TF1200 Client Ui Extension

BECKHOFF

TF1200 Custom Extension Integration with Recipe Management

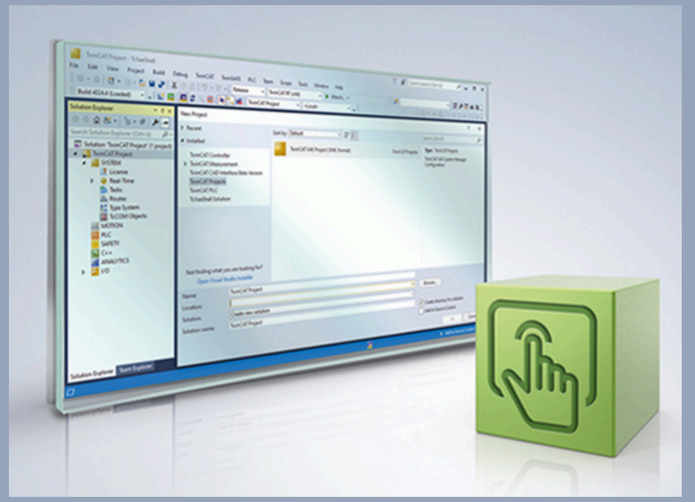
FrontEnd - BackEnd Communication and Real Time Recipe Logging

Mentor - Mr. Sidhant Bora

Presented by Prajwal Nivangune

TF1200 – TwinCAT 3 HMI UI CLient

- **Electron-based** desktop UI Client.
- Used to **Load** and T**est** TwinCAT HMI projects.
- Supports **custom extensions**.
- Enables **frontend - backend IPC communication.**
- Used for **advanced customization** beyond standard HMI.

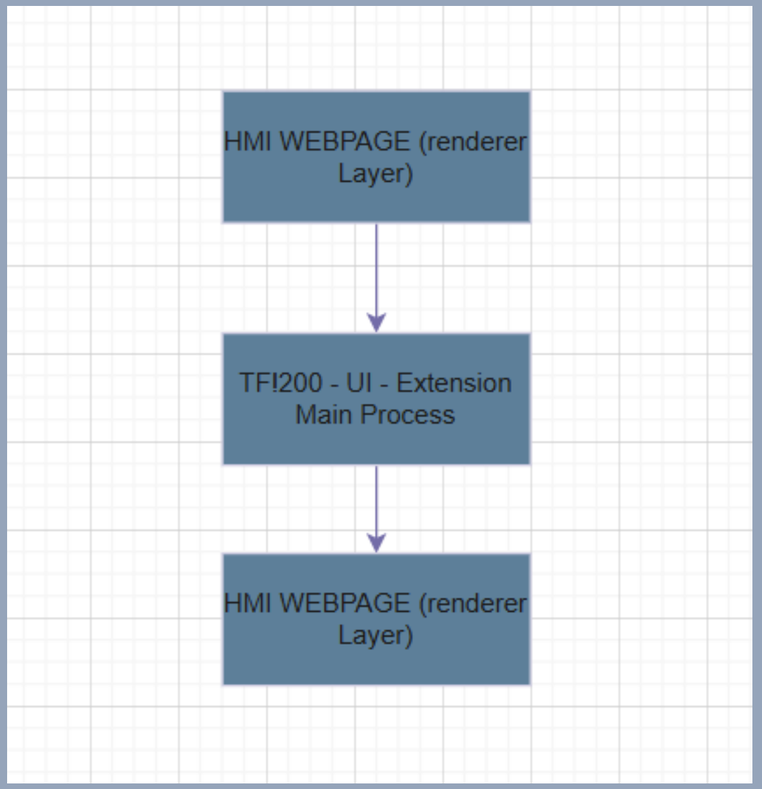


Why Do we use TF1200!

