Innovative Data Analytics

last modification: **2024/02/02**

InDriver Documentation

Innovative Data Analytics **4**
Introduction to InDriver **5**
Getting started - download, installation, requirements **5**
Installation 6
Launch 7
License 8
Setup 11
InServer 11
Configure SQL Servers 12
Install InServerAPI 14
Install the Additional API(s) 15
InDriver first steps 17
Configuring SQL Servers 17
Configuring first task 18
First ‘Hello world’ script 19
Features and examples **20**
InDriver JS Tasks **21**
InDriver JS API **24**
Detailed description: 24
● Null InDriver.debug(String debugMsg); 24
● Null InDriver.debug(String debugMsg, String msgType); 24
● String InDriver.driverName( ); 24
● String InDriver.configuration(String Array path); 24
● String InDriver.currentPath( ); 25
● String InDriver.hookTs(Number hook); 26
● String InDriver.hookTs( ); 26
● Boolean InDriver.import(String api); 26
● Null InDriver.installHook(Number hook); 26
● Boolean InDriver.isHook(Number hook); 27
● String InDriver.messageData( ); 27
● String InDriver.messageSender( ); 27
● String InDriver.messageTs( ); 28

- String InDriver.messageTags( ); 28
- Null InDriver.sendMessage(String dt, String tags, String data); 28
- Null InDriver.setFlag(String flag, String info); 29
- String InDriver.sqlExecute(String server, String query); 29
- Boolean InDriver.sqlExecuteAll(String query); 30
- String InDriver.taskName( ); 31

RestApi **32**

Detailed description: 35

- Null RestApi.begin( ); 35
- Null RestApi.commit( ); 35
- Null RestApi.commitWait( ); 35
- Null RestApi.defineRequest(String request, String def); 35
- Boolean RestApi.getData(String request); 36
- Boolean RestApi.isSucceeded(); 36
- Boolean RestApi.sendRequest(String request); 36
- Boolean RestApi.sendRequest(String request, String def); 36
- Boolean RestApi.wait( ); 36
- Boolean RestApi.wait(Number timeout); 36

ModbusApi **38**

Detailed description: 38

- Null ModbusApi.begin( ); 39
- Null ModbusApiestApi.commit( ); 39
- Null ModbusApi.commitWait( ); 39
- Null ModbusApi.connectDevice(String device, String def); 39
- Null ModbusApi.getAllData( ); 41
- Null ModbusApi.getDeviceData(String device); 42
- Null ModbusApi.getDeviceRequestData(String device, String reg); 42
- Null ModbusApi.getDeviceRequestValue(String device, String reg, Number Array addresses); 43
- Null ModbusApi.isSucceeded( ); 43
- Null ModbusApi.readDevice(String device, String def); 44
- Null ModbusApi.wait( ); 44
- Null ModbusApi.wait(Number timeout); 45
- Null ModbusApi.writeDevice(String device, String def); 45

TsApi **47**

Detailed description: 49

- Null TsApi.aggregate(String aggregator); 49
- Null TsApi.defineAggregator(String aggregator, String server, String table, String timeZone = ‘UTC’, String Array sources=’[]’, String Array intervals =’["1m", "15m", "1h", "1d"]’, Numeric step=10000); 49

ProcessApi **51**

Detailed description: 51

- Null ProcessApi.close(String name); 51

- Null ProcessApi.closeAll( ); 51
- Null ProcessApi.kill(String name); 52
- Null ProcessApi.killAll( ); 52
- Null ProcessApi.pid(String name); 52
- Null ProcessApi.program(String name); 52
- Null ProcessApi.setWorkingDirectory(String name, String directory); 52
- Null ProcessApi.start(String name, String program, String Array args); 52
- Null ProcessApi.state(String name); 52
- Null ProcessApi.waitAllForFinished(Number msecs = 30000); 52
- Null ProcessApi.waitAllForStarted(Number msecs = 30000); 52
- Null ProcessApi.waitForFinished(String name, Number msecs); 53
- Null ProcessApi.waitForStarted(String name, Number msecs); 53
- Null ProcessApi.workingDirectory(String name); 53

FileApi **54**

Detailed description: 54

- Null FileApi.addFileSystemWatcherPath(String path); 54
- Null FileApi.close(String name); 55
- Null FileApi.closeAll( ); 55
- Null FileApi.open(String name ,String file, String Array modes); 55
- Null FileApi.readAll(String name); 55
- Null FileApi.removeFileSystemWatcherPath(String file); 55
- Null FileApi.write(String name, String data); 55

SerialPortApi **57**

Detailed description: 57

- Null SerialPortApi.acceptRead(String name); 57
- Null SerialPortApi.availablePorts( ); 57
- Null SerialPortApi.close(String name); 57
- Null SerialPortApi.closeAll( ); 58
- Null SerialPortApi.error(String name); 58
- Null SerialPortApi.open(String name, String portSettings); 58
- Number SerialPortApi.write(String name, Number Array data); 59
- Null SerialPortApi.writeAndWait(String name, String requestName, Number Array data, Number timeout); 59

TcpSocketApi **61**

Detailed description: 61

- Null TcpSocketApi.acceptRead(String name); 61
- Null TcpSocketApi.connect(String name, String socketSettings); 61
- Null TcpSocketApi.disconnect(String name); 62
- Null TcpSocketApi.disconnectAll( ); 62
- Boolean TcpSocketApi.write(String name, Number Array data); 62
- Boolean TcpSocketApi.writeAndWait(String name, String requestName, Number Array data, Number timeout); 62

TcpServerApi **64**

Detailed description: 64
- Null TcpServerApi.listen(String tcpServerCfg); 64
- Boolean TcpServerApi.write(String name, Number Array data, Number timeout); 64

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

Innovative
Data
Analytics

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.

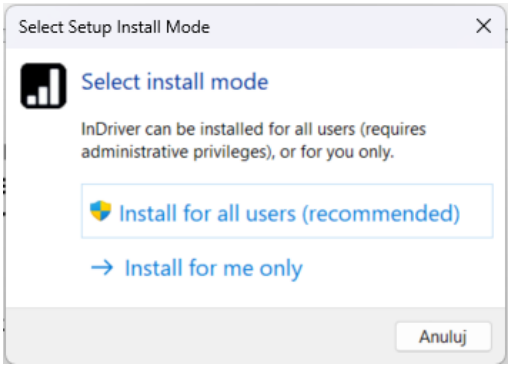
InDriver is provided as a zip archive containing the InSetup.exe installer, which installs both InDriver and InStudio.
Currently, there is an available version for **Windows operating systems**, such as:
Windows 10 and **Windows Server 2016 and later**.

Innovative
Data
Analytics

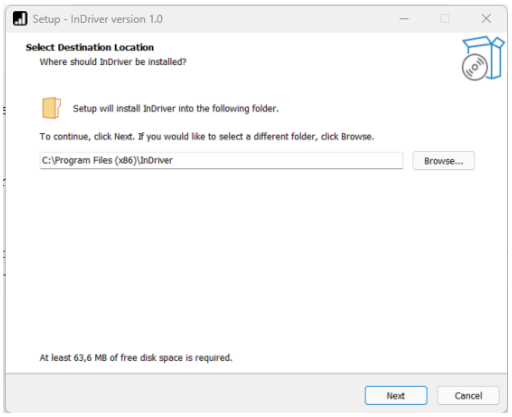
Installation

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

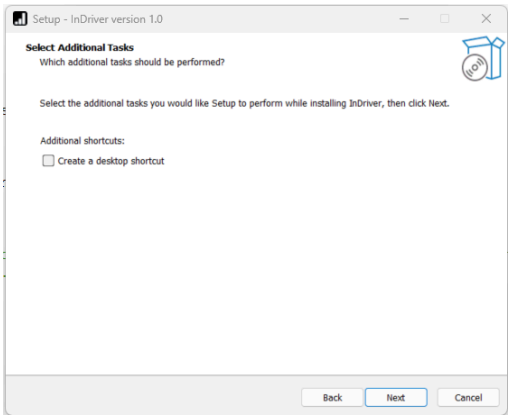
1. Install for all users



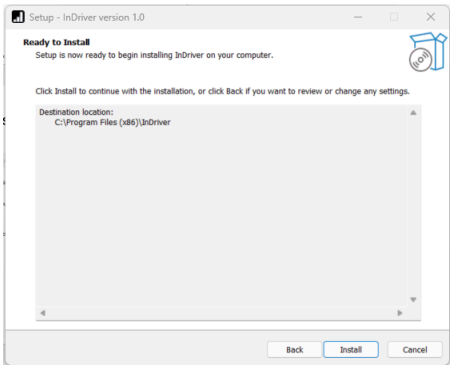
2. Select folder



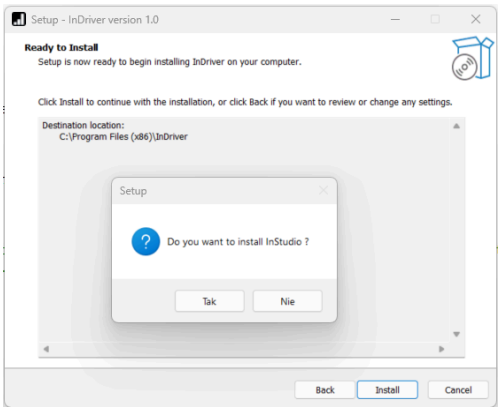
3. Create a desktop shortcut



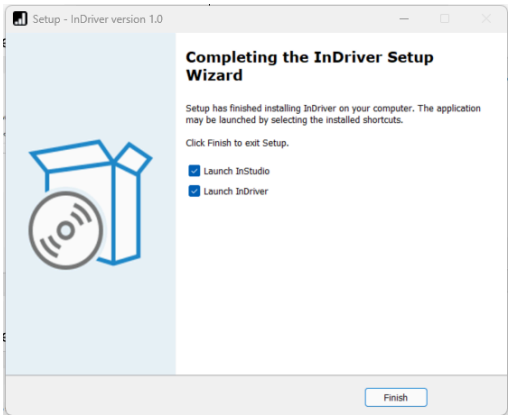
4. Confirm



5. Select to install InStudio



6. Installation complete

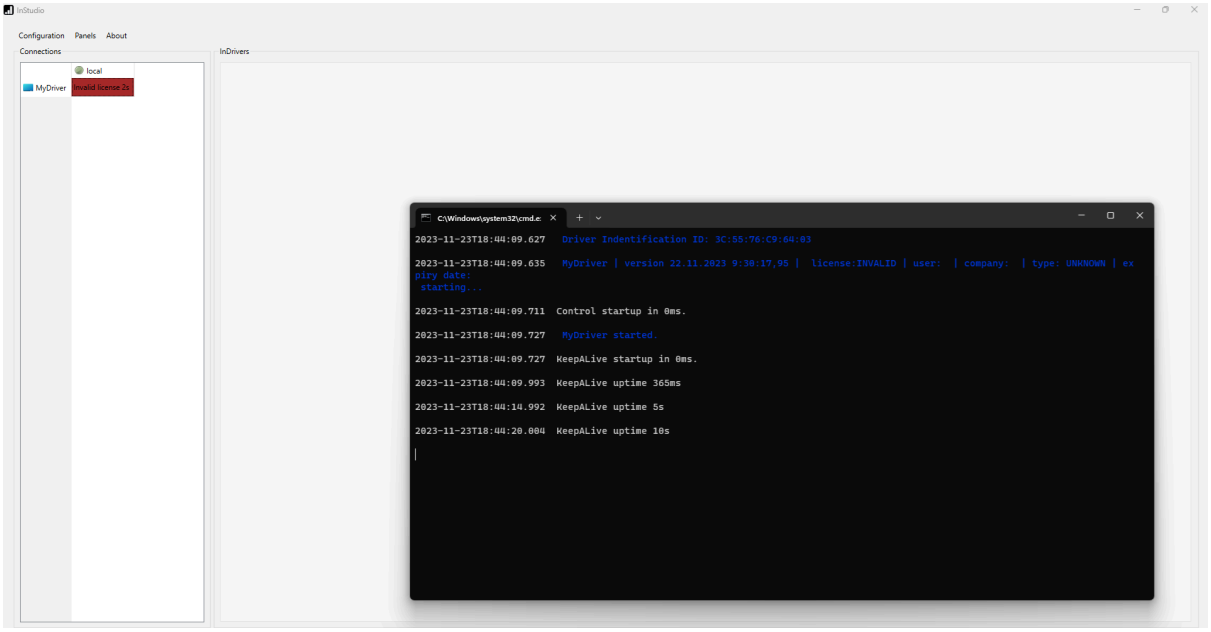


Launch InStudio and InDriver.

