

Filière

Systèmes industriels

Orientation Infotronics

Rapport intermédiaire

Diplôme 2025

Marcelin Puippe

SoftPLC pour l'IoT

Professeur
HEI-Vs, Prof. Métrailler Christopher, christopher.metrailler@hevs.ch

Expert
WAGO Contact SA, Stéphane Rey, stephane.rey@wago.com

Date de soumission
17 February 2025



Informations sur ce rapport

Coordonnées

Auteur: Marcelin Puippe
Bachelor Étudiante
HEI-Vs
E-Mail: marcelin.puippe@hevs.ch

Déclaration sur l'honneur

Je soussigné(e), déclare par la présente que le travail soumis est le résultat d'un travail personnel. Je certifie ne pas avoir eu recours au plagiat ou à d'autres formes de fraude. Toutes les sources d'information utilisées et les citations d'auteurs ont été clairement mentionnées.

Lieu, date: Sion, 25.07.2025

Signature:



Contenu

Remerciements	1
Donnée de travail de bachelor	2
Développement durable	4
1 Introduction	5
1.1 Point de départ	5
1.2 Objectif	5
1.3 Application de démonstration pour maison connectée	6
2 Etat de l'art	7
2.1 n8n [1]	7
2.2 total js [2]	9
2.3 Analyse critique du code existant	9
2.3.1 Ajouter des blocs simples	10
2.3.2 Erreurs non gérées	12
3 Conception	14
3.1 Environnement de développement	15
3.2 Aperçu du système	16
3.3 Bloc de haut niveau	17
3.4 User Interface (UI) Design	18
3.4.1 Vue User WebSocket	18
3.4.2 Vue mode debug	21
3.5 Gestion et stockage des données	22
3.5.1 Node View User WebSocket output	22
3.5.2 View User WebSocket data	22
3.5.3 Mode debug data	24
3.5.4 Node : blocs complexes – transmission des <i>Settings</i>	25
3.6 Conclusion	25
4 Implémentation	26
4.1 WDA	28
4.1.1 inputUpdate.go	28
4.1.2 outputUpdate.go	29
4.2 Intégration des blocs logiques simples	30
4.2.1 Bloc Bool to String	30
4.2.2 Bloc String to Bool	30
4.2.3 SR	33
4.2.4 Counter	33
4.2.5 Concat	34
4.2.6 Retain value	35
4.2.7 Comparator EQ	35
4.2.8 Comparator GT	36
4.2.9 Variables	37
4.3 Intégration des blocs logiques de communication	38
4.3.1 MQTT	41

4.3.2	Node HTTP client	41
4.3.3	Node HTTP serveur	43
4.3.4	MODBUS	47
4.4	Vue programmation	48
4.5	Vue User	49
4.6	Vue mode debug	50
4.7	Améliorations interface	52
4.7.1	Rajouter des raccourcies	52
4.7.2	Nodes	53
4.7.3	Couleurs de connection dynamique	54
4.7.4	Elément sélectionnées	54
4.7.5	Accordion - css	55
4.7.6	Boutons	55
4.7.7	Tools	57
4.8	Gestion des erreurs	58
4.8.1	Outputs	58
4.8.2	Timer	59
4.9	Conclusion	59
5	Validation	60
5.1	WDA défauts	61
5.2	Proof of Concept : maison intelligente	62
5.3	Conclusion	63
6	Conclusion	64
6.1.	Résumé du projet	64
6.1.1.	Fonctionnalités développées :	64
6.1.2.	changement de l'interface	64
6.2.	Comparaison avec les objectifs initial	65
6.3.	Difficultés rencontrées	65
6.4.	Perspectives d'avenir	66
6.4.1.	Idées d'amélioration et extensions du frontend web	66
7	Annexe	67
7.1	Planning	68
7.2	WDA analyse 751-9401	73
7.2.1	Sans wda : accéder IO	73
7.3	WDA Monitoring Lists	75
7.3.1	Création	75
7.3.2	Utilisation	76
7.4	WDA access mode	81
7.5	WDA I/O	82
7.5.1	Configuration des DIO (activation des Outputs) via WDA	82
7.5.2	Activation d'une Output via WDA	83
8	Exemples codes : vue programme + JSON	84
8.1	Exemple mode debug : simple	84
8.2	MQTT configuration détaillé	91
8.3	Exemples MQTT	93

8.3.1	Exemple 1	93
8.3.2	Exemple 2	98
8.4	HTTP Client : Exemple WDA intégré	100
8.5	HTTP Server : Exemples	110
8.6	<i>Home-IO</i>	120
8.7	Exemples Modbus Read Bool	122
8.7.1	Exemple 1 – Modbus Read Bool – sans Quantity	122
8.7.2	Exemple 2 – Modbus Read Bool – avec Quantity	124
8.7.3	Exemple 3 – Modbus Read Bool – démonstration complète de la différence entre Quantity et Addresses	125
8.7.4	Exemple 4 – Modbus Read Bool – sans Addresses	126
8.8	Exemple Modbus Read Value	127
8.9	Exemple Modbus Write Bool	128
8.10	Exemple Modbus Write Value	132
9	Application: Home Controller	134
9.1	Recevoir commande bouttons - HTTP serveur	134
9.2	Gestion enclenchement du chauffage	136
9.3	Gestion porte du garage	142
9.3.1	Modbus - porte du garage	142
9.3.2	logique - porte du garage	145
9.4	Gestion de la lampe de couleur	148
9.5	Gestion lumières	151
10	Ensemble des blocs (Nodes)	152
11	Autres annexes	153
11.1	My Strom : documentation	153
	Glossaire	154

Remerciements

Je tiens à remercier mon répondant de travail de bachelor, M. Christopher Métrailler ainsi que son assitant Michael Clausen qui ont répondu à mes différentes questions, pour leurs conseilles et pour avoir pris de leur temps pour mon travail. Je remercie aussi Arnaud Ducrey pour son TB documenté.

Donnée de travail de bachelor

HES-SO Valais

SYND	ETE	TEVI
X	X	X

Données du travail de bachelor
Aufgabenstellung der Bachelorarbeit

FO 1.2.02.07.FB
che/13.03.2024

Filière / Studiengang SYND	Année académique / <i>Studienjahr</i> 2024-25	No TB / Nr. BA IT/2025/5
Mandant / <i>Auftraggeber</i> <input type="checkbox"/> HES—SO Valais-Wallis <input checked="" type="checkbox"/> Industrie <input type="checkbox"/> Etablissement partenaire <i>Partnerinstitution</i>	Etudiant-e / <i>Student/in</i> Marcelin Puippe Professeur-e / <i>Dozent/in</i> Christopher Métrailler	Lieu d'exécution / <i>Ausführungsort</i> <input checked="" type="checkbox"/> HES—SO Valais-Wallis <input type="checkbox"/> Industrie <input type="checkbox"/> Etablissement partenaire <i>Partnerinstitution</i>
Travail confidentiel / <i>vertrauliche Arbeit</i> <input type="checkbox"/> oui / ja <input checked="" type="checkbox"/> non / nein	Expert-e / <i>Experte/Expertin</i> (<i>nom, prénom, E-Mail / Name, Vorname, E-Mail</i>) Stéphane Rey, WAGO Contact SA, stephane.rey@wago.com	

Titre SoftPLC pour IoT
Description L'entreprise WAGO commercialise des automates programmables industriels basés Linux, également équipé d'un environnement SoftPLC basé Codesys. Un PoC d'un environnement SoftPLC basé Web a été développé dans un précédent projet. Ce travail de diplôme consiste à développer un nouveau HAL permettant d'intégrer les nouvelles interfaces des PLC WAGO CC100 (751-9401 et 751-9402), tel qu'une interface CAN. L'environnement SoftPLC doit être complété afin d'intégrer des modules I/O plus complexes. De nouveaux composants doivent être développés afin d'utiliser ces derniers de manière graphique dans le SoftPLC. Ces développements doivent permettre d'intégrer aisément de nouvelles interfaces et capteurs au PLC WAGO dans le futur, comme des IoT disponibles dans des maisons connectées (CAN, HTTP/MQTT, WiFi, etc.). Un banc de test équipé de plusieurs capteurs permettra de valider le bon fonctionnement des développements. Objectifs du projet individuel (Pr4) <ul style="list-style-type: none"> — Étude et analyse du projet SoftPLC existant. Prise en main de l'environnement de développement et des outils Docker pour le déploiement d'application sur les PLC Wago CC100 — Développement d'applications de démonstration basées Golang (backend) et Javascript/TypeScript (frontend) pour le système existant en utilisant le banc de test existant comme base Objectifs du travail de Bachelor (TB) <ul style="list-style-type: none"> — Modification du Hardware Abstraction Layer (HAL) pour les nouveaux automates CC100. Développement d'un nouveau HAL pour les I/O et le module CAN en utilisant l'API VDX — Définition et implémentation de nouveaux blocs SoftPLC comprenant des blocs de haut niveau, tels que client MQTT, CAN, WebServer, client/serveur HTTP, etc. Extension du système SoftPLC existant et de l'interface web pour permettre l'ajout de ces nouveaux composants — Amélioration et extensions du frontend web avec contrôle du type d'entrées/sorties, des types des signaux. Amélioration de l'interface et de l'expérience utilisateur avec un meilleur visuel et des contrôles automatisés — Développement d'un banc de test physique et d'une application de démonstration pour une maison connectée. Documentation et tests et rédaction du rapport, poster et présentation.

Délais / Termine	
Démarrage du projet individuel (Pr4) <i>Start des individuellen Projekts (Pr4)</i> 17.02.2025	Remise du rapport final / <i>Abgabe des Schlussberichts:</i> 14.08.2025, 12:00
Présentation du projet individuel (Pr4) <i>Präsentation:des individuellen Projekts (Pr4)</i> 02.05.2025	Expositions et Pitch / <i>Ausstellungen und Pitch der Bachelorarbeiten:</i> 22.08.2025 – HEI 25.08.2025 – Monthey 28.08.2025 – Visp
Démarrage du travail de bachelor <i>Start der Bachelorarbeit:</i> 19.05.2025	Défense orale / <i>Mündliche Verfechtung:</i> Semaines/Wochen 36-37 (01-12.09.2025)

Signature ou visa / <i>Unterschrift oder Visum</i>	
Responsable de l'orientation / <i>Leiter/in der Vertiefungsrichtung:</i> 	¹ Etudiant·e / Student/in: 

¹ Par sa signature, l'étudiant·e s'engage à respecter strictement la directive DI.1.2.02.07 « Travail de bachelor ». Durch seine Unterschrift verpflichtet sich er/die Student/in, sich an die Richtlinie DI.1.2.02.07 „Bachelorarbeit“ zu halten.

Développement durable

Traduction de l'anglais par chatgpt [3] :

L'Agenda 2030 pour le développement durable, adopté par l'ensemble des États membres des Nations Unies en 2015, constitue une feuille de route commune pour la paix et la prospérité des peuples et de la planète, aujourd'hui et pour l'avenir. Au cœur de cet agenda se trouvent les 17 Objectifs de Développement Durable (ODD) [4], un appel urgent à l'action lancé à tous les pays – développés comme en développement – dans le cadre d'un partenariat mondial. Ces objectifs reconnaissent que l'éradication de la pauvreté et d'autres privations doit aller de pair avec des stratégies visant à améliorer la santé et l'éducation, à réduire les inégalités et à stimuler la croissance économique – tout en luttant contre les changements climatiques et en œuvrant à la préservation des océans et des forêts.



Fig. 1. – 17 Objectifs de Développement Durable (ODD) [4]

Le projet consiste en la création d'un [Hardware Abstraction Layer \(HAL\)](#) de développement d'automate. Il est donc très probable que le projet soit utiliser dans des applications allant dans le sens d'un développement durable. En effet, l'automatisation des processus industriels peut contribuer à la durabilité en améliorant l'efficacité énergétique (ODD 7), en réduisant les déchets et en optimisant l'utilisation des ressources. De plus, l'automatisation peut également aider à surveiller et à contrôler les émissions de gaz à effet de serre, ce qui contribue à la lutte contre le changement climatique. L'automatisation favorise également la croissance économique (ODD 8) en augmentant la productivité et en réduisant les coûts de production. En outre, l'automatisation peut également améliorer la sécurité au travail (ODD 3) en réduisant le risque d'accidents liés à des tâches dangereuses.

Cependant, le programme peut également être utilisé pour des applications moins durables. Par exemple, il peut être utilisé pour automatiser des processus industriels qui ont un impact environnemental lourd (ODD 13), comme l'extraction de combustibles fossiles ou la production de déchets toxiques (ODD 14-15).

1 | Introduction

Ce document présente les travaux réalisés ainsi que les perspectives à venir pour le projet **PLCSoft** destiné à **WAGO**.

Ce projet consiste en l'ajout de nouvelles fonctionnalités à un **HAL** (Hardware Abstraction Layer) pour le développement d'automates via une interface web. La programmation se fait de manière graphique et modulaire sur une page web, en reliant les différentes fonctions disponibles entre elles. Le but étant de permettre une programmation simple et intuitive. L'objectif principal du projet est l'ajout de nouveaux blocs fonctionnels pouvant être utilisés dans ce cadre de développement et plus spécifiquement l'ajout de blocs de communication réseau. En effet, la communication réseau est essentielle pour le développement de programmes automates modernes et le développement dans des systèmes **Internet of Things (IoT)**, comme nous l'explique l'article [5].

« Scientists claim that the potential benefit derived from this technology will sprout a foreseeable future where the smart objects sense, think and act. Internet of Things is the trending technology and embodies various concepts such as fog computing, edge computing, communication protocols, electronic devices, sensors, geo-location etc. »

Il y a également l'article [6] qui explique l'importance de l'**IoT** dans de nombreux domaines comme la santé, l'agriculture, l'industrie 4.0, la domotique, etc. Il est donc important pour **WAGO** de permettre le développement de programmes automates pour ces systèmes.

1.1 Point de départ

Le projet se construit sur la base de deux programmes déjà développés, lors du TB 2024 :

- **Softplcui-main** : Gérant l'interface web (**frontend**).
- **Softplc-main** : Gérant l'activation des entrées / sorties selon le programme build depuis PLC UI (**backend**).

Le travail effectué précédemment nous prouve la faisabilité du développement d'un tel **HAL**. Les fonctionnalités implémentées par ce précédent projet sont :

- Digital input / output
- Analogue input / output
- Ton (timer retardé à la montée)

1.2 Objectif

L'objectif est l'amélioration et l'implémentation de nouvelles fonctionnalités. Les tâches devant être réalisées sont les suivantes :

- Modifier les programmes actuels pour utiliser la nouvelle interface REST WDA pour piloter les nouveaux automates.
- L'implémentation de nouveaux blocs de haut niveau comme CAN, MQTT, Web-Server, client/serveur HTTP et autres bloc. Il faudra trouver une solution pour faire ces tâches par programmation en bloc.
- Amélioration et extensions du **frontend** web.

- Développement d'un banc de test physique et d'une application de démonstration pour une maison connectée.
- Documentation, tests et rédaction du rapport, poster et présentation.

1.3 Application de démonstration pour maison connectée

L'application de démonstration d'une maison connectée (Fig. 2) a pour but de démontrer les capacités du HAL qui sera développé dans le cadre de ce projet. Il s'agit de réaliser un programme automate permettant de contrôler et de surveiller les différents équipements d'une maison connectée, tels que les lumières, les prises électriques, les capteurs de température, etc. Une page web doit permettre de visualiser l'état des équipements et de les contrôler.

La fonctionnalité de l'application de démonstration qui a été choisie et qui sera développée est la suivante : Depuis une interface web, il sera possible de contrôler une lampe en réglant son intensité et sa couleur. Depuis cette même interface, il sera également possible de régler une consigne de température. Un capteur de température enverra la température et une prise électrique s'activera si la température est trop basse. De plus avec un bouton *Shelly*, on pourra contrôler la porte de garage et avec un autre l'éclairage.

Le résultat final est présenté au Chapitre 5.2 et l'annexe Chapitre 9 montre le programme. Cependant, il sera difficile à comprendre sans avoir assimilé les chapitres précédents.

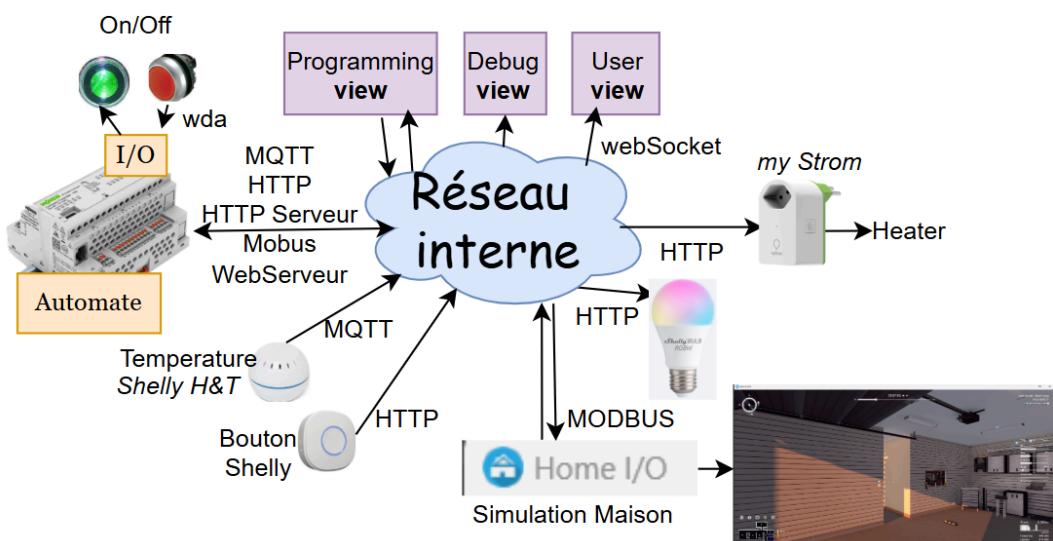


Fig. 2. – Application de démonstration pour **maison connectée**

2 | Etat de l'art

Durant le cours projet 4, plusieurs ressources pouvant être utile pour la suite ont été trouvées. Cette section présente aussi le travail effectué durant le TB 2024.

2.1 n8n [1]

C'est un logiciel d'automation présent en ligne sur GitHub. Il permet une programmation en no-code sur page web comme ce qu'on l'on essaie de faire. Il est surtout conçu pour l'automation de tâche simple. Il permet notamment l'automation de chat alimenté par l'IA, c'est-à-dire des réponses automatiques. Ce n'est pas ce qui nous intéresse mais cela peut nous aider à avoir des idées.

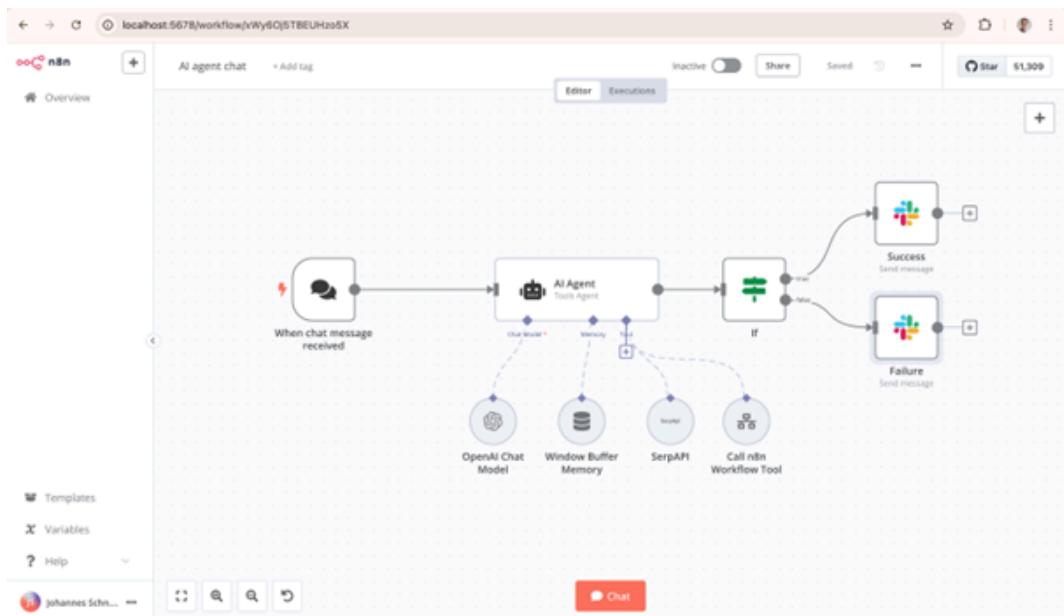


Fig. 3. – Exemple GitHub n8n

L'activation d'une output en n8n peut se faire de la manière suivante. Il suffit d'un bloc HTTP Request1 qu'on configure.

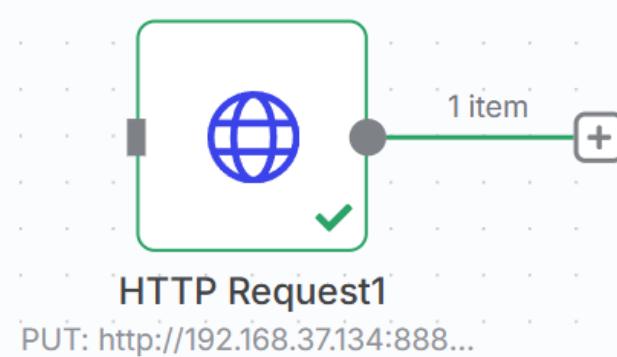


Fig. 4. – Bloc HTTP Request1 en n8n

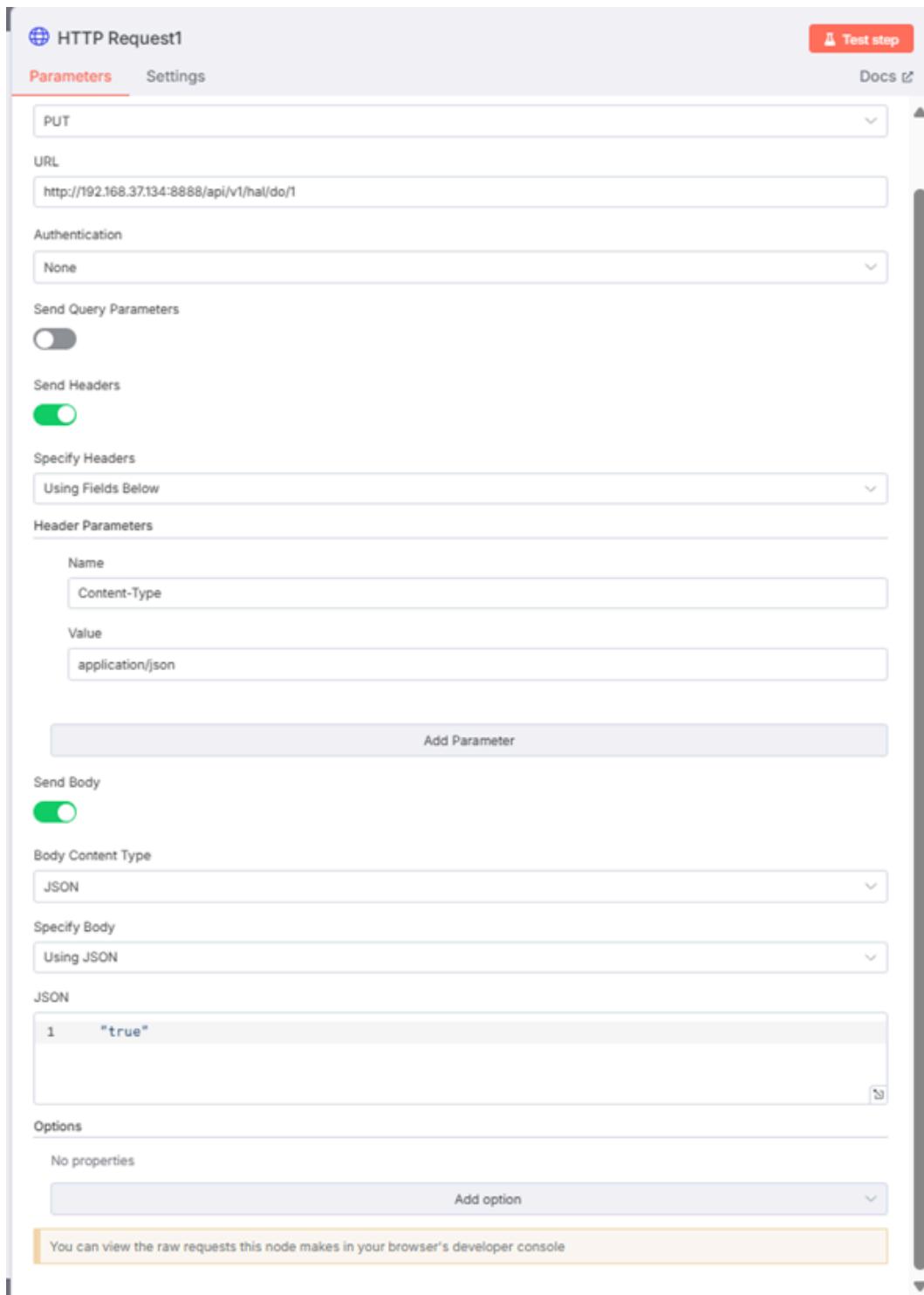


Fig. 5. – configuration HTTP Request1 en n8n pour activation output 2

Le code de n8n est disponible sur GitHub [1]. Une analyse a été faite, mais il utilise une librairie différente de celle que nous utilisons. Nous ne pouvons donc pas nous en inspirer directement. Cependant, il est possible de s'en inspirer pour la création de l'interface graphique et de la logique de programmation.

2.2 totaljs [2]

C'est un logiciel de programmation en no-code. Il est possible de faire des programmes en JS, mais il y a aussi une interface graphique.

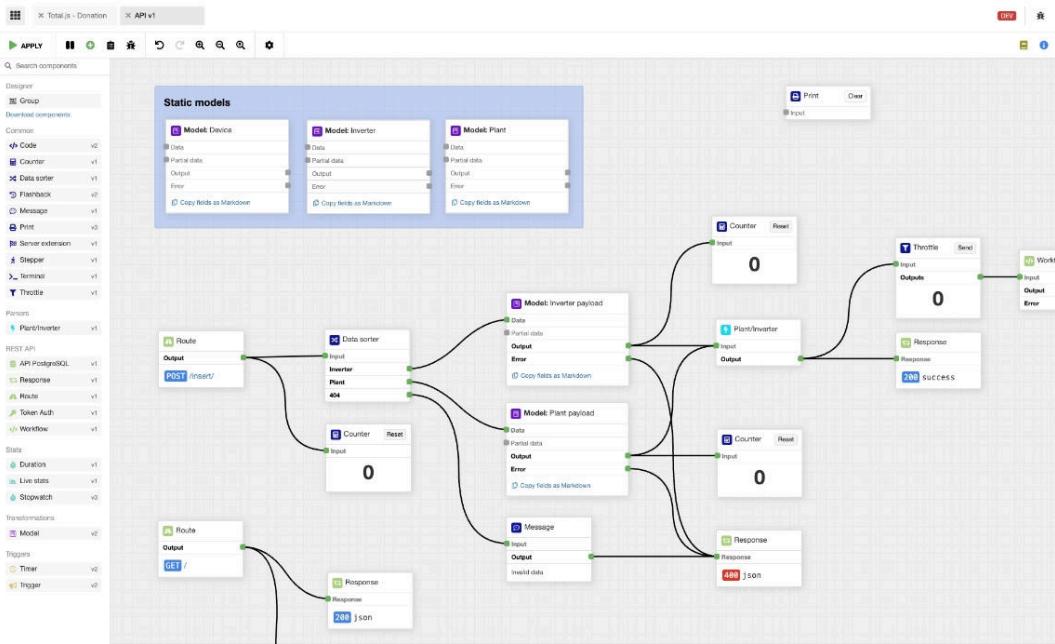


Fig. 6. – Exemple totalJS

2.3 Analyse critique du code existant

Le code existant est un bon point de départ pour la création d'un **HAL** de développement d'automate. Cependant, il y a plusieurs points à améliorer.

Le code est presque totalement fonctionnel malgré quelques petites erreurs. Cependant, la structure actuelle ne permet pas l'intégration de blocs plus complexes que ce qui a été fait. En effet, par exemple, nous pouvons recevoir et transmettre qu'un nombre très limité de paramètres, et la structure de ceux-ci n'est pas très flexible. Le typage des blocs est également très limité. Il utilise le type **float64** pour tous les blocs, ce qui n'est pas adapté à la transmission de données plus complexes. Il serait préférable d'utiliser un type plus flexible, comme un type **any**, un type générique ou au moins un type **string**. De plus, le code est très difficile à lire et à comprendre car il utilise beaucoup d'imbrications de boucles **for** et **if**. Il manque des commentaires. Il n'a également pas prévu la possibilité d'avoir **plusieurs output** pour un bloc.

Au niveau visuel, la structure doit être améliorée. En effet, on ne peut pas continuer à mettre tous les blocs logiques dans le même fichier. Il faudrait au moins créer des fichiers séparés pour chaque type de bloc logique. La taille des blocs n'est également pas adaptée aux besoins.

Le code ne permet pas de changer le type d'un bloc de manière dynamique, car c'est le programme **backend** qui l'envoie à l'initialisation du programme. Cela est donc impossible sans de grosses modifications.

Il n'y a pas non plus de méthode permettant de réinitialiser les nodes, ce qui est nécessaire pour pouvoir créer des blocs plus complexes. En effet, il peut être nécessaire de réinitialiser certains nodes pour pouvoir faire un nouveau *build* proprement.

2.3.1 Ajouter des blocs simples

Il y a également des points positifs. La structure des « Nodes » est facile à utiliser. Pour ajouter un bloc simple, il suffit de créer un fichier dans le dossier **nodes** du programme **backend** (**softplc-main**). On peut, par exemple, copier-coller puis renommer un fichier existant selon le type de bloc à ajouter. Il existe trois types de blocs : **LogicalNode**, **OutputNode** et **InputNode**.



nommer un fichier : il peut parfois y avoir des problèmes si le nom du bloc commence par une lettre en début d'alphabet.

Vous pouvez copier le fichier **OrNode**, puis le modifier. Pour cela, utilisez la fonctionnalité *rechercher/remplacer* afin de remplacer "**Or**" par le nom de votre nouveau bloc. Il faut le faire deux fois :

- une fois avec "**Or**" (majuscule)
- une fois avec "**or**" (minuscule)

Les éléments suivants doivent ensuite être adaptés :

1. La fonction **ProcessLogic**, où vous écrivez la logique spécifique de votre bloc.
2. La variable de type de la structure **nodeDescription**, dont le lien avec la partie visuelle est montré dans la figure Fig. 7.

The screenshot shows a software interface with a sidebar containing components like 'And Node', 'Not Node', and 'Or Node'. A main panel displays a 'Logical gate' component with two inputs ('Input0', 'Input1') and one output ('Output'). The label for the component is 'Label (16) >=1'. Below the interface is a code editor showing the following JavaScript code:

```

11 var orDescription = nodeDescription{ 1 usage
12   AccordionName: "Logical gate",
13   PrimaryType: "LogicalNode",
14   Type_: "OrNode",
15   Display: "Or Node",
16   Label: ">=1",
17   Stretchable: true,
18   Services: []servicesStruct{},
19   SubServices: []subServicesStruct{},
20   Input: []dataTypeNameStruct{{DataType: "bool", Name: "Input"}},
21   Output: []dataTypeNameStruct{{DataType: "bool", Name: "Output"}},
22 }

```

The code highlights the 'Display', 'Label', and 'Stretchable' properties of the 'OrNode' component.

Fig. 7. – **nodeDescription** VS vue – modifier le fichier **OrNode** pour créer un nouveau bloc

2.3.2 Erreurs non gérées



Certaines erreurs n'ont pas été traitées lors de l'ancien TB. Cette section présente ces erreurs qui devront être réglées.

2.3.2.1 Manque lien

Le problème est que le programme plcSoft plante au lieu d'afficher simplement une erreur et de ne pas *build* le programme dans l'automate.

Cependant, le save est possible et le restore peut être fait après avoir relancé le programme.

Remarque Bloc bleu : il y a des bloc bleu qui est le résultat d'un test explication dans la synthèse.

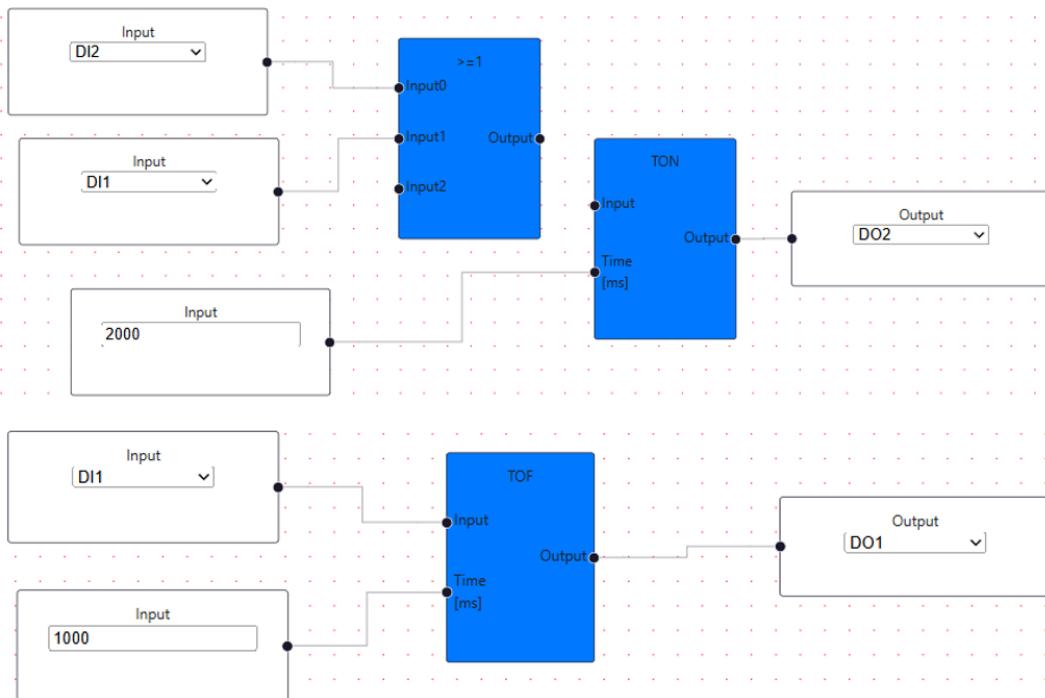


Fig. 8. – Défaut manque lien entre OR et TON

```
OutputNodes:
&{3 digitalOutput [{ D01 {0xc00026f050 Input bool}}]}
&{6 digitalOutput [{ D02 {0xc00026f140 Input bool}}]}
LogicalNode:
&{0 TOFNode [{0xc000228018 Input bool} {0xc00046ae78 Time [ms] value}] [{0 Output bool}] {<nil> false} false 0 false}
&{7 TONNode [{<nil> Input bool} {0xc00041cbbb Time [ms] value}] [{0 Output bool}] {<nil> false} false 0 false}
InputNodes:
&{2 digitalInput [{ DI1 {0xc000228018 Output bool}}]}
&{4 constantInput [{ {0xc00046ae78 Output value}}]}
&{8 constantInput [{ {0xc00041cbbb Output value}}]}
Const:
{4 1000}
{8 2000}
```

Fig. 9. – message plcsoft save défaut

```

panic: runtime error: invalid memory address or nil pointer dereference
[signal 0xc0000005 code=0x0 addr=0x0 pc=0xc1a412]

goroutine 1 [running]:
SoftPLC/nodes.(*TONNode).ProcessLogic(0xc0001813b0)
    C:/Users/puiipp/Desktop/TB_PLCSoft_Wago/Software/Go/softplc-main/softplc-main/nodes/tonNode.go:89 +0x1d2
main.main()
    C:/Users/puiipp/Desktop/TB_PLCSoft_Wago/Software/Go/softplc-main/softPLC.go:31 +0x13c

Process finished with the exit code 2

```

Fig. 10. – Erreur Build manque lien entre OR et TON

2.3.2.2 Plusieurs Output pour une Input



Cela devrait être faisable, pourtant c'est interdit.

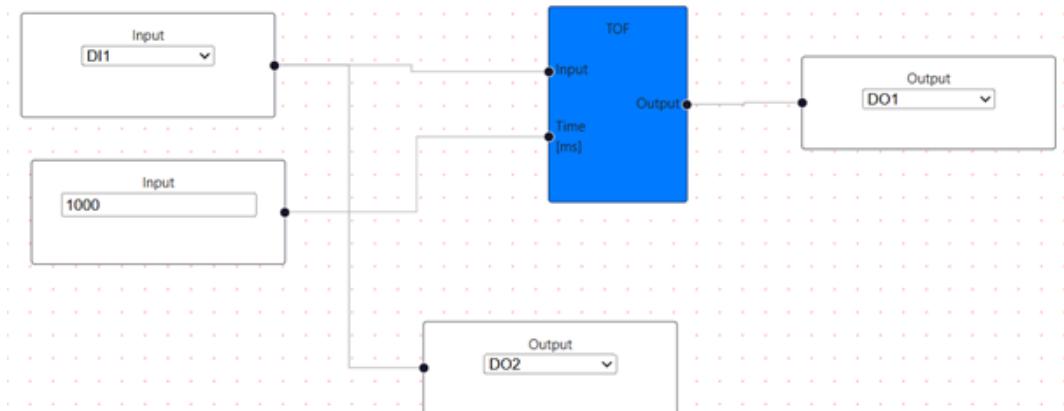


Fig. 11. – Erreur 2 output pour un input, avec une output directement sur l'input

3 | Conception

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliquam quaerat voluptatem. Ut enim aequre doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut.

Contenu

3.1 Environnement de développement	15
3.2 Aperçu du système	16
3.3 Bloc de haut niveau	17
3.4 User Interface (UI) Design	18
3.4.1 Vue User WebSocket	18
3.4.2 Vue mode debug	21
3.5 Gestion et stockage des données	22
3.5.1 Node View User WebSocket output	22
3.5.2 View User WebSocket data	22
3.5.3 Mode debug data	24
3.5.4 Node : blocs complexes – transmission des <i>Settings</i>	25
3.6 Conclusion	25

3.1 Environnement de développement

Appareil	Adresse IP	Utiliser dans soft
Automate WAGO CC100	192.168.37.134	softplc-main
PC	localhost	softplcui-main

La salle **23.N320** utilisé pour connecté l'automate WAGO CC100 sur le réseau.

Logiciel	Appareil	Commentaires
Goland	PC	Développement logiciel
Umlet	PC	Réalisation de schéma basé développement
Firefox	PC	Permettant meilleure visualisation de WDx
HTTPPie	PC	Permettant de tester les requêtes HTTP
miro	site	Réalisation de schéma, prise de note, réflexion, analyse
WDx	Automate	Dénomination générale pour parler de WDM + WDD + WDA : <ul style="list-style-type: none"> • WDM = WAGO Device Model (standard utilisé pour les WDD) • WDD = WAGO Device Description (manifeste décrivant ce que le produit met à disposition) • WDA = WAGO Device Access (accès aux paramètres et IO)
WDA	Automate	Nouvelle librairie, interface REST, WAGO accessible par web en JSON. Permet la récupération des informations de l'automate et le contrôle des entrées/sorties par requête HTTP en format JSON. https://192.168.37.134/wda/parameters?page[limit]=20000 https://192.168.1.126/wda/parameter-definitions?page[limit]=20000

3.2 Aperçu du système

L'aperçu du système est présenté dans le schéma de principe Fig. 12. Il montre les différentes parties du système et comment elles interagissent entre elles. Le schéma n'est pas exhaustif, mais il donne une idée générale des grands principes du système.

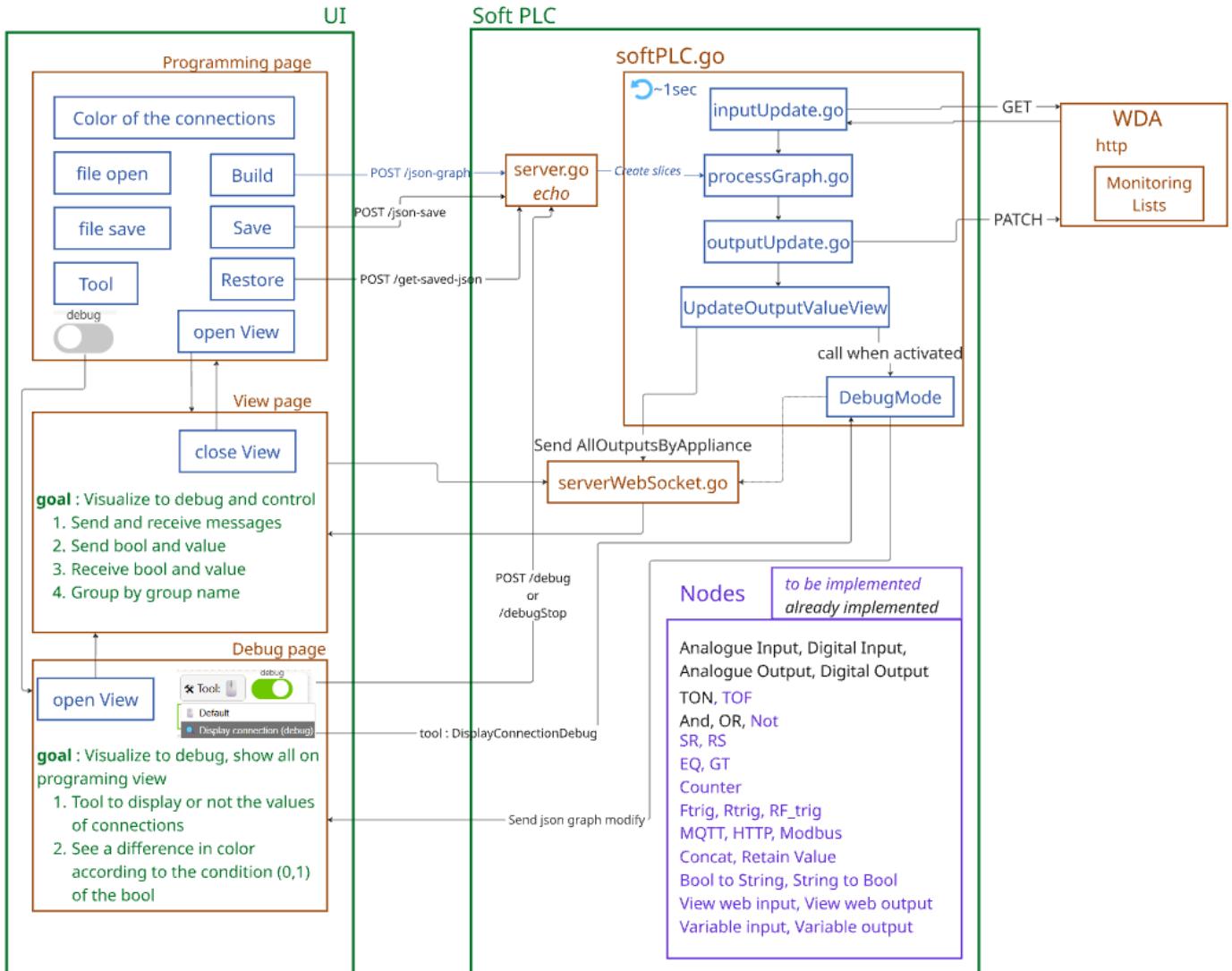


Fig. 12. – aperçu du système - schéma de principe

3.3 Bloc de haut niveau

Il existe plusieurs manière d'aborder le problème. Une des approches est de repérer les points commun entre ces blocs de haut niveau pour essayer d'en tirer une forme générique. On remarque que tous ces blocs ont pour objectif de transmettre et recevoir des données. Il faudra donc commencer par le développement de bloc commun pour une communication. Il faut également des blocs permettant de travailler avec des STRING. Le schéma figure 1 montre le concept d'une telle structure avec tous les blocs qui devront être développé autour pour pouvoir créer une communication.

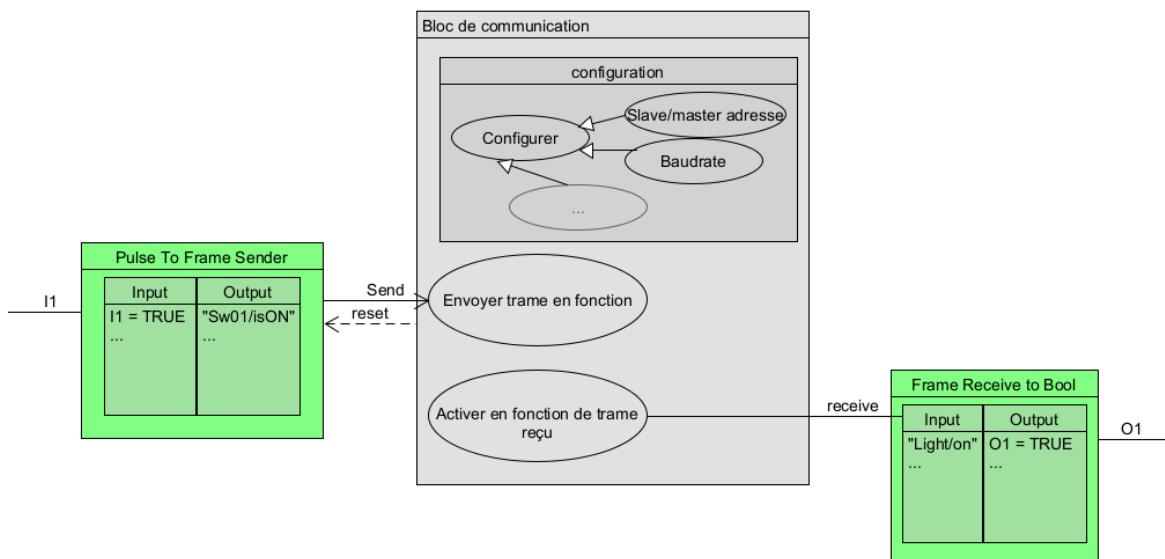


Fig. 13. – Communication principe

L'idée étant d'avoir un bloc communication qui s'occupe de la configuration étant différente pour MQTT, HTTP, CAN, etc. Sur lequel, on pourra double cliquer pour accéder à la page de configuration. Sur ce bloc de communication, on pourrait ensuite venir lier nos 2 blocs permettant la transition de boolean vers nos trame. Par le futur, en mode debug, l'utilisateur pourra voir l'état de la communication grâce au « bloc de communication » et voir ce que la logique combinatoire transmet comme trame grâce au bloc en vert.



3.4 User Interface (UI) Design

L'interface utilisateur utilise *React*, la documentation [7] peut aider. On utilise également *React flow*, dont la documentation [8], qui s'occupe plus particulièrement des mécanismes liés aux Nodes, Edges, Handle, etc.

3.4.1 Vue User WebSocket

La vue user WebSocket permet de visualiser en temps réel l'état des *Outputs* et de gérer les des *Inputs* spécifiques à cette vue. Sur le schéma Fig. 12, elle est représentée par la « View page ».

Cette vue s'accompagne de blocs logiques qui permettent sa création. Ces blocs sont représentés dans la figure Fig. 14. Ils permettent de créer des Inputs et Outputs spécifiques à cette vue, permettant ainsi de visualiser l'état de n'importe quelle connexion dans le programme PLC. Un exemple de ce à quoi pourrait ressembler la vue WebSocket est présenté dans la figure Fig. 17, et le programme qui crée cette vue est présenté dans la figure Fig. 16. Les parties *recevoir* et *envoyer* les messages sont prévues pour du débogage. Les autres parties sont prévues pour contrôler et tester le programme PLC. Remarquez l'impact du champ *Appliance Name*, qui permet de créer des groupes dans la vue WebSocket. Cela facilite le regroupement des Inputs et Outputs par groupe, ce qui est très utile pour la visualisation. L'idée a été inspirée de « Remote Controller » vue en cours de Data Engineering, où une page WebSocket est générée à partir d'un fichier JSON, dont voici le lien https://cyberlearn.hes-so.ch/pluginfile.php/3312976/mod_resource/content/2/index.html.

