Critical manufacturing
an ASM PT company

Writing a Code Task

12.0.1

November 2026

DOCUMENT ACCESS

Public

DISCLAIMER

The contents of this document are under copyright of Critical Manufacturing S.A. it is released on condition that it shall not be copied in whole, in part or otherwise reproduced (whether by photographic, or any other method) and the contents therefore shall not be divulged to any person other than that of the addressee (save to other authorized offices of his organization having need to know such contents, for the purpose for which disclosure is made) without prior written consent of submitting company.

12.0

Copy Page

Connect IoT - Writing a Code Task Tutorial

Overview

The `Code` task runs a TypeScript class inside the controller, in a sandbox, each time it is activated. Every other task declares its ports and settings in its package, so the designer and a reader of the workflow know what it does. A `Code` task has the ports you add, does whatever its code says, and shows its failures only when it runs. Reach for it last, after the tasks and converters that already do the step:

To	Use first
Compute a value from named inputs with a formula	Generic Expression Evaluator
Pick values out of a JSON object or reshape it	JSONata
Change one value on its way along a Data Flow link	A converter. See Converters.
Compute the value of one Control Flow row	An expression. See Control Flow Expressions.
Branch, loop, or retry	The Control Flow tasks. See Connect IoT - Control Flow in Depth Tutorial.
Read from or act on the equipment	`On Equipment Event`, `Execute Equipment Command`, and the other equipment tasks

Table 1: Tasks to use before code

A `Code` task fits one small algorithm over the values that arrived on its inputs, such as parsing, grouping, validating, or assembling a nested object. The workflow keeps the wiring: what reacts to what, in which order, and every effect on the equipment.

This tutorial is for integrators who completed Connect IoT - How Values Travel Tutorial and Connect IoT - Control Flow in Depth Tutorial and can read TypeScript. You build a Data Flow page that reads each lot file, passes its text to a `Code` task, and logs a summary such as `lot-a.txt: 2 lots [LOT-01, LOT-02], 2 since start`. You then test the code with files that break its assumptions.

> **Info**
>
> For the settings, the class contract, and the services of the task, see Custom Code.

Before You Start

- Set up the controller, the watched folder `C:/Lots/Incoming`, and the files `lot-a.txt` and `lot-b.txt` as in Connect IoT - Control Flow in Depth Tutorial.
- Make sure that no other page of the controller reacts to `OnNewFile`, so that no other workflow moves a file before this one reads it. Delete the `File Handling` page of that tutorial, or set up a second controller the same way.
- Create three more test files outside the watched folder:

File	Content, one value per line	What it tests
`lot-c.txt`	`LOT-05`, `LOT-06`	A file without `END`
`lot-d.txt`	Nothing	An empty file
`lot-e.txt`	`LOT-07`, `Lot 8`, `END`	A line that is not a lot

Table 2: Test files for the edge cases

- Copy the files into the watched folder at each checkpoint, and remove them afterward: every file left in the folder is reported again when the connection starts.

Sequence of Steps

Step 1: Wire a First Code Task and Run It

Add a workflow page, select **Data Flow**, name the page `Lot Summary`, and add the following tasks:

Task	Settings to set	Inputs/Outputs to link
`On Equipment Event`	**On Equipment Event** `OnNewFile`	-
`Execute Equipment Command`	**Equipment Command** `ReadFile`	`path` from the event `FullPath`; `Activate` from the event `Success`
`Code`	On the **Inputs** tab, `fileName` and `content`, both of **Type** `String`. On the **Outputs** tab, `summary` (`String`) and `count` (`Integer`).	`fileName` from the event `FileName`; `content` from `ReadFile` `reply`, with the `Any To String` converter; `Activate` from `ReadFile` `Success`
`Log Message`	**Mode** `RawText`	`message` from `Code` `summary`; `Activate` from `Code` `Success`

