

---

# **MEA - Makers a new approach to Electronic Applications**

Lecture Notes

Prof. Riccardo Berta

2025.09.26

## Contents

<b>1 Introduction</b>	<b>8</b>
1.1 Issues . . . . .	11
1.2 Cloud vs Fog computing . . . . .	12
1.3 Scenario example: the connected hotel . . . . .	14
1.4 Adopting Web technologies . . . . .	16
1.4.1 The Web as GUI . . . . .	18
1.4.2 The Web as an API . . . . .	19
1.4.3 A GUI for the "My Office" IoT system . . . . .	19
1.5 Semantic Gap . . . . .	25
1.6 Landscape . . . . .	26
1.7 Hands-on Activity . . . . .	27
<b>2 Edge Devices</b>	<b>27</b>
2.1 Arduino . . . . .	29
2.1.1 Makers Kit . . . . .	33
2.2 Programming . . . . .	34
2.2.1 Controlling input and output . . . . .	36
2.2.2 Timing . . . . .	36
2.2.3 Hello World! . . . . .	36
2.2.4 Data type, Selection, Repetition and Function . . . . .	38
2.2.5 Interrupt . . . . .	42
2.3 Serial communication . . . . .	47
2.3.1 Send and receive text . . . . .	47
2.3.2 Send and receive multiple text fields . . . . .	49
2.3.3 Binary data excange . . . . .	53
2.3.4 Synchronous communication . . . . .	59
2.4 Hands-on Activity . . . . .	64
<b>3 Sensors</b>	<b>65</b>
3.1 Specifications . . . . .	67
3.1.1 Transfer function . . . . .	67
3.1.2 Dynamic Range . . . . .	71
3.1.3 Accuracy . . . . .	72
3.1.4 Hysteresis . . . . .	73
3.1.5 Precision . . . . .	74
3.1.6 Resolution . . . . .	74

3.1.7	Dynamic characteristics . . . . .	75
3.1.8	Reliability . . . . .	76
3.2	Calibration . . . . .	76
3.2.1	Establishing a reference . . . . .	77
3.2.2	Exposure to the reference value . . . . .	77
3.2.3	Adjusting the sensor . . . . .	77
3.2.4	Validation . . . . .	78
3.3	Context Awareness . . . . .	78
3.3.1	Detecting presence . . . . .	79
3.3.2	Detecting Light . . . . .	82
3.3.3	Measuring distance . . . . .	87
3.3.4	Detecting vibration . . . . .	92
3.3.5	Measuring temperature . . . . .	95
3.3.6	Measuring physiological signals . . . . .	97
3.4	Self awareness . . . . .	99
3.4.1	Accelerometers . . . . .	99
3.4.2	Gyroscopes . . . . .	100
3.4.3	Magnetometers . . . . .	101
3.4.4	Orientation . . . . .	101
3.4.5	Sensor fusion . . . . .	109
3.5	Hands-on Activity . . . . .	113
<b>4</b>	<b>Javascript</b>	<b>113</b>
4.1	Basic features . . . . .	114
4.1.1	Variables . . . . .	115
4.1.2	Functions . . . . .	116
4.1.3	Equality . . . . .	118
4.1.4	Const, let and var . . . . .	119
4.1.5	Arrays . . . . .	121
4.1.6	Strings . . . . .	122
4.1.7	Parameter passing . . . . .	124
4.1.8	Optional arguments . . . . .	125
4.1.9	Recursion . . . . .	126
4.1.10	Error Handling . . . . .	127
4.1.11	Modules . . . . .	128
4.2	Code examples . . . . .	129
4.2.1	Factorial of a number . . . . .	129
4.2.2	Binary search . . . . .	130

4.2.3	Bubble sort . . . . .	130
4.2.4	Quicksort . . . . .	131
4.3	Functional Programming . . . . .	132
4.3.1	High order functions . . . . .	132
4.3.2	Filter, map and Reduce . . . . .	133
4.3.3	Closures . . . . .	135
4.3.4	Currying . . . . .	135
4.4	Object Oriented Programming . . . . .	137
4.4.1	Objects . . . . .	137
4.4.2	Factories . . . . .	139
4.4.3	Constructors . . . . .	139
4.4.4	Getter and Setter . . . . .	143
4.4.5	Classes . . . . .	145
4.4.6	Polymorphism . . . . .	147
4.5	Programming paradigms . . . . .	148
4.5.1	Procedural Programming . . . . .	149
4.5.2	Object Oriented Programming . . . . .	149
4.5.3	Functional Programming . . . . .	150
4.5.4	Imperative vs Declarative style . . . . .	151
4.6	More advanced topics . . . . .	154
<b>5</b>	<b>HTTP</b>	<b>155</b>
5.1	Basic features . . . . .	156
5.1.1	Resources and URLs . . . . .	157
5.1.2	Methods . . . . .	159
5.1.3	Status Codes . . . . .	160
5.1.4	Headers . . . . .	162
5.1.5	Message Body . . . . .	163
5.1.6	Example . . . . .	163
5.2	Content Negotiation . . . . .	164
5.3	Caches . . . . .	166
5.3.1	Client-side caching . . . . .	167
5.3.2	Server-side caching . . . . .	167
5.3.3	Cache-Control headers . . . . .	168
5.4	Conditionals . . . . .	169
5.4.1	Last-Modified and If-Modified-Since headers . . . . .	169
5.4.2	ETag and If-None-Match header . . . . .	170

5.5	Authentication . . . . .	171
5.5.1	Basic Authentication . . . . .	171
5.5.2	Digest Authentication . . . . .	171
5.5.3	Bearer Token Authentication . . . . .	172
5.5.4	Certificate-Based Authentication . . . . .	172
5.5.5	Server response . . . . .	172
5.6	Cross-Origin Resource Sharing (CORS) . . . . .	173
5.6.1	Access-Control-Allow-Origin header . . . . .	173
5.6.2	Other CORS headers . . . . .	174
5.7	JSON . . . . .	175
5.7.1	Structure . . . . .	175
5.7.2	Arrays . . . . .	177
5.7.3	Nester structures . . . . .	177
5.7.4	Schema . . . . .	178
5.8	Interacting with HTTP . . . . .	180
5.8.1	From the Browser . . . . .	180
5.8.2	From Javascript . . . . .	182
5.8.3	Postman . . . . .	184
5.9	Hands-on Activity . . . . .	185
<b>6</b>	<b>NodeJS</b>	<b>185</b>
6.1	Getting Started . . . . .	187
6.1.1	Writing applications . . . . .	187
6.1.2	NPM: Node Package Manager . . . . .	187
6.1.3	Modules . . . . .	187
6.1.4	Manifest . . . . .	189
6.1.5	Version Control . . . . .	191
6.2	Synchronous vs Asynchronous . . . . .	192
6.2.1	Synchronous, Single Thread . . . . .	192
6.2.2	Synchronous, Multiple Threads . . . . .	193
6.2.3	Asynchronous Execution Model . . . . .	194
6.2.4	Callback hell . . . . .	195
6.2.5	Function Chaining . . . . .	195
6.2.6	Event Loop . . . . .	196
6.3	The HTTP Module . . . . .	199
6.3.1	Streams . . . . .	200
6.3.2	Pipe . . . . .	201
6.3.3	Serving a big file . . . . .	201

