

# Table of Contents

Example Code Script .....	1
<b>Managing Scripts</b> .....	1
<b>Script Handling</b> .....	2
<b>The Built-in Script Editor</b> .....	3
Going Live .....	5
Useful Shortcuts .....	6
<b>Viewing the Script Log</b> .....	6
<b>Writing Script Code</b> .....	6
JavaScript Basics .....	7
Script API .....	8
Accessing CoDaBix .....	8
Find a Node and Log its Value .....	8
Node Events .....	9
Write a Value to a Node .....	10
Async Functions .....	11
Reading Node's Values Using a Synchronous Read .....	12
File Access .....	13
Basic File Operations .....	13
Reading and Writing Text Files .....	14
HTTP Handlers .....	16
Dynamically Generate a Text Page .....	16
Generate an HTML page with an input form .....	17
Display History Values as Diagram .....	20
Extended HTTP Programming .....	25
WebSocket Connections in an HTTP Handler .....	25
Simple Echo WebSocket .....	30
"Chat" with Background Send Operations .....	31
Sending HTTP Requests .....	35
Simple GET requests .....	36
Accessing a JSON-based REST API .....	36
<b>Recommended Best Practices</b> .....	37
Storing Passwords Securely .....	39



# Script Interface Plugin Development Guide

## Example Code Script

```
// Find the node by specifying the node path.
const pressureNode = codabix.findNode("/Nodes/Injection
molding/Pressure", true);

// Set an interval timer that will call our function every 3 seconds.
timer.setInterval(function () {

    // Generate a random number between 20 and 150.
    let randomNumber = Math.random() * 130 + 20;

    // Write the value to the Node.
    codabix.writeNodeValueAsync(pressureNode, randomNumber);

}, 3000);
```

## Managing Scripts

The Codabix Web Configuration provides the item “Scripts” to create, edit and delete scripts. You can also stop scripts so that they are not executed anymore, without deleting them.

<

Property	Description
Name	Name of the script, used to identify it. This name will also be used in stacktraces when an exception occurred.
Description	You can enter a more detailed description of the script.

Property	Description
Editor Strictness Level	<p>Determines how strict the editor will handle certain code.</p> <p><b>Low</b> (default): The editor will not criticize the cases described for the following options.  <b>Medium</b>: The editor will criticize implicit any types, implicit returns, and fallthrough cases in <code>switch</code> statements.  <b>High</b>: Additionally to the cases described in the “Medium” level, the editor will criticize unused local variables and unused function parameters.</p> <p>Note: For Library Scripts (available in a later version) the level is always “High”.</p>
Script State	<p><b>Enabled</b>: The Script shall be run.  <b>Disabled</b>: The Script shall be stopped.</p>
CurrentScriptState	<p>Shows the actual state of the Script.</p> <p><b>NotRunning</b>: The Script is not running, either because it is disabled or because it has just been created and there is not yet a live code for it.  <b>Running</b>: The Script has been started and there was no error since then (this can also be the case if a Script has recently been restarted after an uncaught exception).  <b>Stopped</b>: The Script has been started and has been running, but no more event listeners or callbacks are active.  <b>StoppedAndScheduledForRestart</b>: The Script has been stopped because an uncaught exception occurred while executing it. It will be automatically restarted after a short period of time.</p>

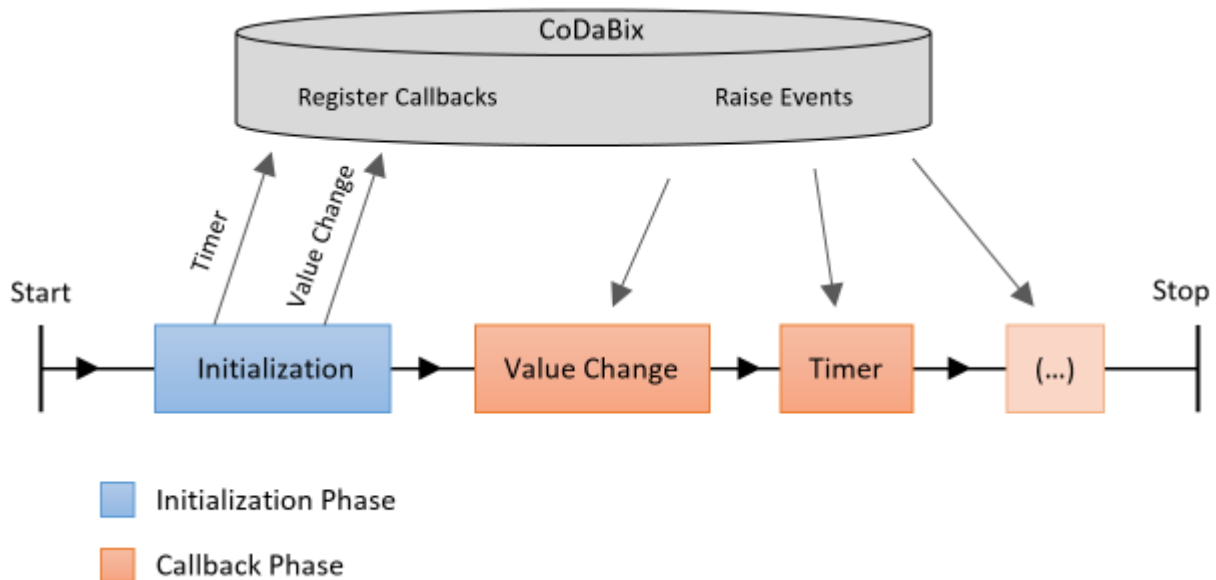
The default value of “Editor Strictness Level” is “Low”. We recommend this setting if you are just beginning with Scripts and JavaScript. If you are an experienced TypeScript developer, we recommend setting the value to “Medium” or “High” so that the editor can help you manage a clean code base.

**Note:** It can take up to 3 seconds until a change (e.g. enabling / disabling the script) will become effective and another 3 seconds until the `CurrentScriptState` and the Script Log are updated.

## Script Handling

Once you created a live version of a script (see section [Going Live](#)), it will automatically be started, as long as its state is set to “Enabled”. On startup, the script is in the “Initialization Phase”. During this phase, the script can register callbacks (e.g. for events or for a timer). When the callback is called, the script is in the “Callback Phase” (but the script can still register further callbacks in this phase).

The following diagram illustrates the phases of a script:




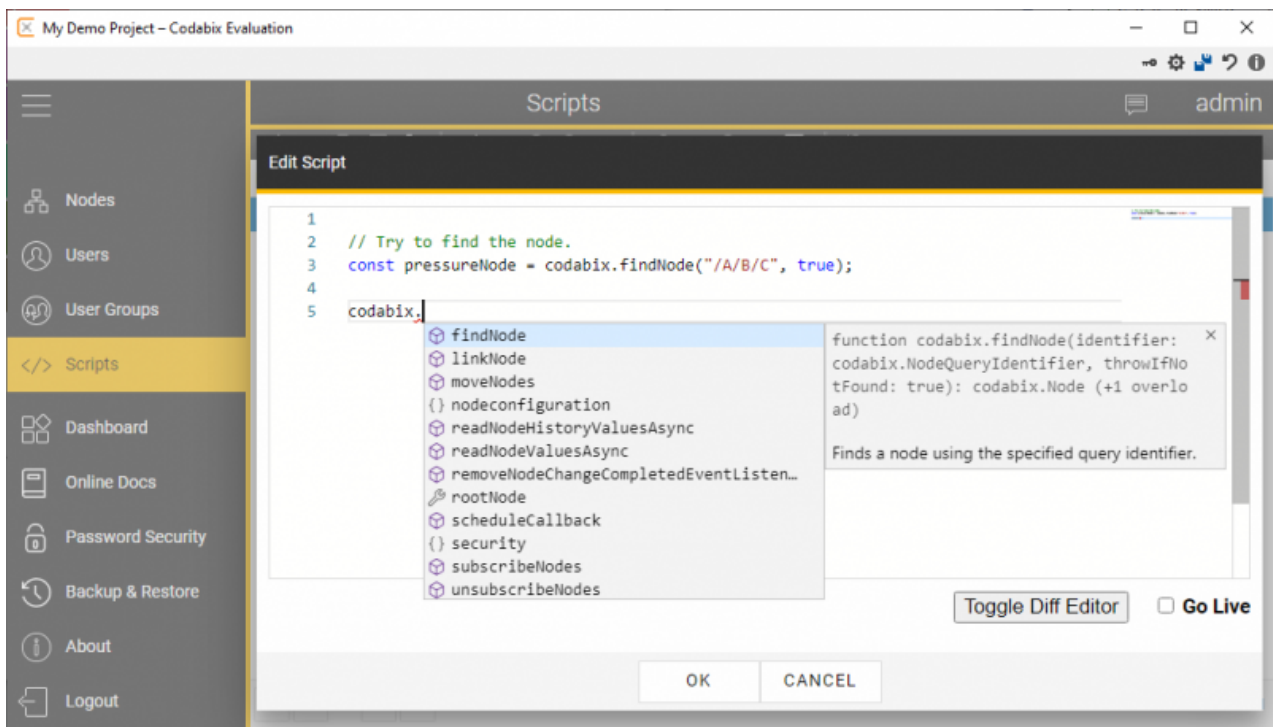
Note: Although not shown in the diagram above, the script can still register new callbacks for other events when it is already in the Callback Phase.

A timeout of about **15000 ms** is applied to the script to protect it against unintended infinite loops, for example `while (true) {}`. If a script is not finished after the timeout, it is stopped and treated as if an uncaught exception has occurred.

In both the Initialization Phase and the Callback Phase the script is automatically restarted after a short period of time (about 3 seconds) when an uncaught exception occurs.

## The Built-in Script Editor

By clicking the Script Editor icon () , the editor will appear and you can write your script code.