Table 3: Tasks of the Lot Summary page

1. On the **Code** tab of the `Code` task, keep the class of the template and replace the body of `main`:

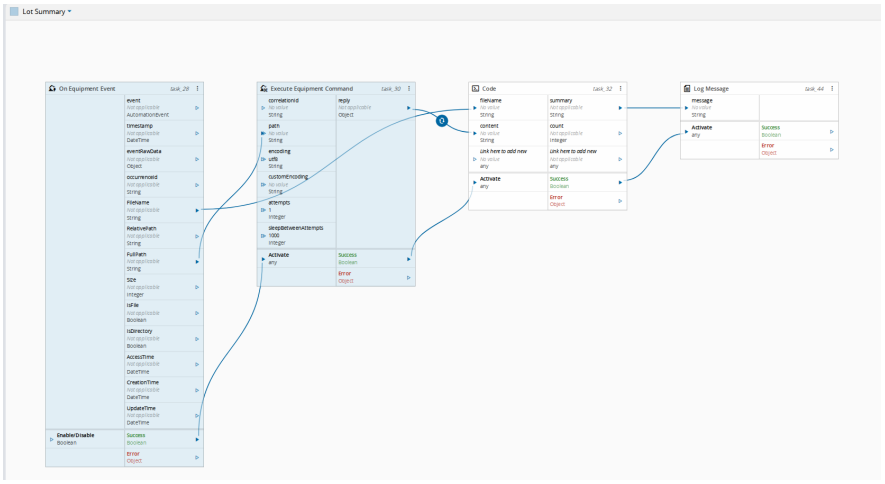
```
import { Framework } from 'framework';

export default class {

    /** Allows accessing external functions */
    private framework: Framework;

    constructor(framework: Framework) {
        this.framework = framework;
    }

    public async main(inputs: any, outputs: any): Promise<any> {
        const text: string = String(inputs.content ?? "");
        return { summary: `${inputs.fileName}: ${text.length} characters`
};
    }
}
```



2. Save the **Automation Controller**, then copy `lot-a.txt` into `C:/Lots/Incoming`. The console shows `lot-a.txt: <length> characters`.

The class is the default export, its constructor receives `framework`, and `main(inputs, outputs)` is the entry point, whose signature you keep. Inside the code, use the names you declared: `inputs.content`, and the key `summary` in the returned object. Outside, the task stores them as the ports `contentIn` and `summaryOut`. Each key of the returned object that names an output is emitted, then `Success`. A key that names no output is dropped, and an output that the object does not name is not emitted. `Log Message` writes only when its `Activate` input receives a value, so the `message` link alone logs nothing.

`fileName` reaches the `Code` task as soon as the event fires, and `content` arrives after the read. Both belong to the execution context of the same file, so they meet in one run. Values that come from two different triggers belong to two execution contexts and never meet in one run.

> **Note**
>
> In Control Flow, the `Code` block takes its inputs as rows, and the blocks after it read its outputs through expressions. Type a row `String` for text: an `Object` row parses text as JSON and fails.

Step 2: Parse the Lines and Reject Bad Data

Replace the body of `main` with the parser. It reads lines up to `END`, skips blank lines, and throws an error for a line that is not a lot:

```
public async main(inputs: any, outputs: any): Promise<any> {
    const fileName: string = inputs.fileName ?? "unknown file";
    const lots: string[] = [];
    for (const line of String(inputs.content ?? "").split("\n")) {
        const value: string = line.trim();
        if (value === "END") {
            break;
        }
        if (value === "") {
            continue;
        }
        if (!/^LOT-\d+$/.test(value)) {
            throw new Error(`Unexpected line '${value}' in ${fileName}`);
        }
        lots.push(value);
    }
    return {
        summary: `${fileName}: ${lots.length} lots [${lots.join(", ")}]`,
        count: lots.length
    };
}
```

1. Save, then copy `lot-a.txt` and `lot-e.txt`.

   The console shows `lot-a.txt: 2 lots [LOT-01, LOT-02]`. For `lot-e.txt`, it shows `Failed to execute code with error 'Unexpected line 'Lot 8' in lot-e.txt'`, followed by the stack trace, and no summary.

When `main` throws, the task logs the reason, emits the error on its `Error` output, and does not emit `Success`, so the tasks activated by `Success` do not run. Link `Error` to the task that handles the failure, or let an `Unhandled Error` task report it. Throw an `Error` object, never a text: the task reports the message of the error, and a text has none.