6.4 Examples . . . . .	203
6.4.1 A file Server . . . . .	204
6.4.2 A Chat Server . . . . .	208
6.5 Express . . . . .	211
6.5.1 Request and Response Objects . . . . .	212
6.5.2 Routing . . . . .	212
6.5.3 Middleware . . . . .	213
6.5.4 A Simple Twitter Clone . . . . .	215
6.6 Why Node.js and IoT? . . . . .	216
6.6.1 Scalability and Performance . . . . .	216
6.6.2 Effective Communication and API Design . . . . .	217
6.6.3 Streaming and Real-Time Data Processing . . . . .	217
6.6.4 Resource Efficiency . . . . .	217
6.7 Hands-on Activity . . . . .	217
<b>7 REST API</b>	<b>218</b>
7.1 Remote Procedure Call . . . . .	219
7.1.1 RPC over HTTP: server side . . . . .	219
7.1.2 RPC client . . . . .	220
7.1.3 Limitations . . . . .	222
7.2 Architectural principles . . . . .	222
7.2.1 Constraint 1: Client-Server . . . . .	223
7.2.2 Constraint 2: Uniform Interface . . . . .	224
7.2.3 Constraint 3: Stateless . . . . .	226
7.2.4 Constraint 4: Cacheable . . . . .	228
7.2.5 Constraint 5: Layered System . . . . .	229
7.2.6 Comparison with SOAP . . . . .	231
7.3 Resources . . . . .	232
7.3.1 Principle 1: Addressable Resources . . . . .	232
7.3.2 Principle 2: Manipulation through Representations . . . . .	234
7.3.3 Principle 3: Self-descriptiveness . . . . .	235
7.3.4 Principle 4: HATEOAS . . . . .	236
7.4 Richardson Maturity Model . . . . .	238
7.4.1 Level 0: The Swamp of POX . . . . .	238
7.4.2 Level 1: Resources . . . . .	241
7.4.3 Level 2: HTTP Verbs . . . . .	243
7.4.4 Level 3: Hypermedia Controls (HATEOAS) . . . . .	244
7.4.5 The Glory of REST . . . . .	246

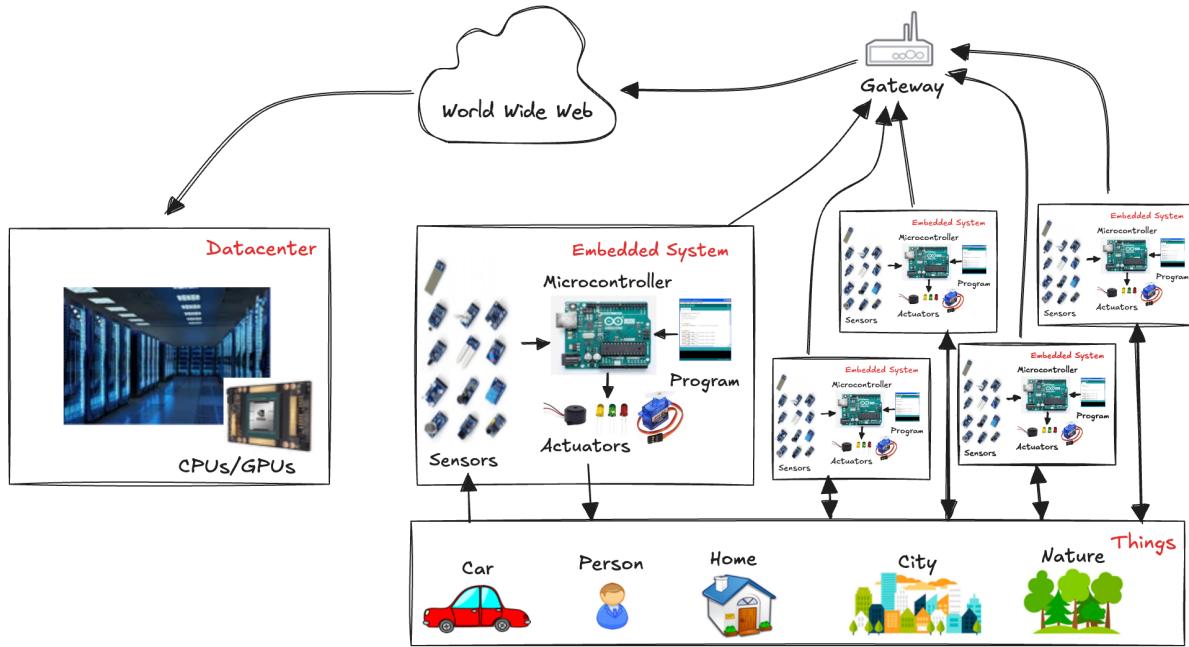
7.5	Real-World Examples . . . . .	246
7.5.1	The X (Twitter) API . . . . .	247
7.5.2	eBay API . . . . .	248
7.5.3	PayPal API . . . . .	249
7.5.4	Comparison . . . . .	250
7.6	Beyond REST: the Real-Time . . . . .	251
7.6.1	Basic Polling . . . . .	252
7.6.2	Publish/Subscribe pattern . . . . .	254
7.6.3	WebHooks . . . . .	255
7.6.4	Long Polling . . . . .	259
7.6.5	WebSockets . . . . .	262
7.7	Hands-on Activity . . . . .	265
<b>8</b>	<b>WoT Proxy</b>	<b>265</b>
8.1	Integration Strategy . . . . .	266
8.1.1	Direct Integration Pattern . . . . .	266
8.1.2	Gateway Integration Pattern . . . . .	267
8.1.3	Cloud Integration Pattern . . . . .	268
8.1.4	Design decision . . . . .	268
8.2	Resource Design . . . . .	269
8.3	Simple Thing . . . . .	270
8.3.1	Entry point . . . . .	271
8.3.2	Servers . . . . .	271
8.3.3	Resources . . . . .	272
8.3.4	Routes . . . . .	273
8.3.5	Benefits . . . . .	275
8.4	Bind the sensors . . . . .	276
8.4.1	Plugin Structure . . . . .	276
8.4.2	Real Hardware . . . . .	279
8.5	Representation Design . . . . .	282
8.5.1	Supported formats . . . . .	282
8.5.2	Middleware implementation . . . . .	283
8.6	Interface Design . . . . .	287
8.6.1	Adding PUT Support . . . . .	287
8.6.2	Observer Pattern . . . . .	288
8.6.3	Body Parsing Middleware . . . . .	291
8.6.4	Bidirectional Serial Communication . . . . .	292

8.7	Real-time data . . . . .	295
8.7.1	WebSocket Server . . . . .	295
8.7.2	Observer changes . . . . .	296
8.7.3	HTTP and WebSocket integration . . . . .	298
8.7.4	WebSocket Client . . . . .	299
8.8	Inegrate other devices . . . . .	300
8.8.1	CoAP device . . . . .	301
8.8.2	Gateway . . . . .	303
8.9	Test-Driven Development . . . . .	308
8.9.1	Using assert . . . . .	308
8.9.2	Unit tests . . . . .	309
8.9.3	Mocha . . . . .	310
8.9.4	Function Testing . . . . .	313
8.9.5	Testing asynchronous function . . . . .	314
8.9.6	Hooks . . . . .	315
8.9.7	Chai . . . . .	316
8.9.8	Bug cost . . . . .	319
8.9.9	Testing the WoT Proxy . . . . .	320
8.10	Hands-on . . . . .	322
<b>9</b>	<b>Persisting Data</b>	<b>322</b>
9.1	MongoDB . . . . .	323
9.1.1	Collection, document, and field . . . . .	324
9.1.2	Arrays and Embedded Documents . . . . .	327
9.1.3	Find documents . . . . .	328
9.1.4	ObjectId . . . . .	329
9.1.5	Indexes . . . . .	329
9.1.6	Query Selectors . . . . .	331
9.1.7	Projection . . . . .	335
9.1.8	Lazy Evaluation . . . . .	336
9.1.9	Ordering and Pagination . . . . .	337
9.1.10	Update documents . . . . .	337
9.1.11	Aggregation . . . . .	339
9.1.12	Replication and Sharding . . . . .	340
9.1.13	Transactions . . . . .	341
9.1.14	Profile and Backup . . . . .	342
9.2	How to model data . . . . .	344
9.2.1	Embedded Documents or Referencing? . . . . .	344