If you have already worked with Visual Studio or VS Code, the Script Editor will look familiar to you (in fact, the editor is based on the Monaco Editor from VS Code). The editor provides IntelliSense for CoDaBix API methods / interfaces and for built-in JavaScript classes as you type, as shown in the screenshot above.

If you hover on variable or method calls, a tooltip appears that shows the type:

```
// Write the value to the node. let randomNumber: number
codabix.writeNodeValueAsync(pressureNode, randomNumber);
```

If you have an error in your script, the editor will show red squiggly lines on it and show the error if you hover on it:

```
Property 'writeNNodeValueAsync' does not exist on type 'typeof codabix'.
// Write any
codabix.writeNNodeValueAsync(pressureNode, randomNumber);
```

When you right-click at a position in the code, a context menu appears with useful commands. For example, you can find all references to a specific variable in your code (and e.g. rename it):

```

11 // Write the value to the node.
12 codabix.writeNodeValueAsync(pressureNode, randomNumber);

```

References – 2 references

```

2 const pressureNode = codabix.findNode("/Nodes/Injection molding/Pressure");
3 if (pressureNode != null) {
4
5 // Set an interval timer that will call our function every 3 seconds.
6 timer.setInterval(function () {
7
8 // Generate a random number between 20 and 150.
9 let randomNumber = Math.random() * 130 + 20;
10
11 // Write the value to the node.
12 codabix.writeNodeValueAsync(pressureNode, randomNumber);
13
14 }, 3000);
15 }
16
17

```

let randomNumber = ...  
pressureNode, random...

## Going Live

The Script Editor allows you to write and save code for the script (“**draft**”) without actually running it. Only when you select “**Go Live**”, the current draft script code will be saved as the “**live version**” and will actually be run. This allows you to progressively work on the script code without affecting the currently executed live version. With the “Toggle Diff Editor”, you can switch to a diff editor to compare your changes between the live version and your current draft.

Edit Script

```

1 // Try to find the node by specifying the node path.
2 const pressureNode = codabix.findNode("/Nodes/Injection molding/Pressure");
3 if (pressureNode != null) {
4
5 // Set an interval timer that will call our function every 3 seconds.
6 timer.setInterval(function () {
7
8 // Generate a random number between 20 and 150.
9 let randomNumber = Math.random() * 130 + 20;
10
11 // Write the value to the node.
12 codabix.writeNodeValueAsync(pressureNode, randomNumber);
13
14 }, 3000);
15 }
16
17

```

```

1 // Try to find the node by specifying the node path.
2 const pressureNode = codabix.findNode("/Nodes/Injection molding/Pressure");
3 if (pressureNode != null) {
4
5 // Set an interval timer that will call our function every 3 seconds.
6 timer.setInterval(function () {
7
8 // Generate a random number
9 // between 20 and 150.
10 let randomNumber = Math.random() * 130 + 20;
11
12 // Write the value to the node now.
13 codabix.writeNodeValueAsync(pressureNode, randomNumber);
14
15 }, 3000);
16 }
17
18

```

Toggle Diff Editor ☐ Go Live

✓ ✕

Once you are finished with editing your script, make sure to check the “**Go Live**” checkbox and then click save. This will make your current script draft become the live version so that it is actually executed.

If you have a compile time error in your script when trying to go live, an error box will appear describing the error:

Error:

```

5 let result = 123 + true;

```

Line 5: Operator '+' cannot be applied to types 'number' and 'boolean'.

Otherwise, the Script Editor will close and the new script code will run after some seconds.

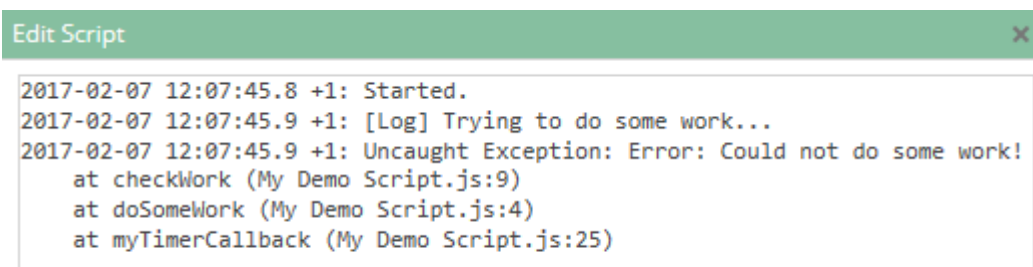
## Useful Shortcuts

- **Ctrl+F:** Find/Replace
- **Ctrl+F2:** Rename (change all occurrences)
- **Shift+F12:** Find all references
- **Ctrl+F12:** Go to definition
- **Ctrl+K, Ctrl+C:** Comment out the selected lines
- **Ctrl+K, Ctrl+U:** Uncomment the selected lines

## Viewing the Script Log

Each script has a log associated with it. When a script has been started (or an uncaught exception occurred), an entry will be made in the log. Additionally, you can create a log entry directly from the script code by calling `logger.log()`.

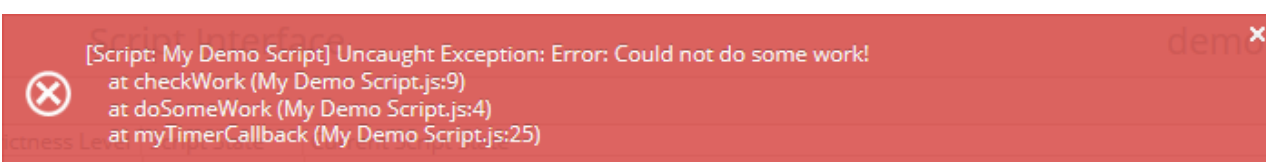
If you click on the log button (📖), a dialog with the log content will appear. For example, if the Script started, but an uncaught exception occurred in a callback, the log might look like this:



```
2017-02-07 12:07:45.8 +1: Started.
2017-02-07 12:07:45.9 +1: [Log] Trying to do some work...
2017-02-07 12:07:45.9 +1: Uncaught Exception: Error: Could not do some work!
    at checkWork (My Demo Script.js:9)
    at doSomeWork (My Demo Script.js:4)
    at myTimerCallback (My Demo Script.js:25)
```

In case an exception occurs, the log entry contains a stack trace showing the line numbers of the script (after the colon) that mark the position in the code at which the corresponding functions were executing when the exception occurred.

**Note:** When an uncaught exception occurs, an error message will also be shown in the Runtime Log:



```
[Script: My Demo Script] Uncaught Exception: Error: Could not do some work!
    at checkWork (My Demo Script.js:9)
    at doSomeWork (My Demo Script.js:4)
    at myTimerCallback (My Demo Script.js:25)
```

## Writing Script Code



## JavaScript Basics

Given below is a brief summarization of JavaScript basics. For a more detailed tutorial, please visit the [JavaScript Guide](#) on MDN.

In a script, you can declare variables that store values with `let` and `const` (`const` means the variable cannot be modified). You can assign values through the `=` operator (whereas `==` is used to check for equality):

```
let count = 123;  
const myText = "Hello World!";
```

JavaScript supports a number of basic value types:

- **number**: A number (which is a double precision floating point value) can represent both integers and decimals. You can use numbers to do calculations, e.g.:

```
let result = (2 + 0.5) * 3;    // 7.5
```

- **boolean**: A boolean is either `true` or `false`. A boolean can be the result of comparison operators and used for control flows like `if`, `while` etc.
- **string**: A string can consist of an arbitrary number of characters and be used to represent text. Strings can be linked using the `+` operator:

```
let displayText = "The result of 5+6 is " + (5+6);    // "The  
result of 5+6 is 11"
```

- **object**: An object stores properties that consist of a **key** (string) and a **value** (any value type). For example, the `codabix` object contains properties that are methods, e.g. `findNode`. Object properties are mostly accessed using dot (`.`) notation (`codabix.findNode(...)`, `codabix.rootNode`, ...).

You can use control flow statements to do comparisons:

```
let result = "The value is"  
if (count > 200) {  
    result += "greater than";  
}  
else if (count < 200) {  
    result += "lower than";  
}  
else {  
    result += "equal to";  
}  
result += " 200.";
```

You can create a function that will contain code which needs to be run more than once. For example, you could create a function that calculates the average of two values:

```
function average(value1, value2) {  
    return (value1 + value2) / 2;  
}  
  
// Calculate the average of 100 and 250 and write it to the Script Log.  
logger.log("The average of 100 and 250 is " + average(100, 250) +  
".");
```

When you run this code, it will print something like this to the script's log:

```
2016-09-28 14:57:41.7 Z: [Log] The average of 100 and 250 is 175.
```

## Script API

The Script Interface Plugin provides the following API namespaces that can be used in a script:

- **codabix**: Contains all CoDaBix related functionality, e.g. to access and modify Nodes.
- **timer**: Contains methods to create a timer, so that you can let a function of your script be called back at a regular interval.
- **logger**: Contains a `log` method that allows you to write to the script log.
- **storage**: Allows you to persist information across restarts of the script.
- **io**: Provides I/O operations, e.g. [File Access](#).
- **net**: Provides network-related operations, e.g. to register an HTTP handler.
- **runtime**: Provides functions to interact with the script runtime environment.

Note: The Script Editor supports **IntelliSense**, so you can see which methods are available in the **codabix** namespace just by typing `codabix.` (notice the dot after “codabix”). Similarly, when a method returns an object (for example a Node), you can again type a dot after it to see which methods it has.