Return the results as one object at the end. `outputs.summary.emit(value)` emits while the code runs, but values emitted on one output in quick succession, with no `await` between them, reach a linked Data Flow task once, with the last value. An output named in the returned object is emitted even when its value is `null`, so give each value a fallback, as `fileName` has.

Step 3: Keep a Running Count in the Data Store

A new instance of the class runs each time the task is activated, and nothing in the code is shared with other tasks, so a class property never reaches the next run. State that must outlive a run goes in the data store of the controller, the same store that the `Store` and `Retrieve` tasks use. Replace the `return` statement of `main` with these lines:

```
const total: number = (await this.framework.dataStore.retrieve("lotsSeen", 0))
+ lots.length;
await this.framework.dataStore.store("lotsSeen", total, "Temporary");
this.framework.logger.info(`${fileName}: ${lots.length} lots, ${total} since
start`);
return {
    summary: `${fileName}: ${lots.length} lots [${lots.join(", ")}], ${total}
since start`,
    count: lots.length
};
```

1. Save, then copy `lot-a.txt`, and then `lot-b.txt`.

   The console shows these messages, each after the timestamp and source of its log line. Each summary of `Log Message` is preceded by the line that the code logs itself:

   ```
   lot-a.txt: 2 lots, 2 since start
   lot-a.txt: 2 lots [LOT-01, LOT-02], 2 since start
   lot-b.txt: 8 lots, 10 since start
   lot-b.txt: 8 lots [LOT-01, LOT-02, LOT-03, LOT-04, LOT-05, LOT-06, LOT-07,
   LOT-08], 10 since start
   ```

`retrieve` returns the default, `0`, until a value is stored. `Temporary` keeps the value in memory, and `Persistent` writes it to disk so that it survives a restart. The `framework` object gives the code these services:

Member	Use
`logger`	`debug`, `info`, `warning`, and `error`, written to the controller log, as `Log Message` does
`dataStore`	`store`, `retrieve`, `delete`, `listKeys`, and the queue methods, shared with every workflow of the controller
`messageBus`	`publish`, and `sendRequest`, which waits for the reply
`system` and `LBOS`	`call` an MES service with an input object built from `LBOS`
`utils`	`convertValueToType`, `sleep`, `ExecuteWithRetry`, and `ExecuteWithSystemErrorRetry`
`driver`	Commands and properties of the driver, only in a workflow associated with a driver

Table 4: Services of the framework object

`ExecuteWithRetry(logger, attempts, sleepBetweenAttempts, code)` runs `code` again after any error, up to `attempts` times, waiting `sleepBetweenAttempts` milliseconds between attempts. `ExecuteWithSystemErrorRetry` takes the same arguments and retries only on MES errors that are known to be transient, such as a deadlock or an object changed by another user. Both return what `code` returns and throw the last error when no attempt succeeds. For example, to call MES with an input object `input`:

```
const output = await this.framework.utils.ExecuteWithSystemErrorRetry(
    this.framework.logger, 3, 1000, async () => await
this.framework.system.call(input));
```

Prefer MES tasks, such as `Execute Service Call`, to calls from code, and the equipment tasks to `framework.driver`: what the code sends is invisible to a reader of the workflow. See Connect IoT - Talking to MES Tutorial.

Step 4: Test the Code Before You Deploy It

1. Copy `lot-c.txt`, `lot-d.txt`, and `lot-e.txt`, one at a time, and compare the console with the expected result:

File	Expected result
`lot-c.txt`	`2 lots [LOT-05, LOT-06]`: the loop ends with the text.
`lot-d.txt`	`0 lots []`: the empty text gives no lot.
`lot-e.txt`	`Failed to execute code with error 'Unexpected line 'Lot 8' in lot-e.txt'`, no summary, and no change to the running total.

   Table 5: Expected results of the edge cases

2. On the **General** tab of the `Code` task, set **Context Expiration Time** to `1`, save, and copy `lot-a.txt`.

   The console shows the warning `Execution Context '<ID>' inputs have expired`, which lists `fileName`, and the summary starts with `unknown file`. `fileName` waited longer than one millisecond for the read to finish.

