide transport fields

- commPort
- mode
- often frameSendDelay
- often timeout

5. For ModbusAPI, use exact fnc values

- ReadCoils
- ReadDiscreteInputs
- ReadHoldingRegisters
- ReadInputRegisters
- WriteSingleCoil
- WriteSingleRegister
- WriteMultipleCoils
- WriteMultipleRegisters

6. For MbusAPI, use exact device values

- RelayPadPulsM4
- ItronCyble
- KamstrupMultical

7. Prefer helper request builders and parsers

Use:

- ar252_ValuesModbusRequest(...)
- multical_KeyValuesModbusRequest(...)
- macrej5_QTValuesModbusRequest(...)
- and matching parse helpers

8. Keep protocol logic separate from business logic

- protocol layer → helper library / ModbusAPI / MbusAPI
- business layer → choose fields and insert into TSAPI

9. Use getDeviceData() as the canonical result source

SMTPApi

SmtpApi provides SMTP email functionality for JavaScript tasks in InDriver.

It supports:

- configuring SMTP server connection,
- composing email messages,
- sending emails immediately or in batches,
- transactional execution using begin() / commit(),
- error handling via lastError().

The API is **stateful**, similar to RestApi and ModbusApi.

Execution modes

1. Immediate mode

```
SmtpApi.send(...)
```

- email is sent immediately

2. Transaction mode

```
SmtpApi.begin();
SmtpApi.send(...);
SmtpApi.send(...);
SmtpApi.commit();
```

- emails are queued and sent together

SmtpApi.begin

Signature

```
SmtpApi.begin(): void
```

Description

Starts a new email transaction.

Behavior

- clears previously queued emails
- resets error state
- switches API to batch mode

Returns

- void

Example

```
SmtpApi.begin();
```

Rules

- Always call begin() before batching multiple emails.

SmtpApi.commit

Signature

SmtpApi.commit(): Boolean

Description

Sends all emails queued since the last begin() call.

Returns

Boolean

Meaning

- true → all emails sent successfully
- false → at least one email failed

Behavior

- iterates through internal email queue
- sends each message sequentially
- stores error message if any failure occurs

Example

```
SmtpApi.begin();

SmtpApi.send({...});
SmtpApi.send({...});

if (!SmtpApi.commit()) {
  InDriver.debug(SmtpApi.lastError(), "critical");
}
```

Rules

- Always check return value of commit() in production code.

SmtpApi.configure

Signature

```
SmtpApi.configure(cfg: String): Boolean

SmtpApi.configure(cfg: Object): Boolean
```

Description

Configures SMTP server settings for subsequent send() operations.

Arguments

cfg

- type: String | Object
- JSON configuration

Supported fields

Field	Type	Required	Description
host	String	✅	SMTP server hostname
port	Number	❌	SMTP port
username	String	❌	authentication username
password	String	❌	authentication password
ssl	Boolean	❌	enable SSL/TLS

Returns

Boolean

- true → configuration is valid
- false → invalid configuration

Behavior

- stores configuration internally
- applies to all subsequent send() calls

Example

```
SmtpApi.configure({
  host: "smtp.company.com",
  port: 587,
  username: "user",
  password: "pass",
```

```
  ssl: true
});
```

Rules

- Must be called before send() unless configuration is already defined.

SmtpApi.send

Signature

SmtpApi.send(mail: String): Boolean

SmtpApi.send(mail: Object): Boolean

Description

Adds an email to the queue or sends it immediately depending on execution mode.

Arguments

mail

- type: String | Object
- JSON email definition

Supported fields

Field	Type	Required	Description
from	String	✅	sender email
to	String / Array	✅	recipient(s)
subject	String	❌	email subject

body	String	❌	email body
attachments	Array	❌	list of file paths

Returns

Boolean

Meaning

- true → email accepted (queued or sent)
- false → validation or configuration error

Behavior

Immediate mode

- email is sent immediately

Transaction mode (begin() used)

- email is added to internal queue
- actual sending happens during commit()

Example — immediate

```
SmtpApi.send({
  from: "system@company.com",
  to: "admin@company.com",
  subject: "Alert",
  body: "Machine stopped"
});
```

Example — batch

```
SmtpApi.begin();

SmtpApi.send({
  from: "system@company.com",
  to: "a@company.com",
  subject: "Report A"
```

```
});

SmtpApi.send({
  from: "system@company.com",
  to: "b@company.com",
  subject: "Report B"
});

SmtpApi.commit();
```

Rules

- send() does not guarantee delivery in transaction mode.
- Delivery is finalized only after commit().

SmtpApi.lastError

Signature

```
SmtpApi.lastError(): String
```

Description

Returns the last SMTP error message.

Returns

String

- empty string → no error
- non-empty string → error description

Example

```
if (!SmtpApi.commit()) {
  InDriver.debug(SmtpApi.lastError(), "critical");
}
```

Typical errors

- connection failure
- authentication failure

- invalid recipient
- timeout
- SMTP server rejection

Recommended usage patterns

Pattern 1 — simple alert

```
SmtpApi.configure({
  host: "smtp.company.com",
  from: "indriver@company.com"
});

SmtpApi.send({
  to: "maintenance@company.com",
  subject: "ALERT",
  body: "Machine stopped"
});
```

Pattern 2 — batch notifications

```
SmtpApi.configure({
  host: "smtp.company.com",
  from: "indriver@company.com"
});

SmtpApi.begin();

for (let user of users) {
  SmtpApi.send({
    to: user.email,
    subject: "Daily report",
    body: user.report
  });
}

if (!SmtpApi.commit()) {
  InDriver.debug(SmtpApi.lastError(), "critical");
}
```

Notes for code generation

When generating code using SmtpApi, follow these rules:

1. Always configure SMTP first:

SmtpApi.configure(...)

2. Use transaction mode for multiple emails:

begin → send → commit

3. Always check for errors:

if (!SmtpApi.commit()) → lastError()

4. Do not assume:

- retry mechanism exists
- persistent SMTP connection
- async execution

TsApi

TsApi provides time-series aggregation support for InDriver tasks. It is built around an internal **aggregator engine** defined in C++, while the SQL package (tsapi.sql) defines the **standard database structures and helper SQL functions** used by the engine. In practice, TsApi is responsible for creating and maintaining aggregation tables derived from a raw JSON time-series table.

Standard TSAPI database model

The SQL package installs a set of helper functions and standard table structures. The most important pieces are:

Raw time-series table

Created by tsapicreatetable(schema_, name_):

- source text
- ts timestamp with time zone
- data jsonb

This is the base table for raw data collection. Each row represents one source at one timestamp with JSON payload in data.

shelly
Columns (3)
source
timestamp
data

	source text	ts timestamp with time zone	data jsonb
1	Shelly	2023-11-05 14:07:00+00	[{"Shelly": {"power1": 696.41, "power2": 264.58, "power3": 8.4, "energy1": 1699654, "energy2": 1734279.8, "energy3": 542975.1, "current1": 1.9, "current2": 1.29, "current3": 0.16, "voltage1": 242.01, "voltage2": 239.93, "voltage3": 239.7, "
2	Shelly	2023-11-05 14:06:50+00	[{"Shelly": {"power1": 356.33, "power2": 249.93, "power3": 7.59, "energy1": 1699654, "energy2": 1734279.8, "energy3": 542975.1, "current1": 1.65, "current2": 1.25, "current3": 0.16, "voltage1": 241.92, "voltage2": 240.36, "voltage3": 239.
3	Shelly	2023-11-05 14:06:40+00	[{"Shelly": {"power1": 356.33, "power2": 249.93, "power3": 7.59, "energy1": 1699654, "energy2": 1734279.8, "energy3": 542975.1, "current1": 1.65, "current2": 1.25, "current3": 0.16, "voltage1": 241.92, "voltage2": 240.36, "voltage3": 239.
4	Shelly	2023-11-05 14:06:30+00	[{"Shelly": {"power1": 356.33, "power2": 249.93, "power3": 7.59, "energy1": 1699654, "energy2": 1734279.8, "energy3": 542975.1, "current1": 1.65, "current2": 1.25, "current3": 0.16, "voltage1": 241.92, "voltage2": 240.36, "voltage3": 239.
5	Shelly	2023-11-05 14:06:20+00	[{"Shelly": {"power1": 356.33, "power2": 249.93, "power3": 7.59, "energy1": 1699654, "energy2": 1734279.8, "energy3": 542975.1, "current1": 1.65, "current2": 1.25, "current3": 0.16, "voltage1": 241.92, "voltage2": 240.36, "voltage3": 239.
6	Shelly	2023-11-05 14:06:10+00	[{"Shelly": {"power1": 356.78, "power2": 255.1, "power3": 7.46, "energy1": 1699654, "energy2": 1734279.8, "energy3": 542975.1, "current1": 1.66, "current2": 1.27, "current3": 0.16, "voltage1": 241.13, "voltage2": 238.98, "voltage3": 238.0
7	Shelly	2023-11-05 14:06:00+00	[{"Shelly": {"power1": 378.55, "power2": 271.69, "power3": 7.82, "energy1": 1699647.8, "energy2": 1734275.6, "energy3": 542975, "current1": 1.81, "current2": 1.33, "current3": 0.16, "voltage1": 241.11, "voltage2": 240.13, "voltage3": 239.
8	Shelly	2023-11-05 14:05:50+00	[{"Shelly": {"power1": 377.28, "power2": 253.67, "power3": 7.23, "energy1": 1699647.8, "energy2": 1734275.6, "energy3": 542975, "current1": 1.81, "current2": 1.27, "current3": 0.16, "voltage1": 241.29, "voltage2": 239.52, "voltage3": 239.
9	Shelly	2023-11-05 14:05:40+00	[{"Shelly": {"power1": 377.28, "power2": 253.67, "power3": 7.23, "energy1": 1699647.8, "energy2": 1734275.6, "energy3": 542975, "current1": 1.81, "current2": 1.27, "current3": 0.16, "voltage1": 241.29, "voltage2": 239.52, "voltage3": 239.
10	Shelly	2023-11-05 14:05:30+00	[{"Shelly": {"power1": 377.28, "power2": 253.67, "power3": 7.23, "energy1": 1699647.8, "energy2": 1734275.6, "energy3": 542975, "current1": 1.81, "current2": 1.27, "current3": 0.16, "voltage1": 241.29, "voltage2": 239.52, "voltage3": 239.

Aggregation table

Created by tsapicreateaggregationtable(schema_, name_):

(source text, ts timestamp with time zone, data jsonb, extras jsonb)

This is the standard structure used by the aggregation engine. The extras column stores aggregation metadata/statistics. In the SQL package, helper functions such as tsapiselectaggdata expose extras->>'status', which confirms that extras is part of the standard aggregation contract.

- shelly_15minutes
  - Columns (4)
    - source
    - ts
    - data
    - extras

	source text	ts timestamp with time zone	data jsonb	extras jsonb
1	Shelly	2023-11-06 13:43:00+00	[{"Shelly": {"power1": 34.19, "power2": 93.55, "power3": 7.63, "energy1": 1705270.4, "energy2": 1740868....	{"status": "real"}
2	Shelly	2023-11-06 13:00:00+00	[{"Shelly": {"power1": 34.02, "power2": 194.73, "power3": 7.46, "energy1": 1705241.5, "energy2": 174075...	{"status": "real", "statistics": [{"Shelly": {"power1": {"avg": 33.60666666666666, "max": 34.02, "min": 32.93, "delta": 0.169999
3	Shelly	2023-11-06 12:00:00+00	[{"Shelly": {"power1": 33.63, "power2": 187.6, "power3": 7.83, "energy1": 1705201.2, "energy2": 1740569....	{"status": "real", "statistics": [{"Shelly": {"power1": {"avg": 33.3575, "max": 33.85, "min": 32.13, "delta": 0.3900000000000057
4	Shelly	2023-11-06 11:00:00+00	[{"Shelly": {"power1": 33.69, "power2": 370.64, "power3": 7.56, "energy1": 1705160.3, "energy2": 174038...	{"status": "real", "statistics": [{"Shelly": {"power1": {"avg": 33.6925, "max": 33.71, "min": 33.67, "delta": -0.0599999999999517
5	Shelly	2023-11-06 10:00:00+00	[{"Shelly": {"power1": 34.35, "power2": 120.03, "power3": 7.84, "energy1": 1705121.5, "energy2": 174019...	{"status": "real", "statistics": [{"Shelly": {"power1": {"avg": 33.894999999999996, "max": 34.35, "min": 33.52, "delta": -0.66000
6	Shelly	2023-11-06 09:00:00+00	[{"Shelly": {"power1": 33.51, "power2": 115.12, "power3": 7.95, "energy1": 1705074.7, "energy2": 174006...	{"status": "real", "statistics": [{"Shelly": {"power1": {"avg": 33.7875, "max": 34.06, "min": 33.51, "delta": 0.840000000000034},
7	Shelly	2023-11-06 08:00:00+00	[{"Shelly": {"power1": 33.23, "power2": 135.7, "power3": 7.83, "energy1": 1705023.7, "energy2": 1739941....	{"status": "real", "statistics": [{"Shelly": {"power1": {"avg": 33.06, "max": 33.51, "min": 32.26, "delta": 0.2800000000000114}, "
8	Shelly	2023-11-06 07:00:00+00	[{"Shelly": {"power1": 109.28, "power2": 110.74, "power3": 7.41, "energy1": 1704940.4, "energy2": 17398...	{"status": "real", "statistics": [{"Shelly": {"power1": {"avg": 76.985, "max": 109.28, "min": 33.56, "delta": -76.0500000000001}, "
9	Shelly	2023-11-06 06:00:00+00	[{"Shelly": {"power1": 145.18, "power2": 254.15, "power3": 8.65, "energy1": 1704756.4, "energy2": 17396...	{"status": "real", "statistics": [{"Shelly": {"power1": {"avg": 140.3725, "max": 145.18, "min": 134.75, "delta": -35.900000000000
10	Shelly	2023-11-06 05:00:00+00	[{"Shelly": {"power1": 178.96, "power2": 181.54, "power3": 43.39, "energy1": 1704581.3, "energy2": 1739...	{"status": "real", "statistics": [{"Shelly": {"power1": {"avg": 146.2625, "max": 178.96, "min": 112.54, "delta": -33.78}, "power2": {