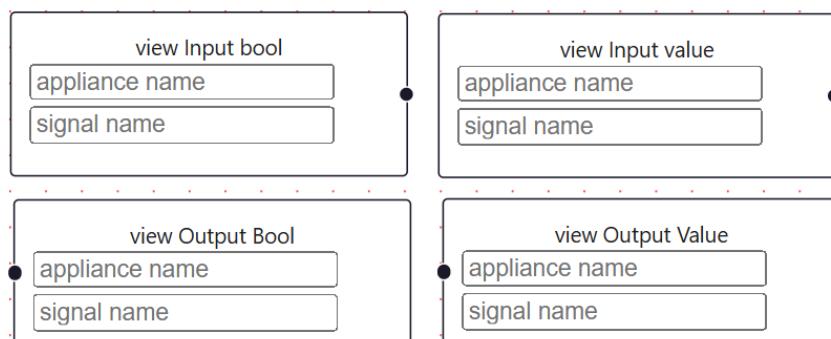


Fig. 14. – blocs vue WebSocket - Inputs et Outputs

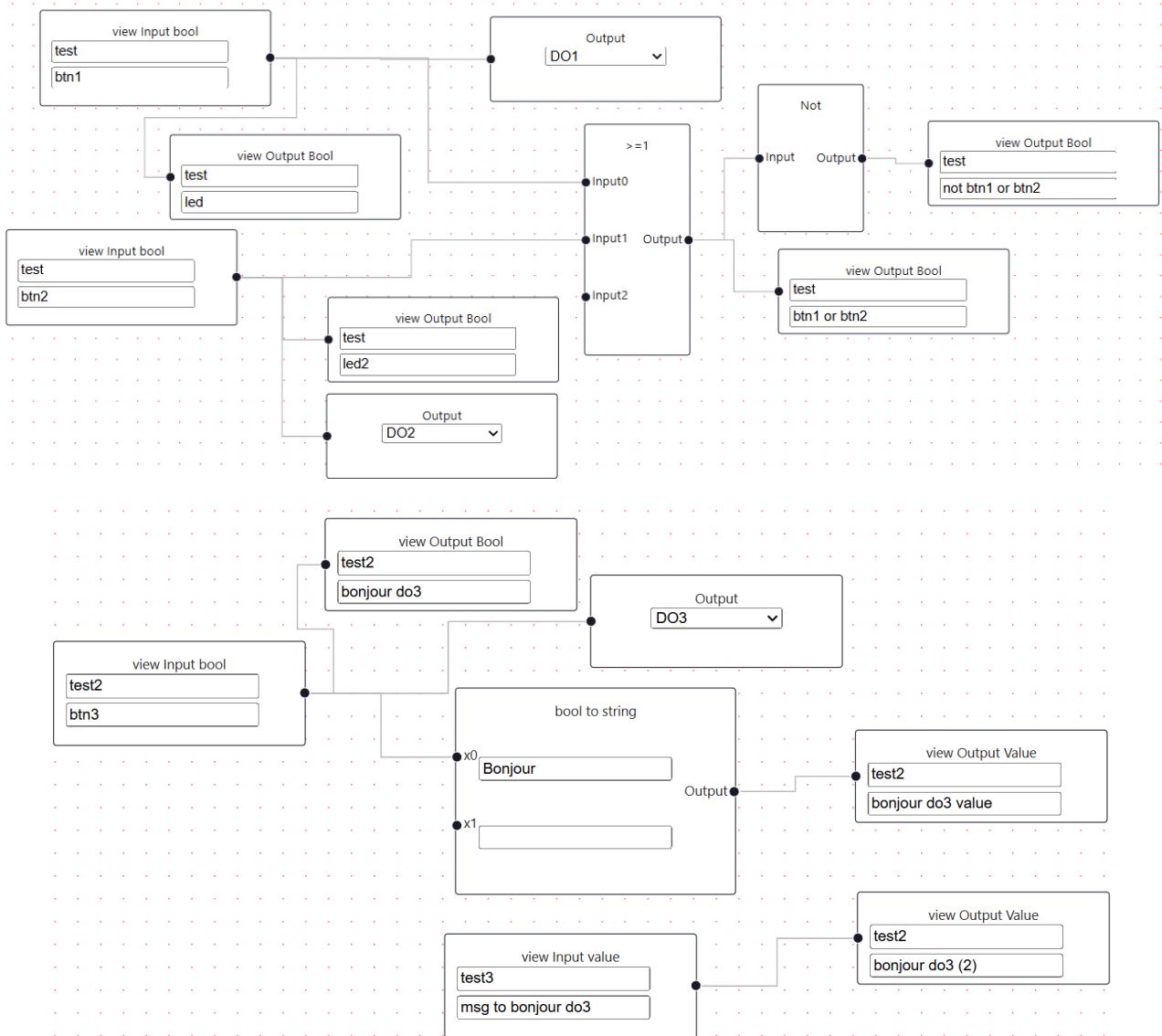


Fig. 16. – Exemple programme WebSocket

[Close view](#)

WebSocket Interface

Enter message Send Envoyer les messages ici

Devices :

test

btn1 btn2 Envoyer les messages sur une input "Bool"

States :

- led : ● OFF
- led2 : ● OFF Visualiser les états des Outputs "bool"
- btn1 or btn2 : ● OFF
- not btn1 or btn2 : ● ON

test2

btn3 Envoyer les messages sur une input "Bool"

States :

- bonjour do3 : ● OFF
- bonjour do3 value : empty Visualiser les états des Inputs "value"
- bonjour do3 (2) :

test3

msg to bonjour do3: Send Envoyer les messages sur une input "value"

Messages :

Recevoir les messages ici

Fig. 17. – exemple vue WebSocket (commentaires en brun)

3.4.2 Vue mode debug

Sur le schéma Fig. 12, elle est représentée par la « Debug page ». Il est également intéressant d'avoir une vue « debug » qui nous permettrait de visualiser les états des I/O de chaque node et même de changer la couleur des edges pour les booléens (rouge = off et vert = on). Pour ce faire, nous utiliserons la technologie des WebSocket également utilisée pour la vue WebSocket (Chapitre 3.4.1). L'idée étant de transmettre un graphique modifié à chaque cycle, la modification du graphique se fera surtout au niveau des *Edges*. Un exemple d'un petit programme qui montre comment sont transmises et affichées les données est détaillé en annexe (Chapitre 8.1).

En figure Fig. 18 se trouve un exemple démontrant entièrement le fonctionnement de ce mode debug. Les edges qui affichent « ??? » sont ceux qu'on n'a pas sélectionnés avec l'outil.

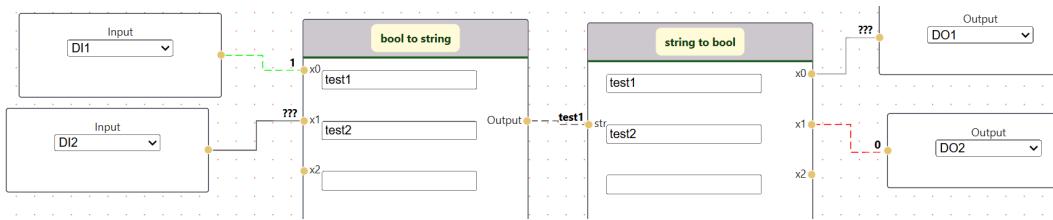


Fig. 18. – exemple vue Mode debug

L'outil en question est représenté par une loupe. Quand il est sélectionné, il ajoute ou enlève les edges de la liste des edges dont on veut afficher les valeurs dans la vue de debug. La boîte à outil est expliquée plus en détail dans la section Chapitre 4.7.7.

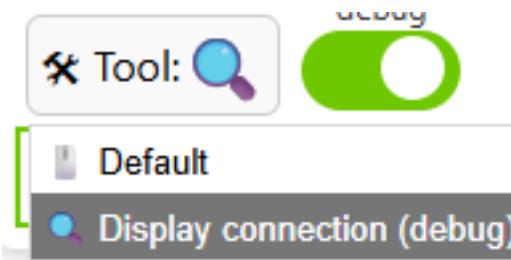


Fig. 19. – Mode debug tool : display connection

La **transmission des données** ainsi que les mécanismes principaux sont décrit dans le schéma se trouvant Chapitre 3.5.3.

3.5 Gestion et stockage des données

La gestion et le stockage des données sont des aspects importants du système. Dans notre cas, les données sont stockées et transmises au format JSON.

3.5.1 Node View User WebSocket output

Pour cette fois, nous n'utilisons pas **parameterValueData** car les **outputs** ont un fonctionnement différent dans le programme, ce qui rend cette implémentation plus complexe. De plus, cela n'est pas nécessaire car, généralement, les **outputs** n'ont pas de paramètres. Il est donc possible de se passer de cette fonctionnalité en utilisant **selectedServiceData** et **selectedSubServiceData**.

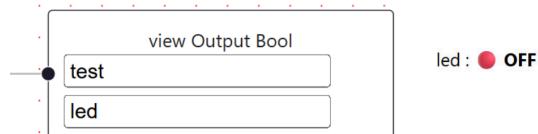


Fig. 20. – exemple de paramétrage de « view Output Bool »

```

"label": "view Output Bool",
"outputHandle": [],
"parameterNameData": null,
"primaryType": "outputNode",
"selectedFriendlyNameData": "default",
"selectedServiceData": "test",
"selectedSubServiceData": "led",
  
```

Liste 1. – view Output Bool, extrait de la structure JSON d'un exemple

3.5.2 View User WebSocket data

Le **WebSocket** est un protocole de communication bidirectionnelle qui permet d'envoyer et de recevoir des données en temps réel. Il est utilisé pour la vue **User** décrite dans le Chapitre 3.4.1.

Dans les figures Fig. 21 et Fig. 22, on peut voir le principe de transmission des données. Les schémas permettent de visualiser toutes les structures de données nécessaires, ainsi que les fonctions (en brun) et les variables (en violet) les plus utiles.

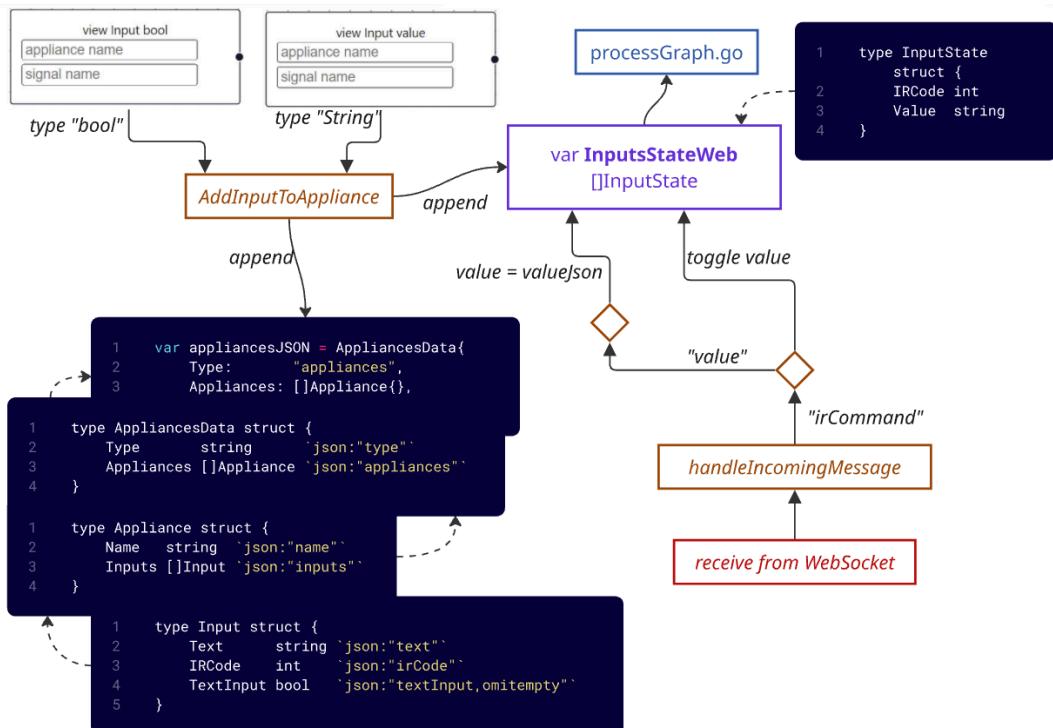


Fig. 21. – principe de transmission des données via WebSocket pour les inputs

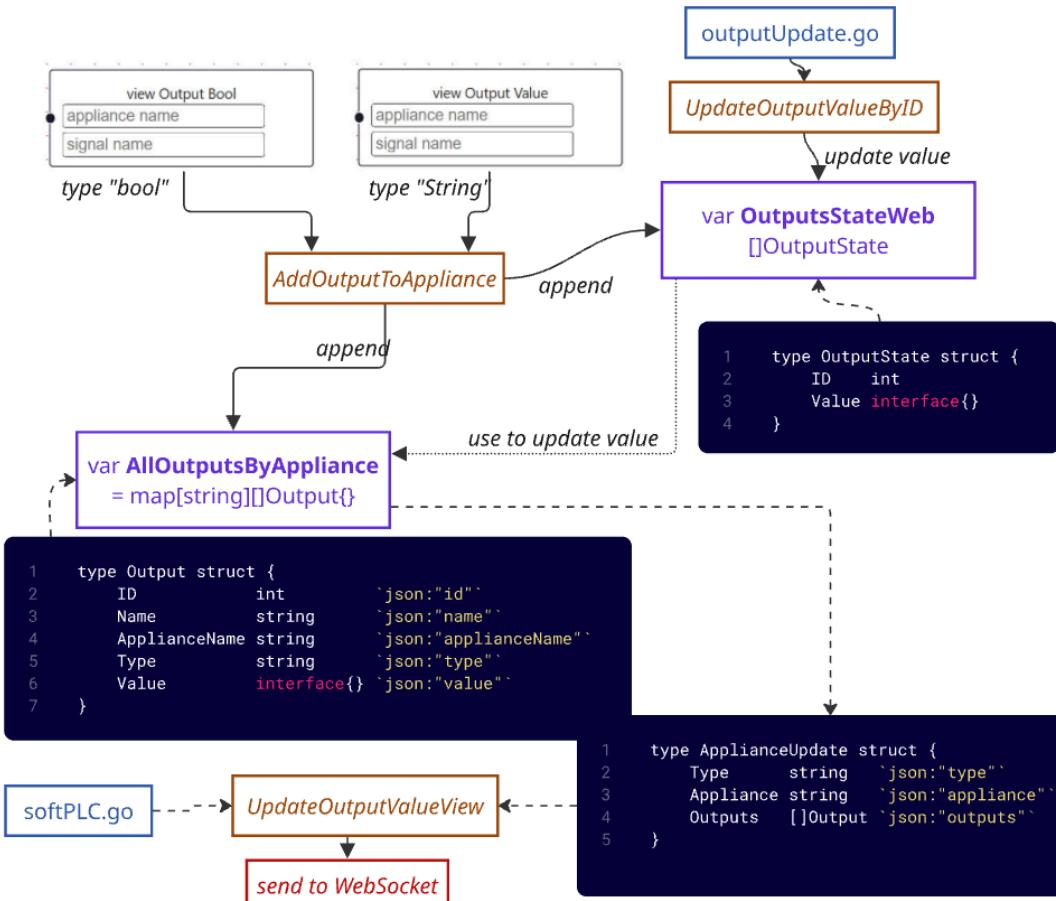


Fig. 22. – principe de transmission des données via WebSocket pour les outputs

3.5.3 Mode debug data

Le schéma Fig. 23 permet de comprendre les mécanismes principaux du mode debug qui utilise également la technologie WebSocket.

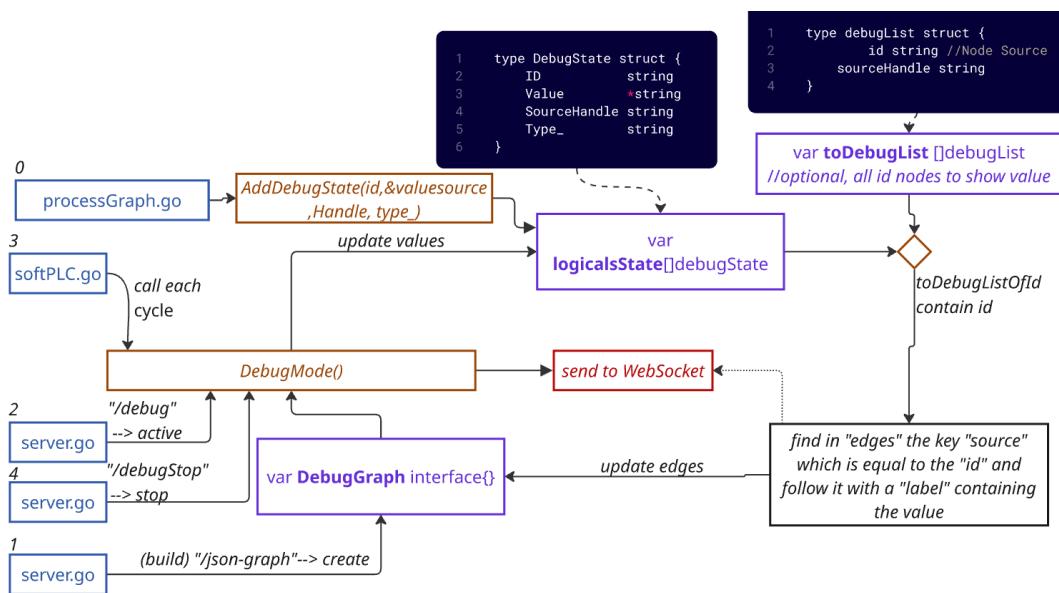


Fig. 23. – Principe de transmission des données via WebSocket et server echo pour le mode debug

3.5.4 Node : blocs complexes – transmission des *Settings*

Pour les blocs complexes, nous devons pouvoir transmettre autant de données que nécessaire. Pour cela, deux tableaux de chaînes (*String*) sont utilisés :

- Un tableau **parameterValueData** contenant les valeurs des paramètres (définis dans le **frontend**).
- Un tableau **parameterNameData** contenant les noms des paramètres (définis dans le **backend**).

Ces tableaux permettent la transmission des données entre le **backend** et le **frontend** pour les *nodes* qui le nécessitent, comme les blocs de communication (MQTT, HTTP, MODBUS), ainsi que les blocs **string to bool** et **bool to string**.

Un exemple de la structure JSON d'un bloc *MQTT* est présenté en annexe, à la section Chapitre 8.2.

3.6 Conclusion

4 | Implémentation

Cette section décrit les différentes étapes qui ont été implémentées durant le projet. Elle est divisée en plusieurs sous-sections, chacune abordant un aspect spécifique. Il y est notamment décrit comment les blocs logiques complexes ont été implémentés, comment WDA a été utilisé, l'ajout d'une vue webSocket ainsi que tout ce qui a dû être mis en place pour que tout cela soit réalisable.

Contenu

4.1 WDA	28
4.1.1 inputUpdate.go	28
4.1.2 outputUpdate.go	29
4.2 Intégration des blocs logiques simples	30
4.2.1 Bloc Bool to String	30
4.2.2 Bloc String to Bool	30
4.2.3 SR	33
4.2.4 Counter	33
4.2.5 Concat	34
4.2.6 Retain value	35
4.2.7 Comparator EQ	35
4.2.8 Comparator GT	36
4.2.9 Variables	37
4.3 Intégration des blocs logiques de communication	38
4.3.1 MQTT	41
4.3.2 Node HTTP client	41
4.3.3 Node HTTP serveur	43
4.3.4 MODBUS	47
4.4 Vue programmation	48
4.5 Vue User	49
4.6 Vue mode debug	50
4.7 Améliorations interface	52
4.7.1 Rajouter des raccourcis	52
4.7.2 Nodes	53
4.7.3 Couleurs de connection dynamique	54
4.7.4 Elément sélectionnées	54
4.7.5 Accordion - css	55
4.7.6 Boutons	55
4.7.7 Tools	57
4.8 Gestion des erreurs	58
4.8.1 Outputs	58
4.8.2 Timer	59
4.9 Conclusion	59

4.1 WDA

L'intégration de **WAGO Device Access** (accès aux paramètres et IO) (**WDA**) est une partie importante du projet. Cette section décrit comment WDA a été utilisé pour communiquer avec l'automate et comment il a été intégré dans le programme.

C'est pour utiliser WDA qu'il a été choisi de remplacer l'automate 751-9401 par un 751-9402. L'analyse de la raison de cette décision se trouve Chapitre 7.2. Vous y trouverez également une analyse de comment était récupéré les I/O dans le programme du TB 2024 qui utilisait une méthode plus rapide.

Pour implémenter WDA, il a fallu modifier le programme `backend` (`softplc-main`) et plus particulièrement les fichiers `inputUpdate.go` et `outputUpdate.go`.

Un des principaux problèmes de WDA est la vitesse, qui est très lente. Après une commande GET sur une I/O, cela peut prendre jusqu'à environ 500ms avant qu'on ait la réponse. Sachant qu'on a besoin de faire une requête GET pour chaque Input et Output, cela représente au total 22 requêtes. La première version du programme faisait une requête GET pour chaque Input et Output, ce qui prenait plusieurs secondes pour récupérer l'état de tous les Inputs et Outputs. Pour résoudre ce problème, il a été trouvé la solution d'utiliser une **Monitoring Lists** (Chapitre 7.3). Cela nous permet de récupérer l'état de des I/O en une seule requête GET. Cela permet de réduire le temps de récupération à environ 500ms. Cependant, la **Monitoring Lists** a un temps de vie limité, donc il faut la recréer périodiquement. Cela est fait toutes les 600 secondes dans le programme.

Toutes les requêtes via WDA doivent avoir le header Fig. 24 et l'authentification Fig. 25.

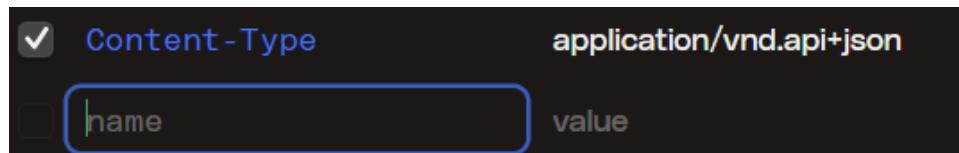


Fig. 24. – Header WDA



Fig. 25. – Authentification WDA

4.1.1 inputUpdate.go

Dans `inputUpdate.go`, l'idée étant de créer et mettre à jour la variable tableau **InputsOutputsState** de structure *InputsOutputs* dont la structure d'un élément est montré Fig. 26, afin de garder le fonctionnement du programme existant, mais cette fois-ci en utilisant WDA.

```

    3 = {inputUpdate.InputsOutputs}
      f FriendlyName = {string} ""
      f Value = {string} "0"
      f Service = {string} "DI1"
      f SubService = {string} ""
      f id = {string} ""

```

Fig. 26. – Exemple d'élément Input/Output State dans le programme `backend`

Au démarrage, `softPLC.go` appelle la fonction `InitInputs` pour créer le client HTTP, appeler la fonction `initClient` et initialiser la variable `InputsOutputsState`, qui est créée par la fonction `mapResultsToInputsOutputs` prenant en paramètre le résultat de la fonction `fetchValues` (Fig. 27), responsable de faire la requête GET afin de récupérer les valeurs des I/O.

La fonction `initClient` a pour rôle de **configurer les DIO** (Chapitre 7.5.1). Par défaut, ils sont configurés en `input` (value = 1 à 8). Il faut donc les configurer en `output` (value = 9 à 16) si on veut avoir que des outputs. De plus, avant de pouvoir lire et écrire des I/O, il faut modifier le **access mode** (Chapitre 7.4) en activant le control mode (value = 2). Par défaut, celui-ci est désactivé (value = 0).

```

results = {map[string]interface{}}
  0 = 0-0-io-channels-22-divalue -> false
    10 key = {string} "0-0-io-channels-22-divalue"
    10 value = {interface{} | bool} false
  > 1 = 0-0-io-channels-24-divalue -> false
  > 2 = 0-0-io-channels-10-doalue -> false

```

Fig. 27. – structure valeur renournée par la fonction `fetchValues`

Ensuite, la fonction `UpdateInputs` est appelée à chaque cycle par `softPLC.go`. La fonction `UpdateInputs` utilise la fonction `updateInputsOutputsState` pour mettre à jour la variable `InputsOutputsState`. La fonction `updateInputsOutputsState` met à jour la variable `InputsOutputsState` avec les nouvelles valeurs récupérées par la fonction `fetchValues` pour récupérer les valeurs des I/O.

En parallèle, la fonction `CreateMonitoringLists` est appelée périodiquement toutes les 600 secondes par `softPLC.go` pour recréer une monitoring lists (Chapitre 7.3).

4.1.2 `outputUpdate.go`

Dans `outputUpdate.go`, l'idée est de mettre à jour les `outputs` après que le programme ait été exécuté. La logique pour déclencher l'envoi des nouvelles valeurs des `outputs` est restée la même que pour l'ancien programme. La différence est que l'on utilise `WDA` pour envoyer les nouvelles valeurs des `outputs`. C'est-à-dire que l'on crée des requêtes `PATCH` pour les `AO` et `DO`, exemple de requête `PATCH` (Chapitre 7.5.2).

4.2 Intégration des blocs logiques simples

L'intégration des blocs logiques simples est une étape importante du projet. Cette section décrit comment ces blocs logiques simples ont été intégrés dans le programme.

4.2.1 Bloc Bool to String

Il correspond au bloc **Pulse to frame Sender** du schéma Fig. 13. Mais il est utilisable pour d'autres fonctionnalités que la communication. Le bloc **Bool to String** permet de transmettre en sortie une chaîne de caractères selon si un booléen est à *true* en face. C'est un bloc de type **stetchable**, c'est-à-dire que le nombre d'entrées est dynamique. La figure Fig. 28 montre un exemple de ce à quoi pourrait ressembler le bloc **Bool to String**. Il est possible d'écrire plusieurs lignes de texte, chaque ligne de texte se trouve en face d'une entrée booléenne. Si l'entrée booléenne est à *true*, la ligne de texte correspondante sera transmise en sortie.

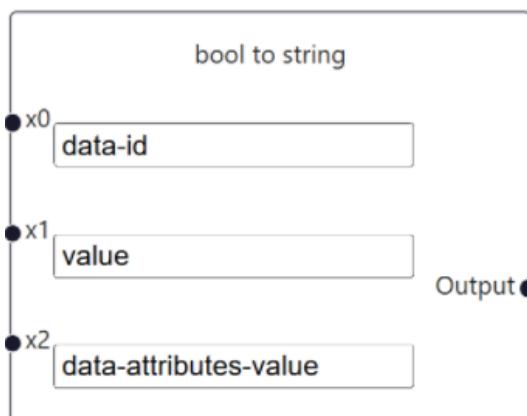


Fig. 28. – bloc Bool to String

i Il est possible d'**activer plusieurs lignes de texte** en même temps. Par exemple, si les entrées booléennes **x0**, **x1** et **x2** sont à *true*, les lignes de texte correspondantes seront transmises en sortie sous la forme d'un tableau. Cependant, la sortie est de type **String**, donc les lignes de texte seront séparées par « „ „ » (deux virgules entourées par des espaces).

4.2.2 Bloc String to Bool

Le bloc **String to Bool** permet de recevoir une chaîne de caractères et de la convertir en booléens. Il est représenté par le bloc **Frame to Pulse Receiver** sur le schéma Fig. 13. Il est également un bloc de type **stetchable** mais au niveau des sorties. La figure Fig. 29 montre un exemple de ce à quoi pourrait ressembler le bloc **String to Bool**. Il est possible d'écrire plusieurs lignes de texte, chaque ligne de texte se trouve en face d'une sortie booléenne. Si la chaîne de caractères reçue contient la ligne de texte correspondante, la sortie booléenne sera à *true*.

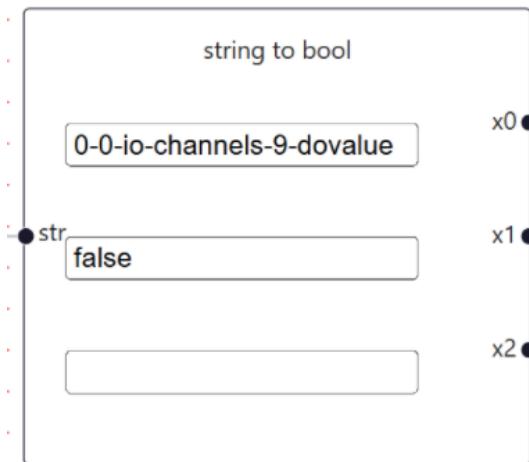


Fig. 29. – bloc String to Bool

Le fonctionnement du bloc **String to Bool** a été réfléchi de manière à ce qu'il soit le plus simple possible, mais pouvant gérer le plus de situations possible. Le schéma bloc simulant toutes les situations se trouve en Fig. 30. Des blocs se trouvent en amont, mais ce qui nous intéresse est ce qu'on retrouve en *input*. Les trois situations de fonctionnement principales sont :

1. Activer un booléen si la chaîne de caractères reçue correspond à une des lignes de texte, (Fig. 31).
2. Traiter un tableau reçu dans son ordre original. Cela est important pour les blocs de communication, car on peut donner le résultat de leurs messages dans le même ordre que l'ordre de demande de requête, ce qui permet de traiter les messages sans se soucier d'avoir plusieurs messages avec les mêmes valeurs, (Fig. 32).
3. Traiter des tableaux comme des caractères simples, (Fig. 33).

Ainsi, il doit être capable de gérer des caractères simples et des pseudos tableaux (lignes de texte séparées par « „ „). Si la chaîne de caractères reçue contient un caractère simple, la sortie booléenne correspondante sera à *true*, pour autant qu'elle existe.

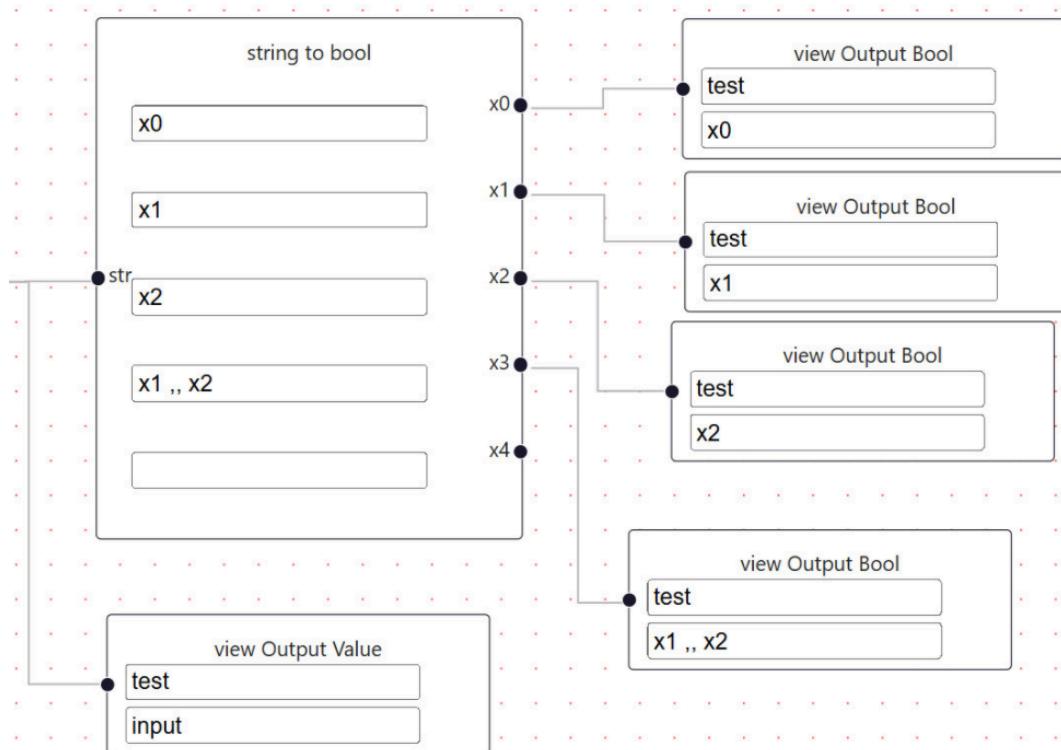


Fig. 30. – exemple - fonctionnement String to Bool

- input : **x0** input : **x1** input : **x2**
- x0 : ON x0 : OFF x0 : OFF
- x1 : OFF x1 : ON x1 : OFF
- x2 : OFF x2 : OFF x2 : ON

Fig. 31. – exemple - fonctionnement String to Bool - situation 1

input : x0 , ABC , x2	input : x0 , x1 , x2	input : x0 .. x1	• input : x0 .. x2
x0 : ON	x0 : ON	x0 : ON	• x0 : ON
x1 : OFF	x1 : ON	x1 : ON	• x1 : OFF
x2 : ON	x2 : ON	x2 : OFF	• x2 : OFF
x1 .. x2 : OFF			

Fig. 32. – exemple - fonctionnement String to Bool - situation 2

input : x1 .. x2	input : x0 .. x1 .. x2
x0 : OFF	x0 : ON
x1 : OFF	x1 : ON
x2 : OFF	x2 : ON
x1 .. x2 : ON	x1 .. x2 : OFF

Fig. 33. – exemple - fonctionnement String to Bool - situation 3

4.2.3 SR

Le bloc **SR** est qui permet de maintenir un état booléen. Il suffit d'une impulsion sur l'entrée **S** (set) pour mettre la sortie à *true* et d'une impulsion sur l'entrée **R1** (reset) pour mettre la sortie à *false*. Le reset à la priorité sur le set, c'est-à-dire que si on a une **S** à *true* et une **R1** à *true*, la sortie sera à *false*.

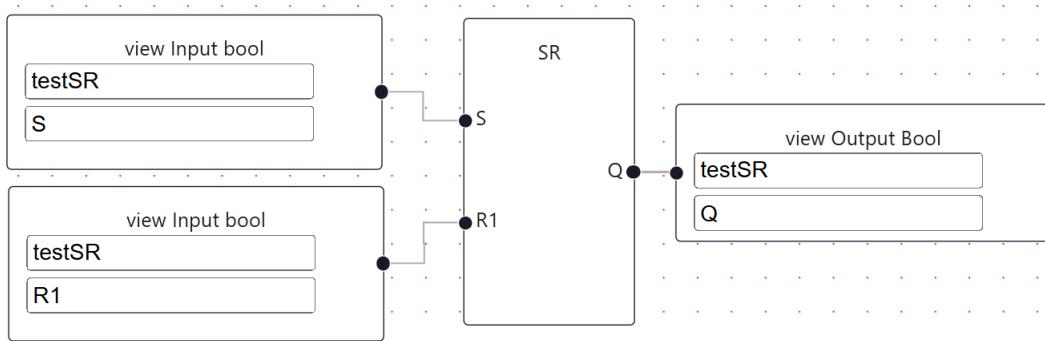


Fig. 34. – exemple - utilisation SR

1) init	2) S = true	3) S = false	4) R = true
Q : OFF	Q : ON	Q : ON	Q : OFF
S : OFF	S : ON	S : OFF	S : OFF
R1 : OFF	R1 : OFF	R1 : OFF	R1 : ON

Fig. 35. – exemple - utilisation SR - visualisation des états

4.2.4 Counter

Le bloc **Counter** permet de compter et décompter des impulsions. Les entrées sont les suivantes :

- **Step** : l'entrée pour choisir le pas d'incrémentation ou de décrémentation du compteur, (par défaut, le pas est de 1).
- **+** : l'entrée pour incrémenter le compteur.
- **-** : l'entrée pour décrémenter le compteur.
- **R** : l'entrée pour remettre le compteur à zéro.

La sortie est **result** qui peut être de format de type **int** ou **float**, c'est le résultat des opérations.

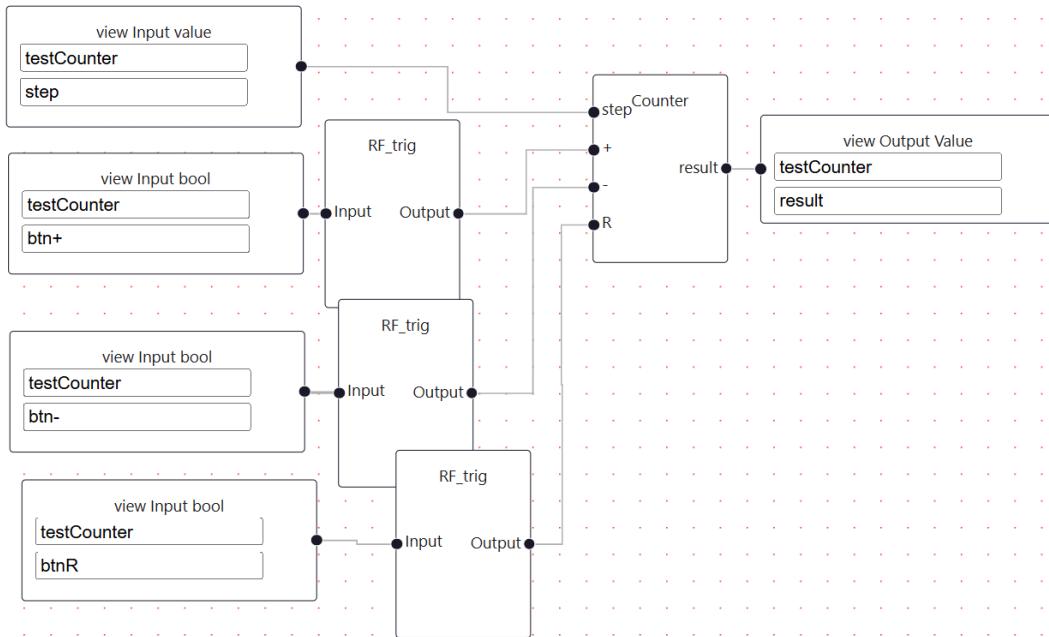


Fig. 36. – exemple - utilisation **Counter**



Noter que l'on utilise des blocs **RF_trig**. Cela permet de générer des impulsions à chaque clic sur les boutons.

4.2.5 Concat

Le bloc **Concat** est utilisable pour assembler des chaînes de caractère de manière dynamique.

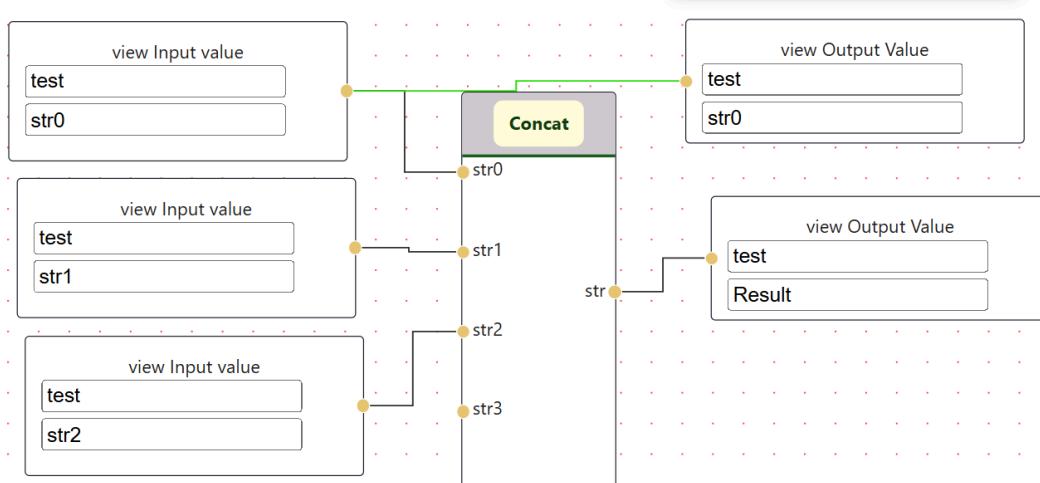
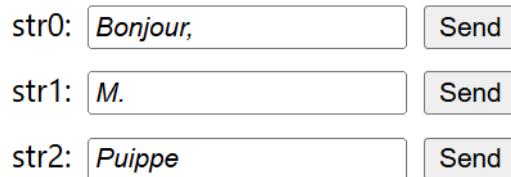


Fig. 37. – exemple - utilisation **Concat**



States :

- Result : **Bonjour, M.Puippe**

Fig. 38. – exemple - utilisation Concat - visualisation des états

4.2.6 Retain value

Le bloc **Retain Value** sert à bloquer une valeur *strIn* et à ne la transmettre sur *strOut* uniquement si *pass* est à *true*.

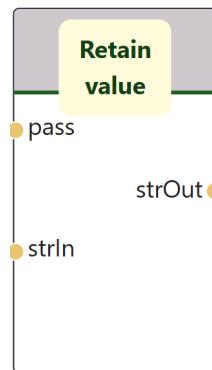
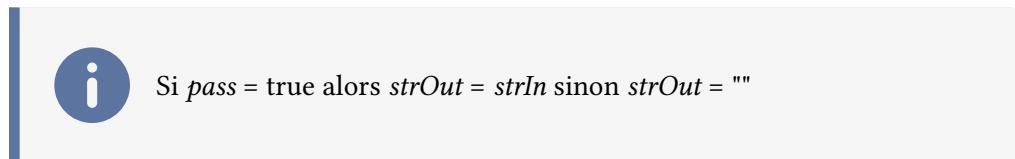


Fig. 39. – bloc **Retain Value**

4.2.7 Comparator EQ

Le bloc **EQ** permet de comparer si deux valeur sont égale et d'activer la sortie si c'est le cas.

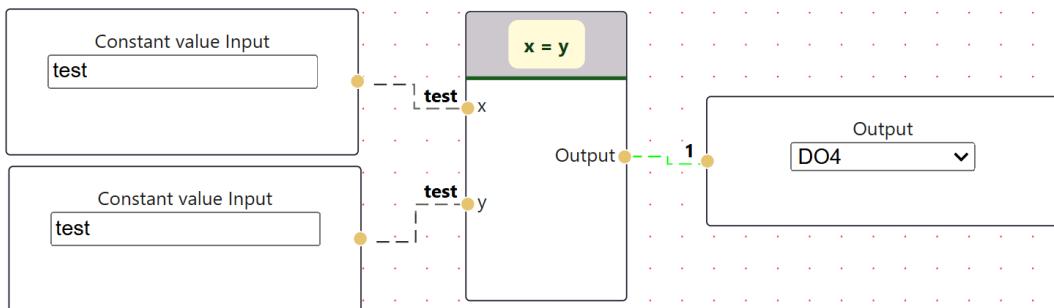


Fig. 40. – bloc **x = y**

4.2.8 Comparator GT

Le bloc **GT** permet de comparer deux valeurs. Si la première valeur est plus grande ou égale, alors la sortie est activée. Le bloc compare les valeurs si elles peuvent être converties en *float*, sinon il compare leur taille (longueur).

La figure Fig. 41 montre la différence entre ces deux cas.

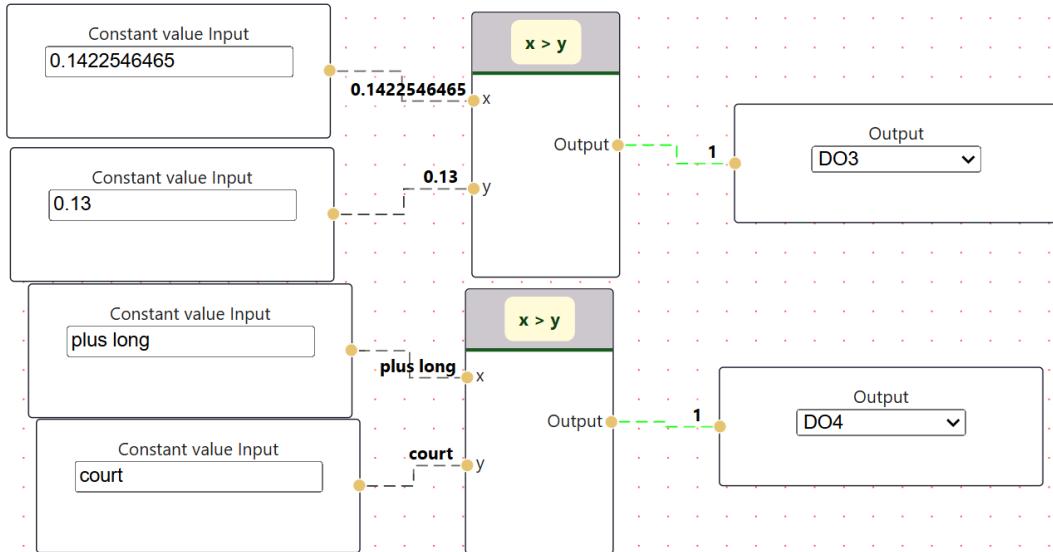


Fig. 41. – bloc **x > y**

4.2.9 Variables

Afin de permettre des fonctions de type boucle de contre-réaction, il a été choisi de rajouter un système de variable. Les blocs Fig. 42 s'utilisent de la manière suivante : il faut faire correspondre les noms pour que les valeurs correspondent. Pour un même nom, une seule **output** doit être définie, mais plusieurs **Input** peuvent porter ce nom.

L'exemple Fig. 43 montre le fonctionnement d'un programme qui, à chaque cycle, change l'état d'un booléen.

L'exemple Fig. 44 montre le fonctionnement d'un programme qui, à chaque cycle, change l'état d'une valeur (**BoolToString_Output**) lorsqu'on a « start ». La séquence est la suivante : « empty » → « Hi » → « Bye » → etc.

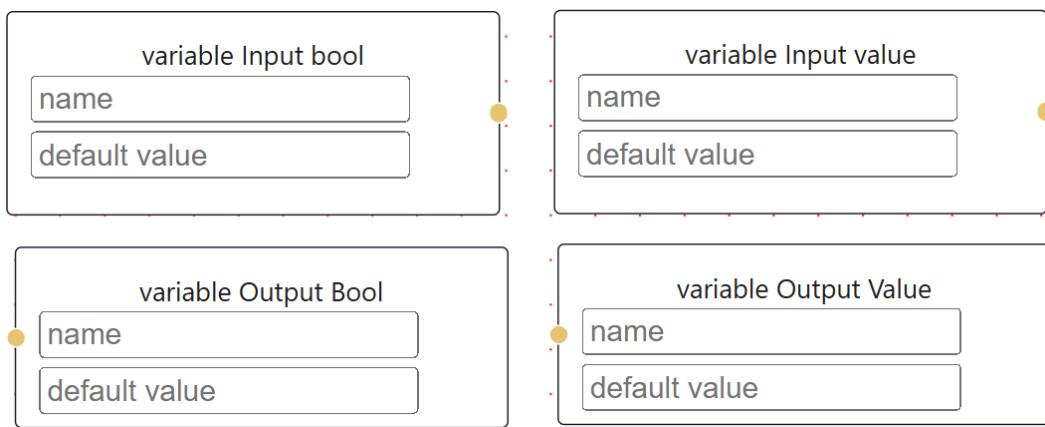


Fig. 42. – blocs variable

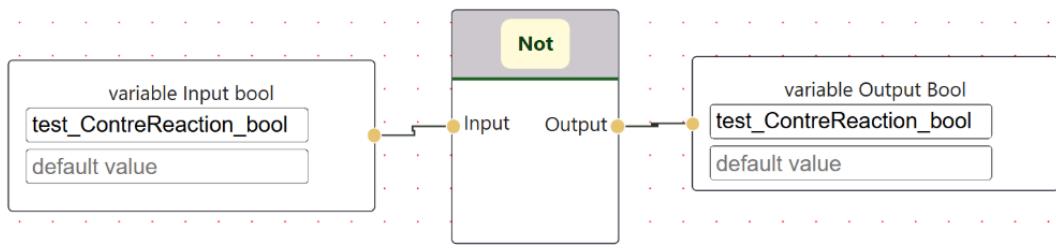


Fig. 43. – exemple - variable bool - contre-réaction

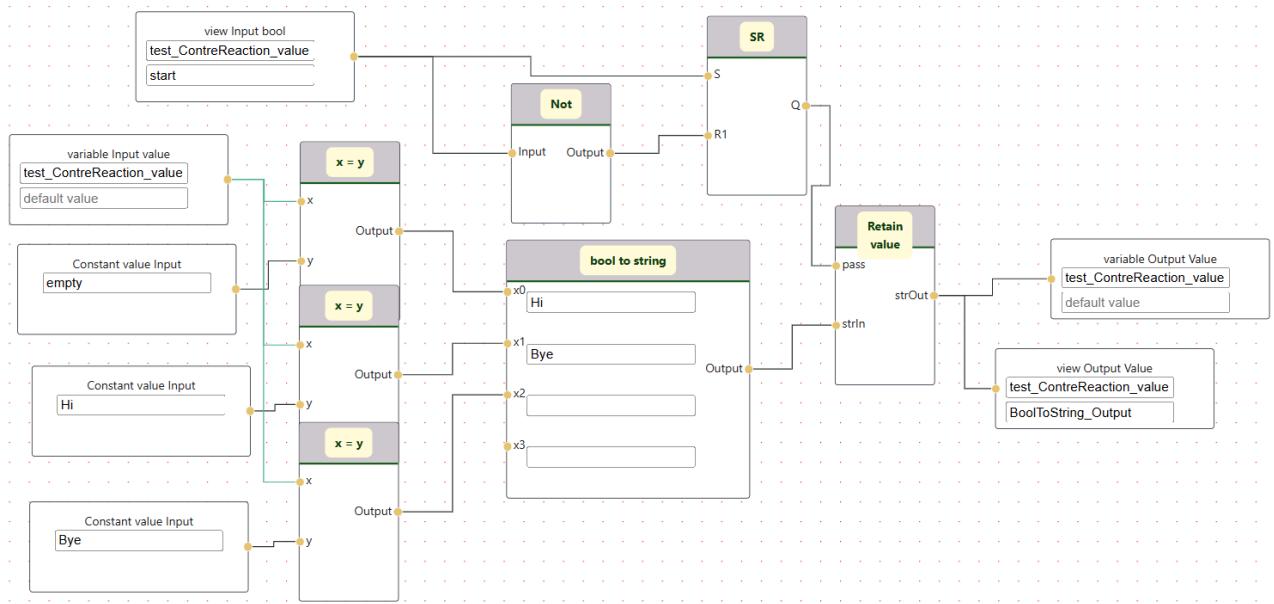


Fig. 44. – exemple - variable value - contre-réaction

4.3 Intégration des blocs logiques de communication

Cette section décrit comment les blocs logiques de communication ont été intégrés dans le programme. Ces blocs peuvent être utilisés pour créer des communications MQTT, HTTP et MODBUS.

Le principe des blocs logiques de communication a été défini dans le schéma Fig. 13. Cependant, il a été adapté aux besoins du projet. En effet, il est parfois nécessaire d'avoir plus d'entrées ou de sorties que ce qui a été défini dans le schéma. De plus, afin d'éviter que l'utilisation ne devienne trop complexe, il a parfois été décidé de séparer le client et le serveur, comme c'est le cas pour HTTP. Toutefois, le principe de base avec l'utilisation de blocs permettant la conversion de booléens en chaînes de caractères et inversement est resté le même. Il a même été élargi pour permettre l'utilisation de constantes, de résultats de concaténation, de variables, etc. En bref, tous les blocs dont la sortie est de type **value** sont compatibles.

La figure Fig. 45 montre la **vue programmation** sans les *settings*. La figure Fig. 46 montre les *settings* dans la **vue programmation**. Il est possible de les ouvrir en cliquant sur le bouton « *settings* » du bloc. Il est également possible d'ouvrir les *settings* de plusieurs blocs en même temps, même s'ils sont du même type. Cela est pratique si l'on souhaite modifier plusieurs blocs logiques de communication simultanément, copier les *settings* d'un bloc à l'autre, ou comparer les *settings* de plusieurs blocs.

Dans la **vue programmation**, les blocs logiques de communication sont définis par le composant **CommunicationHandles** dans le fichier *CommunicationHandles.tsx*. C'est notamment là que l'on retrouve la logique liée à l'affichage de la vue « *settings* ». Un élément React basé sur le composant **CommunicationHandles** est créé dans le fichier *LogicalNode.tsx*, où sont générés dynamiquement tous les *Logical Nodes*. On y calcule également le nombre d'éléments de chaque type à afficher (inputs/outputs).

Le tableau **parameterNameData** contient les noms des *settings* affichés en *placeholder*, et le tableau **parameterValueData** contient les valeurs des *settings* des blocs logiques de communication.

Dans la partie **backend**, les blocs logiques de communication se trouvent à la fin du package *nodes*. Ils ont tous été conçus pour ne pas faire planter le reste du programme en cas de perte de connexion ou de problème de communication. Pour cela, ils utilisent la méthode **go func()** afin de lancer une **goroutine** qui s'occupe de la communication. Cela permet de ne pas bloquer le reste du programme en cas d'erreur de communication.