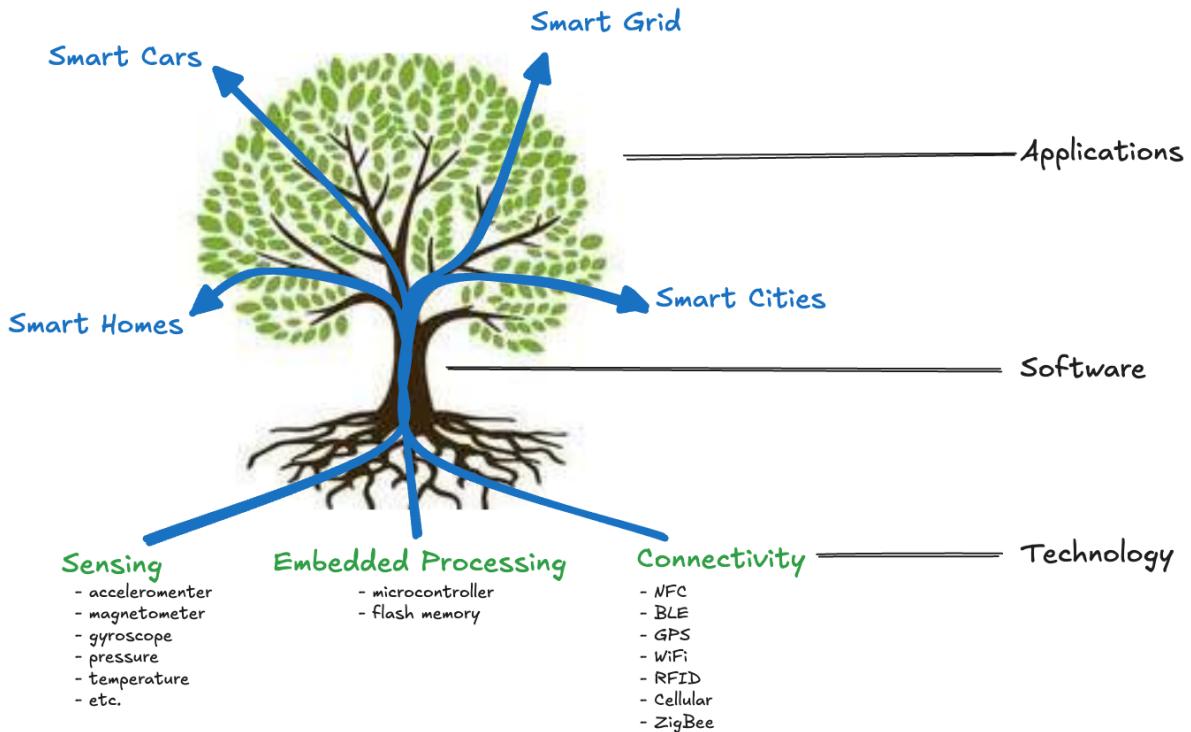
9.2.2	Few or Many Collections? . . . . .	344
9.2.3	Schema-on-read or schema-on-write? . . . . .	345
9.2.4	Write Performance and Data Durability? . . . . .	346
9.3	Object-Relational Mapping (ORM) . . . . .	347
9.3.1	Mongoose . . . . .	347
9.3.2	Schemas . . . . .	348
9.3.3	Persiste and fetch data . . . . .	349
9.4	WoT Proxy with persistence . . . . .	350
9.5	Hands-on Activity . . . . .	356
<b>10</b>	<b>Security</b>	<b>357</b>
10.1	The Share Layer . . . . .	358
10.1.1	Problem 1: Confidentiality . . . . .	359
10.1.2	Problem 2: authenticity . . . . .	360
10.1.3	Problem 3: authorization . . . . .	360
10.1.4	Encryption . . . . .	360
10.2	Transport Layer Security (TLS) . . . . .	361
10.2.1	Certificate . . . . .	363
10.2.2	Authorities . . . . .	364
10.2.3	Let's Encrypt . . . . .	365
10.3	Authentication and Access Control . . . . .	365

## 1 Introduction

The Internet of Things (IoT) is a concept that describes a **vast network of interconnected physical devices** that are embedded with **sensors, actuators, software**, and **connectivity technologies**. These devices are capable of **collecting data from their surroundings, exchanging it over the internet, and acting upon this data to influence the environment**. IoT enables automation, monitoring, and control of systems, making it a powerful tool for creating smarter and more efficient processes across various industries and applications. The following image illustrates a generic IoT system, emphasizing the interconnections of devices and systems across various domains. At the center is the Internet, the network that enables global communication. A gateway serves as the intermediary, connecting the global network to a localized systems. The local network comprises multiple **embedded systems**, each driven by a microcontroller. These microcontrollers execute programs that manage sensors (which gather data from the environment) and actuators (which perform actions based on that data). The data collected by sensors is often transmitted to a **datacenter**, where it is stored and analyzed:

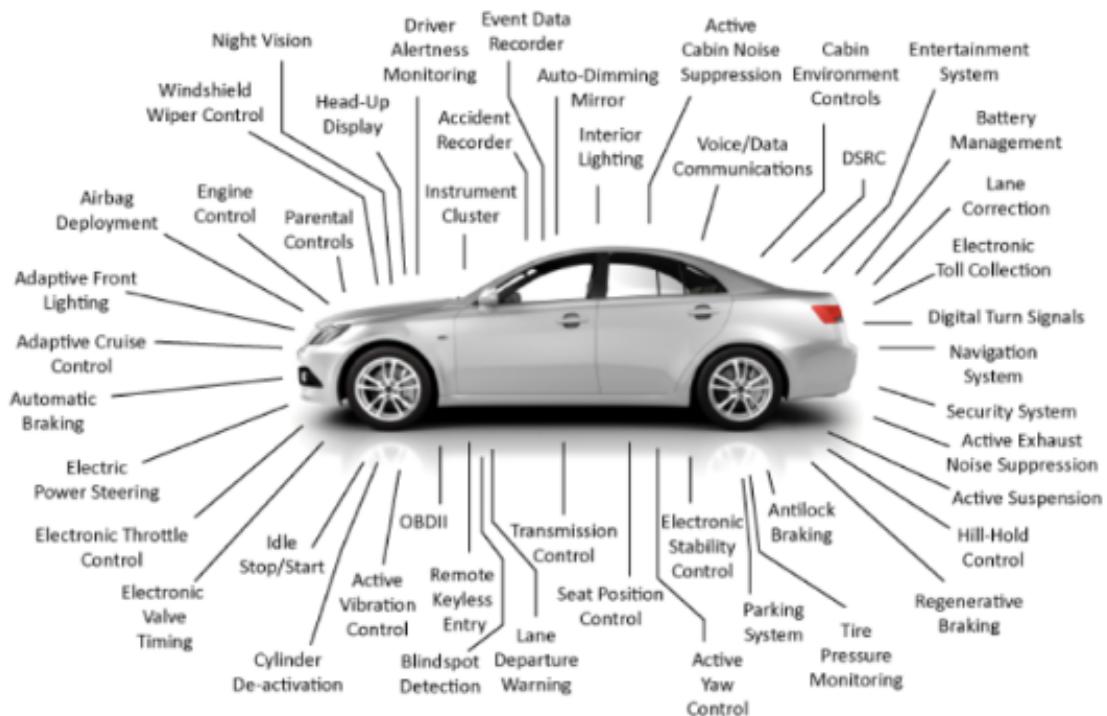


The IoT paradigm consists of three main components: **technologies**, **software**, and **services**.