Aggregation Table column extras [JSON]

```
[
  {
    "Shelly": {
      "power1": 435.74,
      "power2": 592.73,
      "power3": 52.39,
      "energy1": 1706370.5,
      "energy2": 1742011.3,
      "energy3": 545847.8,
      "current1": 1.98,
      "current2": 2.61,
      "current3": 0.39,
      "voltage1": 240.39,
      "voltage2": 238.23,
      "voltage3": 237.82,
      "power_total": 1080.8600000000001,
      "energy_total": 3994229.5999999996
    }
  }
]
```

Aggregation Table column data [JSON]

```
{
  "status": "real",
  "statistics": [
    {
      "Shelly": {
        "power1": {
          "avg": 683.0999999999999,
          "max": 465.31,
          "min": 435.74,
          "delta": 29.409999999999968
        },
        "power2": {
          "avg": 752.745,
          "max": 618.58,
          "min": 294.18,
          "delta": -298.55
        },
        "power3": {
          "avg": 79.865,
          "max": 54.55,
          "min": 52.39,
          "delta": 0.3999999999999986
        },
        "energy1": {
          "avg": 2559695.05,
          "max": 1706545.2,
          "min": 1706370.5,
          "delta": 174.69999999995343
        },
        "energy2": {
          "avg": 2613194.65,
          "max": 1742226.7,
          "min": 1742011.3,
          "delta": 215.39999999990687
        },
        "energy3": {
          "avg": 818788.55,
          "max": 545868.6,
          "min": 545847.8,
          "delta": 20.79999999993015
        },
```


Installed helper SQL functions

The package also installs functions such as:

- tsapicreatetable
- tsapicreateaggregationtable
- tsapideletetable
- tsapiinsert
- tsapiinsertvalues
- tsapiselectdistinctsources
- tsapiselect
- tsapiselectfirst

- tsapiselectlast
- tsapiselectlastn
- tsapiselectprevious
- tsapiselectagg
- tsapiselectaggdata
- tsapiselectaggwheretsin
- tsapiselectvariablefromagg
- tsapiinsertagg

tsapiinstall

Signature

```
tsapiinstall(schema_ text) returns void
```

Description

Installs the TSAPI package in the selected schema. This function registers the API in `inapi_installedapis` and creates the TSAPI SQL functions such as table creation, inserts, and selects.

Arguments

`schema_`

- target schema where TSAPI should be installed

Returns

- `void`

Example

select tsapiinstall('public');

Rules

- Run this once per schema where TSAPI functions should be available.
- Use it after the SQL server is configured for InDriver.

---

tsapiuninstall

Signature

```
tsapiuninstall(schema_ text) returns void
```

Description

Uninstalls TSAPI from the selected schema. It removes the TSAPI entry from `inapi_installedapis` and drops the installed TSAPI functions.

Arguments

`schema_`

- schema from which TSAPI should be removed

Returns

- `void`

Example

```
select tsapiuninstall('public');
```

Rules

- Use this only when TSAPI functions are no longer needed in the schema.
- Uninstalling the package removes the SQL API functions, not necessarily the user tables that were already created. This behavior should be handled separately in database administration workflows.

---

tsapiversion

Signature

```
tsapiversion() returns table (ver text, dbtype text)
```

Description

Returns the installed TSAPI version and the target database type. In the attached script, the reported values are `ver = '2.0'` and `dbtype = 'QPSQL'`.

Returns

- `ver`
- `dbtype`

Example

select * from tsapiversion();

Rules

- Use this to verify that TSAPI is installed and to confirm the database-specific package version.

---

tsapicreatetable

Signature

```
tsapicreatetable(schema_ text, name_ text) returns void
```

Description

Creates a standard raw time-series table if it does not already exist. The created table contains columns `source`, `ts`, and `data`, and TSAPI also creates indexes for `ts` and `source`.

Arguments

`schema_`

- target schema

`name_`

- table name

Returns

- `void`

Example

select tsapicreatetable('public', 'measurements');

Rules

- Use this for raw time-series data.
- The resulting table is suitable for inserts through `tsapiinsert(...)` and `tsapiinsertvalues(...)`.

---

tsapicreateaggregationtable

Signature

```
tsapicreateaggregationtable(schema_ text, name_ text) returns void
```

Description

Creates a standard aggregation table if it does not already exist. The created table contains `source`, `ts`, `data`, and `extras`, and TSAPI also creates indexes for `ts` and `source`.

Arguments

`schema_`

- target schema

`name_`

- aggregation table name

Returns

- `void`

Example

```
select tsapicreateaggregationtable('public', 'measurements_1h');
```

Rules

- Use this for aggregation outputs rather than raw measurements.
- Use `tsapiinsertagg(...)` to bulk insert rows that include both `data` and `extras`.

---

tsapideletetable

Signature

```
tsapideletetable(schema_ text, name_ text) returns void
```

Description

Drops a table if it exists, using `CASCADE`.

Arguments

`schema_`

- target schema

`name_`

- table name

Returns

- `void`

Example

```
select tsapideletetable('public', 'measurements_old');
```

Rules

- Use carefully, because `CASCADE` may remove dependent objects as well.

---

tsapiinsert

Signature

```
tsapiinsert(
  schema_ text,
  table_ text,
  source_ text,
  ts_ timestamp with time zone,
  data_ jsonb
) returns void
```

Description

Inserts a single raw time-series record into a TSAPI table. The inserted row contains the specified `source`, timestamp, and JSON payload in `data`.

Arguments

`schema_`

- schema name

`table_`

- target raw TS table

`source_`

- source identifier

`ts_`

- timestamp of the data point

`data_`

- JSON payload

Returns

- `void`

Example

```
select tsapiinsert(
  'public',
  'measurements',
  'BB_ar252_1',
  now(),
  '{"t": 21.4, "rH": 45.0}'::jsonb
);
```

Rules

- Use this for single inserts from scripts or triggers.
- The target table should follow the raw TS table structure.

---

tsapiinsertvalues

Signature

```
tsapiinsertvalues(
  schema_ text,
  table_ text,
  source_ text,
```

```
  data_ json
) returns void
```

Description

Bulk inserts raw time-series values for one source from a JSON array. Before inserting, the function deletes existing rows for the same source starting from the minimum incoming timestamp used in the JSON array logic. The input array is expected to contain objects with `ts` and `data`.

Arguments

`schema_`

- schema name

`table_`

- target raw TS table

`source_`

- source identifier

`data_`

- JSON array of rows

Expected JSON element structure

```
[
 {
  "ts": "2026-04-21T10:00:00Z",
  "data": { "t": 21.4 }
 }
]
```

This structure is implied by the function reading `i->>'ts'` and `i->>'data'`.

Returns

- `void`

Example

```
select tsapiinsertvalues(
  'public',
  'measurements',
```

```
  'BB_ar252_1',
  '[
    {"ts":"2026-04-21T10:00:00Z","data":{"t":21.4}},
    {"ts":"2026-04-21T10:01:00Z","data":{"t":21.5}}
   ]'::json
);
```

Rules

- Use this for batch inserts for one source.
- The JSON array should be ordered consistently by timestamp in application code, even though the function builds a multi-row insert dynamically.

---

tsapiinsertagg

Signature

```
tsapiinsertagg(
  schema_ text,
  table_ text,
  source_ text,
  dataandextras_ json
) returns void
```

Description

Bulk inserts aggregated time-series rows into an aggregation table. Before inserting, it finds the minimum timestamp in the provided JSON array and deletes existing rows for that source from that timestamp onward. Then it inserts rows containing `ts`, `data`, and `extras`.

Arguments

`schema_`

- schema name

`table_`

- target aggregation table

`source_`

- source identifier

`dataandextras_`

- JSON array of aggregation rows

Expected JSON element structure

```
[
 {
  "ts": "2026-04-21T10:00:00Z",
  "data": { "avg": 21.4 },
  "extras": { "status": "ok" }
 }
]
```

This structure is required because the function reads `ts`, `data`, and `extras` from each JSON element.

Returns

- `void`

Example

```
select tsapiinsertagg(
  'public',
  'measurements_1h',
  'BB_ar252_1',
  '[
    {
     "ts":"2026-04-21T10:00:00Z",
     "data":{"avg":21.4},
     "extras":{"status":"ok"}
    }
  ]'::json
);
```

Rules

- Use this only with aggregation tables created by `tsapicreateaggregationtable(...)`.
- Include both `data` and `extras` in every inserted row.

---

tsapiselectdistinctsources

Signature

```
tsapiselectdistinctsources(schema_ text, table_ text)
```

Description

Returns the distinct source names present in the selected table.

Arguments

`schema_`

- schema name

`table_`

- table name

Returns

- `source`

Example

```
select * from tsapiselectdistinctsources('public', 'measurements');
```

Rules

- Use this when enumerating available time-series sources for dashboards, replication, or device discovery logic.

---

tsapiselect

Signature

```
tsapiselect(
  schema_ text,
  table_ text,
  source_ text,
  tsfrom_ text,
  tsto_ text,
  limit_ integer
) returns table (ts timestamp with time zone, data jsonb)
```

Description

Returns raw time-series rows for one source, optionally filtered by start time, end time, and row limit. Results are ordered ascending by `ts` when a positive limit is used.

Arguments

`schema_`

- schema name

`table_`

- raw TS table

`source_`

- source identifier

`tsfrom_`

- lower time bound as text; empty string disables the lower bound

`tsto_`

- upper time bound as text; empty string disables the upper bound

`limit_`

- row limit; `<= 0` means no explicit limit

Returns

- `ts`
- `data`

Example

```
select * from tsapiselect(
  'public',
  'measurements',
  'BB_ar252_1',
  '2026-04-21T00:00:00Z',
  '2026-04-21T23:59:59Z',
  1000
);
```

Rules

- Use empty string `''` when you do not want a lower or upper bound.
- Use this as the main range query for raw TS data.

---

tsapiselectfirst

Signature

```
tsapiselectfirst(schema_ text, table_ text, source_ text)
```

Description

Returns the earliest raw row for the selected source.

Arguments

`schema_`

- schema name

`table_`

- raw TS table

`source_`

- source identifier

Returns

- `ts`
- `data`

Example

```
select * from tsapiselectfirst('public', 'measurements', 'BB_ar252_1');
```

Rules

- Use this for initialization, first-known-value checks, or historical range anchoring.

---

tsapiselectlast

Signature

```
tsapiselectlast(schema_ text, table_ text, source_ text)
```

Description

Returns the latest raw row for the selected source.

Arguments

`schema_`

- schema name

`table_`

- raw TS table

`source_`

- source identifier

Returns

- `ts`
- `data`

Example

```
select * from tsapiselectlast('public', 'measurements', 'BB_ar252_1');
```

Rules

- Use this for last-value widgets, replication checkpoints, or incremental processing baselines.

---

tsapiselectlastn

Signature

```
tsapiselectlastn(
  schema_ text,
  table_ text,
  source_ text,
  n_ integer
)
```

Description

Returns the last n raw time-series samples for the selected source.