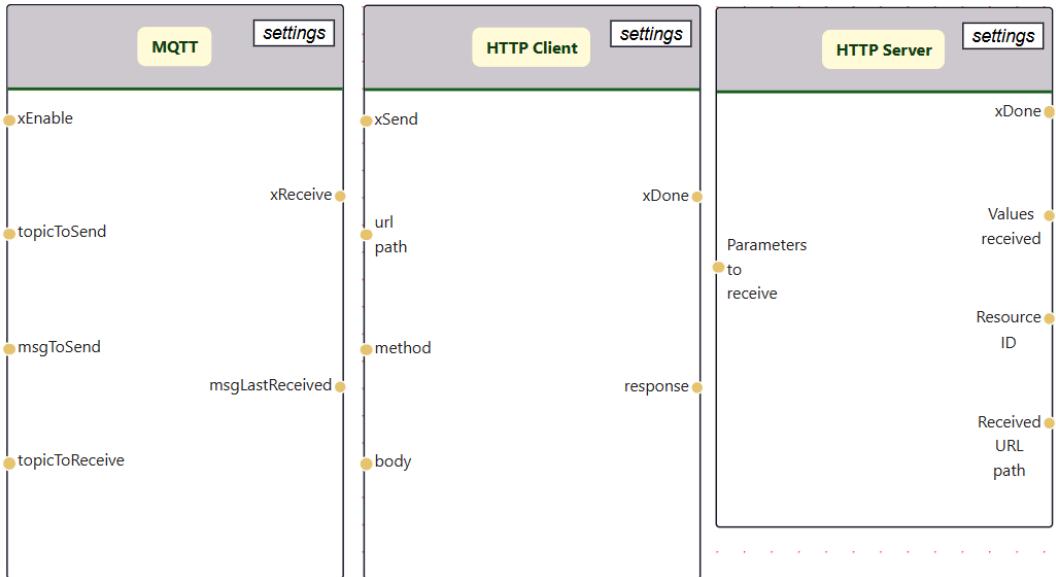


Fig. 45. – blocs logiques complexes - Bloc MQTT, HTTP client et HTTP Serveur

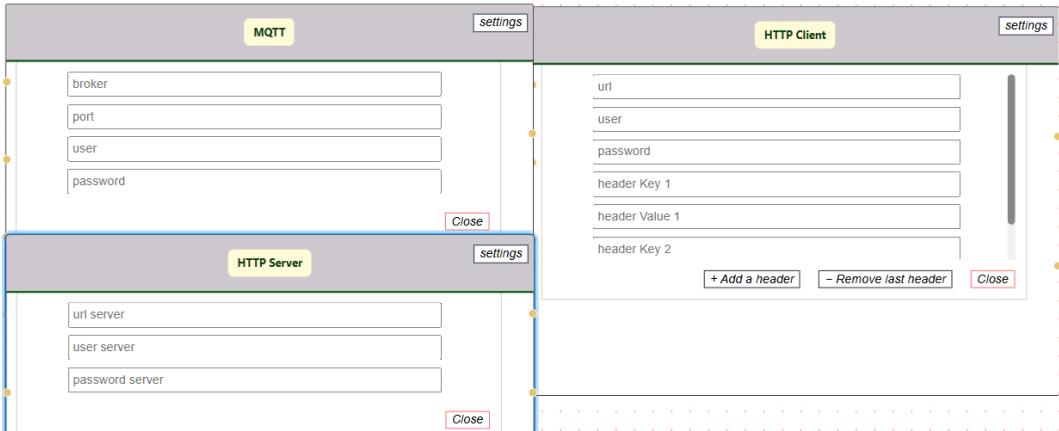


Fig. 46. – blocs logiques complexes - Bloc MQTT, HTTP client et HTTP Serveur - Settings



Fig. 47. – blocs logiques complexes - Bloc MQTT, HTTP client et HTTP Serveur

4.3.1 MQTT

Le bloc **MQTT** permet de communiquer avec un broker MQTT. Il est possible de publier des messages sur des topics ou de s'abonner à des topics pour recevoir des messages.

Le bloc **MQTT** a les **inputs** suivantes :

- **xEnable** : l'entrée pour activer le bloc. Si cette entrée est à *false*, le bloc ne fera rien.
- **topicToSend** : les topics sur lesquels publier (exemple : topic/test1 „ topic/test2).
- **msgToSend** : le message à publier sur les topics de **topicToSend**.
- **topicToReceive** : les topics sur lesquels s'abonner (exemple : topic/test1 „ topic/test2).

Les **outputs** sont :

- **xReceive** : impulsions lorsque le bloc a reçu un message sur un des topics auxquels on s'est abonné.
- **msgLastReceived** : le dernier message reçu sur les topics auxquels on s'est abonné.

Des **exemples** d'utilisations du bloc **MQTT** sont présentés en annexe au Chapitre 8.3.

Lorsqu'un message est reçu, la fonction *messageHandler()* est appelée. Cette fonction s'occupe de ne pas rater de messages, cependant c'est ensuite la fonction *ProcessLogic()* qui s'occupe de traiter les messages reçus, gérer dynamiquement les *subscribes* et *unsubscribes* et écrire les **outputs**.

L'ordre des topics données sur **topicToReceive** est le même que l'ordre des messages reçus sur **msgLastReceived**. Cela permet de traiter les messages dans le même ordre que les topics auxquels on s'est abonné.

La fonction *makeConnectLostHandler(n *MqttNode)* permet de gérer la perte de connexion avec le broker MQTT. Elle s'assure de relancer la connexion et de réabonner aux topics si la connexion a un problème.

4.3.2 Node HTTP client

Le package Go [9] a été utilisé. Pour le Node HTTP client, il est possible de configurer les paramètres suivants :

- **url** : l'URL de la requête HTTP.
- **user** : l'utilisateur pour l'authentification HTTP.
- **password** : le mot de passe pour l'authentification HTTP.
- **Headers** : les en-têtes HTTP à envoyer avec la requête.
 - **key** x : le nom de l'en-tête x.
 - **value** x : la valeur de l'en-tête x.



les *Headers* sont des paires clé-valeur, par exemple : `{"Content-Type": "application/json"}`. Il faut donc deux paramètres pour chaque Header. De plus, il est possible de mettre plusieurs Headers.

Le bloc peut prendre dynamiquement les *inputs* suivantes :

- **xSend** : un booléen pour envoyer lorsque qu'il passe à *true*.
- **url path** : la suite du chemin de l'URL de la requête HTTP. Il est ajouté à la suite du paramètre *URL* pour donner l'URL final.
- **Method** : la méthode HTTP à utiliser (GET, POST, PATCH, PUT, DELETE, HEAD, OPTIONS), par défaut GET.
- **Body** : le corps de la requête HTTP, qui peut être au format JSON ou autre.

Le bloc nous retournera les paramètres suivants :

- **xDone** : un booléen pour indiquer si la requête a été effectuée avec succès.
- **Response** : la réponse de la requête HTTP.

Des **exemples** d'utilisations du bloc **HTTP client** sont présentés en annexe au Chapitre 8.4 qui montre comment l'utiliser avec WDA.

4.3.3 Node HTTP serveur

Le package Go [9] a été utilisé. L'exemple [10] permet d'en comprendre davantage sur la création d'un serveur HTTP en Go. Pour le déploiement d'un serveur HTTP sur Docker, la documentation [11] a été trouvée.

Le Node HTTP serveur permet de créer un serveur HTTP qui écoute les requêtes entrantes. Le but est de pouvoir recevoir une requête venant de n'importe où, par exemple une **appliance** HTTP qui veut activer une sortie automate. Il doit être possible de créer une ressource (POST), de modifier une ressource (PATCH), de lire une ressource (GET) et de supprimer une ressource (DELETE).

Cette ressource peut être créée avec une requête POST sur **http://192.168.39.56:8080/flatten**, par exemple. Cette ressource s'appelle **flatten** car elle permet d'aplatir les données pour les rendre plus digestes pour une utilisation dans le programme de l'automatique. Il est possible de créer plusieurs ressources, mais elles doivent être uniques, c'est pourquoi on retourne un **id** après un POST, suivi de **result:{ ... }** qui contient les paramètres. La figure Fig. 48 montre un exemple de requête POST sur le serveur HTTP.

```
POST ✅ http://192.168.39.56:8080/flatten
Body • ▾
1 ▾ {
2   "param1" : "value1",
3   "param2" : "value2",
4   "param3" : "value3",
5   ▾ "data": {
6     "attributes": {
7       "value": true
8     }
9   }
10 }

Request POST Response 200
HTTP/1.1 200 OK (4 headers)

1 ▾ {
2   "id": 7,
3   "result": {
4     "data-attributes-value": true,
5     "param1": "value1",
6     "param2": "value2",
7     "param3": "value3"
8   }
9 }
```

Fig. 48. – exemple de requête POST sur le serveur HTTP

On doit également pouvoir recevoir des requêtes HTTP qui ne possèdent pas de *body* ou de *headers*. Ce qui est le cas pour les requêtes envoyées par certaines **appliances**, comme les boutons *shelly*, ce qui est possible via l'output **Received URL path**.

Le node utilise **bus.go** pour ne pas perdre d'événements. Les messages stockés dans le bus sont ensuite traités tranquillement dans la fonction *ProcessLogic()* du node.

Les ressources sont stockées dans une variable **storage**, commune à tous les nodes HTTP serveur. L'accès concurrent à **storage** est protégé par un verrou (**sync.Mutex**).

Le node supporte l'authentification HTTP Basic.

Pour le Node HTTP Server, il est possible de configurer les paramètres suivants :

- **url** : l'URL du serveur HTTP (par défaut : localhost:8080).
- **user** : l'utilisateur pour l'authentification HTTP des requêtes autorisées.
- **password** : le mot de passe pour l'authentification HTTP des requêtes autorisées.

Le bloc peut prendre dynamiquement les *inputs* suivants :

- **Parameters to receive** : les paramètres à recevoir dans la requête HTTP. Ils sont séparés par des virgules (exemple : `param1 , , param2 , , param3`). Ils sont liés à la sortie *Values received*.

Les **outputs** sont :

- **xDone** : un booléen pour indiquer si la requête a été effectuée avec succès.
- **Values received** : les valeurs dans le *body* reçues des paramètres donnés dans *Parameters to receive*. Elles sont dans le même ordre que les *Parameters to receive*. C'est-à-dire que si on a `param1 , , param3 , , data-attributes-value` dans **Parameters to receive** et qu'on exécute la requête POST Fig. 48, alors **Values received** sera `value1 , , value3 , , true`. Elle fonctionne également pour les requêtes PATCH et PUT.
- **Resource ID** : l'identifiant de la ressource créée lors d'une requête POST, ou modifiée lors d'un PATCH, ou supprimée lors d'un DELETE.
- **Received URL path** : le chemin de l'URL de la requête HTTP reçue, sans le *host* et le *port*. Par exemple, si on reçoit la requête GET : <http://192.168.39.56:8080/short1>, alors *Received URL path* sera `short1`.

Des **exemples** d'utilisations du bloc **HTTP serveur** sont présentés en annexe au Chapitre 8.5 qui montre comment l'utiliser.

Exemples de requêtes HTTP (ici *localhost:8080* est équivalent à *192.168.39.56:8080*) :

- GET : <http://192.168.39.56:8080/short1>, alors *Received URL path* sera égal à `short1` (Fig. 54).
- POST : <http://192.168.39.56:8080/flatten>, alors une ressource sera créée avec un *id* (Fig. 48).
- PATCH : <http://localhost:8080/parameters/flatten/7> (deux possibilités équivalentes : Fig. 49 et Fig. 50), alors la ressource avec l'*id* 7 sera modifiée.
- GET : <http://localhost:8080/parameters/flatten/7>, alors la ressource avec l'*id* 7 sera lue (Fig. 51).
- DELETE : <http://localhost:8080/parameters/flatten/7>, alors la ressource avec l'*id* 7 sera supprimée (Fig. 52).
- PUT : <http://localhost:8080/message>, alors si les paramètres du *body* sont parmi les *Parameters to receive*, alors ils seront renvoyés dans *Values received* (Fig. 53).

```
PATCH ✎ http://localhost:8080/parameters/flatten/7

Body ● ↴
1 ▼ {
2 ▼   "data": {
3 ▼     "attributes": {
4       "value": false
5     }
6   },
7   "param1": "PatchValue1"
8 }
```

Request PATCH Response 200

HTTP/1.1 200 OK

Connection

Content-Length

Date

Fig. 49. – exemple de requête PATCH sur le serveur HTTP

```
PATCH ✎ http://localhost:8080/parameters/flatten/7

Params Headers 1 Auth ● Body ● ↴
1 ▼ {
2   "data-attributes-value": false,
3   "param1": "PatchValue1"
4 }
```

Request PATCH Response 200

HTTP/1.1 200 OK

Connection

Content-Length

Fig. 50. – exemple de requête PATCH sur le serveur HTTP version 2

```
GET ✎ http://localhost:8080/flatten/7

Auth ● ↴
❶ admin
❷ ..... ↴

Request GET Response 200
HTTP/1.1 200 OK (4 headers)

1 ▼ {
2   "data-attributes-value": false,
3   "param1": "PatchValue1",
4   "param2": "value2",
5   "param3": "value3"
6 }
```

Fig. 51. – exemple de requête GET sur le serveur HTTP pour lire une ressource

```
DELETE http://192.168.39.56:8080/flatten/7

Auth ● ▾
  ↳ admin
    ⏮ •••••
  ↳ Connection
  ↳ Date
```

Request DELETE Response 204
 ▼ HTTP/1.1 204 No Content

Fig. 52. – exemple de requête DELETE sur le serveur HTTP pour supprimer la ressource 7

```
PUT http://localhost:8080/message

Body ● ▾
  ↳ {
    "testPut": "ValueOftestPut"
  }
  ↳ Connection
  ↳ Content-Length
  ↳ Date
```

Request PUT Response 200
 ▼ HTTP/1.1 200 OK

Fig. 53. – exemple de requête PUT sur le serveur HTTP

```
GET http://192.168.39.56:8080/short1

Body ▾
  ↳ 1
  ↳ Connection
  ↳ Date
```

Request GET Response 200
 ▼ HTTP/1.1 200 OK

Fig. 54. – exemple de requête GET sur le serveur HTTP avec un path quelconque

4.3.4 MODBUS

Les documentations utilisées pour la création des blocs MODBUS sont :

- [12] : pour la compréhension du protocole MODBUS.
- [13] : pour comprendre comment l'implémenter en Go.

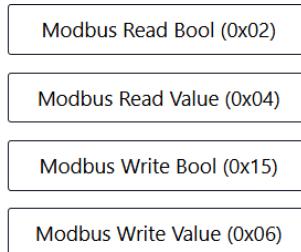


Fig. 55. – Accordion – Modbus

Les blocs Modbus ont été réalisés dans le but de permettre la lecture et l'écriture de toutes les valeurs de *Home-IO*, ce qui peut expliquer certains choix de fonctions utilisées. Les blocs logiques de communication MODBUS développés sont les suivants :

- **Modbus Read Bool** : permet de lire des valeurs booléennes dans un dispositif esclave MODBUS, correspondant au code fonction 0x02. Dans le programme Go, la fonction *ReadDiscreteInputs* est utilisée pour la lecture. La fonction *ReadCoils* n'a pas été retenue car elle ne convient pas à *Home-IO*. Des exemples d'utilisation sont présentés en annexe à la section Chapitre 8.7.
- **Modbus Read Value** : permet de lire des valeurs entières dans un dispositif esclave MODBUS, correspondant au code fonction 0x04. Dans le programme Go, la fonction *ReadInputRegisters* est utilisée. Voir la section Chapitre 8.8 pour des exemples.
- **Modbus Write Bool** : permet d'écrire des valeurs booléennes dans un dispositif esclave MODBUS, correspondant au code fonction 0x15. Dans le programme Go, la fonction *WriteMultipleCoils* est utilisée. Voir la section Chapitre 8.9.
- **Modbus Write Value** : permet d'écrire des valeurs entières dans un dispositif esclave MODBUS, correspondant au code fonction 0x06. Dans le programme Go, la fonction *WriteSingleRegister* est utilisée. Voir la section Chapitre 8.10.

Ces blocs se configurent à l'aide du **host** et du **port** du serveur MODBUS.

Il existe deux types de blocs Modbus : les blocs de lecture (*Read*) et les blocs d'écriture (*Write*). Les deux types possèdent les **inputs** suivantes :

- **xEnable** : permet d'activer le bloc.
- **UnitID** : identifiant de l'esclave MODBUS auquel accéder.
- **Adresses** : adresses des registres à lire ou écrire. On peut spécifier plusieurs adresses séparées par des virgules (ex. : 0,,2,,4). Il faudra définir *Quantity* pour la lecture ou fournir *NewValues* pour l'écriture. Le comportement de ces blocs dépend de la relation entre **Adresses** et **Quantity** ou **NewValues** (Fig. 56 et Fig. 57).

Les blocs de lecture ont également l'**input** :

- **Quantity** (Fig. 56) : nombre de registres à lire. Par défaut, cette valeur est fixée à 1. Plusieurs valeurs peuvent être fournies, séparées par des virgules (ex. : 0 , , 2 , , 4).

Les blocs d'écriture ont également l'**input** :

- **NewValues** (Fig. 57) : valeurs à écrire dans les registres. Plusieurs valeurs peuvent être fournies, séparées par des virgules (ex. : 0 , , 2 , , 4).

Et les **outputs** suivantes :

- **xDone** : activée si la communication avec l'esclave est établie et qu'aucune erreur ne s'est produite.
- **ValuesReceived** : valeur(s) reçue(s) pour les blocs de lecture. Pour les blocs d'écriture, cette sortie contient la réponse du serveur, et peut aussi signaler les erreurs de communication.

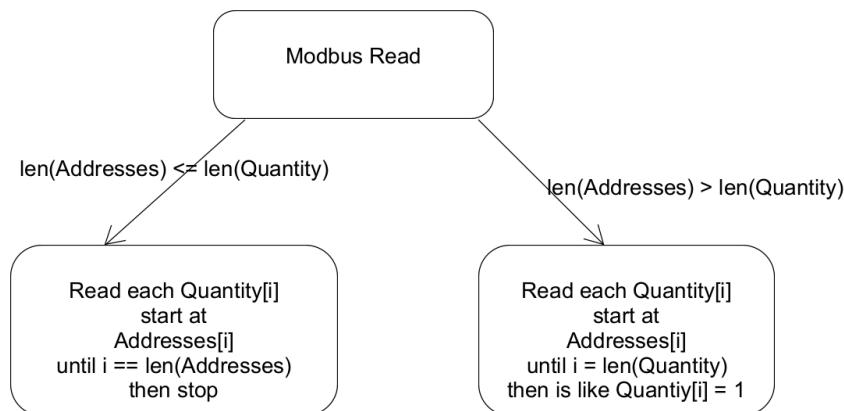


Fig. 56. – Modbus Read – Comportement en fonction de la relation entre la quantité (**Quantity**) et le nombre d'adresses (**Addresses**)

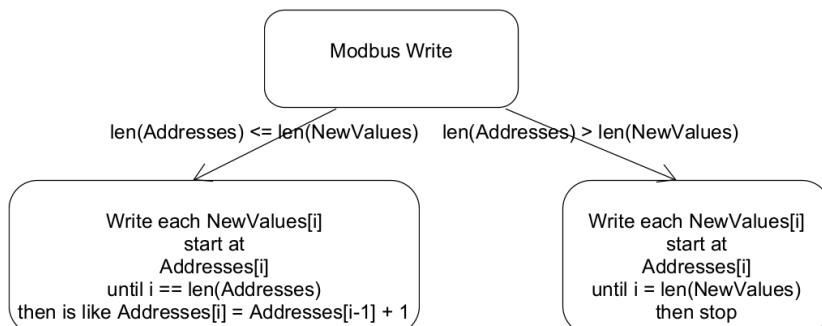


Fig. 57. – Modbus Write – Comportement en fonction de la relation entre **NewValues** et le nombre d'adresses (**Addresses**)

4.4 Vue programmation

La vue programmation permet de créer des programmes PLC en utilisant une interface graphique. Elle est représentée par la « Programming Page » sur le schéma Fig. 12. Cette vue permet de créer par **drag and drop** des blocs logiques, des **Inputs** et des **Outputs**, et de les connecter entre eux pour créer un programme PLC.

Du côté **frontend**, la majorité de la logique est centralisée dans le fichier *App.tsx*. Les fichiers *InputNode.tsx*, *LogicalNode.tsx* et *OutputNode.tsx* jouent également un rôle important, car ils gèrent l'affichage des *nodes* selon leur *primaryType*.

On retrouve ensuite différents fichiers dans le dossier « handles », dédiés à des types particuliers, comme par exemple les *nodes* de communication.

La section Chapitre 4.7 décrit plus en détail les aspects liés à l'interface visuelle.

Tous les nodes réalisés sont présentés en annexe au Chapitre 10.

4.5 Vue User

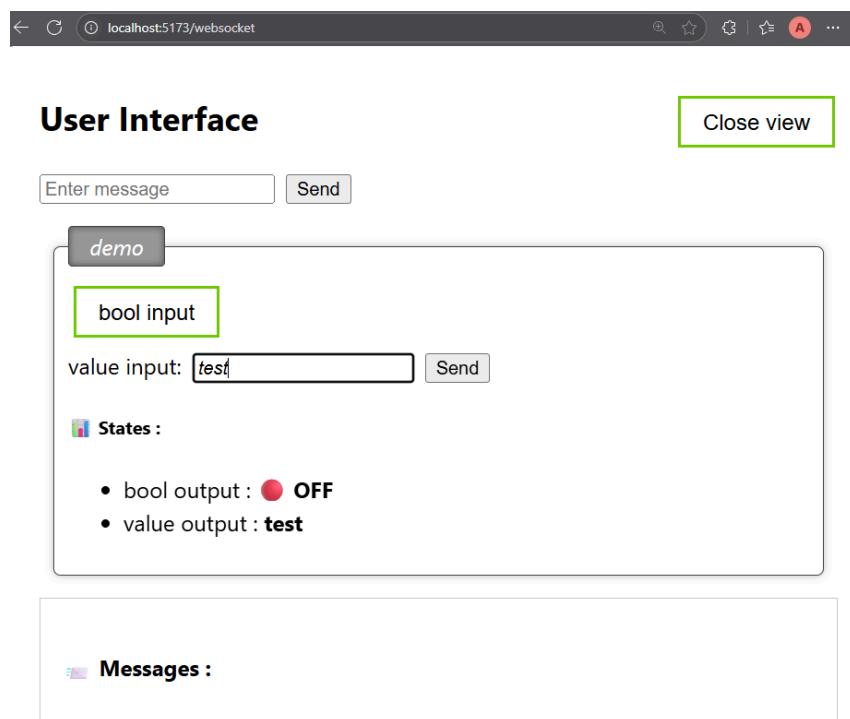


Fig. 58. – vue User : bref aperçu

Le rôle de cette vue a déjà été expliqué Chapitre 3.4.1. Elle est implémentée côté **backend** dans **serverWebSocket.go**, suivant les schémas du chapitre précédent en Fig. 21 et Fig. 22.

Du côté **frontend**, la vue est créée dans le fichier **user.tsx** du dossier **webSocketInterface**. C'est là qu'on gère l'affichage des éléments triés dans les appliances.

Pour passer d'une vue à l'autre, on utilise react-router-dom [14] : pour naviguer entre les routes (avec **useNavigate**).

```

const navigate = useNavigate();
const openView = () => { navigate('/websocket'); };
...
const navigate = useNavigate();
const goBackView = () => { navigate(-1); };

```

Liste 2. – **Vue User** : navigation entre les pages

4.6 Vue mode debug

Le rôle de cette vue a déjà été expliqué Chapitre 3.4.2. Elle est implémentée côté **backend** dans **serverWebSocket.go**, suivant les schémas du chapitre précédent Chapitre 3.5.3.

Après avoir dû déboguer avec cet outil, il a finalement été choisi d'afficher la valeur des connexions par défaut, et que l'outil **display connection** permette de cacher celles qui prennent trop de place, par exemple. Cependant, côté **backend**, dans la fonction *DebugMode*, il y a la première version qui fait l'inverse en commentaire.

Les outils communiquent avec le **backend** par la fonction *handleIncomingMessage*, responsable de recevoir les messages *webSocket*. C'est à ce moment-là qu'on ajoute ou enlève les **edges** de la variable *toDebugList*.

Les connexions reçoivent l'animation « dash 1s linear infinite » pour donner l'impression d'un flux de données.

Du côté **frontend**, la vue est créée grâce au fichier **debug.tsx** du dossier **webSocketInterface**.

La vue debug est également réduite comparée à la vue programmatation, en utilisant une sous-classe *css* « *hide-when-debug* ».

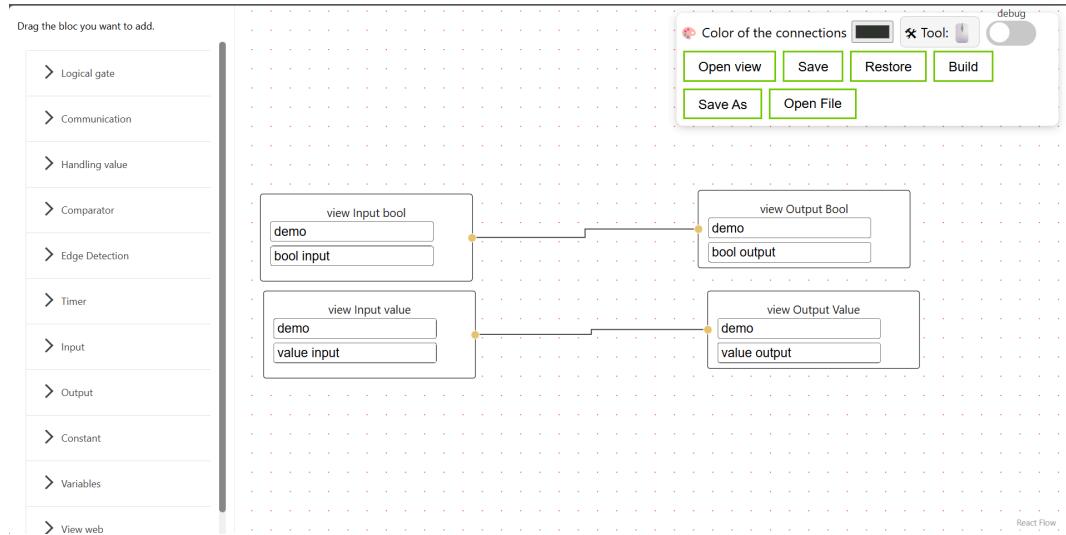


Fig. 59. – vue programmatation : plus d'éléments visible que vue debug

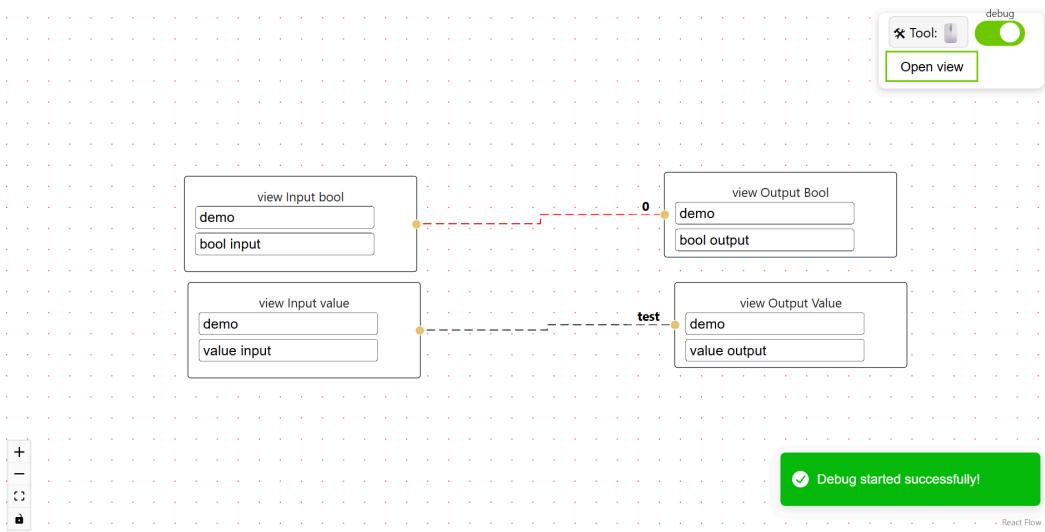


Fig. 60. – vue debug : moins d’éléments visible que vue debug



Remarquez que les connexions (edges) sont bien différentes en mode debug. Il a donc fallu créer un nouveau type d’edge, car le type « step » défini par la librairie ne suffisait plus. Ce type a été créé dans le fichier ***CustomEdgeStartEndDebug.tsx***.

La Fig. 61 montre comment les données se transmettent entre le côté **frontend** et le côté **backend**. Cela permet notamment de comprendre comment le passage d’un graphique à l’autre est effectué.

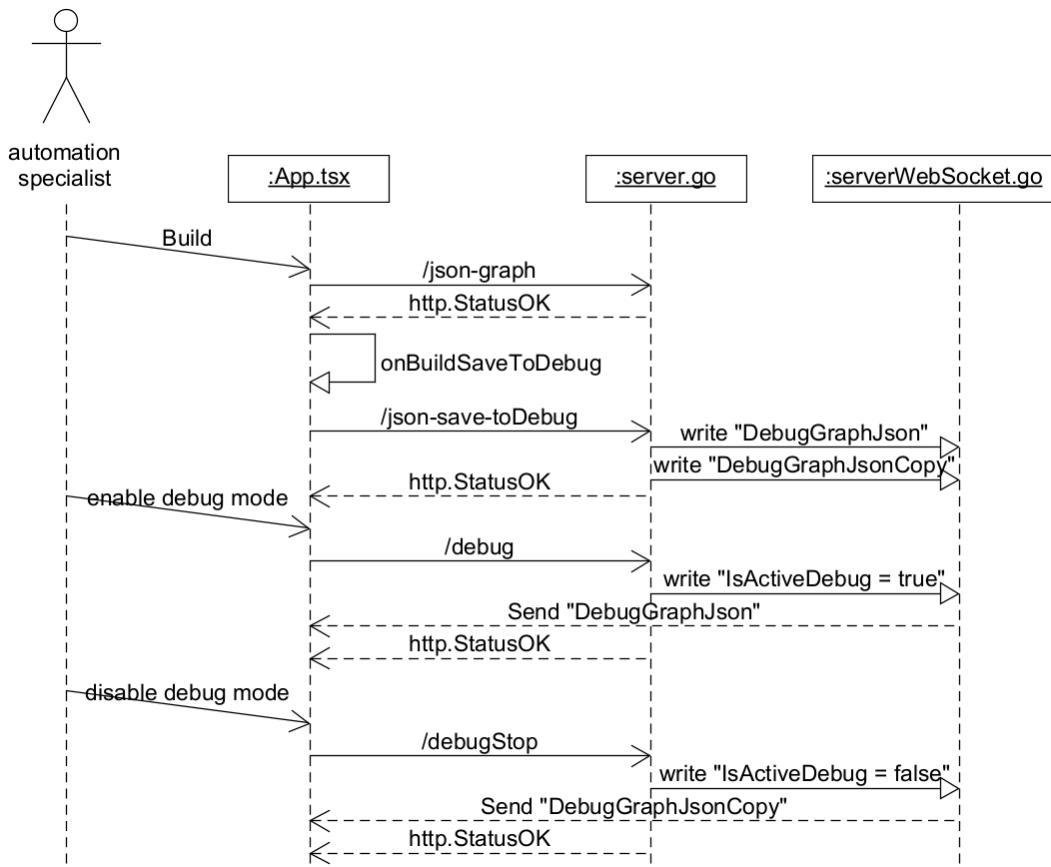


Fig. 61. – vue debug : Diagramme de séquence - Utilisation normal - lien entre **frontend** et **backend**

4.7 Améliorations interface

4.7.1 Rajouter des raccourcis

Le fichier `useKeyboardShortcuts.tsx` a été créé pour l'occasion. Il est appelé dans `App.tsx`. Les raccourcis qui ont été rajoutés sont :

- `ctrl + c` : copie les nodes/edges sélectionnés
- `ctrl + v` : coller
- `ctrl + x` : couper
- `ctrl + z` : annule la dernière modification (undo)
- `ctrl + y` : rétablit la modification annulée (redo)

undo / redo : Le principe est d'avoir deux piles `redoStack` et `undoStack`. On utilise `pushToUndoStack()` pour créer une pile, et `useDebouncedUndo.tsx` qui vérifie lorsqu'il y a des modifications et utilise un petit délai pour éviter de pousser plusieurs fois à cause d'une modification mineure survenant au même moment.

4.7.2 Nodes

Un node standard est constitué de trois éléments principaux, comme le montre Fig. 62. Pour chacun de ces éléments, une classe CSS a été créée. Il est également possible de spécifier plus particulièrement pour des blocs un peu plus complexes, comme le montre Liste 3, par exemple pour agrandir légèrement pour les blocs de communication. Cette structure permet de changer les couleurs et tailles pour chaque type de node.

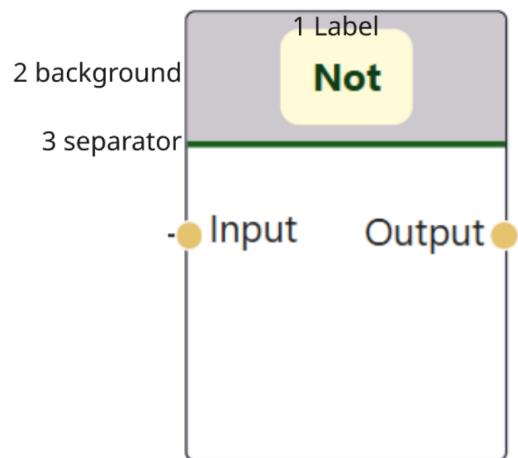


Fig. 62. – exemple - node standard

```
/*Communication*/
.ntb-Communication{
  height: 60px;
}
.react-flow__node .dl-Communication{
  top: 15px;
}

.ns-Communication{
  top: 60px;
}
```

Liste 3. – **css**, exemple pour agrandir légèrement pour les blocs de communication

Un autre type de node Fig. 63 est celui avec un « menu déroulant ». Ils sont conçus pour changer le *Node* de manière rapide sans refaire les connexions. Cela est possible uniquement si les nodes ont les mêmes entrées et sorties, et qu'elles sont du même type. Cela est donc possible pour les *timer* et les *trigger*.

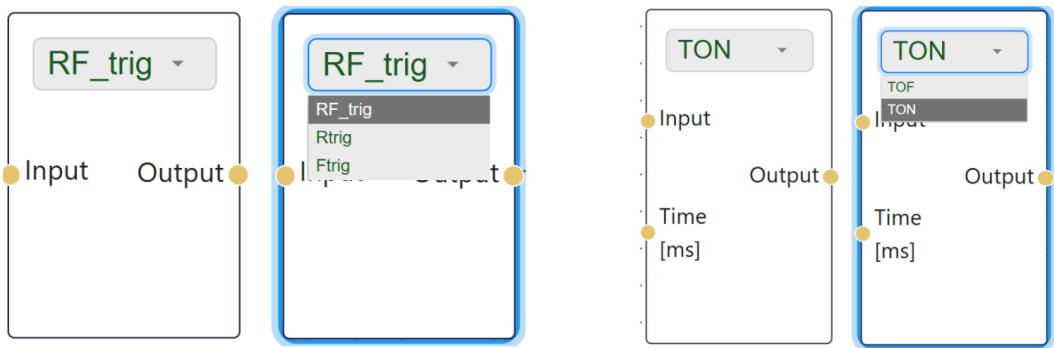
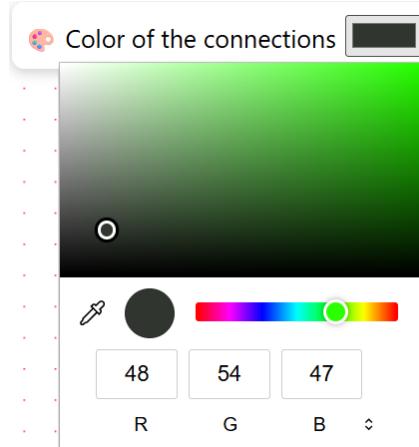


Fig. 63. – Node avec menu déroulant

4.7.3 Couleurs de connection dynamique

Pour que l'utilisateur puisse améliorer la lisibilité, il a été rajouté un outil permettant de choisir la couleur des connexions sélectionnées. Cet outil a été inspiré de « Custom Nodes » [15], qui montre un exemple d'utilisation de l'*input* de type *color* Fig. 64.

Fig. 64. – outil modification couleur (*input* de type *color*)

4.7.4 Elément sélectionnées

Quand une connection est sélectionnée, elle devient plus large Fig. 65. C'est le meilleur critère d'apparence qu'on peu modifier pour garder la même couleur.



Fig. 65. – connection : select VS not select

Quand un Node est sélectionné, il devient coloré sur le tour avec une légère ombre Fig. 66.

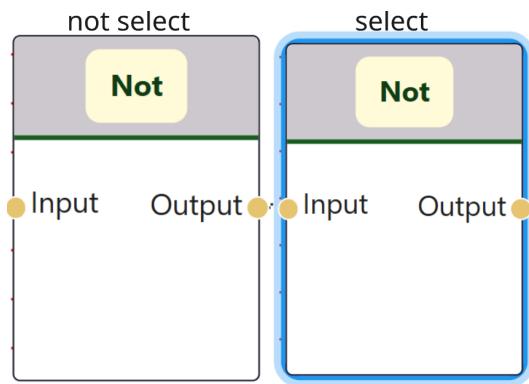


Fig. 66. – Node : select VS not select

4.7.5 Accordion - css

Drag the bloc you want to add.

```

/* scrollbar */
 Sidebar::-webkit-scrollbar
Sidebar::-webkit-scrollbar-thumb
Sidebar::-webkit-scrollbar-thumb:hover
Sidebar::-webkit-scrollbar-track

```

```

.dndflow .dndnode
.dndflow .dndnode:hover

```

Fig. 67. – Accordion - css sélecteurs

4.7.6 Boutons

L'exemple de [16] a été utilisé. Plusieurs classes de boutons ont été définies dans le css (Liste 5). Le sélecteur de classe de base est « .button ». Un exemple d'utilisation serait :

```
<button className={"button button1"} onClick={openView}>Open view</button>
```

Tous les styles définis dans les deux sélecteurs de classe CSS (dans l'exemple `.button` et `.button1`) correspondants seront cumulés, et les styles de `.button1` peuvent écraser ceux de `.button` s'ils affectent les mêmes propriétés.

On peut utiliser les sélecteurs définis dans Liste 5, de la même manière que `.button1`. Ils sont représentés en Fig. 68.

```
/*Buttons*/
.button {
    background-color: #04AA6D; /* Green */
    border: none;
    color: white;
    padding: 8px 16px;
    text-align: center;
    text-decoration: none;
    display: inline-block;
    font-size: 16px;
    margin: 4px 4px;
    transition-duration: 0.4s;
    cursor: pointer;
}
.button1 {
    background-color: white;
    color: black;
    border: 2px solid #04AA6D;
}

.button1:hover {
    background-color: #04AA6D;
    color: white;
}

.buttonNode {
    background-color: white;
    color: black;
    border: 1px solid #1a192b;
    font-style: italic;
}
.buttonNode:hover {
    background-color: #1a192b;
    color: white;
}
```

Liste 5. – **css**, Bouttons

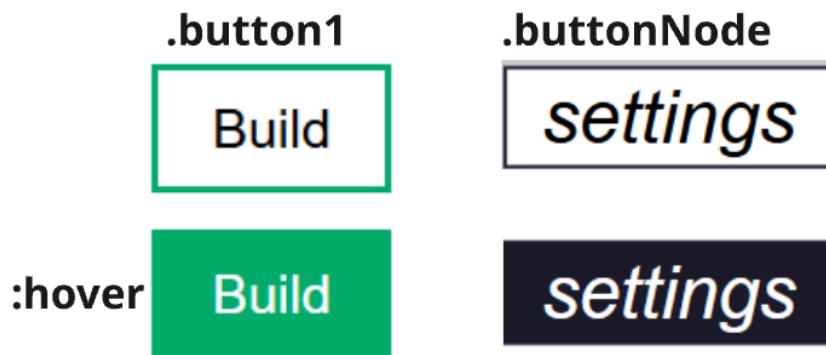


Fig. 68. – exemple - bouttons - visualisation

4.7.7 Tools

Pour permettre de modifier la manière d'interagir avec le graphique, un menu déroulant appelé **Tool** a été ajouté. Il permet de choisir entre différents outils.

Plusieurs outils ont été définis :

- **DisplayConnectionDebug** : permet de sélectionner les connexions à afficher en mode debug.
- **comment** : permet d'ajouter un commentaire à l'endroit où l'on clique.

Les outils peuvent fonctionner de deux manières :

- En récupérant les informations depuis le **backend**, comme illustré en Liste 6.
- En les utilisant directement depuis le **frontend**, comme illustré en Liste 7.

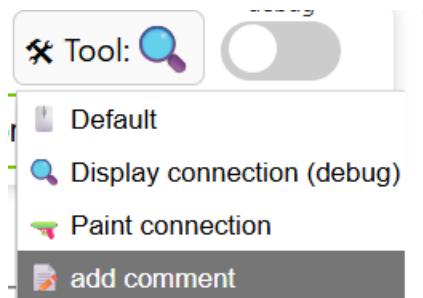


Fig. 69. – menu déroulant **Tool**

```
map[source:40 sourceHandle:Output tool:DisplayConnectionDebug
type:edge_clicked]
```

Liste 6. – **Message WebSocket au backend** : clic effectué sur un edge avec l'outil « `DisplayConnectionDebug` »

```
/* add comment */
const onPaneClick = useCallback(
  (event: React.MouseEvent) => {
    if (tool === 'comment') {
      ...
    }
  }
)
```

Liste 7. – **Extrait code App.tsx** : Exemple utilisation de l'outil « `comment` »



Fig. 70. – **Tool** : exemple commentaires

4.8 Gestion des erreurs

Pour envoyer un message d'erreur sur la page de programmation, il faut utiliser dans le programme : **serverResponse.ResponseProcessGraph** = « message à envoyer ». Cela doit se faire avant la fin de la vérification, donc pour mettre des messages pour un **Node** précis, il faut utiliser l'appel dans la méthode **GetOutput** de celui-ci. C'est ce qui a été utilisé pour Chapitre 4.8.2.

4.8.1 Outputs

Il est interdit d'avoir deux fois la même *Output*, même si le type est le même.

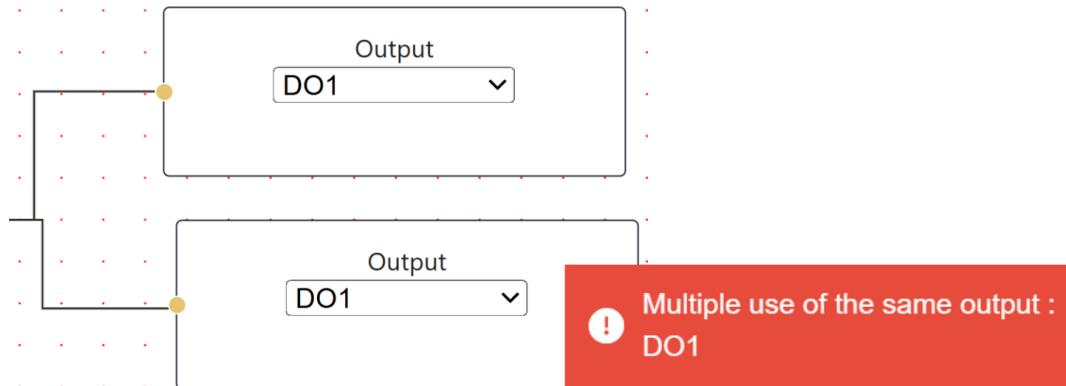


Fig. 71. – exemple - erreur **Output**

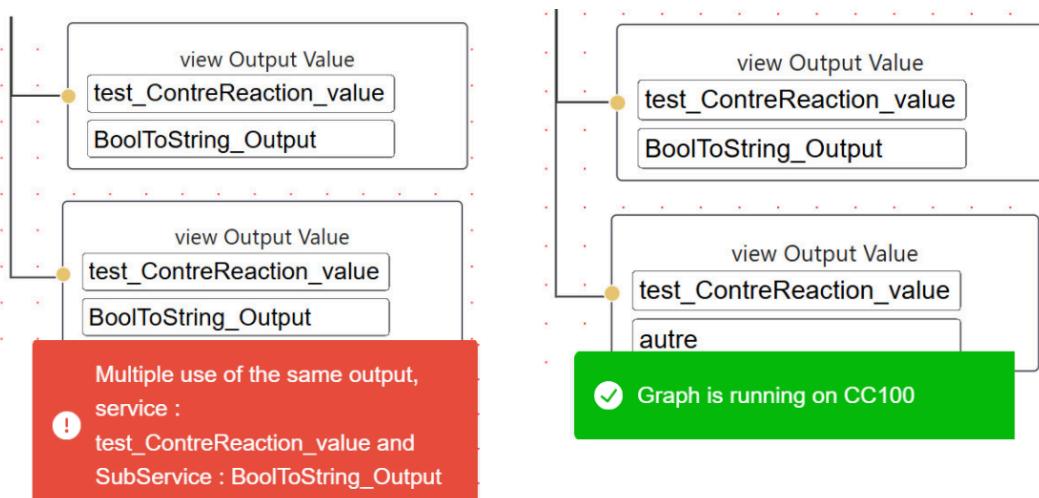


Fig. 72. – exemple - erreur **view Output**

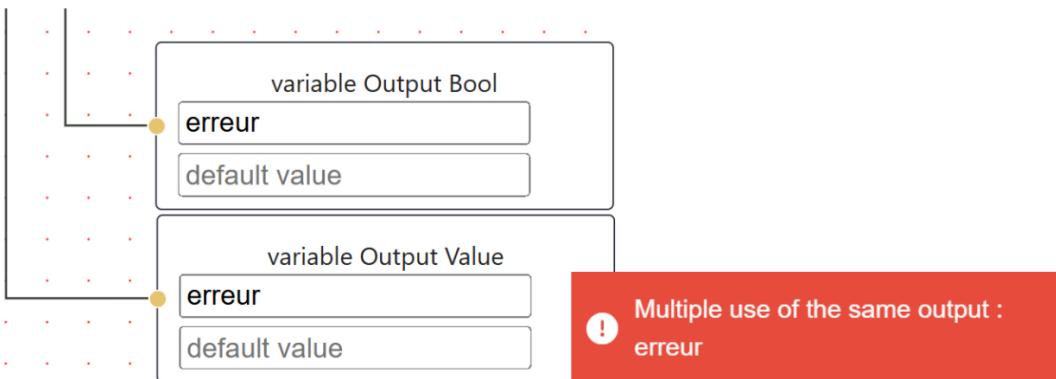


Fig. 73. – exemple - erreur variable Output

4.8.2 Timer

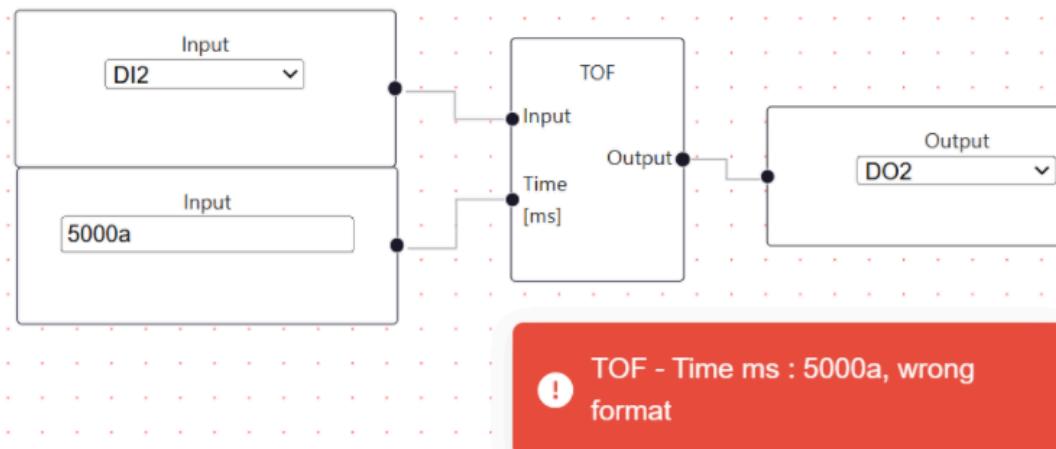


Fig. 74. – exemple - erreur TOF

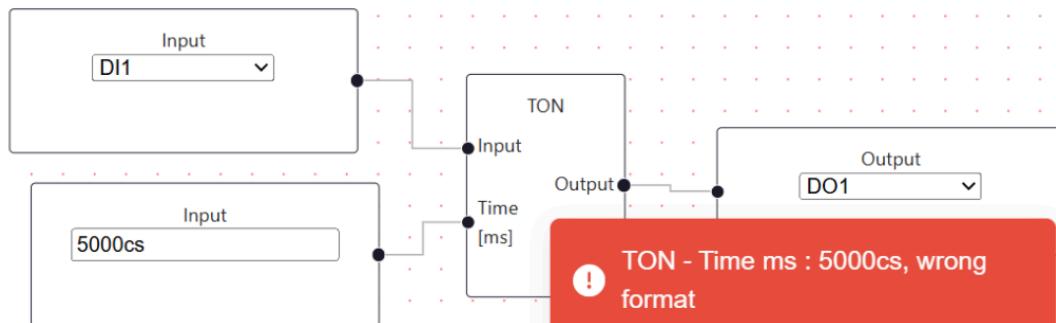


Fig. 75. – exemple - erreur TON

4.9 Conclusion

5 | Validation

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliquam quaerat voluptatem. Ut enim aequre doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut.

Contenu

5.1 WDA défauts	61
5.2 Proof of Concept : maison intelligente	62
5.3 Conclusion	63

5.1 WDA défauts

Le principal défaut de WDA, en plus d'être lent, est qu'il n'est pas possible d'écrire plusieurs **outputs** en une seule requête. Il est donc nécessaire de faire une requête pour chaque **output** que l'on souhaite écrire, ce qui ajoute environ 500 ms à chaque fois.

Cela peut être problématique si l'on souhaite écrire plusieurs **outputs** en même temps, car cela ralentit le temps de cycle. Par exemple, si l'on souhaite écrire 8 **outputs**, il faudra faire 8 requêtes, ce qui ajoutera environ 4 secondes au temps de cycle. À cela s'ajoutent le temps de la requête pour lire les **inputs**, ainsi que le temps de la requête pour la création de la **monitoring list**.

On ne peut donc pas garantir un temps de cycle inférieur à 5 secondes, ce qui est problématique pour une application qui demande de la rapidité ou un temps de cycle précis. La figure Fig. 76 présente le programme qui prend le plus de temps, et la figure Fig. 77 montre le temps de cycle de ce programme selon les étapes effectuées. On remarque de grandes variations du temps de cycle.