At the roots, the **foundational technologies**, including sensing components such as accelerometers, magnetometers, gyroscopes, pressure sensors, and temperature sensors, are supported

by embedded processing elements like microcontrollers and flash memory, along with connectivity technologies such as NFC, BLE, GPS, Wi-Fi, RFID, Cellular, and ZigBee, which facilitate data exchange. The trunk represents the **software layer**, which acts as the central framework connecting the hardware to higher-level functions. From this foundation, the branches extend to various **IoT applications**, such as smart homes, smart cars, smart grids, and smart cities, symbolizing the diverse use cases and advancements driven by IoT technology. As an example, the integration of IoT in **smart cars** enhances connectivity, enabling real-time data exchange, advanced driver assistance, and seamless interaction with smart infrastructure for improved safety and efficiency:



A modern car is today surrounded by an array of advanced features and technologies. Several systems work together to enhance safety, comfort, performance, and connectivity in vehicles. These features include safety systems like airbags, adaptive cruise control, lane departure warning, and blindspot detection. There are also performance-enhancing technologies, such as electronic stability control, regenerative braking, and adaptive front lighting. Additionally, comfort and entertainment are addressed through systems like active cabin noise suppression, voice/data communications, and entertainment controls. Connectivity and monitoring are represented by components like event data recorders, navigation systems, and tire pressure monitoring.

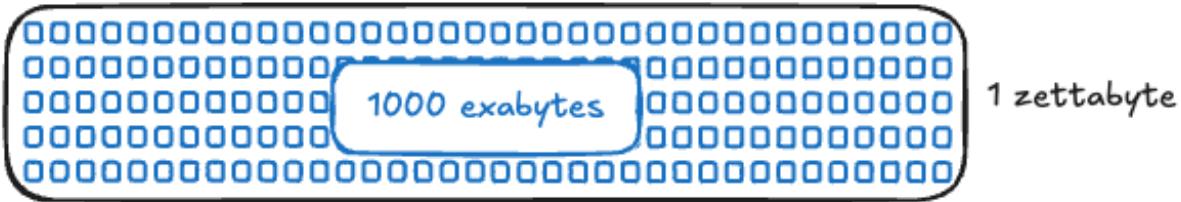
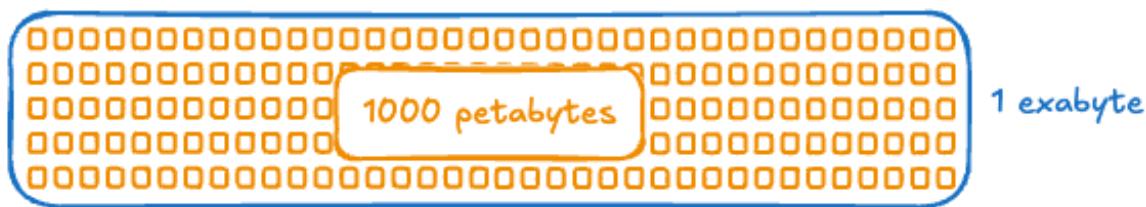
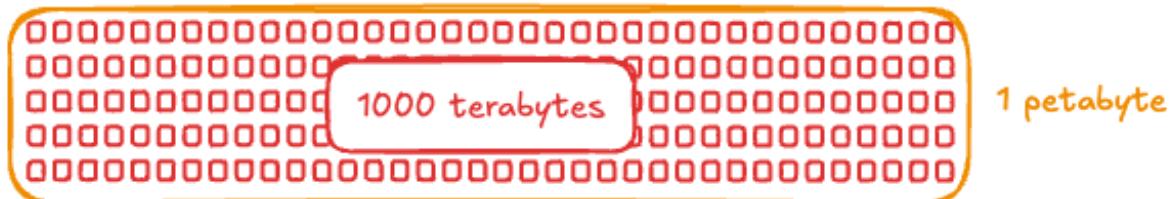
The IoT offers numerous **advantages** that make it a transformative technology. One major ben-

efit is **automation and efficiency**, as IoT systems reduce the need for human intervention while optimizing resource utilization. **Data-driven decision-making** is another significant advantage, as IoT provides real-time insights that can improve operations and planning. Additionally, IoT enhances **quality of life** by introducing smart systems into homes, cities, and healthcare environments, ensuring greater convenience, safety, and overall well-being. As of 2023, the IoT landscape is marked by **remarkable growth and adoption**. There are approximately **16.7 billion IoT devices worldwide**, a figure that surpasses twice the global population and is anticipated to exceed 27 billion by 2025. The global IoT market reflects this rapid expansion, with a valuation of **\$662.21 billion in 2023**, projected to surge to approximately \$3.3 trillion by 2030. A significant contributor to this growth is the increasing popularity of smart home devices; in 2022 alone, 857 million units were shipped globally, with this number expected to rise to 1.09 billion by 2027. These statistics highlight the pervasive impact of IoT across industries and households, underscoring its role in shaping the future of technology.

## 1.1 Issues

One major issue is the data problem, involving challenges in **volume, variety, velocity, veracity, and value**. IoT devices generate **enormous amounts of data**, often referred to as **big data**, which poses significant challenges in terms of storage, processing, and analysis. The sheer volume of data can overwhelm traditional data management systems:

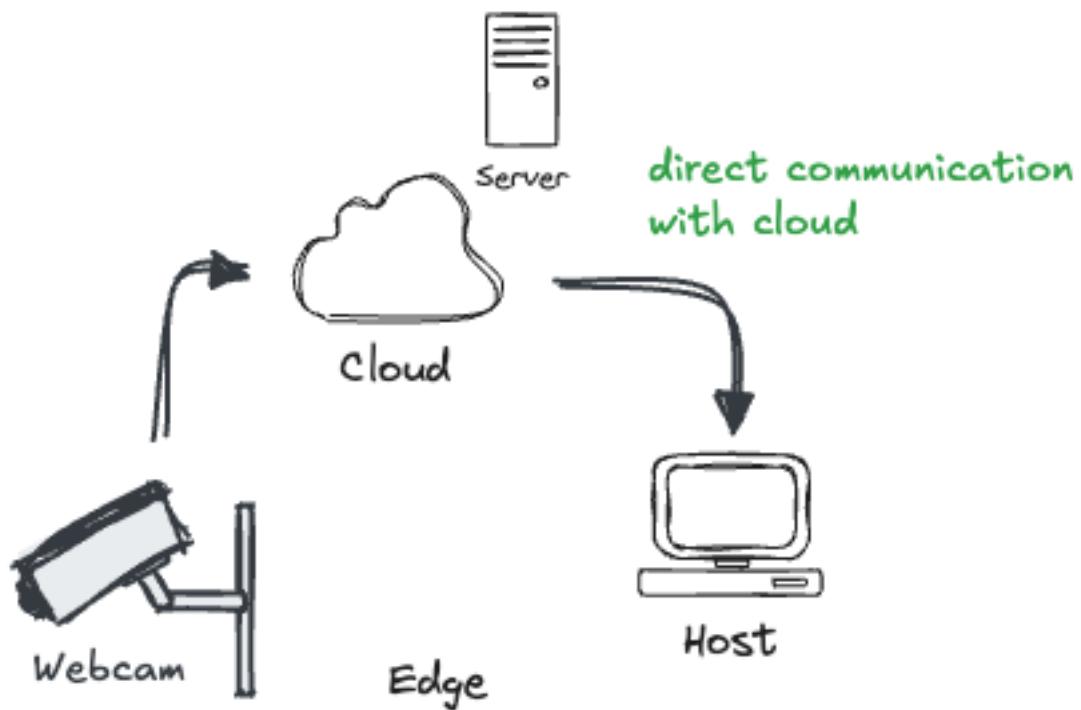
### □ 1 terabyte



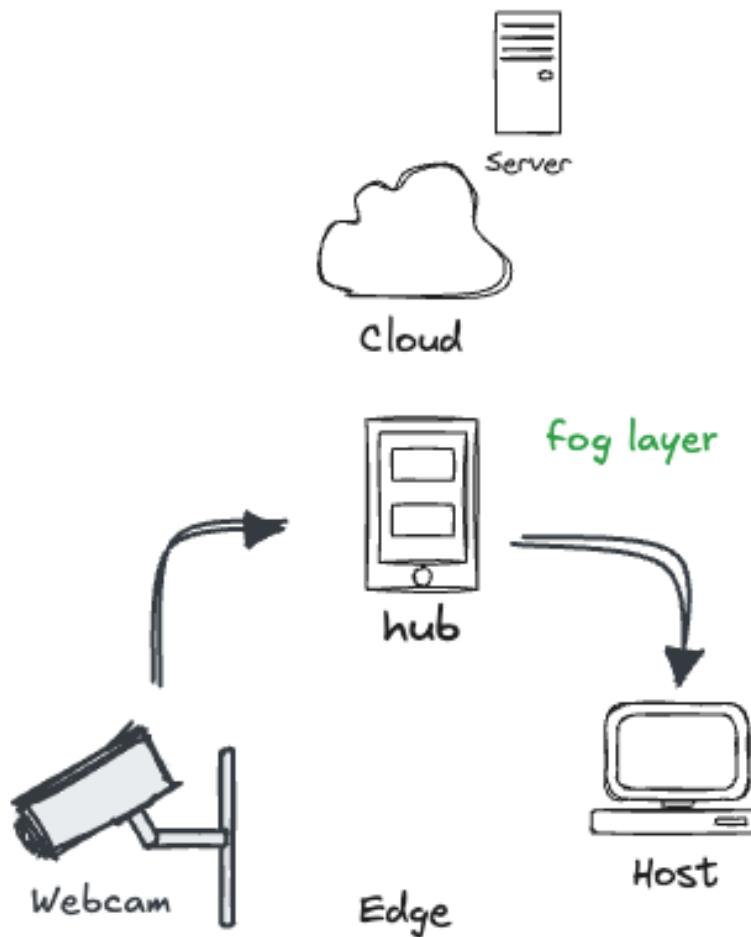
The variety of data is another issue, as IoT devices produce **diverse types of information**, such as sensor readings, video streams, and logs, often in different formats. This diversity makes integration, interoperability, and standardization difficult to achieve, especially when devices come from different manufacturers or use **incompatible protocols**. The **speed at which IoT data is generated** and needs to be processed, is also a significant challenge. Many IoT applications, such as real-time monitoring or autonomous vehicles, require immediate processing of data to make time-critical decisions. The veracity of IoT data, which refers to its **accuracy and reliability**, is another problem. Data collected by IoT devices can be noisy, incomplete, or prone to errors due to hardware malfunctions or environmental factors. Poor data quality can lead to inaccurate analyses, jeopardizing the effectiveness of IoT applications. Finally, the value of IoT data depends on the **ability to extract actionable insights** from it. Raw data often requires extensive preprocessing, aggregation, and analysis to reveal meaningful patterns, trends, or predictions. Additionally, concerns around **data privacy and security** compound these problems. IoT systems collect sensitive information, which, if not adequately protected, could lead to breaches or misuse. Striking a balance between data utility and privacy is an ongoing challenge in the IoT domain.

## 1.2 Cloud vs Fog computing

The existing cloud computing paradigm cannot address the challenges of the rapidly growing IoT ecosystem. **Cloud computing** refers to the delivery of computing services, including storage, servers, databases, and software, over the internet. It allows users to access and pay for services on demand without owning physical infrastructure. Data from devices, such as webcams, is transmitted directly to the cloud for processing and storage. While cloud computing provides a scalable and flexible solution, it can face challenges such as high latency, network bandwidth limitations, and security issues, especially in IoT applications requiring real-time responses:



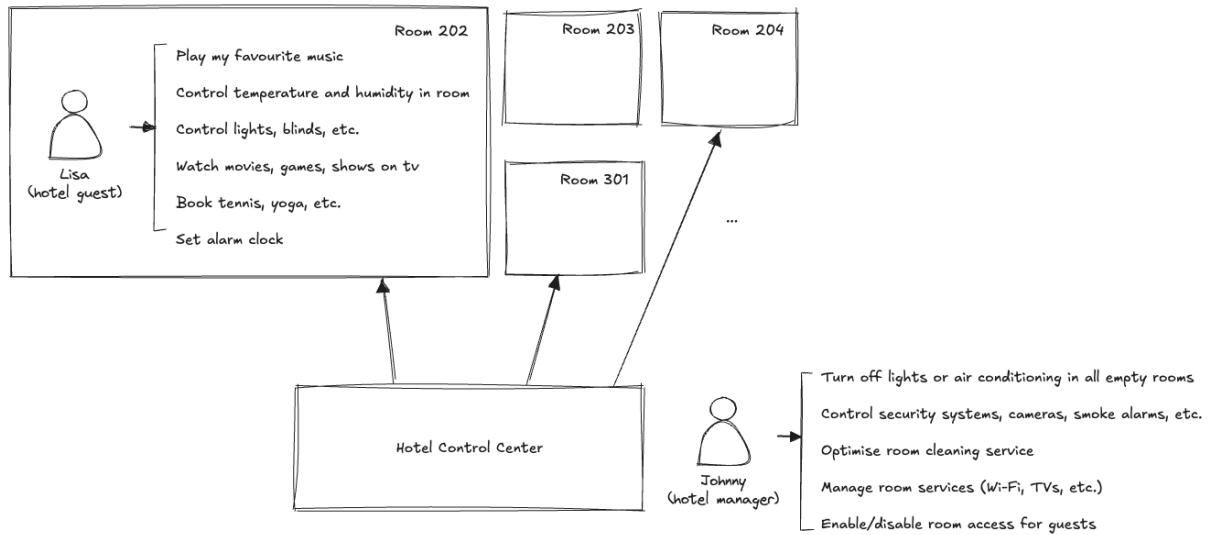
To address these challenges, **fog computing** introduces a **distributed paradigm** that extends cloud computing closer to **the edge of the network**. In this model, an intermediate "fog layer," represented by devices such as hubs or fog nodes, **processes data locally** before sending it to the cloud. This approach reduces latency and alleviates network congestion by handling tasks closer to data sources. For example, a webcam in a fog computing system would send its data first to a fog node for initial processing, minimizing the need to transmit large amounts of raw data to a remote cloud:



The growth of IoT underscores the need for fog computing, as the traditional cloud model struggles to sustain the increasing demands of scalability, reliability, and security. By adding a hierarchical layer between devices and the cloud, fog computing optimizes data management and supports the real-time processing capabilities required for modern IoT applications. This hybrid approach integrates the strengths of both cloud and edge computing, providing a more sustainable and efficient solution for the IoT-driven future.