This function is intended for cases where the caller needs the most recent group of samples instead of only one latest row. Typical use cases include:

- recent trend windows,
- last-N-value calculations,
- short history retrieval for charts,
- preparing input for lightweight analytics.

Arguments

`schema_`

- schema name

`table_`

- raw time-series table name

`source_`

- source identifier

`n_`

- number of latest samples to return

Returns

- `ts`
- `data`

Behavior

- selects rows for one source
- returns the newest n rows
- intended semantics: “last N samples”
- in practical use, results should be treated as the most recent sample window for the source

Example

```
select * from tsapiselectlastn(
  'public',
```

```
  'measurements',
  'BB_ar252_1',
  100
);
```

Example use case

```
select * from tsapiselectlastn(
  'public',
  'measurements',
  'BB_Pion2_Gaz',
  10
);
```

This returns the 10 most recent samples for `BB_Pion2_Gaz`.

Rules

- Use this when a script or dashboard needs a recent fixed-size sample window.
- Use `tsapiselectlast(...)` when only one latest sample is needed.
- Use `tsapiselect(...)` when a time range is more important than sample count.

Recommended usage

Use `tsapiselectlastn(...)` for:

- recent charts,
- rolling calculations,
- anomaly checks on recent values,
- previewing the newest device data before aggregation.
-

---

tsapiselectprevious

Signature

```
tsapiselectprevious(
  schema_ text,
  table_ text,
  source_ text,
  ts_ text
)
```

Description

Returns the latest raw row strictly earlier than the provided timestamp.

Arguments

`schema_`

- schema name

`table_`

- raw TS table

`source_`

- source identifier

`ts_`

- comparison timestamp as text

Returns

- `ts`
- `data`

Example

```
select * from tsapiselectprevious(
  'public',
  'measurements',
  'BB_ar252_1',
  '2026-04-21T10:00:00Z'
);
```

Rules

- Use this to find the previous known value before a point in time.
- Useful for gap filling and incremental comparison logic.

---

tsapiselectagg

Signature

```
tsapiselectagg(
  schema_ text,
```

```
  table_ text,
  source_ text,
  tsfrom_ text,
  tsto_ text,
  limit_ integer
)
```

Description

Returns aggregation rows for one source, including both aggregated `data` and `extras`. It supports optional time bounds and an optional limit.

Arguments

Same semantics as `tsapiselect(...)`, but for aggregation tables.

Returns

- `ts`
- `data`
- `extras`

Example

```
select * from tsapiselectagg(
  'public',
  'measurements_1h',
  'BB_ar252_1',
  '',
  '',
  100
);
```

Rules

- Use this when the consumer needs full aggregation metadata, not just the status field.

---

tsapiselectaggdata

Signature

```
tsapiselectaggdata(
  schema_ text,
```

```
  table_ text,
  source_ text,
  tsfrom_ text,
  tsto_ text,
  limit_ integer
)
```

Description

Returns aggregation rows for one source, but instead of the full `extras` JSON it returns only `extras->>'status'` as `status`.

Arguments

Same semantics as `tsapiselectagg(...)`.

Returns

- `ts`
- `data`
- `status`

Example

```
select * from tsapiselectaggdata(
  'public',
  'measurements_1h',
  'BB_ar252_1',
  '',
  '',
  100
);
```

Rules

- Use this for lightweight aggregation queries when only the aggregation status is needed.
- Prefer `tsapiselectagg(...)` if the caller needs all fields from `extras`.

---

tsapiselectaggwheretsin

Signature

```
tsapiselectaggwheretsin(
```

```
  schema_ text,
  table_ text,
  source_ text,
  tsin_ text,
  limit_ integer
)
```

Description

Returns aggregation rows whose timestamps are contained in the provided `IN (...)` expression text.

Arguments

`schema_`

- schema name

`table_`

- aggregation table

`source_`

- source identifier

`tsin_`

- SQL text used after `ts in`
- expected to contain a valid SQL tuple/list expression

`limit_`

- optional row limit

Returns

- `ts`
- `data`
- `extras`

Example

```
select * from tsapiselectaggwheretsin(
  'public',
  'measurements_1h',
  'BB_ar252_1',
```

```
  '("2026-04-21T10:00:00Z","2026-04-21T11:00:00Z")',
  10
);
```

Rules

- Use this only when the caller already has an explicit list of timestamps.
- Build `tsin_` carefully, because it is passed as SQL text.

---

tsapiselectvariablefromagg

Signature

```
tsapiselectvariablefromagg(
  schema_ text,
  table_ text,
  source_ text,
  tsfrom_ text,
  tsto_ text,
  limit_ integer
) returns table (ts timestamp with time zone, data jsonb, extras jsonb)
```

Description

Returns aggregation rows using the same observable signature and query shape as `tsapiselectagg(...)` in the attached script. The visible SQL snippet shows it selecting `ts`, `data`, and `extras` for a source with optional time bounds and limit.

Arguments

Same semantics as `tsapiselectagg(...)`.

Returns

- `ts`
- `data`
- `extras`

Example

```
select * from tsapiselectvariablefromagg(
  'public',
  'measurements_1h',
  'BB_ar252_1',
```

```
  '',
  '',
  100
);
```

Rules

- In the attached version of `tsapi.sql`, its visible behavior matches `tsapiselectagg(...)`.
- For final production docs, it is worth confirming whether a later revision specializes this function semantically beyond the visible snippet.

Recommended usage patterns

Pattern 1 — create and insert raw TS data

```
select tsapicreatetable('public', 'measurements');

select tsapiinsert(
  'public',
  'measurements',
  'BB_ar252_1',
  now(),
  '{"t":21.4,"rH":45.0}'::jsonb
);
```

This is the standard pattern for raw time-series storage.

Pattern 2 — query latest raw value

```
select * from tsapiselectlast('public', 'measurements', 'BB_ar252_1');
```

Use this for last-value panels and incremental processing checkpoints.

Pattern 3 — insert and query aggregation data

```
select tsapicreateaggregationtable('public', 'measurements_1h');

select tsapiinsertagg(
  'public',
  'measurements_1h',
  'BB_ar252_1',
  '[
    {
      "ts":"2026-04-21T10:00:00Z",
```

```
    "data":{"avg":21.4},
    "extras":{"status":"ok"}
   }
  ]'::json
);

select * from tsapiselectaggdata(
  'public',
  'measurements_1h',
  'BB_ar252_1',
  '',
  '',
  100
);
```

This is the standard aggregation-table workflow.

Rules for code generation

When generating SQL that uses TSAPI functions, follow these rules:

1. Use `tsapicreatetable(...)` for raw data and `tsapicreateaggregationtable(...)` for aggregation output tables.
2. Use `tsapiinsert(...)` for single-row raw inserts and `tsapiinsertvalues(...)` for bulk inserts for one source.
3. Use `tsapiinsertagg(...)` only for aggregation tables, and always include both `data` and `extras` in the JSON array.
4. Use `tsapiselect(...)`, `tsapiselectfirst(...)`, `tsapiselectlast(...)`, and `tsapiselectprevious(...)` for raw-table retrieval.
5. Use `tsapiselectagg(...)` or `tsapiselectaggdata(...)` for aggregation tables, depending on whether full `extras` or only `status` is needed.
6. Pass empty string `''` for optional time parameters when no lower or upper bound should be applied.
7. Treat `tsin_` in `tsapiselectaggwheretsin(...)` as raw SQL text and construct it carefully.

Time Series JSON Data Aggregation

Supported aggregation periods

The aggregator configuration parses period strings and maps them into three groups:

- minute periods: values ending in m, for example "1m", "15m"
- hour periods: values ending in h, for example "1h"
- day periods: values ending in d, for example "1d"

The defaults from JSAggregatorCfg are:

- minutes: {1, 15}
- hours: {1}
- days: {1}

So the default aggregation cascade is equivalent to:

- 1m
- 15m
- 1h
- 1d

Aggregation table naming

The aggregator creates a cascade of aggregation tables derived from the source table. The manual states that table names are based on the source table plus interval suffixes such as:

- table_1minute
- table_15minutes
- table_1hour
- table_1day

The constructor of JSAggregator confirms that it automatically creates aggregation tables for every configured minute/hour/day interval and builds a cascade from the raw table into those derived tables.

Time zone handling

TsApi accepts a time zone string for the aggregator. In JSAggregatorCfg, the string is converted into QTimeZone. If the provided time zone is invalid, the runtime falls back to UTC. The time zone is especially relevant for day-based aggregation boundaries.

Step size

The aggregator uses stepSize to limit how much data is processed in one aggregation cycle. The manual describes this as the number of source rows handled per aggregation cycle, with the documented default of 10000. The C++ API also exposes setAggregatorStepSize(...) for runtime tuning.

Aggregator source filtering

An aggregator may process:

- all sources from the input table, or
- only a selected subset of sources.

The config parser reads sourcesJsonArray into sourcesList. If the array is empty, aggregation is not restricted to named sources. If it contains values, only those sources are aggregated.

Aggregator debug mode

The manual says debug mode shows computation time and SQL insert time for each aggregated source and table. The C++ API exposes setAggregatorDebugMode(...), which toggles the internal mDebugMode flag of the selected aggregator.

TsApi.aggregate

Signature

```
TsApi.aggregate(aggregatorName: String): void
```

Description

Triggers one aggregation cycle for a previously defined aggregator.

This function is intended to be called from onHook(). It tells the internal aggregation engine to process the next portion of data for the named aggregator. In the C++ implementation, if the aggregator exists, aggregate() is executed on the corresponding JSAggregator instance. If the aggregator name is not defined, the engine throws a RangeError.

Arguments

aggregatorName

- type: String
- name previously used in TsApi.defineAggregator(...)

Returns

- void

Behavior

- looks up the aggregator in the internal map
- runs the next aggregation step
- throws a RangeError if the aggregator does not exist

Example

```
function onHook() {
  TsApi.aggregate("a");
}
```

Rules

- Call this only after the aggregator has been defined.
- Place it in onHook(), not onStartup(), unless you intentionally want a one-shot run.

TsApi.defineAggregator

Signature

```
TsApi.defineAggregator(
  aggregatorName: String,
  serverName: String,
  sourceTableName: String
): void

TsApi.defineAggregator(
  aggregatorName: String,
  serverName: String,
  sourceTableName: String,
  timeZone: String
): void

TsApi.defineAggregator(
  aggregatorName: String,
  serverName: String,
  sourceTableName: String,
  timeZone: String,
  sourcesJsonArray: String
): void

TsApi.defineAggregator(
  aggregatorName: String,
```

```
  serverName: String,
  sourceTableName: String,
  timeZone: String,
  sourcesJsonArray: String,
  periodsJsonArray: String
): void

TsApi.defineAggregator(
  aggregatorName: String,
  serverName: String,
  sourceTableName: String,
  timeZone: String,
  sourcesJsonArray: String,
  periodsJsonArray: String,
  stepSize: Number
): void
```

Description

Creates and registers a named aggregation engine.

This is the central TsApi configuration function. It binds an aggregator name to:

- an SQL server connection,
- a raw time-series source table,
- a time zone,
- an optional source filter,
- an optional set of aggregation periods,
- an optional processing step size.

Arguments

aggregatorName

- type: String
- unique logical name of the aggregator

serverName

- type: String
- configured SQL server name used by InDriver

sourceTableName

- type: String
- raw time-series table name
- typically a schema-qualified table such as:

"public.weather"

The source table is expected to follow the TSAPI raw table standard:

(source text, ts timestamp with time zone, data jsonb)

timeZone

- type: String
- optional
- default: "UTC"

Examples:

"UTC"

"Europe/Warsaw"

"America/Chicago"

If invalid:

- runtime falls back to UTC

sourcesJsonArray

- type: String
- optional
- JSON array of source names

Examples:

"[]"

'["SENSOR_1","SENSOR_2"]'

Behavior:

- empty array means all sources
- non-empty array restricts aggregation to listed sources

periodsJsonArray

- type: String
- optional
- JSON array of period strings

Supported format:

- "1m"
- "15m"
- "1h"
- "1d"

Only strings ending with:

- m
- h
- d

are parsed into minute/hour/day period buckets.

Default behavior from JSAggregatorCfg:

- minutes {1, 15}
- hours {1}
- days {1}

stepSize

- type: Number
- optional
- number of source rows processed per aggregation cycle

The code exposes this field in the configuration object and manual documents default usage around 10000 rows per cycle.