## Accessing CoDaBix

### Find a Node and Log its Value

Let's assume you installed CoDaBix with the “Demo-Data (Continuous)” plugin and want to access the Node **Nodes → Demo-Data → Temperature**. To do this, you first need to get the Node path or the Identifier of the Node. To do this, open the Node view in the CoDaBix Web Configuration, select the relevant Node and click the **Access** symbol (🔑). Then copy the “Absolute Node Path”. We then specify this path in the `codabix.findNode()` method as well as a `true` parameter so that the method throws an exception if the node could not be found:

```
// Find the "Temperature" node and check if the node  
// has a value.  
const temperatureNode = codabix.findNode("/Nodes/Demo-  
Nodes/Temperature", true);  
if (temperatureNode.value !== null) {
```

```
// OK, node has a value. Now log that value.  
logger.log("Current Temperature: " + temperatureNode.value.value);  
}
```

Your script log will then look like this:

```
2016-09-28 15:08:45.2 Z: Started.  
2016-09-28 15:08:45.3 Z: [Log] Current Temperature: 71  
2016-09-28 15:08:45.3 Z: Stopped.
```

However, in this example only one value is logged. This is because when the script is started, the code which finds the Node and logs the value is run, but after that the script is finished.

Now, if we want to log the value not only once but every 5 seconds, we can do this by creating a timer and supplying a function that the timer will call at a regular interval (note: Instead of function () {...}, for a callback you should use a **fat arrow function**: () => {...}).

```
// Find the "Temperature" node.  
const temperatureNode = codabix.findNode("/Nodes/Demo-  
Nodes/Temperature", true);  
  
// Now create a timer that will log the node's value  
// every 5 seconds.  
const interval = 5000;  
timer.setInterval(() => {  
  
    // If the node has a value, log it.  
    if (temperatureNode.value != null) {  
        logger.log("Current Temperature: " +  
temperatureNode.value.value);  
    }  
  
}, interval);
```

When you run this script your script Log will look like this after some seconds:

```
2016-09-28 15:15:42.6 Z: Started.  
2016-09-28 15:15:47.6 Z: [Log] Current Temperature: 70  
2016-09-28 15:15:52.6 Z: [Log] Current Temperature: 75  
2016-09-28 15:15:57.6 Z: [Log] Current Temperature: 63  
2016-09-28 15:16:02.6 Z: [Log] Current Temperature: 71
```

## Node Events

The example above uses a timer that calls a function in a regular interval. However, it is also possible to register for specific events of a Node:

- **ValueChanged:** Raised when a value has been written to the node (property value).  
**Note:** This event is also raised if the new value is equal to the previous value. To determine if the value has actually changed, you can check the `isValueChanged` property of the listener argument.

- **PropertyChanged:** Raised when a property (other than value) of the Node has been changed, e.g. name, displayName etc.
- **ChildrenChanged:** Raised when one or more children Nodes of the current Node have been added or removed.

You can handle the event by adding an **Event Listener** (callback) to the Node whose event you are interested in.

Example:

```
// Find the "Temperature" node and add a handler for the
"ValueChanged" event.
const temperatureNode = codabix.findNode("/Nodes/Demo-
Nodes/Temperature", true);

temperatureNode.addValueChangedEventListener(e => {

    // Log the old and the new value of the node.
    logger.log("Old Value: " + e.oldValue?.value
        + ", New Value: " + e.newValue?.value);

});
```

**Note:** You cannot (synchronously) read or write Node values (or do other changes to Nodes) from within an Node event listener. If you want to do this, use `codabix.scheduleCallback()` to schedule a callback that is executed as soon as possible after the event listener is left.

### Write a Value to a Node


You can also write values to a Node from a script. For example, in the Node configuration, select the **“Nodes”** Node and create a datapoint Node with the name “Counter” and select **on Value Change** for “History Options”. Then create a script with the following code:

```
const counterNode = codabix.findNode("/Nodes/Counter", true);

// Declare a counter variable.
let myCounter = 0;

// Set a callback that increments the counter and writes the
// current value to the node until the value is 5.
let timerID = timer.setInterval(() => {
    myCounter = myCounter + 1;
    codabix.writeNodeValueAsync(counterNode, myCounter);

    if (myCounter == 5)
        timer.clearInterval(timerID);
}, 500);
```

Then, change back to the Nodes view, select the “Counter” Node and open the history values ():

History Values					
Node	Category	Status	Receive Timestamp	Creation Timestamp	Value
[37] Counter	0	0	10.10.2016 11:25:38	10.10.2016 11:25:38	5
[37] Counter	0	0	10.10.2016 11:25:38	10.10.2016 11:25:38	4
[37] Counter	0	0	10.10.2016 11:25:37	10.10.2016 11:25:37	3
[37] Counter	0	0	10.10.2016 11:25:37	10.10.2016 11:25:37	2
[37] Counter	0	0	10.10.2016 11:25:36	10.10.2016 11:25:36	1

You can see now that the values 1, 2, 3, 4 and 5 have been written to the Node with a delay of 0.5 seconds.

Note: There is also an easier way of writing this code (withouth callbacks) using an **Async Function** as shown in the next chapter.

## Async Functions

The example shown in the previous chapter creates a timer by setting a callback. However, such code can quickly become confusing when you have more complex conditions. **Async Functions** allow to write the code in a simpler way as the code looks like synchronous code (without callbacks). CoDaBix provides a few async functions that can be recognized from their name ending in Async.

For example, the above code can be rewritten like this, using the `timer.delayAsync()` function:

```
const counterNode = codabix.findNode("/Nodes/Counter", true);

// Write the values 1 to 5 to the node, and wait 0.5 seconds between
// each write.
for (let i = 1; i <= 5; i++) {
  await codabix.writeNodeValueAsync(counterNode, i);
  await timer.delayAsync(500);
}
```

Notice using the keyword `await` here. **Await** means something like “interrupt the execution here until the asynchronous operation of the function has completed”. In this case the `delayAsync` function returns a Promise object that is fulfilled after 0.5 seconds have passed. As soon as the Promise is fulfilled the execution continues. Other async functions also return Promise objects, which may be fulfilled with a value, if applicable.

If you did not write `await`, your code would not wait 0.5 seconds but instead immediately continue with the next iteration. It is important to note that when awaiting an operation, in the meantime other code can still be executed, e.g. you may receive an event from the Node while your code pauses at the `await` position.

In the example above we also use `await` to wait for `writeNodeValueAsync`. Writing Node

values may take some time if the Node is connected to a device, e.g. a S7 PLC device. In that case, execution would pause until the value has actually been written to the device. If you do not want to wait for that, you can also remove the `await` here (in which case you will need to write the `void` operator before the function call to signal to the compiler that the returned Promise has intentionally been discarded).

However, if you try this code directly it will not yet work because in order to use `await` you need to mark the surrounding function using the `async` keyword. If the surrounding `async` function is the **main** function, you should wrap it in a call to `runtime.handleAsync()` so that uncaught exceptions are not silently swallowed:

### Async-Template.js

```
runtime.handleAsync(async function () {  
  
    // Your code here...  
  
} ());
```

Here is a complete example of a Script using Async Functions:

```
runtime.handleAsync(async function () {  
  
    for (let i = 0; i < 10; i++) {  
        logger.log("Iteration " + i);  
        await timer.delayAsync(1000);  
    }  
  
} ());
```

**Note:** If you want to use an `async` function as callback for an event listener, you should also wrap it in `runtime.handleAsync`, as shown in the following case where we handle the event when a Node gets a new value:

```
const myNode = codabix.findNode("/Nodes/A", true);  
  
myNode.addValueChangedEventListener(() => runtime.handleAsync(async  
() => {  
    // Do async stuff...  
}) ());
```

### Reading Node's Values Using a Synchronous Read

Another example of an `async` function is `codabix.readNodeValuesAsync()`. This method invokes a [Synchronous Read](#) on a device and reading values from the device may take some time. Therefore you should use `await` to pause the execution until the resulting values arrive:

```
runtime.handleAsync(async function () {
```

```
const node1 = codabix.findNode("/Nodes/A", true);
const node2 = codabix.findNode("/Nodes/B", true);

// Do a synchronous read, and pause until the values arrive from
the device.
let [nodeValue1, nodeValue2] = await
codabix.readNodeValuesAsync(node1, node2);

logger.log("Node1 Value: " + (nodeValue1?.value)
    + ", Node2 Value: " + (nodeValue2?.value));

} ());
```

## File Access

The namespaces **io.file**, **io.directory** and **io.path** contain methods and classes to work with files and directories. For example, you can read and write text files, copy, move or delete files, and enumerate all files in a specific directory.

Note that file access is subject to the **Access Security** restrictions that have been defined in the [CoDaBix Project Settings](#).

Most of the I/O operations are implemented as **Async Functions** that return a Promise object. This is because I/O operations may take some time to complete (depending on the file system). In order to not block CoDaBix, the I/O operations are run in the background. You can call them in the async functions using the **await** keyword.

**Note:** On Windows 10 Version 1511 and older (and Windows Server 2012 R2 and older), e.g. on Windows 7, the **maximum file path length** is limited to **260 characters** (MAX\_PATH). On Windows 10 Version 1607 and higher (as well as Windows Server 2016 and higher) longer path names can be used. However, for this you will need to enable the setting "Enable Win32 long paths" in the Windows Group Policy, see [Enabling Win32 Long Path Support](#).

## Basic File Operations

**Enumerate files of the CoDaBix project "log" directory:**

```
runtime.handleAsync(async function () {

    // Get the path to the Codabix "log" directory. We use the
    Codabix-defined environment
    // variable "%CodabixProjectDir%" for this case.
    // combinePath is an OS independent way to combine path elements.
    const codabixLogDir = io.path.combinePath(
        runtime.expandEnvironmentVariables("%CodabixProjectDir%"),
        "log");
    const fileList = await io.directory.GetFilesAsync(codabixLogDir);

    let result = "";
```

```
for (let file of fileList) {
    result += "File: " + file + "\n";
}

logger.log("Files in " + codabixLogDir + ":\n" + result);

} ());
```

The result might look like this:

#### Edit Script

```
2017-01-17 13:08:02.7 Z: [Log] Files in C:\Users\developer4\Desktop\NewCodabixData6\log:
File: OPC-UA Server Interface.Default Channel.log
File: S7 TCP-IP Device.New Channel 1.log
```

- `io.path.combinePath()`: Combines two or more path elements into one path in an OS independent way. For example, on Windows the path separator is \, while on Linux it is /. This method automatically uses the correct separator to combine the paths.
- `io.directory.GetFilesAsync()`: Returns a `string[]` array containing the file names of the specified directory (similarly, `io.directory.getDirectoriesAsync()` returns the subdirectory names).
- `io.file.copyFileAsync()`: Copies a file.
- `io.file.moveFileAsync()`: Renames or moves a file.
- `io.file.deleteFileAsync()`: Deletes a file.
- `runtime.expandEnvironmentVariables()`: Replaces the name of each environment variable (enclosed in two % characters) in the specified string with their value. Codabix defines the following environment variables in addition to the OS's variables:
  - `CodabixProjectDir`: Contains the path to the currently used Codabix project directory.
  - `CodabixInstallDir`: Contains the path where Codabix has been installed.

