The Role of APIs in Your IoT Solution

What is an API?

An **API (Application Programming Interface)** is a set of rules and protocols that allows two software applications to communicate with each other. In the context of your .js files, you are using a **REST API** (Representational State Transfer), which communicates over the internet using standard HTTP methods (like GET, POST, PUT).

In your applications, the client-side JavaScript is acting as the consumer:

- **To read data:** It uses a GET request to ask a device/server for a status (e.g., *Is the fan on?* or *What is the rainfall total?*—which would be replacing the Mockoon data).
- **To send a command:** It uses a POST or PUT request to change a state (e.g., *Turn on the sprinkler*).

Mock API vs. True Native API

Feature	**Mock API (Mockoon)**	**True Native API (Shelly/Tasmota)**
Purpose	Simulates the API's behavior for early-stage development and testing edge cases.	Provides real-time control and data from a physical device.
Dependency	None; it runs locally and is fully predictable.	Requires a physical, powered, and networked device.
Response	Fixed or easily dynamic data. **Instant** response time, low latency.	Real, often dynamic data. Response time includes device processing and network latency.
Key Benefit	Allows frontend and backend teams to develop in **parallel** and test error	**Final validation** that your application can successfully

Feature	Mock API (Mockoon)	True Native API (Shelly/Tasmota)
	handling (e.g., a "500 Internal Server Error") without impacting a live system.	command and monitor a physical object in the real world.

Transitioning from Mock to Real API Endpoints

The transition is straightforward because both your Mockoon setup and a real device's API (like Shelly or Tasmota) are structured similarly—they are typically **RESTful endpoints** accessed via HTTP.

The modification in your JavaScript file would primarily involve changing the target **URL**.

1. Update the Endpoint URL

If your Mockoon mock was set up to respond to http://localhost:3000/api/v1/sprinkler/status, your code would look something like this:

JavaScript

```
const SPRINKLER_API_URL = 'http://localhost:3000/api/v1/sprinkler/status';

fetch(SPRINKLER_API_URL)
    .then(response => response.json())
    // ... rest of the code
```

To use a real Shelly or Tasmota device, you would replace that URL with the device's local IP address and its specific command path:

JavaScript

```
// Example for a real device
const SPRINKLER_API_URL = 'http://192.168.1.50/relay/0?turn=on'; // Shelly API command
// OR
// const SPRINKLER_API_URL = 'http://192.168.1.51/cm?cmnd=Power%20ON'; // Tasmota command

fetch(SPRINKLER_API_URL, { method: 'POST' }) // Use the appropriate HTTP method
    .then(response => {
        if (response.ok) {
            console.log('Device command sent successfully.');
        }
    })
    // ... rest of the code
```

2. Match the Command Structure

A real device API often has a very simple command structure. For instance:

- **Shelly:** To get status: http://[DEVICE_IP]/status. To toggle relay 0: http://[DEVICE_IP]/relay/0?turn=toggle. Shelly supports a **well-documented REST API** right out of the box.
- **Tasmota:** Uses a simple **command structure** that is often sent via a GET request, such as http://[DEVICE_IP]/cm?cmnd=Status%2010 to get a detailed status or http://[DEVICE_IP]/cm?cmnd=Power%20ON to turn it on.

The key is to consult the specific API documentation for the device/firmware you are using (Shelly, Tasmota, etc.) to ensure your JSON data or URL parameters match what the device expects.

Tasmota and Shelly: Ideal for Real API Testing 💡

You are spot-on in identifying these products as cheap, accessible options for testing. They are popular for rapid prototyping and home automation for several reasons:

1. **Local Control (Native API):** Both Shelly and devices flashed with Tasmota or ESPHome prioritize **local network control**, meaning your browser/application can talk directly to the device via its IP address without needing the manufacturer's cloud service. This provides a true **Native API**.
2. **Affordability:** Smart plugs and relay switches utilizing the ESP8266 or ESP32 chipsets (which Tasmota and ESPHome run on) are very inexpensive, making them an ideal test platform.
3. **Well-Documented APIs:**
    - **Shelly** devices come with their own firmware that provides a full, documented HTTP REST API and MQTT support.
    - **Tasmota** is a **third-party open-source firmware** that you can flash onto many devices. It also provides an easy-to-use HTTP API (via cmnd=...) and is a standard way to ensure all your various devices have a consistent API.

Native API Support in Commercial Products

The question of how many products support a **native API** (meaning a local, direct interface without relying on a cloud server) is critical in the IoT space.

There is a growing trend toward providing local control, but it is not universal:

- **Enthusiast/Prosumer Devices:** Products designed for the enthusiast market, especially those using open platforms like **Home Assistant**, often provide a robust local API. **Shelly** is a prime example, providing a native local API and MQTT (a message-based IoT protocol) support right out of the box.
- **Cloud-Centric Devices:** Many mainstream, consumer-grade products (like some older smart bulbs, security cameras, or virtual assistants) are **cloud-dependent**. They do not have a documented local API; your commands are sent to the manufacturer's cloud server, which then relays the instruction back to the device over the internet. This model is often less desirable for developers because it introduces external points of failure (cloud service downtime) and latency.
- **Open-Source Firmware:** For cloud-dependent devices that use popular chipsets (like the ESP8266), the developer community has created open-source alternatives like **Tasmota** or **ESPHome**. These firmwares effectively give the device a **new, local, native API** and full user control, which is the main reason they are so popular for local testing.

In short, while an exact number is impossible to determine, products that support **local (native) control** are typically highly valued by the developer and automation community, and are becoming a standard feature for devices that want to integrate into local-first smart home hubs.