Returns

- void

Behavior

When an aggregator is defined:

- a JSAggregator object is created and stored under aggregatorName
- aggregation tables for configured minute/hour/day periods are created automatically
- an aggregation cascade is built from the raw source table through all derived tables

Examples

Minimal definition

```
InDriver.import("TsApi");
TsApi.defineAggregator("a", "azureserver", "public.weather");
```

Explicit time zone

```
TsApi.defineAggregator(
  "weatherAgg",
  "azureserver",
  "public.weather",
  "Europe/Warsaw"
);
```

Specific sources only

```
TsApi.defineAggregator(
  "weatherAgg",
  "azureserver",
  "public.weather",
  "UTC",
  '["Krakow","Chicago"]'
);
```

Custom periods

```
TsApi.defineAggregator(
  "weatherAgg",
  "azureserver",
  "public.weather",
  "UTC",
  '["Krakow","Chicago"]',
  '["1m","5m","1h","1d"]'
);
```

Custom step size

```
TsApi.defineAggregator(
  "weatherAgg",
  "azureserver",
  "public.weather",
  "UTC",
  '["Krakow","Chicago"]',
  '["1m","5m","1h","1d"]',
  20000
);
```

Rules

- Use schema-qualified table names when possible.
- Pass JSON arrays as strings, because the exported API expects QString.
- Prefer explicit periods when the use case is performance-sensitive.

TsApi.setAggregatorDebugMode

Signature

```
TsApi.setAggregatorDebugMode(aggregatorName: String, mode: Boolean): void
```

Description

Enables or disables debug mode for a selected aggregator.

The manual describes debug mode as exposing timing/debug information for calculations and SQL insert operations performed by the aggregator. The C++ implementation sets an internal mDebugMode flag on the matching aggregator if it exists. If the name is missing, nothing happens.

Arguments

aggregatorName

- type: String

mode

- type: Boolean

Returns

- void

Behavior

- if aggregator exists → debug mode flag is updated
- if aggregator does not exist → no exception, no action

Example

```
TsApi.setAggregatorDebugMode("weatherAgg", true)
```

Rules

- Use during development or when diagnosing aggregation performance.
- Do not assume this function validates the aggregator name.

TsApi.setAggregatorStepSize

Signature

```
TsApi.setAggregatorStepSize(aggregatorName: String, stepSize: Number): void
```

Description

Updates the step size for an existing aggregator.

Step size controls how much data the aggregation engine processes during one TsApi.aggregate(...) cycle. This is useful for tuning throughput versus execution time. The C++ implementation updates the internal step size only if the aggregator exists.

Arguments

aggregatorName

- type: String

stepSize

- type: Number
- expected positive integer

Returns

- void

Behavior

- if aggregator exists → step size is changed
- if aggregator does not exist → no exception, no action

Example

```
TsApi.setAggregatorStepSize("weatherAgg", 5000);
```

Rules

- Use lower values when hooks are frequent and latency matters.
- Use higher values when catching up large backlogs.

Recommended TSAPI usage pattern

onStartup:

```
InDriver.import("TsApi");

TsApi.defineAggregator(
    "weatherAgg",
    "azureserver",
```

```
    "public.weather",
    "UTC",
    "[]",
    '["1m","15m","1h","1d"]',
    10000
);

TsApi.setAggregatorDebugMode("weatherAgg", true);

InDriver.installHook(60000);
```

onHook:

```
TsApi.aggregate("weatherAgg");
```

This pattern matches the intended TsApi workflow: define once during startup, then trigger incremental aggregation from onHook().

Notes for code generation

When generating code using TsApi, follow these rules:

1. Import TsApi in onStartup().
2. Define aggregators in onStartup().
3. Call TsApi.aggregate(...) in onHook().
4. Use raw source tables with this standard structure:
    - source
    - ts
    - data
5. Assume aggregation tables use:
    - source
    - ts
    - data
    - extras
6. Pass sourcesJsonArray and periodsJsonArray as JSON strings.
7. Use valid period suffixes only: m, h, d.
8. If time zone is uncertain, use "UTC" explicitly.

Important implementation insights

The code and SQL together show that TsApi is not just a wrapper around one SQL function. It is a coordinated system:

- the **InDriver** manages aggregators, table cascade, scheduling logic, and cross-database SQL translation,
- the **SQL side** defines the standard raw/aggregation table contract and helper database functions such as:
  - tsapicreatetable
  - tsapicreateaggregationtable
  - tsapiselect
  - tsapiselectfirst
  - tsapiselectlast
  - tsapiselectprevious
  - tsapiinsertagg

That structure is exactly why TsApi is well suited for AI-generated code: it has a stable data contract, a fixed aggregation model, and clearly defined configuration parameters.

ProcessApi

ProcessApi provides process lifecycle management for external programs in InDriver.

It allows:

- starting named processes,
- monitoring execution state,
- waiting for start/finish,
- stopping or killing processes,
- managing multiple processes simultaneously.

The API is **stateful** — processes are stored internally under a unique name.

Execution model

Processes are managed by **name**:

```
ProcessApi.start("myProcess", "python", ["script.py"]);
```

Then controlled via:

```
ProcessApi.waitForFinished("myProcess");
ProcessApi.kill("myProcess");
```

ProcessApi.start

Signature

ProcessApi.start(name: String, program: String, args: Array): String

Description

Starts a new process and registers it under a given name.

Arguments

name

- type: String
- unique process identifier

program

- type: String

- executable path or command

args

- type: Array
- list of arguments

Returns

String

Meaning

- process identifier or status message

Behavior

- creates a new process instance
- associates it with name
- if process with same name exists → may overwrite or fail depending on runtime

Example

```
ProcessApi.start("job1", "python", ["script.py"]);
```

ProcessApi.close

Signature

```
ProcessApi.close(name: String): void
```

Description

Gracefully closes a running process.

Behavior

- sends termination signal
- allows process to exit cleanly

Example

```
ProcessApi.close("job1");
```

ProcessApi.closeAll

Signature

ProcessApi.closeAll(): void

Description

Gracefully closes all managed processes.

ProcessApi.kill

Signature

ProcessApi.kill(name: String): void

Description

Forcefully terminates a process.

Behavior

- immediate termination
- no cleanup guaranteed

Example

ProcessApi.kill("job1");

ProcessApi.killAll

Signature

ProcessApi.killAll(): void

Description

Forcefully terminates all processes.

ProcessApi.waitForStarted

Signature

ProcessApi.waitForStarted(name: String, msecs: Number = 30000): Boolean

Description

Waits until a process starts.

Arguments

name

- process name

msecs

- timeout in milliseconds (default: 30000)

Returns

Boolean

- true → started successfully
- false → timeout or failure

Example

```
ProcessApi.start("job1", "python", ["script.py"]);

if (!ProcessApi.waitForStarted("job1")) {
  InDriver.debug("Process failed to start", "critical");
}
```

ProcessApi.waitForFinished

Signature

```
ProcessApi.waitForFinished(name: String, msecs: Number = 30000): Boolean
```

Description

Waits until a process finishes.

Returns

- true → finished
- false → timeout

Example

```
ProcessApi.waitForFinished("job1", 10000);
```

ProcessApi.waitAllForStarted

Signature

ProcessApi.waitAllForStarted(msecs: Number = 30000): Boolean

Description

Waits until all processes are started.

ProcessApi.waitAllForFinished

Signature

ProcessApi.waitAllForFinished(msecs: Number = 30000): Boolean

Description

Waits until all processes are finished.

ProcessApi.setWorkingDirectory

Signature

ProcessApi.setWorkingDirectory(name: String, dir: String): void

Description

Sets working directory for a process before start.

Example

```
ProcessApi.setWorkingDirectory("job1", "c:/scripts");
```

ProcessApi.workingDirectory

Signature

ProcessApi.workingDirectory(name: String): String

Description

Returns working directory of the process.

ProcessApi.program

Signature

ProcessApi.program(name: String): String

Description

Returns program path used to start the process.

ProcessApi.pid

Signature

ProcessApi.pid(name: String): Number

Description

Returns operating system process ID.

Example

```
let pid = ProcessApi.pid("job1");
```

ProcessApi.state

Signature

```
ProcessApi.state(name: String): String
```

Description

Returns current process state.

Possible values (typical)

NotRunning
Starting
Running
Finished

ProcessApi.status

Signature

ProcessApi.status(name: String): Object

Description

Returns detailed process status.

Returns

Object

Example structure

```
{
  state: "Running",
  pid: 1234,
  program: "python",
  workingDirectory: "c:/scripts"
}
```

ProcessApi.remove

Signature

```
ProcessApi.remove(name: String): void
```

Description

Removes process from internal registry.

Behavior

- does NOT stop process automatically
- only removes reference

Recommended usage patterns

Pattern 1 — run and wait

```
ProcessApi.start("job1", "python", ["script.py"]);

if (ProcessApi.waitForFinished("job1")) {
  InDriver.debug("Done");
} else {
  InDriver.debug("Timeout", "critical");
}
```

Pattern 2 — multiple processes

```
ProcessApi.start("job1", "python", ["a.py"]);
ProcessApi.start("job2", "python", ["b.py"]);

ProcessApi.waitAllForFinished();
```

Pattern 3 — watchdog

```
if (ProcessApi.state("job1") !== "Running") {
  ProcessApi.start("job1", "service.exe", []);
}
```

Notes for AI code generation

When generating code using ProcessApi, follow these rules:

1. Always use unique process names:
"job1", "collector", "analyzer"
2. Wait for start when required:
waitForStarted()
3. Use timeout protection:
waitForFinished(name, timeout)
4. Use kill for hard failures:
kill(name)
5. Clean registry:
remove(name)

Important behavior

- processes are **persistent objects**
- multiple processes can run in parallel
- API does NOT auto-clean processes
- remove() does NOT stop process
- kill() is forceful
- close() is graceful

PdfApi

PdfApi provides PDF reading and rendering functionality for JavaScript tasks in InDriver.

It allows:

- loading a PDF file,
- checking whether a document is loaded,
- extracting text from a page or page range,
- reading page labels,
- reading document metadata,
- working with password-protected PDFs,
- rendering pages as images,
- exporting page images to disk.

The API is **stateful**: all functions operate on the currently loaded PDF document.

PdfApi.load

Signature

PdfApi.load(file: String): String

Description

Loads a PDF document from disk.

Arguments

file

- type: String
- full or relative file path to the PDF file

Returns

String

Possible return values:

- "OK"
- "DataNotYetAvailable"
- "FileNotFound"
- "InvalidFileFormat"
- "IncorrectPassword"
- "UnsupportedSecurityScheme"
- "Unknown"

Behavior

- clears the previously remembered loaded file path
- attempts to load the specified PDF
- if loading succeeds, the document becomes the active document for all other PdfApi calls

Example

```
InDriver.import("PdfApi");

let status = PdfApi.load(InDriver.currentPath() + "/document.pdf");
InDriver.debug("Load status: " + status);
```

Rules

- Always check the returned status before calling other PdfApi functions.
- For protected PDFs, call PdfApi.setPassword(...) before load(...) if needed.

PdfApi.isLoaded

Signature

PdfApi.isLoaded(): Boolean

Description

Checks whether a PDF document is currently loaded and ready.

Returns

Boolean

- true → document is loaded and ready
- false → no document is loaded or loading failed

Example

```
if (!PdfApi.isLoaded()) {
  InDriver.debug("PDF is not loaded", "critical");
}
```

Rules

- Use this as a guard before reading text, labels, metadata, or rendering images.

PdfApi.pageText

Signature

```
PdfApi.pageText(page: Number): String
```

Description

Returns the extracted text of a single PDF page.

Arguments

page

- type: Number
- zero-based page index

Returns

String

- page text if the page is valid
- empty string "" if the page index is invalid

Behavior

- validates page index against document page count
- if the index is invalid, returns empty string and writes a debug message
- if a codec has been set with PdfApi.setCodec(...), the extracted text is decoded using that codec
- otherwise, default text extraction is used

Example

```
let page0 = PdfApi.pageText(0);
InDriver.debug(page0);
```

Rules

- Page index is zero-based.
- If text looks incorrectly decoded, try PdfApi.setCodec("Utf8") or another supported codec before calling pageText(...).

PdfApi.readText

Signature

```
PdfApi.readText(from: Number = 0, count: Number = -1): String
```

Description

Returns concatenated text from a page range of the currently loaded PDF.