## Reading and Writing Text Files

### Writing a text file in one step:

```
runtime.handleAsync(async function () {

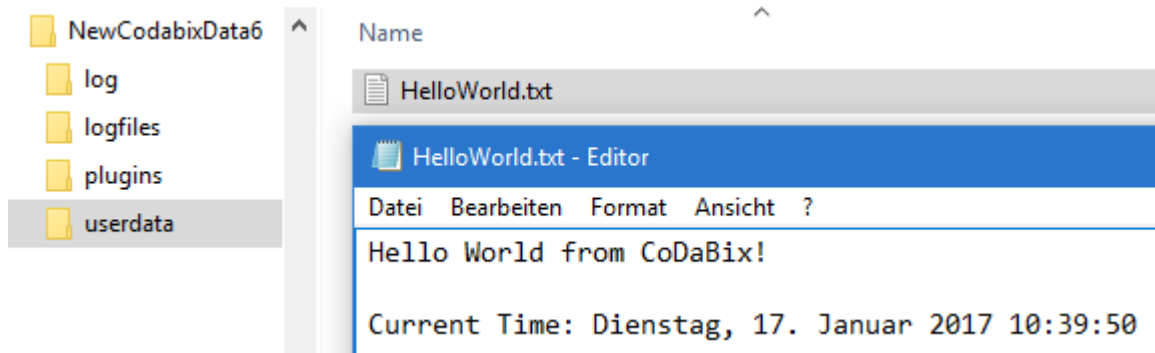
    // Create a string and write it into a textfile (HelloWorld.txt).
    let filePath =
io.path.combinePath(runtime.expandEnvironmentVariables("%CodabixProjectDir%"),
        "userdata", "HelloWorld.txt");
    let content = "Hello World from CoDaBix!\r\n\r\n" +
        "Current Time: " + new Date().toLocaleString();

    await io.file.writeAllTextAsync(filePath, content);

} ());
```



This will create a textfile HelloWorld.txt in the userdata directory of your Codabix project directory that might look like this:



By calling `io.file.writeAllTextAsync()`, the file is written in one step (and with `io.file.readAllTextAsync()`, it is read in one step). However, you can also read or write text files on a line-by-line basis as in the following example:

### Reading a text file line-by-line:

```
runtime.handleAsync(async function () {

    // We want to read from the current Codabix Runtime Log file.
    const runtimeLogDir = io.path.combinePath(
        runtime.expandEnvironmentVariables("%CodabixProjectDir%"),
        "log");
    const runtimeLogFiles = await
    io.directory.GetFilesAsync(runtimeLogDir);

    // The returned files are ordered ascending by their name; so we
    use the last
    // entry to get today's log file.
    const logFile = io.path.combinePath(runtimeLogDir,
        runtimeLogFiles[runtimeLogFiles.length - 1]);

    // Open the file using a Reader and read the first 5 lines.
    let result = "";
    let reader = await io.file.openReaderAsync(logFile);
    try {
        let lines = await reader.readLinesAsync(5);
        for (let i = 0; i < lines.length; i++) {
            // Append the line to the result string
            result += "\n[" + (i + 1) + "]: " + lines[i];
        }
    }
    finally {
        // Ensure to close the reader when we are finished.
        await reader.closeAsync();
    }

    // Log the result string.
    logger.log(result);
});
```

```
} ());
```

This code reads the first 5 lines of the current Codabix runtime logfile and prints it to the script log:

Edit Script

```
2017-01-17 15:23:36.1 +1: [Log]
[1]: 2017-01-17 09:11:20.7 +01:00: [Info] CoDaBix® Engine starting...
[2]:   • CoDaBix® Version:      0.10.1
[3]:   • OS Version:           Windows 10 Anniversary Update (Version 1607, RS1) [10.0.14393]
[4]:   • .NET Version:         .NET Framework 4.6.2 [4.6.01586]; CLR: v4.0.30319
[5]:   • Runtime Architecture: AMD64 (64-bit)
2017-01-17 15:23:36.0 +1: Started.
```

## HTTP Handlers

The namespace `net` provides methods to register **HTTP Handlers**, which you can use to specify a script function that shall be called every time a client sends an HTTP request to the Codabix web server. For example, you can dynamically generate HTML pages, similar to PHP or ASP.NET. Additionally, you can handle incoming **WebSocket connections** (see next chapter).

To register a HTTP handler you must call the `net.registerHttpRequest()` method and pass a path as well as a callback to it. The callback will then be called each time an HTTP request for the URL path `/scripthandlers/<path>` arrives, where `<path>` represents the registered path.

Note that a particular path can only be registered once across all scripts at the same time. If another script has already registered a handler for the same path, the method throws an exception.

Also note that the path must be specified as “raw”, meaning it should be URL-encoded. For example, if the user should be able to enter `/scripthandlers/März` as address in the browser, you would need to specify `M%C3%A4rz` as path.

The callback needs to be a function that returns a `Promise` object (which happens automatically when using an **async function**). When a HTTP request arrives, the function is called, and the HTTP request stays active until the `Promise` object is “fulfilled” (the `async` function returns). The callback gets a `net.HttpContext` as parameter which contains the properties `request` and `response` in order to access objects representing the request of the client and the response to the client.

When the callback throws an exception (or rejects the returned `Promise`), the request is aborted and a warning is logged to the runtime log, but the script continues to run.

### Dynamically Generate a Text Page

Imagine we want to output the current time and the current number of Codabix Nodes as text page when the user enters <http://localhost:8181/scripthandlers/hello-world> in the browser on the current machine (provided that the **Local HTTP Port** is set to the default value of 8181). To do this, the following script code registers an HTTP handler for this path which outputs a text containing the information when requested:

```
runtime.handleAsync(async function () {

    net.registerHttpRequest("hello-world", async ctx => {
        let response = ctx.response;

        // Create a text with the current date/time and the number of
        // Codabix nodes.
        let text = "Hello World! Time: " + new
Date().toLocaleTimeString() + "\n\n" +
        "Node Count: " + countNodes(codabix.rootNode.children);

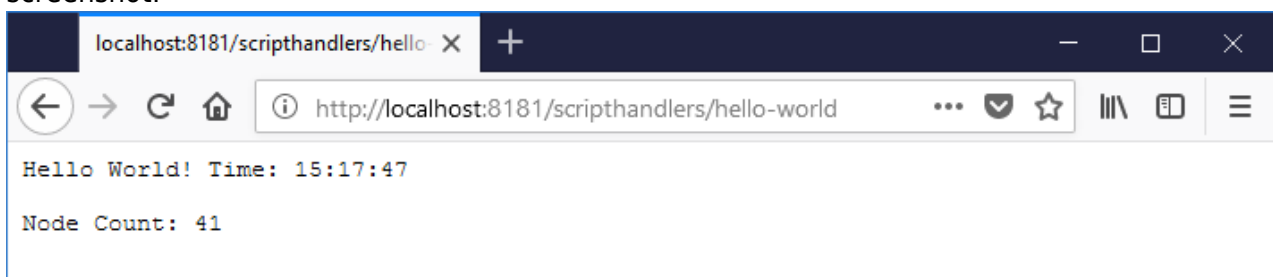
        // Set the Content-Type to the plain text format so that the
        // browser knows
        // how to display the document.
        response.contentType = "text/plain";

        // Finally, write the generated text into the response.
        await response.writeBodyCompleteAsync(text);
    });

    function countNodes(nodes: codabix.Node[]): number {
        let count = nodes.length;
        for (let node of nodes) {
            count += countNodes(node.children);
        }
        return count;
    }

})();
```

When opening this URL in a browser you will get an output similar to the one in this screenshot:



When refreshing the page with F5, you should see the current date and time being displayed.

The script uses the method `writeBodyCompleteAsync()` of the response object to write the generated text to the browser (using UTF-8 as text encoding). It is recommended to use this method instead of `writeBodyAsync()` when you can generate the whole response text in the script. In contrast, you can use `writeBodyAsync()` when you only want to generate small pieces of the response text and send them immediately to the client.