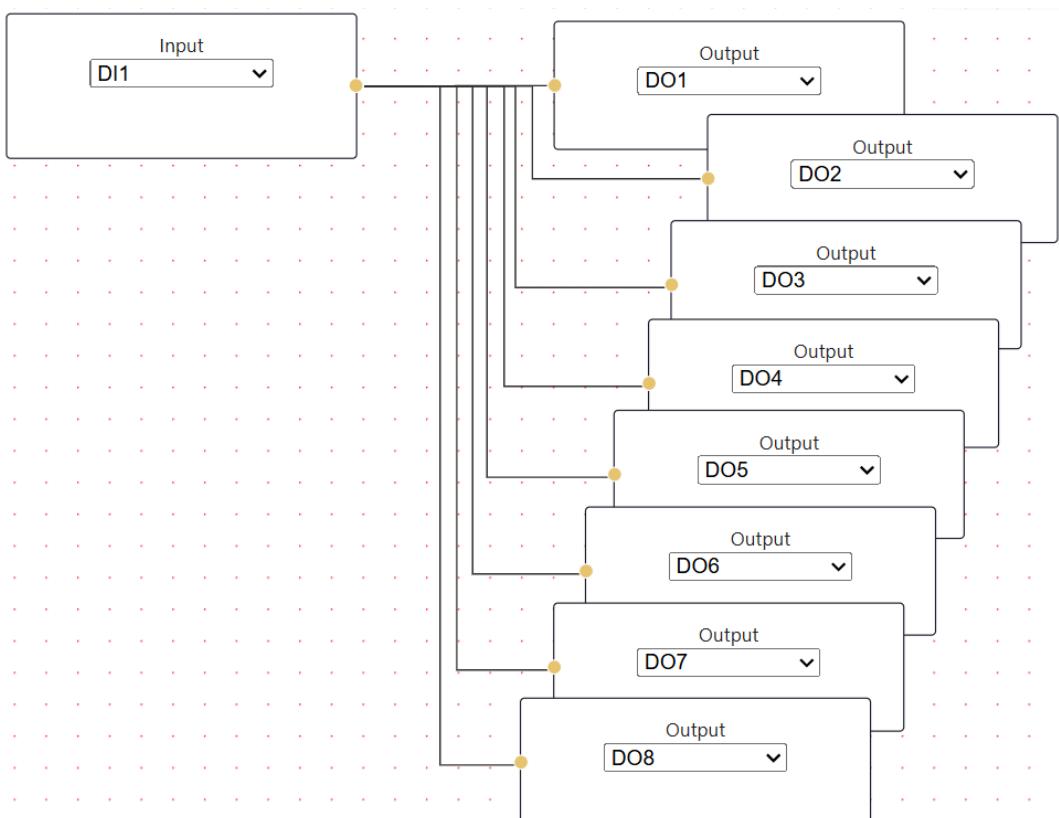


Fig. 76. – Programme - 8 outputs

```

total time value is 462.785ms
total time value is 470.7045ms
CreateMonitoringLists is completed
total time value is 1.2104112s
total time value is 464.4686ms
total time value is 472.6222ms
total time value is 465.091ms
total time value is 476.8153ms
total time value is 467.5461ms
CreateMonitoringLists is completed
total time value is 5.0711889s
CreateMonitoringLists is completed
total time value is 662.6448ms
total time value is 467.0168ms
total time value is 575.5504ms

```

Création Monitoring Lists
=> généralement entre 610 ms et 1s

Temps pour l'activation des 8 outputs
=> généralement entre 4.5 s et 5.5 s

Fig. 77. – Analyse programme - 8 outputs

5.2 Proof of Concept : maison intelligente

Le programme de la maison intelligente est un exemple d'application développé grâce aux nouvelles fonctionnalités mises en œuvre durant ce TB. Il démontre les possibilités de connectivité entre des appareils utilisant différents protocoles de communication, et prouve que le point principal du cahier des charges est respecté.

Cet exemple permet de contrôler la porte du garage, le chauffage et les lumières d'une partie de la maison. L'utilisateur peut paramétrier différents éléments, comme la couleur et la luminosité de la lampe *Shelly*, l'allumage manuel des lampes, ainsi que les consignes et l'activation du chauffage.

Le schéma des différents appareils connectés a déjà été défini en Fig. 2.

Le programme est présenté en annexe au Chapitre 9. Il est divisé en plusieurs parties, chacune correspondant à un élément de la maison.

La section Chapitre 9.1 présente la logique pour recevoir les commandes des boutons *Shelly* grâce au bloc **HTTP serveur**.

La section Chapitre 9.2 détaille le programme responsable de l'enclenchement du chauffage. Il est possible de le contrôler manuellement grâce à l'interface utilisateur, ou automatiquement en fonction de la température.

La section Chapitre 9.3 décrit le programme de la porte du garage. Celle-ci peut être ouverte ou fermée grâce à un bouton déclenchant une requête HTTP.

La section Chapitre 9.4 présente le programme de la lampe *Shelly Bulb*. L'utilisateur peut en contrôler la couleur et la luminosité via l'interface utilisateur. Il est également possible de l'allumer ou de l'éteindre manuellement depuis cette même interface.

La section Chapitre 9.5 présente le programme des autres lumières. Il permet de contrôler les éclairages selon l'emplacement dans la maison : un appui court sur le bouton *Shelly* allume les lampes de la pièce où l'on se trouve, tandis qu'un appui long les éteint toutes. Cette fonctionnalité est uniquement disponible pour les pièces **Kitchen** et **Garage**.

Bloc	Appareil	Adresse IP	Rôle	documentations
MODBUS	Home IO	localhost:1502	Simulateur de la maison connectée	Aperçu Chapitre 8.6
HTTP Server	Shellybutton	192.168.39.225	Ouverture/fermeture porte du garage	Manuel [17]
HTTP Server	Shellybutton	192.168.39.226	Lampes monitoring	Manuel [17]
HTTP Client	Shelly Bulb Duo RGBW	192.168.39.223	Lampe de couleur en physique	Modèle [18] Manuel [19]
HTTP Client	My Strom	192.168.37.59	Relais chauffage	Chapitre 11.1
MQTT	Shelly H&T	192.168.39.224	Capteur de température	Manuel [20] Doc MQTT [21]

5.3 Conclusion

6. Conclusion

6.1. Résumé du projet

L'automate a été câblé et configuré. Il est prêt à être utilisé pour la suite du projet. Les programmes softplc-main et softplcui-main ont pu être testés et fonctionnent comme décrit dans le travail précédent de TB. Toutefois, la partie analogique n'a pas été testée, mais elle ne semble pas fonctionnelle, car elle ne figurait pas parmi les points traités dans le TB précédent.

Par ailleurs, le bloc Appliance Input ne fonctionne pas et fait planter l'interface. De nombreuses améliorations, décrites dans la partie « Chapitre 6.4.1 », restent possibles.

6.1.1. Fonctionnalités développées :

- Un nouveau bloc timer de type TOR a été ajouté. Cela prouve la faisabilité de l'ajout de blocs simples et montre que le programme est à la fois robuste et adaptable.
- Ajout de la fonctionnalité de sauvegarde/restauration via un fichier Très pratique, elle évite de devoir réécrire le code à chaque modification ou chargement du programme.
- Ajout d'une **SlideBar** [22], une fonctionnalité nécessaire pour l'ergonomie de l'interface, permet de voir les éléments plus bas.

Il a été décidé de concentrer les efforts sur l'ajout de fonctionnalités permettant de gagner du temps lors du développement et des tests.

6.1.2. changement de l'interface

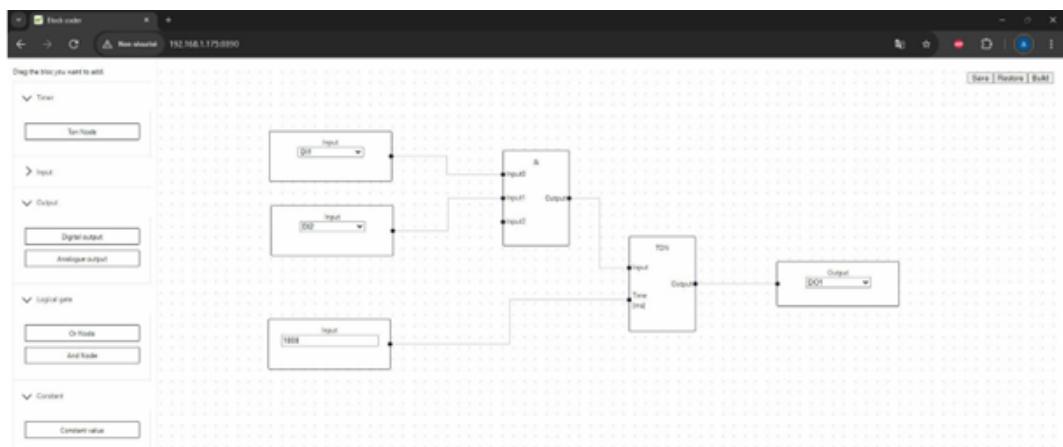


Fig. 78. – Interface avant

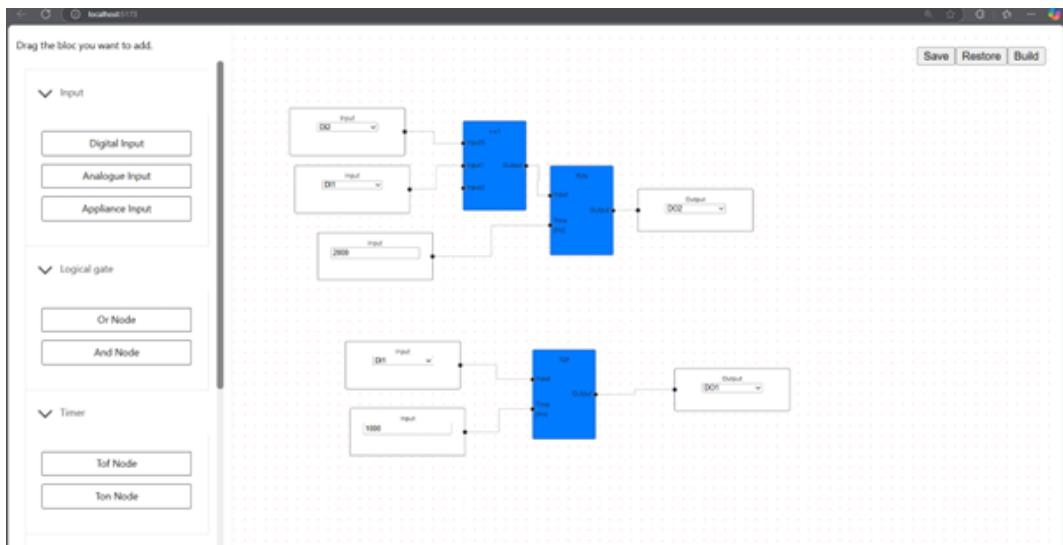


Fig. 79. – Interface après

La différence est la **slide Bar** car avant si on ouvrait tous on n'avait pas accès aux composants du bas. On voit également le nouveau bloc de type **TOR** qui a été ajouté.

Bloc bleu : il sont le résultat d'un test qui a été fait l'objectif était de voir comment était géré le style css des blocs. Le résultat est qu'il est géré par groupe. Ainsi, tous les Blocs *LogicalNode* ont le même type. Il faudra donc améliorer la structure pour rendre plus facile l'attribution de style si on veut plus personnaliser.



6.2. Comparaison avec les objectifs initial

Les objectifs fixés par pr4 sont atteints. Le programme a pu être testé et permet de créer des programmes très simples. Un nouveau bloc a été ajouté et testé sur l'automate. Le principe de fonctionnement des codes a été vus et il est possible d'ajouter de nouveaux. Cependant, le programme n'est pas parfait. Il reste des points à améliorer, notamment des erreurs .

6.3. Difficultés rencontrées

La documentation de WDA n'est pas suffisante pour comprendre le fonctionnement de la *library*. Il y a beaucoup de paramètres différents, mais on ne trouve pas ceux qui nous intéressent, la majorité d'entre eux sont pour modifier des paramètres de la configuration automatique. Cependant, il a pu être remarqué que les modèles **741-9402** et **751-9401** ne sont pas les mêmes. La documentation du 741-9402 est plus complète, et l'utilisation des entrées/sorties (I/O) y est clairement expliquée. En revanche, il n'a toujours pas été trouvé de documentation concernant l'utilisation du module CAN.

6.4. Perspectives d'avenir

6.4.1. Idées d'amélioration et extensions du **frontend web**

Il y a de nombreuses possibilités d'amélioration pour l'interface utilisateur.

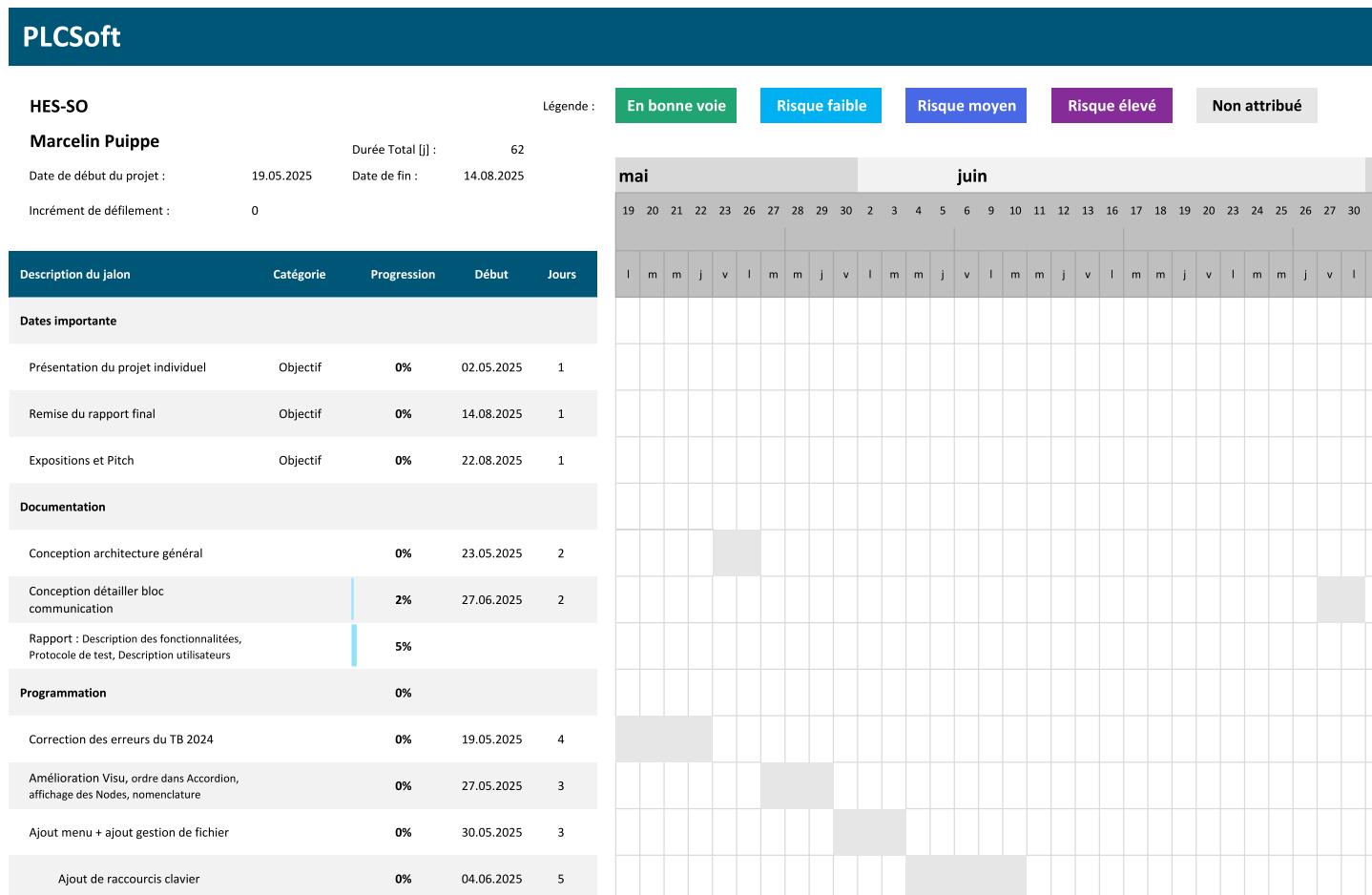
- Interdire les liens qui passent sur un bloc, ajouter une intelligence de connexion.
- Interdire les boucles de rétroaction (comme dans Codesys) ou les gérer proprement.
- Ajouter des blocs logiques contenant un champ (pour les Inputs, c'est déjà en partie fait, mais non fonctionnel, et il n'y a pas de système de seuil).
- Améliorer la nomenclature : éviter d'utiliser « Output » pour l'analogique et le digital, et « Input » pour les constantes. Une idée serait d'ajouter un menu déroulant sur le bloc pour choisir le type.
- Permettre de coder le **frontend** indépendamment du **backend**, c'est-à-dire générer les accordéons à partir d'un fichier, qui peut être mis à jour lorsqu'on est connecté.
- Ajout de raccourcis clavier :
 - Rendre la touche *Delete* fonctionnel.
 - Ctrl + C / V / A / Z / Y.
 - Clic + glisser = multi-sélection.
 - Touche O pour placer un Output, I pour un Input.
 - Touche Espace pour placer un composant identique au précédent.
 - Shift + clic gauche + glisser pour dupliquer.
 - Une autre idée intéressante : une touche (Ctrl + Alt + C) pour ajouter automatiquement tous les blocs nécessaires autour d'un bloc ou groupe sélectionné, avec des valeurs par défaut. Par exemple, on sélectionne un bloc TON, on appuie sur la touche, et le système ajoute automatiquement une constante de 1 seconde, une entrée DIO1 (ou la suivante si déjà utilisée), et une sortie DO1. Les valeurs par défaut ne sont pas obligatoires, on peut faire sans.
 - Touche dédiée pour activer ou désactiver les valeurs par défaut.
- L'ordre des Inputs, Outputs, blocs logiques, etc. dans l'accordéon n'est jamais le même, ce qui rend l'utilisation plus pénible car on ne peut pas s'habituer.
- Amélioration de l'aspect visuel : couleurs et autres éléments graphiques.
- Ajout d'une barre de menu en haut :
 - Affichage des raccourcis clavier
 - Aide
 - Choix de l'emplacement d'enregistrement par l'utilisateur
 - Ajout d'un mode Debug

7 | Annexe

Contenu

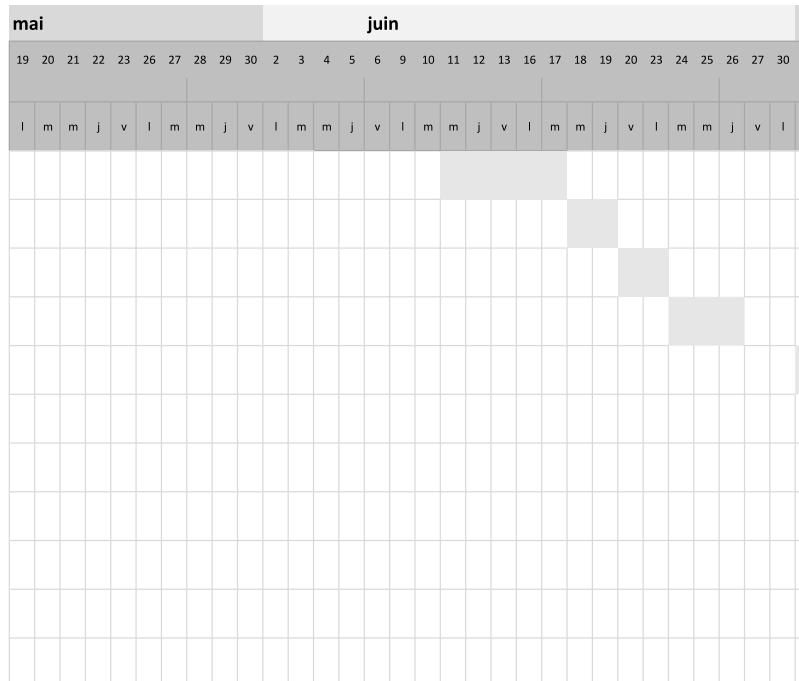
7.1	Planning	68
7.2	WDA analyse 751-9401	73
7.2.1	Sans wda : accéder IO	73
7.3	WDA Monitoring Lists	75
7.3.1	Création	75
7.3.2	Utilisation	76
7.4	WDA access mode	81
7.5	WDA I/O	82
7.5.1	Configuration des DIO (activation des Outputs) via WDA	82
7.5.2	Activation d'une Output via WDA	83
8.1	Exemple mode debug : simple	84
8.2	MQTT configuration détaillé	91
8.3	Exemples MQTT	93
8.3.1	Exemple 1	93
8.3.2	Exemple 2	98
8.4	HTTP Client : Exemple WDA intégré	100
8.5	HTTP Server : Exemples	110
8.6	<i>Home-IO</i>	120
8.7	Exemples Modbus Read Bool	122
8.7.1	Exemple 1 – Modbus Read Bool – sans Quantity	122
8.7.2	Exemple 2 – Modbus Read Bool – avec Quantity	124
8.7.3	Exemple 3 – Modbus Read Bool – démonstration complète de la différence entre Quantity et Addresses	125
8.7.4	Exemple 4 – Modbus Read Bool – sans Addresses	126
8.8	Exemple Modbus Read Value	127
8.9	Exemple Modbus Write Bool	128
8.10	Exemple Modbus Write Value	132

7.1 Planning



Date de début du projet : 19.05.2025 Date de fin : 14.08.2025
 Incrément de défilement : 0

Description du jalon	Catégorie	Progression	Début	Jours
Ajout mode Debug		0%	11.06.2025	5
Création bloc "TRIG, R_TRIG, F_TRIG"		0%	18.06.2025	2
Création bloc logiques contenant un champ + système de seuil		0%	20.06.2025	2
Création bloc "bool to frame"		0%	24.06.2025	3
Création bloc "bloc de communication MQTT"		0%	01.07.2025	10
Création bloc "bloc de communication HTTP"		0%	15.07.2025	5
Création bloc "bloc de communication WebServer"		0%	22.07.2025	5
Création bloc "bloc de communication CAN"		0%	29.07.2025	8
Debugage			08.08.2025	
Tester sur maison connectée, application de démonstration			08.08.2025	2
Reserve			12.08.2025	1



PLCSoft

HES-SO

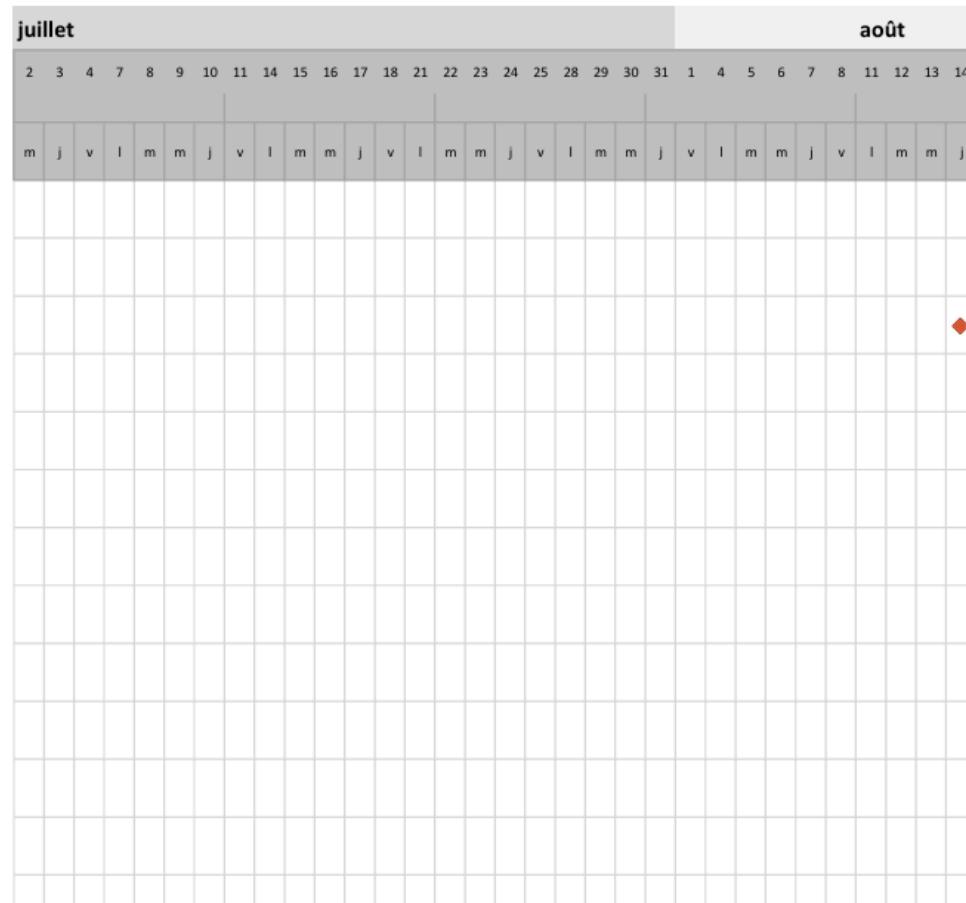
Marcelin Puippe

Durée Total [j] : 62
 Date de début du projet : 19.05.2025 Date de fin : 14.08.2025
 Incrément de défilement : 32

Légende :

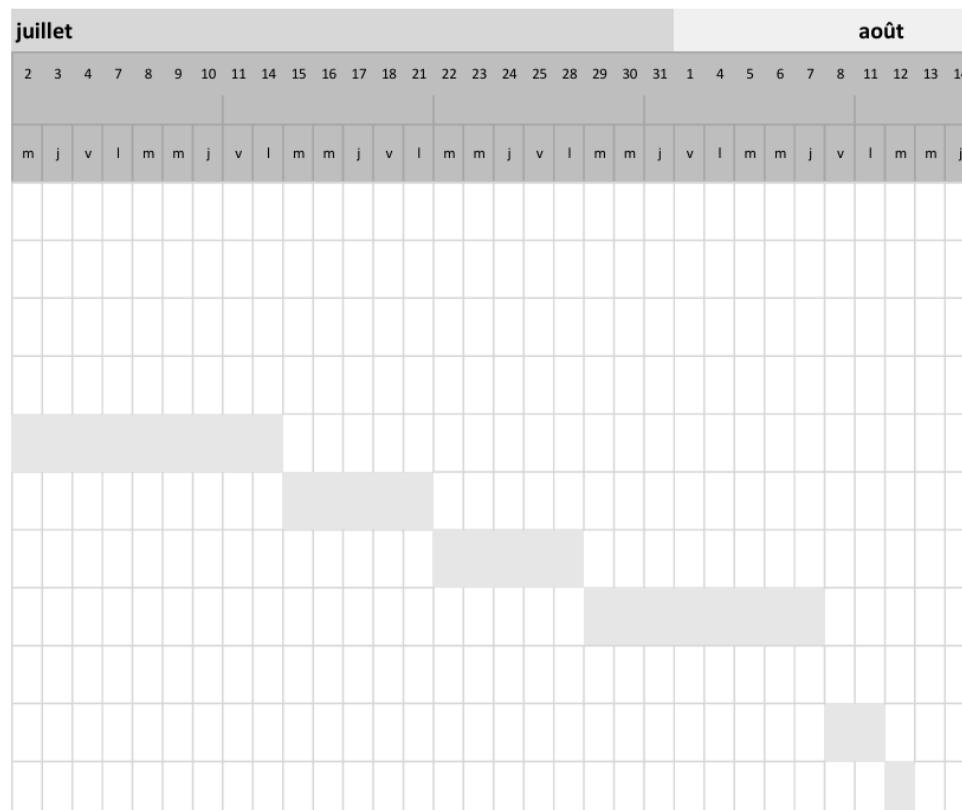
En bonne voie
Risque faible
Risque moyen
Risque élevé
Non attribué

Description du jalon	Catégorie	Progression	Début	Jours
Dates importante				
Présentation du projet individuel	Objectif	0%	02.05.2025	1
Remise du rapport final	Objectif	0%	14.08.2025	1
Expositions et Pitch	Objectif	0%	22.08.2025	1
Documentation				
Conception architecture général		0%	23.05.2025	2
Conception détailler bloc communication		2%	27.06.2025	2
Rapport : Description des fonctionnalités, Protocole de test, Description utilisateurs		5%		
Programmation				
Correction des erreurs du TB 2024		0%	19.05.2025	4
Amélioration Visu, ordre dans Accordion, affichage des Nodes, nomenclature		0%	27.05.2025	3
Ajout menu + ajout gestion de fichier		0%	30.05.2025	3
Ajout de reconnaissance visuelle		0%	04.06.2025	5



Date de début du projet : 19.05.2025 Date de fin : 14.08.2025
 Incrémentation de défilement : 32

Description du jalon	Catégorie	Progression	Début	Jours
Ajout mode Debug		0%	11.06.2025	5
Création bloc "TRIG, R_TRIG, F_TRIG"		0%	18.06.2025	2
Création bloc logiques contenant un champ + système de seuil		0%	20.06.2025	2
Création bloc "bool to frame"		0%	24.06.2025	3
Création bloc "bloc de communication MQTT"		0%	01.07.2025	10
Création bloc "bloc de communication HTTP"		0%	15.07.2025	5
Création bloc "bloc de communication WebServer"		0%	22.07.2025	5
Création bloc "bloc de communication CAN"		0%	29.07.2025	8
Debugage			08.08.2025	
Tester sur maison connectée, application de démonstration			08.08.2025	2
Reserve			12.08.2025	1



7.2 WDA analyse 751-9401

Les liens donnés dans la section suivante dépendent de l'adresse de l'automate.

Documentation API : <https://192.168.37.134/openapi/wda.openapi.html>

Documentation 751-9401 : [23]

Json Parameter : [https://192.168.37.134/wda/parameter-definitions?page\[limit\]=20000](https://192.168.37.134/wda/parameter-definitions?page[limit]=20000)

7.2.1 Sans wda : accéder IO

C'est comme cela que c'était fait avant le début du projet.

Lien pour accéder sur page web : 192.168.37.134:8888/api/v1/hal/io

Résultat affiché sur la page web :

```
{<      di      >:[false,false,false,false,false,false],<      do      >:
[false,false,false],<    ai    >:[0.336,0.343],<    ao    >:[0,0],<    temp   >:
[16778.26508951407,16778.26508951407]}
```

Ensuite, on peut lire et écrire pour travailler avec les I/O.

Activer une output :

Dans le fichier « OutputUpdate.go » de softplc-main. Pour activer DO1 : **PUT http://192.168.37.134:8888/api/v1/hal/do/0**

```
47     req, err := http.NewRequest("PUT", url, data)  err: nil    req: PUT http://192.168.37.134:8888/api/v1/hal/do/0
48     if err != nil {
49         fmt.Println(err)
50     }
51     req.Header.Set("Content-Type", "application/json")
52
53     client := &http.Client{}  client: *http.Client | 0xc000020220
54     resp, err := client.Do(req)  resp: *http.Response | 0xc000490120
55     if err != nil {
```

Fig. 82. – programme : OutputUpdate.go

C'est à partir de la ligne 54 qu'on a la lampe allumée. Plus de détails dans le dossier « autre » puis dossier « Request ».

```
{"di": [false, false, false, false, false, false], "do": [false, false, false, false], "ai": [0.336, 0.343], "ao": [0, 0], "temp": [16778.26508951407, 16778.26508951407]}
GET /api/v1/hal/do/0 HTTP/1.1
Host: 192.168.37.134:8888
User-Agent: Go-http-client/1.1
Accept-Encoding: gzip

HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
Vary: Origin
Date: Fri, 11 Apr 2025 09:46:07 GMT
Content-Length: 162

{"di": [true, false, false, false, false, false], "do": [false, false, false, false], "ai": [0.336, 0.343], "ao": [0, 0], "temp": [16778.26508951407, 16778.26508951407]}
PUT /api/v1/hal/do/0 HTTP/1.1
Host: 192.168.37.134:8888
User-Agent: Go-http-client/1.1
Content-Length: 4
Content-Type: application/json
Accept-Encoding: gzip

trueHTTP/1.1 204 No Content
Vary: Origin
Date: Fri, 11 Apr 2025 09:46:07 GMT
```

Fig. 83. – wireShark Http stream

7.2.1.1 Avec wda

751-9402 : <https://192.168.37.134/wda/parameters/0-0-io-channelcompositions-1-channels>

```

{
  "data": {
    "attributes": {
      "dataRank": "array",
      "dataType": "instance_ref",
      "path": "IO/ChannelCompositions/1/Channels"
    },
    "value": [
      {"id": "0-0-io-channelcompositions-1-channels", "name": "X12", "status": "21"}, {"id": "0-0-io-channelcompositions-1-channels", "name": "DI1", "status": "22"}, {"id": "0-0-io-channelcompositions-1-channels", "name": "DI2", "status": "22"}, {"id": "0-0-io-channelcompositions-1-channels", "name": "DI3", "status": "23"}, {"id": "0-0-io-channelcompositions-1-channels", "name": "DI5", "status": "24"}, {"id": "0-0-io-channelcompositions-1-channels", "name": "DI6", "status": "25"}, {"id": "0-0-io-channelcompositions-1-channels", "name": "DI7", "status": "26"}, {"id": "0-0-io-channelcompositions-1-channels", "name": "DI8", "status": "27"}, {"id": "0-0-io-channelcompositions-1-channels", "name": "DI9", "status": "28"}]
  }
}

```

Fig. 84. – automate 751-9402 accéder aux IO

C'est possible avec un automate 751-9402 [24], cependant on a le modèle 751-9401 [23]. Cela ne signifie pas que c'est impossible, mais que ce n'est pas documenté.

7.3 WDA Monitoring Lists

Documentation (accessible uniquement si l'automate est disponible) : <https://192.168.37.134/openapi/WDA.openapi.html#tag/Monitoring-Lists>

7.3.1 Création

i Remarquer que nous insérons un **timeout** de 600 secondes, ce qui correspond à 10 minutes. Il est possible de le modifier si besoin. A noter qu'il faudra **recréer** la Monitoring List à la fin de ce délai pour continuer à monitorer les entrées et sorties de l'automate.

De plus, il faudra mémoriser l'ID de la Monitoring List créée, car il sera nécessaire pour les requêtes GET et DELETE. Il est possible de créer plusieurs Monitoring Lists, mais il faudra alors faire attention à ne pas dépasser le nombre maximum autorisé par l'automate.

POST <https://192.168.37.134/wda/monitoring-lists>

Fig. 85. – commande post Monitoring List

```
{
  "data": {
    "type": "monitoring-lists",
    "attributes": {
      "timeout": 600
    },
    "relationships": {
      "parameters": {
        "data": [
          { "id": "0-0-io-channels-21-divalue", "type": "parameters" },
          { "id": "0-0-io-channels-22-divalue", "type": "parameters" },
          { "id": "0-0-io-channels-23-divalue", "type": "parameters" },
          { "id": "0-0-io-channels-9-dovalue", "type": "parameters" },
          { "id": "0-0-io-channels-10-dovalue", "type": "parameters" },
          { "id": "0-0-io-channels-11-dovalue", "type": "parameters" }
        ]
      }
    }
  }
}
```

Liste 8. – **body send**, exemple de création d'une Monitoring List pour les 3 premières Inputs et 3 premières Outputs de l'automate

```
{
  "data": {
    "attributes": {
      "timeout": 600
    },
    "id": "26",
    "links": {
      "self": "/WDA/monitoring-lists/26"
    },
    "relationships": {
      "parameters": {
        "links": {
          "related": "/WDA/monitoring-lists/26/parameters"
        }
      }
    },
    "type": "monitoring-lists"
  },
  "jsonapi": {
    "version": "1.0"
  },
  "meta": {
    "doc": "/openapi/WDA.openapi.html#operation/createMonitoringList",
    "version": "1.4.1"
  }
}
```

Liste 9. – **body response**, exemple de création d'une Monitoring List pour les 3 premières Inputs et 3 premières Outputs de l'automate

7.3.2 Utilisation

i Pour utiliser la Monitoring List, il faut faire une requête GET sur l'URL de la Monitoring List créée précédemment. Il est possible de récupérer l'état de toutes les entrées et sorties en une seule requête GET.

On remarque qu'il y a beaucoup d'informations qui ne nous intéressent pas dans la réponse. Il y a que « path » et « value » de « attributes » qui nous intéressent.

```
GET ↴ https://192.168.37.134/wda/monitoring-lists/26/parameters
```

Fig. 86. – commande GET Monitoring List

```
{
  "data": [
    {
      "attributes": {
        "dataRank": "scalar",
        "dataType": "boolean",
        "path": "IO/Channels/21/DIValue",
        "value": false
      },
      "id": "0-0-io-channels-21-divalue",
      "links": {
        "self": "/WDA/parameters/0-0-io-channels-21-divalue"
      },
      "relationships": {
        "definition": {
          "links": {
            "related": "/WDA/parameters/0-0-io-channels-21-divalue/
definition"
          }
        },
        "device": {
          "links": {
            "related": "/WDA/parameters/0-0-io-channels-21-divalue/device"
          }
        }
      },
      "type": "parameters"
    },
    {
      "attributes": {
        "dataRank": "scalar",
        "dataType": "boolean",
        "path": "IO/Channels/22/DIValue",
        "value": false
      },
      "id": "0-0-io-channels-22-divalue",
      "links": {
        "self": "/WDA/parameters/0-0-io-channels-22-divalue"
      },
      "relationships": {
        "definition": {
          "links": {
            "related": "/WDA/parameters/0-0-io-channels-22-divalue/
definition"
          }
        },
        "device": {
          "links": {
            "related": "/WDA/parameters/0-0-io-channels-22-divalue/device"
          }
        }
      },
      "type": "parameters"
    }
  ]
}
```

Liste 10. – **body response**, exemple d'utilisation d'une Monitoring List pour les 3 premières Inputs et 3 premières Outputs de l'automate

```
{
  "attributes": {
    "dataRank": "scalar",
    "dataType": "boolean",
    "path": "IO/Channels/23/DIValue",
    "value": false
  },
  "id": "0-0-io-channels-23-divalue",
  "links": {
    "self": "/WDA/parameters/0-0-io-channels-23-divalue"
  },
  "relationships": {
    "definition": {
      "links": {
        "related": "/WDA/parameters/0-0-io-channels-23-divalue/
definition"
      }
    },
    "device": {
      "links": {
        "related": "/WDA/parameters/0-0-io-channels-23-divalue/device"
      }
    }
  },
  "type": "parameters"
},
{
  "attributes": {
    "dataRank": "scalar",
    "dataType": "boolean",
    "path": "IO/Channels/9/D0Value",
    "value": false
  },
  "id": "0-0-io-channels-9-dovalue",
  "links": {
    "self": "/WDA/parameters/0-0-io-channels-9-dovalue"
  },
  "relationships": {
    "definition": {
      "links": {
        "related": "/WDA/parameters/0-0-io-channels-9-dovalue/
definition"
      }
    },
    "device": {
      "links": {
        "related": "/WDA/parameters/0-0-io-channels-9-dovalue/device"
      }
    }
  },
  "type": "parameters"
}
}
```

Liste 11. – **body response**, exemple d'utilisation d'une Monitoring List pour les 3 premières Inputs et 3 premières Outputs de l'automate

```
{
  "attributes": {
    "dataRank": "scalar",
    "dataType": "boolean",
    "path": "IO/Channels/10/D0Value",
    "value": false
  },
  "id": "0-0-io-channels-10-dovalue",
  "links": {
    "self": "/WDA/parameters/0-0-io-channels-10-dovalue"
  },
  "relationships": {
    "definition": {
      "links": {
        "related": "/WDA/parameters/0-0-io-channels-10-dovalue/
definition"
      }
    },
    "device": {
      "links": {
        "related": "/WDA/parameters/0-0-io-channels-10-dovalue/device"
      }
    }
  },
  "type": "parameters"
},
{
  "attributes": {
    "dataRank": "scalar",
    "dataType": "boolean",
    "path": "IO/Channels/11/D0Value",
    "value": false
  },
  "id": "0-0-io-channels-11-dovalue",
  "links": {
    "self": "/WDA/parameters/0-0-io-channels-11-dovalue"
  },
  "relationships": {
    "definition": {
      "links": {
        "related": "/WDA/parameters/0-0-io-channels-11-dovalue/
definition"
      }
    },
    "device": {
      "links": {
        "related": "/WDA/parameters/0-0-io-channels-11-dovalue/device"
      }
    }
  },
  "type": "parameters"
}
]
```

Liste 12. – **body response**, exemple d'utilisation d'une Monitoring List pour les 3 premières Inputs et 3 premières Outputs de l'automate

```
"jsonapi": {  
    "version": "1.0"  
,  
    "links": {  
        "first": "/WDA/monitoring-lists/26/parameters?  
page[limit]=255&page[offset]=0",  
        "last": "/WDA/monitoring-lists/26/parameters?  
page[limit]=255&page[offset]=0",  
        "self": "/WDA/monitoring-lists/26/parameters?  
page[limit]=255&page[offset]=0"  
    },  
    "meta": {  
        "doc": "/openapi/WDA.openapi.html#operation/  
getMonitoringListParameters",  
        "version": "1.4.1"  
    }  
}
```

Liste 13. – **body response**, exemple d'utilisation d'une Monitoring List pour les 3 premières Inputs et 3 premières Outputs de l'automate

7.4 WDA access mode

Afin de pouvoir **écrire** ou **lire** les entrées et sorties de l'automate, il faut activer le mode d'accès WDA correspond. Le paramètre « Value » peut prendre une des 3 valeurs entières suivantes :

- 0 : no access (no read/write access)
- 1 : monitor mode (read-only mode)
- 2 : control mode (read/write mode)

```
PATCH 🔳 https://192.168.37.134/wda/parameters/0-0-io-iocheckaccessmode
```

Fig. 87. – commande PATCH pour changer le mode d'accès WDA

```
{
  "data": {
    "id": "0-0-io-iocheckaccessmode",
    "type": "parameters",
    "attributes": {
      "value": 2
    }
  }
}
```

Liste 14. – **body send**, exemple changer le mode d'accès WDA à « control mode » (read/write mode)

7.5 WDA I/O

7.5.1 Configuration des DIO (activation des Outputs) via WDA

Le bornier X5 de l'automate est composé de **8 DIO**. On peut donc choisir lesquels configurer en **Input** et lesquels configurer en **Output**. Cela peut se faire via une commande **PATCH** comme le montre Fig. 90.

Le paramètre « **value** » doit prendre un tableau de valeurs entières dont chaque index correspond à une **DIO**. Par exemple, si l'on veut configurer **DIO1** pour être une **Output**, on doit mettre 9 à l'index 0 du tableau. Si l'on veut que ce soit une **Input**, on doit mettre 1 à l'index 0 du tableau.

Les figures Fig. 88 et Fig. 89 montrent les valeurs d'activation des **Inputs** et **Outputs** pour chaque **DIO**.

X5	DI1	1	DO1	9
	DI2	2	DO2	10
	DI3	3	DO3	11
	DI4	4	DO4	12
	DI5	5	DO5	13
	DI6	6	DO6	14
	DI7	7	DO7	15
	DI8	8	DO8	16

Fig. 88. – valeurs d'activation des inputs Fig. 89. – valeurs d'activation des outputs

PATCH → <https://192.168.37.134/wda/parameters/0-0-io-channelcompositions-4-channels>

Fig. 90. – commande PATCH pour activer toutes les outputs via WDA

```
{
  "data": {
    "id": "0-0-io-channelcompositions-4-channels",
    "type": "parameters",
    "attributes": {
      "value": [
        9,
        10,
        11,
        12,
        13,
        14,
        15,
        16
      ]
    }
  }
}
```

Liste 15. – **body send**, exemple pour passer toutes les DIO en output

7.5.2 Activation d'une Output via WDA

PATCH  <https://192.168.37.134/wda/parameters/0-0-io-channels-9-dovalue>

Fig. 91. – commande PATCH pour activer une output via WDA

```
{
  "data": {
    "id": "0-0-io-channels-9-dovalue",
    "type": "parameters",
    "attributes": {
      "value": true
    }
  }
}
```

Liste 16. – **body send**, exemple pour passer la valeur d'une output à true (ici DIO1) dans l'automate via WDA

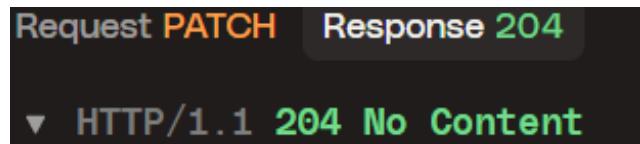


Fig. 92. – **Response** : commande PATCH pour activer une output via WDA

8 | Exemples codes : vue programme + JSON

8.1 Exemple mode debug : simple

L'objectif de cet exemple est de démontrer les changements effectués sur le graphique lorsqu'on passe en mode debug. Les modifications sont faites sur les **edges**. La principale différence entre « debug 1 » et « debug 2 » est l'actionnement du bouton DI1. La valeur affichée est transmise avec la structure suivante Liste 17. À cela sont rajoutés, le style, l'animation et autres.

```
"data": {
    "label": "valeur affichée"
},
```

Liste 17. – Mode debug : extrait structure envoi données valeur affichée

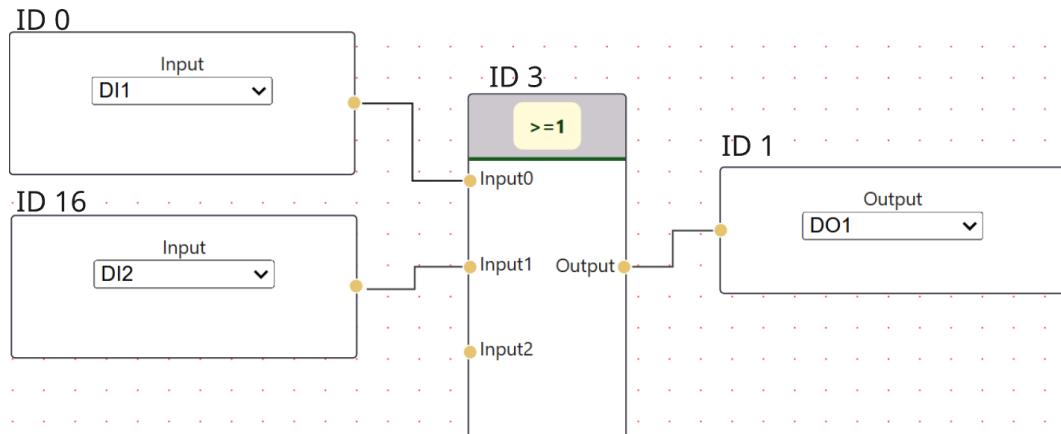


Fig. 93. – Mode debug simple : vue programmation

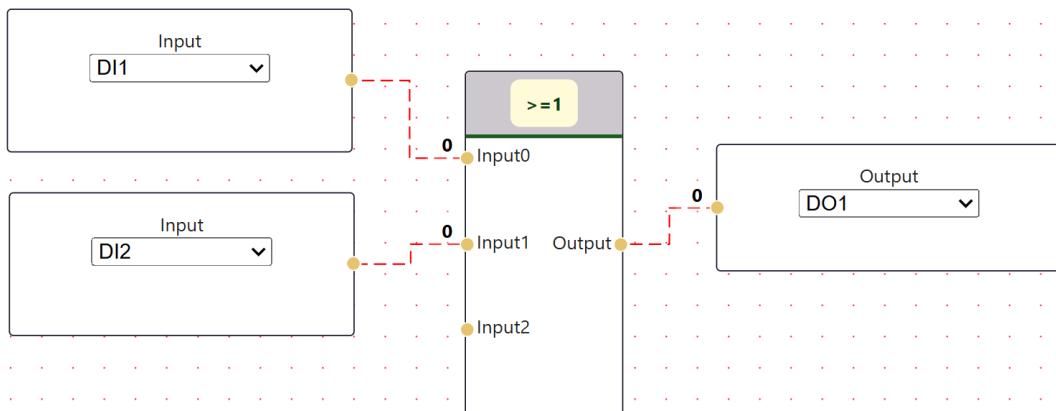


Fig. 94. – Mode debug simple : vue debug 1

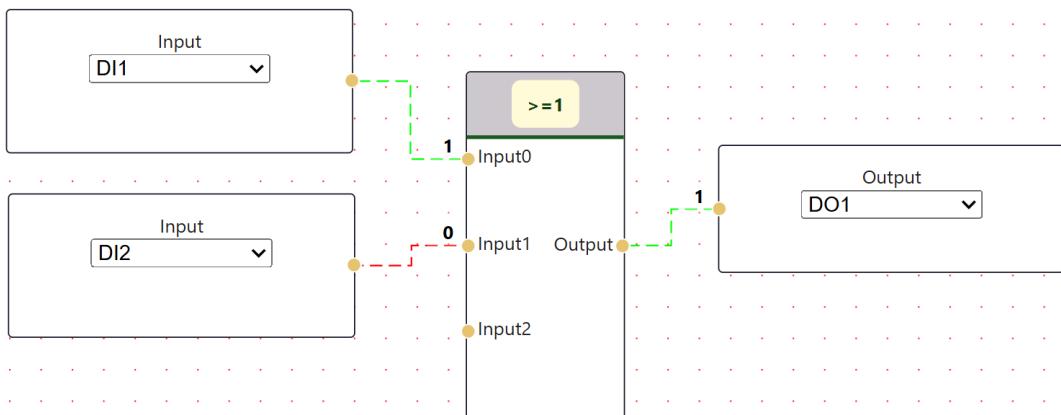


Fig. 95. – **Mode debug simple** : vue debug 2 (DI1 = true)

```
{
  "edges": [
    {
      "id": "reactflow__edge-00output-3Input0",
      "selected": false,
      "source": "0",
      "sourceHandle": "Output",
      "style": {
        "stroke": "#000000",
        "strokeWidth": 1
      },
      "target": "3",
      "targetHandle": "Input0",
      "type": "step"
    },
    {
      "id": "reactflow__edge-30output-1Input",
      "source": "3",
      "sourceHandle": "Output",
      "style": {
        "stroke": "#30362F",
        "strokeWidth": 1
      },
      "target": "1",
      "targetHandle": "Input",
      "type": "step"
    },
    {
      "id": "reactflow__edge-160output-3Input1",
      "source": "16",
      "sourceHandle": "Output",
      "style": {
        "stroke": "#30362F",
        "strokeWidth": 1
      },
      "target": "3",
      "targetHandle": "Input1",
      "type": "step"
    }
  ],
  "nodes": [
    {
      ...
    }
  ]
}
```

Liste 18. – **Mode debug simple** : Edges graphique format JSON (vue programmation)

```

"edges": [
{
  "data": {
    "label": "0"
  },
  "id": "reactflow__edge-00output-3Input0",
  "label": "???",
  "labelBgBorderRadius": 4,
  "labelBgPadding": [
    8,
    4
  ],
  "labelBgStyle": {
    "color": "#333",
    "fill": "white",
    "fillOpacity": 0.5
  },
  "selected": false,
  "source": "0",
  "sourceHandle": "Output",
  "style": {
    "animation": "dash 1s linear infinite",
    "stroke": "#FF0000",
    "strokeDasharray": "8 4",
    "strokeWidth": 1
  },
  "target": "3",
  "targetHandle": "Input0",
  "type": "customDebugEdge"
},
{
  "data": {
    "label": "0"
  },
  "id": "reactflow__edge-30output-1Input",
  "label": "???",
  "labelBgBorderRadius": 4,
  "labelBgPadding": [
    8,
    4
  ],
  "labelBgStyle": {
    "color": "#333",
    "fill": "white",
    "fillOpacity": 0.5
  },
  "source": "3",
  "sourceHandle": "Output",
  "style": {
    "animation": "dash 1s linear infinite",
    "stroke": "#FF0000",
    "strokeDasharray": "8 4",
    "strokeWidth": 1
  },
  "target": "1",
  "targetHandle": "Input",
  "type": "customDebugEdge"
}
]

```

```
{  
  "data": {  
    "label": "0"  
  },  
  "id": "reactflow__edge-160output-3Input1",  
  "label": "???",  
  "labelBgBorderRadius": 4,  
  "labelBgPadding": [  
    8,  
    4  
  ],  
  "labelBgStyle": {  
    "color": "#333",  
    "fill": "white",  
    "fillOpacity": 0.5  
  },  
  "source": "16",  
  "sourceHandle": "Output",  
  "style": {  
    "animation": "dash 1s linear infinite",  
    "stroke": "#FF0000",  
    "strokeDasharray": "8 4",  
    "strokeWidth": 1  
  },  
  "target": "3",  
  "targetHandle": "Input1",  
  "type": "customDebugEdge"  
},  
],  
"nodes": [  
  {  
    ...  
  }
```

Liste 20. – **Mode debug simple** : Edges graphique format JSON (vue debug 1)

```

"edges": [
{
  "data": {
    "label": "1"
  },
  "id": "reactflow__edge-00output-3Input0",
  "label": "???",
  "labelBgBorderRadius": 4,
  "labelBgPadding": [
    8,
    4
  ],
  "labelBgStyle": {
    "color": "#333",
    "fill": "white",
    "fillOpacity": 0.5
  },
  "selected": false,
  "source": "0",
  "sourceHandle": "Output",
  "style": {
    "animation": "dash 1s linear infinite",
    "stroke": "#00FF00",
    "strokeDasharray": "8 4",
    "strokeWidth": 1
  },
  "target": "3",
  "targetHandle": "Input0",
  "type": "customDebugEdge"
},
{
  "data": {
    "label": "1"
  },
  "id": "reactflow__edge-30output-1Input",
  "label": "???",
  "labelBgBorderRadius": 4,
  "labelBgPadding": [
    8,
    4
  ],
  "labelBgStyle": {
    "color": "#333",
    "fill": "white",
    "fillOpacity": 0.5
  },
  "source": "3",
  "sourceHandle": "Output",
  "style": {
    "animation": "dash 1s linear infinite",
    "stroke": "#00FF00",
    "strokeDasharray": "8 4",
    "strokeWidth": 1
  },
  "target": "1",
  "targetHandle": "Input",
  "type": "customDebugEdge"
}
]

```

```
{  
  "data": {  
    "label": "0"  
  },  
  "id": "reactflow__edge-160output-3Input1",  
  "label": "???",  
  "labelBgBorderRadius": 4,  
  "labelBgPadding": [  
    8,  
    4  
  ],  
  "labelBgStyle": {  
    "color": "#333",  
    "fill": "white",  
    "fillOpacity": 0.5  
  },  
  "source": "16",  
  "sourceHandle": "Output",  
  "style": {  
    "animation": "dash 1s linear infinite",  
    "stroke": "#FF0000",  
    "strokeDasharray": "8 4",  
    "strokeWidth": 1  
  },  
  "target": "3",  
  "targetHandle": "Input1",  
  "type": "customDebugEdge"  
},  
],  
"nodes": [  
  {  
    ...  
  }
```

Liste 22. – **Mode debug simple** : Edges graphique format JSON (vue debug 2)

8.2 MQTT configuration détaillé

Aucun des paramètres n'est obligatoire, par défaut le port est **1883** et le serveur tourne sur l'automate. Les « Settings » rentrés dans la vue sont données par **parameterValueData**.