Arguments

from

- type: Number
- optional
- default: 0
- zero-based index of the first page to read

count

- type: Number
- optional
- default: -1
- number of pages to read

Returns

String

- concatenated text from selected pages
- pages are joined with newline separators
- empty string "" if from is outside valid page range

Behavior

- if count < 0, text is read from from to the last page
- if count >= 0, reads at most count pages
- empty pages are skipped
- trailing extra newline is trimmed from final result

Examples

```
let allText = PdfApi.readText();
let fromPage5 = PdfApi.readText(5);
let intro = PdfApi.readText(0, 2);
```

Example

```
InDriver.import("PdfApi");
PdfApi.load(InDriver.currentPath() + "/manual.pdf");

let intro = PdfApi.readText(0, 2);
InDriver.debug("Intro text:\n" + intro);
```

Rules

- Use readText() for full-document extraction.
- Use pageText(page) when you need strict page-by-page handling.

PdfApi.pageLabel

Signature

```
PdfApi.pageLabel(page: Number): String
```

Description

Returns the label of a specific page.

Page labels may differ from numeric page indexes in PDFs that use custom numbering schemes.

Arguments

page

- type: Number
- zero-based page index

Returns

String

Example

```
let label = PdfApi.pageLabel(0);
InDriver.debug("Page Label: " + label);
```

Rules

- Use pageLabel(...) when user-facing page numbering matters.
- Do not assume page label equals page + 1.

PdfApi.allPageLabels

Signature

```
PdfApi.allPageLabels(): String[]
```

Description

Returns labels of all pages in the currently loaded PDF.

Returns

String[]

Behavior

- iterates through all pages
- returns one label per page in order

Example

```
let labels = PdfApi.allPageLabels();
InDriver.debug(labels);
```

Rules

- Use when building page selectors or page navigation logic.

PdfApi.setCodec

Signature

```
PdfApi.setCodec(codec: String): Boolean
```

Description

Sets the codec used for text decoding during PDF text extraction.

Arguments

codec

- type: String
- codec name, for example:
  - "Utf8"

- another codec name supported by the runtime environment

Returns

Boolean

- true → codec accepted
- false → unsupported codec; runtime falls back to "Utf8" and logs a debug message

Example

```
PdfApi.setCodec("Utf8");
InDriver.debug("Codec set to Utf8");
```

Rules

- If PDF text output looks corrupted, set codec explicitly before calling pageText(...) or readText(...).
- Unsupported codec names are not fatal; they revert to "Utf8".

PdfApi.pageCount

Signature

```
PdfApi.pageCount(): Number
```

Description

Returns the total number of pages in the loaded PDF document.

Returns

Number

Example

```
let count = PdfApi.pageCount();
InDriver.debug("Total Pages: " + count);
```

Rules

- Use to validate page indexes before calling pageText, pageLabel, asImageBase64, or savePageAsImage.

PdfApi.setPassword

Signature

```
PdfApi.setPassword(password: String): void
```

Description

Sets the password used to open a password-protected PDF.

Arguments

password

- type: String

Returns

- void

Behavior

- stores the password in the current PDF session
- typically used before calling load(...) on protected PDFs

Example

```
PdfApi.setPassword("securePassword123");
let status = PdfApi.load(InDriver.currentPath() + "/protected.pdf");
InDriver.debug("Load status: " + status);
```

Rules

- Use before load(...) when password protection is expected.
- If load(...) returns "IncorrectPassword", update password and retry.

PdfApi.password

Signature

```
PdfApi.password(): String
```

Description

Returns the password currently stored in the PDF session.

Returns

String

Example

```
let pwd = PdfApi.password();
InDriver.debug("Current Password: " + pwd);
```

Rules

- Mostly useful for diagnostics or workflow verification.
- Avoid logging passwords in production unless absolutely necessary.

PdfApi.status

Signature

```
PdfApi.status(): String
```

Description

Returns the current document status.

Returns

String

Possible values:

- "Null"
- "Loading"
- "Ready"
- "Unloading"
- "Error"

Example

```
let status = PdfApi.status();
InDriver.debug("Pdf status: " + status);
```

Rules

- PdfApi.isLoaded() is the most direct readiness check.
- status() is useful for diagnostics and UI/debug output.

PdfApi.metadata

Signature

```
PdfApi.metadata(): String
```

Description

Returns document metadata as serialized JSON text.

Returns

String

The returned JSON includes at least:

- Author
- CreationDate
- Creator
- ModificationDate
- Keywords
- Producer
- Subject
- Title
- FileSize
- LastModified

Example

```
let meta = PdfApi.metadata();
InDriver.debug(meta);
```

Rules

- Parse with JSON.parse(...) if structured access is needed:

```
let meta = JSON.parse(PdfApi.metadata());
```

PdfApi.asImageBase64

Signature

PdfApi.asImageBase64(page: Number, format: String = "PNG"): String

Description

Renders a PDF page as an image and returns it as a Base64 data URL.

Arguments

page

- type: Number
- zero-based page index

format

- type: String
- optional
- default: "PNG"
- image format used when encoding the rendered page

Returns

String

- Base64 data URL, for example:

data:image/png;base64,...

- empty string "" if rendering fails or page index is invalid

Behavior

- validates page index
- renders page using document page size
- encodes rendered image into selected format
- returns a data:image/<format>;base64,... string

Example

```
let img = PdfApi.asImageBase64(0);
InDriver.debug(img);
```

Rules

- Use this for embedding PDF page previews in HTML, dashboards, or JSON payloads.
- The returned value is not raw base64 only; it is a full data URL.

PdfApi.savePageAsImage

Signature

```
PdfApi.savePageAsImage(page: Number, path: String): String
```

Description

Renders the specified PDF page and saves it as an image file.

Arguments

page

- type: Number
- zero-based page index

path

- type: String
- full output file path

Returns

String

- saved file path if successful
- empty string "" on error

Behavior

- validates page index
- renders page to image
- writes image to disk
- returns the same path if save succeeds

Example

```
PdfApi.load(InDriver.currentPath() + "/report.pdf");
let out = PdfApi.savePageAsImage(0, InDriver.currentPath() + "/page0.png");
InDriver.debug("Saved image: " + out);
```

Rules

- Use for exporting thumbnails or snapshots of PDF pages.
- If return value is empty, treat the save as failed.

PdfApi.close

Signature

```
PdfApi.close(): void
```

Description

Closes the currently loaded PDF document.

Returns

- void

Behavior

- unloads the active PDF document
- after closing, text extraction, metadata, and rendering functions should be considered unavailable until a new document is loaded

Example

```
PdfApi.close();
```

Rules

- Call close() when the PDF is no longer needed, especially in long-running tasks.

Recommended usage pattern

onStartup:

```
InDriver.import("PdfApi");

PdfApi.setCodec("Utf8");

let status = PdfApi.load(InDriver.currentPath() + "/manual.pdf");
if (status !== "OK") {
    InDriver.debug("PDF load failed: " + status, "critical");
    return;
}

InDriver.debug("Pages: " + PdfApi.pageCount());
InDriver.debug("Metadata: " + PdfApi.metadata());

let intro = PdfApi.readText(0, 2);
InDriver.debug(intro);
```

```
let img = PdfApi.savePageAsImage(0, InDriver.currentPath() + "/page0.png");
InDriver.debug("Saved image: " + img);

PdfApi.close();
```

Notes for AI code generation

When generating code using PdfApi, follow these rules:

1. Import PdfApi in onStartup():

```
InDriver.import("PdfApi");
```

2. Always load a PDF before reading text or metadata:

```
let status = PdfApi.load(path);
if (status !== "OK") { ... }
```

3. Use PdfApi.isLoaded() or PdfApi.status() as readiness checks.
4. Validate or infer page indexes from PdfApi.pageCount().
5. Parse PdfApi.metadata() if structured access is needed.
6. Use PdfApi.setPassword(...) before load(...) for protected documents.
7. Use PdfApi.close() when finished.

FileApi

FileApi provides file and directory operations for JavaScript tasks in InDriver.

It supports:

- file handles (named files),
- reading/writing data,
- directory navigation,
- filesystem monitoring,
- file utilities.

Files are managed using a **handle name (name)**, not directly by path after opening.

FileApi.addFileSystemWatcherPath

Signature

```
FileApi.addFileSystemWatcherPath(file: String): Boolean
```

Description

Adds a file or directory to the filesystem watcher.

Arguments

file

- path to file or directory

Returns

- true → watcher added
- false → error

Example

```
FileApi.addFileSystemWatcherPath("c:/data/config.json");
```

FileApi.appendLine

Signature

```
FileApi.appendLine(name: String, line: String): Boolean
```

Description

Appends a single line to an opened file.

Arguments

name

- file handle name

line

- text to append

Returns

- true → success
- false → error

Example

```
FileApi.open("log", "c:/log.txt", ["Append"]);
FileApi.appendLine("log", "New event");
```

FileApi.close

Signature

```
FileApi.close(name: String): void
```

Description

Closes an opened file.

Example

```
FileApi.close("log");
```

FileApi.closeAll

Signature

```
FileApi.closeAll(): void
```

Description

Closes all opened files.

FileApi.copy

Signature

```
FileApi.copy(from: String, to: String): Boolean
```

Description

Copies a file from one location to another.

Example

```
FileApi.copy("c:/a.txt", "c:/backup/a.txt");
```

FileApi.dirGoTo

Signature

```
FileApi.dirGoTo(dir: String): Boolean
```

Description

Changes current working directory to a subdirectory.

FileApi.dirGoToApplicationDirectory

Signature

```
FileApi.dirGoToApplicationDirectory(): Boolean
```

Description

Sets working directory to InDriver application directory.

FileApi.dirGoToPath

Signature

```
FileApi.dirGoToPath(path: String): Boolean
```

Description

Changes working directory to an absolute path.

FileApi.dirGoUp

Signature

```
FileApi.dirGoUp(): Boolean
```

Description

Moves working directory one level up.

FileApi.dirList

Signature

```
FileApi.dirList(fileTypeFilter: String = ""): String
```

Description

Returns directory content as serialized string.

Example

```
let list = FileApi.dirList("*.txt");
InDriver.debug(list);
```

FileApi.dirLoadFiles

Signature

```
FileApi.dirLoadFiles(files: String): String
```

Description

Loads multiple files based on input definition.

FileApi.dirSaveFiles

Signature

```
FileApi.dirSaveFiles(filesArray: String): String
```

Description

Saves multiple files based on structured input.

FileApi.fileExists

Signature

```
FileApi.fileExists(path: String): Boolean
```

Description

Checks whether a file exists.

Example

```
if (FileApi.fileExists("c:/data.txt")) {
  InDriver.debug("File exists");
}
```

FileApi.fileSize

Signature

```
FileApi.fileSize(name: String): Number
```

Description

Returns size of an opened file in bytes.

FileApi.isOpen

Signature

```
FileApi.isOpen(name: String): Boolean
```

Description

Checks whether a file handle is currently open.

FileApi.open

Signature

```
FileApi.open(name: String, file: String, modes: Array = ["ReadOnly"]): String
```

Description

Opens a file and assigns it to a handle.

Arguments

name

- file handle identifier

file

- file path

modes

- possible values:
    - "ReadOnly"
    - "WriteOnly"
    - "Append"
    - "ReadWrite"

Returns

- "OK" → success
- error string → failure

Example

```
FileApi.open("f1", "c:/data.txt", ["ReadWrite"]);
```

FileApi.readAll

Signature

```
FileApi.readAll(name: String): String
```

Description

Reads entire content of an opened file.

FileApi.readAllBase64

Signature

```
FileApi.readAllBase64(name: String): String
```

Description

Reads file and returns Base64 encoded content.

FileApi.readLines

Signature

```
FileApi.readLines(name: String): Array
```

Description

Reads file and returns array of lines.

Example

```
let lines = FileApi.readLines("f1");
```

FileApi.removeFile

Signature

```
FileApi.removeFile(path: String): Boolean
```

Description

Deletes a file from disk.

FileApi.removeFileSystemWatcherPath

Signature

```
FileApi.removeFileSystemWatcherPath(file: String): Boolean
```

Description

Removes file or directory from watcher.

FileApi.write

Signature

```
FileApi.write(name: String, data: String): Number
```

Description

Writes data to an opened file.

Returns

- number of bytes written

Example

```
FileApi.write("f1", "Hello World");
```

Recommended usage patterns

Pattern 1 — logging

onStartup:

```
InDriver.import("FileApi");

FileApi.open("log", "c:/log.txt", ["Append"]);

if (FileApi.isOpen("log")) {
    FileApi.appendLine("log", "Application started");
}
```

Pattern 2 — read + process

```
FileApi.open("f1", "c:/data.txt", ["ReadOnly"]);

let content = FileApi.readAll("f1");
InDriver.debug(content);

FileApi.close("f1");
```

Pattern 3 — pipeline

```
let data = InDriver.sqlExecute("local", "select * from table");

FileApi.open("out", "c:/export.json", ["WriteOnly"]);
FileApi.write("out", JSON.stringify(data));
FileApi.close("out");
```

Rules for code generation

When generating code using FileApi, follow these rules:

1. Always open file first

FileApi.open(name, path, modes)

2. Always check if open

if (!FileApi.isOpen(name)) { ... }

3. Use correct mode

- Read → "ReadOnly"
- Write → "WriteOnly"
- Append → "Append"

4. Close file after use

FileApi.close(name)

5. Prefer appendLine for logs

FileApi.appendLine(...)