- To inject custom **JavaScript logic** and run **Node.js** backend processes.
- to enable f**ile system access**.
- To create **modular runtime plugins** and **professional industrial extensions.**

Project Objective

- Capture **“Teach Recipe”** and **“Activate Recipe”** Events.
- Extract selected recipe value.
- Send Value to Backend.
- Log action with timestamp in a File.
- Demonstrate real - time Integration with IPC

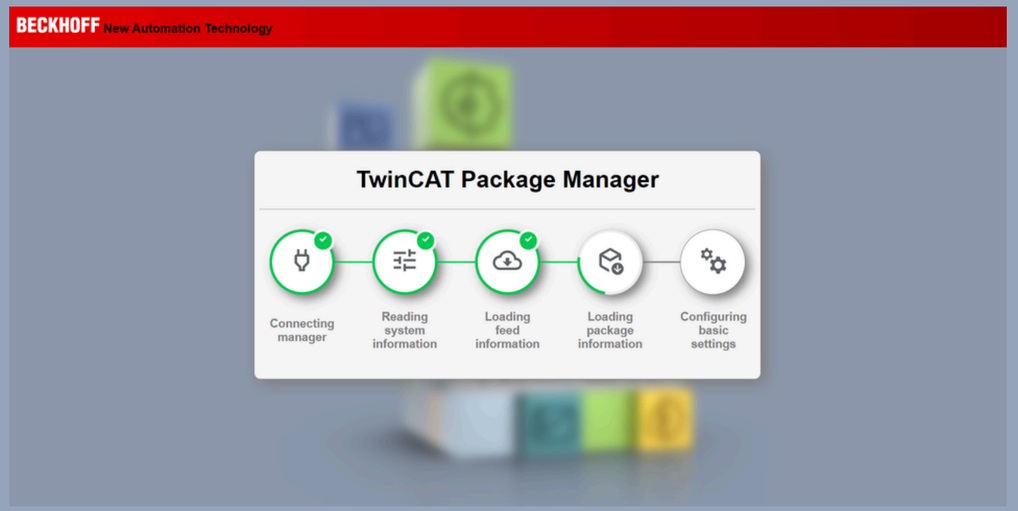


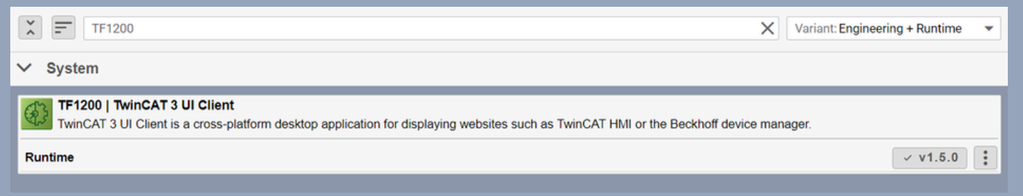
TF-1200 Installation And Setup

- Install **TF1200** using **TwinCAT Package Manager.**
- Locate the **config** File.

```
"C:\Users\<User>\AppData\Roaming\Beckhoff\TF1200-UI-Client\config.json"
```

- Publish the **HMI project**.
- Connect **TF1200** to **Published Runtime**.





Creating Custom Extension Folder

- Create folder in any local directory (e.g., C:\MyExtensions) which will be referenced in TF1200 config.json.

```
C:\MyExtensions\
```

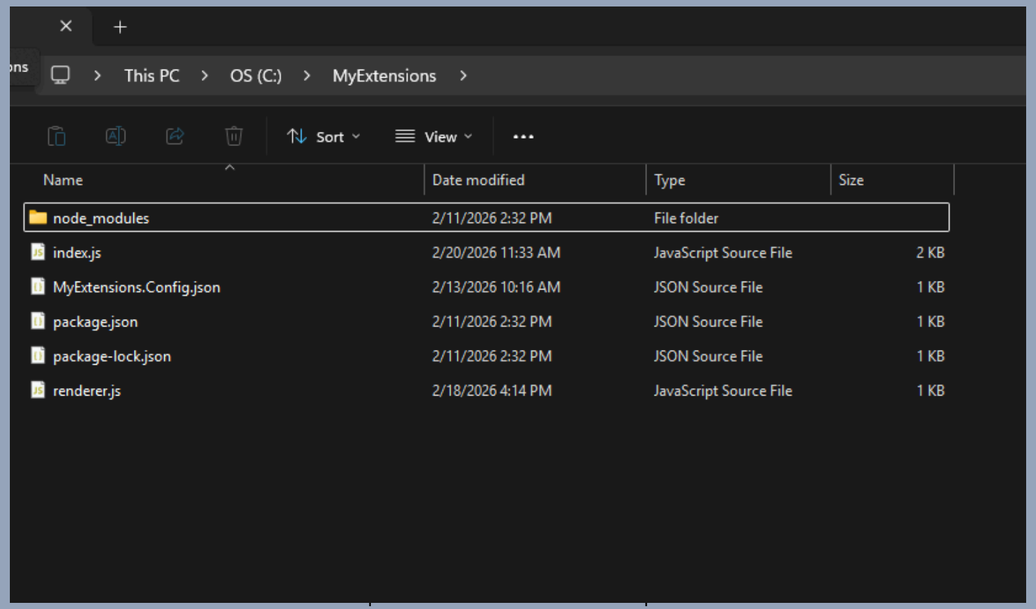
- Inside the Extensions Folder Create these Files:

