Document Object Model for IoT

Building IoT Systems with HTML and JavaScript

Alvaro Fernandez
a@domiot.org
domiot.org

7 June 2025

Abstract

By using HTML and JavaScript, developers can create interactive IoT systems in the same way they create web pages.
IoT systems are often programmed by linking low-level input and output components, rather than defining interactions in terms of meaningful, higher-level abstractions. Systems' behavior is frequently implemented in a non-standardized, rigid manner. Even data structures, intended for organizing data, are sometimes misapplied to "script" device behavior. This makes systems difficult to understand, maintain, and adapt, where even small changes require specialized reprogramming.
While existing efforts, such as the W3C Web of Things, provide standardized APIs and semantic models to address these challenges, we argue that widely adopted web technologies, specifically HTML and the DOM API, can further simplify IoT development and make it more accessible.
We propose extending the Document Object Model (DOM) for IoT (DOMIoT) and treating interactive IoT systems like interactive documents. Developers can describe these systems using meaningful HTML elements and attributes, and implement behavior with standard JavaScript. Scripts can listen and react to events such as `press`, `release` or `change`, modifying the physical environment using the same methods they use to update content on a web page.

1. Demonstration

Consider the following simple yet illustrative scenario:

To assist customers, a supermarket has installed large buttons with images at the entrance of each aisle. When a customer presses a button, the corresponding shelving unit lights up in blue.

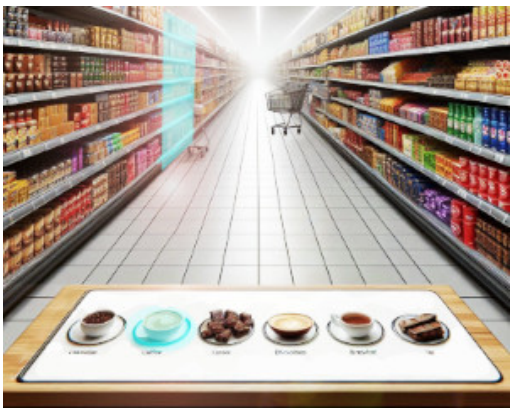

Figure 1: Image of a shelving unit lighting up in blue after a button is pressed. (image generated with dall.e.)

To simplify the creation of this experience, we propose using the same approach as in web development with HTML and JavaScript:

```
<html>
  <iot-aisle id="aisle6" name="Coffee, Hot Beverages, Cookies & Chocolate">

    <iot-button-binding id="a6ButtonBinding" location="/dev/buttons6">
    <iot-relay-binding id="a6RelayBinding" location="/dev/relay12">

    <iot-button id="a6Product1Button" shelving-unit-id="a6L1" binding="a6ButtonBinding">
    <iot-button id="a6Product2Button" shelving-unit-id="a6L2" binding="a6ButtonBinding">
    <iot-button id="a6Product3Button" shelving-unit-id="a6L3" binding="a6ButtonBinding">
    <iot-button id="a6Product4Button" shelving-unit-id="a6R1" binding="a6ButtonBinding">
    <iot-button id="a6Product5Button" shelving-unit-id="a6R2" binding="a6ButtonBinding">
    <iot-button id="a6Product6Button" shelving-unit-id="a6R3" binding="a6ButtonBinding">

    <iot-shelving-unit id="a6L1" name="Ground Coffee"
                                 style="background-color:white;" binding="a6RelayBinding">
    <iot-shelving-unit id="a6L2" name="Coffee Pods & K-Cups"
                                 style="background-color:white;" binding="a6RelayBinding">
    <iot-shelving-unit id="a6L3" name="Cookies and Biscuits"
                                 style="background-color:white;" binding="a6RelayBinding">

    <iot-shelving-unit id="a6R1" name="Premium Chocolate & Candy"
                                 style="background-color:white;" binding="a6RelayBinding">
    <iot-shelving-unit id="a6R2" name="Tea Selection"
                                 style="background-color:white;" binding="a6RelayBinding">
    <iot-shelving-unit id="a6R3" name="Snack Cakes, Muffins, Mini Pastries"
                                 style="background-color:white;" binding="a6RelayBinding">

  </iot-aisle>
  <script src="domiot.min.js"></script>
```

```
  <script>
    // retrieve all the buttons of the aisle number 6 in order
    const buttons = document.querySelectorAll('#aisle6 iot-button');

    // Light up each shelving unit in blue when its button is pressed,
    // and switch it back to white when the button is released.
    for (let i = 0; i < buttons.length; i++) {
      const button = buttons[i];

      // when a button is pressed, light up the corresponding shelving unit in blue.
      button.addEventListener('press', (ev) => {
        const shelvingUnitId = ev.target.getAttribute('shelving-unit-id');
        if (!shelvingUnitId) return;

        const shelvingUnit = document.getElementById(shelvingUnitId);
        if (!shelvingUnit) return;

        // change the background color of the shelving unit to blue,
        // this changes the light color.
        shelvingUnit.style.setProperty('background-color','blue');

      });

      // when a button is release light up its corresponding shelving unit
      // in white (normal ligths).
      button.addEventListener('release', (ev) => {
        const shelvingUnitId = ev.target.getAttribute('shelving-unit-id');
        if (!shelvingUnitId) return;

        const shelvingUnit = document.getElementById(shelvingUnitId);
        if (!shelvingUnit) return;

        // change the background color of the shelving unit to white,
        // this changes the light color.
        shelvingUnit.style.setProperty('background-color','white');

      });
    }
  </script>
</html>
```