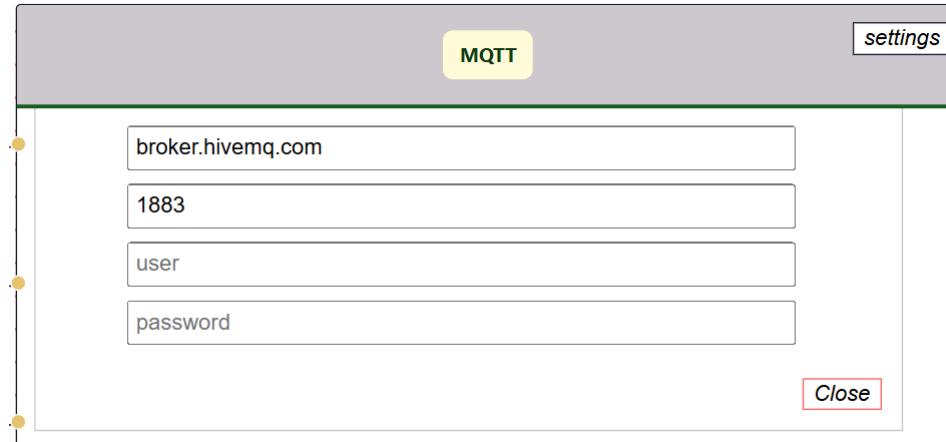


Fig. 96. – exemple de paramétrage de « Mqtt »

```
"inputHandle": [
    {
        "dataType": "bool",
        "name": "xEnable"
    },
    {
        "dataType": "value",
        "name": "topicToSend"
    },
    {
        "dataType": "value",
        "name": "msgToSend"
    },
    {
        "dataType": "value",
        "name": "topicToReceive"
    }
],
"label": "MQTT",
"outputHandle": [
    {
        "dataType": "bool",
        "name": "xDone"
    },
    {
        "dataType": "value",
        "name": "msg"
    }
],
"parameterNameData": [
    "broker",
    "port",
    "user",
    "password"
],
"parameterValueData": [
    "broker.hivemq.com",
    "1883",
    "",
    ""
]
```

Liste 23. – MQTT : extrait de la structure JSON d'un exemple

8.3 Exemples MQTT

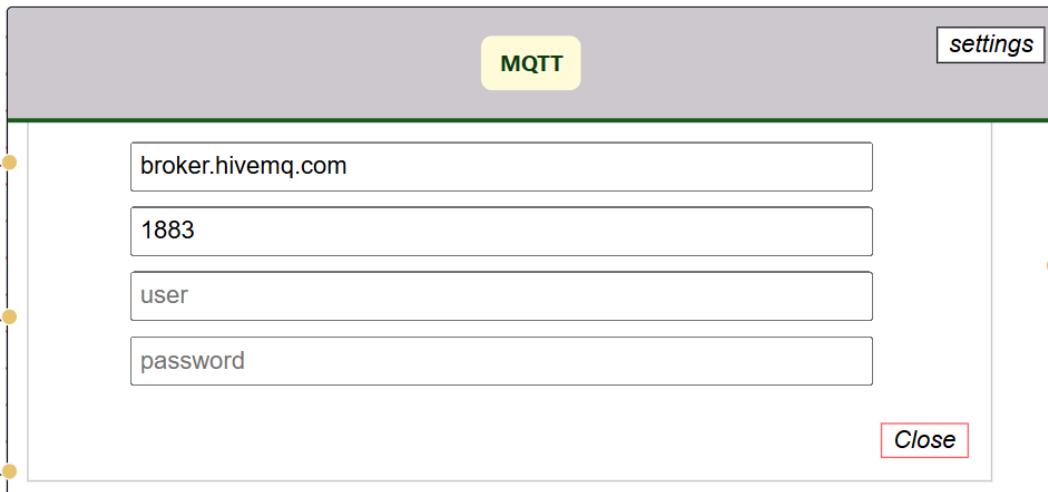


Fig. 97. – Settings MQTT utilisés pour les exemples

8.3.1 Exemple 1

Dans cet exemple, on reçoit les messages envoyés sur les 3 topics topic/test/1, topic/test/2, topic/test/3 (Fig. 100). Cet exemple montre l'envoi d'un message sur le topic « topic/test/100 » (Fig. 101) et sur les topics « topic/test/101 », « topic/test/102 » en simultané (Fig. 102). Il montre également ce qui se passe lorsque l'on reçoit plusieurs topics en même temps (Fig. 103). On remarque également que les messages reçus sont affichés dans le même ordre que les topics à recevoir, ce qui facilite son utilisation.

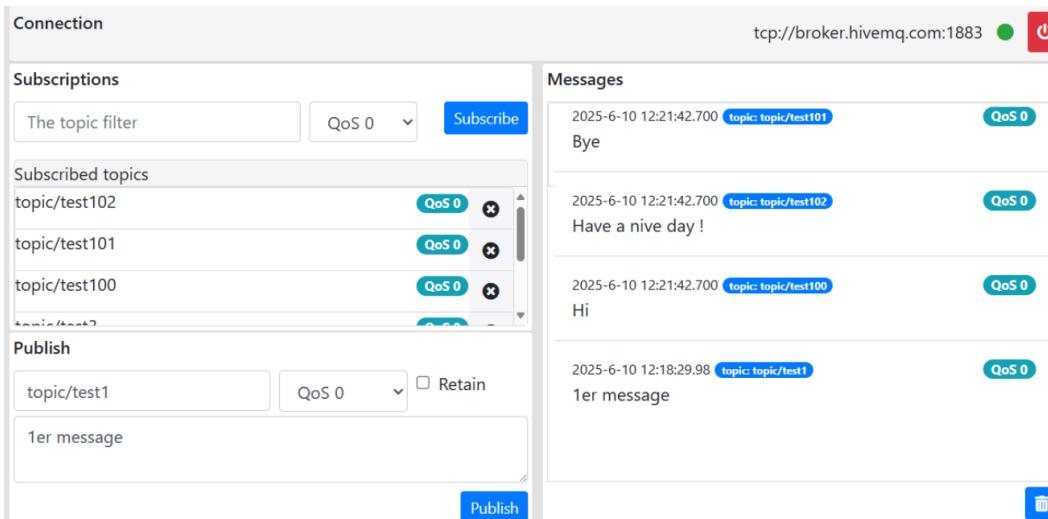


Fig. 98. – MQTT.cool : visualisation des tests clients réalisés pour l'exemple 1

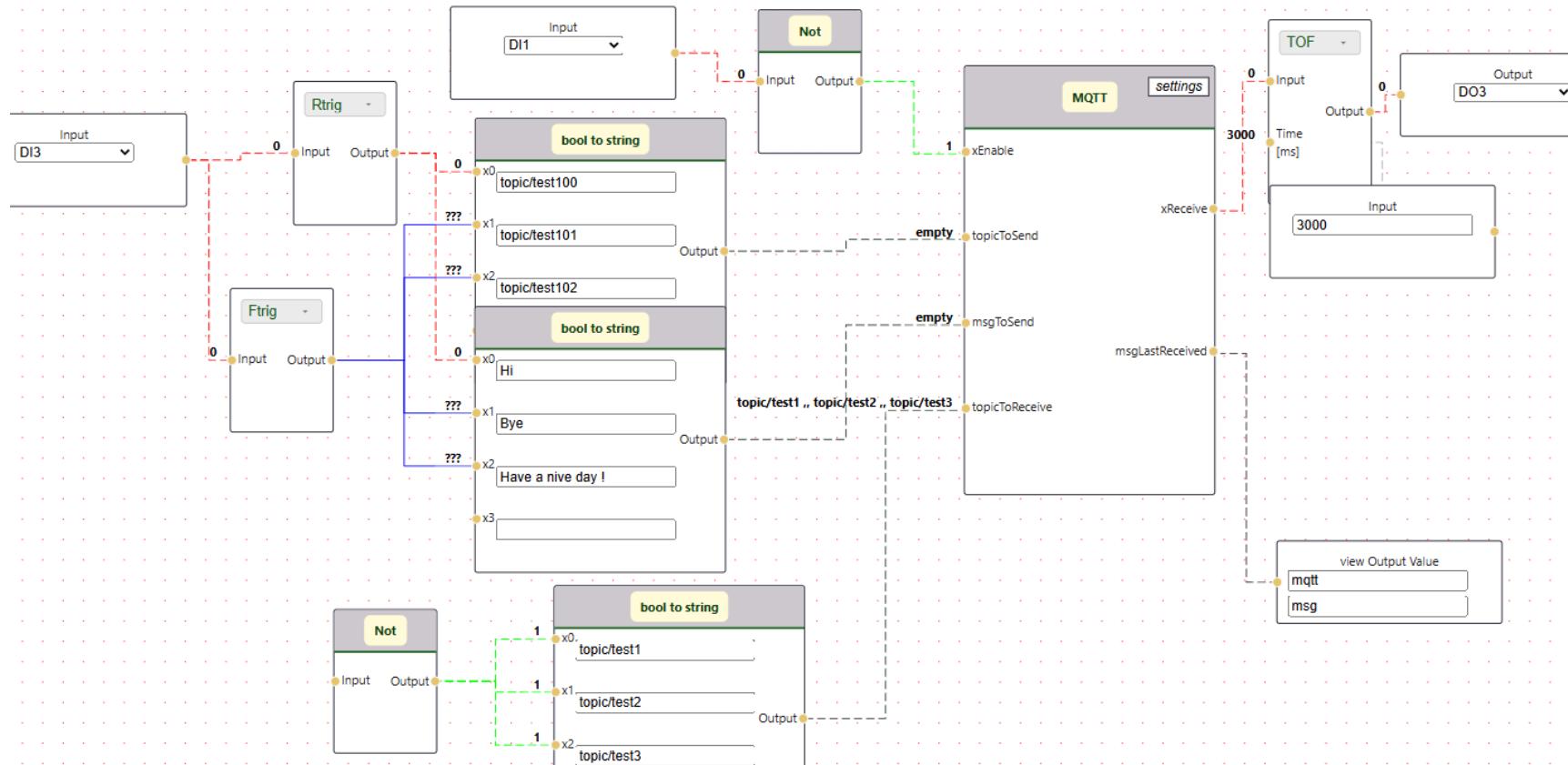


Fig. 99. – MQTT : Vue debug initiale

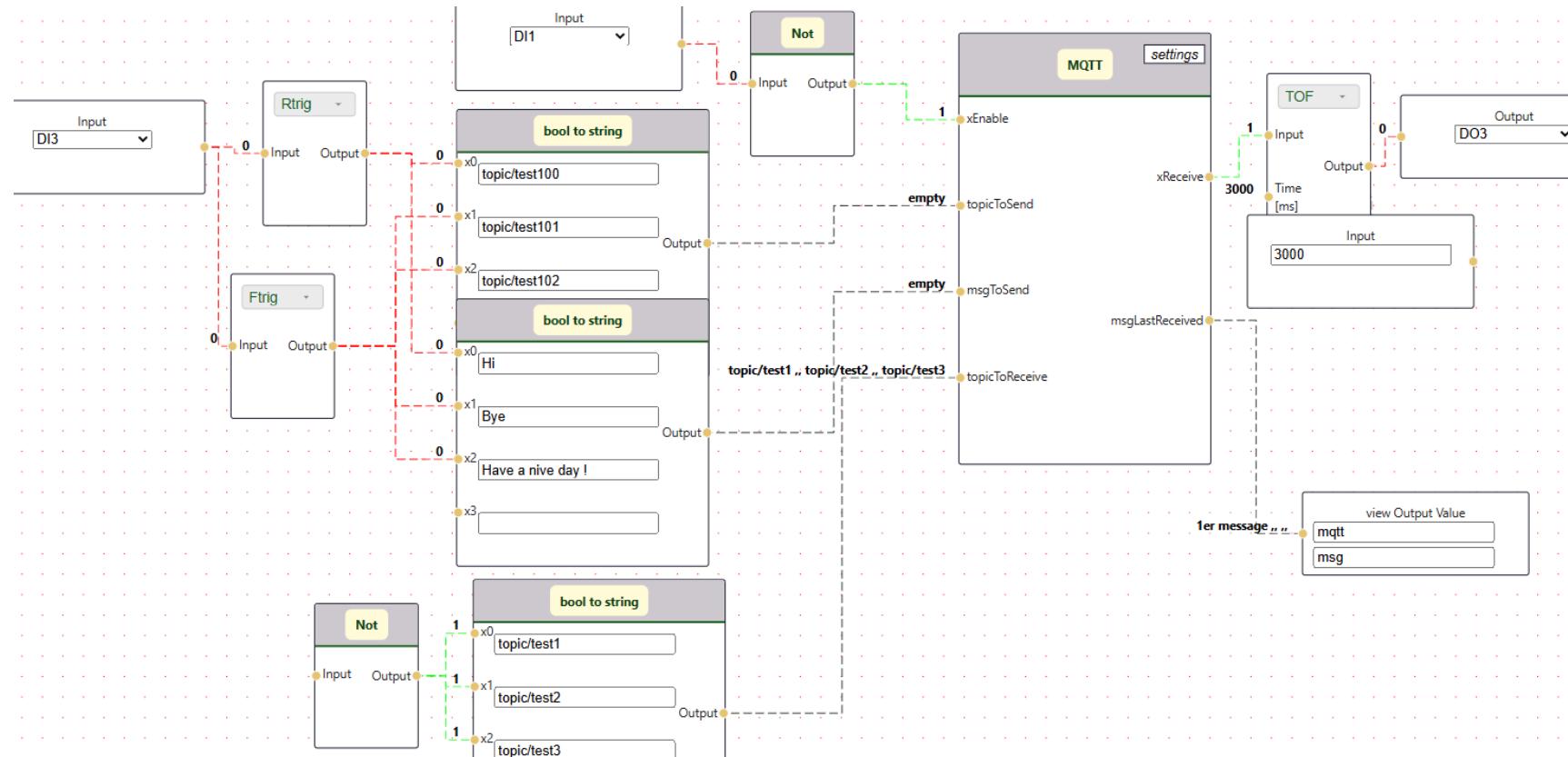


Fig. 100. – MQTT : Vue debug message reçu (« 1er message »)

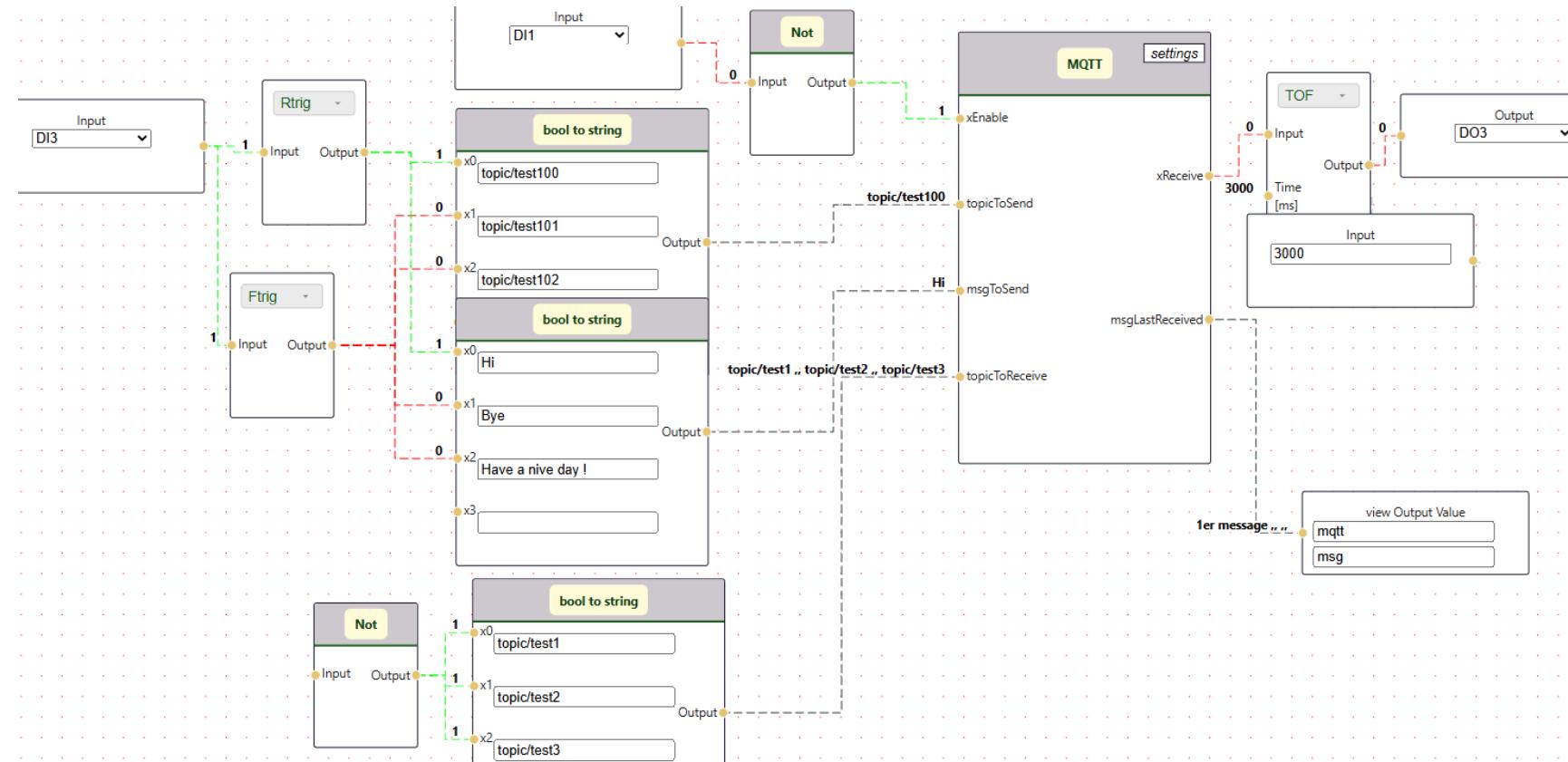


Fig. 101. – MQTT : Vue debug impulsions *Rising edge* -> message envoyé « Hi » sur le topic « topic/test/100 »

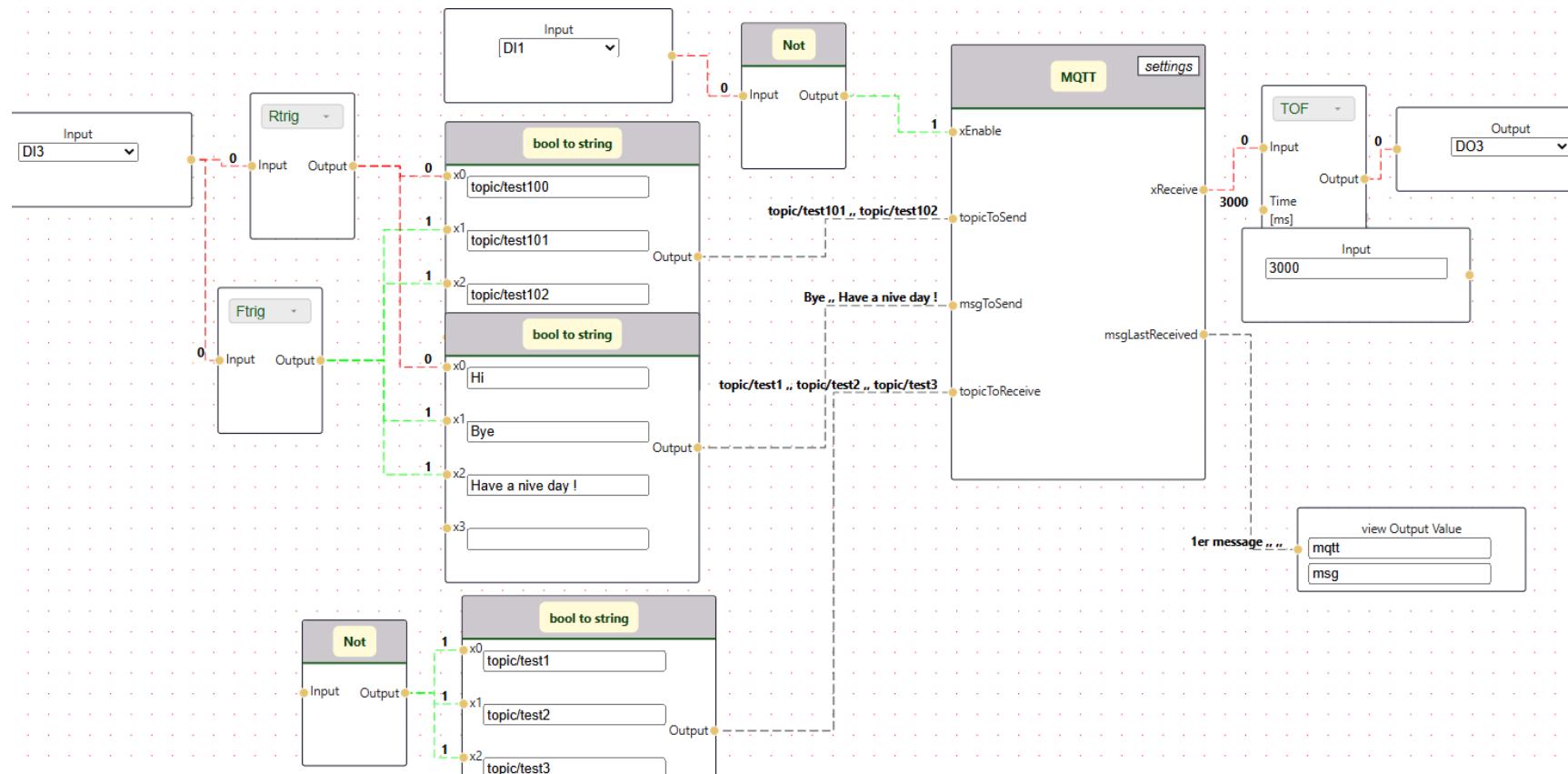


Fig. 102. – MQTT : Vue debug impulsions Falling edge -> messages envoyés : « Bye » sur le topic « topic/test/101 » et « Have a nice day ! » sur le topic « topic/test/102 »

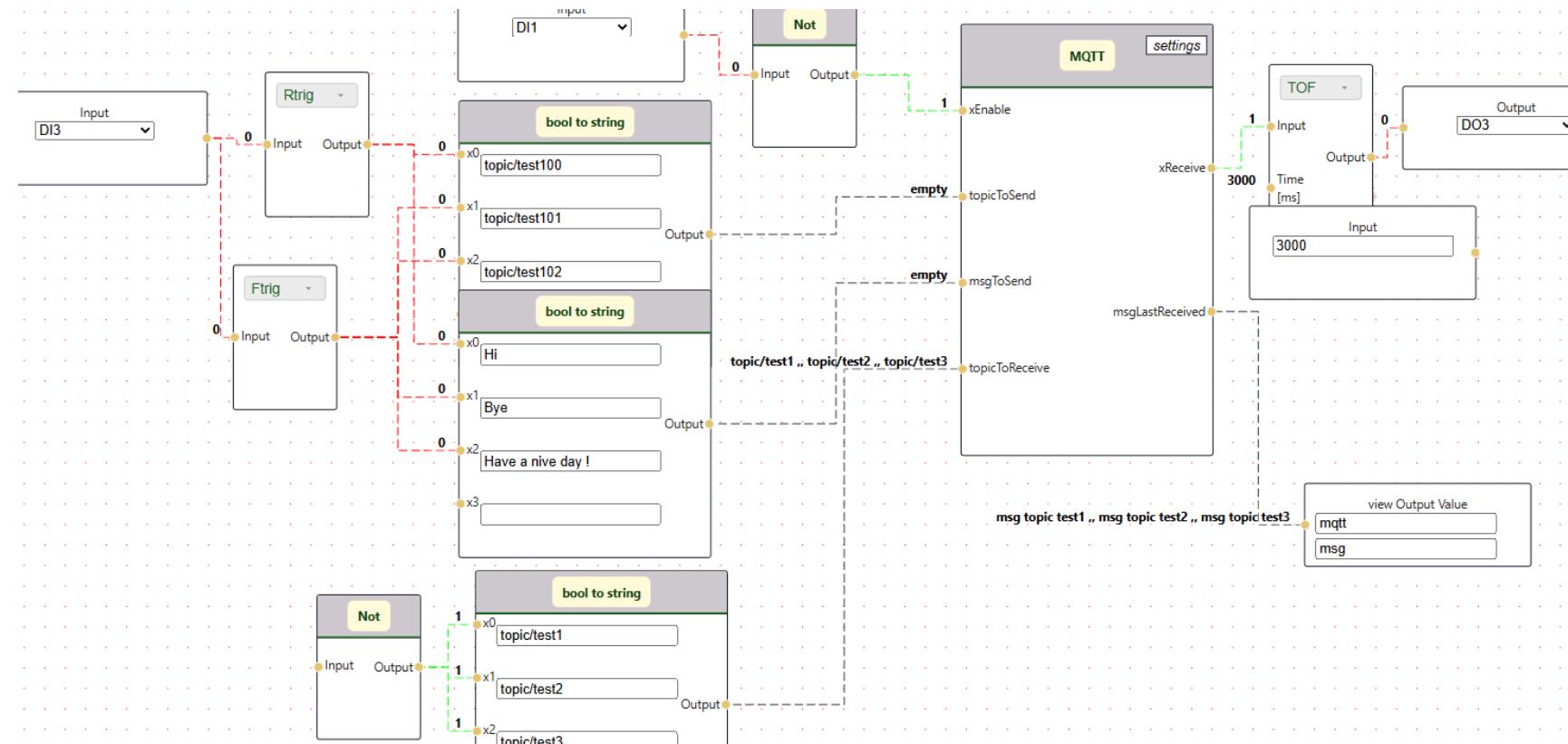
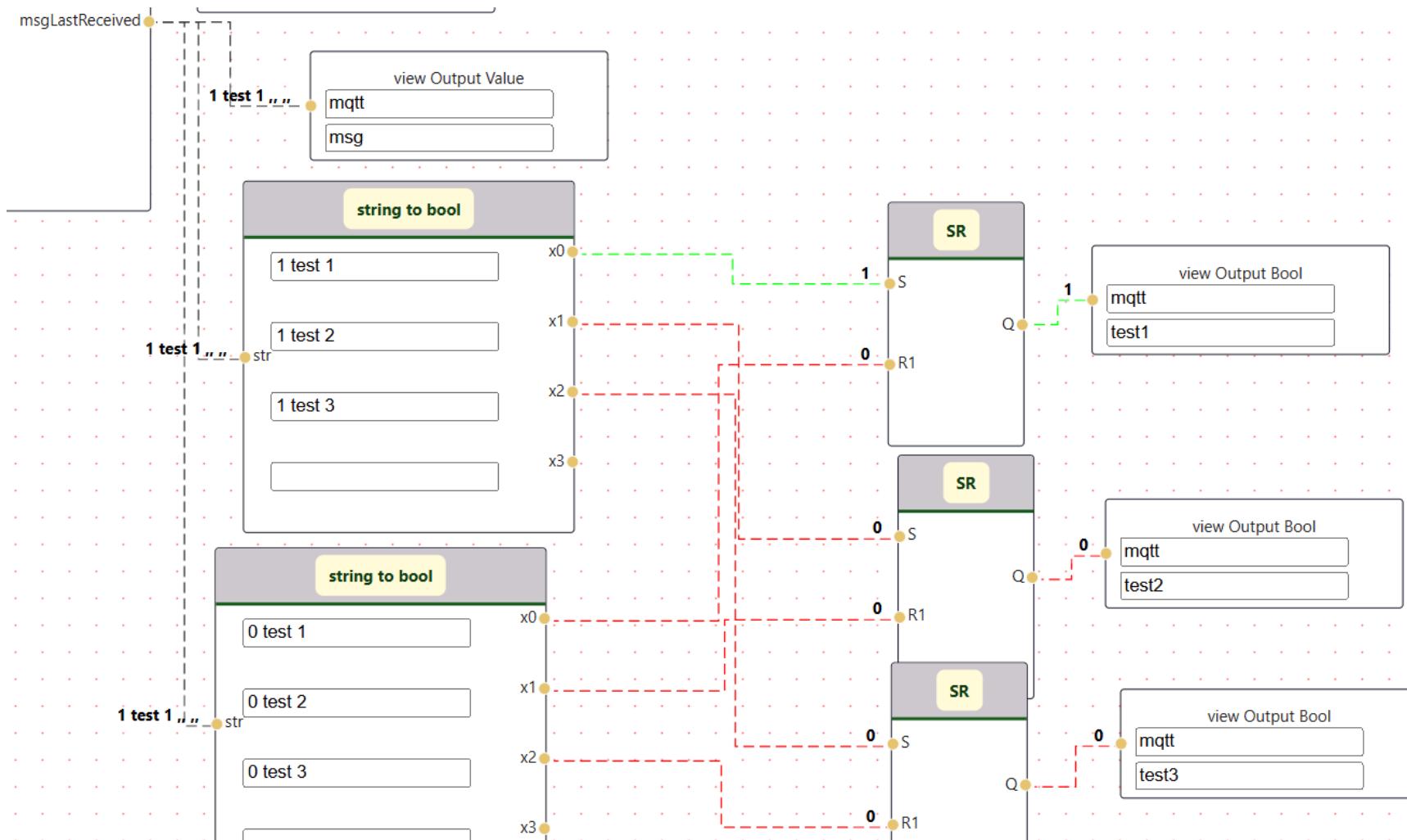


Fig. 103. – MQTT : Vue debug message reçu des 3 topics en simultané

8.3.2 Exemple 2

Cet exemple est similaire au précédent, mais cette fois-ci, on traite la sortie « **msgLastReceived** » pour activer des sorties quand on reçoit un message **1 topic x** sur le topic x et désactiver les sorties quand on reçoit un message **0 topic x** sur le topic x. Cet exemple démontre très bien la puissance que peut avoir notre programme.

Fig. 104. – MQTT : Exemple traitement sortie « `msgLastReceived` »

8.4 HTTP Client : Exemple WDA intégré

L'exemple suivant montre que l'on peut utiliser le bloc **HTTP Client** pour communiquer avec l'automate WAGO via **WDA**. Il est ainsi possible de lire ou d'écrire des entrées/sorties de l'automate, voire celles d'un autre automate.

Le code (Fig. 109) permet d'activer et de désactiver la sortie **DIO1**, et également de créer une **monitoring-list**.

L'objectif est de reproduire avec le bloc **HTTP Client** de PLCSoft ce que l'on pourrait faire avec **HTTPie**. Les requêtes HTTPie équivalentes sont illustrées dans les figures suivantes :

- Activation de la sortie DIO1 : Fig. 106
- Désactivation de la sortie DIO1 : Fig. 107
- Création d'une monitoring-list : Fig. 108



La sortie **xDone** s'active uniquement lorsqu'une réponse est reçue, ce qui explique pourquoi elle ne s'active pas immédiatement.

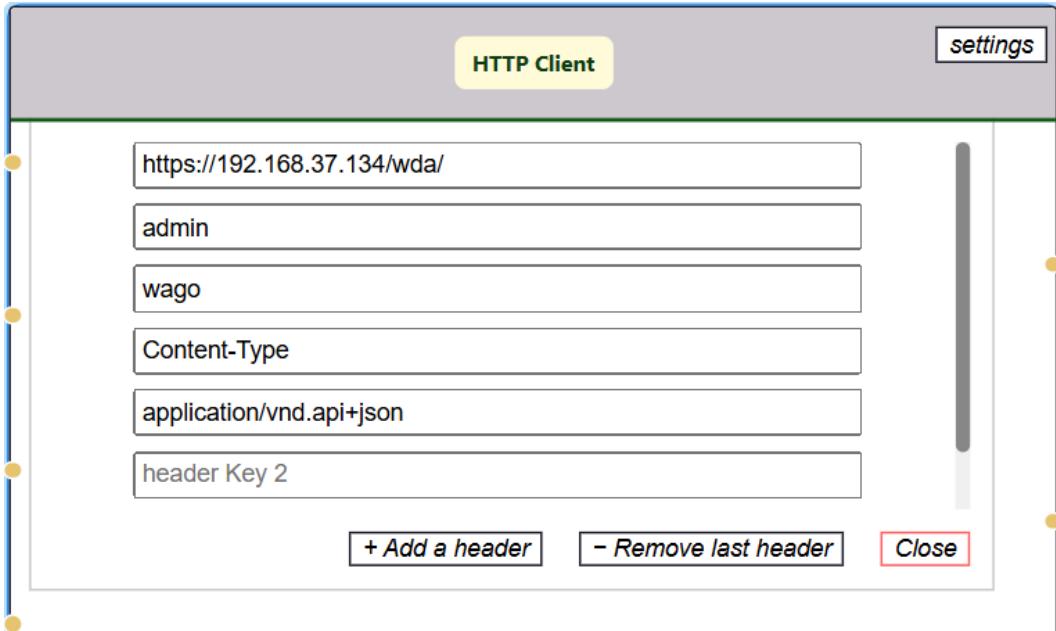


Fig. 105. – **HTTP Client** : vue programmation – configuration du bloc pour **WDA**

PATCH <https://192.168.37.134/wda/parameters/0-0-io-channels-9-dovalue>

Params Headers 1 Auth ● Body ●

```
1 ▼ {
2   ▼ "data": {
3     "id": "0-0-io-channels-9-dovalue",
4     "type": "parameters",
5   ▼ "attributes": {
6     "value": true
7   }
8 }
9 }
```

Request PATCH Response 204

▼ HTTP/1.1 204 No Content

Cache-Control	no-store
Connection	close
Content-Security-Policy	default-src 'self'
Date	Thu, 17 Jul 2025 15:23:46 GMT
Pragma	no-cache
Referrer-Policy	no-referrer

Fig. 106. – HTTP Client : HTTPIe – activation – requête envoyée et réponse reçue

The screenshot shows the HTTPie interface with a request and response displayed.

Request:

```
PATCH https://192.168.37.134/wda/parameters/0-0-io-channels-9-dovalue
```

Body:

```
1 ▼ {  
2 ▼   "data": {  
3       "id": "0-0-io-channels-9-dovalue",  
4       "type": "parameters",  
5 ▼       "attributes": {  
6           "value": false  
7       }  
8   }  
9 }
```

Response:

Request PATCH Response 204

HTTP/1.1 204 No Content

Header	Value
Cache-Control	no-store
Connection	close
Content-Security-Policy	default-src 'self'
Date	Thu, 17 Jul 2025 15:26:40 GMT
Pragma	no-cache
Referrer-Policy	no-referrer

Fig. 107. – HTTP Client : HTTPie – désactivation – requête envoyée et réponse reçue

```

POST: https://192.168.37.134/wda/monitoring-lists
Send

Params Headers 1 Auth • Body ●

1 ▼ {
2   ▼ "data": {
3     "type": "monitoring-lists",
4     ▼ "attributes": {
5       "timeout": 600
6     },
7     ▼ "relationships": {
8       "parameters": {
9         "data": [
10        { "id": "0-0-io-channels-21-divalue", "type": "parameters" },
11        { "id": "0-0-io-channels-22-divalue", "type": "parameters" },
12        { "id": "0-0-io-channels-23-divalue", "type": "parameters" },
13        { "id": "0-0-io-channels-9-dovalue", "type": "parameters" },
14        { "id": "0-0-io-channels-10-dovalue", "type": "parameters" },
15        { "id": "0-0-io-channels-11-dovalue", "type": "parameters" }
16      ]
17    }
18  }
19}
20}
21}
22}

Request POST Response 201
HTTP/1.1 201 Created (17 headers)

1 ▼ {
2   ▼ "data": {
3     "attributes": {
4       "timeout": 600
5     },
6     "id": "18",
7     "links": {
8       "self": "/wda/monitoring-lists/18"
9     },
10    "relationships": {
11      "parameters": {
12        "links": {
13          "related": "/wda/monitoring-lists/18/parameters"
14        }
15      }
16    },
17    "type": "monitoring-lists"
18  },
19  "jsonapi": {
20    "version": "1.0"
21  },
22  "meta": {
23    "doc": "/openapi/wda.openapi.html#operation/createMonitoringList",
24    "version": "1.4.1"
25  }
26}

```

Fig. 108. – **HTTP Client** : HTTPie – création d'une monitoring-list – requête envoyée et réponse reçue

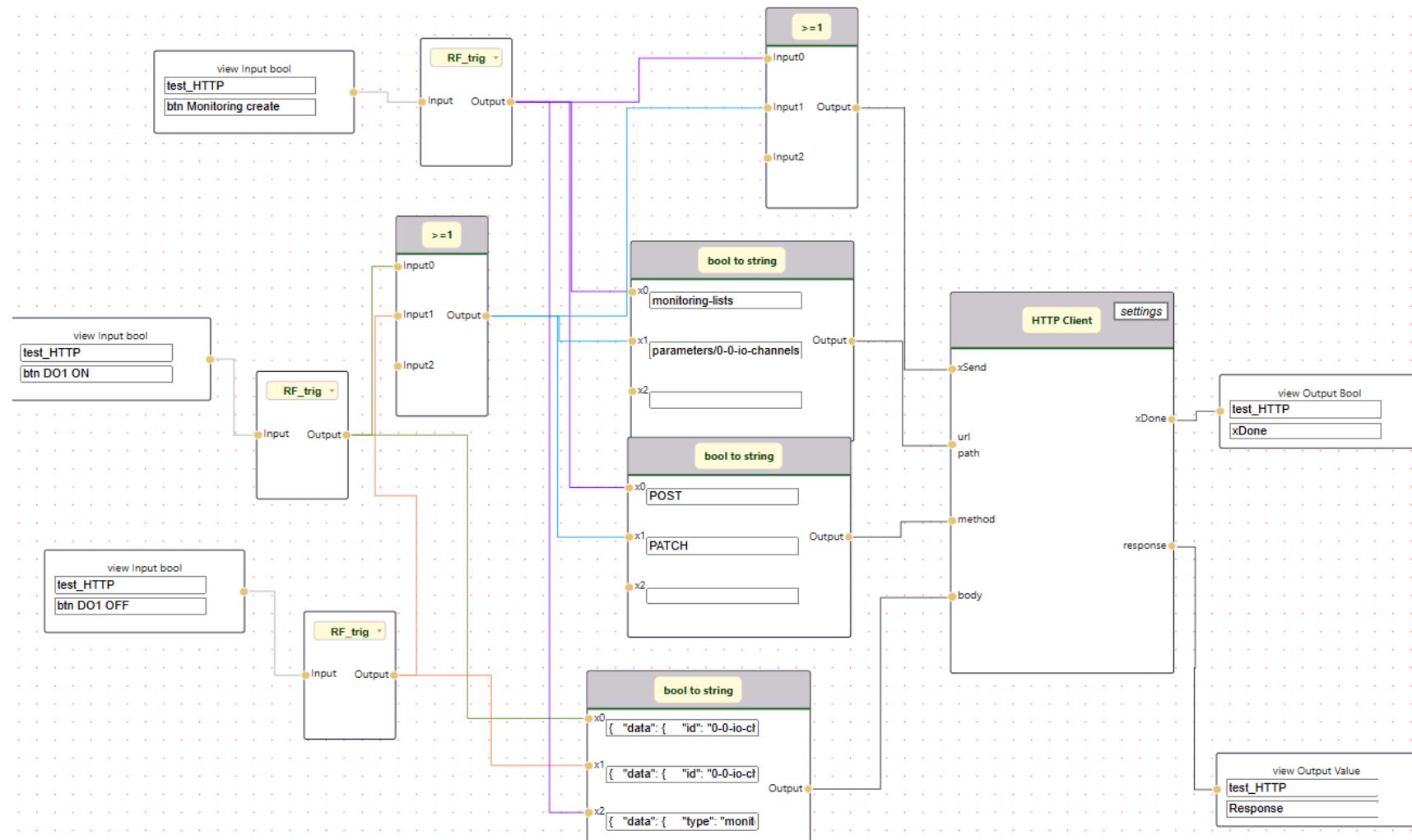
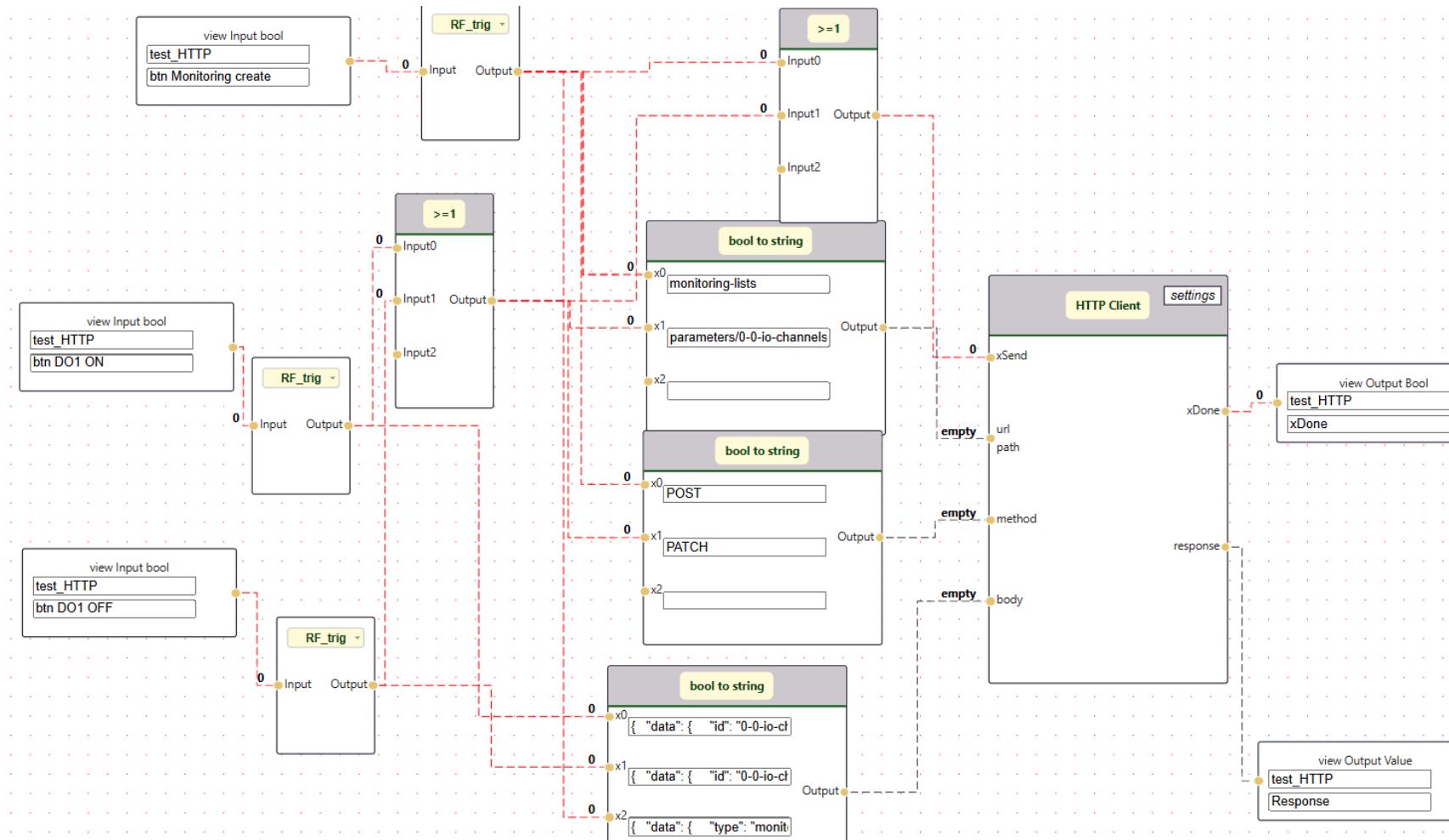


Fig. 109. – HTTP Client : vue programmation – bloc HTTP Client configuré pour WDA