Innovative
Data
Analytics

Launch

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:

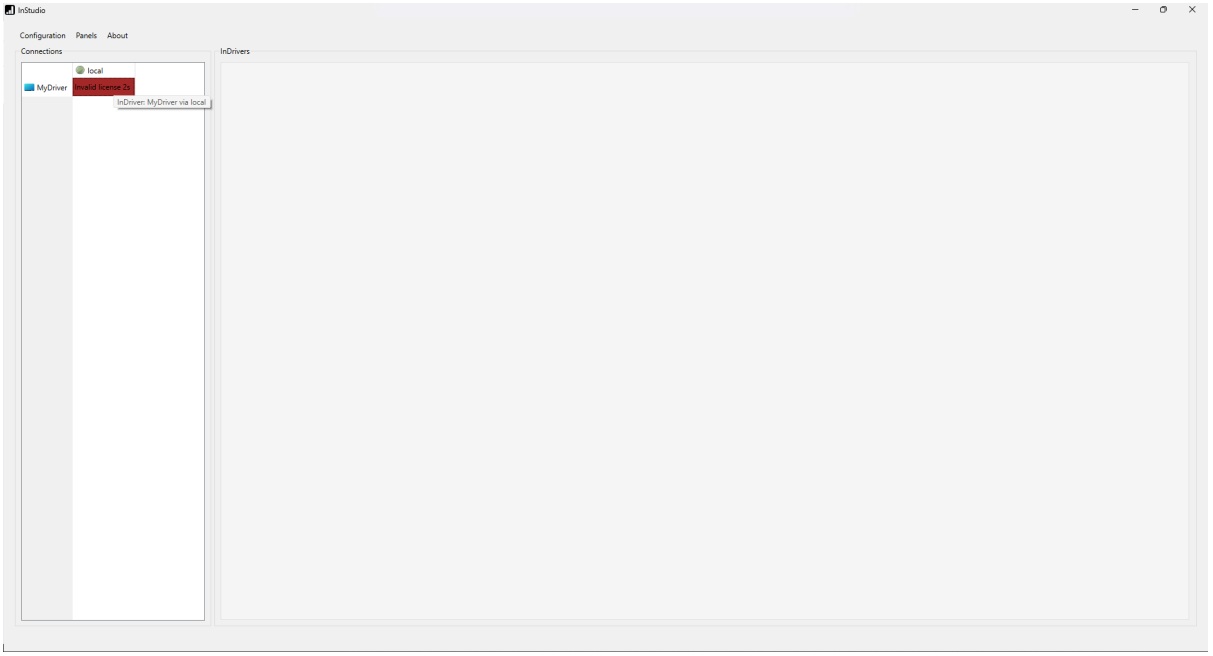


Innovative
Data
Analytics

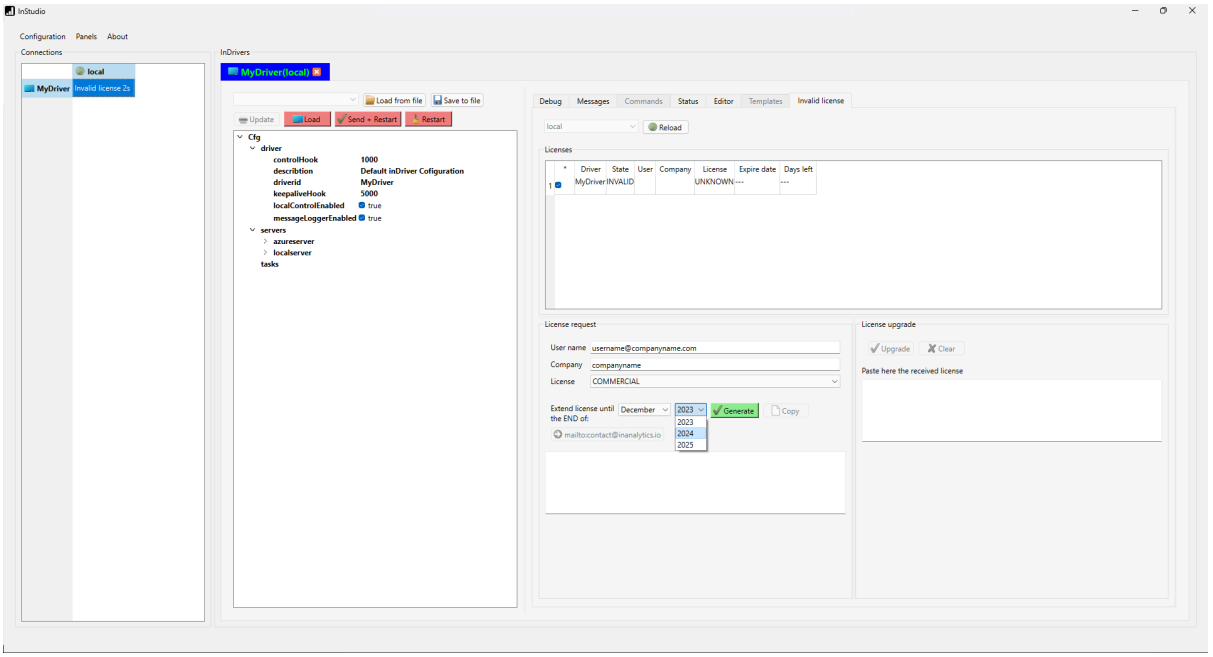
License

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.



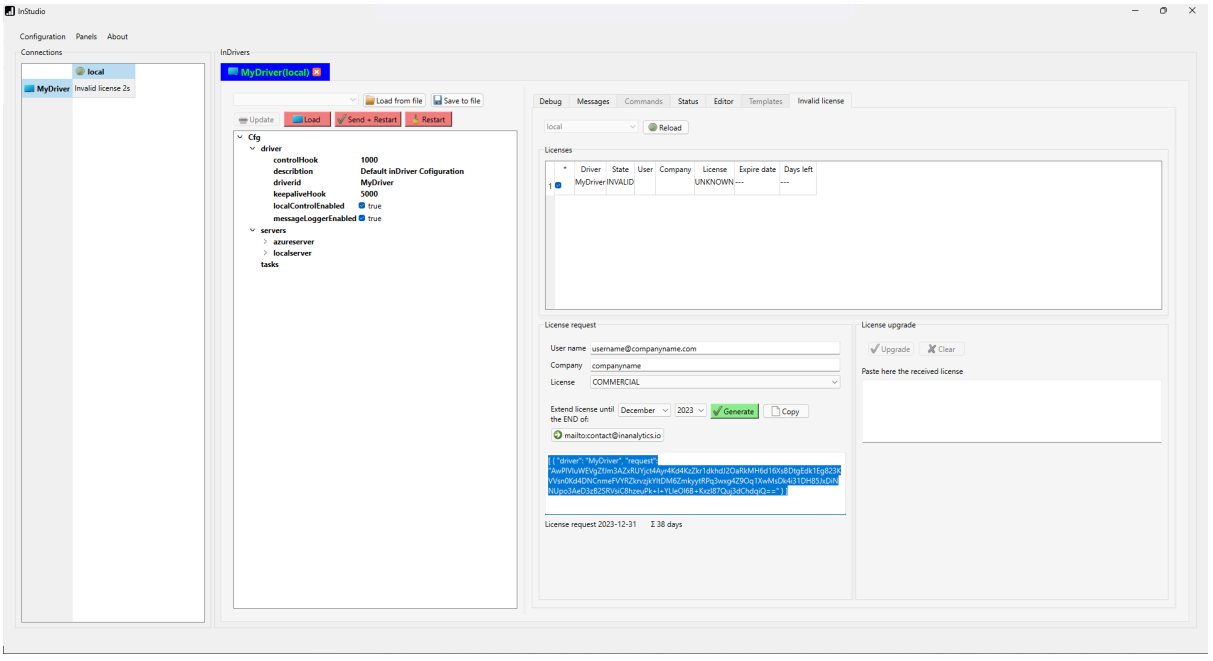
3. Press 'Generate,' copy the generated license request, and send it to contact@inanalytics.io. Alternatively, you can click the 'mailto:contact@inalaytic.io' button, and your default mail app will open.

   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.

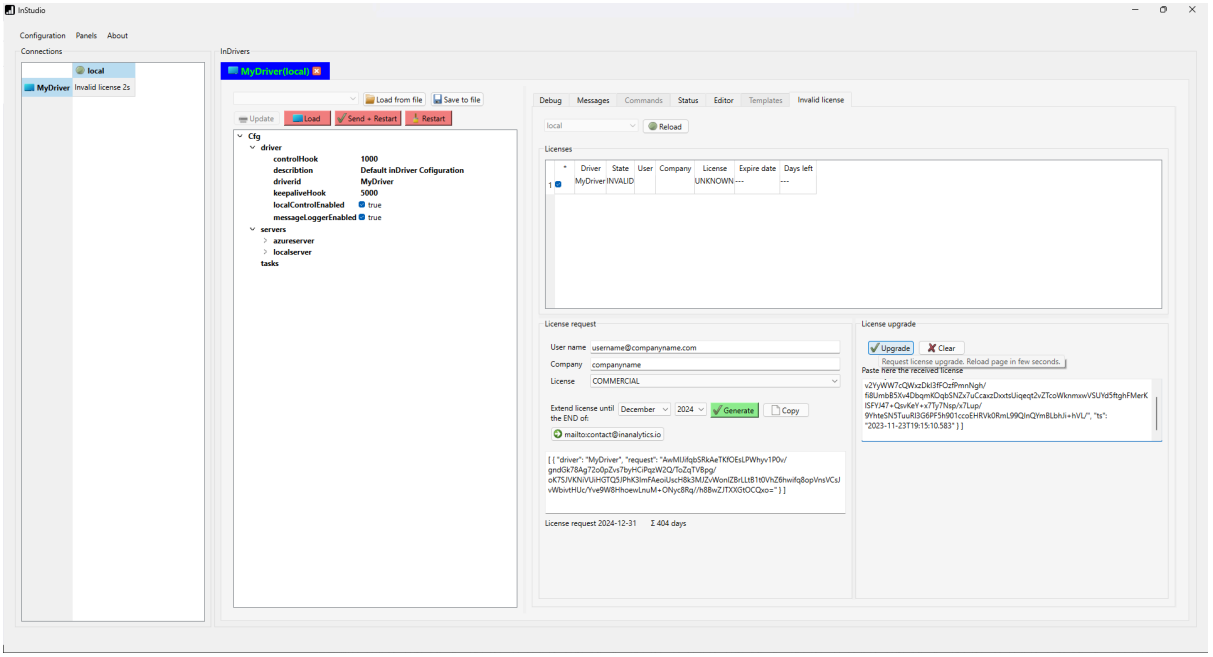
   Remark:

   - 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.

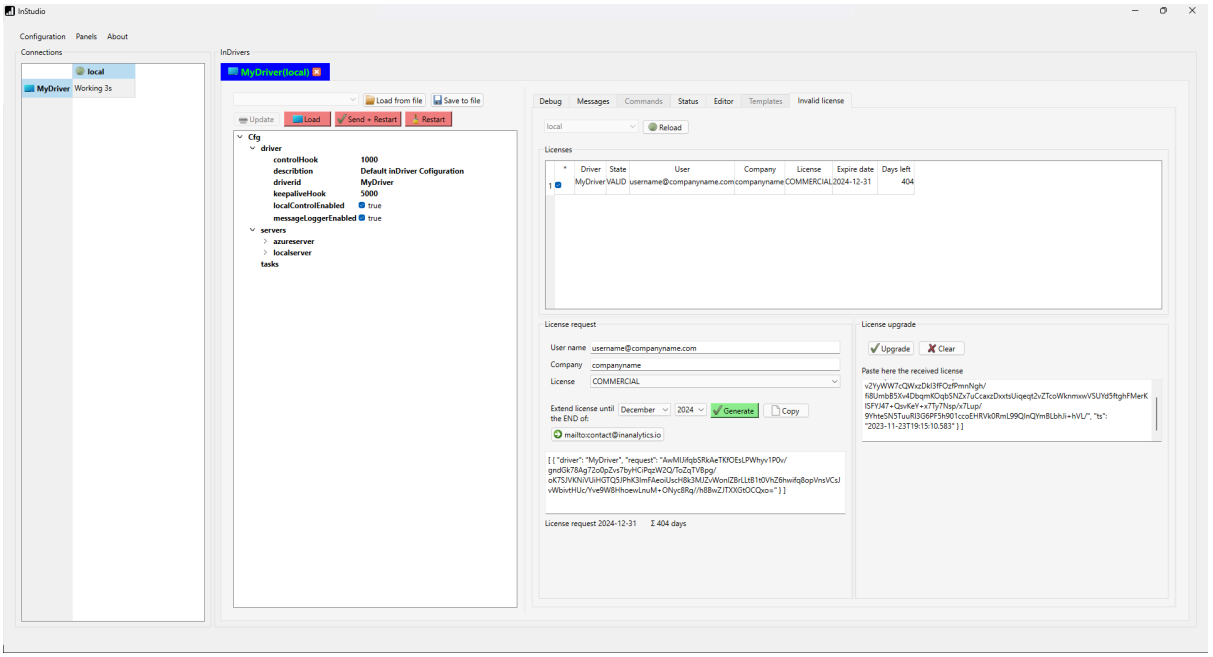


Innovative
Data
Analytics

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.



Innovative
Data
Analytics

Setup

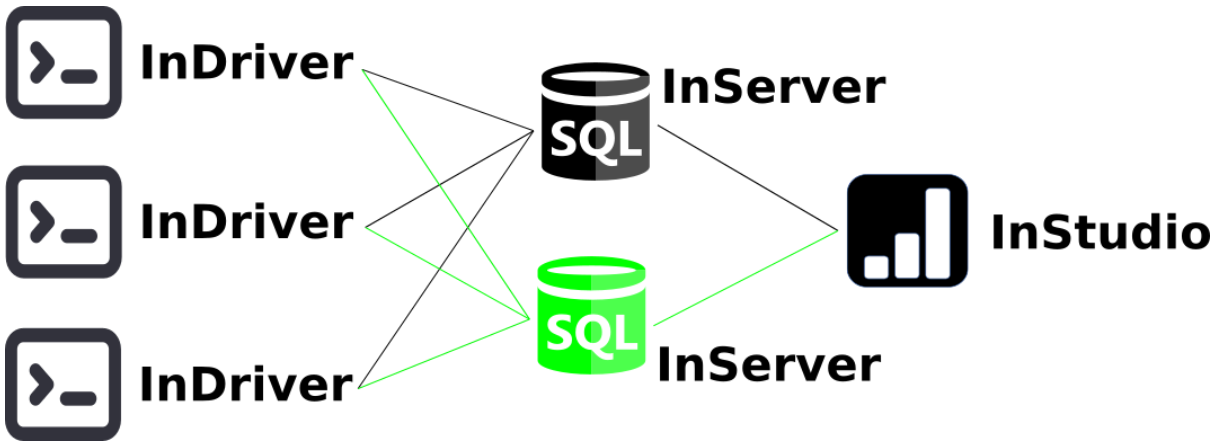
InStudio Setup allows you to configure SQL Servers, and user data, install InServer, and manage other APIs included in the InDriver package, as well as manage licenses.