1. **index.js** (Backend Logic)
2. **renderer.js** (Frontend Bridge)
3. **MyExtensions.Config.json**

- Install the **NPM Package**

```
npm install @beckhoff/tc-ui-client-ext
```

This will create the files **package.json** and **package-lock.json** as well as the **directory node_modules**.



This Folder Represents one complete extension

TF1200 Configuration

MyExtension.Config.json

Explains :

- name
- version
- entryPoint
- rendererScript

Registering Extension in TF1200

The Custom Created Extension is mentioned in the Tf1200 UI Client Config.json
here, “my-extension” = runtime namespace (if namespace in postMessage does not match config.json - IPC will Fail)
Used on IPC messages.

Setting Start URL

After Publishing HMI ;
The Setting URL is added so that the TF1200 loads real runtime instead of liveview.

{} MyExtensions.Config.json > ...

```
{
    "name": "MyExtensions",
    "version": "1.0.0",
    "entryPoint" : "index.js",
    "rendererScript" : "renderer.js"
}
```

MyExtension.Config.json

```
"extensions": {
    "my-extension": {
        "name": "MyExtensions",
        "directory": "C:/MyExtensions/",
        "version": "^1.0.0"
    }
},
```

Tf1200 Config.json

Code

```
http://<IPC-IP>:1010
```

Updated TF1200 config:

Code

```
"startUrl": "http://<IPC-IP>:1010"
```

Tf1200 Config.json

Front-End - renderer.JS

- TF1200 injects **renderer.js** into webpage.
- Available Object here is

```
tcuiclient
```

"**tcuiclient**" is the javascript bridge object injected by TF1200 into the loaded webpage. It allows the **frontend** (HMI page) to **communicate** with the **backend** (Node extension).

Main Functions of renderer.js :

- **postMessage()**
- **on()**
- **once()**

all these functions are used to send data to backend

```
function onExtensionReady() {
    /* code to be executed once the extension is ready */
}

tcuiclient.postMessage("System.extensions").then(extensions => {
    if (extensions[__extensionName] === "ready") {
        onExtensionReady();
    } else {
        tcuiclient.on(__extensionName + ".ready", onExtensionReady);
    }
});
```

renderer.js is injected by TF1200 into the loaded webpage and exposes the tcuiclient bridge object for IPC communication.

```
renderer.js > ...
1   tcuiclient.on("my-extension.teachResponse", function(data) {
2       console.log("Backend confirmed teaching:", data.recipe);
3   });
4
5
```

this is script that we use in our TF1200 Custom Extension Integration with Recipe Management Project to confirm the message from Backend

Teach Recipe Button Logic

Inside **RecipeManagement**:

Added JavaScript in onPressed:

- Get combobox control
- Extract selected value
- Send via IPC

Flow:

User clicks Teach

↓

Recipe name extracted

↓

Sent to backend

JavaScript

```javascript
if (typeof tcuiclient !== "undefined") {

    var selectedRecipe = TcHmi.Controls.get("TcHmiCombobox_RecipeList").getSelectedValue()

    tcuiclient.postMessage("my-extension.teachRecipe", {
        recipe: selectedRecipe || "Unknown"
    });

} else {
    console.log("tcuiclient not available");
}
```

This is the teachRecipe Logic where we have added the renderer function which will get the selectedValue and provide to the backend using the postMessage() which includes the command and the namespace.

BackEnd Logic – index.js

Backend :

- Extends the **TcUiClientExt** - a parent Class that Imports all the functions like emit (), onMessage (command, args).
- Handles onMessage(command, args).
- Uses switch-case.
- calls reusable logging function.