6. Use Base64 for binary data

FileApi.readAllBase64(...)

7. Validate file existence when needed

FileApi.fileExists(path)

SerialPortApi

SerialPortApi provides serial communication (RS232 / RS485) for JavaScript tasks in InDriver.

It allows:

- opening and configuring serial ports,
- sending and receiving data,
- reading buffers,
- managing connection lifecycle.

Ports are managed using a **handle name (name)**.

SerialPortApi.availablePorts

Signature

```
SerialPortApi.availablePorts(): Array
```

Description

Returns a list of available serial ports on the system.

Returns

Array

- list of port names (e.g. "COM1", "COM3")

Example

```
let ports = SerialPortApi.availablePorts();
InDriver.debug(ports);
```

SerialPortApi.close

Signature

```
SerialPortApi.close(name: String): void
```

Description

Closes an opened serial port.

Example

```
SerialPortApi.close("port1");
```

SerialPortApi.closeAll

Signature

```
SerialPortApi.closeAll(): void
```

Description

Closes all opened serial ports.

SerialPortApi.flush

Signature

```
SerialPortApi.flush(name: String): void
```

Description

Clears input and output buffers of the serial port.

SerialPortApi.isOpen

Signature

```
SerialPortApi.isOpen(name: String): Boolean
```

Description

Checks whether a serial port is open.

Returns

- true → open
- false → closed

SerialPortApi.open

Signature

```
SerialPortApi.open(name: String, port: String, config: Object): Boolean
```

Description

Opens and configures a serial port.

Arguments

name

- internal handle name

port

- system port name (e.g. "COM3")

config

- configuration object:

```
{
  baudRate: Number,
  dataBits: Number,
  parity: "None" | "Even" | "Odd",
  stopBits: Number,
  flowControl: "None" | "Hardware" | "Software"
}
```

Returns

- true → success
- false → error

Example

```
SerialPortApi.open("p1", "COM3", {
```

```
  baudRate: 9600,
  dataBits: 8,
  parity: "None",
  stopBits: 1,
  flowControl: "None"
});
```

SerialPortApi.read

Signature

```
SerialPortApi.read(name: String): String
```

Description

Reads available data from serial port buffer.

Returns

- received data as string
- empty string if no data

SerialPortApi.readAll

Signature

```
SerialPortApi.readAll(name: String): String
```

Description

Reads entire buffer content.

Behavior

- clears buffer after reading

SerialPortApi.readLine

Signature

```
SerialPortApi.readLine(name: String): String
```

Description

Reads a single line from the serial port.

Behavior

- reads until newline (\n)

SerialPortApi.setTimeout

Signature

```
SerialPortApi.setTimeout(name: String, msecs: Number): void
```

Description

Sets read timeout.

SerialPortApi.write

Signature

```
SerialPortApi.write(name: String, data: String): Number
```

Description

Writes data to serial port.

Returns

- number of bytes written

Example

```
SerialPortApi.write("p1", "AT\r\n");
```

Recommended usage patterns

Pattern 1 — simple communication

onStartup:

```
InDriver.import("SerialPortApi");

SerialPortApi.open("p1", "COM3", {
    baudRate: 9600,
    dataBits: 8,
    parity: "None",
    stopBits: 1,
    flowControl: "None"
});
```

Pattern 2 — polling device

onHook

```
let response = SerialPortApi.readAll("p1");

if (response) {
    InDriver.debug(response);
}
```

Pattern 3 — request/response

```
SerialPortApi.write("p1", "READ\n");

let response = SerialPortApi.readLine("p1");
InDriver.debug(response);
```

Rules for code generation

When generating code using SerialPortApi:

1. Always open port first

SerialPortApi.open(name, port, config)

2. Always check state

SerialPortApi.isOpen(name)

3. Use correct read method

- streaming → read()
- full buffer → readAll()
- line-based → readLine()

4. Handle timing (hooks)

- serial communication should run in onHook()

5. Always close if not persistent

SerialPortApi.close(name)

TcpSocketApi

TcpSocketApi provides client-side TCP socket communication for JavaScript tasks in InDriver.

It allows:

- opening named TCP client connections,
- sending bytes,
- waiting for replies,
- asynchronous receive handling through onMessage,
- tracking whether a socket is busy with a pending request.

Sockets are managed by a unique name.

TcpSocketApi.acceptRead

Signature

```
TcpSocketApi.acceptRead(name: String): void
```

Description

Confirms that received data should be treated as an accepted response and that waiting for data may stop.

This function is especially important when writeAndWait(...) is used. In the documented request/response pattern, incoming data arrives through onMessage, and acceptRead(...) is called after the user script decides that the received bytes constitute the expected answer.

Arguments

name

- type: String
- socket name

Returns

- void

Example

function onMessage()

```
if (InDriver.taskName() === InDriver.messageSender()) {
   let data = JSON.parse(InDriver.messageData());
   if (JSON.stringify(data.bytes) === "[1,2,3]") {
      TcpSocketApi.acceptRead(data.socketName);
   }
}
```

Rules for AI

- Use acceptRead(...) only in response-handling logic.
- Call it when the received message matches the expected reply.

TcpSocketApi.connect

Signature

```
TcpSocketApi.connect(name: String, tcpCfg: String = ""): Boolean

TcpSocketApi.connect(name: String, tcpCfg: Object): Boolean
```

Description

Creates or recreates a named TCP client socket connection.

If a socket with the same name already exists, it is closed and opened again with the new configuration.

Arguments

name

- type: String
- unique socket identifier

tcpCfg

- type: String | Object
- socket configuration

Supported configuration keys visible from the attached files and manual:

- address
- port

- mode
- timeout
- bufferSize

Supported mode values

- "Ignore"
- "Flow"
- "Buffer"

Meaning of read modes

- Ignore — ignore incoming bytes
- Flow — onMessage is triggered when a part of data arrives
- Buffer — data is buffered and onMessage receives the entire current buffer when more bytes arrive

Default behavior

From the attached files, if fields are omitted:

- address defaults to 127.0.0.1
- port defaults to 0
- timeout defaults to 30000
- mode defaults to Flow
- bufferSize defaults to 256

Returns

Boolean

- true → socket connected successfully
- false → connection failed

Example

```
InDriver.import("TcpSocketApi");

let ok = TcpSocketApi.connect("socket", {
  address: "127.0.0.1",
  port: 50000,
  mode: "Buffer",
  timeout: 5000,
  bufferSize: 256
});

if (!ok) {
```

```
  InDriver.debug("TCP socket connection failed", "critical");
}
```

Rules

- Import TcpSocketApi in onStartup().
- Use stable socket names.
- Reuse the same name if you want reconnect-with-replace behavior.

TcpSocketApi.disconnect

Signature

```
TcpSocketApi.disconnect(name: String): void
```

Description

Disconnects the named TCP socket.

Arguments

name

- type: String
- socket name

Returns

- void

Example

```
TcpSocketApi.disconnect("socket");
```

Rules

- Use when a client socket should be closed but not necessarily removed from application logic.

TcpSocketApi.disconnectAll

Signature

```
TcpSocketApi.disconnectAll(): void
```

Description

Disconnects all registered TCP sockets.

Returns

- void

Example

```
TcpSocketApi.disconnectAll();
```

TcpSocketApi.isBusy

Signature

```
TcpSocketApi.isBusy(name: String): Boolean
```

Description

Returns whether the socket is currently busy, typically because a request is still being processed or waiting for completion.

Arguments

name

- type: String
- socket name

Returns

Boolean

- true → socket is busy
- false → socket is idle or not found

Example

```
if (!TcpSocketApi.isBusy("socket")) {
```

```
  TcpSocketApi.write("socket", [0x01, 0x02, 0x03]);
}
```

Rules for AI

- Use this before sending new requests when the protocol expects one outstanding request at a time.

TcpSocketApi.write

Signature

```
TcpSocketApi.write(
  name: String,
  data: ByteArray,
  request: String = "",
  timeout: Number = 0
): Boolean
```

Description

Writes data to the specified TCP socket.

Arguments

name

- type: String
- socket name

data

- type: ByteArray
- bytes to send

Typical JavaScript usage:

[0x01, 0x02, 0x03]

request

- type: String
- optional request identifier
- stored with the communication context and can appear later in onMessage

timeout

- type: Number
- optional
- timeout in milliseconds

Returns

Boolean

- true → write initiated successfully
- false → socket not found or write failed

Example

```
TcpSocketApi.write("socket", [0x01, 0x02, 0x03]);
```

Rules

- Use for non-blocking send logic.
- Handle incoming data in onMessage.

TcpSocketApi.writeAndWait

Signature

```
TcpSocketApi.writeAndWait(
  name: String,
  data: ByteArray,
  request: String = "",
  timeout: Number = 0
): Boolean
```

Description

Writes data to the specified TCP socket and waits for a response until timeout or until the incoming data is explicitly accepted.

This is a blocking request/response helper. The waiting ends when:

- timeout occurs, or
- the script calls TcpSocketApi.acceptRead(socketName) after verifying the received data.

Arguments

name

- type: String
- socket name

data

- type: ByteArray

request

- type: String
- optional request identifier

timeout

- type: Number
- timeout in milliseconds

Returns

Boolean

- true → request completed successfully
- false → socket not found, write failed, or timeout/accept failed

Example

onStartup:

```
InDriver.import("TcpSocketApi");

TcpSocketApi.connect("socket", '{"address":"127.0.0.1","port":50000}');
```

onHook:

```
TcpSocketApi.writeAndWait("socket", [0x01, 0x02, 0x03], "req1", 5000);
```

onMessage:

```
if (InDriver.taskName() === InDriver.messageSender()) {
    let data = JSON.parse(InDriver.messageData());
    if (JSON.stringify(data.bytes) === "[1,2,3]") {
```

```
        TcpSocketApi.acceptRead(data.socketName);
    }
}
```

Rules

- Use this only when the protocol has a clear request/response pattern.
- In onMessage, validate the received payload before calling acceptRead(...).

TcpSocketApi.onMessage behavior

Description

When data is received on a TCP socket, the task onMessage() handler is triggered.

In this case:

- InDriver.messageSender() is set to the current task name
- InDriver.messageData() contains JSON-formatted socket event data

Example message payload

```
let d = JSON.parse(InDriver.messageData());

/*
d =
{
 "bytes": "[79,75]",
 "data": "OK",
 "request": "request name",
 "socketName": "name of socket"
}
*/
```

Rules

- Parse InDriver.messageData() as JSON.
- Use d.socketName to route the response to the correct socket.
- Use d.request if request correlation is needed.

Recommended usage patterns

Pattern 1 — TCP client socket request/response

onStartup:

```
InDriver.import("TcpSocketApi");
TcpSocketApi.connect("socket", {
    address: "127.0.0.1",
    port: 50000,
    mode: "Buffer",
    timeout: 5000,
    bufferSize: 256
});
```

onHook:

```
TcpSocketApi.writeAndWait("socket", [0x01, 0x02, 0x03], "req1", 5000);
```

onMessage:

```
if (InDriver.taskName() === InDriver.messageSender()) {
    let data = JSON.parse(InDriver.messageData());
    if (JSON.stringify(data.bytes) === "[1,2,3]") {
        TcpSocketApi.acceptRead(data.socketName);
    }
}
```

TcpServerApi

TcpServerApi provides TCP server functionality for JavaScript tasks in InDriver.

It allows:

- listening for incoming TCP connections,
- receiving data asynchronously through onMessage,
- writing replies back to a connected client socket.

The current documented API surface contains:

- listen
- write
- close

TcpSocketApi.close

Signature

```
TcpSocketApi.close(name: String): void
```

Description

Closes and removes the TCP socket instance associated with the given name.

Unlike disconnect(...), which only closes the connection, close(...):

- fully removes the socket from internal management,
- releases all associated resources,
- makes the socket name available for reuse without implicit state.

