BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2026
http://excel.fit.vutbr.cz

# Visual Tool for Processing Incoming Messages from IoT Devices

Danil Tasmassys*

**Abstract**
It is hard for users to program IoT on a mobile device due to the small screen, syntactic errors, and a limited keyboard. This work presents a visual programming tool written on React/Zustand with "smart" validation. User creates logic using types, and on output has JavaScript code. It allows users to create programs on the go and lowers the entry threshold to work with IoT.

*xtasmad00@stud.fit.vut.cz, *Faculty of Information Technology, Brno University of Technology*

## 1. Introduction

IoT can be used everywhere, but the code for it is typically written using desktop applications. There are no IDEs specifically designed for mobile devices, and using existing tools, such as Scratch or Node-RED, is not completely convenient on mobile devices. However, there is a need for Field engineering, where users have only a mobile device and need to change code [1].

The core of this problem is the heterogeneity of IoT networks, where devices use different payload formats, requiring conversion [2]. To solve this, non-professional users should be enabled to create and modify conversion functions and their rules themselves.

Nesting in such applications uses indentation to show nesting and connections between blocks. However, mobile devices have limited horizontal space, so horizontal scrolling isn't ideal. Also, input precision on a mobile device is another problem, as it is easy to make mistakes on a small screen. Therefore, there is a need for protection against users' mistakes at the logic level.

Furthermore, traditional text-based programming is more error-prone and has a high cognitive load, as users must memorize strict rules and mentally trace dependencies [3].

### 1.1 Limitations of Existing Tools

Existing solutions based on the Blockly library or Node-RED offer many tools and a strong community of ready-made functions and plugins; however, they often lean towards an "endless" canvas, which is not convenient to use on a mobile device. To navigate these canvases, mobile users will need to frequently zoom and scroll in all directions [1].

### 1.2 Proposed Solution

The solution presented in this work is based on the following Mobile-First guideline [4] and aims to use flat nesting with visual separation of branches as an alternative to indentation nesting, which isn't ideal for mobile devices. To ensure a smooth user experience on mobile devices, the application uses a normalized state management, enabling atomic updates to a logic tree without triggering full-page re-renders.

## 2. UI Architecture

As shown in Figure 1, the interface does not resemble the desktop-oriented "infinite" canvas and was designed following the Mobile-First principle [4]. The interface favors a vertical and scroll-optimized layout. A linearized representation of nested structures was implemented. Instead of using horizontal indentation, a flat nesting and visual separation of the branches was implemented by dividing them into slides, as shown in Figure 4. It allows blocks to have the same full-screen width, which is critical for visibility on mobile devices.

The user doesn't see the entire program tree, eliminating the need to zoom. Instead, navigation between branches is implemented using pagination and swiping. This allows users to focus on the specific block.

### 2.1 Thumb-Driven Ergonomics

The screen was divided into three functional zones based on the thumb's reach:

- **Bottom zone**: For frequently used elements of navigation.
- **Center zone**: Area of the visual focus and the main content (program).
- **Header**: For rarely used elements or to protect from the missclicks.

This division follows the "one eyeball and one thumb" interaction model, ensuring that users can use the application single-handedly while distracted in field conditions.

### 2.2 Digital Twin Synchronization

Another important part of the application is that both the decoder and the encoder can be implemented in the same program. It is needed for target platforms that require both functions for normal operation, as well as for the so-called "Digital Twins", which require two-way communication [5].

## 3. Viewports

One of the main features of the presented tool is that condition blocks, specifically "if else" and "else" aren't added below the original "if" block, but are moved to the branches on the right side of the original "if" block. It presents as an alternative to nesting, reduces the vertical length of the program, and lowers the cognitive load on the user by allowing them to mentally separate these branches.

Additionally, if a new "if" block was created below the first "if" block and a new branch was added to the new block, previous branches are moved to the right, and the new branch is inserted between them. It creates a matrix structure, where all execution paths that are connected form their own visible column.

## 4. Structural integrity engine

Touchscreens have lower precision than mouse-driven desktop applications, leading to higher error-proneness in program creation. To minimize these errors, the structural integrity engine was implemented.

It is starting atomic on each change in the program to show users their mistakes. Figure 3 shows how the application reacts to a user's incorrect actions. If a block is not in the correct context, the engine visually marks it. It prevents the creation of logically incomplete or syntactically wrong constructions before code generation.

## 5. Output

As shown in the creation pipeline in Figure 5, the final part of the application work is code generation from the visual tree to JavaScript, as shown in Figure 2. The generation engine traverses an abstract syntax tree and generates clean code for IoT systems.

### 5.1 Source Code Emission

Initially, the program stores all created logic in a flat array. It's needed for faster application performance, since React is optimized to work with arrays and only transforms branches into an abstract syntax tree when the user wants to generate code. Then, the code generator checks its content and, based on its payload, translates code into the corresponding JavaScript syntactic construct.

## 6. Conclusions

As part of the work, a host application for the UI library was created and adapted for mobile devices. The main achievement is the rejection of a desktop paradigm in favor of viewports and branching. The structural integrity engine helps mitigate the disadvantages of touchscreens, ensuring program integrity.

## Acknowledgements

I would like to thank my supervisor, Ing. Petr John, for his guidance and support during the preparation of this work.

## References

[1] P. P. Ray. A survey on visual programming languages in internet of things. *Scientific Programming*, 2017, 2017.

[2] Andrea Zanella, Nicola Bui, Angelo Castellani, Lorenzo Vangelista, and Michele Zorzi. Internet of things for smart cities. *IEEE Internet of Things journal*, 1(1):22–32, 2014.

[3] Thomas RG Green and Marian Petre. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages & Computing*, 7(2):131–174, 1996.

[4] Luke Wroblewski. *Mobile First*. A Book Apart, 2011.

[5] Werner Kritzinger, Matthias Karner, Georg Traar, Jan Henjes, and Wilfried Sihn. Digital twin in manufacturing: A categorical literature review and classification. *IFAC-PapersOnLine*, 51(11):1016–1022, 2018.