In this example, we use HTML to describe the interactive IoT system: aisle number 6, the buttons at the entrance, and the shelving units on the left and right sides of the aisle. After describing the components of our system, we use JavaScript and the standard DOM API methods to script the system's behavior: a shelving unit lights up in blue when a person presses the corresponding physical button (triggering a `press` event on the button element).

In addition to lighting up a shelving unit in blue when a button is pressed, the supermarket wants to project a commercial on the floor to promote one product per shelving unit:

```
  ...
  <iot-aisle id="aisle6" name="Coffe ...
    ...
    <iot-video-binding id="a6VideoBinding" location="/dev/video6">
    ...
    <iot-button id="a6Product2Button" shelving-unit-id="a6L2" binding="a6ButtonBinding">
    ...
    <iot-shelving-unit id="a6L2" name="Coffee Pods & K-Cups"
                                 video-src="/path/to/commercial.mp4"
                                 style="background-color:white;" binding="a6RelayBinding">

    <iot-video id="a6Video" binding="a6VideoBinding"></iot-video>
    ...
  </iot-aisle>
  ...
  <script>
    // retrieve the aisle 6 video player
    let a6Video = document.getElementById("a6Video");

    const buttons = document.querySelectorAll('#aisle6 iot-button');

    for (let i = 0; i < buttons.length; i++) {
      const button = buttons[i];

      // when a button is pressed, light up the corresponding
      // shelving unit in blue and project a commercial.
      button.addEventListener('press', (ev) => {
        ...
        // light the shelving unit in blue.
        shelvingUnit.style.setProperty('background-color','blue');

        // retrieve the commercial to play and play it.
        const videoSrc = shelvingUnit.getAttribute('video-src');
        if (videoSrc) {
          a6Video.src = videoSrc;
          a6Video.load();
          a6Video.play();
        }
      });
      ...
    }
  </script>
</html>
```

In this example, we continue using JavaScript to light up a shelving unit in blue, project a promotional video on the floor when a person presses a button on the table.

Figure 2: Image showing how The "Coffee Pods & K-Cups" shelving unit lights up in blue when its button is pressed, at the same time a commercial is projected on the floor. (image generated with dall.e.)

If products are moved to different shelving units or a brand bids higher to include its promotional video, we can easily update the script to change the behavior of the interactive IoT experience.