Fig. 110. – **HTTP Client** : vue debug – état initial du code

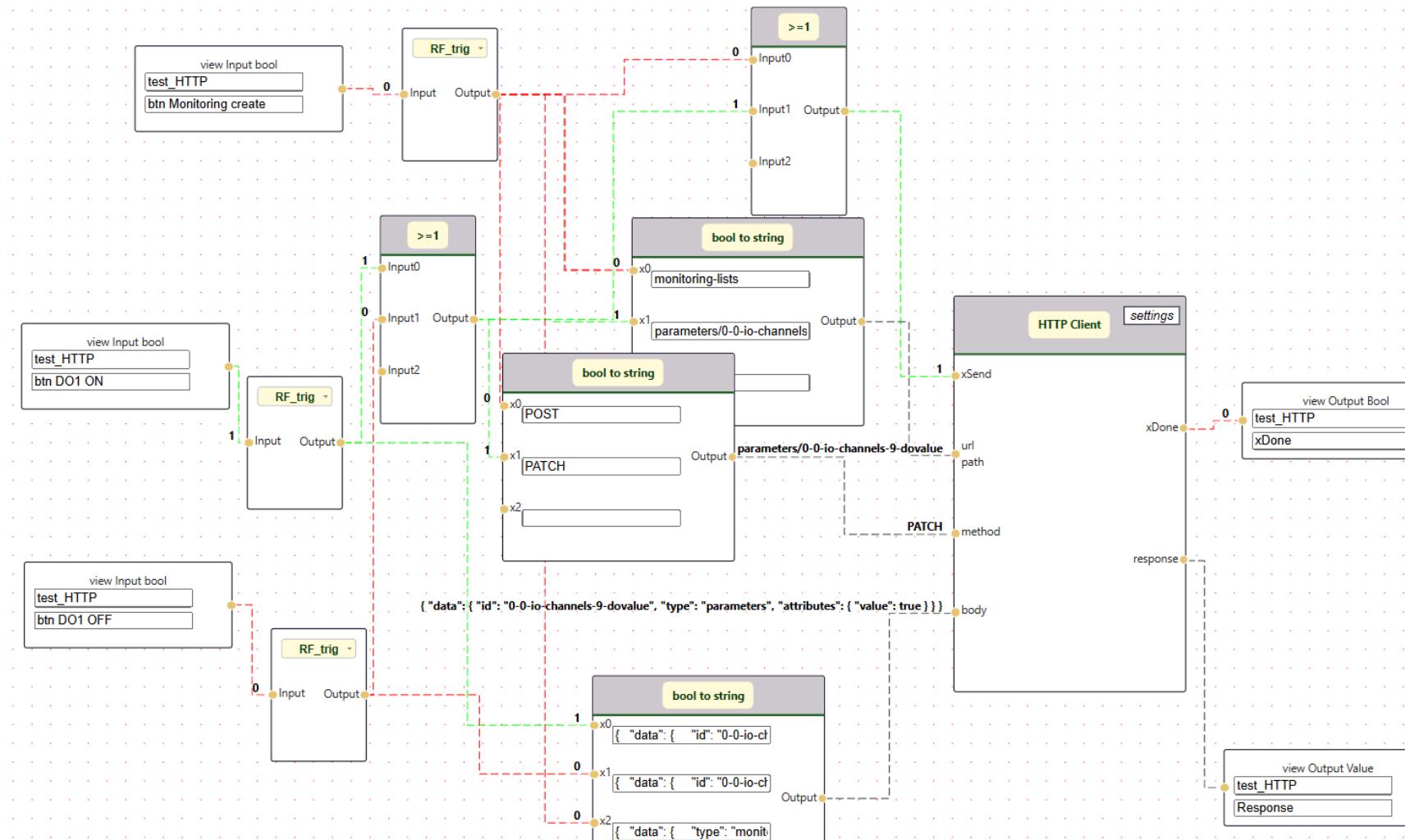


Fig. 111. – HTTP Client : vue debug – activation de la sortie DIO1

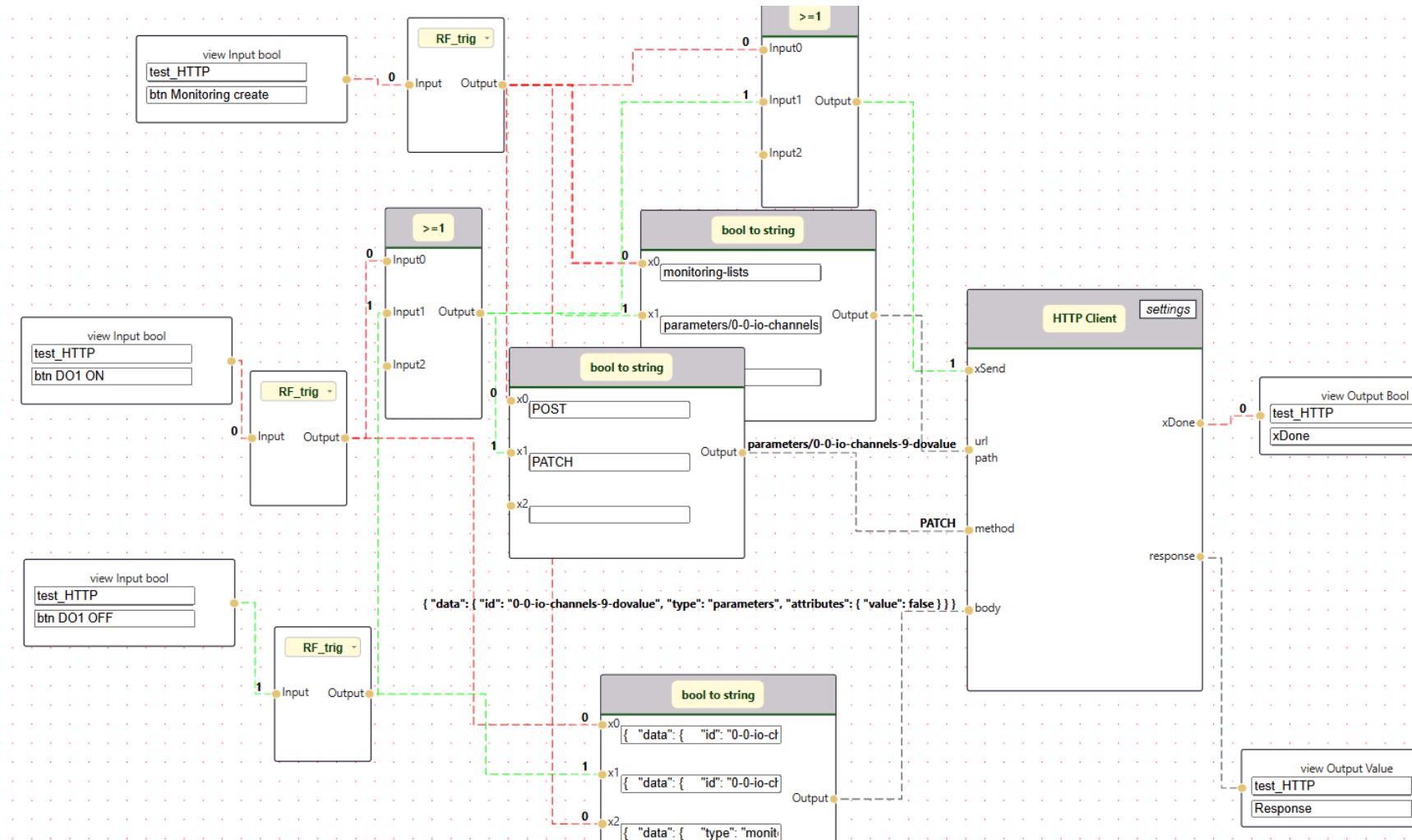


Fig. 112. – HTTP Client : vue debug – désactivation de la sortie DIO1

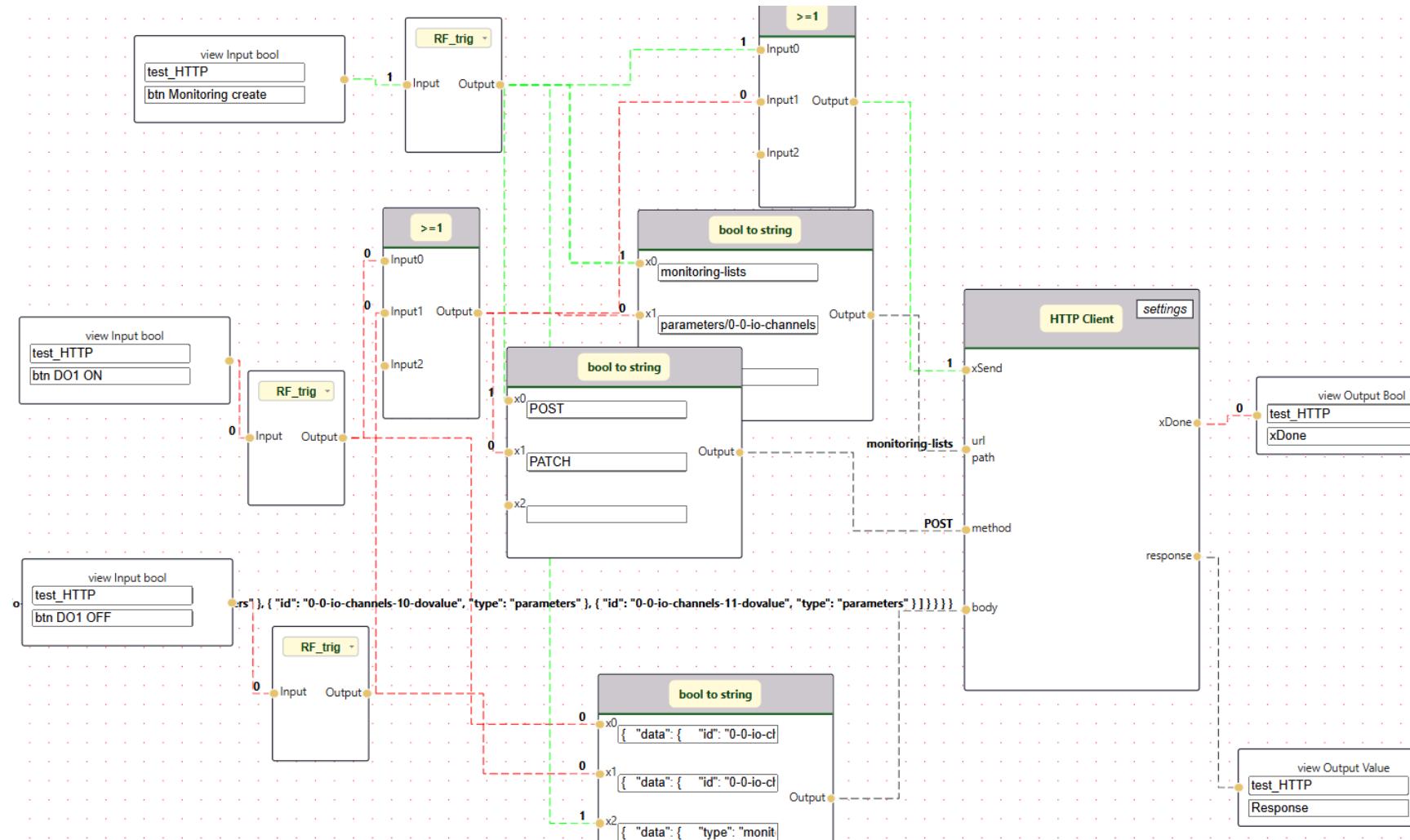


Fig. 113. – HTTP Client : vue debug – création d'une monitoring-list

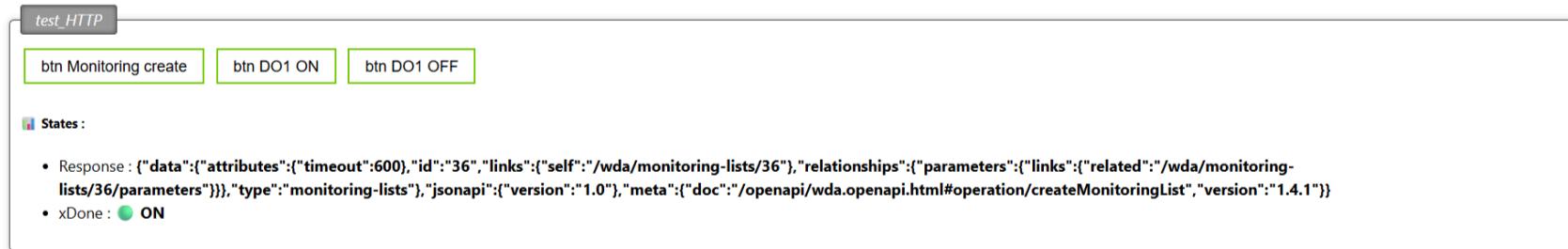


Fig. 114. – **HTTP Client** : vue utilisateur – réponse à la création d'une monitoring-list

8.5 HTTP Server : Exemples

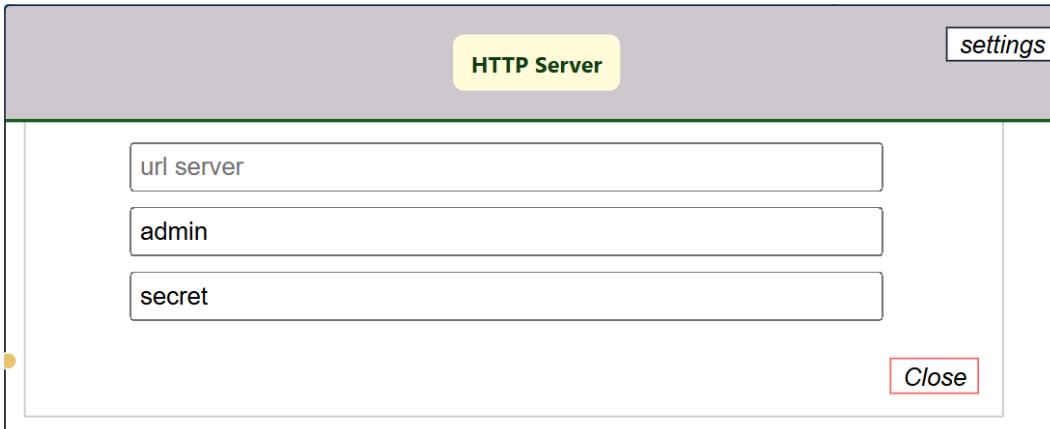


Fig. 115. – HTTP Server : Vue programmation – configuration du bloc

L'exemple suivant montre comment utiliser le bloc **HTTP Server**.

La figure Fig. 116 présente la vue debug initiale du bloc **HTTP Server**.

En Fig. 117, on observe une requête **POST** envoyée avec **HTTPPie**, tandis que la figure Fig. 118 montre la vue debug au moment de la réception de cette requête.

De même, la figure Fig. 119 illustre une requête **PATCH** envoyée avec **HTTPPie**, et Fig. 120 montre la réception de cette requête côté debug. On y remarque que la lampe "**D03**" s'allume, car le paramètre **param3 = true**.

La figure Fig. 121 montre une requête **PUT** envoyée avec **HTTPPie**. Sa réception est illustrée dans la figure Fig. 122. Enfin, en Fig. 123, on constate que les paramètres envoyés avec la requête **PUT** ne sont pas actifs. Cela démontre la possibilité d'avoir des comportements dynamiques selon la nature ou la présence des paramètres.

La figure Fig. 124 illustre l'envoi d'une requête **GET** avec un chemin arbitraire (ici **short1**). Ce mécanisme est particulièrement utile pour gérer des **appliances** qui n'envoient ni *body* ni *headers* dans leurs requêtes.

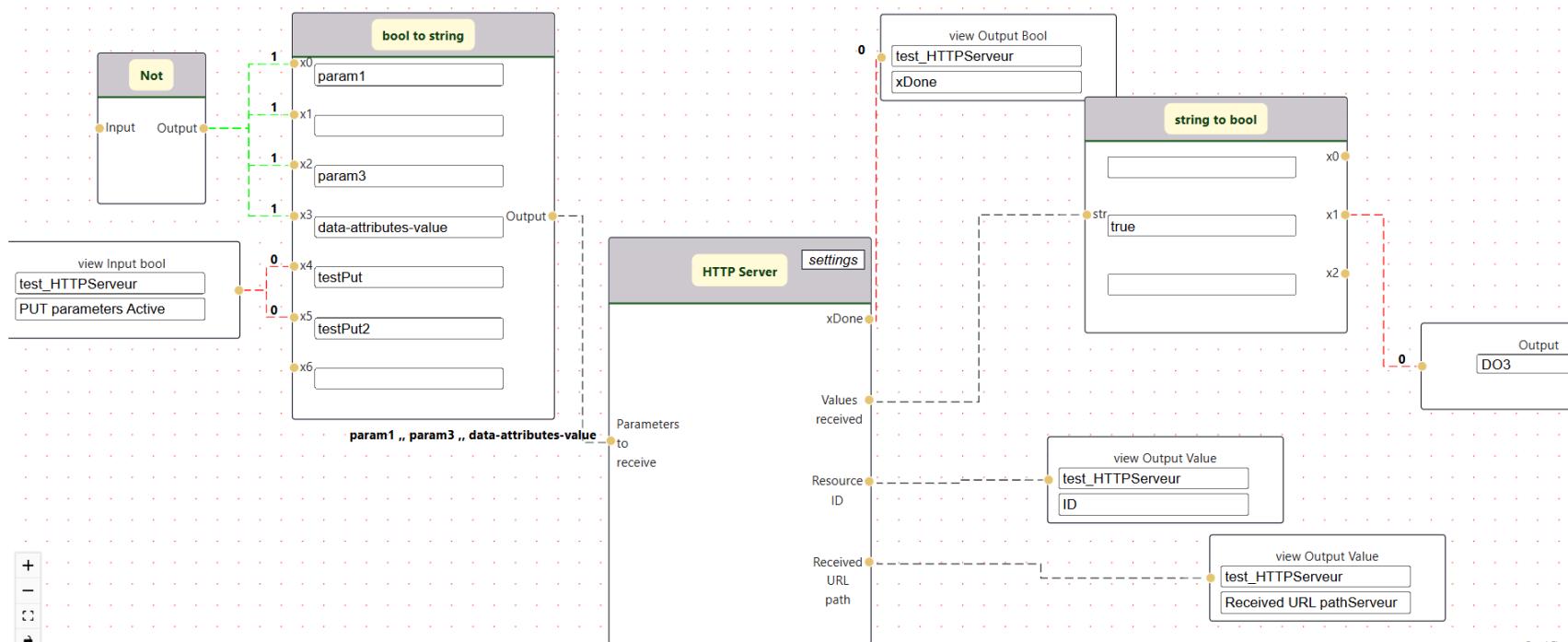


Fig. 116. – HTTP Server : Vue debug - initiale

The screenshot shows the HTTPie interface. The URL is `http://192.168.39.56:8080/flatten`. The Request tab shows a POST method. The Response tab shows a 200 OK status with the following JSON content:

```
HTTP/1.1 200 OK (4 headers)

{
    "id": 2,
    "result": {
        "data-attributes-value": true,
        "param1": "value1",
        "param2": "value2",
        "param3": "value3"
    }
}
```

Fig. 117. – HTTP Server : HTTPie - Crédit d'une ressource avec HTTPie (POST)

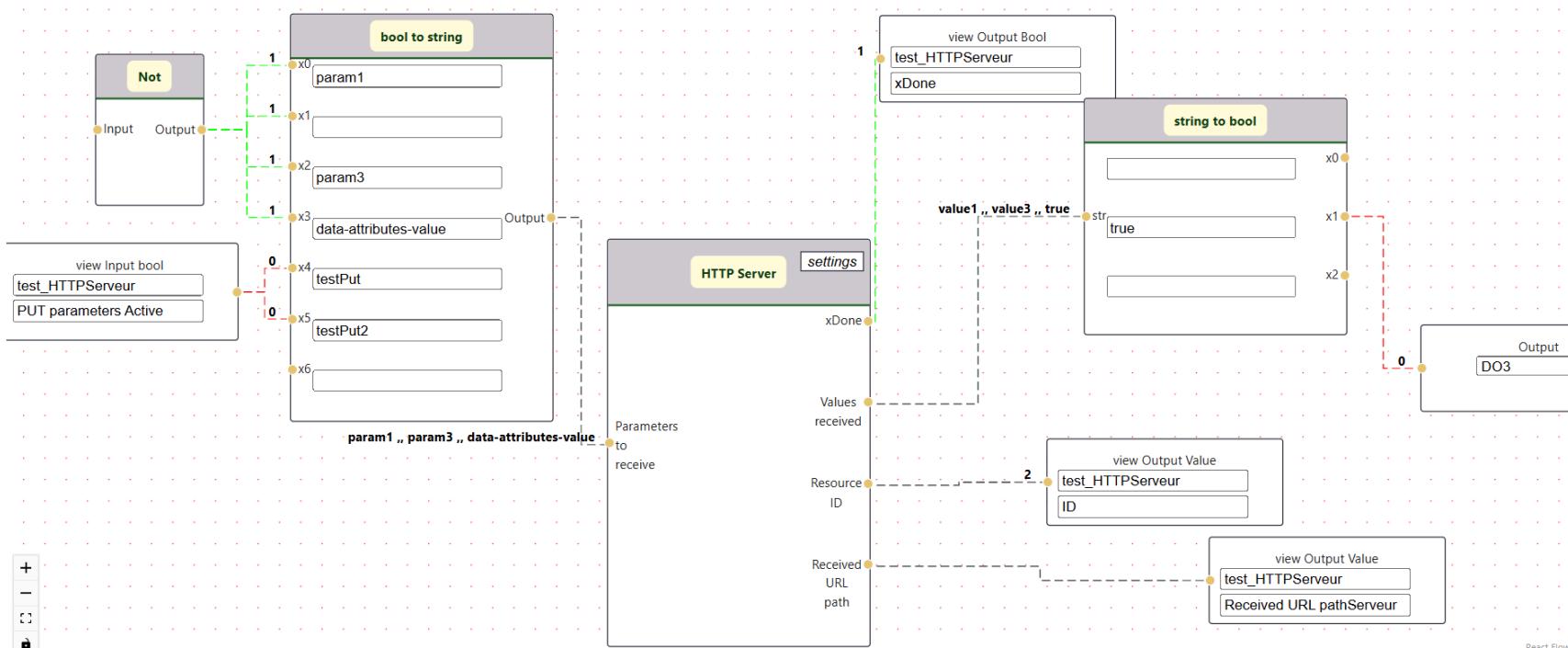


Fig. 118. – HTTP Server : Vue debug - Crédit d'une ressource avec HTTPie (POST)

PATCH ↗ http://localhost:8080/parameters/flatten/2

Params	Headers 1	Auth	Body	Request	Response
				PATCH	200
1	▼ {			▼ HTTP/1.1 200 OK	
2	"data-attributes-value": false,			Connection	close
3	"param3": true			Content-Length	0
4	}			Date	Mon, 2
5					

```
1 ▼ {
2     "data-attributes-value": false,
3     "param3": true
4 }
```

Fig. 119. – HTTP Server : HTTPie - Modification d'une ressource (PATCH)

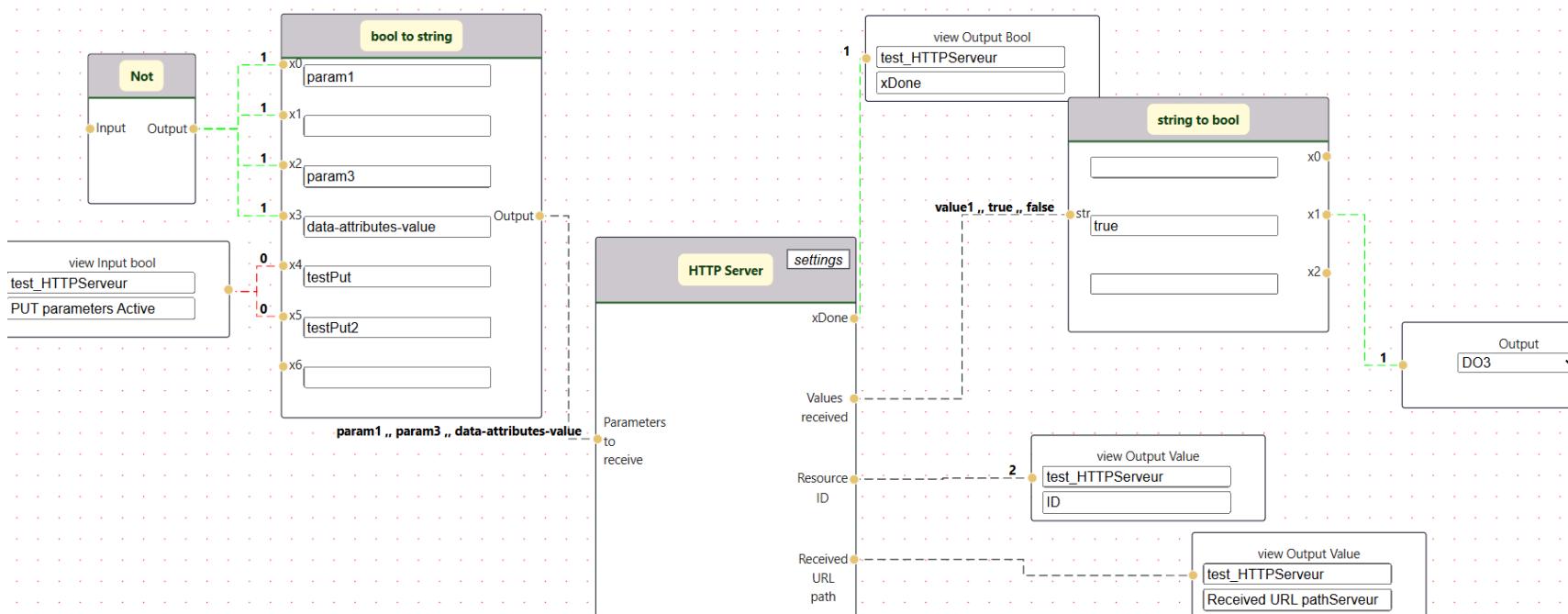


Fig. 120. – HTTP Server : Vue debug - Modification d'une ressource (PATCH)

PUT `http://192.168.39.56:8080/message`

Params Headers 1 Auth ● Body ●

```
1 ▼ {  
2     "testPut": "ValueOftestPut",  
3     "testPut2": "put2"  
4 }  
5
```

Request PUT Response 200

▼ HTTP/1.1 200 OK

Connection	close
Content-Length	0
Date	Mon, 21 Jul 2025 11:33:12 GMT

Fig. 121. – HTTP Server : HTTPie - PUT

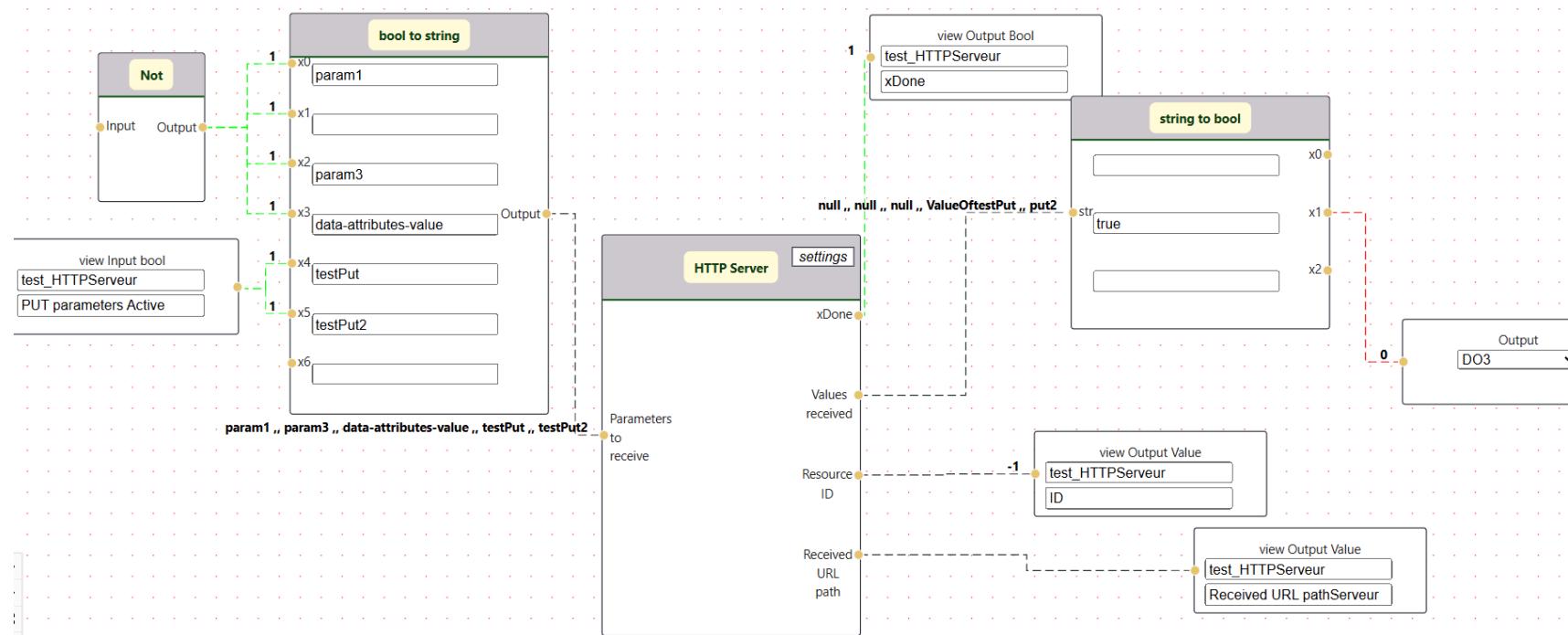


Fig. 122. – HTTP Server : Vue debug - PUT

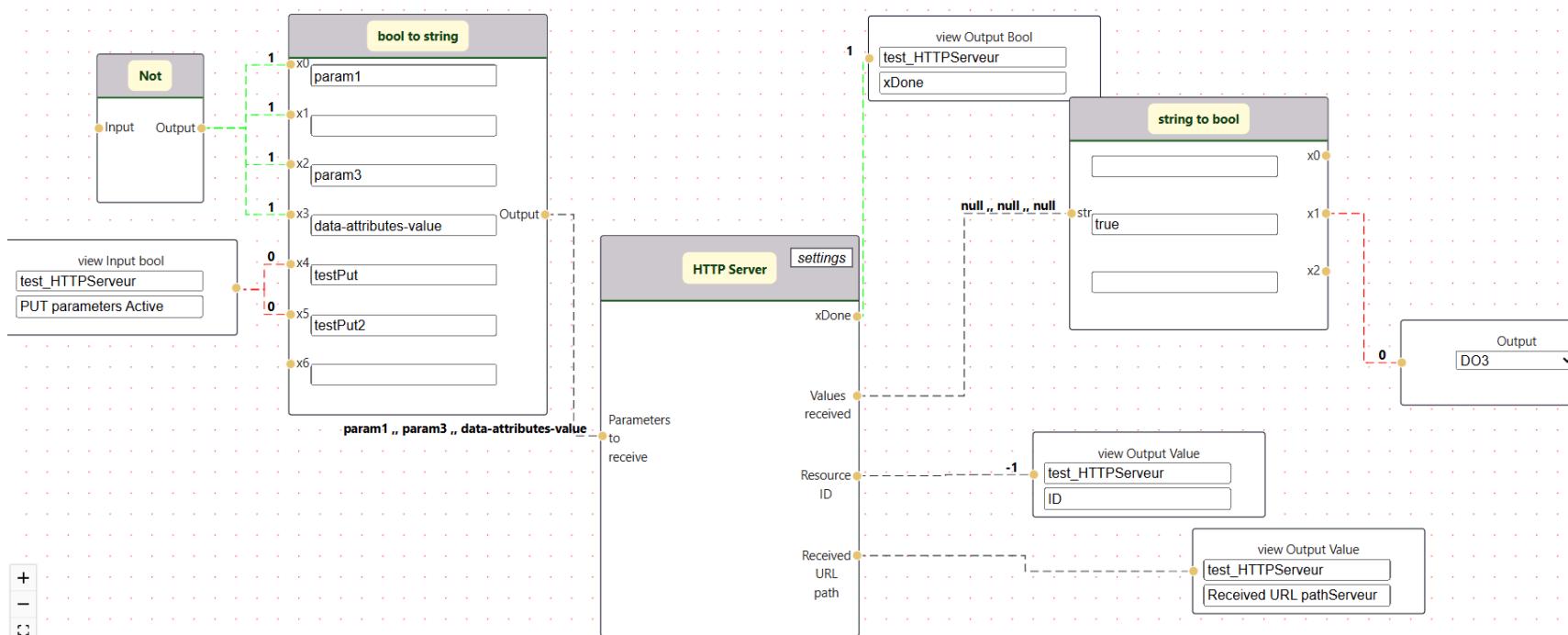


Fig. 123. – HTTP Server : Vue debug - PUT avec les paramètres non actifs

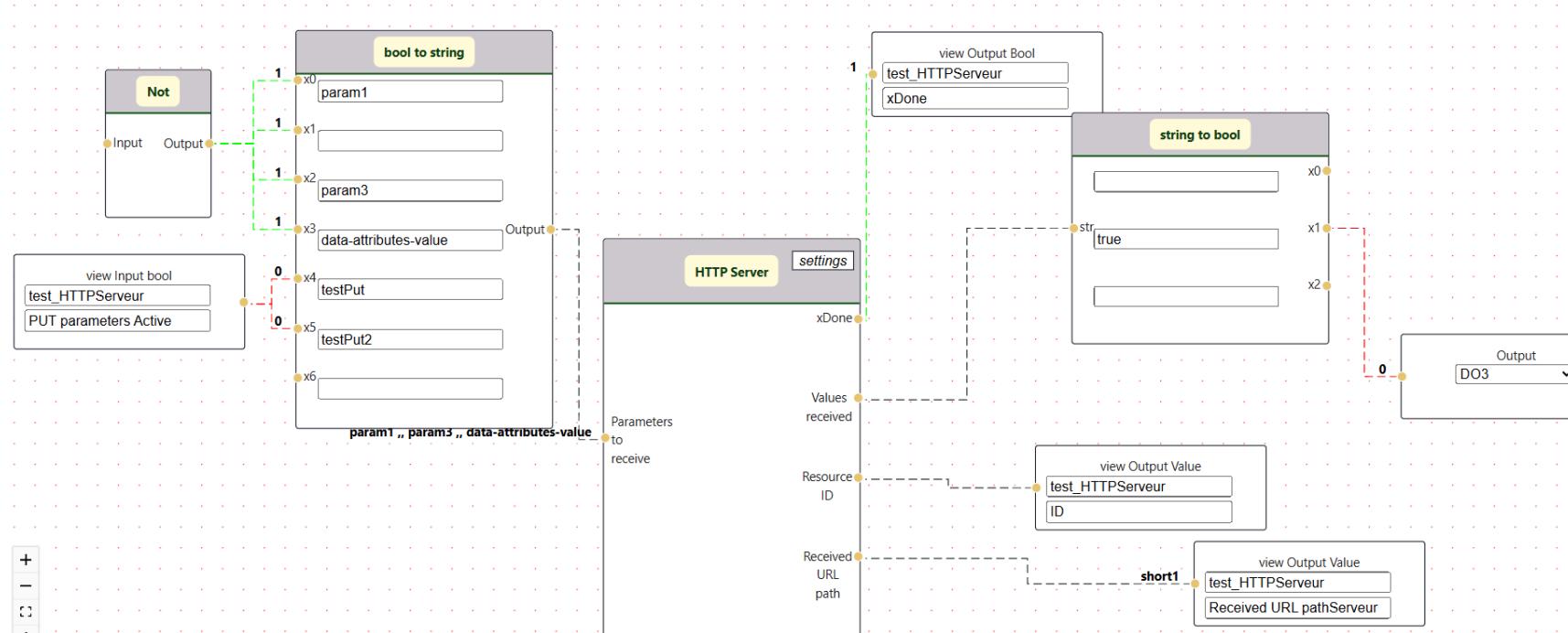


Fig. 124. – HTTP Server : Vue debug - GET avec un path quelconque ici 192.168.39.56:8080/short1

8.6 Home-IO

6

Garage

Write Coils (0x15)	0	Bool	<i>off</i>	<i>on</i>	-	Light Passage
	1	Bool	<i>off</i>	<i>on</i>	-	Light Garage
	2	Bool	<i>stop</i>	<i>up</i>	-	Move Shutters Up
	3	Bool	<i>stop</i>	<i>down</i>	-	Move Shutters Down
	4	Bool	<i>stop</i>	<i>up</i>	-	Open Garage Door
	5	Bool	<i>stop</i>	<i>down</i>	-	Close Garage Door
	0	Bool	<i>idle</i>	<i>pressed</i>	-	Button Passage
	1	Bool	<i>idle</i>	<i>pressed</i>	-	Button Garage
	2	Bool	<i>idle</i>	<i>pressed</i>	-	Garage Door Control Up Button
	3	Bool	<i>idle</i>	<i>pressed</i>	-	Shutter Control Up Button
	4	Bool	<i>idle</i>	<i>pressed</i>	-	Garage Door Control Down Button
	5	Bool	<i>idle</i>	<i>pressed</i>	-	Shutter Control Down Button
	6	Bool	<i>undefined</i>	<i>open</i>	-	Shutters Open
Read Discrete Inputs (0x02)	7	Bool	<i>undefined</i>	<i>closed</i>	-	Shutters Closed
	8	Bool	<i>undefined</i>	<i>open</i>	-	Garage Door Open
	9	Bool	<i>undefined</i>	<i>closed</i>	-	Garage Door Closed
	10	Bool	<i>idle</i>	<i>blocked</i>	-	Garage IR Detector
	11	Bool	<i>idle</i>	<i>motion</i>	-	Motion Detector
	12	Bool			-	Light Sensor
	0	UInt16	0(closed)	1000(open)	%	Shutter Position
	1	UInt16	0	100	%	Brightness level
	0	UInt16	0	100	%	Light Level Passage
	1	UInt16	0	100	%	Light Level Garage

Fig. 125. – Home-IO : registres de l'interface de garage

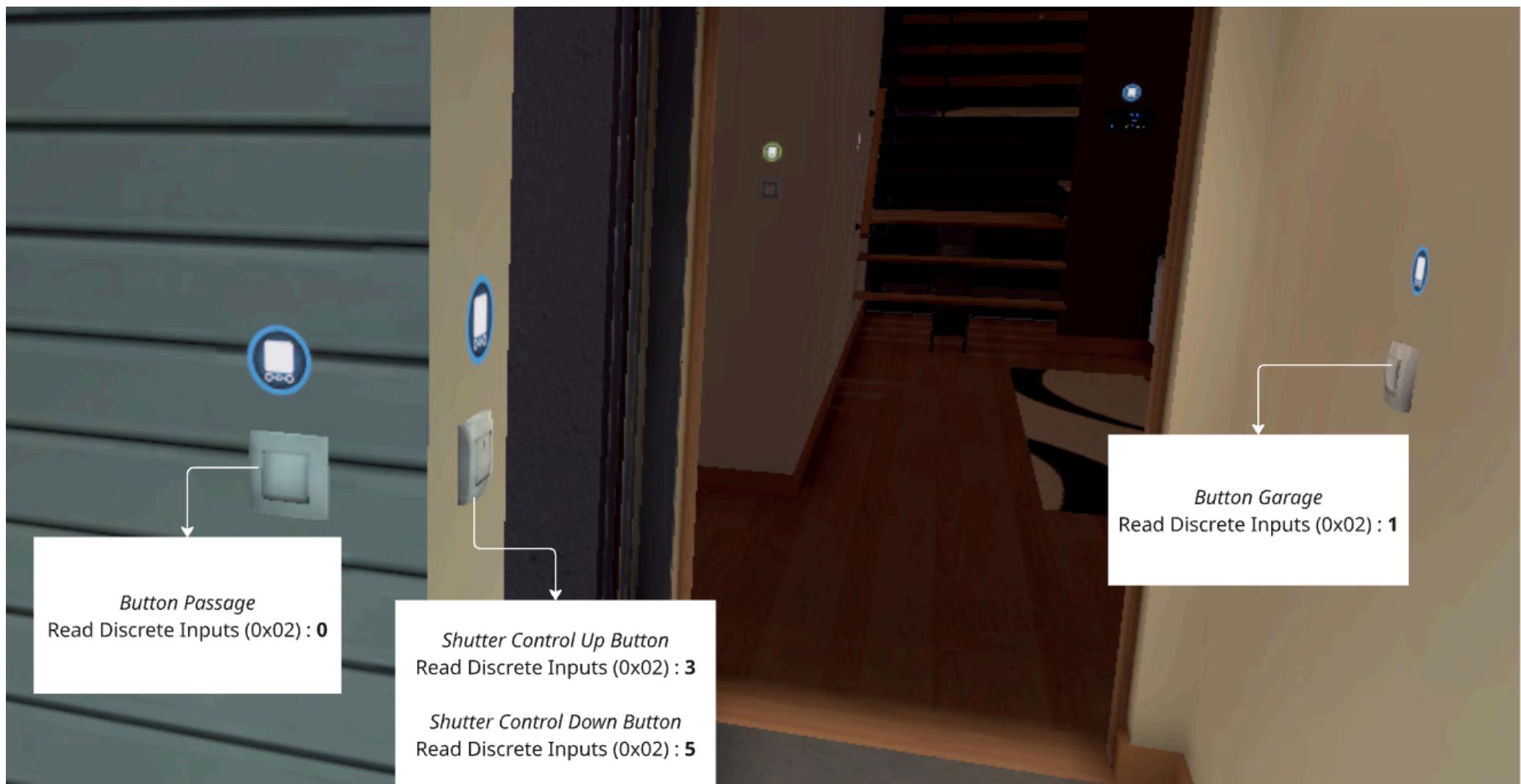


Fig. 126. – Home-IO : emplacement des boutons dans le garage – 1

8.7 Exemples Modbus Read Bool

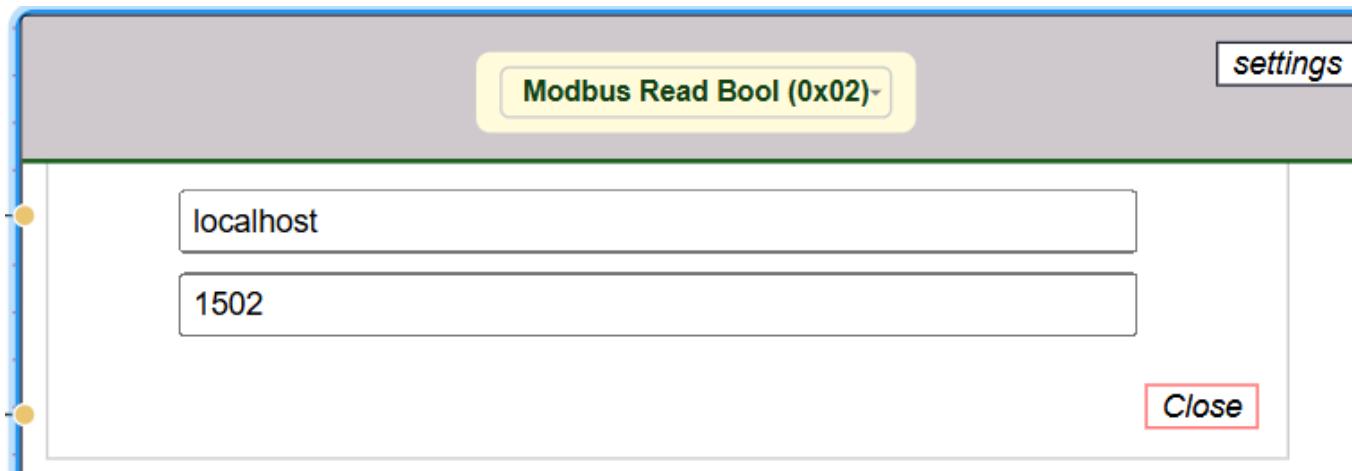
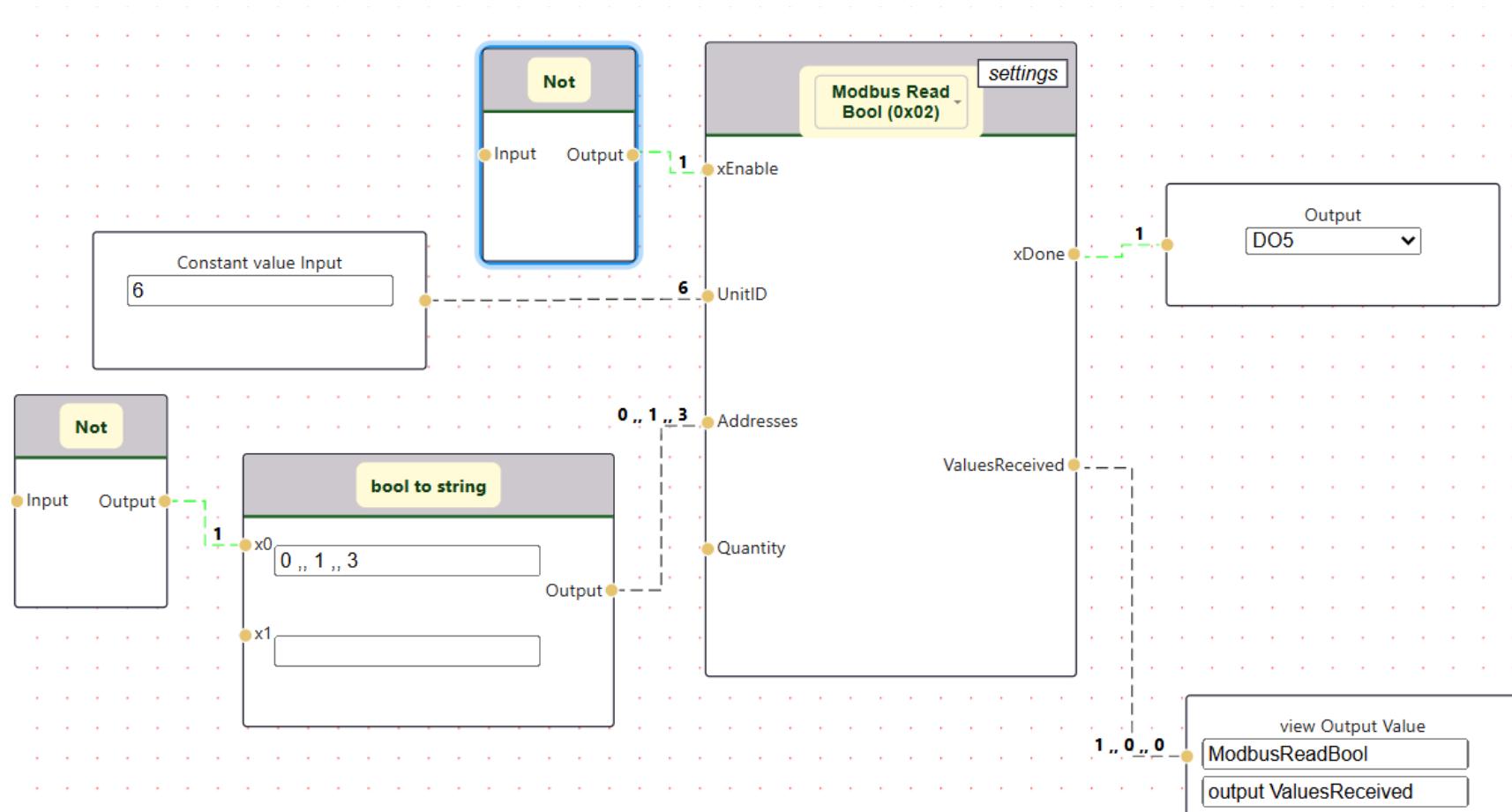


Fig. 127. – Modbus Read Bool : vue programmation – configuration du bloc

8.7.1 Exemple 1 – Modbus Read Bool – sans Quantity

Dans cet exemple, l'entrée *Quantity* n'étant pas renseignée, on lit un seul registre aux adresses 0, 1 et 3 de l'unité « garage ». Cela correspond à la lecture des états des boutons présentés en Fig. 126.

Fig. 128. – Modbus Read Bool : vue debug – exemple 1 sans Quantity

8.7.2 Exemple 2 – Modbus Read Bool – avec Quantity

Cet exemple réalise exactement la même opération que l'exemple précédent, mais cette fois avec l'entrée *Quantity* renseignée. On lit donc toujours les registres aux adresses 0, 1 et 3 de l'unité « garage ». À noter que le bloc « constant value Input » peut être remplacé par un bloc de type « bool to string » et inversement.

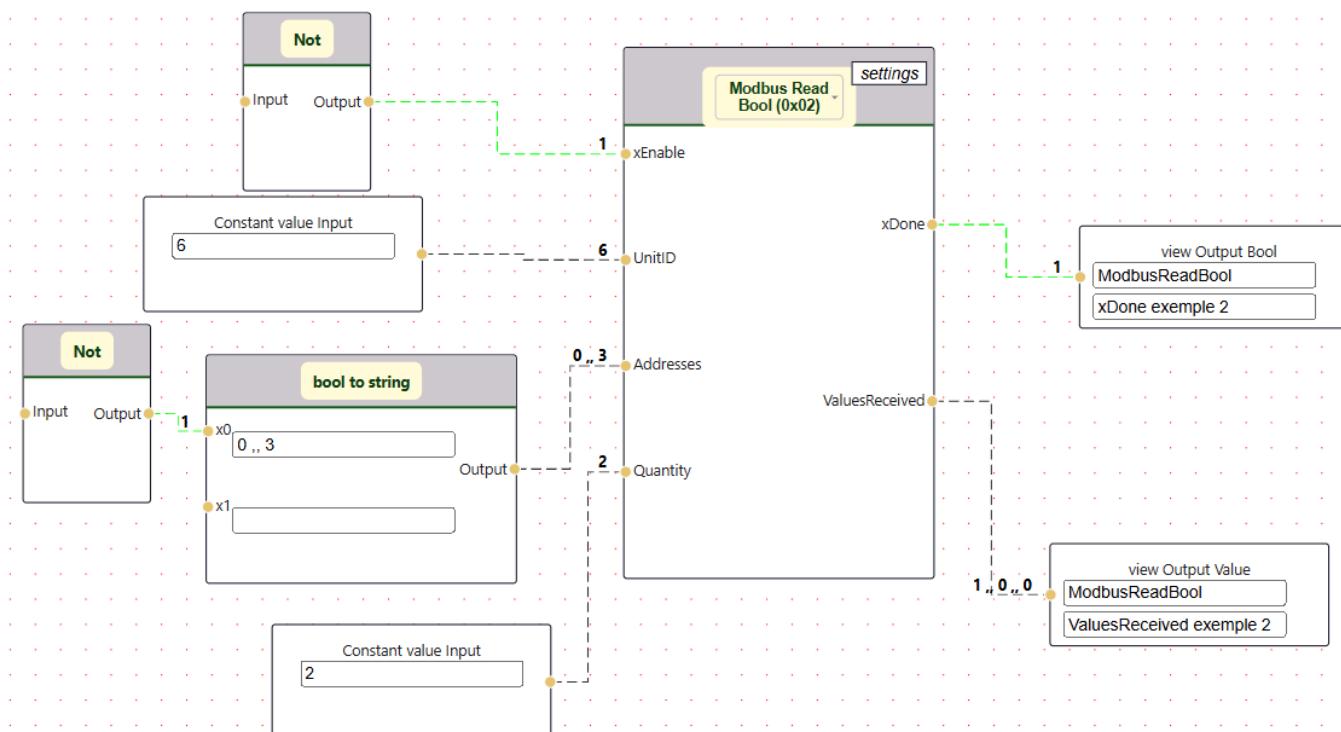


Fig. 129. – Modbus Read Bool : vue debug – exemple 2 avec *Quantity*

8.7.3 Exemple 3 – Modbus Read Bool – démonstration complète de la différence entre Quantity et Addresses

Cet exemple lit les registres aux adresses 0, 1, 3, 4 et 5 de l'unité « garage ». Le *Quantity* fixé à 7 n'est pas pris en compte car toutes les adresses sont spécifiées via *Addresses*.

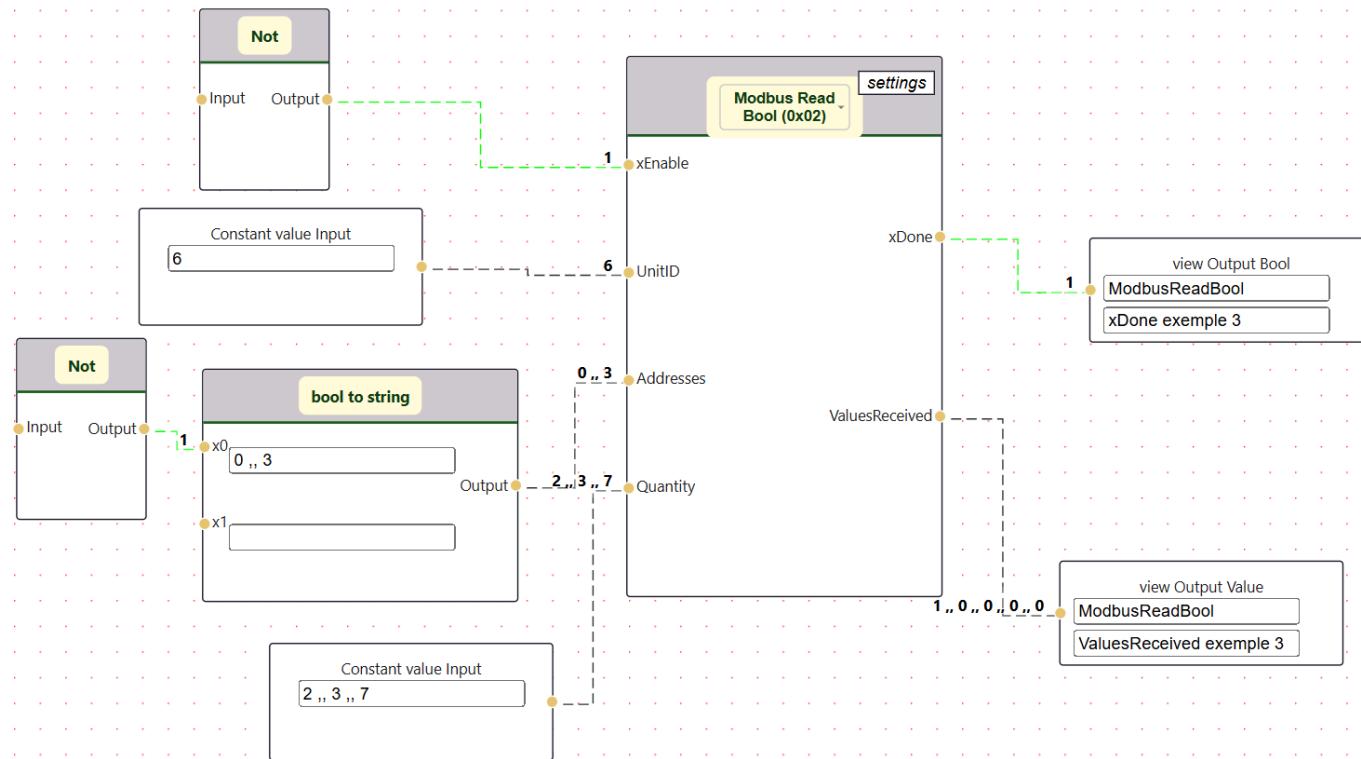


Fig. 130. – Modbus Read Bool : vue debug – exemple 3 avec Quantity

8.7.4 Exemple 4 – Modbus Read Bool – sans Adresses

Dans cet exemple, les registres aux adresses 0 et 1 de l'unité « garage » sont lus. Le *Quantity* fixé à 2 est pris en compte car *Addresses* n'est pas défini. Par défaut, l'adresse utilisée est donc 0.