### Generate an HTML page with an input form

Imagine we want to create an HTML page where the user can enter a node path. When sending

the form, the value of the specified node shall be read using a synchronous read and then be output on the page. This is done by the following script code:

```
runtime.handleAsync(async function () {

    const readNodeHandlerPath = "read-node-value";
    net.registerHttpRequest(readNodeHandlerPath, async ctx => {
        let request = ctx.request;
        let response = ctx.response;

        // Create an HTML page with a form in order to enter a node
        path.
        // When entered, we find the node and start a synchronous
        read.

        // Check whether the form has already been sent. If not, we
        write
        // an example path in the text box.
        const inputNodePathName = "nodePath";
        const inputNodePathValue =
        request.queryString[inputNodePathName];
        let resultMessage = "";

        if (inputNodePathValue != undefined) {
            // The form has been sent, so find the node now.
            let node = codabix.findNode(inputNodePathValue);
            if (!node) {
                // We couldn't find the node.
                resultMessage = `Error! Node with path
                '${inputNodePathValue}' was not found!`;
            }
            else {
                // Start a synchronous read... (This may take a while,
                // depending on the device.)
                // The HTTP request stays active during that time.
                let resultValue = (await
                codabix.readNodeValuesAsync(node))[0];
                if (resultValue == null) // The node doesn't yet have
                a value
                resultMessage = `Result: Node does not have a
                value!`;
                else
                resultMessage = `Value of Node
                '${inputNodePathValue}': ${resultValue.value} ${node.unit || ""}`;
            }
        }

        let html = `<!DOCTYPE html>
        <html>
        <head>
```

```
<meta charset="UTF-8" />
<title>Read Node Value</title>
</head>
<body>
  <h1>Read Node Value!</h1>
  <form method="get" action="{xml.encode(readNodeHandlerPath)}">
    Enter Node Path:
    <input type="text" name="{xml.encode(inputNodePathName)}"
style="width: 250px;"
      value="{xml.encode(inputNodePathValue == undefined ?
"/Nodes/Demo-Nodes/Temperature" : inputNodePathValue)}" />
    <button>OK</button>
  </form>
  <p>
    {xml.encode(resultMessage)}
  </p>
</body>
</html>`;

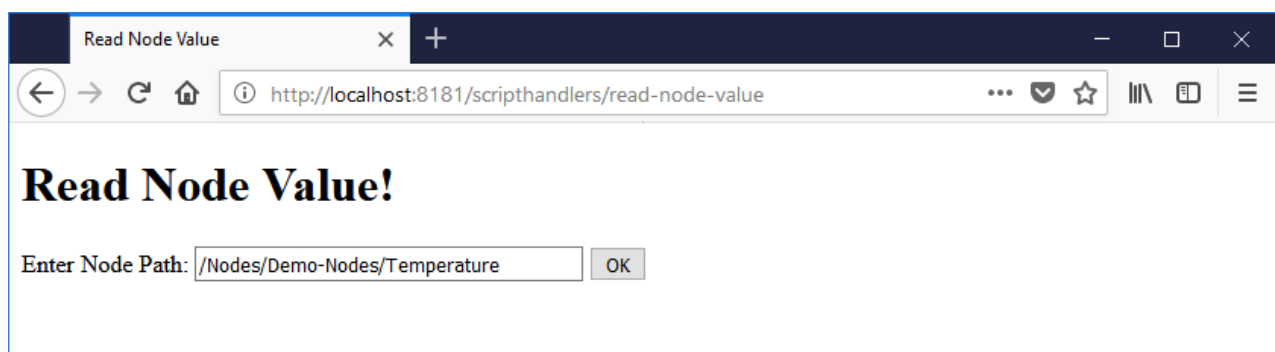
// Set the Content-Type to HTML, and write the generated HTML
into the response.
response.contentType = "text/html";
await response.writeBodyCompleteAsync(html);
});

}());
```

**Note:** The script uses the method `io.util.encodeXml()` in order to encode strings so that they can safely be placed within HTML or XML text (or attribute values), without providing an attack surface for HTML injection or Cross-Site-Scripting (XSS). This is especially important when outputting strings in an HTML page that might originate from the user or external sources.

When you now open the URL

<http://localhost:8181/scripthandlers/read-node-value> in the browser, the following form is shown:



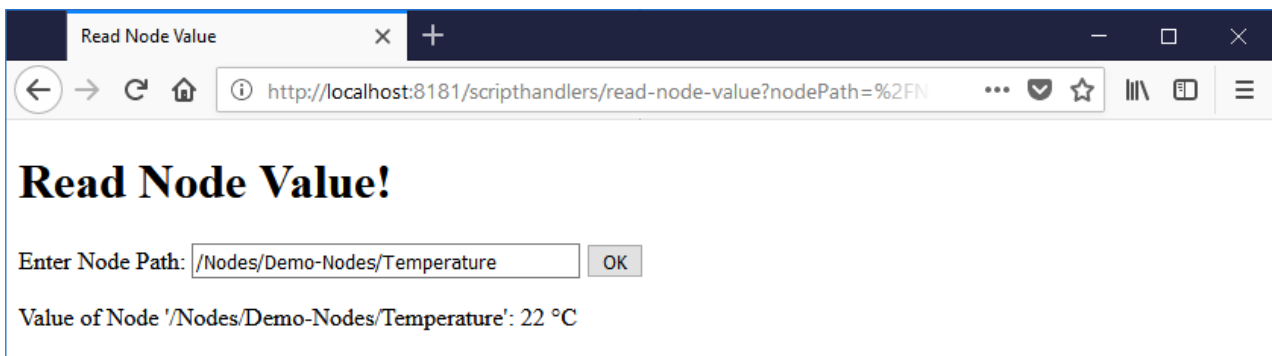
Read Node Value

http://localhost:8181/scripthandlers/read-node-value

## Read Node Value!

Enter Node Path:

When clicking on the “OK” button (and you have the Demo Data plugin installed), the current value of the temperature demo node will be shown:

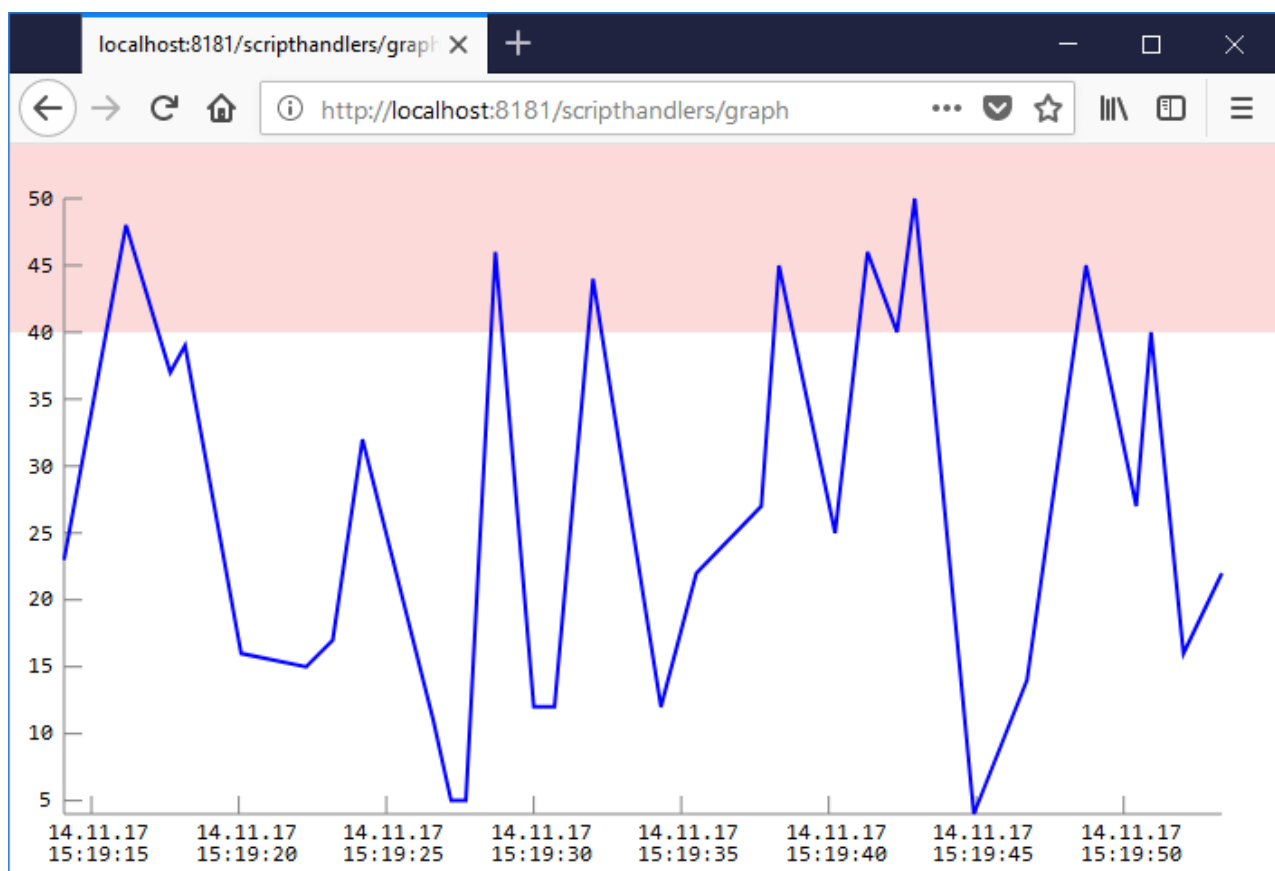


This means the script has reacted on the sent form and has output the current value of the node. Because the form uses the GET method, the parameter is visible in the URL as appended query string: `?nodePath=%2FNodes%2FDemo-Nodes%2FTemperature`

Similarly, you could enter a path to a S7 variable, an OPC UA Client variable, and so on. In these cases, not only the currently stored value of the node is output, but a synchronous read is started by the call to method `codabix.readNodeValuesAsync()` which reads the value from the device and then outputs it in the HTML page.