Notice that if the computer running the script is powerful enough and connected to a screen, we can take advantage of this by using a web browser to interpret the script, project the video, and play the sound. We simply need to use the integrated HTML5 `<video>` element and include it within the `<body></body>` tags:

```
...
  <body>
    <video id="a6Video" muted controls></video>
  </body>
  <script src="domiot.min.js"></script>
  <script>
  ...
        // retrieve the commercial to play and play it using HTML5 video.
        const videoSrc = shelvingUnit.getAttribute('video-src');
        if (videoSrc) {
          a6Video.src = videoSrc;
          a6Video.load();
          a6Video.play();
        }
...
```

Other smart store enhancements could be implemented, for example adjusting the behavior of the shop lights based on the time of day, month of the year, or latitude, and adjusting the air conditioning based on local weather conditions.

2. Introduction

Interactive shops and homes, intelligent surveillance systems, smart logistics and factories, and many other IoT systems rely on software to define their behavior.

However, at the software level, IoT systems are often represented in terms that mirror their physical components (e.g., `motion_detector_003`, `relay_026`, `screen_002`) rather than abstractions meaningful to humans (e.g., `room`, `blue`, `door`). This may explain why behavior is frequently implemented by linking low-level input and output components such as connecting `motion_detector_003` to `relay_025` and `relay_026` instead of expressing richer interactions between semantic elements such as "when a `person enter`s a `room`, the `lamp`s gradually increase their `brightness`, and `music` begins to `play`".
Moreover, not using a standardized programming interface (API) for managing the elements of an IoT system and their interactions may contribute to the use of rigid and ad hoc architectures. Sometimes, this even result in the misapplication of data structures like maps or JSON, intended for organizing data, to "script" device behavior, rather than using scripts.
As a result, developers often struggle to understand, modify or debug such systems. This situation also hinders interoperability and lowers the quality of data collection, which in turn affects data analysis.

Developers need a technology that encourage describing interactive IoT systems using semantic elements, making explicit their meaning and their relationships, coupled with a programming interface to manipulate them. Ideally, such technology should be simple, easy to learn, use, and adopt.

HTML is a markup language that allows developers to define custom tags and attributes with domain-specific meaning. This makes HTML well-suited for describing IoT systems.
The Document Object Model (DOM) is a programming interface that represents an HTML document as a tree of nodes, and which methods allow scripts to dynamically access and change the tree content. It also provides methods for listening to and responding to user or system-generated events [3]. This means the DOM can represent an IoT system as a structured tree of nodes and enable scripts to react to physical events and update the system.

HTML and DOM appear to be well suited for the development of interactive IoT systems. Moreover, these technologies are open, platform-independent and easy to learn. They are widely adopted, enjoy support from a large community, and are familiar to millions of developers.

In this paper, we propose using HTML to describe IoT systems, and extending the DOM programming interface to enable the updating of physical components using JavaScript, just as we update components on a web page. The DOM for IoT (DOMIoT) fully preserves the original DOM API, exposing exactly the same methods as those used in web development such as `getElementById`,

`setAttribute` and `addEventListener`.

Other initiatives, such as the W3C Web of Things [4], have made valuable contributions by defining APIs for IoT interaction. However, we argue that the functionality provided by HTML and the DOM API is sufficient, has long been available, and is well suited to this purpose with minimal tuning. We believe our proposal is compatible with the W3C's vision.

This work builds upon concepts introduced in a prior patent by the author [5], which proposed using HTML, the DOM and a web browser to describe and orchestrate IoT systems. The solution enabled junior developers and trainees, familiar only with web development, to contribute effectively to building IoT experiences.

Some terms in this document are defined and described in the HTML standard and the DOM standard [1, 2, 3] in the context of web programming. The terms presented in this paper are in the context of IoT.

3. IoT System

An IoT system is a network of physical devices that sense, communicate, react and interact with the environment, and that can collect data.