Arguments

name

- type: String
- socket name

Returns

- void

Behavior

- if socket exists:
  - connection is closed,
  - socket object is removed,
- if socket does not exist:
  - function has no effect

Example

```
TcpSocketApi.close("socket");
```

Example — safe lifecycle

onStartup:

```
InDriver.import("TcpSocketApi");

TcpSocketApi.connect("socket", {
   address: "127.0.0.1",
   port: 50000
});
```

onShutdown:

```
TcpSocketApi.close("socket");
```

Difference: disconnect vs close

Function	Behavior
disconnect(name)	closes connection but keeps socket object
close(name)	closes connection AND removes socket

Rules

When generating code:

1. Use disconnect(...) when:
    - you plan to reconnect later using same socket instance
2. Use close(...) when:
    - socket is no longer needed
    - task is shutting down
    - you want clean state reset
3. Prefer close(...) in:

onShutdown()

4. Do NOT call write(...) or writeAndWait(...) after close(...)

TcpServerApi.listen

Signature

```
TcpServerApi.listen(tcpServerCfg: String): void
```

Description

Starts a TCP server and listens for incoming connections.

Arguments

tcpServerCfg

- type: String
- JSON configuration

Supported configuration keys from the manual:

- mode
- address
- port
- bufferSize

Supported mode values

- "Ignore"
- "Flow" (default)
- "Buffer"

Supported address values

- "Any" (default)
- "AnyIPv4"
- "AnyIPv6"
- "LocalHost"
- "LocalHostIPv6"
- "Broadcast"

port

- valid TCP port number
- default: 50000

bufferSize

- used when mode is "Buffer"

Returns

- void

Example

```
InDriver.import("TcpServerApi");

TcpServerApi.listen(
  '{"port":50000,"address":"Any","readMode":"Buffer","bufferSize":256}'
);
```

Rules

- Start the server in onStartup().
- Use Any when the server should accept remote connections.
- Use LocalHost when the server is intended for local-only communication.

TcpServerApi.write

Signature

```
TcpServerApi.write(name: String, data: ByteArray, timeout: Number): Boolean
```

Description

Writes data to the TCP client socket associated with the given connection name.

The manual describes this as sending a reply to a connected socket and waiting until the answer is accepted or timeout occurs, with behavior depending on the configured server read mode.

Arguments

name

- type: String
- socket/connection name, typically taken from received message payload (data.socketName)

data

- type: ByteArray

timeout

- type: Number
- timeout in milliseconds

Returns

Boolean

- true → write succeeded
- false → write failed

Example

onMessage:

```
if (InDriver.taskName() === InDriver.messageSender()) {
    let data = JSON.parse(InDriver.messageData());

    if (JSON.stringify(data.bytes) === "[1,2,3]") {
        TcpServerApi.write(data.socketName, "bytes accepted", 5000);
    }

    if (data.data === "text data") {
        TcpServerApi.write(data.socketName, "text data accepted", 5000);
    }
}
```

Rules for AI

- Use data.socketName from incoming onMessage payload when replying.
- Validate received bytes or text before writing a reply.

TcpServerApi.onMessage behavior

Description

When the server receives data from a connected client, the task onMessage() handler is triggered.

The payload structure follows the same documented pattern as TCP socket reads:

```
let d = JSON.parse(InDriver.messageData());

/*
d =
{
 "bytes": "[79,75]",
 "data": "OK",
 "request": "request name",
 "socketName": "name of socket"
}
*/
```

Rules for AI

- Parse incoming data with JSON.parse(InDriver.messageData()).
- Use socketName when sending server replies.

Recommended usage patterns

Pattern 1 — TCP client socket request/response

onStartup

```
InDriver.import("TcpSocketApi");
TcpSocketApi.connect("socket", {
   address: "127.0.0.1",
   port: 50000,
   mode: "Buffer",
   timeout: 5000,
```

```
    bufferSize: 256
});
```

onHook

```
TcpSocketApi.writeAndWait("socket", [0x01, 0x02, 0x03], "req1", 5000);
```

onMessage

```
if (InDriver.taskName() === InDriver.messageSender()) {
    let data = JSON.parse(InDriver.messageData());
    if (JSON.stringify(data.bytes) === "[1,2,3]") {
        TcpSocketApi.acceptRead(data.socketName);
    }
}
```

Pattern 2 — TCP server echo/ack

onStartup

```
InDriver.import("TcpServerApi");
TcpServerApi.listen('{"port":50000,"address":"Any","mode":"Buffer","bufferSize":256}');
```

onMessage

```
if (InDriver.taskName() === InDriver.messageSender()) {
    let data = JSON.parse(InDriver.messageData());
    TcpServerApi.write(data.socketName, "accepted", 5000);
}
```

Rules for AI code generation

When generating code using TcpSocketApi and TcpServerApi, follow these rules:

1. Import the correct module in onStartup():

```
InDriver.import("TcpSocketApi");
InDriver.import("TcpServerApi");
```

2. For TCP client sockets:

- call connect(...) before write(...) or writeAndWait(...)

- use writeAndWait(...) only for request/response protocols
- use acceptRead(...) in onMessage() when the expected response is received

3. For TCP server:

- call listen(...) in onStartup()
- reply using TcpServerApi.write(data.socketName, ...)

4. Always parse incoming socket data:

```
let data = JSON.parse(InDriver.messageData());
```

5. Use "Flow" or "Buffer" depending on whether the protocol is chunk-based or whole-buffer based.
6. Do not invent additional socket methods beyond the documented API surface. The current user-facing API from the attached files/manual is limited to the methods listed above.

UdpSocketApi

UdpSocketApi provides UDP socket communication for JavaScript tasks in InDriver.

It allows:

- creating named UDP sockets,
- binding sockets to local address and port,
- sending datagrams,
- sending datagrams and waiting for an accepted response,
- receiving incoming datagrams asynchronously through onMessage,
- checking whether a socket is busy with a pending request.

Sockets are managed by a unique name.

UdpSocketApi.acceptRead

Signature

```
UdpSocketApi.acceptRead(name: String): void
```

Description

Marks the current incoming datagram as accepted for the specified socket.

This function is mainly used together with UdpSocketApi.writeAndWait(...). The waiting operation can complete when the script decides that the received response is the expected one and explicitly calls acceptRead(...).

Arguments

name

- type: String
- socket name

Returns

- void

Example

onMessage:

```
if (InDriver.taskName() === InDriver.messageSender()) {
   let data = JSON.parse(InDriver.messageData());
   if (data.socketName === "udp1" && data.data === "OK") {
```

```
        UdpSocketApi.acceptRead("udp1");
    }
}
```

Rules

- Use acceptRead(...) only when request/response behavior is needed.
- Call it only after validating the received message.

UdpSocketApi.bind

Signature

```
UdpSocketApi.bind(name: String, udpCfg: String = ""): Boolean
UdpSocketApi.bind(name: String, udpCfg: Object): Boolean
```

Description

Creates or recreates a named UDP socket and binds it to a local address and port.

If a socket with the same name already exists, it is closed and reopened with the new configuration.

Arguments

name

- type: String
- unique socket identifier

udpCfg

- type: String | Object
- JSON socket configuration

Supported keys visible in the source:

- address
- port
- timeout

Configuration fields

address

- type: String
- local bind address

port

- type: Number
- local UDP port

timeout

- type: Number
- timeout in milliseconds
- default: 30000 if omitted

Returns

Boolean

- true → socket bound successfully
- false → bind failed

Example

```
InDriver.import("UdpSocketApi");

let ok = UdpSocketApi.bind("udp1", {
  address: "0.0.0.0",
  port: 50000,
  timeout: 5000
});

if (!ok) {
  InDriver.debug("UDP bind failed", "critical");
}
```

Rules

- Bind the socket in onStartup().
- Use stable socket names.
- Rebinding the same name replaces the old socket.

UdpSocketApi.isBusy

Signature

```
UdpSocketApi.isBusy(name: String): Boolean
```

Description

Returns whether the specified UDP socket is currently busy.

Arguments

name

- type: String
- socket name

Returns

Boolean

- true → socket is busy
- false → socket is idle or not found

Example

```
if (!UdpSocketApi.isBusy("udp1")) {
  UdpSocketApi.write("udp1", "192.168.0.10", 6000, [0x01, 0x02], "req1", 5000);
}
```

Rules

- Use before sending a new request if only one outstanding request should exist.

UdpSocketApi.write

Signature

```
UdpSocketApi.write(
  name: String,
  address: String,
  port: Number,
  data: ByteArray,
  request: String = "",
  timeout: Number = 0
): Boolean
```

Description

Sends a UDP datagram to the specified remote address and port.

Arguments

name

- type: String
- bound socket name

address

- type: String
- target IP address or hostname

port

- type: Number
- target UDP port

data

- type: ByteArray
- bytes to send

Typical JavaScript usage:

[0x01, 0x02, 0x03]

request

- type: String
- optional request identifier

timeout

- type: Number
- optional timeout in milliseconds

Returns

Boolean

- true → datagram send initiated successfully
- false → socket not found or send failed

Example

```
UdpSocketApi.write(
  "udp1",
  "192.168.0.10",
  6000,
```

```
  [0x01, 0x02, 0x03],
  "ping1",
  5000
);
```

Rules

- Use for non-blocking UDP send logic.
- Handle replies asynchronously in onMessage.

UdpSocketApi.writeAndWait

Signature

```
UdpSocketApi.writeAndWait(
  name: String,
  address: String,
  port: Number,
  data: ByteArray,
  request: String = "",
  timeout: Number = 0
): Boolean
```

Description

Sends a UDP datagram and waits until a matching response is accepted or the timeout expires.

Arguments

name

- type: String

address

- type: String

port

- type: Number

data

- type: ByteArray

request

- type: String
- optional request identifier

timeout

- type: Number
- timeout in milliseconds

Returns

Boolean

- true → request completed successfully
- false → socket not found, send failed, or timeout/accept failed

Example

onHook

```
UdpSocketApi.writeAndWait(
    "udp1",
    "192.168.0.10",
    6000,
    [0x10, 0x20],
    "req1",
    5000
);
```

onMessage

```
if (InDriver.taskName() === InDriver.messageSender()) {
    let data = JSON.parse(InDriver.messageData());
    if (data.socketName === "udp1" && data.data === "OK") {
        UdpSocketApi.acceptRead("udp1");
    }
}
```

Rules

- Use only when the protocol has request/response semantics.
- Validate incoming data before calling acceptRead(...).

UdpSocketApi.onMessage behavior

Description

When a UDP datagram is received, onMessage() is triggered. The source files show that the runtime sends a direct internal message with tag "SockedData" and inserts the last request id under key "request" if available.

The incoming payload should be treated as JSON text available through:

let data = JSON.parse(InDriver.messageData());

Typical fields include:

- socketName
- request
- received bytes / textual representation of data

Rules

- Parse InDriver.messageData() as JSON.
- Use data.request to correlate replies when needed.

XmlReaderApi

XmlReaderApi provides streaming XML reading functionality for JavaScript tasks in InDriver.

It allows:

- opening an XML file,
- adding XML data manually,
- stepping through XML tokens,
- reading element names, text, attributes, namespaces, DTD data, and processing instructions,
- checking parser state and position,
- reading element text with configurable behavior.

The API is stateful and operates on the currently loaded XML reader state.

XmlReaderApi.addData

Signature

```
XmlReaderApi.addData(data: String): void
```

Description

Adds XML text directly into the XML reader.

Arguments

data

- type: String
- XML text fragment

Returns

- void

Example

```
XmlReaderApi.clear();
XmlReaderApi.addData("<root><a>1</a></root>");
```

Rules

- Use this when XML comes from memory, not from file.

- Call clear() first if you want a clean parser state.

XmlReaderApi.addExtraNamespaceDeclaration

Signature

```
XmlReaderApi.addExtraNamespaceDeclaration(prefix: String, namespaceUri: String): void
```

Description

Adds an extra namespace declaration to the XML reader.

Arguments

prefix

- type: String

namespaceUri

- type: String

Returns

- void

Example

```
XmlReaderApi.addExtraNamespaceDeclaration("x", "http://example.com/ns");
```

XmlReaderApi.atEnd

Signature

```
XmlReaderApi.atEnd(): Boolean
```

Description

Returns whether the XML reader is at the end of the stream.

Returns

Boolean

Example

```
while (!XmlReaderApi.atEnd()) {
  XmlReaderApi.readNext();
}
```

XmlReaderApi.attribute

Signature

```
XmlReaderApi.attribute(attr: String): String
```

Description

Returns the value of the specified attribute for the current element.

Arguments

attr

- type: String
- attribute name

Returns

String

Example

```
if (XmlReaderApi.isStartElement()) {
  let id = XmlReaderApi.attribute("id");
}
```

XmlReaderApi.characterOffset

Signature

```
XmlReaderApi.characterOffset(): Number
```

Description

Returns the character offset of the current parser position.