Core logic:

- Read recipe name
- Generate ISO timestamp
- Append to recipe-log.txt

Logging Mechanism :

- Node.js fs module
- path.join(__dirname, "recipe-log.txt")

Here __dirname is used to avoid hardcoded paths which ensure portability.

```
JS index.js > ...
1   const { TcUiClientExt } = require("@beckhoff/tc-ui-client-ext");
2   const fs = require("fs");
3   const path = require("path");
4
5   class MyExtensions extends TcUiClientExt {
6
7       constructor() {
8           super();
9           console.log("Recipe Extension Backend Started");
10      }
11
12      onMessage(command, args) {
13
14          if (command === "teachRecipe") {
15
16              const recipeName = args.recipe || "Unknown";
17              const timeStamp = new Date().toISOString();
18              const logMessage = `[${timeStamp}] Recipe "${recipeName}" was taught\n`;
19
20              console.log(logMessage);
21
22              const filePath = path.join("C:/MyExtensions/", "recipe-log.txt");
23
24              fs.appendFile(filePath, logMessage, (err) => {
25                  if (err) {
26                      console.error("Error writing log:", err);
27                  } else {
28                      console.log("Recipe log updated.");
29                  }
30              });
31
32              this.emit("teachResponse", {
33                  recipe: recipeName
34              });
35          }
36      }
37  }
38
39  module.exports = MyExtensions;
40
```

this is script that we use in our TF1200 Custom Extension Integration with Recipe Management Project

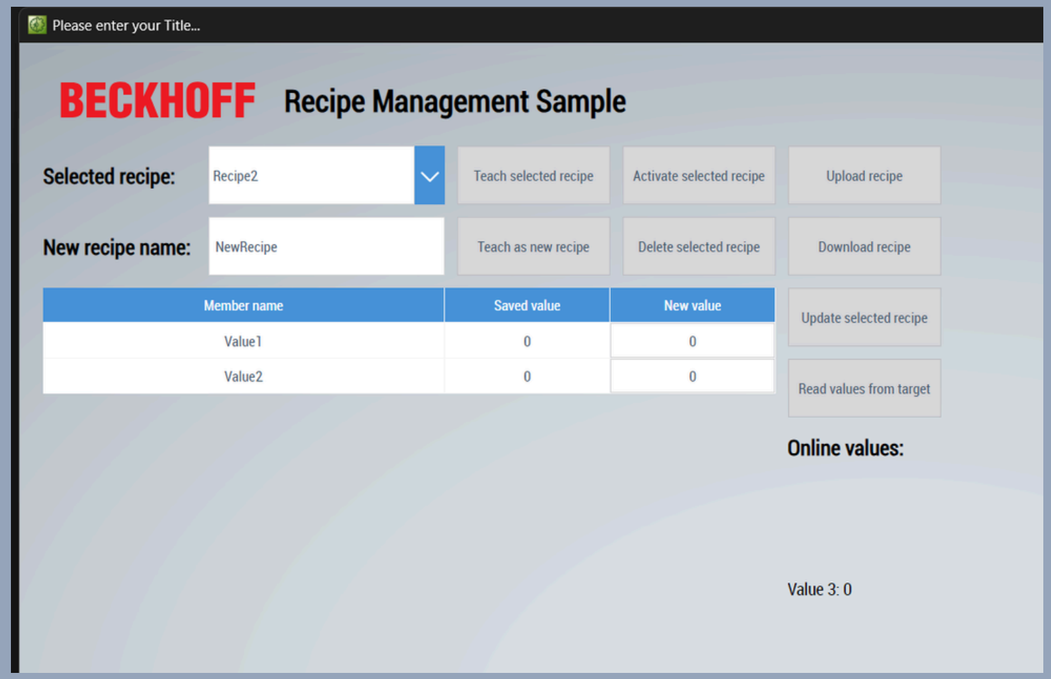
IPC Communication Flow

Step-by-step :

1. Button Pressed
2. Renderer Sends:

   my-extension.teachRecipe
3. Electron Splits :