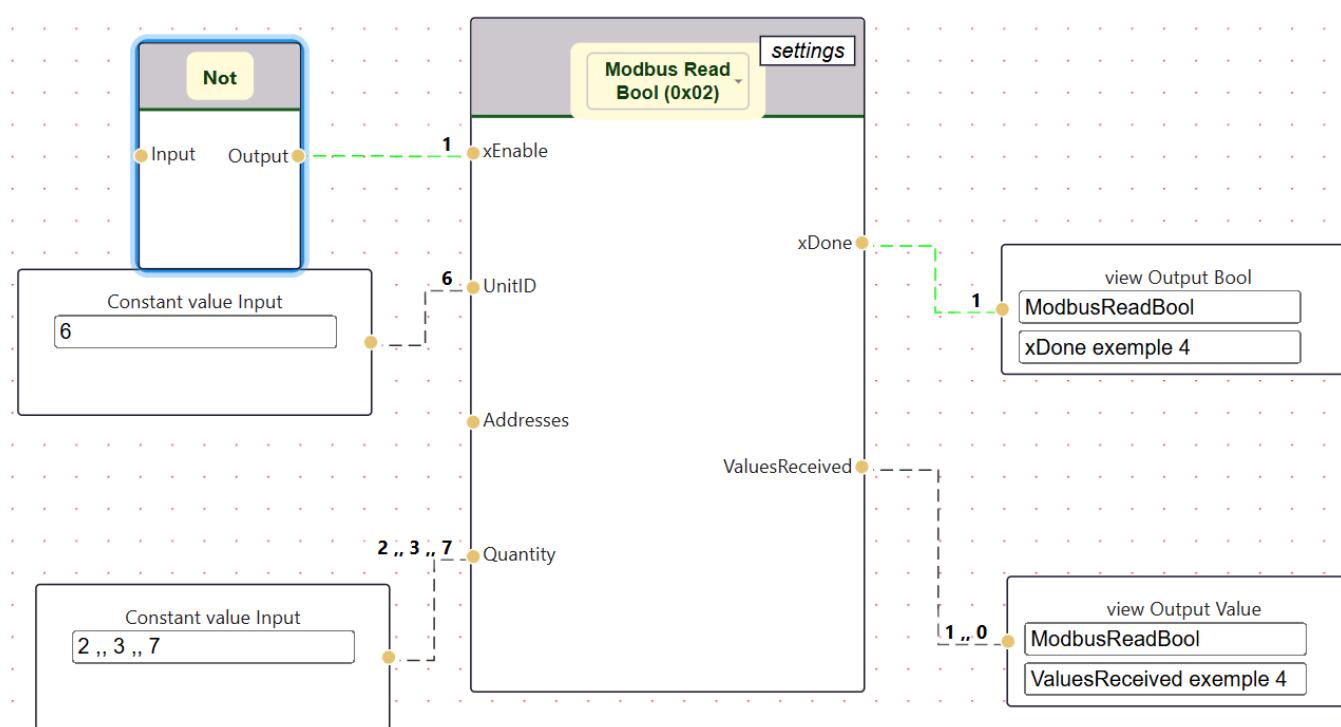


Fig. 131. – Modbus Read Bool : vue debug – exemple 4 avec **Quantity**

8.8 Exemple Modbus Read Value

Avec *Home-IO*, une seule valeur est renvoyée, donc la différence entre *Quantity* et *Addresses* n'a pas d'effet visible. Toutefois, la logique reste la même que pour *Modbus Read Bool*.

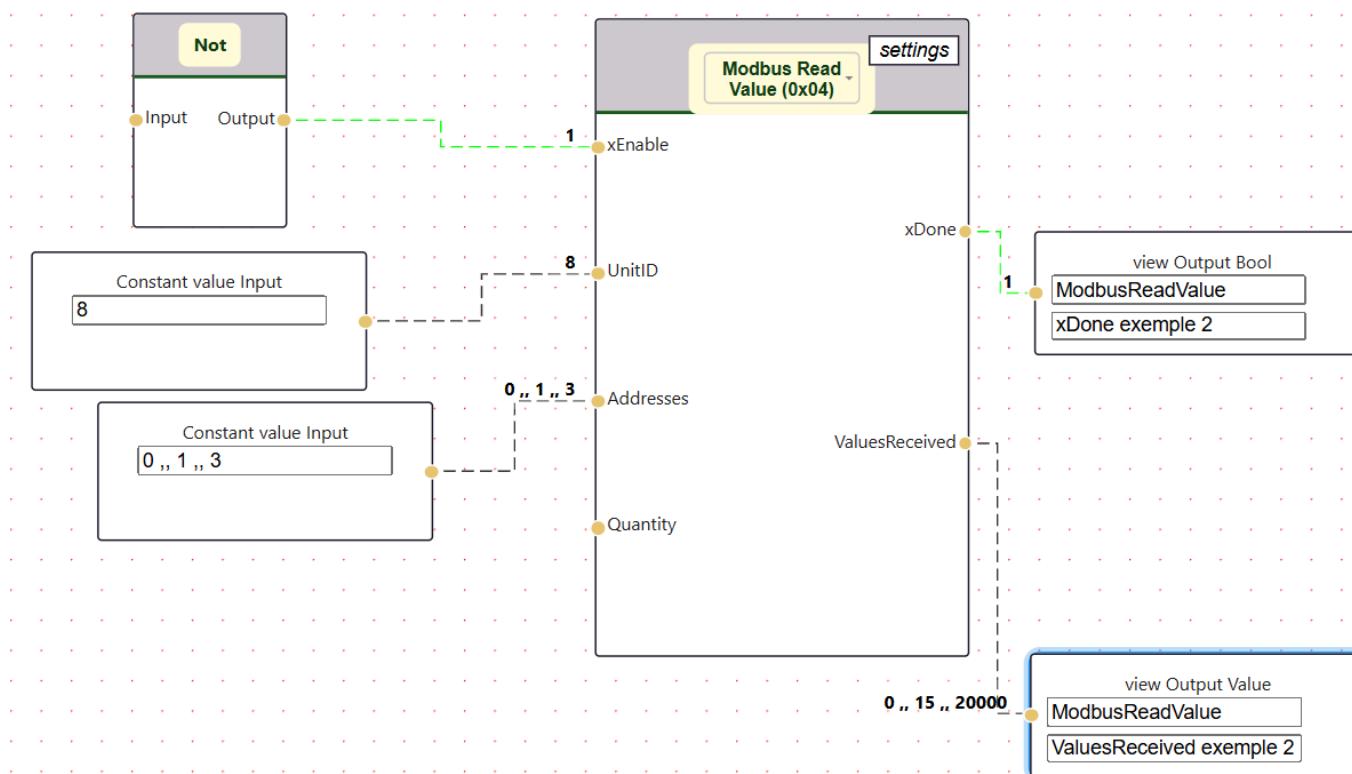


Fig. 132. – Modbus Read Value : vue debug – exemple 1 sans Quantity

8.9 Exemple Modbus Write Bool

Cet exemple permet d'allumer les lampes du garage et du passage de l'unité « garage » via la vue **User WebSocket**. En Fig. 136, la sortie *ValueReceived* affiche *1 „ 1*, ce qui signifie que deux requêtes Modbus ont été faites et que, dans chaque cas, un seul *coil* a été écrit.

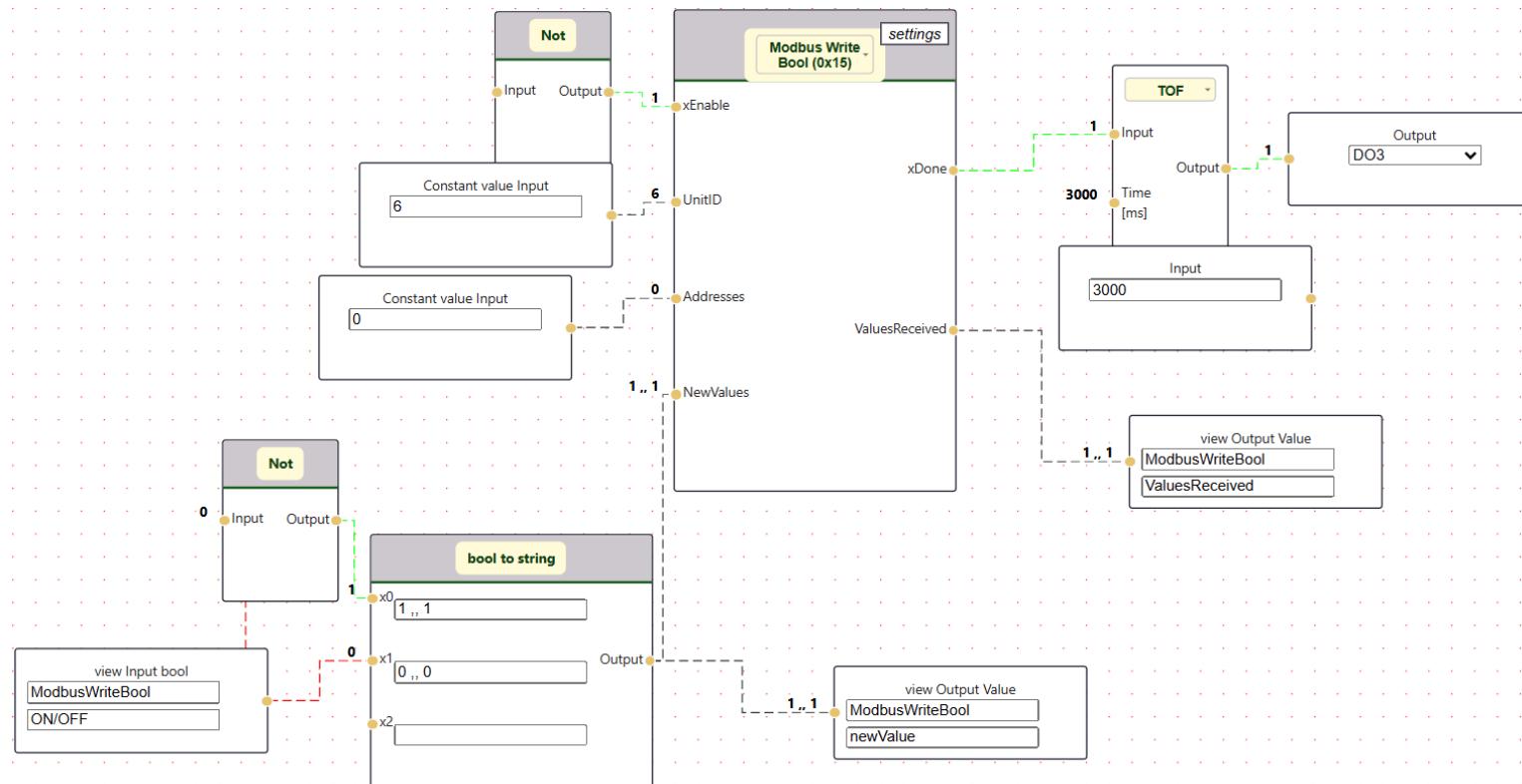


Fig. 133. – Modbus Write Bool : vue debug – lampes allumées (« on »)

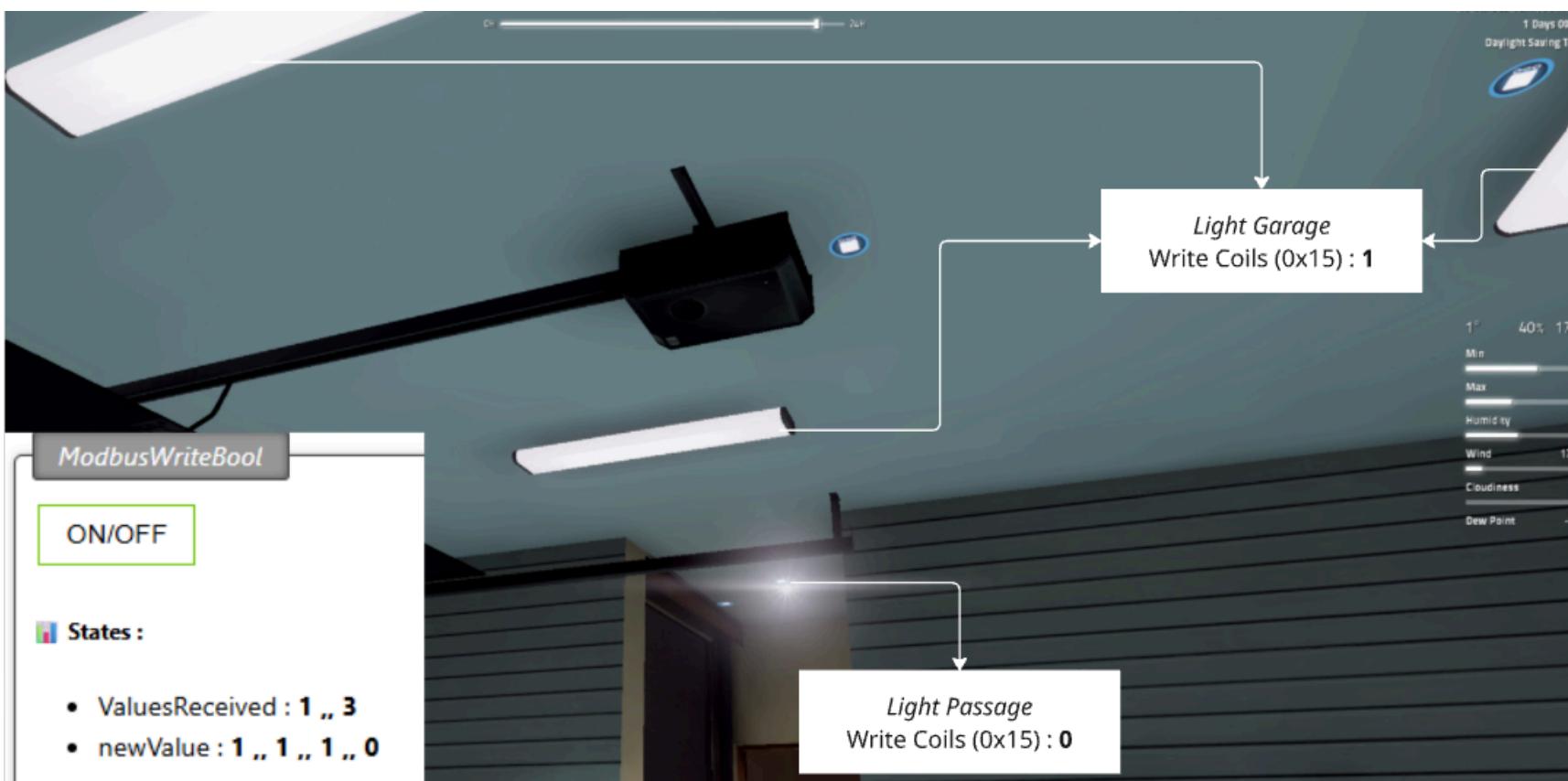


Fig. 134. – Modbus Write Bool : résultat du point de vue utilisateur – lampes « on »

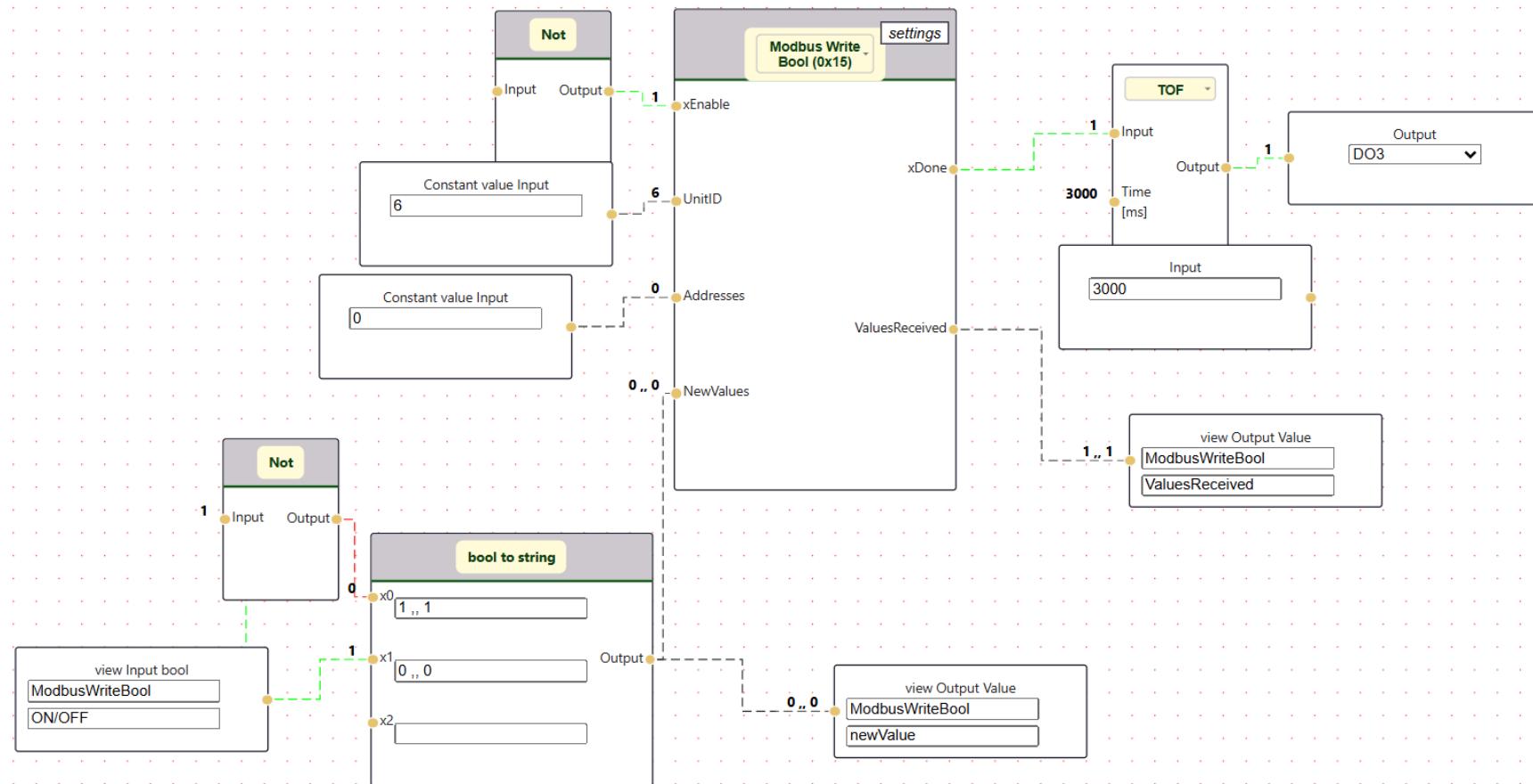


Fig. 135. – Modbus Write Bool : vue debug – lampes éteintes (< off >)

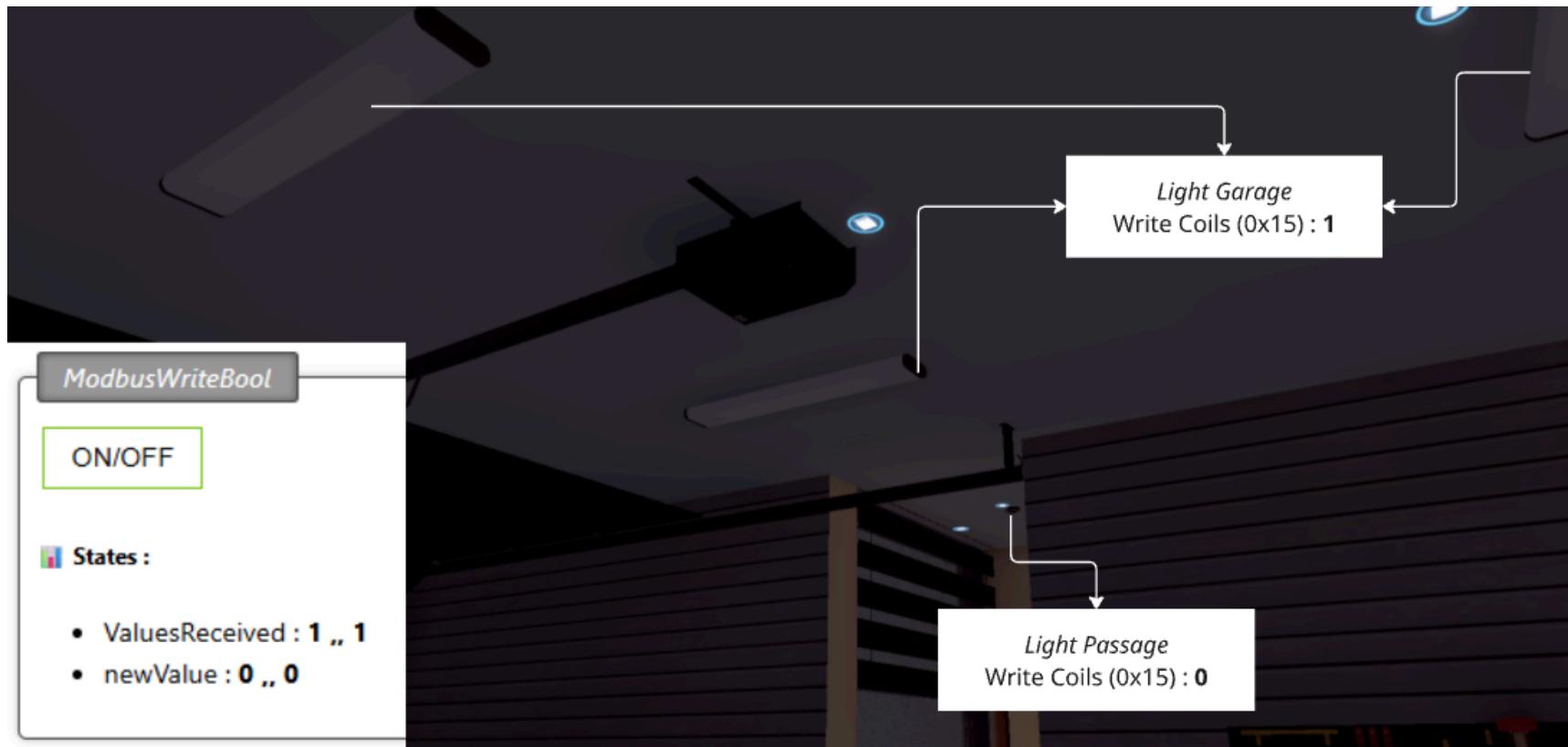


Fig. 136. – Modbus Write Bool : résultat du point de vue utilisateur – lampes « off »

8.10 Exemple Modbus Write Value

Cet exemple permet de régler l'intensité des lampes du garage via la vue **User WebSocket**.

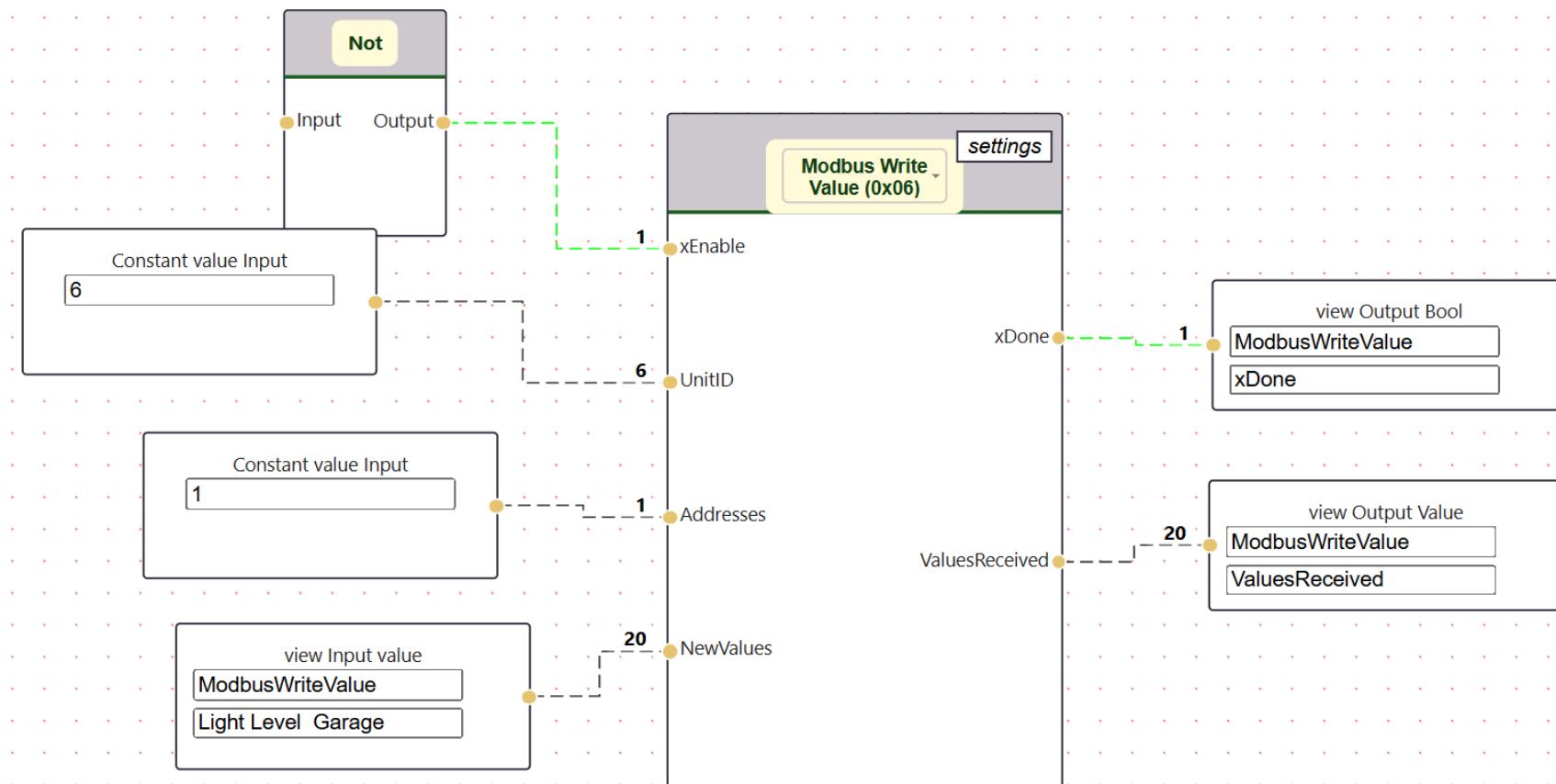


Fig. 137. – Modbus Write Value : vue debug – configuration du bloc



Fig. 138. – Modbus Write Value : résultat du point de vue utilisateur – lampes à intensités variables

9 | Application: Home Controller

9.1 Reçvoir commande bouttons - HTTP serveur

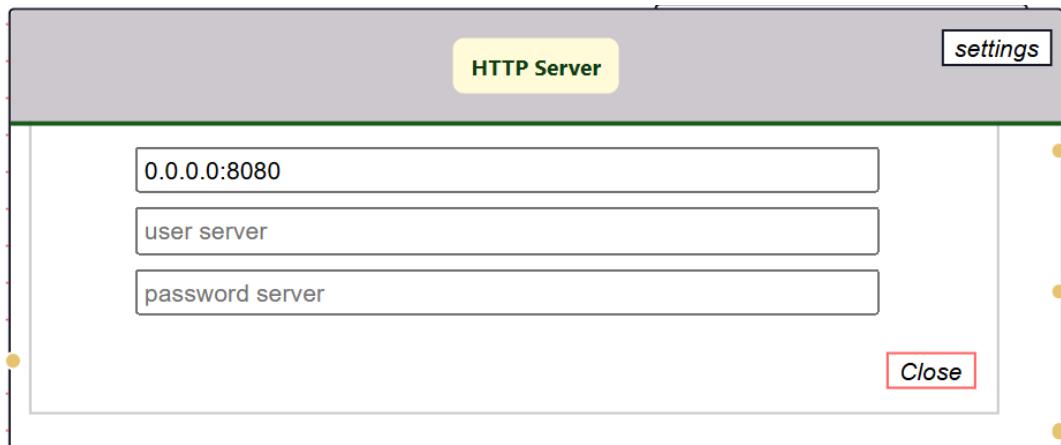


Fig. 139. – **Gestion bouttons** : vue programmation - configuration du bloc *HTTP serveur*

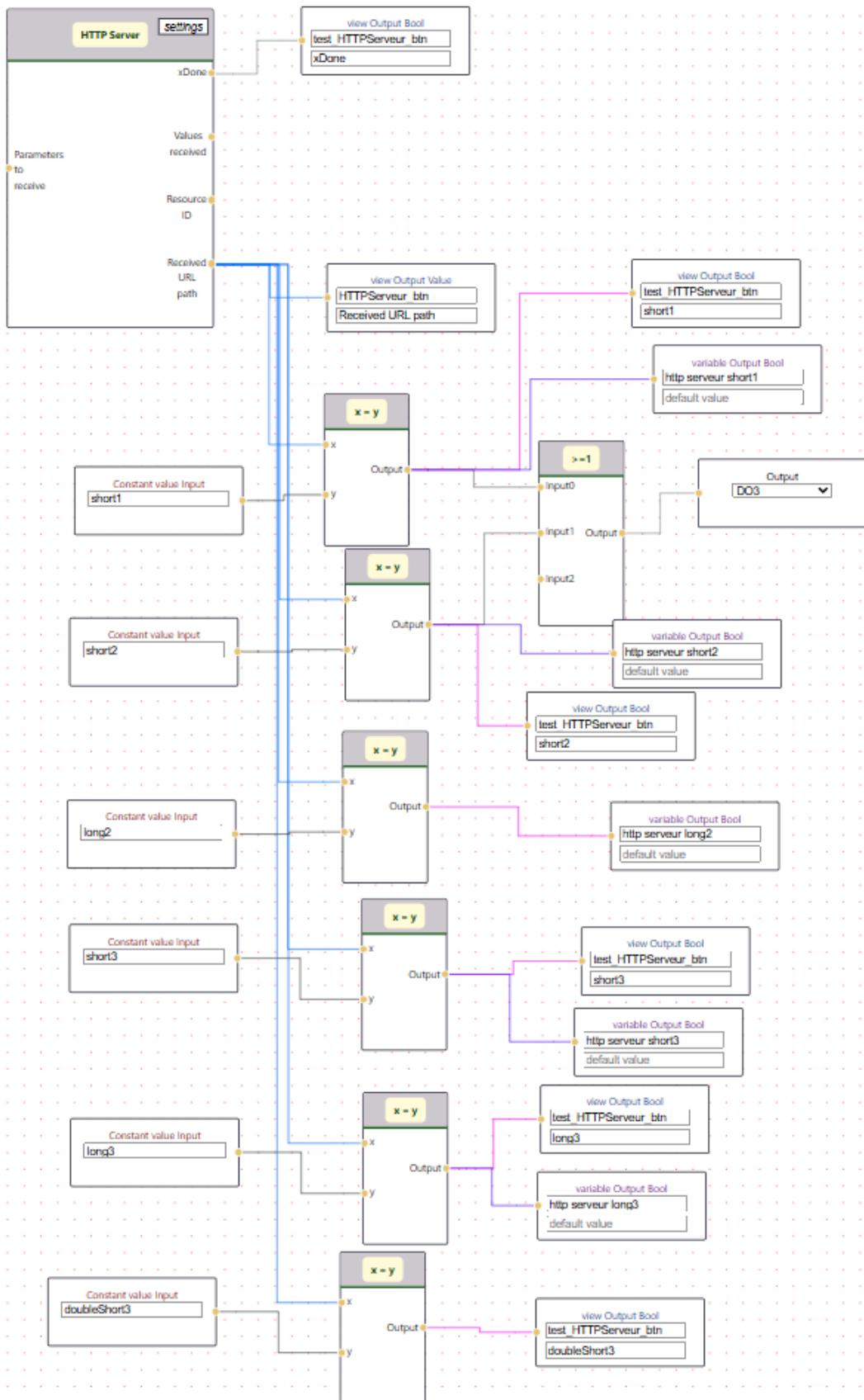


Fig. 140. – Gestion bouttons : vue programmation - complet

9.2 Gestion enclenchement du chauffage

L'idée est de faire du tout ou rien sur un relais *myStrom*. Cependant, on offre également la possibilité d'activer ou désactiver manuellement via l'interface. Sur l'input *topicToReceive* du bloc *MQTT*, on a la constante « shellies/shelly-s1-Heater-Home/sensor/temperature » qu'on ne voit pas en complet sur le capture d'écran.

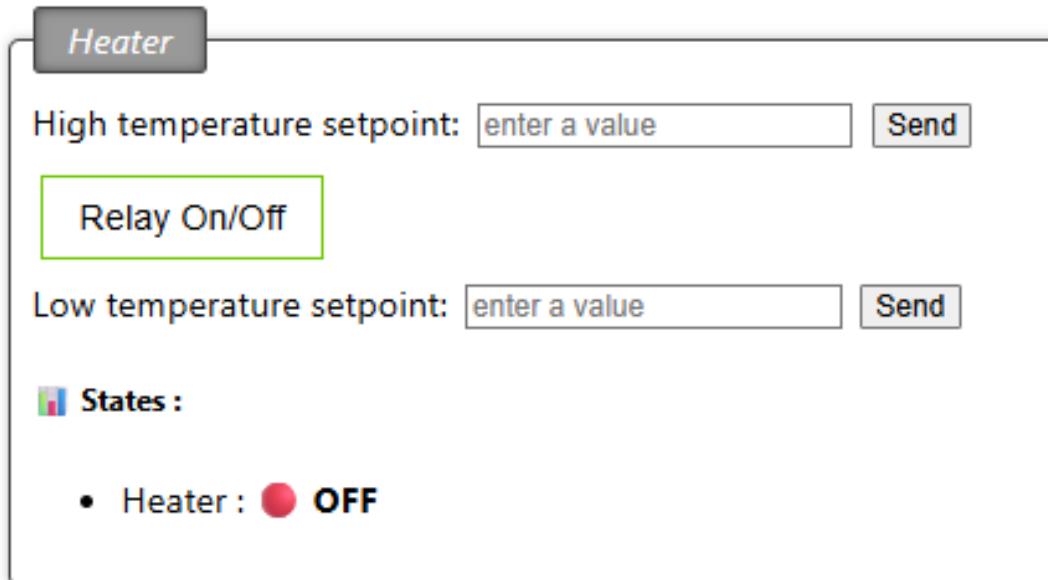


Fig. 141. – **Gestion du chauffage** : vue user

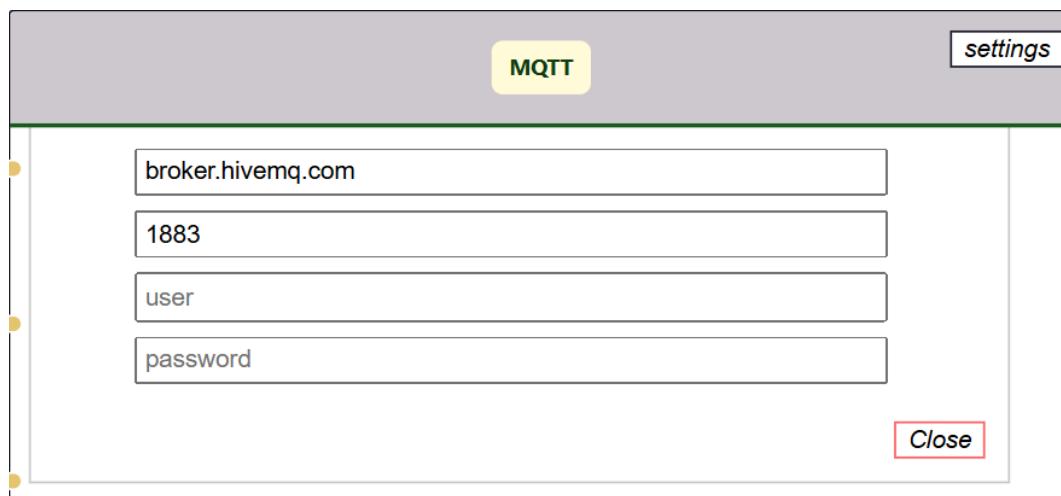


Fig. 142. – **Gestion du chauffage** : vue programmation - configuration du bloc *MQTT*

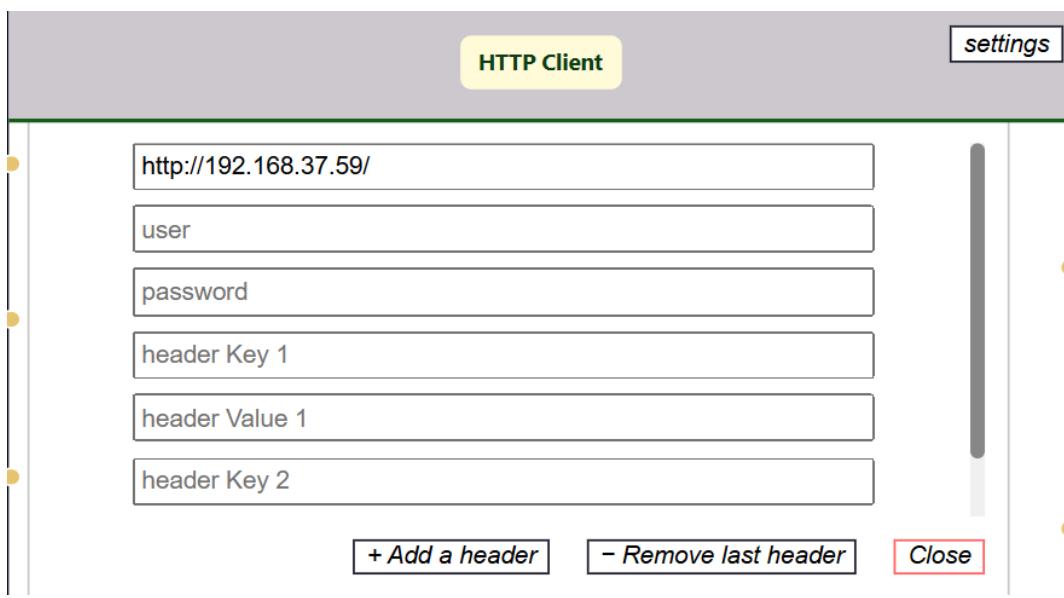


Fig. 143. – **Gestion du chauffage** : vue programmation - configuration du bloc *HTTP Client*

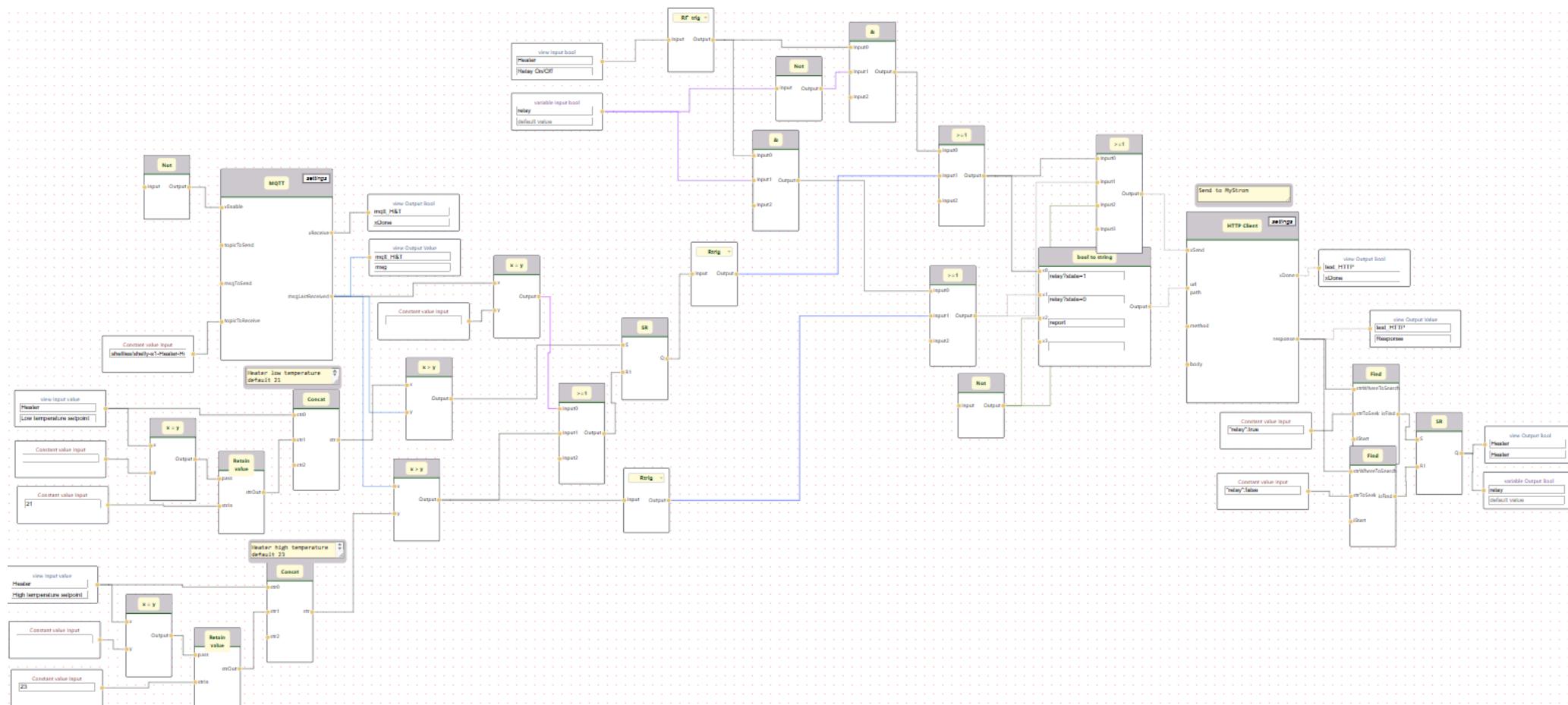


Fig. 144. – Gestion du chauffage : vue programmation - vue en complet

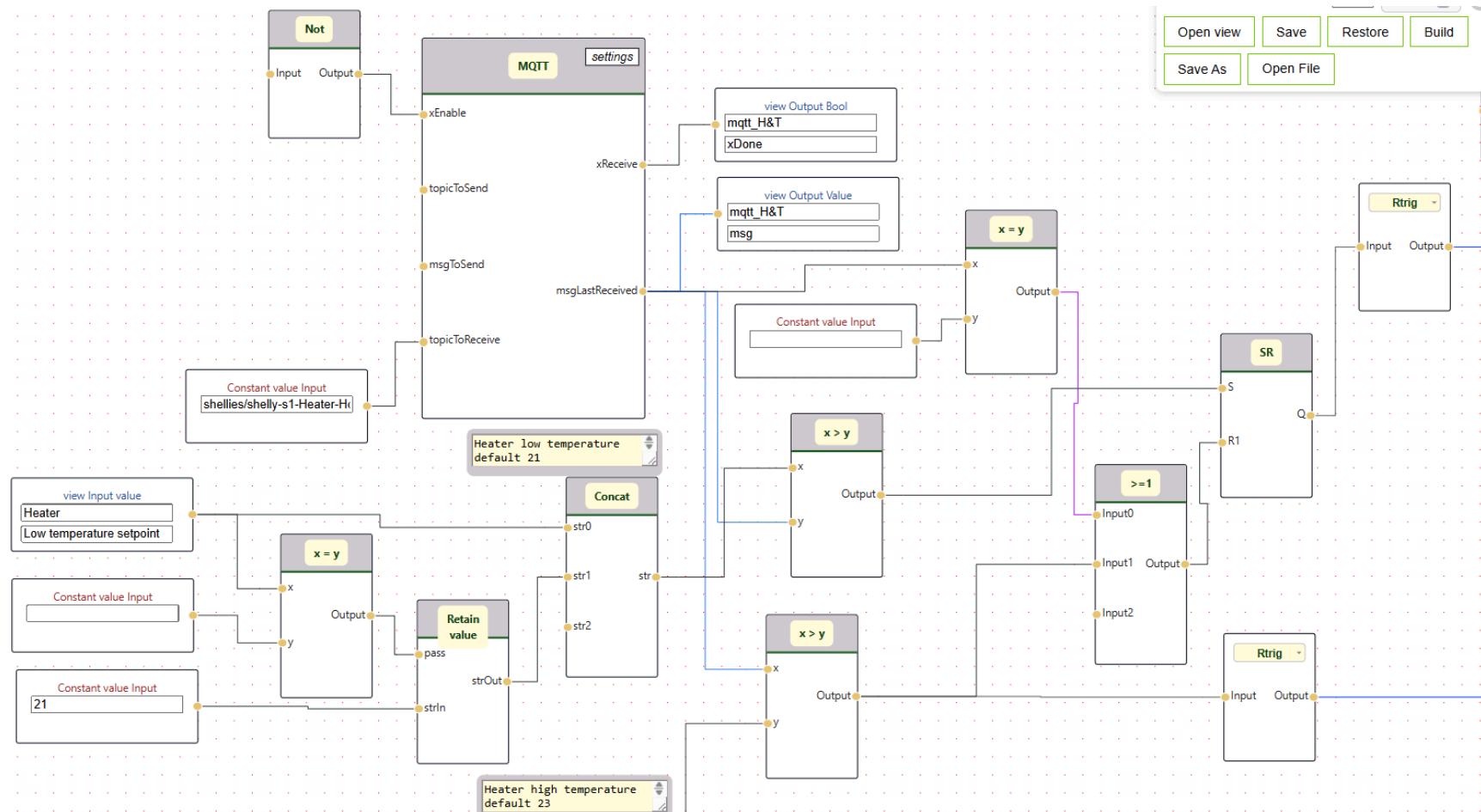


Fig. 145. – Gestion du chauffage : vue programmation - MQTT attente température

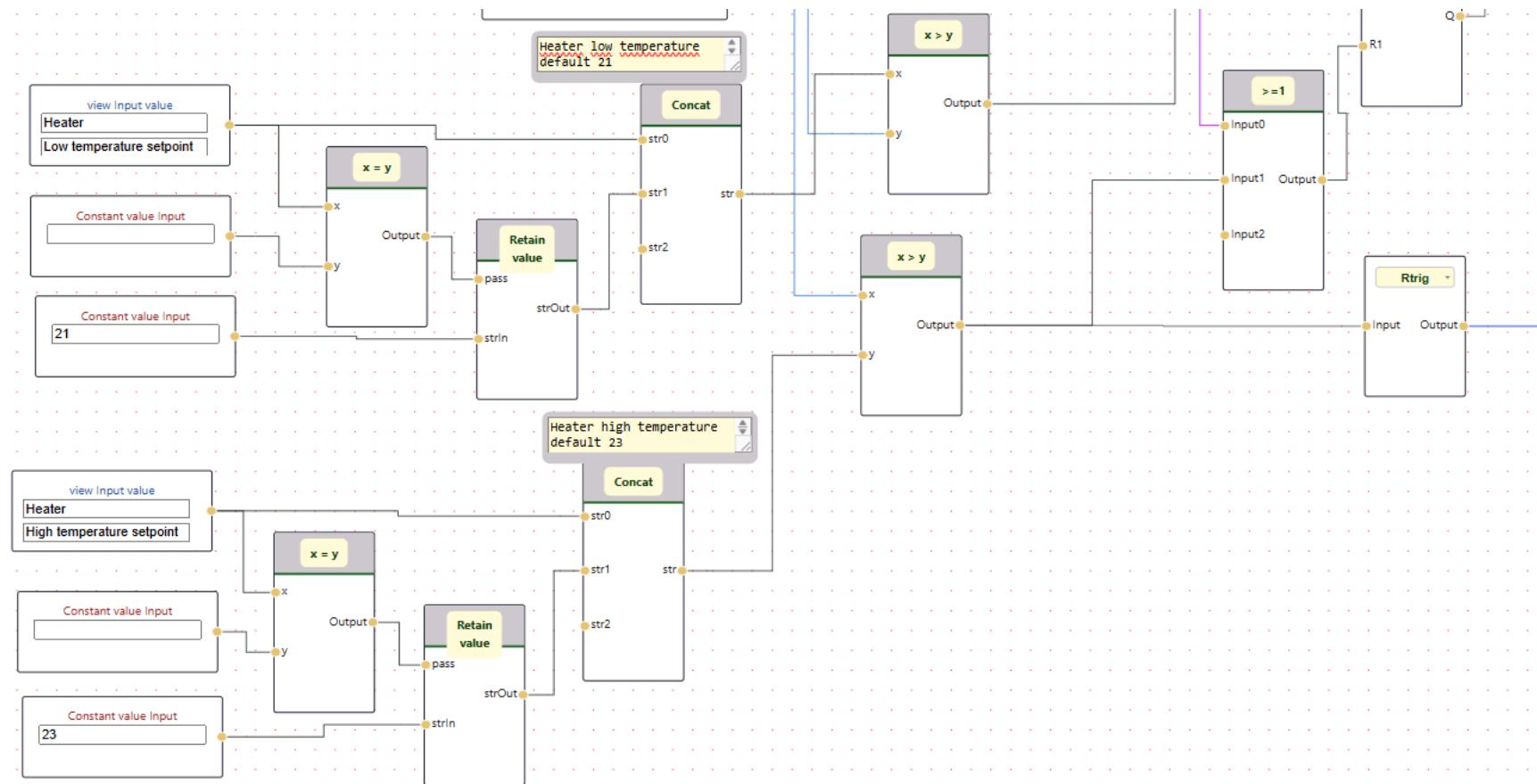
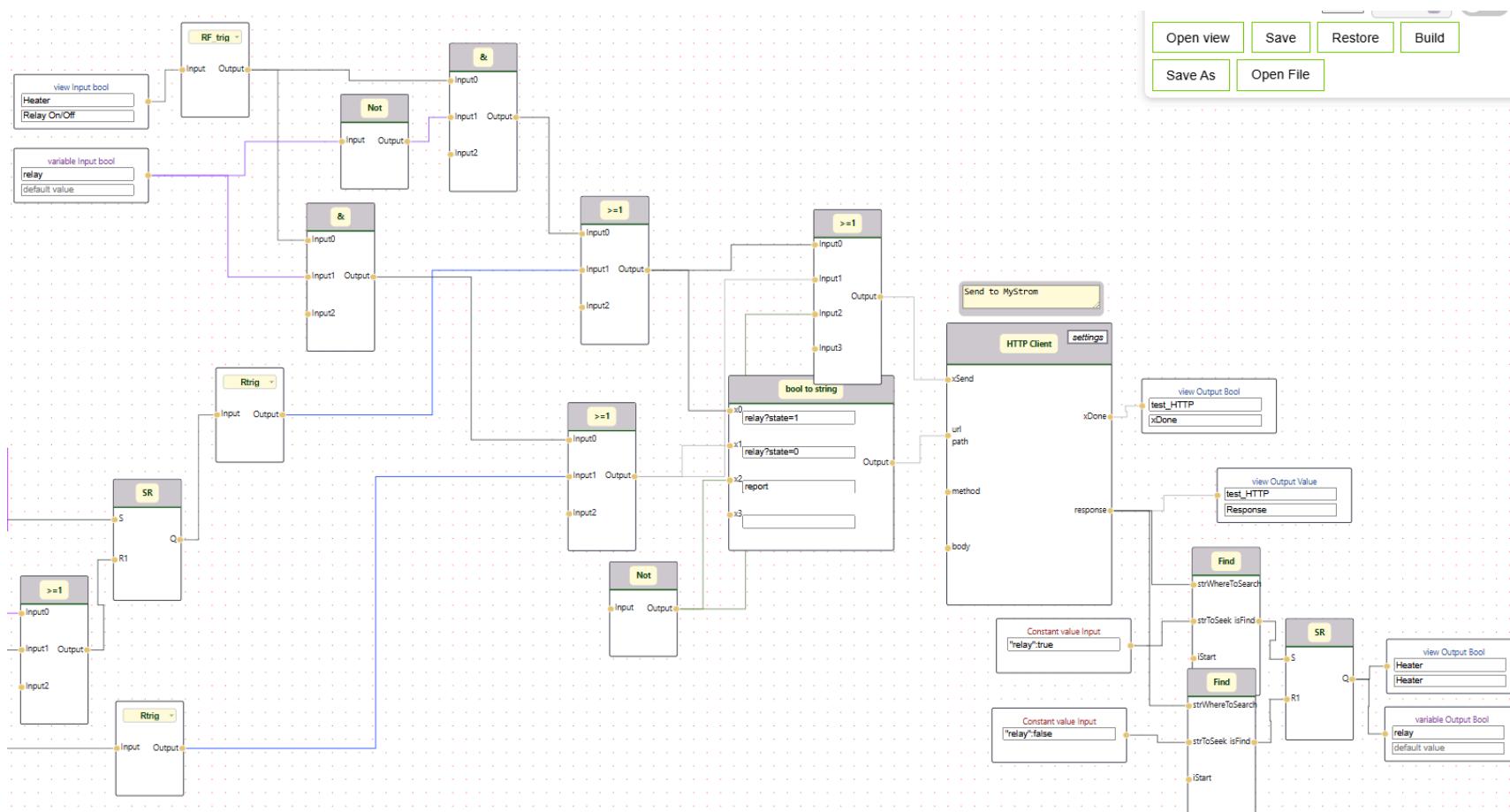