### 1.3 Scenario example: the connected hotel

We consider a smart hotel scenario, where various systems are interconnected to provide enhanced guest experiences and operational efficiency:

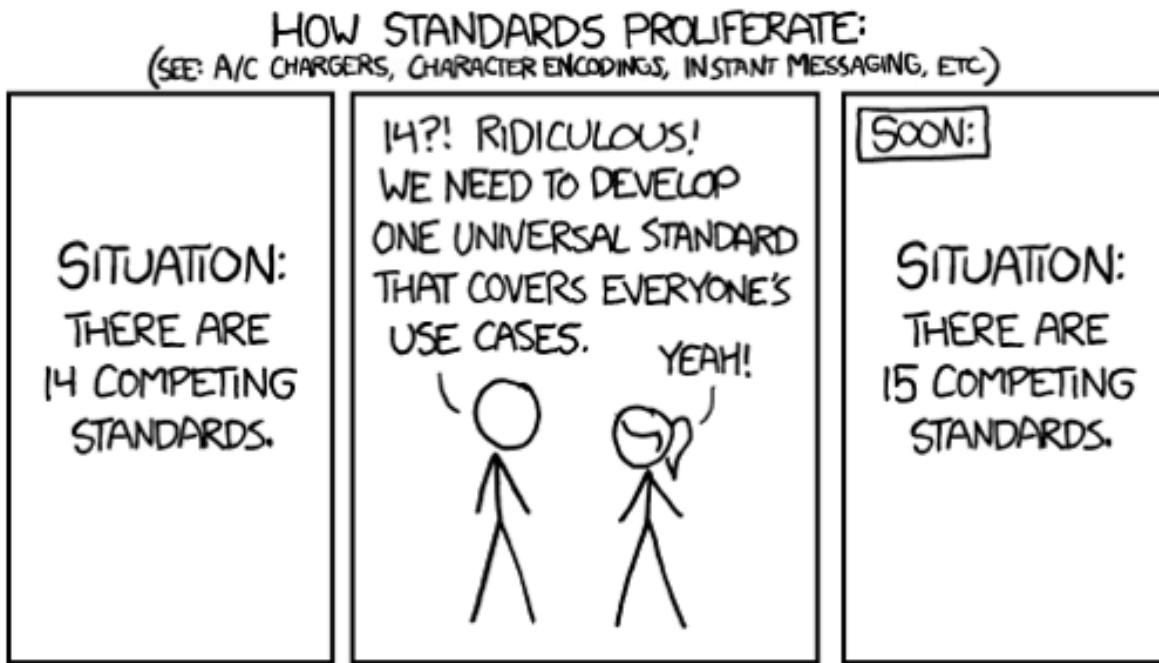


**Guest Perspective:** Lisa can interact with her room environment using a mobile device or in-room controls. She can customize her room settings, including music playback, temperature control, lighting, and entertainment options. She can also book services like tennis or yoga and set alarms.

**Hotel Control Center Perspective:** Johnny has access to a central control center where he can manage various aspects of the hotel. This includes: energy management, security and safety, room services, cleaning efficiency and guest access.

The diagram shows how the guest interface and the hotel control center are connected, enabling two-way communication and data exchange. This allows for real-time monitoring, personalized services, and proactive management of hotel operations. This is a simplified representation of a connected hotel. In reality, the system could be integrated with other hotel systems like reservation, billing, and customer relationship management for a more comprehensive solution.

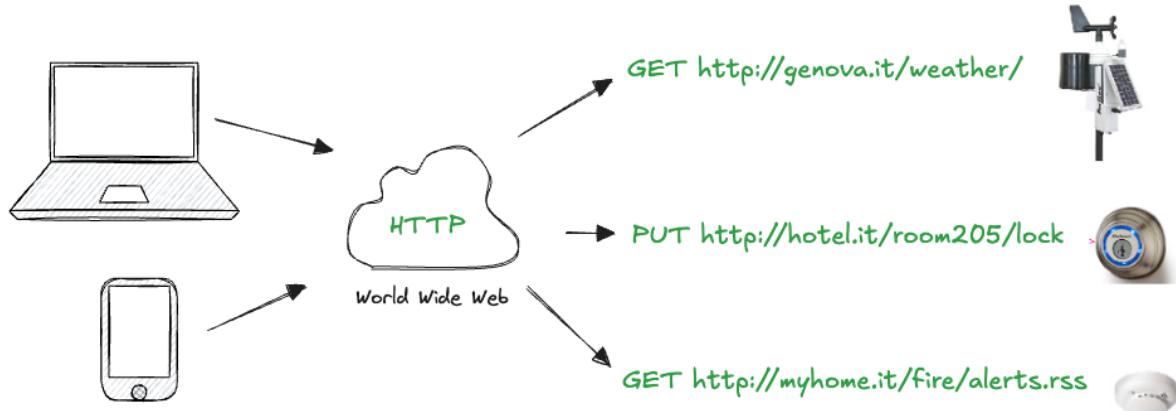
The smart connected hotel faces **several challenges**. One of the primary issues is the **complexity and time required to integrate disparate systems** and devices into a cohesive network. This difficulty stems from the need to manage varying protocols, data formats, and integration methods unique to each vendor solutions. Additionally, the lack of standardized data formats often leads to **data silos**, where valuable information (such as guest preferences, energy consumption, or maintenance requests) remains isolated, limiting its potential for analysis, sharing, and actionable insights. This fragmentation not only hampers operational efficiency but also introduces **security vulnerabilities**. Systems with differing security protocols and levels of protection increase the risk of cyberattacks and data breaches. A promising solution lies in the **adoption of common standards**, however, a significant risk remains: the **proliferation of competing standards**, a dilemma humorously captured in the following comic:



This risk stems from the fact that various organizations and industries have independently developed their own IoT standards, leading to a lack of uniformity across the ecosystem. This lack of consistency poses several challenges. Firstly, it hinders interoperability between devices and systems from different manufacturers, making it difficult for them to communicate and work together seamlessly. Secondly, it complicates the integration process, as systems architects must navigate a maze of disparate standards to connect various IoT components. Finally, this lack of standardization can slow down the adoption of IoT solutions on a broader scale, as businesses and consumers may be hesitant to invest in technologies that lack compatibility and create uncertainty around future upgrades and expansions.

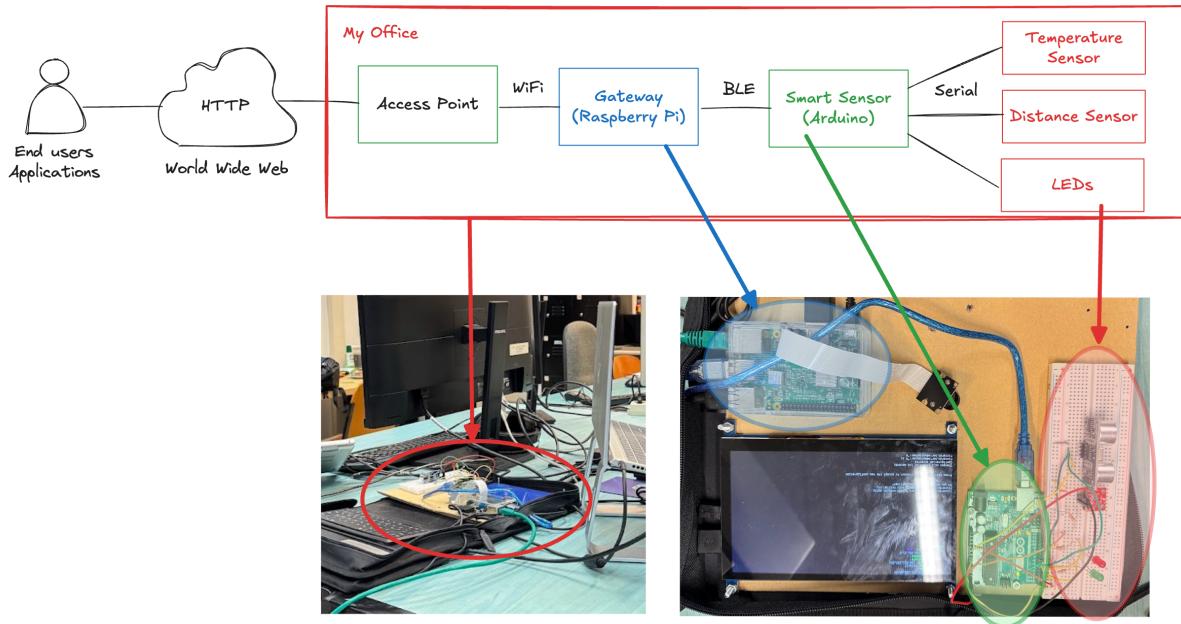
#### 1.4 Adopting Web technologies

The **Web of Things (WoT)** is a concept designed to address the standardization issues in the IoT ecosystem. IoT encompasses a wide range of devices and technologies, often with different communication protocols, data formats, and functionalities, making it difficult to interconnect and integrate these devices seamlessly. The WoT idea relies on **reusing existing web standards** (such as HTTP, WebSockets, and RESTful APIs), which are already **well-established and widely supported**. By using these standards, IoT devices can communicate over the web, allowing for easier integration with other web-based services and applications.