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.

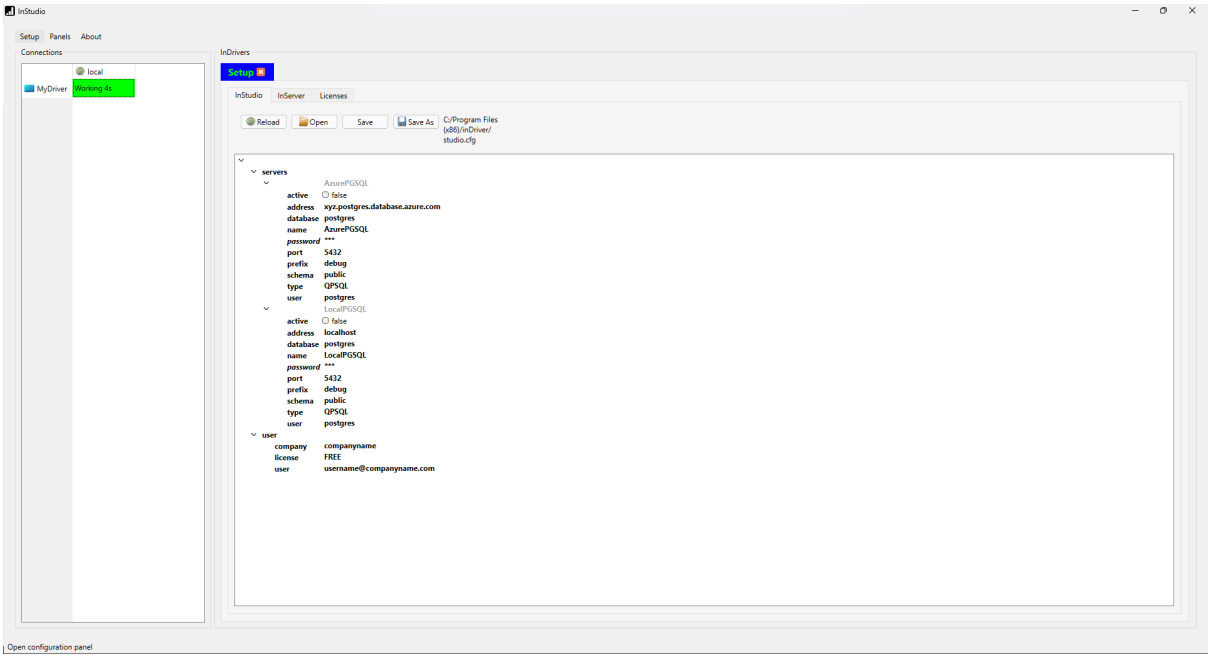
Innovative
Data
Analytics

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:

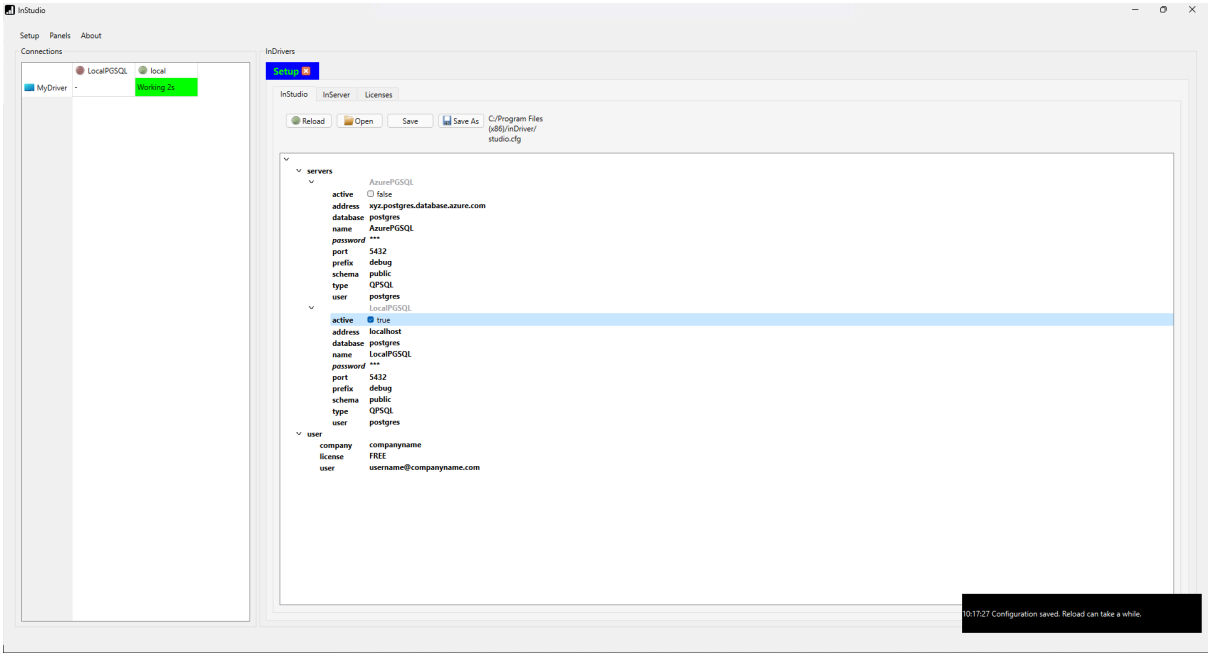
- 'Address': SQL database address
- 'Database': Database name
- 'Password': Password for the specified user (It is safe - the system uses encryption to store passwords).
- 'User': Database user name
- 'Type': Database type (choose from the following options):
    - QDB2: IBM DB2 (version 7.1 and above)
    - 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.

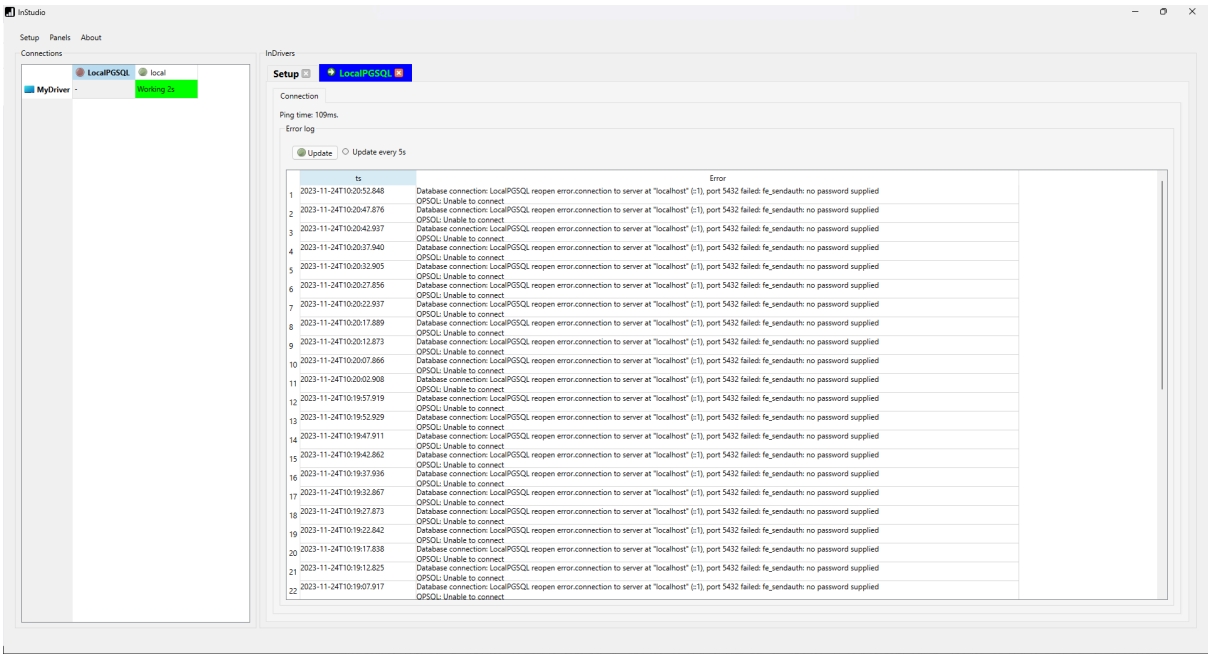


If valid data is provided, after a few seconds, the connection table on the left should display configured servers in the horizontal header, indicated in green for successful connections or red if the database connection failed.

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.



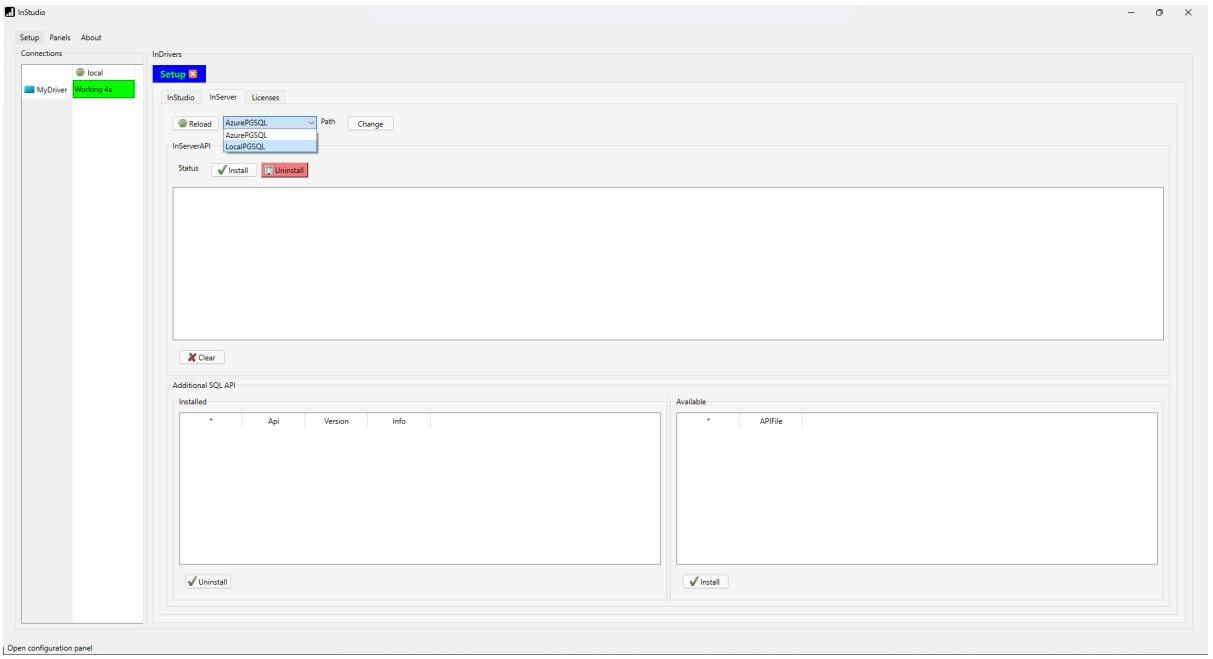
To debug the reason for any error, click on the server name, which opens a server window with a table of errors. Click 'Refresh,' and the last 100 errors will be updated.