Fig. 146. – **Gestion du chauffage** : vue programmation - Gestion automatique selon température (seuil)

Fig. 147. – Gestion du chauffage : vue programmation - envoi au relais de *My Strom* (**on** ou **off**)

9.3 Gestion porte du garage

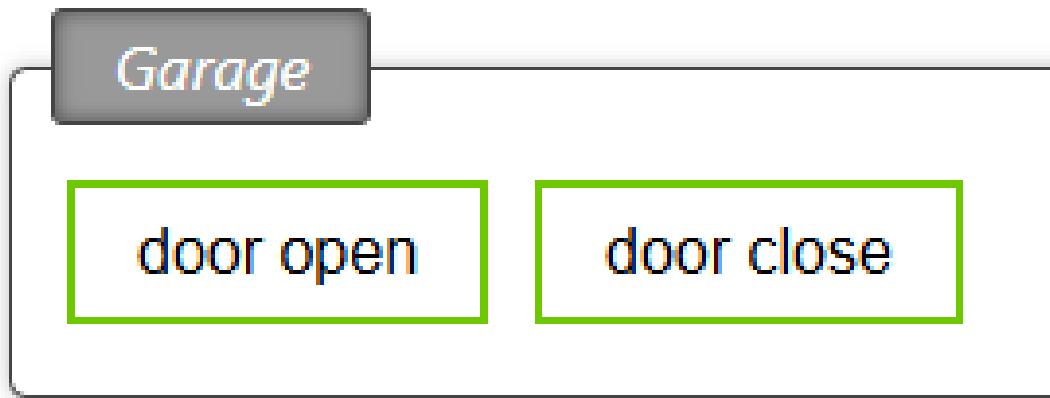


Fig. 148. – **porte du garage** : vue user

9.3.1 Modbus - porte du garage

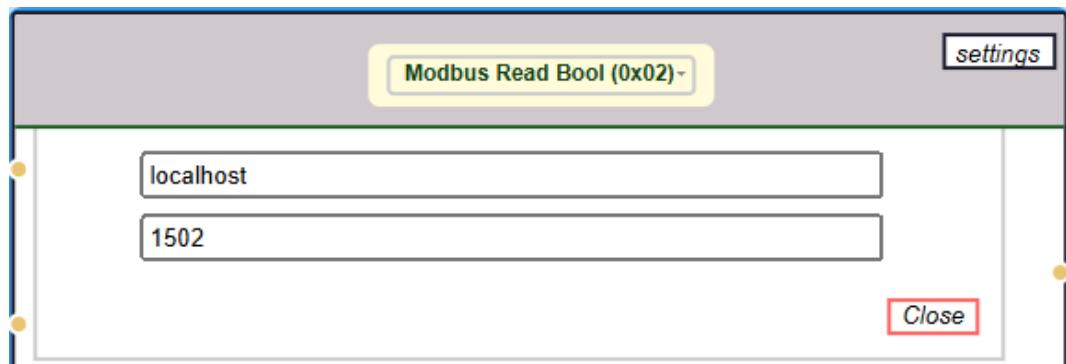


Fig. 149. – **porte du garage** : vue programmation - Modbus - configuration

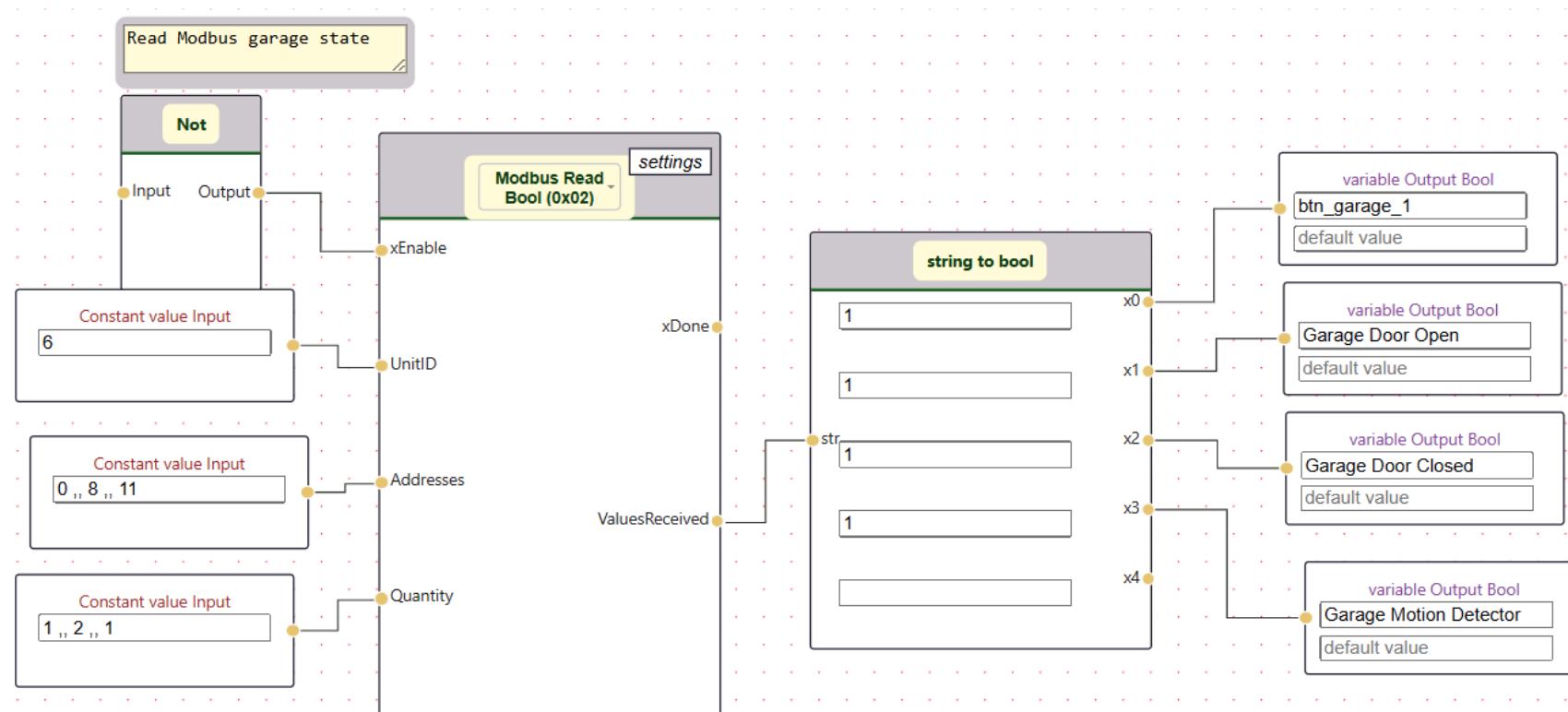


Fig. 150. – garage : vue programmation - Modbus - lecture

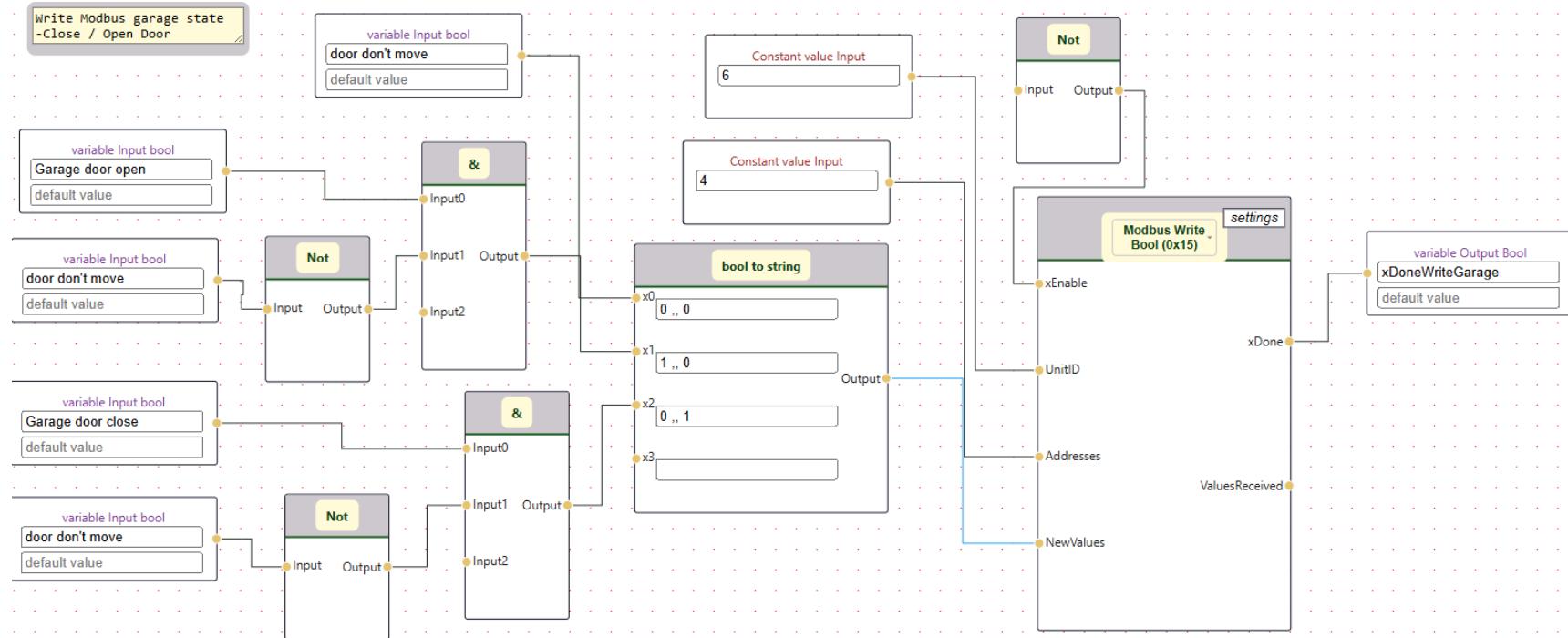


Fig. 151. – garage : vue programmation - Modbus - écriture

9.3.2 logique - porte du garage

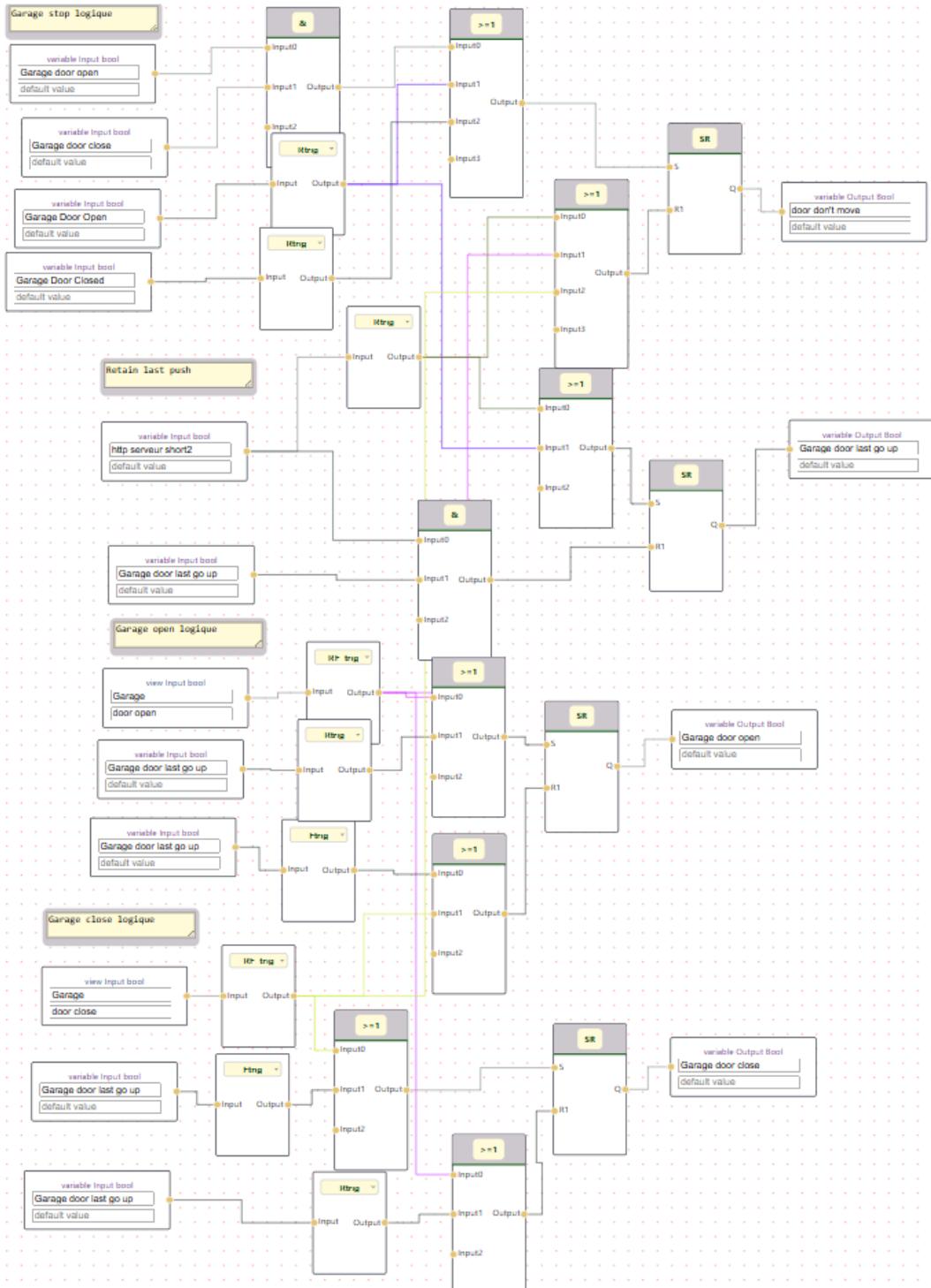


Fig. 152. – porte du garage : vue programmation - logique - vue en complet

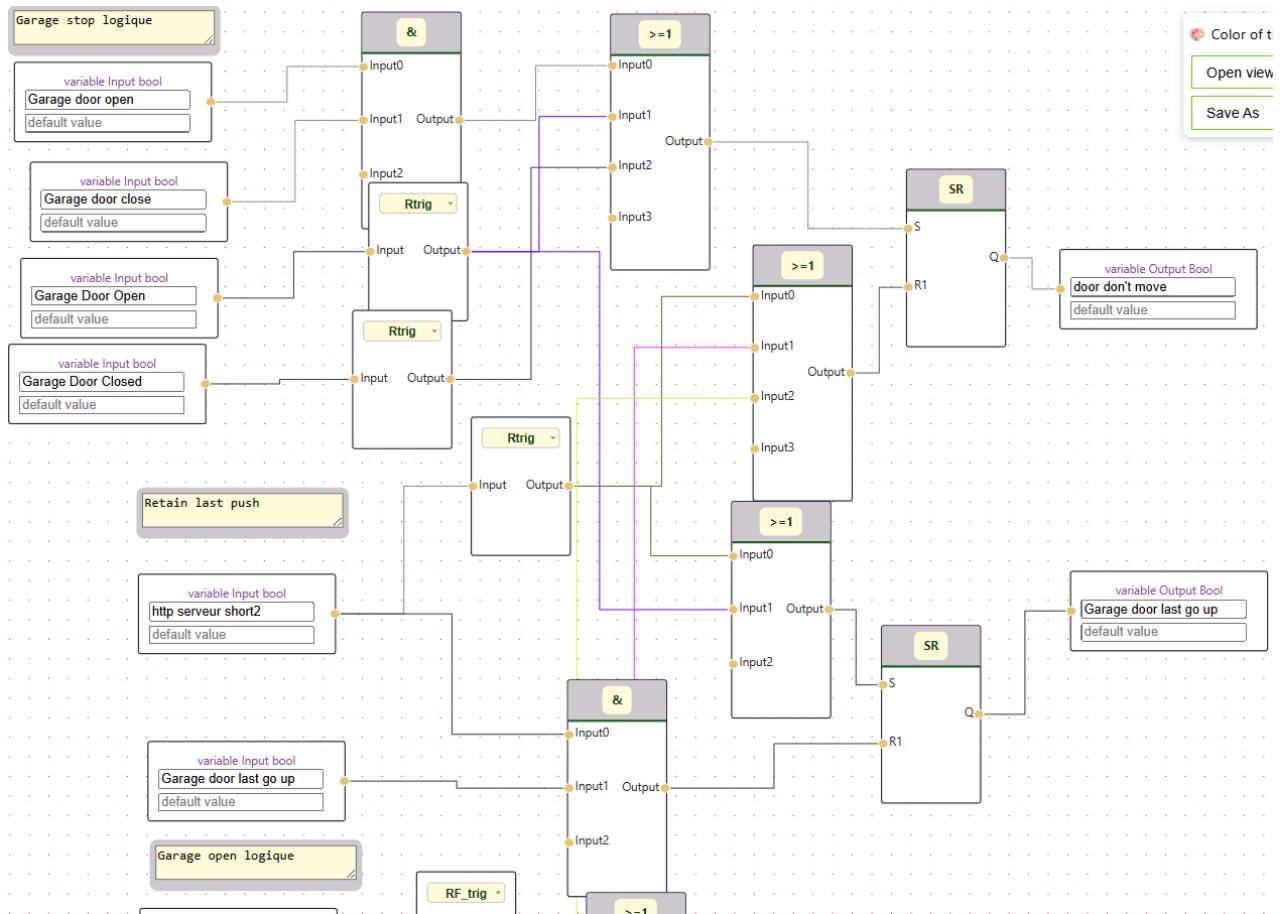


Fig. 153. – porte du garage : vue programmation - logique - écriture des variables « door don't move » et « Garage door last go up »

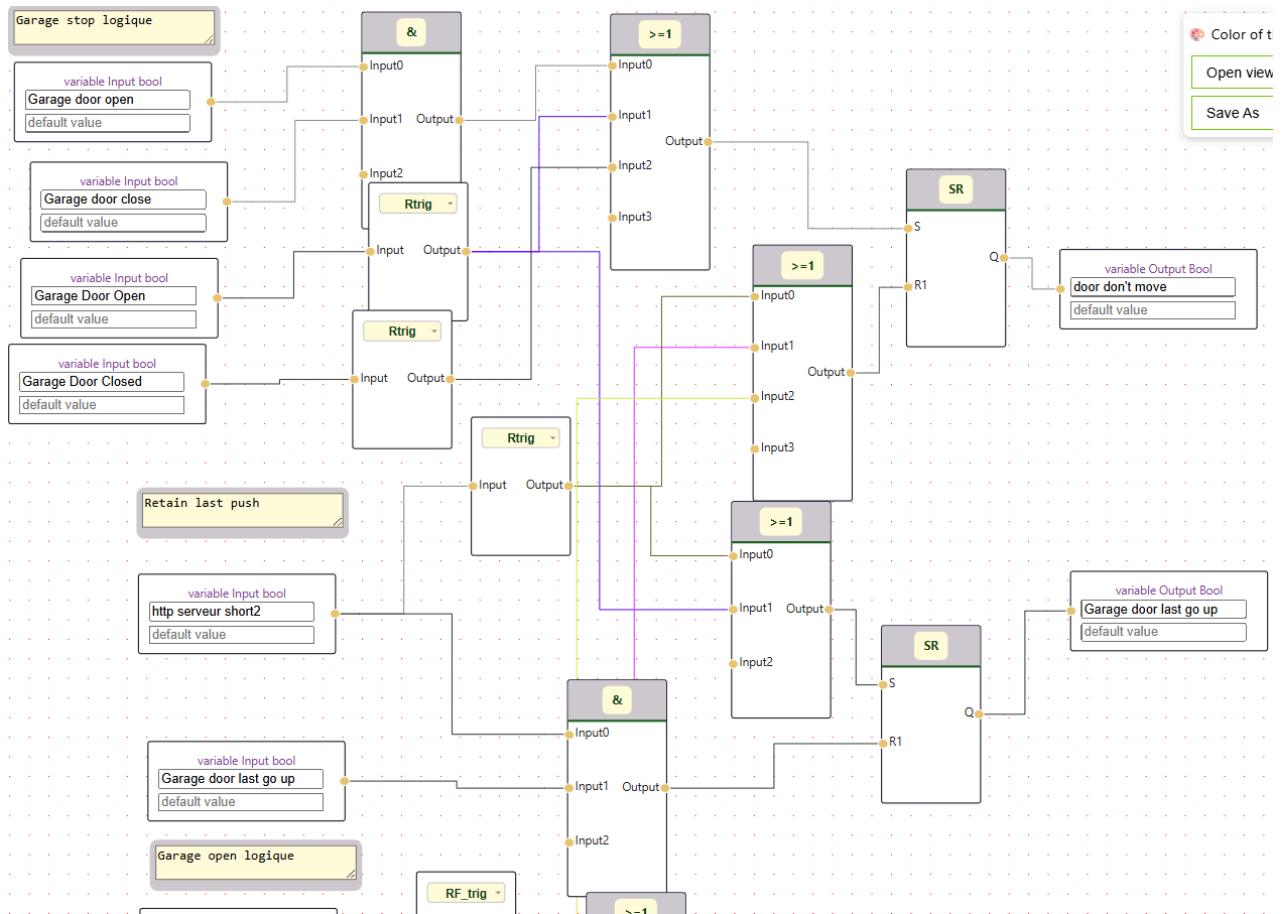


Fig. 154. – porte du garage : vue programmation - logique - écriture des variables
« Garage door open » et « Garage door close »

9.4 Gestion de la lampe de couleur

L'idée de ce programme est de permettre à l'utilisateur de changer la couleur et luminosité de la lampe. C'est aussi lui qui est responsable d'allumer la lampe en vert quand elle est ouverte au complet sinon en rouge.

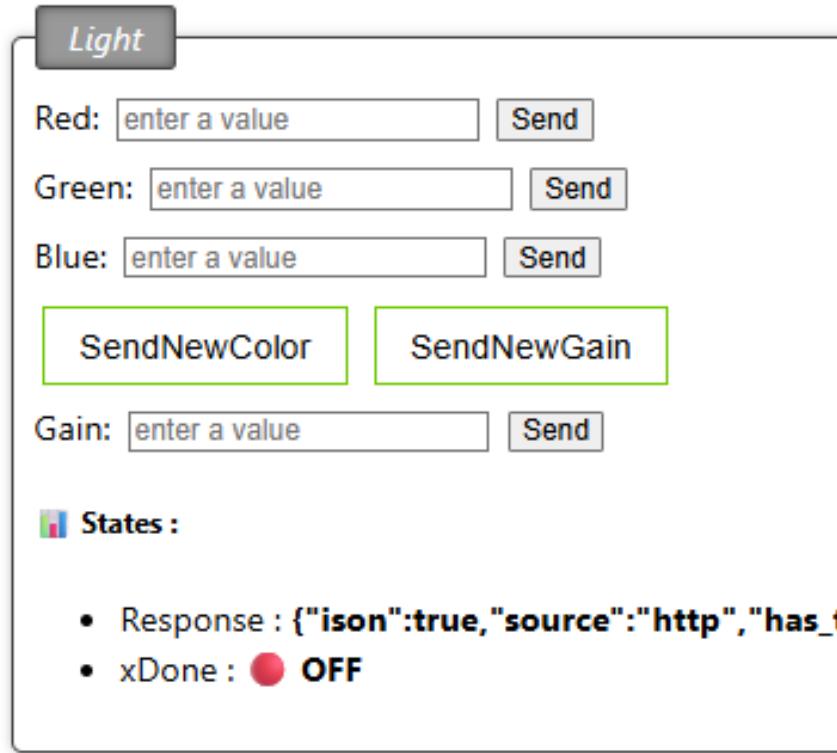


Fig. 155. – Gestion de la lampe de couleur *Shelly* : vue user

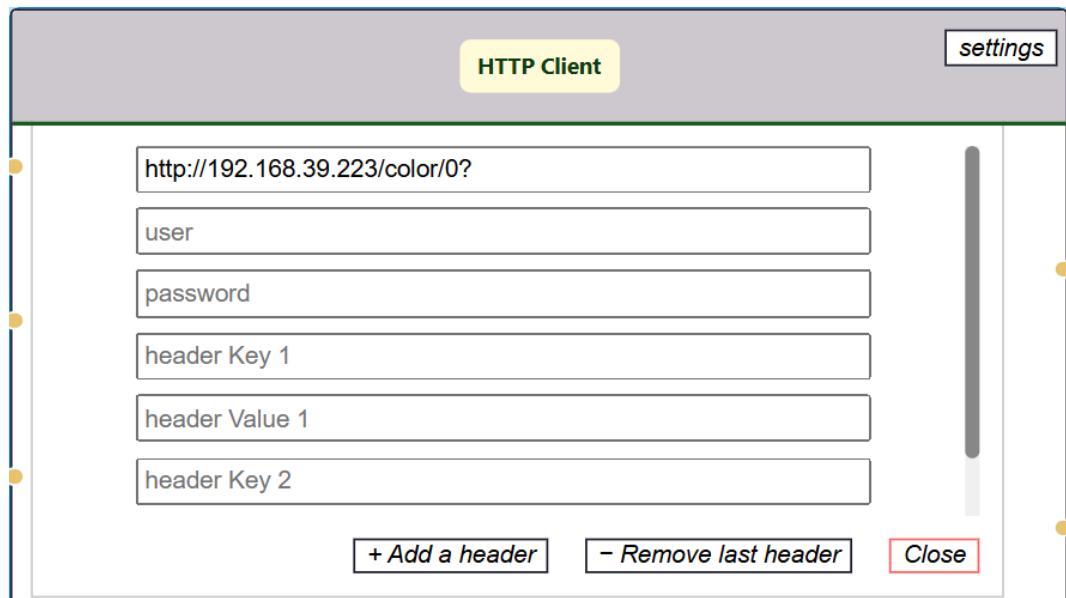


Fig. 156. – Gestion de la lampe de couleur *Shelly* : vue programmation - configuration du bloc *HTTP Client*

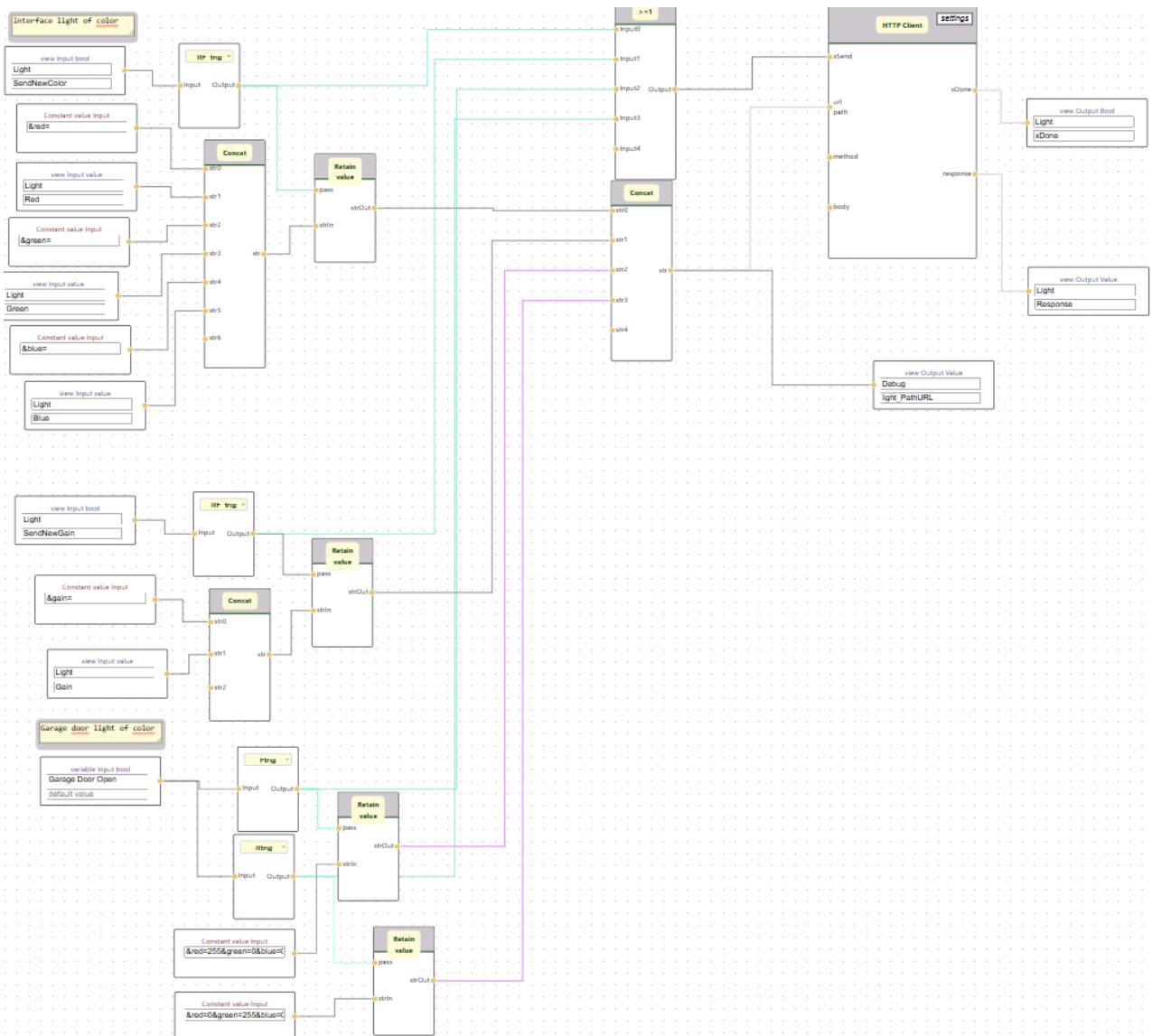


Fig. 157. – Gestion de la lampe de couleur Shelly : vue programmation - vue en complet

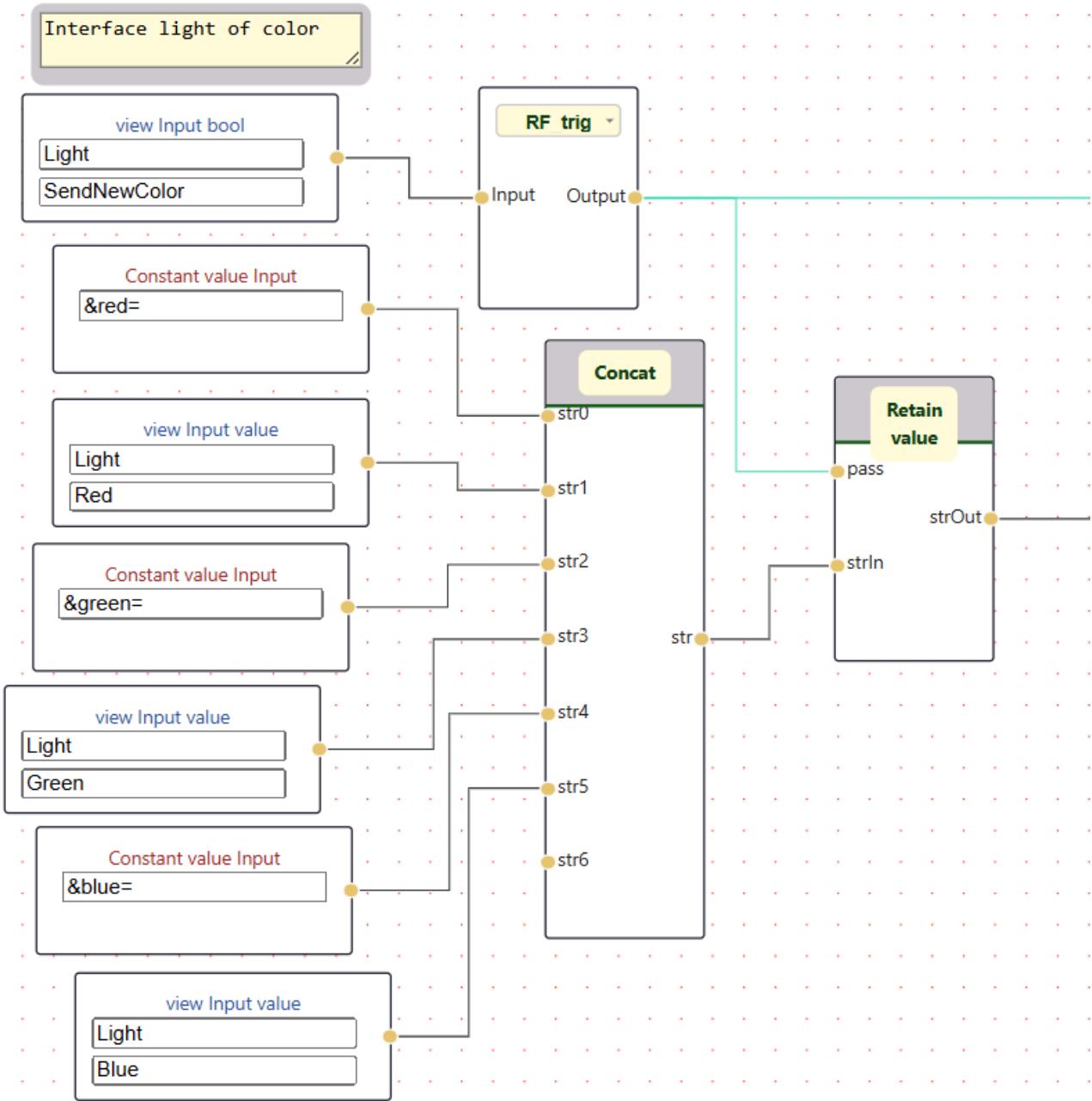


Fig. 158. – Gestion de la lampe de couleur **Shelly** : vue programmation - création url path pour changement des couleurs

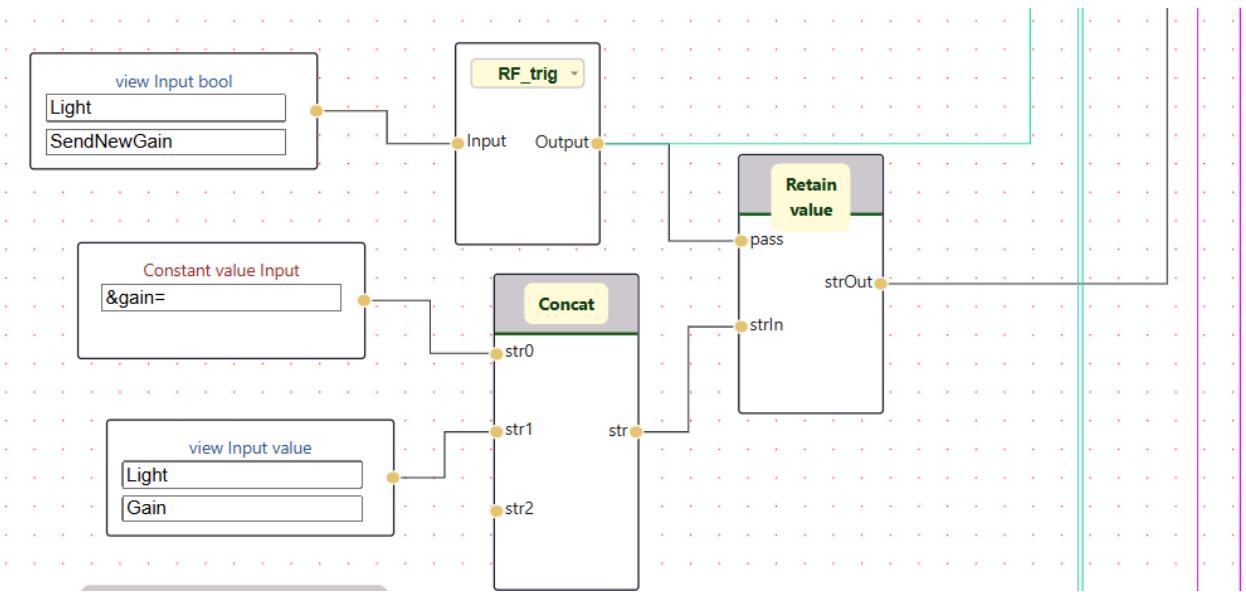


Fig. 159. – **Gestion de la lampe de couleur Shelly** : vue programmation - création url path pour changement de luminosité

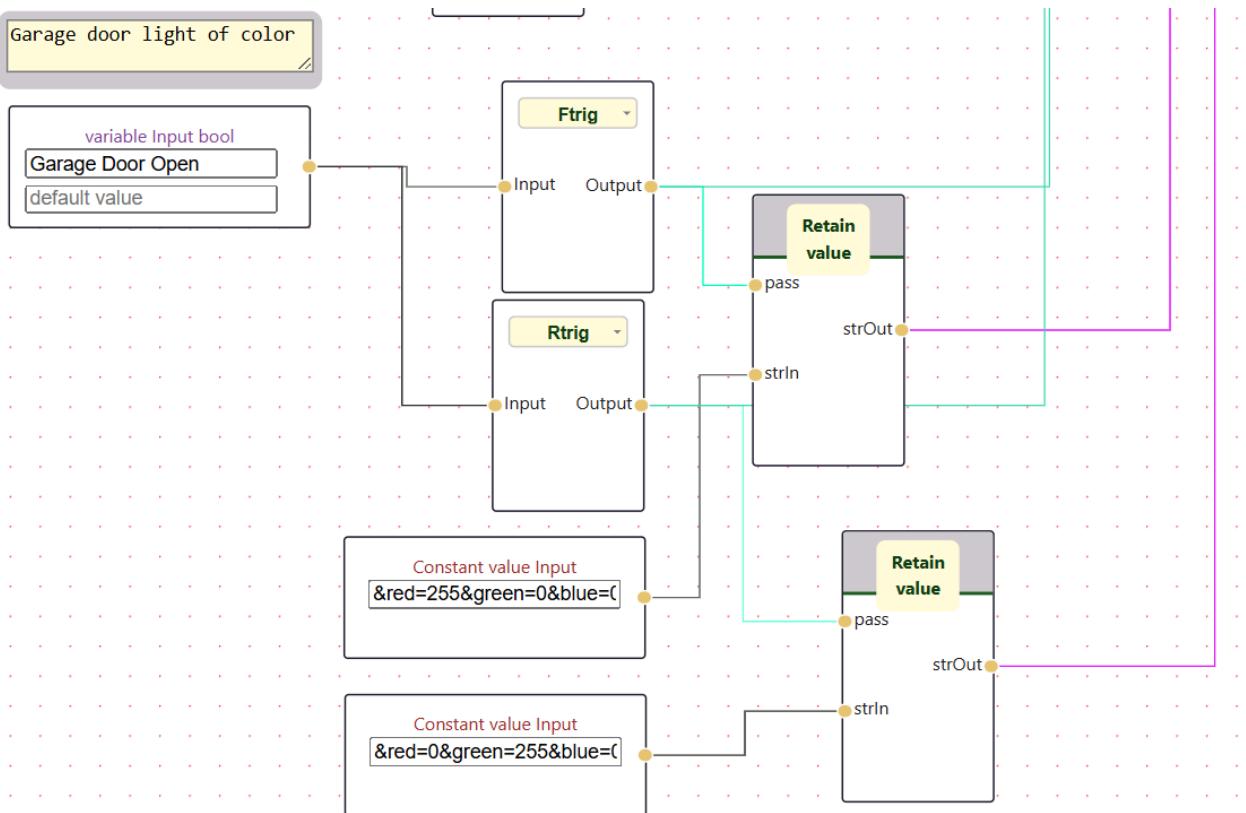


Fig. 160. – **Gestion de la lampe de couleur Shelly** : vue programmation - création url path pour changement des couleurs - **porte garage**

9.5 Gestion lumières

10 | Ensemble des blocs (Nodes)

11 | Autres annexes

11.1 My Strom : documentation

My Strom est un relais qui permet de contrôler le chauffage. Il est possible de le contrôler via une requête HTTP.

The screenshot shows the Postman mobile application interface displaying API documentation for the `api.mystrom.ch` service. The top navigation bar shows the URL `api.mystrom.ch`. Below the navigation, there are two main sections:

- GET Toggle**: This section shows the URL `http://[switch_ip]/toggle`. A note states: "Toggles the switch. Returns a JSON object with the following field: relay : meaning if the relay/switch has now been set to off (false) or on (true).".
- GET Turn off**: This section shows the URL `http://[switch_ip]/relay?state=0`. A note states: "Turns the switch off. Does not return anything."

Below these sections, there is a **PARAMS** section for the "Turn off" request, which includes a parameter `state` with a value of `0`. A note explains: "The value the relay/switch should be set to. 1 = turn on, 0 = turn off".

At the bottom of the screen, there are two "Example Request" and "Example Response" cards. The first card is for the "Toggle" request, showing the curl command: `curl --location -g 'http://[switch_ip]/toggle'`. The second card is for the "Turn off" request, showing the curl command: `curl --location -g 'http://[switch_ip]/relay?state=0'`.

Fig. 161. – My Strom : Documentation

Glossaire

Software

HAL – Hardware Abstraction Layer: A HAL is a layer of software that abstracts the hardware details of a computer system, allowing higher-level software to interact with the hardware without needing to know the specifics of the hardware implementation. [4](#), [5](#), [6](#), [9](#)

backend: Correspond au programme softplc-main. C'est dans celui-ci qu'on crée les blocs qui sont ensuite envoyés côté frontend. C'est lui qui est chargé d'exécuter les blocs après un build. [5](#), [9](#), [10](#), [25](#), [28](#), [29](#), [39](#), [49](#), [50](#), [51](#), [52](#), [57](#), [66](#)

frontend: Correspond au programme softplcui-main. C'est lui qui s'occupe de l'affichage dans la page web de tous les éléments et qui transmet ensuite les blocs au backend. [4](#), [5](#), [25](#), [49](#), [50](#), [51](#), [52](#), [57](#), [66](#)

Technology

IoT – Internet of Things: The Internet of Things (IoT) refers to the network of physical objects embedded with sensors, software, and other technologies to connect and exchange data with other devices and systems over the internet. [5](#)

termes généraux

WDA – WAGO Device Access (accès aux paramètres et IO): WDA est une interface REST qui permet d'accéder aux paramètres et aux entrées/sorties des automates WAGO. Elle facilite la communication entre le logiciel de contrôle et l'automate en utilisant des requêtes HTTP pour récupérer et modifier les données. [28](#)

stretchable: Stretchable fait référence à la capacité d'un bloc d'étendre son nombre d'entrées ou sorties en fonction des besoins. Cela permet une flexibilité dans la conception. C'est un paramètre pouvant être choisi lors de la création d'un bloc. [30](#)

Bibliographie

- [1] « N8n-Io/N8n ». Consulté le: 27 avril 2025. [En ligne]. Disponible sur: <https://github.com/n8n-io/n8n>
- [2] « JavaScript Libraries and Components for Web Development ». Consulté le: 27 avril 2025. [En ligne]. Disponible sur: <https://www.totaljs.com/>
- [3] « ChatGPT ». Consulté le: 28 février 2025. [En ligne]. Disponible sur: <https://chatgpt.com/>
- [4] « THE 17 GOALS | Sustainable Development ». Consulté le: 27 avril 2025. [En ligne]. Disponible sur: <https://sdgs.un.org/goals>
- [5] N. Sharma, M. Shamkuwar, et I. Singh, « The History, Present and Future with IoT », in *Internet of Things and Big Data Analytics for Smart Generation*, V. E. Balas, V. K. Solanki, R. Kumar, et M. Khari, Éd., Cham: Springer International Publishing, 2019, p. 27-51. doi: [10.1007/978-3-030-04203-5_3](https://doi.org/10.1007/978-3-030-04203-5_3).
- [6] R. Chataut, A. Phoummalayvane, et R. Akl, « Unleashing the Power of IoT: A Comprehensive Review of IoT Applications and Future Prospects in Healthcare, Agriculture, Smart Homes, Smart Cities, and Industry 4.0 », *Sensors*, vol. 23, n° 16, 2023, doi: [10.3390/s23167194](https://doi.org/10.3390/s23167194).
- [7] « Using TypeScript – React ». Consulté le: 1 juillet 2025. [En ligne]. Disponible sur: <https://react.dev/learn/typescript>
- [8] « Quickstart ». Consulté le: 1 juillet 2025. [En ligne]. Disponible sur: <https://reactflow.dev/learn>
- [9] « Http Package - Net/Http - Go Packages ». Consulté le: 18 juin 2025. [En ligne]. Disponible sur: <https://pkg.go.dev/net/http#ListenAndServe>
- [10] V. SOYSOUVANH, « Clients et Serveurs HTTP en Go : Guide Complet ». Consulté le: 17 juin 2025. [En ligne]. Disponible sur: <https://certiquizz.com/fr/cours/programmation/go/go-beyond-maitrisez-go-pour-l-innovation-cloud-ia-et-devops/http-client-et-serveur>
- [11] C. Nicholson, « Craignicholson/Simplehttp ». Consulté le: 18 juin 2025. [En ligne]. Disponible sur: <https://github.com/craignicholson/simplehttp>
- [12] « Les Codes Fonctions Modbus ». Consulté le: 21 juillet 2025. [En ligne]. Disponible sur: <https://yutec.fr/automatisme/protocole-de-communication/fonctions-modbus/>
- [13] « Goburrow/Modbus ». Consulté le: 21 juillet 2025. [En ligne]. Disponible sur: <https://github.com/goburrow/modbus>
- [14] « React Router / useNavigate – DevDocs ». Consulté le: 23 juillet 2025. [En ligne]. Disponible sur: https://devdocs.io/react_router/hooks/use-navigate
- [15] « Custom Nodes ». Consulté le: 30 juin 2025. [En ligne]. Disponible sur: <https://reactflow.dev/examples/nodes/custom-node>

- [16] « CSS Buttons ». Consulté le: 1 juillet 2025. [En ligne]. Disponible sur: https://www.w3schools.com/css/css3_buttons.asp
- [17] « SHELLY BUTTON1 USER MANUAL Pdf Download | ManualsLib ». Consulté le: 22 juillet 2025. [En ligne]. Disponible sur: <https://www.manualslib.com/manual/2017524/Shelly-Button1.html>
- [18] « Documentation about Shelly Bulb Duo RGBW ». Consulté le: 22 juillet 2025. [En ligne]. Disponible sur: <https://www.shelly.com/blogs/documentation/shelly-bulb-duo-rgbw>
- [19] « Webhooks / HTTP(S) Requests – The Unofficial Shelly Guide! ». Consulté le: 22 juillet 2025. [En ligne]. Disponible sur: <https://shelly.guide/webhooks-https-requests/>
- [20] « Mode d'emploi Shelly H&T (2 Des Pages) ». Consulté le: 22 juillet 2025. [En ligne]. Disponible sur: <https://www.modesdemploi.fr/shelly/ht/mode-d-emploi>
- [21] « MQTT | Shelly Technical Documentation ». Consulté le: 22 juillet 2025. [En ligne]. Disponible sur: <https://shelly-api-docs.shelly.cloud/gen2/ComponentsAndServices/Mqtt/>
- [22] « Accordion | React Bootstrap ». Consulté le: 27 avril 2025. [En ligne]. Disponible sur: <https://react-bootstrap.netlify.app/docs/components/accordion/>
- [23] « WAGO Download Center - WAGO Device Model - 751-9401 ». Consulté le: 27 avril 2025. [En ligne]. Disponible sur: <https://downloadcenter.wago.com/wago/null/details/m2ddbfyuihrn44rp2h>
- [24] « WAGO Download Center - WAGO Device Model - 751-9402 ». Consulté le: 27 avril 2025. [En ligne]. Disponible sur: <https://downloadcenter.wago.com/wago/null/details/m2dd83m229zdbc5cau>