### Display History Values as Diagram

With an HTTP handler you can also generate an SVG image that displays history values in a diagram (in the following example, using the Gradient demo node):



This can be done with the following script code:

```
// Utilities for generating a SVG diagram.
namespace SvgLibrary {
  export interface Value {
    date: number,
    value: number
  }

  type DPoint = [number, number];

  const escapeXml = xml.encode;

  /**
   * Formats pixel coordinates. A max. precision of 3 should be good
   enough here.
   * @param x
   */
  let pform = (x: number): string => (Math.round(x * 1000) /
1000).toString();

  function determineGap(x: number): number {
    let sign = x < 0 ? -1 : 1;
    x = Math.abs(x);
    let tenLog = Math.floor(Math.log10(x));

    if (x > 5 * 10 ** tenLog)
      x = 10 * 10 ** tenLog;
    else if (x > 2 * 10 ** tenLog)
      x = 5 * 10 ** tenLog;
    else if (x > 1 * 10 ** tenLog)
      x = 2 * 10 ** tenLog;
    else
      x = 1 * 10 ** tenLog;

    return x * sign;
  }

  function formatTwoDigits(values: number[], joinStr: string):
string {
    let outStr = "";
    for (let i = 0; i < values.length; i++) {
      if (i > 0)
        outStr += joinStr;
      let str = values[i].toString();
      while (str.length < 2)
        str = "0" + str;
      outStr += str;
    }
    return outStr;
  }

  export function generateValueChartSvg(values: Value[], width:
```

```
number, height: number,
  minValue?: number, maxValue?: number, useStairway = false):
string {

  // Ensure the array isn't empty.
  if (values.length == 0)
    throw new Error("No history values available.");

  let outStr = "";

  // Determine the maximum and minimum values.
  // Ensure the values are ordered by date.
  values.sort((a, b) => a.date - b.date);

  let minV: number | null = null, maxV: number | null = null;
  let minD = values[0].date;
  let maxD = values[values.length - 1].date;
  for (let n of values) {
    minV = minV === null ? n.value : Math.min(minV, n.value);
    maxV = maxV === null ? n.value : Math.max(maxV, n.value);
  }
  if (minV == null || maxV == null)
    throw new Error("Could not determine min/max");

  // Ensure that if all values are the same we don't get
  Infinity/NaN.
  if (maxV - minV < 0.00001 * minV)
    maxV = minV + 0.00001 * minV, minV = minV - 0.00001 *
minV;

  const padding = 30;
  let yTop = maxV + (maxV - minV) / (height - 2 * padding) *
padding; // 20 pixel padding
  let yBottom = minV - (maxV - minV) / (height - 2 * padding) *
padding;
  let xLeft = minD - (maxD - minD) / (width - 2 * padding) *
padding;
  let xRight = maxD + (maxD - minD) / (width - 2 * padding) *
padding;

  let convCoords = (coords: DPoint): DPoint =>
    [(coords[0] - xLeft) / (xRight - xLeft) * width,
      (yTop - coords[1]) / (yTop - yBottom) * height];

  // Create the svg and draw the points.
  outStr += `<svg xmlns="http://www.w3.org/2000/svg"
version="1.1" baseProfile="full" width="${width}px"
height="${height}px" viewBox="0 0 ${width} ${height}">`;

  let convMax = maxValue == null ? null : convCoords([0,
```



```

maxValue]);
    let convMin = minValue == null ? null : convCoords([0,
minValue]);
    // Draw rects for the min and max values.
    if (convMax != null && convMax[1] >= 0)
        outStr += `<rect x="0" y="0" width="${width}"
height="${pform(convMax[1])}" fill="#fcdada"/>`;
    if (convMin != null && convMin[1] < height)
        outStr += `<rect x="0" y="${pform(convMin[1])}"
width="${width}" height="${pform(height - convMin[1])}"
fill="#fcdada"/>`;

    // Draw a line for the x and y axis. We simply draw it at the
bottom / left.
    let conv1 = convCoords([minD, minV]);
    let conv2 = convCoords([maxD, minV]);
    outStr += `<line x1="${escapeXml(pform(conv1[0]))}"
y1="${escapeXml(pform(conv1[1]))}" x2="${escapeXml(pform(conv2[0]))}"
y2="${escapeXml(pform(conv2[1]))}" stroke="grey" stroke-width="1"/>`;
    conv1 = convCoords([minD, maxV]);
    conv2 = convCoords([minD, minV]);
    outStr += `<line x1="${escapeXml(pform(conv1[0]))}"
y1="${escapeXml(pform(conv1[1]))}" x2="${escapeXml(pform(conv2[0]))}"
y2="${escapeXml(pform(conv2[1]))}" stroke="grey" stroke-width="1"/>`;

    // Now draw some small lines which indicates the continuous x
and y values.
    // We initially use 25 pixel then get the smallest number of
1*10^n, 2*10^n or 5*10^n that is >= our number.
    const fontSize = 12;
    let xLineGap = determineGap(40 / (width - 2 * padding) *
(xRight - xLeft));
    let xStart = Math.floor(minD / xLineGap + 1) * xLineGap;
    let yLineGap = determineGap(20 / (height - 2 * padding) *
(yTop - yBottom));
    let yStart = Math.floor(minV / yLineGap + 1) * yLineGap;
    for (let x = xStart; x <= maxD; x += xLineGap) {
        let conv1 = convCoords([x, minV]);
        let conv2 = [conv1[0], conv1[1] - 10];
        outStr += `<line x1="${escapeXml(pform(conv1[0]))}"
y1="${escapeXml(pform(conv1[1]))}" x2="${escapeXml(pform(conv2[0]))}"
y2="${escapeXml(pform(conv2[1]))}" stroke="grey" stroke-width="1"/>`;
        let xDate = new Date(x);
        let textContent1 = formatTwoDigits([xDate.getDate(),
xDate.getMonth() + 1, xDate.getFullYear() % 100], ".");
        let textContent2 = formatTwoDigits([xDate.getHours(),
xDate.getMinutes(), xDate.getSeconds()], ":");
        outStr += `<text x="${escapeXml(pform(conv1[0] -
textContent1.length * fontSize / 4))}" y="${escapeXml(pform(conv1[1] +
14))}" style="font-family: Consolas, monospace; font-size:
${fontSize}px;">${escapeXml(textContent1)}</text>`;
    }

```

```
        outStr += ``;
    }
    for (let y = yStart; y <= maxV; y += yLineGap) {
        let conv1 = convCoords([minD, y]);
        let conv2 = [conv1[0] + 10, conv1[1]];
        outStr += ``;
    }

    // Draw the points.
    let pointList = "";
    let prevPoint: DPoint | null = null;
    for (let i = 0; i < values.length; i++) {
        let convPoint = convCoords([values[i].date,
values[i].value]);
        if (useStairway && prevPoint !== null) {
            // Use the previous y coordinate with the current x
            coordinate.
            pointList +=
`${pform(convPoint[0])},${pform(prevPoint[1])} `;
        }
        pointList +=
`${pform(convPoint[0])},${pform(convPoint[1])} `;
        prevPoint = convPoint;
    }
    outStr += `
```

```

        // then display them in an SVG diagram.
        let historyValues = await
codabix.readNodeHistoryValuesAsync(gradientNode, null, null, 30);
        let svgValues: SvgLibrary.Value[] = [];
        for (let historyValue of historyValues) {
            if (typeof historyValue.value != "number")
                throw new Error("Expected a number value");

            svgValues.push({
                value: historyValue.value as number,
                date: historyValue.timestamp.getTime()
            });
        }

        // Generate the SVG diagram.
        let resultString: string;
        try {
            resultString = SvgLibrary.generateValueChartSvg(svgValues,
700, 400, undefined, 40);
            // This worked, so set the Content-Type to SVG.
            response.contentType = "image/svg+xml";
        }
        catch (e) {
            // Output the error.
            resultString = String(e);
            response.statusCode = 500;
            response.contentType = "text/plain";
        }

        await response.writeBodyCompleteAsync(resultString);
    });
}());

```

## Extended HTTP Programming

The previously shown examples generate an HTML page or an SVG image that displayed in the browser, but doesn't change afterwards. To create dynamic HTML5 apps, you could also create a static HTML page with a JavaScript/TypeScript, which communicates with a script at the Codabix side, e.g. using JSON (to serialize and deserialize JSON you can use `JSON.stringify()` and `JSON.parse()`). This way, the script in the browser can regularly request new information.

The next chapter describes WebSocket connections which your HTML page can use to create bidirectional, socket-like connections in order to communicate with a Codabix Script.

## WebSocket Connections in an HTTP Handler

In a Codabix Script you can also run **server-side WebSocket connections**, that are created e.g. in a browser script using the **WebSocket** class that is available there. WebSocket allows to send data in both directions (server to client and client to server) at any time as long as the connection is established, in contrast to regular HTTP requests where the client would need to

constantly send new HTTP requests in order to check if the server has any new data ("polling"). For example, in a Codabix Script you could add an event listener for the `ValueChanged` event to a node, and every time a new value is written to the node, send that new value using `WebSocket` to all currently connected browsers for displaying the value

In order to accept a `WebSocket` connection, first check with `ctx.isWebSocketRequest` if the request is actually a `WebSocket` request. If yes, call `ctx.acceptWebSocketAsync()` which returns a `net.RawWebSocket` object. On this object you can call `receiveAsync()` to receive message, and call `sendAsync()` to send message. The `Promise` object returned by these methods is fulfilled once the send/receive operation has completed. In case an error occurs when sending or receiving (e.g. when the connection has already been closed), the methods throw an exception (or reject the returned `Promise`).

**Note:** On a specific `RawWebSocket` object you can send and receive messages at the same time (meaning both a `receiveAsync()` and a `sendAsync()` operation can be outstanding).