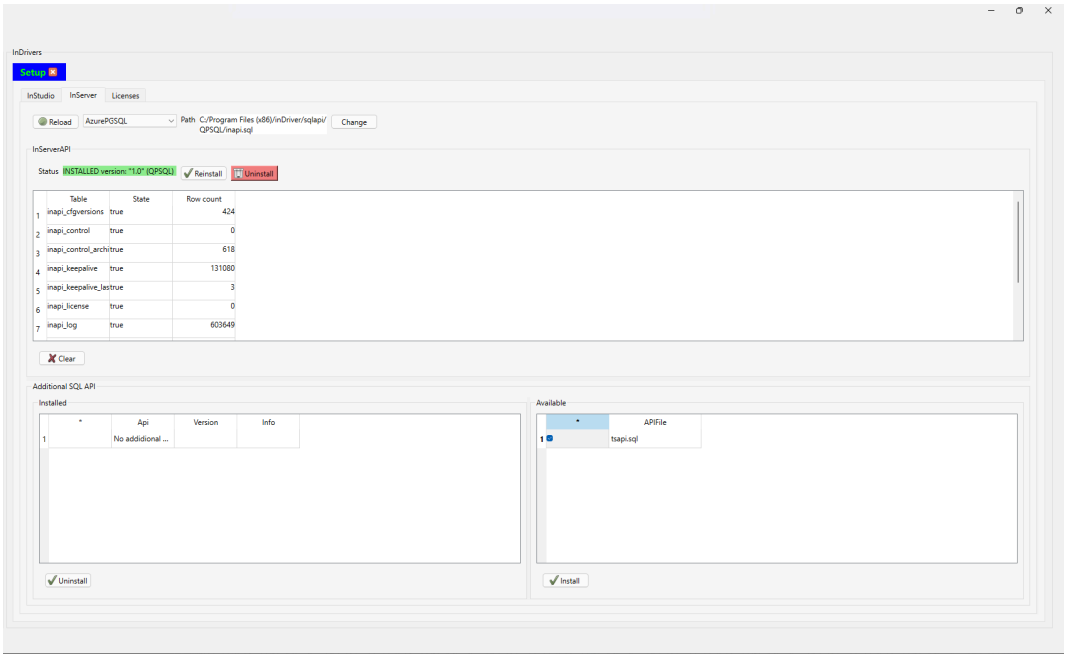
Innovative
Data
Analytics

Install InServerAPI

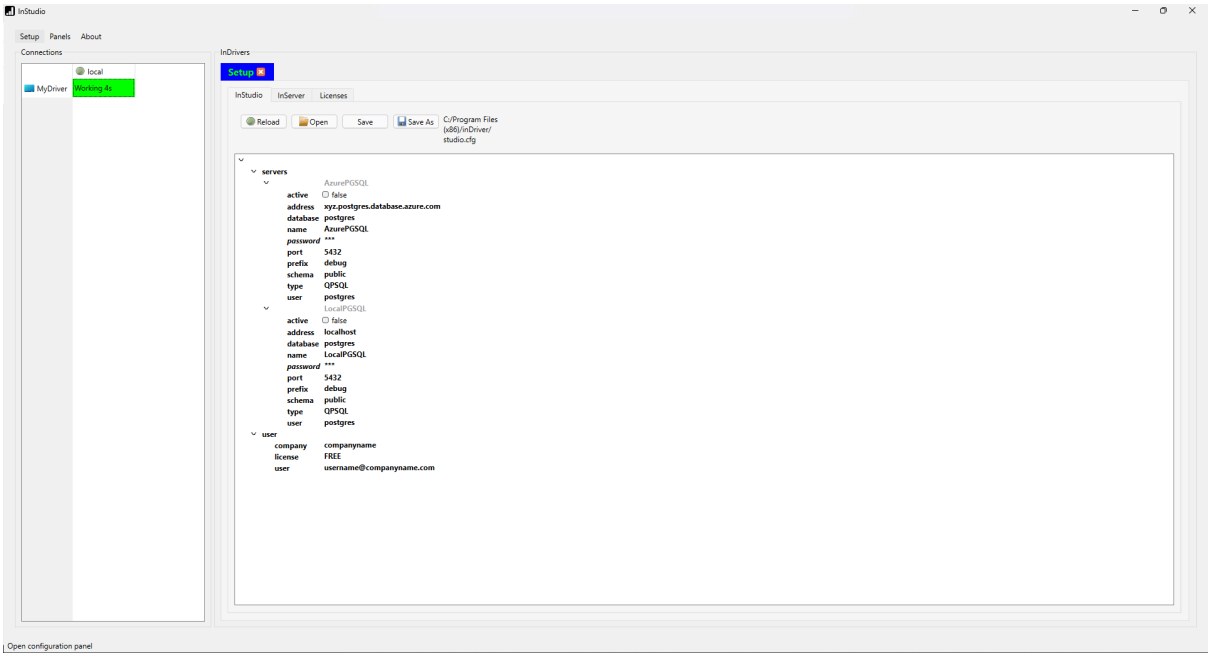
When SQL Server(s) are configured correctly, the next step is to install InServerAPI. InServerAPI is a set of SQL functions and tables used by InDrivers and InStudio to communicate with each other, and store logs, configurations, messages, etc.
To install InServerAPI, open the 'InServer' tab, select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



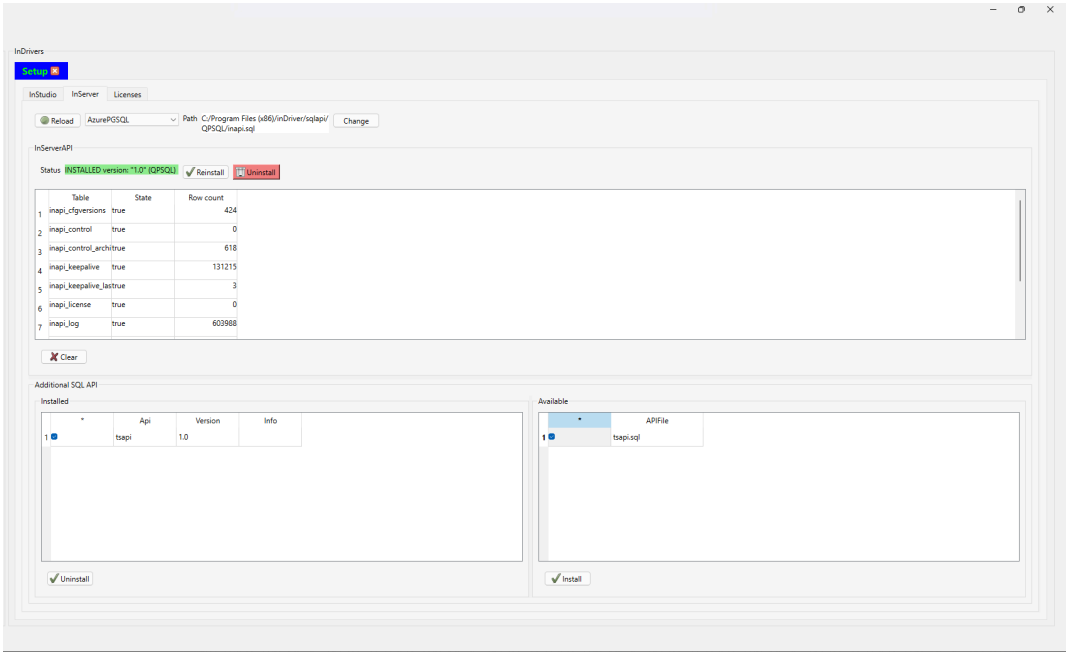
Innovative
Data
Analytics



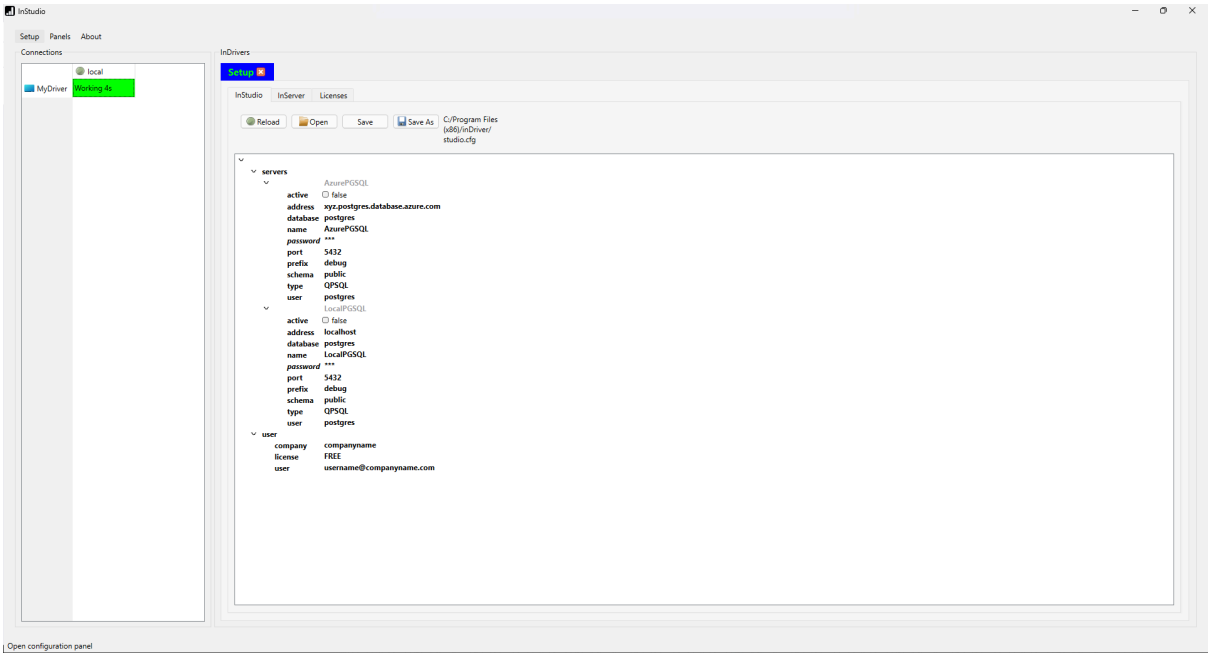
Install the Additional API(s)

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.



Innovative
Data
Analytics

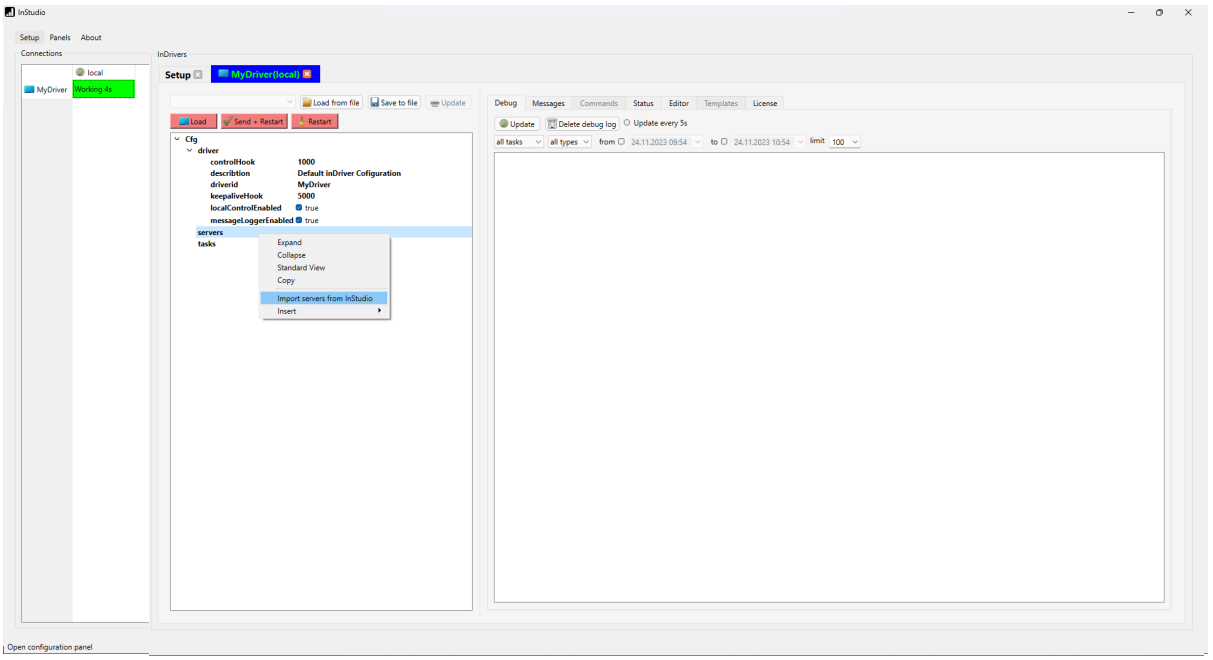


Innovative
Data
Analytics

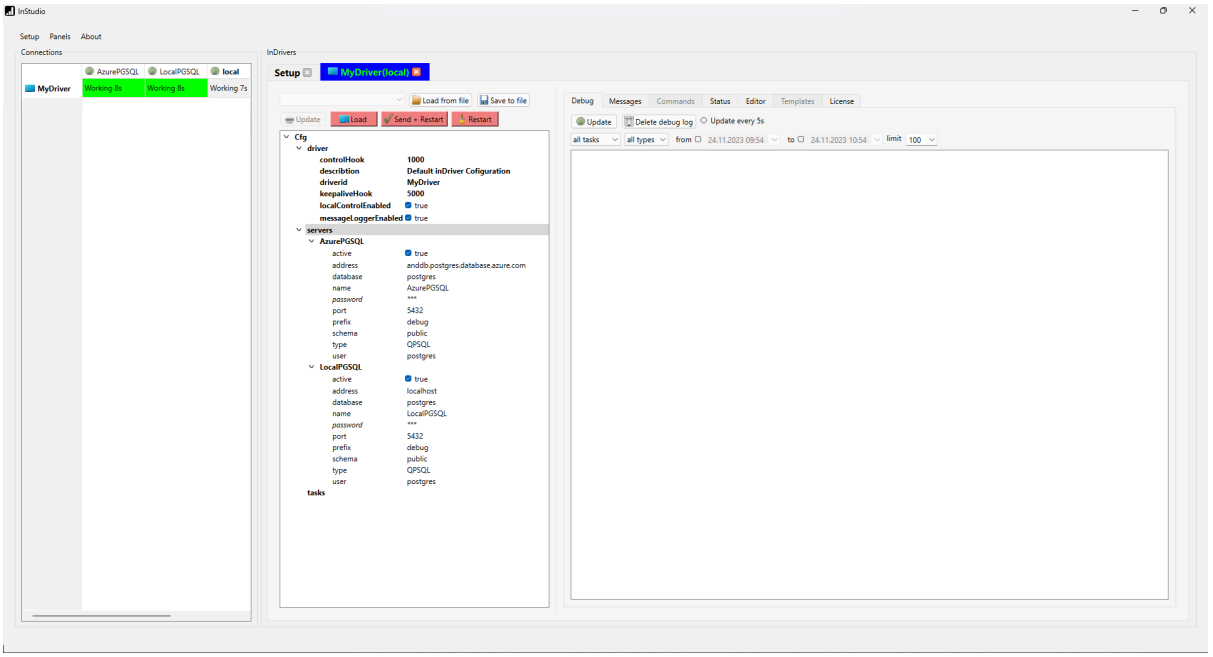
InDriver first steps

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.