4. I/O

Inputs are signals, data, or information collected from the environment, typically through sensors or networks. Examples include: temperature readings, motion detection signals, button press/release signals, GPS coordinates or weather data received from the internet.

Outputs are signals, data, or information sent to update or affect the environment, typically directed to actuators, displays, or server interfaces. Examples include: a signal sent to a relay, an alert sent via the internet or a an information transmitted to an AI agent.

Input components produce electrical or digital signals that are processed by the drivers, while output components receive electrical or digital signals output by the drivers.

5. Drivers

Drivers are programs that enable communication with physical I/O components. The DOM for IoT uses bindings to allow communication between the DOM elements and the drivers.

6. HTML Document

In the context of IoT, an HTML document is a strucured and hierarchical description of an IoT system. The HTML document is formed of contextual semantic elements such as `room`, `product`, `video` and `wardrobe` and attributes such as `locked=true`, `message="Welcome"` and `style="background-color:white;"`, related to how the IoT system is *perceived* by a user or a non-technical person, which for most cases do not correspond with the physical input and output components of the system such as cameras, motion detectors, and LEDs.

```
<html>
  <iot-hotel name="Domos Hotel">
    <iot-room id="room334">
      <iot-door id="r334MainDoor" locked message="Bienvenue à Paris"
                                          binding="lockBinding LCDBinding">
    </iot-room>
  </iot-hotel>
</html>
```

Figure 3: Simplified example of an HTML document of a smart hotel.

When the HTML describing an IoT system is parsed the result is a DOM tree.

The **DOM tree** is a data structure that represents the HTML document as a hierarchical tree of Node objects. Each node corresponds to a document component, such as element or attribute. This tree structure defines parent-child and sibling relationships. The DOM tree can be accessed, traversed and modified using the DOM for IoT API. To link the DOM tree elements with the physical components the DOM for IoT use bindings.

```
html
|-- iot-hotel -- name = "Domos Hotel"
    |-- iot-room -- id = "room334"
        |-- iot-door --- id = "r334MainDoor"
                     |-- locked
                     |-- message = "Bienvenue à Paris"
                     |-- binding = "lockBinding LCDBinding"
```

Figure 4: Simplified example of a DOM tree of a smart hotel.

7. DOMIoT

The DOMIoT (Document Object Model for IoT) is a programming interface (API) that allows to dynamically access and change the content of the nodes of a DOM tree, dispatch and listen to events, and communicate with the drivers of the physical components of an IoT system. Its methods mirror those of the DOM in web development, elements can be accessed using methods such as `document.getElementById` and `document.querySelectorAll`. Attributes can

be accessed using the `el.getAttribute` method and modified (including style properties) using methods such as `el.setAttribute` and `el.style.setProperty`. Events can be dispatched using `el.dispatchEvent` and listened using the `el.addEventListener` method.

```
const door = document.getElementById('r334MainDoor');
if (!door) return;

door.addEventListener('change', (ev) => {
  if (ev.target.locked) {
    door.setAttribute('message', 'Locked.');
  } else {
    door.setAttribute('message', 'Unlocked.');
  }
});

door.setAttribute('unlocked', true);
```

The script above retrieves the `door` of room 334 and sets and `unlocked` attribute using the `setAttribute` method, triggering a `change` event. Since the `door`'s `locked` attribute is not present, it updates the `door`'s `message` attribute value to "Unlocked".

8. Elements

An element refers to a single node in the HTML document structure, typically representing part of the logical structure (e.g. `<iot-room></iot-room>`, `<iot-door/>`, `<iot-wardrobe></iot-wardrobe>`).
Elements have attributes associated that provide additional information.

9. Attributes

Attributes are included inside the opening tag of elements in an HTML document.

```
<iot-door id="r334MainDoor" locked message="Bienvenue à Paris"
                                    binding="lockBinding LCDBinding">
```

In this example, the `message` attribute contains a text message to display on the door's LCD screen, while the `locked` attribute indicates that the the door is locked (using an electronic lock).