To ensure devices can understand and exchange meaningful data, WoT emphasizes **semantic interoperability**, which involves defining common data models and vocabularies. This helps ensure that devices with different technical architectures can interpret each other's data in a standardized way. API can be used to allow devices to **expose functionalities in a web-friendly manner**. It provides a high-level interface that simplifies the interaction between devices and applications, making it easier to control and monitor IoT devices from different manufacturers. By adopting open web technologies and common standards, WoT can facilitate device integration across various industries, reducing the complexity and cost of implementing IoT solutions.

Consider a typical end-to-end IoT system, we call it "My Office", designed to collect data from various sensors and transmit it to the cloud for analysis and remote access:



At the core of the system is the **Gateway** (represented by a Raspberry Pi board), which acts as

a bridge between sensors and the internet. The gateway gathers data from a smart sensor (and Arduino board) using Bluetooth Low Energy (BLE) wireless communication. The smart sensor provides environmental data (temperature) and detects motion (PIR sensor). Once aggregated, the data is transmitted to the cloud, enabling remote access and control for end-users and applications. The system also supports device control, such as managing leds. This "My Office" example highlights a fundamental IoT architecture, showcasing the key components involved in collecting, transmitting, and utilizing sensor data.

#### 1.4.1 The Web as GUI

In the "My Office" IoT system, the web can serve as a **user-friendly interface** for interacting with sensors and devices. Imagine a web page displaying real-time data from the temperature sensor, the PIR motion sensor, or even a live video feed from a connected camera:

This web page, accessible from anywhere with an internet connection, enables users to monitor the environment, control devices, and analyze data seamlessly. Users can view current temperature readings, detect motion activity, and check sensor statuses. They can also remotely control devices, such as turning LEDs on or off, adjusting appliance settings, or automating actions like turning on lights when motion is detected. Additionally, the web interface offers tools for analyzing data trends, generating reports, and gaining insights into the system's performance. The interface is **dynamically generated**, reflecting real-time data and device status. This web-based approach offers several advantages:

- **Accessibility:** it can be accessed from any device with a browser, such as computers, smartphones, or tablets.
- **Ease of use:** the familiar web interface ensures intuitive navigation
- **Remote control:** users can manage and monitor the system from anywhere
- **Data visualization:** information is presented clearly, using charts and graphs for better insights.

By leveraging the web, the "My Office" IoT system becomes more accessible, interactive, and powerful, offering users a seamless way to monitor and control their environment.

### 1.4.2 The Web as an API

While we often interact with the web through browsers, designed for human consumption, it can also serve as a powerful **Application Programming Interface (API)**. An API defines a set of rules and tools that allow different software applications to communicate and interact with each other. In the context of the "My Office" IoT system, this means enabling other software applications to interact with the sensors and devices. Imagine a weather forecasting application that wants to access temperature data from the "My Office" system. Instead of a human browsing the web page to see the temperature, the weather application can directly request this data via an API. This involves sending HTTP requests to specific URLs, similar to how a web browser would, but with the key difference being that the application is requesting data in a machine-readable format like **JSON (JavaScript Object Notation)**, which is a lightweight data-interchange format that uses human-readable text to represent structured data. It's derived from JavaScript, but it's language-independent and can be used by various programming languages. This machine-to-machine communication allows for seamless integration between different systems. The "My Office" system could expose its data and functionality through a well-defined API, enabling other applications to:

- **Retrieve sensor data:** access real-time temperature readings, humidity levels, motion detection events, and other sensor data.
- **Control devices:** remotely turn on/off lights, adjust thermostat settings, and trigger other actions based on sensor readings or user commands.
- **Build custom applications:** developers can create new applications that leverage the "My Office" system's data and functionality, such as home automation systems, energy management tools, or security monitoring systems.

The web browser is a tool for humans to surf the web, we need a different tool to interact with APIs. **Postman** provides a user-friendly interface for sending HTTP requests, inspecting responses, and testing API interactions. This allows developers to quickly prototype and test their applications before deploying them. The role of web as an API extends its functionality beyond human interaction, it enables seamless communication between different software systems, opening up a world of possibilities for integration and automation. As an example, with Postman, we can send a GET request to the "My Office" system to retrieve the list of available sensors or a GET on a specific sensor to get its current temperature reading:

### 1.4.3 A GUI for the "My Office" IoT system

Let's put on our developer hat and start building an application that interacts with the "My Office" WoT device. It is not important to grasp all the details of the following source code, but

only to get a general idea. We create a simple web page that retrieves temperature data from "My Office" and displays it to the user (see *01.polling-sensor.html*). In general, we have a page described in HTML and made interactive with **JavaScript** code:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Polling Temperature</title>
  </head>
  <body>
    <h1>Current Temperature</h1>
    <h2 id="temp"></h2>
    <script>
      ....
    </script>
  </body>
</html>
```

The JavaScript code fetches the temperature data from the "My Office" system using an HTTP GET request and updates the web page with the latest reading:

```
// Ensures the code executes only after the HTML document is fully loaded
document.addEventListener("DOMContentLoaded", () => {

  // Use an asynchronous call to fetch data without reloading the page
  // The endpoint is expected to return a JSON object with temperature data.
  function fetchTemperature() {

    // Use the Fetch API to make an HTTP request to an endpoint,
    // specifying the URL and headers to accept JSON data
    fetch('https://makers.diten.unige.it/iot/sensors/temperature',
      {headers: {'Accept': 'application/json'}})

    // Once the response arrives, the callback functions convert the data in JSON,
    .then(response => response.json())

    // Then update the temperature value displayed on the webpage and set a timeout to
    // the temperature again
    .then(data => {
```

```

    document.getElementById('temp').textContent = `${data.value} ${data.unit}`;
    setTimeout(fetchTemperature, 1000);
}

// If an error occurs, log the error to the console
.catch(error => console.error('Error:', error));
}

// Call the fetchTemperature function to start fetching temperature data
fetchTemperature();
});

```

This simple example demonstrates how the web can serve as a powerful interface for interacting with IoT devices, enabling real-time data visualization and user engagement. By leveraging the potential and wide availability of web technologies, we can effortlessly develop rich and highly sophisticated GUI that connect seamlessly with IoT systems. For example we can visualize real-time data exploiting the Google Charts library (see *02.polling-sensor-chart.html*):

```

// Loads the Google Charts library and specifies the callback function
// drawChart() to be called once the library is loaded
google.charts.load('current', { packages: ['corechart'] });
google.charts.setOnLoadCallback(drawChart);

function drawChart() {
    // Define the maximum number of data points to display on the chart
    const maxDataPoints = 10;

    // Create a new LineChart object and specify the HTML element to render the chart
    const chart = new
        google.visualization.LineChart(document.getElementById('chart'));

    // Initialize a table for storing chart data. First row defines the column
    // headers:
    // Time (x-axis) and Temperature (y-axis)
    // Second row adds an initial data point with the current time and a placeholder
    // value
    const data = google.visualization.arrayToDataTable([
        ['Time', 'Temperature'],
        [new Date().toLocaleTimeString(), 0]
    ]);
}

```