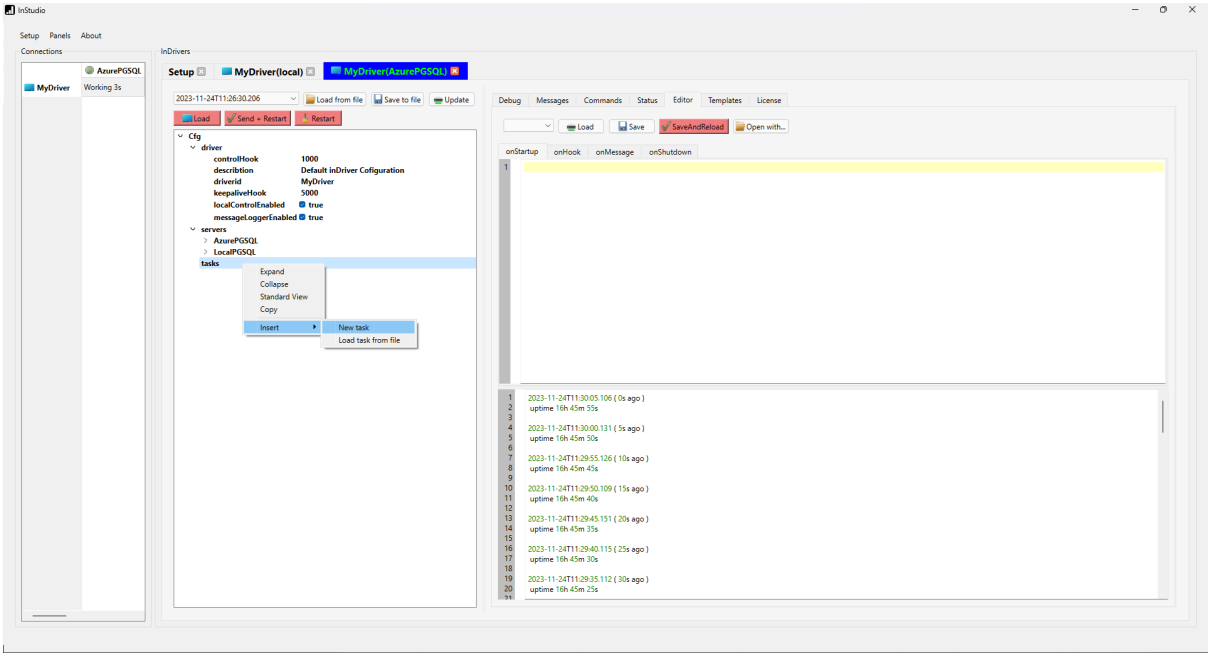
Innovative
Data
Analytics



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

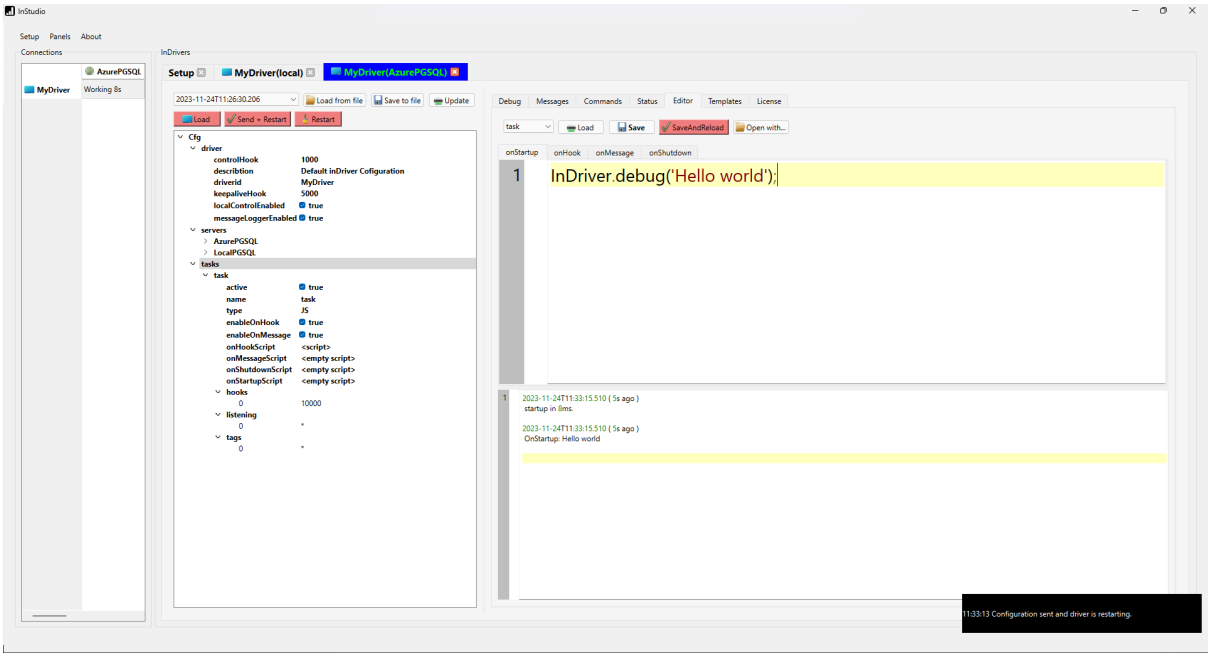
To add a new task, click 'Insert' in the 'Tasks' context menu.



Innovative Data Analytics

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!



Innovative
Data
Analytics

Features and examples

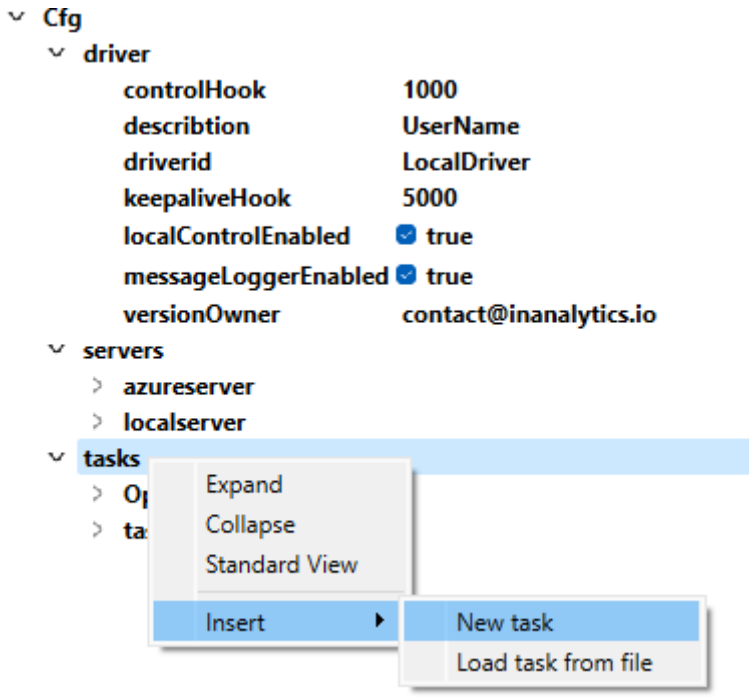
To be done soon :)

Innovative
Data
Analytics

InDriver JS Tasks

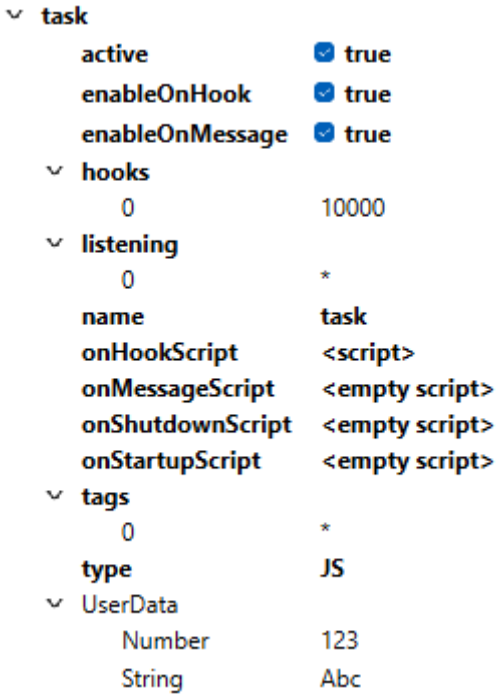
InDriver can simultaneously execute multiple tasks. To insert a new task, click 'Insert' on the tasks content menu and select either 'New task' or 'Load task from file.' The second option enables loading saved task configurations, including example tasks with names starting from '@...' provided with 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:

Innovative
Data
Analytics



"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**’

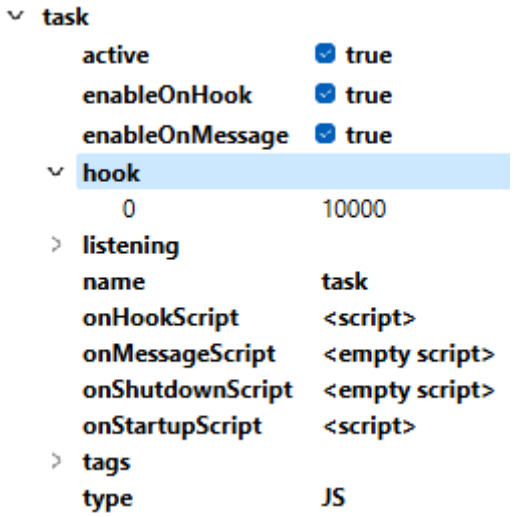
Additionally, the user can add other JSON Objects, Variables, or Arrays that will store user data. In this example '**userData**' object is added with Number and String values.

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.

InDriver JS API

InDriver's JavaScript tasks enhance programming efficiency through additional API Objects. The default InDriver object is equipped with an API specifically designed to interact with other tasks, access configuration, control hooks, and messages, execute SQL functions on configured servers, and create additional API Objects.

Detailed description:

- **Null InDriver**.debug(String *debugMsg*);
- **Null InDriver**.debug(String *debugMsg*, String *msgType*);

  Outputs a debug message to the Debug console. The ***msgType*** parameter can take on values such as 'debug' (default), 'critical', 'fatal', 'info', or 'warning'.

  Example:

  ⇒ **onHook**

```
InDriver.debug('This is my debug line');
InDriver.debug('This is my critical debug line','critical');
```

- **String InDriver**.driverName( );

  Returns the name of the **InDriver**.

  Example:

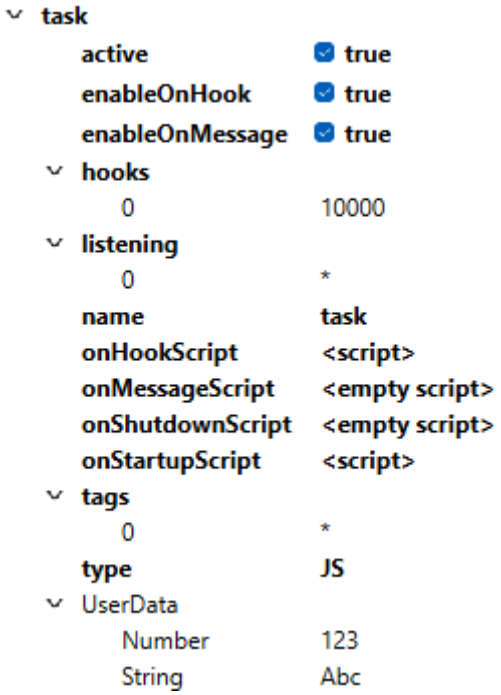
  ⇒ **onHook**

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

- **String InDriver**.configuration(String Array *path*);

  Returns a JSON string that includes the task's configuration located at the specified ***path***.

  Example:



⇒ **onStartup**

```
InDriver.configuration([]);
// RETURNS:  { "UserData": { "Number": 123, "String": "Abc" }, "active": true, "enableonHook": true, "enableonMessage": true, "hooks": [ 10000 ], "listening": [ "*" ], "name": "task", "onHookScript": "let ts= InDriver.hookTs();\nInDriver.debug('hooks => ' +InDriver.hooks()+' => ISO '+ ts.toISOString());", "onMessageScript": "", "onShutdownScript": "", "onStartupScript": "\nInDriver.setFlag('busy','calculations in progress');\nInDriver.debug(InDriver.configuration([]));\nInDriver.debug(InDriver.configuration([\"UserData\"]));", "tags": [ "*" ], "type": "JS" }

InDriver.debug(InDriver.configuration(["UserData"]));
//RETURNS
{ "Number": 123, "String": "Abc" }
```

- **String InDriver**.currentPath( );

  Returns the directory that contains the InDriver.exe.

  ⇒ **onStartup**

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

Innovative
Data
Analytics

- **String InDriver**.hookTs(**Number** *hook*);
- **String InDriver**.hookTs( );

Returns DateTime of currently executed hook

Read more about Hooks in the ‘onHook’ function description.

Example:

⇒ **onHook**

```
const  ts= InDriver.hookTs(10000);

// or

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

.

- **Boolean InDriver**.import(**String** *api*);

Creates a JavaScript object with InDriver API.

List of available InDriver APIs:

- 'ModbusApi'
- 'RestApi'
- 'TsApi'"

The function returns true when the API object is successfully created; otherwise, it returns false when the ***`apiName`*** is incorrect.

Example:

⇒ **onStartup**

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

When `InDriver.import('Modbus');` is called, the 'Modbus' object is created, offering a set of additional functions.

- **Null InDriver**.installHook(**Number** *hook*);

Initiates the execution of the `onHookScript` for each ***hook*** defined in milliseconds.

Typically called in the 'onStartupScript,' but can be used anywhere. Each JS task can install multiple hooks that will be simultaneously executed, for example, every 10s, 60s, and 3600s. Hooks are synchronized with the system clock.

Read more about Hooks in the 'onHook' function description.

Example:

⇒ **onStartup**

```
InDriver.installHook(10000); // 10 seconds
InDriver.installHook(60000); //1-minute
InDriver.installHook(3600000); //1-hour
```

.

- **Boolean InDriver**.isHook(**Number** *hook*);

Returns true if the current execution of the onHook function is due to the ***hook*** interval; otherwise, returns false.

Read more about Hooks in the 'onHook' function description.

Example:

⇒ **onHook**

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

- **String InDriver**.messageData( );

Returns the data attached by another task when invoking `InDriver.sendMessage` with tags listened to by this task.

Example:

⇒ **onMessage**

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

- **String InDriver**.messageSender( );

Returns the name of another task that invoked `InDriver.sendMessage` with tags

listened to by this task.

Example:

⇒ **onMessage**

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

- **String InDriver**.messageTs( );

  Returns the timestamp (DateTime) on which another task invoked `InDriver.sendMessage` with tags listened to by this task

  Example:

  ⇒ **onMessage**

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

- **String InDriver**.messageTags( );

  Returns the tags attached by another task when invoking `InDriver.sendMessage` with tags listened to by this task.

  Tags are used to identify different messages; for example, all messages with data that needs to be logged in the database can be marked with the 'archive' tag.

  Example:

  ⇒ **onMessage**

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

- **Null InDriver**.sendMessage(String *dt*, String *tags*, String *data*);

  Sends a message to all listening tasks. Each task can listen to all other tasks ('*'), specified tasks, or specified tags.

  Arguments: ***dt*** for timestamp, ***tags*** as an array describing the message, and ***data*** as a string containing data sent with the message.

  Example:

  ⇒ **onHook**