Modifying an attribute can trigger an update to the physical components. For example, replacing the `locked` attribute with `unlocked` in a `door` element causes the electronic lock to unlock and displays "Unlocked" on the door's LCD screen, as demonstrated in the example in Section 7.

The special `style` attribute allows you to apply CSS directly to an element to control its appearance, as demonstrated in the supermarket example in Section 1.

10. Events

An event is an action or occurrence. Events can be user actions, such as `press` [a button] or `move` [in a room] or occurrences, such as [heart] `beat` or `change` [in a battery level]. Events have a type (e.g. `press`, `release`, `move`) and can have values attached such as `x` and `y`. Events also have a `target`, which is the element on which the event was originally triggered.

```
{
  "type": "move",
  "target": {
    "id": "person1",
    "tagName": "IOT-PERSON"
  },
  "detail": {
    "x": 11500,
    "y": 25200
  }
}
```

Figure 5: Partial structure of an event

As an example, if a person is moving within the gym of a hotel toward the sauna, `move` events are dispatched on an `iot-person` element. The `move` event carries two values, `x` and `y`, corresponding to the position of the `iot-person` element. Events can be listened to using the `addEventListener` method:

```
person.addEventListener('move', (ev) => {
  const position = ev.detail;
  if (position.y > 30000 && position.x > 12000 && position.x < 18000) {
    const audioAdvisory = document.getElementById('saunaSafetyInstructionsAudio');
    if (audioAdvisory) {
      audioAdvisory.play().catch(err => console.error('Audio play failed:', err));
    }
  }
});
```

The script above uses the `addEventListener` method to listen for the `move` event, triggering an audio advisory on sauna safety when a person enters a designated area of a hotel gym.

11. Bindings

The `binding` elements link the DOM elements with physical components.

They have two core attributes: `id` and `location`, without which they can't work.

```
<iot-lock-binding id="lockBinding" location="/dev/lock">
<iot-LCD-binding id="LCDBinding" location="/dev/lcd">
```

The `id` attribute identifies the binding so it can be referenced by other elements. The `location` attribute specifies a reference to the driver with which the binding communicates. This reference can be a file path (as in the example above), a URL, a configuration formatted like a `style` attribute, or any other type of reference. In all cases, the binding logic and operation are determined by its type and implemented within each binding class.

Bindings listen for changes in the attributes of the elements that reference them (including CSS property changes). They parse these attributes into values interpretable by a driver and then communicate these values to that driver. Bindings also consume and interpret values from drivers, updating the DOM accordingly by modifying an attribute or dispatching an event.

For an element to use a binding, it must include the binding's `id` in its `binding` attribute. An element can use more than one binding. In the following example an `iot-door` element uses two bindings identified by `lockBinding` and `LCDBinding`.

```
<iot-door binding="lockBinding:0 LCDBinding:0" unlocked>
```

Numeric indexes can be specified in the `binding` attribute value to indicate the element's position within the binding, though it is not mandatory. In the example above, `iot-door` has index `0` in both bindings. Indexes help indicate the position of parsed values within the driver's communication message. The index is specified by appending a number (from 0 to n) after a colon character ':' following the binding id. If no index is provided, the binding assigns one based on the order of appearance of elements in the HTML document.

```
<iot-door binding="lockBinding LCDBinding" unlocked>
```

In this example, no indexes are specified. Assuming these are the first references to `lockBinding` and `LCDBinding` within an element, the `iot-door` element will have the index `0` in both bindings. The result is the same as in the previous example that specifies indexes.