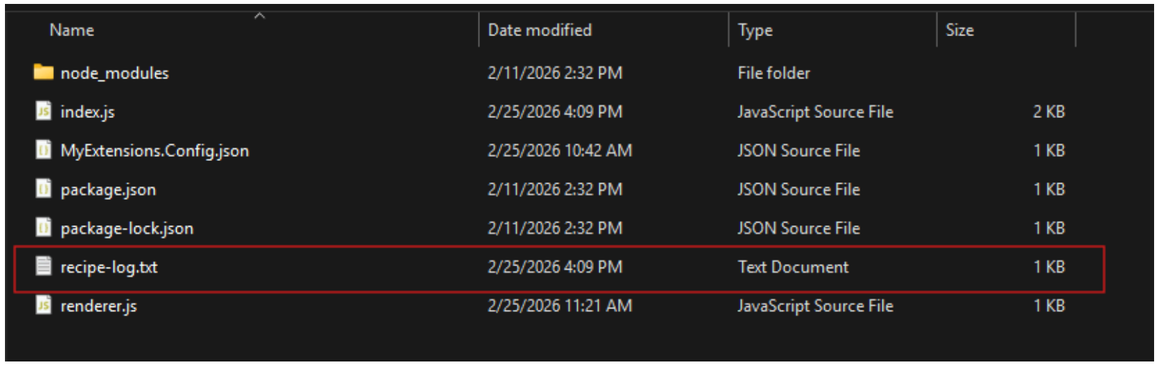
   namespace + command
4. Backend executes the logic
5. File written
6. Response emitted
7. The renderer receives confirmation



This is the HMI screen on the TF1200 UI Client Extension.

Output File Example

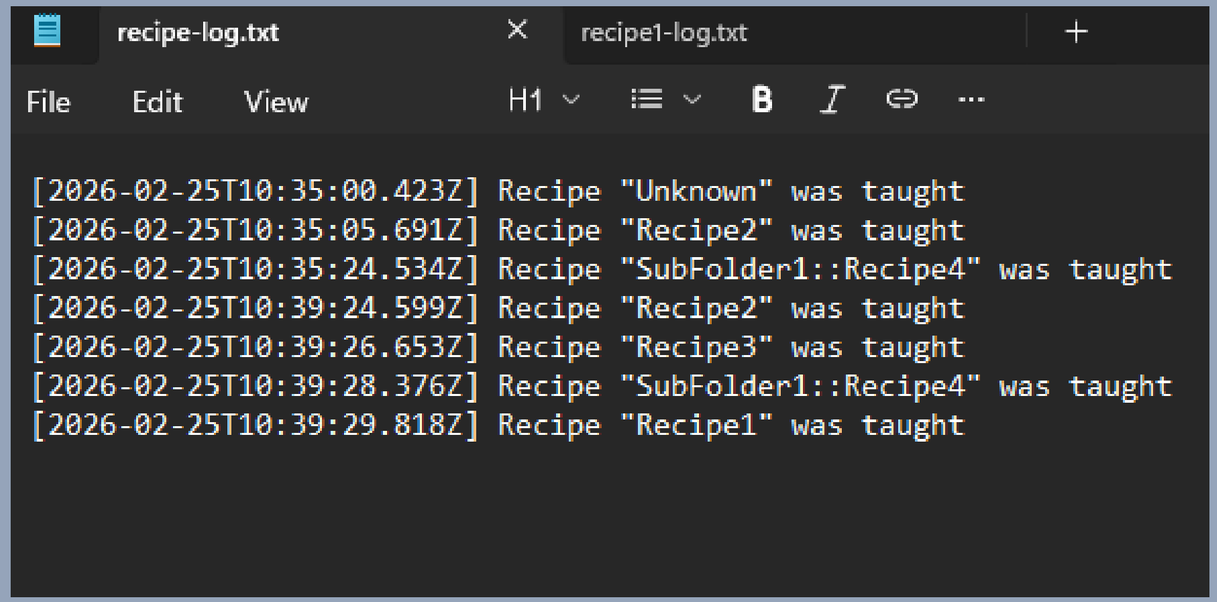
File Created :



Name	Date modified	Type	Size
node_modules	2/11/2026 2:32 PM	File folder	
index.js	2/25/2026 4:09 PM	JavaScript Source File	2 KB
MyExtensions.Config.json	2/25/2026 10:42 AM	JSON Source File	1 KB
package.json	2/11/2026 2:32 PM	JSON Source File	1 KB
package-lock.json	2/11/2026 2:32 PM	JSON Source File	1 KB
recipe-log.txt	2/25/2026 4:09 PM	Text Document	1 KB
renderer.js	2/25/2026 11:21 AM	JavaScript Source File	1 KB

Each Entry Includes:

- Timestamp
- Recipe Name
- Action Type



These are the logs that tell the timestamp, Recipe Name, and Action Type for each recipe saved in “recipe-log.txt”

THANK YOU!!!