```
var cd= new Date();

var data = InDriver.sqlExecute('LocalDatabase', 'select * from public.table;');

InDriver.sendMessage(cd.getDate(), {'archive'}, data);
```

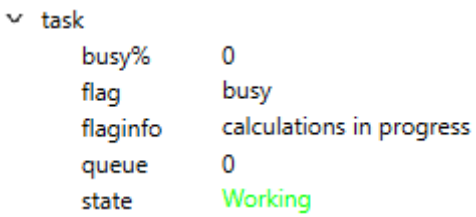
- **Null InDriver**.setFlag(String *flag*, String *info*);

  Sets a ***flag*** with associated flag information (***info***). Flags play a role as additional information is displayed in the 'Status' tab.

  Example:

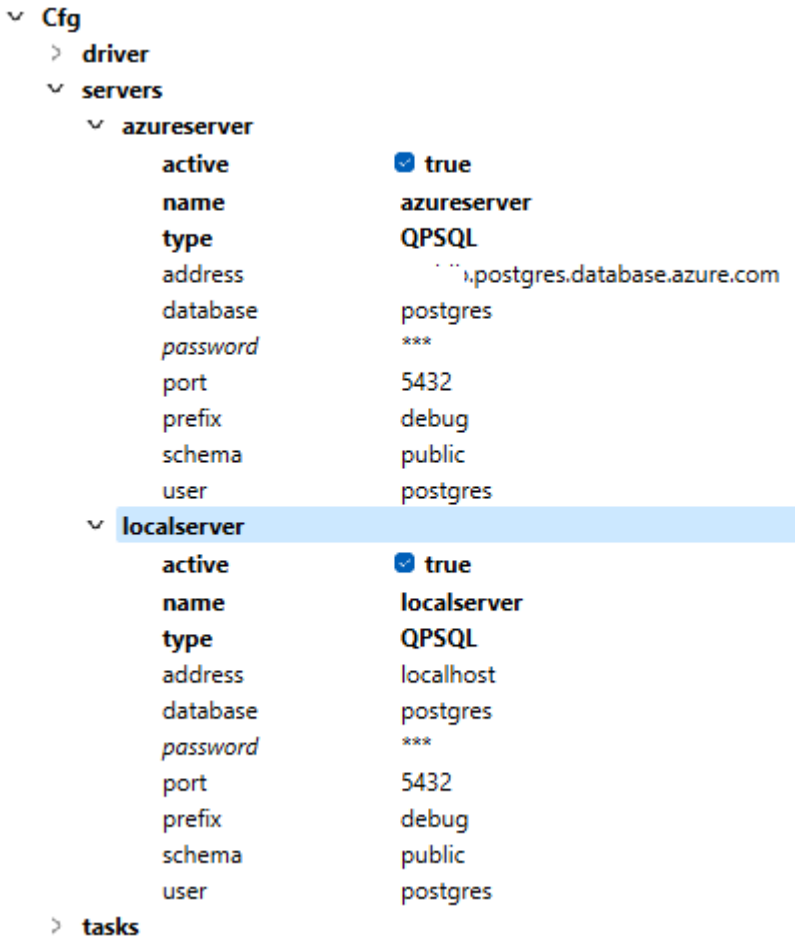
  ⇒ **onHook**

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



- **String InDriver**.sqlExecute(String *server*, String *query*);

  Executes an SQL ***query*** on the database related to the specified ***server***. Returns a JSON array with the retrieved rows. Each row is represented as a JSON object with variable keys corresponding to columns and values corresponding to their values. Servers are defined in the InDriver configuration item ***server***.



Example:

⇒ **onHook**

```
var data = InDriver.sqlExecute('localserver', 'select * from public.table');
```

- **Boolean InDriver**.sqlExecuteAll(String *query*);

Executes an SQL ***query*** on all database servers defined in InDriver. Returns true when the execution succeeds on all servers or false if even one execution fails.Remark: If false is returned, some queries may have been executed successfully.

Example:

⇒ **onHook**

```
var b = InDriver.sqlExecuteAll( 'insert into public.table ( ts timestamp with time
```

```
zone, task text ) values (now(), ‘’‘+InDriver.taskName()+’’’);’);
```

- **Boolean InDriver**.sqlIsSucceeded( );

Returns true if last sqlExecute or sqlExecuteAll have been executed successfully otherwise, it returns false.

Example:

⇒ **onHook**

```
var data = InDriver.sqlExecute(‘localserver’, ‘select * from public.table’);
if (!InDriver.sqlIsSucceeded()) {
        InDriver.debug(InDriver.sqlLastError());
}
```

- **String InDriver**.sqlLastError( );

Returns an error string if the last sqlExecute or sqlExecuteAll was not executed successfully, otherwise, it returns an empty string.

Example:

⇒ **onHook**

```
var data = InDriver.sqlExecute(‘localserver’, ‘select * from public.table’);
if (!InDriver.sqlIsSucceeded()) {
        InDriver.debug(InDriver.sqlLastError());
}
```

- **String InDriver**.taskName( );

Returns the name of the task.

Example:

⇒ **onHook**

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

Innovative
Data
Analytics

Innovative
Data
Analytics

RestApi

The RestApi object provides functions that simplify the programming of RESTful API calls. The following illustrative examples showcase three solutions enabled by the JS API provided with InDriver: simple single request calling, simultaneous calling of multiple requests, and collecting data from RestAPI and mixed APIs, such as ModbusApi and RestApi.

Example 1: **Calling a single request** and waiting for a response or timeout.

Retrieve the weather forecast for Kraków, Poland, and log the data into the Azure SQL Database Server every 1 minute. Replace 'your_app_id' with your OpenWeatherMap.org application ID.

⇒ **onStartup**

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

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

InDriver.installHook(60000);
```

⇒ **onHook**

```
 RestApi.sendRequest('Krakow);
 if (RestApi.isSucceeded()) {
    let ts= InDriver.hookTs();
    const data = RestApi.getData('Krakow');
    InDriver.debug(data);
    InDriver.sqlExecute("azureserver", "insert into public.weather (source, ts, data )
values ( 'Krakow','"+ts.toISOString()+"',$$"+data+"$$);");
 }
```

Example 2: **Simultaneously calling multiple requests** and waiting for responses or timeouts.

Retrieve the weather forecast for Kraków, Poland, and Chicago, USA, and log the data into the Azure SQL Database Server every 1 minute. This method ensures that both requests are executed simultaneously.

⇒ **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"}}');

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

InDriver.installHook(60000);
```

⇒ **onHook**

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

 if (RestApi.isSucceeded()) {
    let ts= InDriver.hookTs();
    const dataFromKrakow = RestApi.getData('Krakow');
    const dataFromChicago = RestApi.getData('Chicago');
    InDriver.debug('Krakow: ' + dataFromKrakow + '\nChicago: ' +
dataFromChicago);
    InDriver.sqlExecute("azureserver", "insert into public.weather (source, ts, data )
values ( 'Krakow','"+ts.toISOString()+"',$$"+dataFromKrakow+"$$);"),
( 'Chicago','"+ts.toISOString()+"',$$"+dataFromChicago+"$$);");
 }
```

Example 3: **Simultaneously calling multiple mixed API requests** and waiting for responses or timeouts.

Retrieve the weather forecast for Kraków, Poland, and IO states from a Modbus device (Moxa IOLogic) simultaneously. This method ensures that both requests are executed simultaneously.

⇒ **onStartup**

```
InDriver.import("RestApi");
InDriver.import("ModbusApi");

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"}}');

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

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

⇒ onHook

```
Modbus.begin();
RestApi.begin();

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

Modbus.commit();
RestApi.commit();

Modbus.wait();
RestApi.wait();

const weatherData = RestApi.getData('Krakow');
const modbusData = Modbus.getAllData();

let ts = InDriver.hookTimestamp();

InDriver.sqlExecute("azureserver", "insert into public.weather (source, timestamp,
data ) values ( 'Krakow','"+ts.toISOString()+"',$$"+weatherData +"$$);");

InDriver.sqlExecute("azureserver", "insert into public.iologic (source, timestamp,
data ) values ( 'IoLogic','"+ts.toISOString()+"',$$"+modbusData+"$$);");
```

Detailed description:

- **Null RestApi**.begin( );

  Similar to SQL, this function initiates a block of code where multiple `RestApi.sendRequest` calls can be called. Following `RestApi.begin()`, all `RestApi.sendRequest()` calls are not executed immediately but will be processed after the execution of the `RestApi.commit()` or `RestApi.commitWait()` commands. Learn more about making simultaneous calls to multiple REST API requests or even mixed API requests (e.g., RestApi and ModbusApi)

- **Null RestApi**.commit( );

  This function concludes the block of code initiated by `RestApi.begin()` and executes all `RestApi.sendRequest` functions called within the block. The function does not wait for responses or timeouts. For a version that does wait until all answers are completed or timed out, refer to `RestApi.commitWait()`.

- **Null RestApi**.commitWait( );

  This function concludes the block of code initiated by `RestApi.begin()`, executes all `RestApi.sendRequest` calls, and waits for responses or timeouts. For a version that does not wait until all answers are completed or timed out, refer to `RestApi.commit()`.

- **Null RestApi**.defineRequest(String request, String def);

  Defines a RESTful API data request with the name '**request**', which can be invoked using the `sendRequest` function. Multiple requests can be defined, and they can be easily called by providing only their names.
  The request parameters are stored in 'def' as JSON and consist of the following keys: '**url**', '**timeout**', '**type**', and '**headers**'. Below is an example of a REST API definition:

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

Headers object can contain standard headers or any custom (raw) headers.

List of Standard Headers:

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

- **Boolean RestApi**.getData(String *request*);

  "The function returns the data obtained from the previous call to `RestApi.sendRequest()` for the specified request name ***request***, as defined in `RestApi.defineRequest()`.

- **Boolean RestApi**.isSucceeded();

  Returns true if the preceding call to `RestApi.sendRequest()` or block's functions: `RestApi.commit()` or `RestApi.commitWait()` was successful, indicating that a valid response was received before the specified timeout.

- **Boolean RestApi**.sendRequest(String *request*);
- **Boolean RestApi**.sendRequest(String *request*, String *def*);

  Sends a RESTful API request with the name ***request***, as defined using the `RestApi.defineRequest` function. The function waits until the response is received or until the specified ***timeout*** is reached if not called in a block starting from RestApi.begin(). When `RestApi.defineRequest` is called within the block starting from RestApi.begin(), the request is postponed until RestApi.commit() or RestApi.commitWait() is called. Learn more about making simultaneous calls to multiple REST API requests or even mixed API requests (e.g., RestApi and ModbusApi).
  `RestApi.sendRequest(String request, String def)` replaces the request definition key's values, provided in the previous `RestApi.defineRequest` call. When the request was not previously defined, the new request is created with the name ***request*** and parameters ***def*** but is not added to the defined request.

The function returns **false** if the request name provided in the ***name*** argument has not been defined previously; otherwise, it returns **true**.

- **Boolean RestApi**.wait( );

  The function waits for the completion of the previously called `RestApi.commit()`. It continues to wait until the response is received or until the specified timeouts, as defined in the request, are reached.

- **Boolean RestApi**.wait(Number timeout);

  The function waits for the completion of the previously called `RestApi.commit()`. It continues to wait until the response is received or until the specified ***timeout*** is reached.

Innovative
Data
Analytics

ModbusApi

The ModbusApi object offers functions that streamline the programming of Modbus device read and write operations. ModbusApi supports both instant calls to the device and grouping calls using SQL-like `begin()` and `commit()` or `commitWait()` blocks. Please refer to the RestApi description for detailed information on instant and grouping calls.

⇒ onStartup

```
InDriver.import("RestApi");
InDriver.import("ModbusApi");

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

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

InDriver.installHook(60000);
```

⇒ onHook

```
Modbus.begin();
RestApi.begin();

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

Modbus.commit();
RestApi.commit();

Modbus.wait();
RestApi.wait();

const weatherData = RestApi.getData('Krakow');
const modbusData = Modbus.getAllData();

let ts = InDriver.hookTimestamp();

InDriver.sqlExecute("azureserver", "insert into public.weather (source, ts, data )
values ( 'Krakow','"+ts.toISOString()+"',$$"+weatherData +"$$);");