12. System Operation

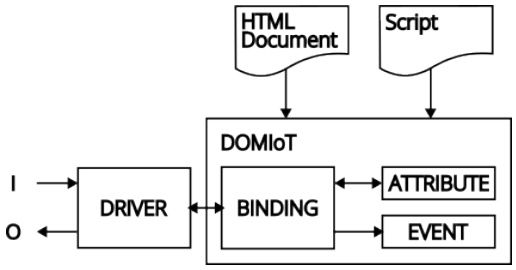

Figure 6: Setup and operation of an IoT system using HTML and the DOMIoT

The setup and operation of an IoT system using HTML and the DOMIoT is as follows:

Physical components (e.g., sensors, actuators, displays, etc.) should be in place and connected directly or indirectly to a computer such as a single-board computer (SBC) with the appropriate drivers installed, along with a DOMIoT implementation running. Once the HTML is parsed and the DOM tree is generated, the DOMIoT establishes the declared bindings, linking DOM elements to physical components. Scripts can use the DOMIoT API to access and modify element attributes through methods such as `getElementById`, `getAttribute`, `setAttribute`, and `setProperty`. When a `binding` detects a relevant change in an attribute (including CSS properties), it translates this change into a driver-understandable value and communicates it to the driver, for example, by writing to a driver file. In the other direction, physical changes are communicated from the physical components to the bindings through their drivers, allowing the bindings to update element attributes (including CSS properties) and/or dispatch events. Scripts listening to these events, which are triggered by physical interactions, an can then react by updating attributes, resulting in changes to the physical components of the interactive IoT system.

13. Semantic Structure

When we refer to semantics in this paper, we do not mean semantic HTML, although the concept is inspired by it. Since there is currently no HTML for IoT standard, IoT elements are for the moment custom elements and do not carry a predefined semantic meaning. Their semantics are instead defined by the developer based on the domain of their use, making them inherently domain-dependent. As a result, these semantics may not translate directly across different domains. Given the wide variety of IoT applications, it may be wise not to aim for a universal, cross-domain standard. We suggest grouping and documenting element implementations by domain. Considering this, we can state that for elements, attributes, and events to be semantic, they should clearly convey their meaning and role within their domain to developers, machines, data analysts, AI agents, and others.

Table 1 presents examples comparing non-semantic and semantic elements.

Semantics in IoT are often addressed through topics such as Semantic IoT and IoT ontologies. Some initiatives, such as the W3C's Web of Things (WoT) [4], provide frameworks that can be used to describe and thus document elements' semantics. However, it is important to acknowledge that some of these approaches may introduce a level of complexity and overhead that is not well-suited to all IoT scenarios. Just as HTML elements, attributes, and events on the web are sufficiently self-descriptive through their representation, elements, attributes, and events in IoT, should be sufficiently self-descriptive through an HTML or a JSON representation.

Non-semantic elements	**Semantic elements**
Physical space + motion sensors + camera + image recognition software	`room`, `person`
Door + Door lock + LCD screen	`door` element with `locked`/`unlocked` and `message` attributes
LED module + enclosure	`lamp` with `style` and `brightness` attributes
Relay + Different color LED rope lights	`background-color` style of a `shelving-unit`
Touch sensor	`button`
4 touch sensors	`slider`

1000 sensors
+ 500 actuators
+ 20 displays
	`supermarket`
	`hotel`
	`house`

4 touch sensors touched one after another in a certain order	`slide` event dispatched on a `slider` element
Physical button pressed	`press` event dispatched on a `button` element

Table 1: Examples of non-semantic vs semantic elements.

A **Non-semantic structure** refers to the lack of inherent meaning or purpose

in the chosen elements or sub-structures. Its form or usage is not primarily guided by intrinsic meaning but by what is actually there (a collection of sensors and actuators). In such a structure, the hierarchy may not accurately reflect the relationships between elements. The structure may be correct, but its lack of meaning erodes understandability.

A **Semantic structure** refers to the organization or arrangement of elements within a given system where this organization is predicated on the meaning, role, and logical interrelationships of these constituent elements. This organization goes beyond the underlying technical structure.

The HTML document and the DOM tree are represented as hierarchical trees, reflecting how elements are nested and related as parents, children, and siblings. These technologies encourage developers to consider elements in terms of their meaning, role, and relationships within these structures. Attributes convey both descriptive properties and the current state of elements, enhancing their semantic meaning.
Even though HTML and the DOMIoT encourage the use of semantic structures, they do not, per se, enforce it. The adoption of a semantic structure is ultimately the developer's responsibility.