For the following `WebSocket` examples, we use a static HTML page that establishes a `WebSocket` connection to Codabix. Therefore, please first copy the following file `CodabixWebsocket.html` into the `webfiles` folder in your project directory:

#### [CodabixWebsocket.html](#)

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>CoDaBix WebSocket Example</title>
  <style>
    #connect-container {
      float: left;
      width: 400px
    }

    #connect-container div {
      padding: 5px;
    }

    #console-container {
      float: left;
      margin-left: 15px;
      width: 400px;
    }

    #console {
      border: 1px solid #CCCCCC;
      border-right-color: #999999;
      border-bottom-color: #999999;
      height: 170px;
      overflow-y: scroll;
      padding: 5px;
    }
  </style>
</head>
<body>
  <div id="connect-container">
    <div>
      <input type="text" value="ws://localhost:8080"/>
      <input type="button" value="Connect" />
    </div>
  </div>
  <div id="console-container">
    <div>
      <div id="console"></div>
    </div>
  </div>
</body>
</html>
```

```
        width: 100%;
        white-space: pre-wrap;
    }

    #console p {
        padding: 0;
        margin: 0;
    }
</style>
<script>
"use strict";
document.addEventListener("DOMContentLoaded", function() {

    var ws = null;

    var connectButton = document.getElementById('connect');
    var disconnectButton = document.getElementById('disconnect');
    var echoButton = document.getElementById('echo');
    var messageBox = document.getElementById('message');
    var consoleArea = document.getElementById('console');

    function setConnected(connected, fullyConnected) {
        connectButton.disabled = connected;
        disconnectButton.disabled = !connected;
        echoButton.disabled = !(connected && fullyConnected);
    }
    setConnected(false);

    function connect() {
        if (ws != null)
            return;

        if (typeof WebSocket == "undefined")
            alert("WebSocket is not supported by this browser.");

        var wsUrl = (window.location.protocol == "https:" ? "wss:"
: "ws:") + '//' +
            window.location.host + "/scripthandlers/websocket";
        var localWs = ws = new WebSocket(wsUrl);
        setConnected(true);
        log("Info: WebSocket connecting...");

        ws.onopen = function () {
            if (localWs != ws)
                return;

            setConnected(true, true);
            log("Info: WebSocket connection opened.");
        };
        ws.onmessage = function (event) {
            if (localWs != ws)
```

```
        return;

        log("Received: " + event.data);
    };
    ws.onclose = function (event) {
        if (localWs !== ws)
            return;

        setConnected(false);
        log("Info: WebSocket connection closed, Code: " +
event.code +
        (event.reason == "" ? "" : ", Reason: " +
event.reason));
        ws = null;
    };
}

function disconnect() {
    if (ws == null)
        return;

    setConnected(false);
    log("Info: WebSocket connection closed by user.");
    ws.close();
    ws = null;
}

function sendText() {
    if (ws == null)
        return;

    var message = messageBox.value;
    ws.send(message);
    log("Sent: " + message);
}

function log(message) {
    var p = document.createElement('p');
    p.style.wordWrap = 'break-word';
    p.appendChild(document.createTextNode(message));
    consoleArea.appendChild(p);
    while (consoleArea.childNodes.length > 25) {
        consoleArea.removeChild(consoleArea.firstChild);
    }
    consoleArea.scrollTop = consoleArea.scrollHeight;
}

// Add event listeners to the buttons.
connectButton.onclick = function() {
    connect();
};
```

```
};

disconnectButton.onclick = function() {
    disconnect();
};

echoButton.onclick = function() {
    sendText();
};

}, false);
</script>
</head>
<body>
<div>
    <div id="connect-container">
        <div>
            <button id="connect">Connect</button>
            <button id="disconnect">Disconnect</button>
        </div>
        <div>
            <textarea id="message" style="width: 350px">Hello
world!</textarea>
        </div>
        <div>
            <button id="echo">Send Message</button>
        </div>
    </div>
    <div id="console-container">
        <div id="console"></div>
    </div>
</div>
</body>
</html>
```

You should now be able to open this page when opening

<http://localhost:8181/webfiles/CodabixWebsocket.html> in your browser (provided that the Local HTTP Port for the Codabix web server in the Codabix Project Settings is set to the default value of 8181 and the option “Server Static Web Files” has not been disabled):



## Simple Echo WebSocket

In the simplest case you can receive messages from a WebSocket, and send them back directly to the client, as shown in the following Codabix Script:

```
runtime.handleAsync(async function () {

    const maxMessageLength = 10000;

    net.registerHttpRequest("websocket", async ctx => {
        // Check if the client sent a WebSocket request.
        if (!ctx.isWebSocketRequest) {
            ctx.response.statusCode = 400;
        }
        else {
            // Accept the WebSocket request.
            let ws = await ctx.acceptWebSocketAsync();

            // In a loop, receive messages from the client and send
            // them back.
            // Once the client closes the connection, receive() will
            // return null
            // or throw an exception.
            while (true) {
                let message = await ws.receiveAsync(maxMessageLength);
                if (message == null || message.length ==
maxMessageLength)
                    break; // connection closed or message too large

                logger.log("Received message: " + message);

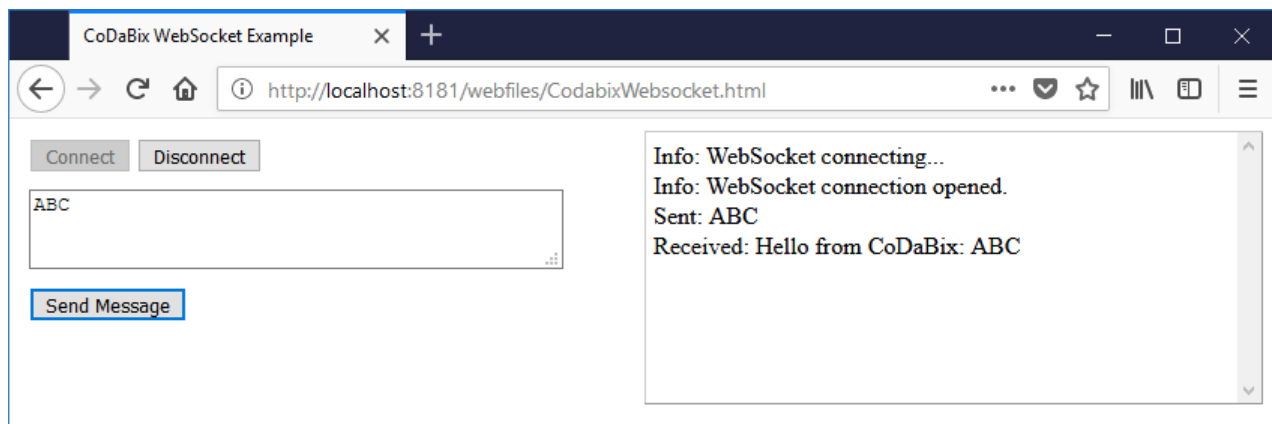
                // Send the message back to the client.
                await ws.sendAsync("Hello from CoDaBix: " + message);
            }
        }
    });
});

}());
```



The HTTP handler accepts a WebSocket connection, and then reads messages from the client and sends them back, until the client closes the connection.

In the HTML page, you can establish a connection with the "Connect" button and send messages to Codabix, which should be sent back immediately:



### "Chat" with Background Send Operations

Imagine we want to use WebSocket to send the current value of the Temperature demo node to all connected users as soon as a new value is written in Codabix. Additionally, users should be able to chat with one another.

Because the `sendAsync()` method (asynchronously) blocks until the data has been fully sent, we cannot just call it every time we want to send a new temperature value or a chat message, because the previous message might not yet have been sent completely to the client. Instead, we use a **queue** where the messages to be sent are put. Another script function then removes a message from the queue and sends them using the WebSocket. Once the send operation is completed, it continues with removing the next message from the queue, or waits until at least one new message is in the queue.

In the following script code, this is done by the class `WebSocketHandler` which you can also reuse for other scripts.

(In case you have run the previously shown Echo WebSocket script, please disable it, as both scripts would register the same HTTP path.)

```
runtime.handleAsync(async function () {

    /**
     * A utility class that implements a Send Queue for WebSockets.
     This allows you to
     * push messages to be sent to the client to the queue without
     having to wait until
     * the client has received it.
     */
    class WebSocketSender {
        // The queued messages to be sent to the client.
        private sendQueue: string[] = [];
        // The callback to resolve the promise of the sender function.
        private sendResolver: (() => void) | null = null;
```

```
private senderIsExited = false;

constructor(private ws: net.RawWebSocket) {
    // Start the sender function which will then wait for the
    next message queue entry.
    runtime.handleAsync(this.runSenderAsync());
}

/**
 * Adds the specified message to the send queue.
 */
public send(s: string) {
    // If the sender already exited due to an error, do
    nothing.
    if (this.senderIsExited)
        return;

    // TODO: Check if the number of buffered messages exceeds
    a specific limit.
    this.sendQueue.push(s);

    // If the sender waits for the next queue entry, release
    the sender.
    if (this.sendResolver) {
        this.sendResolver();
        this.sendResolver = null;
    }
}

private async runSenderAsync() {
    try {
        // In a loop, we will wait until the next send queue
        entry arrives.
        while (true) {
            for (let i = 0; i < this.sendQueue.length; i++) {
                await this.ws.sendAsync(this.sendQueue[i]);
            }
            this.sendQueue = [];

            // Create a Promise and cache the resolve handler,
            so that the
            // send() method can fulfill the Promise later.
            await new Promise<void>(resolve =>
this.sendResolver = resolve);
        }
    } catch (e) {
        // An error occurred, e.g. when the client closed the
        connection.
        // This means the receive handler should also get an
```

```
exception when it
    // tries to receive the next message from the client.
    logger.log("Send Error: " + e);
}
finally {
    this.senderIsExited = true;
}
}
}

// The maximum message size which we will receive.
const maxReceiveLength = 10000;

interface ChatUser {
    name: string;
    sender: WebSocketSender;
}

// The set of currently connected users.
const chatUsers = new Set<ChatUser>();
let currentUserId = 0;

// Register for the ValueChanged event of the "Temperature" node.
Every time a
// new value occurs, we will broadcast it to all connected users.
const temperatureNode = codabix.findNode("/Nodes/Demo-
Nodes/Temperature", true);

const getTemperatureValueMessage = function (value:
codabix.NodeValue | null) {
    return `Temperature: ${value && value.value}
${temperatureNode.unit || ""}`;
};

temperatureNode.addValueChangedEventListener(e => {
    // Send the new value to all connected users.
    const message = getTemperatureValueMessage(e.newValue);
    chatUsers.forEach(u => {
        u.sender.send(message);
    });
});

// Register the websocket handler.
net.registerHttpRequest("websocket", async ctx => {
    // Check if the client sent a WebSocket request.
    if (!ctx.isWebSocketRequest) {
        ctx.response.statusCode = 400;
    }
}
```