InDriver.sqlExecute("azureserver", "insert into public.iologic (source, ts, data )
values ( 'IoLogic','"+ts.toISOString()+"',$$"+modbusData+"$$);");
```

Innovative
Data
Analytics

Detailed description:

- **Null ModbusApi**.begin( );

  Similar to SQL, this function initiates a block of code where multiple `ModbusApi.readDevice()` or `ModbusApi.writeDevice()` calls can be called. Following `ModbusApi.begin()`, `ModbusApi.readDevice()`, and `ModbusApi.writeDevice()` calls are not executed immediately but will be processed after the execution of the `ModbusApi.commit()` or `ModbusApi.commitWait()` commands. Learn more about making simultaneous reads or writes to multiple Modbus devices or even mixed API requests (e.g., RestApi and ModbusApi).

- **Null ModbusApiestApi**.commit( );

  This function concludes the block of code initiated by `ModbusApi.begin()` and executes all `ModbusApi.readDevice()` or `ModbusApi.writeDevice()` functions called within the block. The function does not wait for responses or timeouts. For a version that does wait until all answers are completed or timed out, refer to `ModbusApi.commitWait()`.

- **Null ModbusApi**.commitWait( );

  This function concludes the block of code initiated by `ModbusApi.begin()`, executes all `ModbusApi.readDevice()` or `ModbusApi.writeDevice()` functions, and waits for responses or timeouts. For a version that does not wait until all answers are completed or timed out, refer to `ModbusApi.commit()`.

- **Null ModbusApi**.connectDevice(String *device*, String *def*);

  Creates a Modbus device connection named ***device*** with connection parameters included in **def**. Once the connection is created, ModbusApi.readDevice() or ModbusApi.writeDevice() functions can be invoked with the connected device's name. Multiple connections can be defined. Users do not have to keep checking if the connection is still alive because InDriver performs non-stop checks, and when a disconnection occurs, it is automatically reconnected. The connection parameters are stored in ***def*** as JSON and consist of the following keys:

  - for **TCP** mode: '**mode**' = 'TCP', '**networkAddress**', '**networkPort**' default 502, '**timeoutMs**' default 3000, '**numberOfRetries**' default 3.

  - for **RTU** mode: '**mode**' = 'RTU', '**serialPortName'**, '**parity**' default even (2), '**baudRate**' default 9600, '**dataBits**' default 8,'**stopBits**' default 1,

'**timeoutMs**' default 3000, '**numberOfRetries**' default 3.

Default parameters are not obligatory.

Below are examples of connection definitions (***def***):

1. minimal TCP definition

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

2. full TCP definition

```
{
  "mode": "TCP",
  "networkAddress": "192.168.0.22",
  "networkPort": 502,
  "timeoutMs": 3000,
  "numberOfRetries": 3
}
```

3. minimal RTU definition

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

4. full TCP definition

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

Below are examples of the use of ModbusApi.connecDevice()

```
ModbusApi.connecDevice( ‘MoxaIOLogicTCP’,’{ "mode": "TCP", "networkAddress": "192.168.0.22"}’);

ModbusApi.connecDevice( ‘MoxaIOLogicRTU’,’{ "mode": "RTU", "serialPortName": "COM1"}’);
```

- **Null ModbusApi**.getAllData( );

  Returns data collected for all defined devices in the previous ModbusApi.readDevice() or multiple ModbusApi.readDevice() function calls when invoked within a block between ModbusApi.begin() and ModbusApi.commit() or ModbusApi.commitWait().

  For example, from a block with two ModbusApi.readDevice() functions that read devices Moxa1 and Moxa2, as shown below:

```
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();
```

  `ModbusApi.getAllData()` returns the collected data when the readout is successful:

```
{
 "Moxa1": {
        "Read": {
        "coilsA": {
        "1": true,
        "2": false,
        "3": true,
        "4": false
        },
        "coilsB": {
        "5": true,
        "6": false,
        "7": true,
        "8": false
        }
        }
 },
 "Moxa2": {
        "Read": {
        "coils": {
```

```
        "1": true,
        "2": false,
        "3": true,
        "4": false,
        "5": true,
        "6": false,
        "7": true,
        "8": false
        }
        }
  }
}
```

- **Null ModbusApi**.getDeviceData(String *device*);

  Similar to the `ModbusApi.getAllData()` function, this function returns a portion of the collected data related to the specified device named **device** as defined in the `ModbusApi.readDevice()` function. In the above example presented for the `ModbusApi.getAllData()` function, `ModbusApi.getDeviceData("Moxa1")` returns:

```
{
  "Moxa1": {
        "Read": {
        "coilsA": {
        "1": true,
        "2": false,
        "3": true,
        "4": false
        },
        "coilsB": {
        "5": true,
        "6": false,
        "7": true,
        "8": false
        }
        }
  }
}
```

- **Null ModbusApi**.getDeviceRequestData(String *device*, String *reg*);

Similar to the `ModbusApi.getDeviceData()` function, this function returns a portion of the collected data related to the specified device named ***device*** and the register named ***reg*** as defined in the `ModbusApi.readDevice()` function. In the above example presented for the `ModbusApi.getDeviceAllData()` function, `ModbusApi.getDeviceRequestData("Moxa1", "coilsA")` returns:

```
{
  "Moxa1": {
          "Read": {
          "coilsA": {
          "1": true,
          "2": false,
          "3": true,
          "4": false
          }
          }
  }
}
```

- **Null ModbusApi**.getDeviceRequestValue(String *device*, String *reg,* Number Array *addresses*);

  Similar to the `ModbusApi.getDeviceRequestData()` function, this function returns the selected values whose addresses are listed in an array named ***addresses***, related to the specified device named ***device***, and the register named ***reg*** as defined in the `ModbusApi.readDevice()` function. In the above example presented for the `ModbusApi.getDeviceAllData()` function, `ModbusApi.getDeviceRequestValue("Moxa1", "coilsA", [1, 2, 3])` returns

```
5
```

  Binary 101

- **Null ModbusApi**.isSucceeded( );

  Returns true if the preceding call to `RestApi.readDevice()` or `RestApi.writeDevice()` or block's functions: `RestApi.commit()` or `RestApi.commitWait()` was successful, indicating that a valid response was received before the specified timeout.

- **Null ModbusApi**.readDevice(String *device*, String *def*);

  Sends a Modbus read device request to the device with the name ***device*** as defined using the `ModbusApi.connectDevice()` function. The function waits until the response is received or until the specified ***timeout*** is reached if not called in a block starting from `ModbusApi.begin()`.
  When `ModbusApi.readDevice` is called within the block starting from `ModbusApi.begin()`, the request is postponed until `ModbusApi.commit()` or `ModbusApi.commitWait()` is called.
  Learn more about making simultaneous calls to multiple REST API requests or even mixed API requests (e.g., `RestApi` and `ModbusApi`).

  The argument `def` is passed as JSON and includes the definition of the Modbus read request. It consists of the following key-value pairs:
  - 'name' - the name of the request,
  - 'type' - register type name, which can be one of the following types:
    - 'COILS',
    - 'DISCRETEINPUTS',
    - 'HOLDINGREGISTERS',
    - 'INPUTREGISTERS')
  - 'address' - the starting address of the request,
  - 'size' - the number of expected coils or registers to read. Refer to the device's Modbus Register Map and Modbus protocol and connected device's limitations.

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

  Below is an example of Modbus Read Device Request:

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

  Before using the `ModbusApi.readDevice` function, the device must be connected using the `ModbusApi.connectDevice` function.

  The function returns **false** if the device name provided in the ***name*** argument has not been defined previously; otherwise, it returns **true**.

- **Null ModbusApi**.wait( );

  The function waits for the completion of the previously called

`ModbusApi.commit()`. It continues to wait until the response is received or until all specified request timeouts, as defined in the request, are reached.

- **Null ModbusApi**.wait(Number ***timeout***);

  The function waits for the completion of the previously called `ModbusApi.commit()`. It continues to wait until the response is received or until the specified ***timeout*** is reached.

- **Null ModbusApi**.writeDevice(String *device*, String *def*);

  Similar to the `ModbusApi.readDevice()` function, the `ModbusApi.writeDevice()` function executes a Modbus device write request.

  The argument `def` is passed as JSON and includes the definition of the Modbus write request. It consists of the following key-value pairs:

  - '**name**': the name of the request,
  - '**type**': register type name, which can be one of the following types:
    - '**COILS**',
    - '**DISCRETEINPUTS**',
    - '**HOLDINGREGISTERS**',
    - '**INPUTREGISTERS**',
  - '**address**': the starting address of the request,
  - '**data**': an array with values corresponding to addresses starting from the specified starting address - ***address***.

  Refer to the device's Modbus Register Map, Modbus protocol, and the connected device's limitations.

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

  Below is an example of Modbus Write Device Request:

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

  Before using the `ModbusApi.writeDevice` function, the device must be connected using the `ModbusApi.connectDevice` function.

  The function returns **false** if the device name provided in the ***name*** argument has not

been defined previously; otherwise, it returns **true**.

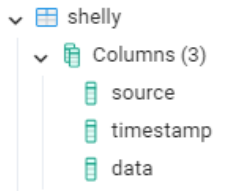
Innovative
Data
Analytics

TsApi

The TsApi provides functions that enhance the programming of Time Series Data. A time series is a sequence of data points indexed in chronological order, typically taken at equally spaced intervals. InDriver primarily utilizes JSON objects for data and configuration. The Time Series JSON data provided by InDriver's API offers an efficient and universal approach to data collection. This enables the processing and logging of data from various sources into database tables in a consistent manner.

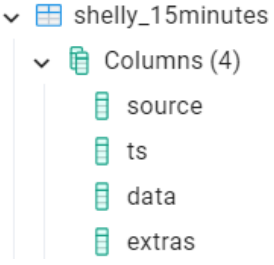
TsApi includes functions for data processing, such as aggregation or forecasting, and encapsulates database vendor-dependent SQL queries for time series data processing. This design allows dashboards to be independent of the underlying database vendor, utilizing TsApi functions instead of regular SQL queries. TsApi versions dedicated to different databases are installed with InDriver, providing the same functionality while accommodating differences to remain compatible with the specific database in use.

Example of JSON Time Series Table.



	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.

Example of JSON Time Series Aggregation Table.



	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": 0.1699999
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": 0.39000000000000057
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": -0.059999999999951
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": -0.660000
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": 0.8400000000000034},
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": -76.05000000000001}, "
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": {"

Innovative Data Analytics

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
        },
```

Innovative Data Analytics

Detailed description:

- **Null TsApi**.aggregate(String *aggregator*);

  This function triggers the **Aggregation Engine** to perform the next part of calculations and should be implemented in the onHook function.

  Example:

  ⇒ **onHook**

```
TsApi.aggregate("a");
```

- **Null TsApi**.defineAggregator(String *aggregator,* String *server,* String *table,* String *timeZone = 'UTC',* String Array *sources='[]',* String Array *intervals ='["1m", "15m", "1h", "1d"]',* Numeric *step=10000*);

  The function establishes an **Aggregation Engine** with the name ***aggregator*** to perform aggregation from the source table named ***table*** on the server named ***server***. The aggregator creates aggregation tables for each specified aggregation interval.

  The default intervals are set to **1 minute**, **15 minutes**, **1 hour**, and **1 day**, with the default time zone (***timeZone***)set to '**UTC**.' The time zone is relevant for 1-day aggregations to determine the UTC time when the day starts in the specified time zone. The default sources are an empty array (***sources***: **[]**), indicating that data from all sources will be aggregated. If the source array includes declared sources, the aggregation is performed only on these specified sources.

  The default ***step*** size is set to **10,000**, meaning that every aggregation cycle processes 10,000 rows of the source table. An aggregation cycle is triggered by the TsApi.aggregate() function.

  The names of the aggregation tables for these intervals are composed of the source table name plus the interval name, for example, 'table_1minute,' 'table_15minutes,' 'table_1hour,' and 'table_1day.' The aggregation tables have the same columns as the source table, plus one additional column named 'extras,' which is used to store aggregation statistic data.

  Example:

  ⇒ **onStartup**

Innovative
Data
Analytics

```
InDriver.import('TsApi');
TsApi.defineAggregator("a", "azureserver", "public.weather");
```

ProcessApi

The ProcessApi provides functions used to start and close external programs.

General Example:

Example:

⇒ **onStartup**

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

⇒ **onHook**

```
InDriver.debug(“go  = ”+ProcessApi.start("backup",'"C:\\backup.bat”',""));
ProcessApi.waitForStarted("backup");

InDriver.debug("pid = "+ProcessApi.pid("backup\n dir =
"+ProcessApi.workingDirectory("backup\n prg =
"+ProcessApi.program("backup")+”\n stat = "+ProcessApi.state("backup"));

ProcessApi.waitForFinished("backup");
```

⇒ **onShutdown**

```
ProcessApi.killAll()
```

Detailed description:

- **Null ProcessApi**.close(String *name);*

  Closes all communication with the process associated with the ***name*** and kills it.

- **Null ProcessApi**.closeAll(*);*

  Closes all communication with all processes started with ProcessApi.start function.

- **Null ProcessApi**.kill(String *name);*

  Kills the process associated with the ***name***, causing it to exit immediately.

- **Null ProcessApi**.killAll(*);*

  Kills all processes started with ProcessApi.start function.

- **Null ProcessApi**.pid(String *name);*

  Returns the native process identifier for the process associated with the ***name***, if available. If no process is currently running, 0 is returned.

- **Null ProcessApi**.program(String *name);*

  Returns the program of the process associated with the ***name***.

- **Null ProcessApi**.setWorkingDirectory(String *name,* String *directory);*

  Sets the working ***directory*** of the process associated with the ***name***.

- **Null ProcessApi**.start(String *name,* String *program,* String Array *args*);

  Starts the given ***program*** in a new process, passing the command line arguments in **args.** Associate the process handle with the ***name.***

- **Null ProcessApi**.state(String *name);*

  Returns the current state of the process associated with the ***name***.

- **Null ProcessApi**.waitAllForFinished(Number *msecs = 30000);*

  Blocks until all of the processes have **finished** or until ***msecs*** milliseconds have passed.

  Returns true if the processes finished; otherwise returns false (if the operation timed out or if an error occurred).

- **Null ProcessApi**.waitAllForStarted(Number *msecs = 30000);*

  Blocks until all of the processes have **started** or until ***msecs*** milliseconds have passed.

  Returns true if the processes finished; otherwise returns false (if the operation timed

out or if an error occurred).

- **Null ProcessApi**.waitForFinished(String *name,* Number *msecs);*

  Blocks until the process associated with the ***name*** has **finished** or until ***msecs*** milliseconds have passed.

  Returns true if the process is finished; otherwise returns false (if the operation timed out or if an error occurred).

- **Null ProcessApi**.waitForStarted(String *name,* Number *msecs);*

  Blocks until the process associated with the ***name*** has **started** or until ***msecs*** milliseconds have passed.

  Returns true if the process is started; otherwise returns false (if the operation timed out or if an error occurred).