14. Potential

The following outlines some of the potential offered by this approach.

As demonstrated in the example in Section 1, web applications can be integrated with IoT. By combining HTML, DOMIoT, and a web browser with a lightweight server running on a computer such as a single-board computer (SBC), it is possible to interconnect web elements with IoT elments within the same HTML structure and script.

HTML and the DOM are among the most widely used technologies, supported by a vast community and a rich ecosystem of tools and libraries, all of which are fully compatible with the approach presented in this paper.

Script files can be generated dynamically by a program that assists the user through an intuitive interface to define behavior. System behavior can then be updated remotely simply by replacing the script file, with no compilation required.

General-purpose AI models have been extensively trained on HTML and web-based JavaScript. When provided with an HTML document describing an IoT system, such AIs are well-equipped to generate scripts using the DOMIoT.

Using a semantic structure facilitates communication with AI agents. For example, consider the message: "I'm coming home. You can start the washing machine. When I'm five minutes from home, please prepare coffee. When I open

the door, I want a relaxing scenario." In this context, elements such as `person`, `house`, `washing-machine`, `coffee-machine`, `lamp`, and `audio` are semantically meaningful and interpretable by agents. AI agents can use these well-defined concepts not only to execute simple commands, but also to script, execute, store, and improve scenarios over time.

The structure of a semantic event contains richer data than a simple sensor-value pair does, including a type, the element that triggered the event, and associated values. Events can be sent over the internet to an event broker for use in analytics, metrics, or statistical studies.

The examples in this paper do not cover distributed systems. However, in real-world scenarios, such as smart homes, hotels, and shops, it may be undesirable to centralize all behavior. In such cases, a decentralized architecture can be adopted, where multiple subsystems operate independently, each processing its own HTML document and associated script, much like separate pages of a website. These subsystems can communicate over a local network using the `binding` element or other methods to coordinate actions.

15. Conclusion

We propose using HTML and an extension of the DOM for IoT (DOMIoT) to develop interactive IoT systems. HTML is used as is, while DOMIoT exposes the same methods as the standard DOM API in web development, with additional binding capabilities that enable communication with physical components via drivers.

The hierarchical nature of HTML and the DOM tree encourages (although does not enforce) the use of semantic elements and structures, while the DOM API provides a standardized way to access, modify, and interrelate these elements.

Taken together, these technologies and principles contribute to producing structures, code, and data that are easier to understand for multiple actors, such as developers, other systems, AI agents, and data analysts, resulting in more maintainable and adaptable code, improved interoperability, and higher data quality.

The proposed approach may inform future efforts to build more intelligent IoT systems.

Acknowledgments

I would like to sincerely thank all my coworkers at Indigo. In particular, I am grateful to Alexander Fuentes for the insightful discussions that helped shape the initial specifications in 2018-2019, and to Mohamed Akaarir for critically challenging these specifications and helping to refine the solution. I also extend my thanks to all colleagues who participated in testing and validating the solution.

Furthermore, I wish to thank CEOs Laurent Meoni and Guillaume Waline for their continuous support.

References

[1] WHATWG, *HTML Living Standard*, `https://html.spec.whatwg.org/`

[2] WHATWG, *DOM Living Standard*, `https://dom.spec.whatwg.org/`

[3] Mozilla Foundation, *Document Object Model (DOM)*, `https://developer.mozilla.org/docs/Web/API/Document_Object_Model`

[4] W3C, *Web of Things*, `https://www.w3.org/WoT/`

[5] Alvaro Fernandez-Yanez, Guillaume Waline, and Laurent Meoni, *Interactive Communication System Especially for the Analysis of Interaction Events*, French Patent FR3101178A1, filed September 2019, published March 2021, available at: `https://patents.google.com/patent/FR3101178A1/en`