```
    else {
      // Accept the WebSocket request.
      let ws = await ctx.acceptWebSocketAsync();
      await handleWebSocketAsync(ws);
    }
  });

  // The function that handles the WebSocket connection.
  const handleWebSocketAsync = async function (ws: net.RawWebSocket)
  {
    // Create a send queue which we use to send messages to the
    client.
    let sender = new WebSocketSender(ws);

    // Generate a new user name, then notify the existing users
    // that a new user joined.
    let user = {
      name: "User-" + ++currentUserId,
      sender: sender
    };
    chatUsers.add(user);

    try {
      const userJoinedMessage = `${user.name} joined.`;
      chatUsers.forEach(u => {
        u.sender.send(userJoinedMessage);
      });

      // Send the initial temperature value to the new user.
      user.sender.send(getTemperatureValueMessage(temperatureNode.value));

      // Now start to receive the messages from the client.
      while (true) {
        let receivedMessage: string | null;
        try {
          receivedMessage = await
ws.receiveAsync(maxReceiveLength);
          if (receivedMessage == null ||
receivedMessage.length >= maxReceiveLength)
            break; // connection closed or message too
large

        }
        catch (e) {
          logger.log("Receive Error: " + e);
          break;
        }

        // Broadcast the received message.
        const broadcastMessage = `${user.name}: ` +
receivedMessage;
```

```
chatUsers.forEach(u => {  
    u.sender.send(broadcastMessage);  
});  
}  
}  
finally {  
    // The client has closed the connection, or there was an  
    // error when receiving.  
    // Therefore, we notify the other users that the current  
    // user has left.  
    chatUsers.delete(user);  
  
    const userLeftMessage = `${user.name} left.`;  
    chatUsers.forEach(u => {  
        u.sender.send(userLeftMessage);  
    });  
}  
}  
  
}());
```

Once a user connects, it is assigned a user name (e.g. User-6) and a message is sent to all connected users. The new user also receives the current value of the temperature node. As soon as a new value is written to the temperature node or one of the users sends a chat message, it is sent to all connected users (practically a broadcast):

The top screenshot shows a browser window titled "CoDaBix WebSocket Example" at the URL "http://localhost:8181/webfiles/CodabixWebsocket.html". The interface includes a "Connect" button, a "Disconnect" button, a text input field containing "Hi, new User!", and a "Send Message" button. Below the input field, the text "User-6" is displayed. To the right, a scrollable log shows the following messages: "Info: WebSocket connection opened.", "Received: User-6 joined.", "Received: Temperature: -14 °C", "Received: Temperature: 62 °C", "Received: Temperature: 48 °C", "Received: User-7 joined.", "Sent: Hi, new User!", "Received: User-6: Hi, new User!", and "Received: Temperature: 32 °C".

The bottom screenshot shows the same browser window but for "User-7". The "Connect" button is now disabled, and the "Disconnect" button is enabled. The text input field is empty, and the "Send Message" button is disabled. Below the input field, the text "User-7" is displayed. The log on the right shows: "Info: WebSocket connecting...", "Info: WebSocket connection opened.", "Received: User-7 joined.", "Received: Temperature: 48 °C", "Received: User-6: Hi, new User!", and "Received: Temperature: 32 °C".

## Sending HTTP Requests

The namespace `net.httpClient` provides methods that allow you to send a HTTP request to

another server. This allows you e.g. to call external REST APIs using JSON objects.

**Important:** When sending requests over the internet (or other potentially insecure networks), please make sure that you always use `https`: URLs if possible, as `http`: URLs use an insecure connection and therefore do not provide server authenticity and data confidentiality/integrity. This is especially important when sending confidential login credentials.

**Note:** Currently, the request and response bodies can only use text data, not binary data.

### Simple GET requests

The following code issues a simple GET request and logs the response body (if present):

```
let response = await net.httpClient.sendAsync({
  url: "https://www.codabix.com/en/start"
});

if (response.content) {
  logger.log("Result: " + response.content.body);
}
```

### Accessing a JSON-based REST API

When you want to access an URL that can return a JSON result, you can use `JSON.parse()` to convert the JSON response body string into a JavaScript object, and access it.

The following example accesses an external REST API to get weather data (temperature) in a regular interval, and stores it in the `/Nodes/Temperature` node:

```
let temperatureNode = codabix.findNode("/Nodes/Temperature",
true);

while (true) {
  let valueToWrite: codabix.NodeValue;
  try {
    let response = await net.httpClient.sendAsync({
      url:
"https://api.openmeteo.com/observations/openmeteo/1001/t2"
    });

    let result = JSON.parse(response.content!.body);
    valueToWrite = new codabix.NodeValue(result[1]);
  }
  catch (e) {
    logger.logWarning("Could not retrieve weather data: " +
e);

    valueToWrite = new codabix.NodeValue(null, undefined, {
      statusCode: codabix.NodeValueStatusCodeEnum.Bad,
      statusText: String(e)
    });
  }
}
```

```
    }

    // Write the temperature value.
    await codabix.writeNodeValueAsync(temperatureNode,
valueToWrite);

    // Wait 5 seconds
    await timer.delayAsync(5000);
}
```

The following code demonstrates how to send a POST request to the CoDaBix-internal REST API (for <encrypted-password> you will need to specify the encrypted user password):

```
let result = await net.httpClient.sendAsync({
    // Note: For external servers you should "https:" for a secure
    connection.
    url: "http://localhost:8181/api/json",
    method: "POST",
    content: {
        headers: {
            "content-type": "application/json"
        },
        body: JSON.stringify({
            username: "demo@user.org",
            password:
codabix.security.decryptPassword("<encrypted-password>"),
            browse: {
                na: "/Nodes"
            }
        })
    }
});

let jsonResult = JSON.parse(result.content!.body);
// TODO: Process JSON result...
```

## Recommended Best Practices

The following list gives some recommendations for performant and clean scripts:

- If you are not writing a Library Script, place your whole code inside an IIFE (immediately-invoked function expression) as follows:

[Blank Template.js](#)

```
runtime.handleAsync(async function () {
```

```
// Your code here...  
  
} ());
```

This avoids polluting the global scope and the TypeScript compiler can detect unused local variables and does not complain about using anonymous types in exported variables.

We recommend to mark the function with the `async` attribute so that you are able to call asynchronous functions using the `await` keyword. In that case you should pass the returned `Promise` object to `runtime.handleAsync()` to ensure uncaught exceptions are not silently swallowed, as shown in the code example above.

- Always use `let` or `const` to declare variables, not `var` as the latter is not block-scoped but function-scoped.
- Instead of using `arguments` which is a “magic”, array-like object, use rest parameters (`...args`) which is a real `Array` instance and doesn't contain previous function parameters.
- When enumerating an array, don't use `for-in` - use `for-of` instead, as the former doesn't guarantee any enumeration order.
- When throwing an exception, don't throw a primitive value like `throw "My Error Message"`; . Instead, create and throw `Error` object instances, like `throw new Error("My Error Message")`; . This ensures that you can retrieve the stack trace from where the exception occurred.
- If possible, avoid creating closures in code that is called very often (e.g. in a loop), because creating a closure is expensive in JavaScript. Instead, check if you can reuse a function by passing arguments.

For example, when you want to handle a node's `ValueChanged` event and write a new value in it (for which `codabix.scheduleCallback()` must be used), avoid the following code that creates a closure every time the event is invoked:

```
myNode.addValueChangedEventListener(e =>  
  codabix.scheduleCallback(() => {  
    void codabix.writeNodeValueAsync(otherNode, e.newValue);  
  }));
```

Instead, you can create a closure once and then using the following variant of `codabix.scheduleCallback()` in the event listener that allows to pass arguments:

```
const scheduledHandler = (newVal: codabix.NodeValue) => {  
  void codabix.writeNodeValueAsync(otherNode, newVal);  
};  
myNode.addValueChangedEventListener(e =>  
  codabix.scheduleCallback(scheduledHandler, e.newValue));
```



## Storing Passwords Securely

In some cases, you need to store passwords in script code, for example to access external password-protected HTTP services. To avoid having to store a password in plaintext, you can first encrypt it once using the CoDaBix Web Configuration by opening the menu item “Password Security”, and then at the start of your script code you can decrypt it by calling `codabix.security.decryptPassword()`. This means that the password is **encrypted** with the **password key** of the back-end database which was generated using a cryptographically secure random number generator when creating the database. The same mechanism is also used when storing passwords in datapoint nodes of type “Password”, which e.g. is used by device plugins.

This provides additional protection:

- Passwords are not displayed in clear text, so you can show Script code to other people, without allowing them to directly read the password.
- If you disable the option *Include Password Key* when creating a backup, such passwords cannot be extracted (decrypted) from the backup.

**Note:** You can generate a new password key in the CoDaBix Settings, which means passwords that were encrypted with the previous key are no longer readable.

For example, if you wanted to use the password “test” in your Script code, open the menu item “Password Security” in the CoDaBix Web Configuration, and then enter the password. After clicking on “Encrypt”, the encrypted password will be printed.

Example:

Password:

Encrypted Password: `FNjeFt5k85PbW5ly1q2h/Ctf7RqTW2JToUTNo/cwWFrL7oOUXJBewshFutKLosDjh6C8pGGgEa6m4MZqBF1RiQ==#`

You can now copy the encrypted password and store it in the Script code where you need it:

```
// Decrypt the password that we need to run a HTTP request.
const myPassword = codabix.security.decryptPassword(
  "K8D/VWnVQG45HSF1D1G94RwXzrSxH3ARlzzhHekoAdf+prcwe5y6S4FhPUug1Ycw#" );

// ...

function doRequest() {
  let request = {
    user: "myUser",
    password: myPassword
  }
  // TODO: Send the request...
}
```

From:  
<https://www.codabix.com/> - **CoDaBix®**

Permanent link:  
<https://www.codabix.com/en/plugins/interface/scriptinterfaceplugin/scriptinterface.development.guide>

Last update: **2021/07/30 13:41**