- **Null ProcessApi**.workingDirectory(String *name);*

  If the process associated with the ***name*** has been assigned a working directory, this function returns the working directory that the process will enter before the program has started.

Innovative
Data
Analytics

FileApi

The FileApi provides an interface for reading from and writing to files and an interface for monitoring files and directories for modifications.

Detailed description:

- **Null FileApi**.addFileSystemWatcherPath(String *path);*

  Adds a file or directory ***path*** to the file system watcher.

  When the system detects any change in the specified ***path***, the ***onMessage*** function is invoked with a relevant tag: "**FileChanged**" for file changes and "**DirectoryChanged**" for directory changes and message data as a JSON object with a key "path" and the value of the path where the change is detected.

  Example:

  ⇒ **onStartup**

```
InDriver.import("FileApi");
FileApi.addFileSystemWatcherPath(InDriver.currentPath()+"/driver.cfg");
```

  ⇒ **onMessage**

```
if (InDriver.messageSender() === InDriver.taskName()) {
    InDriver.debug(InDriver.messageTags()+":"+InDriver.messageData());
}

/*
When driver.cfg is changed onMessage is called

onMessage: [FileChanged]:
{
  "message": {
          "path": "C:/MyDevelopment/debug/driver.cfg",
          "type": "JSscript"
  }
}
*/
```

Innovative
Data
Analytics

- **Null FileApi**.close(String *name);*

  Closes file related to ***name*** opened with FileApi.open function.

- **Null FileApi**.closeAll(*);*

  Closes all files opened with FileApi.open function.

- **Null FileApi**.open(String *name* ,String *file,* String Array *modes);*

  Opens the file pointed to by the 'file' parameter and assigns a handle with the specified 'name'. If the file is already open, the function returns **'AlreadyOpen'**. The 'modes' parameter is an array of strings, including one or more of the following flags: **ReadOnly**, **WriteOnly**, **ReadWrite**, **Append**, **Truncate**, **Text**, **Unbuffered**, **NewOnly**, **ExistingOnly**.If the mode is ReadOnly or ReadWrite and the file does not exist, the function returns **'NotOpened'**; otherwise, it returns **'Opened'**.

  Example:

  ⇒ **onHook**

```
InDriver.import('FileApi');
FileApi.open("cfgFile", InDriver.currentPath()+"/driver.cfg", "ReadOnly");
let data = FileApi.readAll("cfgFile");
InDriver.debug(data);
FileApi.close("cfgFile");
```

- **Null FileApi**.readAll(String *name);*

  Reads all remaining data from the file associated with handle ***name***;

- **Null FileApi**.removeFileSystemWatcherPath(String *file);*

  Removes the specified path from the file system watcher.

  Example:

  ⇒ **onShutdown**

```
FileApi.removeFileSystemWatcherPath( InDriver.currentPath()+"/driver.cfg");
```

- **Null FileApi**.write(String *name,* String *data);*

  Writes ***data*** from a zero-terminated string of 8-bit characters to the file. Returns the number of written bytes, or -1 if an error occurred.

Example: copy ‘driver.cfg’ to ‘driver.cfg_copy’

⇒ **onHook**

```
InDriver.debug(FileApi.open("cfg",InDriver.currentPath()+"/driver.cfg",["ReadOnly"]))
;
InDriver.debug(FileApi.open("cfg_copy",InDriver.currentPath()+"/driver.cfg_copy",["
WriteOnly"]));
let data = FileApi.readAll("cfg");
InDriver.debug(data);
FileApi.write("cfg_copy",data);
FileApi.closeAll();
```

Innovative
Data
Analytics

SerialPortApi

The SerialPortApi provides functions for accessing serial ports, writing data, handling write requests with a specified waiting timeout, and reading data asynchronously.

Detailed description:

- **Null SerialPortApi**.acceptRead(String *name*);

  Accepts data coming to the serial port associated with the specified `name`, and it stops waiting for data initiated by the `SerialPortApi.writeAndWait` function.

- **Null SerialPortApi**.availablePorts( );

  Generates a JSON array containing information about the available serial ports on the system.

  An example of port info for a system with two COM ports.

```
[
  {
    "description": "Communication Port",
    "location": "\\\\.\\COM1",
    "manufacturer": "(Standard Port Types)",
    "name": "COM1",
    "serialNumber": "",
    "vendorIdentifier": ""
  },
  {
    "description": "Communication Port",
    "location": "\\\\.\\COM2",
    "manufacturer": "(Standard Port Types)",
    "name": "COM2",
    "serialNumber": "",
    "vendorIdentifier": ""
  }
```

- **Null SerialPortApi**.close(String *name*);

  Closes serial port related to ***name*** opened with SerialPortApi.open function.

- **Null SerialPortApi**.closeAll(*);*

  Closes all ports opened with SerialPortApi.open function.

- **Null SerialPortApi**.error(String *name);*

  Returns the last error associated with the serial port operation identified by the specified `name`.

- **Null SerialPortApi**.open(String *name,* String *portSettings);*

  Opens the serial port, assigns a handle with the specified 'name,' and configures the serial port settings based on the provided `portSettings` JSON object. If the JSON object is empty or if some parameters are missing, default values will be assigned. The JSON object should contain the following key-value pairs:

- `"mode"` with available values:
  - `"Ignore"`: Ignore incoming bytes.
  - `"Flow"` (default): Call the `onMessage` function after a specified number of bytes arrive at the serial port.
  - `"Buffer"`: Buffer bytes with a size of `"bufferSize"` and call the `onMessage` function after the next part of bytes arrives, returning the entire buffer.
- `"timeout"`: Timeout for the write request, default is 3000ms.
- `"bufferSize"`: Size of the buffer, default size is 256 bytes.
- `"baudRate"`: Baud rate, default is 9600.
- `"parity"` with available values:
  - `"NoParity"` (default)
  - `"EvenParity"`
  - `"OddParity"`
  - `"SpaceParity"`
  - `"MarkParity"`
- `"stopBits"` with available values:
  - `"OneStop"` (default)
  - `"OneAndHalfStop"`
  - `"TwoStop"`
- `"dataBits"` with available values in the range between 5 and (default) 8.

Example JSON Object with Default Serial Port Settings:

```
{
  "mode": "Flow",
  "timeout": 3000,
  "bufferSize": 256,
  "baudRate": 9600,
```

```
  "parity": "NoParity",
  "stopBits": "OneStop",
  "dataBits": 8
}
```

Example:

⇒ **onStartup**

```
InDriver.import('SerialPortApi');
SerialPortApi.open("COM1", '{ "mode": "Flow",  "timeout": 3000,  "bufferSize": 256,
"baudRate": 9600,  "parity": "NoParity",  "stopBits": "OneStop",  "dataBits": 8}');
```

- **Number SerialPortApi**.write(String *name,* Number Array *data);*

Writes data to the device. Returns the number of bytes that were written, or -1 if an error occurred.

Example:

⇒ **onHook**

```
InDriver.import('SerialPortApi');
SerialPortApi.open("COM1", '{ "mode": "Flow",  "timeout": 3000,  "bufferSize": 256,
"baudRate": 9600,  "parity": "NoParity",  "stopBits": "OneStop",  "dataBits": 8}');
```

⇒ **onHook**

```

SerialPortApi.write("COM1",[0x1, 0x2, 0x3]);
```

- **Null SerialPortApi**.writeAndWait(String *name,* String *requestName,* Number Array *data,* Number *timeout);*

Writes data to the device and waits for an answer until the timeout or accepted data arrives at the serial port. This is a blocking function. The behavior depends on the specified mode:

- When `mode = 'Flow'`, the `onMessage` function will be called when a part of the data appears on the serial port.
- When `mode = 'Buffer'`, the `onMessage` function will be called each time data appears on the serial port, providing the entire buffer.

To consider the data as a valid answer, the `SerialPortApi.acceptRead()` function should be called, indicating that the waiting for data can be stopped.

Example:

⇒ **onHook**

```
InDriver.import('SerialPortApi');
```

⇒ **onHook**

```
SerialPortApi.writeAndWait("COM1","GetDataRequest",[0x1, 0x2, 0x3],3000);
```

⇒ **onMessage**

```
SerialPortApi.acceptRead("COM1");
```

Innovative Data Analytics

TcpSocketApi

The SocketApi provides functions for accessing sockets, writing data, handling write requests with a specified waiting timeout, and reading data asynchronously.

Detailed description:

- **Null TcpSocketApi**.acceptRead(String *name);*

  Accepts data coming to the socket associated with the specified `name`, and it stops waiting for data initiated by the `TcpSocket.writeAndWait` function.

- **Null TcpSocketApi**.connect(String *name,* String *socketSettings);*

  Attempts to make a connection to the *address* on the given *port* from the provided `socketSettings` JSON object, and assigns a handle with the specified 'name'.

  If the JSON object is empty or if some parameters are missing, default values will be assigned. The JSON object should contain the following key-value pairs:

- `"mode"` with available values:
  - `"Ignore"`: Ignore incoming bytes.
  - `"Flow"` (default): Call the `onMessage` function after a specified number of bytes arrive at the serial port.
  - `"Buffer"`: Buffer bytes with a size of `"bufferSize"` and call the `onMessage` function after the next part of bytes arrives, returning the entire buffer.
- `"timeout"`: Timeout for the write request, default is 3000ms.
- `"bufferSize"`: Size of the buffer, default size is 256 bytes.
- `"address"`: may be an IP address in string form (e.g., "127.0.0.1"), or it may be a hostname (e.g., "example.com")
- `"port"` : valid port number:

Example JSON Object with Default Socket Settings:

```
{
  "address":”127.0.0.1”,
  "port": 50000,
  "mode": "Flow",
  "timeout": 3000,
  "bufferSize": 256
}
```

Example:

⇒ **onStartup**

```
InDriver.import('TcpSocketApi');
TcpSocketApi.connect("socket",'{"address":"127.0.0.1","port":50000}');
```

- **Null TcpSocketApi**.disconnect(String *name);*

  Disconnect socket related to ***name*** opened with `TcpSocket.connect` function.

- **Null TcpSocketApi**.disconnectAll(*);*

  Closes all ports opened with `TcpSocket.open` function.

- **Boolean TcpSocketApi**.write(String *name,* Number Array *data);*

Writes data to the socket associated with handle '*name*'. Returns true when all bytes are written, or false if the timeout is reached.

Example:

⇒ **onHook**

```
InDriver.import('TcpSocketApi');
TcpSocketApi.connect("socket",'{"address":"127.0.0.1","port":50000}');
```

⇒ **onHook**

```
TcpSocketApi.write("socket",[0x1, 0x2, 0x3]);
```

- **Boolean TcpSocketApi**.writeAndWait(String *name,* String *requestName,* Number Array *data,* Number *timeout);*

Writes data to the socket associated with handle '*name*' and waits for an answer until the timeout or accepted data arrives at the socket. This is a blocking function. The behavior depends on the specified mode:

- When `mode = 'Flow'`, the `onMessage` function will be called when a part of the data appears on the serial port.
- When `mode = 'Buffer'`, the `onMessage` function will be called each time data appears on the serial port, providing the entire buffer.

To consider the data as a valid answer, the `TcpSocket.acceptRead()` function should be called, indicating that the waiting for data can be stopped.

Example:

⇒ **onHook**

```
InDriver.import('TcpSocketApi');
TcpSocketApi.connect("socket",'{"address":"127.0.0.1","port":50000}');
```

⇒ **onHook**

```
TcpSocketApi.writeAndWait("socket",[0x1,0x2,0x3],"req1");
```

⇒ **onMessage**

```
if (InDriver.taskName() === InDriver.messageSender()) {
	let data = JSON.parse(InDriver.messageData());
	if (JSON.stringify(data.bytes)==="[1,2,3]")) { // loop back
		TcpSocketApi.acceptRead(data.socketName);
	}
}
```

Innovative Data Analytics

TcpServerApi

The SocketAPI offers a TCP-based server, enabling the acceptance of incoming TCP connections and facilitating the reading and writing of data to them.

Detailed description:

- **Null TcpServerApi**.listen(String *tcpServerCfg);*

  Tells the server to listen for incoming connections on address and port from tcpServerCfg JSON Object.

  If the JSON object is empty or if some parameters are missing, default values will be assigned. The JSON object should contain the following key-value pairs:

- `"mode"` with available values:
  - `"Ignore"`: Ignore incoming bytes.
  - `"Flow"` (default): Call the `onMessage` function after a specified number of bytes arrive at the serial port.
  - `"Buffer"`: Buffer bytes with a size of `"bufferSize"` and call the `onMessage` function after the next part of bytes arrives, returning the entire buffer.
- `"address"` with available values:
  - `"Any"`(default)
  - `"AnyIPv4"`
  - `"AnyIPv6"`
  - `"LocalHost"`
  - `"LocalHostIPv6"`
  - `"Broadcast"`
- `"port"` : valid port number, default 50000:

Example

```
InDriver.import('TcpServerApi');
TcpServerApi.listen('{"port":50000,"address":"Any","readMode":"Buffer","bufferSize"
:256}');
```

- **Boolean TcpServerApi**.write(String *name,* Number Array *data,* Number *timeout);*

Writes data to the socket associated with handle '*name*' and waits for an answer until the timeout or accepted data arrives at the socket. The behavior depends on the specified mode:

- When `mode = 'Flow'`, the `onMessage` function will be called when a part of the data appears on the serial port.
- When `mode = 'Buffer'`, the `onMessage` function will be called each time data appears on the serial port, providing the entire buffer.

Example:

⇒ **onHook**

```
InDriver.import('TcpServerApi');
TcpServerApi.listen('{"port":50000,"address":"Any","readMode":"Buffer","bufferSize"
:256}');
```

⇒ **onHook**

```

```

⇒ **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");
        }
        if (data.data==="text data"){
                TcpServerApi.write(data.socketName, "text data accepted");
        }
}
```