Returns

Number

XmlReaderApi.clear

Signature

```
XmlReaderApi.clear(): void
```

Description

Clears the XML reader state.

Returns

- void

Example

```
XmlReaderApi.clear();
```

XmlReaderApi.close

Signature

```
XmlReaderApi.close(): void
```

Description

Closes the currently opened XML file.

Returns

- void

Example

```
XmlReaderApi.close();
```

XmlReaderApi.columnNumber

Signature

```
XmlReaderApi.columnNumber(): Number
```

Description

Returns the current parser column number.

Returns

Number

XmlReaderApi.documentEncoding

Signature

```
XmlReaderApi.documentEncoding(): String
```

Description

Returns the document encoding reported by the XML stream.

Returns

String

XmlReaderApi.documentVersion

Signature

```
XmlReaderApi.documentVersion(): String
```

Description

Returns the XML document version.

Returns

String

XmlReaderApi.dtdName

Signature

```
XmlReaderApi.dtdName(): String
```

Description

Returns the DTD name of the current document, if present.

Returns

String

XmlReaderApi.dtdPublicId

Signature

```
XmlReaderApi.dtdPublicId(): String
```

Description

Returns the DTD public identifier.

Returns

String

XmlReaderApi.dtdSystemId

Signature

```
XmlReaderApi.dtdSystemId(): String
```

Description

Returns the DTD system identifier.

Returns

String

XmlReaderApi.entityDeclarations

Signature

```
XmlReaderApi.entityDeclarations(): String
```

Description

Returns entity declarations as serialized JSON text.

Returns

String

The returned JSON contains objects with fields such as:

- name
- notationName
- publicId
- systemId
- value

Example

```
let entities = JSON.parse(XmlReaderApi.entityDeclarations());
```

Rules

- Parse the returned string with JSON.parse(...) if structured access is needed.

XmlReaderApi.entityExpansionLimit

Signature

```
XmlReaderApi.entityExpansionLimit(): Number
```

Description

Returns the current entity expansion limit.

Returns

Number

XmlReaderApi.hasError

Signature

```
XmlReaderApi.hasError(): Boolean
```

Description

Returns whether the XML reader is currently in an error state.

Returns

Boolean

Example

```
if (XmlReaderApi.hasError()) {
  InDriver.debug(XmlReaderApi.lastError(), "critical");
}
```

XmlReaderApi.hasStandaloneDeclaration

Signature

```
XmlReaderApi.hasStandaloneDeclaration(): Boolean
```

Description

Returns whether the XML document has a standalone declaration.

Returns

Boolean

XmlReaderApi.isCDATA

Signature

```
XmlReaderApi.isCDATA(): Boolean
```

Description

Returns whether the current token is CDATA.

Returns

Boolean

XmlReaderApi.isCharacters

Signature

```
XmlReaderApi.isCharacters(): Boolean
```

Description

Returns whether the current token is character data.

Returns

Boolean

XmlReaderApi.isComment

Signature

```
XmlReaderApi.isComment(): Boolean
```

Description

Returns whether the current token is a comment.

Returns

Boolean

XmlReaderApi.isDTD

Signature

```
XmlReaderApi.isDTD(): Boolean
```

Description

Returns whether the current token is a DTD token.

Returns

Boolean

XmlReaderApi.isEndDocument

Signature

```
XmlReaderApi.isEndDocument(): Boolean
```

Description

Returns whether the current token is end-of-document.

Returns

Boolean

XmlReaderApi.isEndElement

Signature

```
XmlReaderApi.isEndElement(): Boolean
```

Description

Returns whether the current token is an end element.

Returns

Boolean

XmlReaderApi.isEntityReference

Signature

```
XmlReaderApi.isEntityReference(): Boolean
```

Description

Returns whether the current token is an entity reference.

Returns

Boolean

XmlReaderApi.isProcessingInstruction

Signature

```
XmlReaderApi.isProcessingInstruction(): Boolean
```

Description

Returns whether the current token is a processing instruction.

Returns

Boolean

XmlReaderApi.isStandaloneDocument

Signature

```
XmlReaderApi.isStandaloneDocument(): Boolean
```

Description

Returns whether the XML document is standalone.

Returns

Boolean

XmlReaderApi.isStartDocument

Signature

```
XmlReaderApi.isStartDocument(): Boolean
```

Description

Returns whether the current token is the start of the document.

Returns

Boolean

XmlReaderApi.isStartElement

Signature

```
XmlReaderApi.isStartElement(): Boolean
```

Description

Returns whether the current token is a start element.

Returns

Boolean

XmlReaderApi.isWhitespace

Signature

```
XmlReaderApi.isWhitespace(): Boolean
```

Description

Returns whether the current token is whitespace.

Returns

Boolean

XmlReaderApi.lastError

Signature

```
XmlReaderApi.lastError(): String
```

Description

Returns the current XML parser error string.

Returns

String

Example

```
if (XmlReaderApi.hasError()) {
  InDriver.debug(XmlReaderApi.lastError(), "critical");
}
```

XmlReaderApi.lineNumber

Signature

```
XmlReaderApi.lineNumber(): Number
```

Description

Returns the current parser line number.

Returns

Number

XmlReaderApi.name

Signature

```
XmlReaderApi.name(): String
```

Description

Returns the current element or token name.

Returns

String

Example

```
if (XmlReaderApi.isStartElement()) {
  InDriver.debug(XmlReaderApi.name());
}
```

XmlReaderApi.namespaceProcessing

Signature

```
XmlReaderApi.namespaceProcessing(): Boolean
```

Description

Returns whether namespace processing is enabled.

Returns

Boolean

XmlReaderApi.namespaceUri

Signature

```
XmlReaderApi.namespaceUri(): String
```

Description

Returns the namespace URI of the current token.

Returns

String

XmlReaderApi.open

Signature

```
XmlReaderApi.open(file: String): String
```

Description

Opens an XML file for reading.

Arguments

file

- type: String
- path to XML file

Returns

String

- empty string if opening succeeds and there is no current parser error
- error text such as:
    - "Failed to open file: <path>"
    - or current parser error text

Example

```
let err = XmlReaderApi.open("c:/data/config.xml");
if (err) {
  InDriver.debug(err, "critical");
}
```

Rules

- Treat non-empty return value as an error or parser status message.
- Open the file before calling readNext().

XmlReaderApi.prefix

Signature

```
XmlReaderApi.prefix(): String
```

Description

Returns the namespace prefix of the current token.

Returns

String

XmlReaderApi.processingInstructionData

Signature

```
XmlReaderApi.processingInstructionData(): String
```

Description

Returns the data part of the current processing instruction.

Returns

String

XmlReaderApi.processingInstructionTarget

Signature

```
XmlReaderApi.processingInstructionTarget(): String
```

Description

Returns the target part of the current processing instruction.

Returns

String

XmlReaderApi.qualifiedName

Signature

```
XmlReaderApi.qualifiedName(): String
```

Description

Returns the qualified name of the current token.

Returns

String

XmlReaderApi.readElementText

Signature

```
XmlReaderApi.readElementText(
  behaviour: String = "ErrorOnUnexpectedElement"
): String
```

Description

Reads and returns the text content of the current element.

Arguments

behaviour

- type: String
- optional
- supported values:
  - "ErrorOnUnexpectedElement"
  - "IncludeChildElements"
  - "SkipChildElements"

Returns

String

- element text
- empty string if an unsupported behavior value is used

Example

```
if (XmlReaderApi.isStartElement() && XmlReaderApi.name() === "title") {
  let text = XmlReaderApi.readElementText("SkipChildElements");
  InDriver.debug(text);
}
```

Rules

- Use "SkipChildElements" when nested markup should not cause an error.
- Use "IncludeChildElements" if child text must be preserved.

XmlReaderApi.readNext

Signature

```
XmlReaderApi.readNext(): String
```

Description

Advances the XML reader to the next token and returns the token type as string.

Returns

String

Possible values include:

- "NoToken"

- "Invalid"
- "StartDocument"
- "EndDocument"
- "StartElement"
- "EndElement"
- "Characters"
- "Comment"
- "DTD"
- "EntityReference"
- "ProcessingInstruction"

Example

```
while (!XmlReaderApi.atEnd()) {
  let token = XmlReaderApi.readNext();
  InDriver.debug(token);
}
```

XmlReaderApi.readNextStartElement

Signature

```
XmlReaderApi.readNextStartElement(): Boolean
```

Description

Advances the reader until the next start element is found.

Returns

Boolean

- true → start element found
- false → no more start elements or error

Example

```
while (XmlReaderApi.readNextStartElement()) {
  InDriver.debug(XmlReaderApi.name());
}
```

XmlReaderApi.setEntityExpansionLimit

Signature

```
XmlReaderApi.setEntityExpansionLimit(limit: Number): void
```

Description

Sets the entity expansion limit of the XML reader.

Arguments

limit

- type: Number

Returns

- void

XmlReaderApi.setNamespaceProcessing

Signature

```
XmlReaderApi.setNamespaceProcessing(enabled: Boolean): void
```

Description

Enables or disables namespace processing.

Arguments

enabled

- type: Boolean

Returns

- void

Example

```
XmlReaderApi.setNamespaceProcessing(true);
```

XmlReaderApi.skipCurrentElement

Signature

```
XmlReaderApi.skipCurrentElement(): void
```

Description

Skips the current element, including its child content.

Returns

- void

Example

```
if (XmlReaderApi.isStartElement() && XmlReaderApi.name() === "ignore") {
  XmlReaderApi.skipCurrentElement();
}
```

XmlReaderApi.text

Signature

```
XmlReaderApi.text(): String
```

Description

Returns the text of the current token.

Returns

String

Example

```
if (XmlReaderApi.isCharacters()) {
  InDriver.debug(XmlReaderApi.text());
}
```

XmlReaderApi.tokenString

Signature

```
XmlReaderApi.tokenString(): String
```

Description

Returns the current token as a string representation provided by the XML reader.

Returns

String

XmlReaderApi.tokenType

Signature

```
XmlReaderApi.tokenType(): String
```

Description

Returns the current token type as a normalized token name.

Returns

String

Possible values include:

- "NoToken"
- "Invalid"
- "StartDocument"
- "EndDocument"
- "StartElement"
- "EndElement"
- "Characters"
- "Comment"
- "DTD"
- "EntityReference"
- "ProcessingInstruction"

Recommended usage patterns

Pattern 1 — read XML file token by token

onStartup:

```
InDriver.import("XmlReaderApi");

let err = XmlReaderApi.open("c:/data/config.xml");
if (err) {
   InDriver.debug(err, "critical");
   return;
}

while (!XmlReaderApi.atEnd()) {
   let token = XmlReaderApi.readNext();

   if (XmlReaderApi.isStartElement()) {
      InDriver.debug("Element: " + XmlReaderApi.name());
   }

   if (XmlReaderApi.isCharacters()) {
      InDriver.debug("Text: " + XmlReaderApi.text());
   }
}

if (XmlReaderApi.hasError()) {
   InDriver.debug(XmlReaderApi.lastError(), "critical");
}

XmlReaderApi.close();
```

Pattern 2 — read selected element text

```
while (XmlReaderApi.readNextStartElement()) {
  if (XmlReaderApi.name() === "title") {
     let title = XmlReaderApi.readElementText("SkipChildElements");
     InDriver.debug("Title: " + title);
  } else {
     XmlReaderApi.skipCurrentElement();
  }
}
```

Pattern 3 — parse XML from memory

```
XmlReaderApi.clear();
XmlReaderApi.addData("<root><item id='1'>A</item></root>");
```

```
while (!XmlReaderApi.atEnd()) {
  XmlReaderApi.readNext();

  if (XmlReaderApi.isStartElement()) {
     InDriver.debug(XmlReaderApi.name());
     InDriver.debug(XmlReaderApi.attribute("id"));
  }
}
```

Rules for AI code generation

When generating code using UdpSocketApi and XmlReaderApi, follow these rules:

1. For UdpSocketApi:
    - bind sockets in onStartup()
    - use write(...) for non-blocking sends
    - use writeAndWait(...) only when request/response confirmation is needed
    - parse incoming UDP payload from InDriver.messageData()
    - call acceptRead(...) only after validating the received response
2. For XmlReaderApi:
    - open the file before reading
    - use readNext() or readNextStartElement() to advance the parser
    - use isStartElement(), isCharacters(), text(), attribute() to inspect content
    - check hasError() and lastError() after parsing
    - call close() when done
3. Do not invent additional methods beyond the attached source files. UdpSocketApi exposes only:
    - bind
    - write
    - writeAndWait
    - acceptRead
    - isBusy

    and XmlReaderApi exposes only the methods listed in jsxmlapi.h.