3. Set **Context Expiration Time** back to `10000`, the default, and save.

Context Expiration Time is how long, in milliseconds, the input values of an execution context wait for a new value or an activation before they are dropped. `0` keeps the input values until the activation. Set it longer than the slowest task between the first value and `Activate`.

Before you deploy the code to a production controller, also review what the sandbox does not guard:

- No module can be loaded: beyond the JavaScript built-ins, the code reaches only what `framework` offers, and `eval` is disabled.
- The task never interrupts running code. When **Execution Expiration Time** is set, a run that takes longer than that time fails with an error and its later outputs are dropped, but the code keeps running. A loop that does not end blocks the **Automation Controller**, so bound every loop.
- The code cannot subscribe to anything. It runs only when `Activate` receives a value, so react to events with trigger tasks.
- The controller decodes the stored code as ASCII. Write any other character in the code as an escape, such as `"\u00B0"` for the degree sign.
- A task with no code, or with code the controller cannot load, logs `No code defined` or `Failed to compile script` when the workflow starts, and every activation then fails with `No code defined`.

> **Info**
>
> To inspect the values of a run with breakpoints, see Debug Automation Controller. To run the same files as an automated test, use the Test Orchestrator File-Raw simulator ↗.

Validate the Result

1. Empty `C:/Lots/Incoming`, and note the running total of the last summary in the console.
2. Copy `lot-a.txt`, `lot-b.txt`, `lot-c.txt`, and `lot-d.txt`, one at a time. The console shows one summary per file, with `2`, `8`, `2`, and `0` lots, and the running total grows by `12`.
3. Copy `lot-e.txt`. The console shows the error of Step 2, and the running total does not change.

What You Learned

- Code comes last, after the tasks, converters, and expressions that already do the step.
- The class of the template is the contract: the default export receives `framework`, and `main` returns one object whose keys name the declared outputs.
- Inside the code you use the declared names, and the task stores its ports with the suffixes `In` and `Out`.
- State that outlives a run goes in the data store, and a failure is an `Error` thrown from `main`.
- **Context Expiration Time** decides how long inputs wait for `Activate`, and the sandbox does not stop a long run.

Next Steps

When the same code appears in a second `Code` task, keeps state of its own, needs to react to events, or is needed by a second controller, move it into an Automation Tasks Library, where it can have automatic tests. See Automation Tasks Library and the task customization guide. To recognize the designs that fail in production, continue to Workflow Anti-Patterns and Troubleshooting by Symptom ↗.

Legal Information

Disclaimer

The information contained in this document represents the current view of Critical Manufacturing on the issues discussed as of the date of publication. Because Critical Manufacturing must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Critical Manufacturing, and Critical Manufacturing cannot guarantee the accuracy of any information presented after the date of publication. This document is for informational purposes only.

Critical Manufacturing makes no warranties, express, implied or statutory, as to the information herein contained.

Confidentiality Notice

All materials and information included herein are being provided by Critical Manufacturing to its Customer solely for Customer internal use for its business purposes. Critical Manufacturing retains all rights, titles, interests in and copyrights to the materials and information herein. The materials and information contained herein constitute confidential information of Critical Manufacturing and the Customer must not disclose or transfer by any means any of these materials or information, whether total or partial, to any third party without the prior explicit consent by Critical Manufacturing.

Copyright Information

All title and copyrights in and to the Software (including but not limited to any source code, binaries, designs, specifications, models, documents, layouts, images, photographs, animations, video, audio, music, text incorporated into the Software), the accompanying printed materials, and any copies of the Software, and any trademarks or service marks of Critical Manufacturing are owned by Critical Manufacturing unless explicitly stated otherwise. All title and intellectual property rights in and to the content that may be accessed through use of the Software is the property of the respective content owner and is protected by applicable copyright or other intellectual property laws and treaties.

Trademark Information

Critical Manufacturing is a registered trademark of Critical Manufacturing.

All other trademarks are property of their respective owners.

Copyright © 2023 Critical Manufacturing. All rights reserved.