```

// Configure chart properties: title, how to smooths the line between points,
// → animation
// when the chart updates and the position of the legend
const options = {
    title: 'Temperature',
    curveType: 'function',
    animation: { duration: 1000, easing: 'in' },
    legend: { position: 'bottom' }
};

// Add a new data point to the chart. If the number of rows exceeds the maximum,
// remove the oldest data point before adding the new one
function addDataPoint(value) {
    if (data.getNumberOfRows() > maxDataPoints)
        data.removeRow(0);
    data.addRow([new Date().toLocaleTimeString(), value]);
    chart.draw(data, options);
}

// Fetch temperature data from the server and add it to the chart
function fetchTemperature() {
    fetch('https://makers.diten.unige.it/iot/sensors/temperature',
        {headers: {'Accept': 'application/json'}})
        .then(response => response.json())
        .then(result => {
            addDataPoint(result.value);
            setTimeout(fetchTemperature, 2000);
        })
        .catch(error => console.error('Error fetching data:', error));
}

// Start fetching temperature data to update the chart
fetchTemperature();

```

While polling the sensor in the previous code works adequately, it introduces **inefficiency**: rather than repeatedly fetching the temperature from the device every few seconds, it would be far more efficient if the system were **notified** only when a temperature change occurs, and only when it happens. This highlights one of the major impedance mismatches between the **Web traditional request-response model** and the **event-driven architecture that IoT applications** often rely on. The ideal solution to address this challenge is the use of **WebSockets**, which allows for real-time, bi-directional communication between the server and client, enabling the

system to receive updates only when a change occurs (see *03.websockets-temp-graph.html*):

```
// Create a WebSocket connection to the temperature sensor.
// Note the wss:// protocol, used for WebSocket connections, which allows for
// persistent, two-way communication between the client and the server.
var socket = new WebSocket('wss://makers.diten.unige.it/iot/sensors/temperature');

// The "onmessage" event handler is registered to listen for incoming messages
// from the WebSocket. When a message is received, the data is parsed from JSON
// and added to the chart data
socket.onmessage = function (event) {
    const cleanedData = event.data.replace(/\"/g, '').trim();
    const result = parseFloat(cleanedData);
    addDataPoint(result);
};

// The "onerror" event handler is registered to log any errors that occur during
// the WebSocket connection. If an error occurs, the message 'WebSocket error!' is
// logged
socket.onerror = function (error) {
    console.error('WebSocket error:', error);
};
```

This setup efficiently addresses the inefficiencies of polling, providing a more scalable and real-time solution. The web page dynamically updates the temperature chart whenever new data is received, ensuring that users have access to the latest information without unnecessary delays. By leveraging WebSockets, we can create responsive, real-time applications that seamlessly interact with IoT devices.

We explored some ways to read sensor data, but what about **writing data to a device**? For example, we may want to send a command to control a device, such as toggling an LED on or off. This introduces the need to send instructions or data to the device, not just read it. One way to do this is by using a simple **HTML form** that sends a **POST request** to the device (see *04.actuator-form.html*):

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Send Message to Actuator</title>
```

```

</head>
<body>
  <h1>Send Message to Actuator</h1>

  <!-- The form uses the POST method to send data directly to the device -->
  <form action="https://makers.diten.unige.it/iot/actuators/leds/1" method="post">
    <label for="message">Enter a message for the Led 1</label>
    <input type="text" id="message" name="value" placeholder="{ 'value': false }"
      required>
    <button type="submit">Send to Pi</button>
  </form>
</body>
</html>

```

While this method works well in principle, there's a small issue: browsers do not natively submit a JSON payload body in a POST request. Instead, due to legacy reasons, browsers use a format called "application/x-www-form-urlencoded" when submitting form data. This format is fine for sending simple key-value pairs, but it isn't ideal for sending structured data like JSON objects. Moreover, HTML forms not supports PUT method, only support GET and POST methods. To handle this, we can implement a JavaScript function that overrides the default behavior of the form submission. This function gathers the form data, convert it into a JSON object, and then send it as the body of the POST request to the API. By doing so, we ensure the data is sent in the desired JSON format. Additionally, the function can provide feedback by displaying the result of the operation upon a successful request (refer to *05.actuator-ajax-json.html*):

```

// Add an event listener to the form to handle the 'submit' event
document.getElementById('message-form').addEventListener('submit', function (event) {

  // Prevent the form's default behavior, which is to submit and reload the page
  event.preventDefault();

  // Get the value entered in the input field with id 'message'
  const message = document.getElementById('message').value;

  // Use the Fetch API to send a POST request to the server
  fetch('https://makers.diten.unige.it/iot/actuators/leds/1', {
    method: 'PUT',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ value: message }),
  })

```

```
// Check if the response was successful
.then(response => {
  console.log(response);
})

// Handle the successful response (e.g., log the server's response)
.then(data => {
  console.log('Message sent successfully:', data);
})

// Handle any errors that occurred during the fetch
.catch(error => {
  console.error('Error sending message:', error);
});
});
```

Abbiamo appena visto come il web possa essere utilizzato per creare un'interfaccia verso i dispositivi, permettendo di interagire con essi in modo semplice ed efficace. Tuttavia, questo non è l'obiettivo principale del corso. Nei prossimi moduli, ci concentreremo su quello che è il nostro scopo principale: esplorare come le stesse tecnologie web possano essere applicate non solo per consumare le API, ma anche per implementarle. In questo modo, acquisiremo una comprensione approfondita dei principi e delle pratiche necessarie per creare API scalabili, universali e applicabili al contesto IoT, utilizzando strumenti e metodologie ampiamente diffusi, non solo HTTP, JavaScript e JSON, ma anche altre tecnologie web.

## 1.5 Semantic Gap

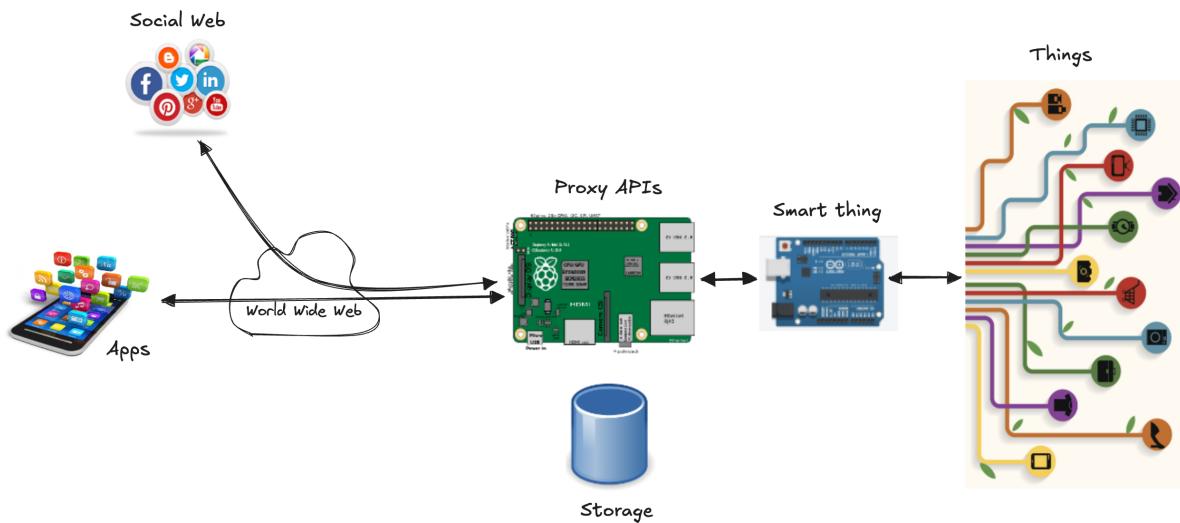
We can consider the challenge of **understanding and interpreting** the meaning of data exchanged between devices and applications. For example, when working with JSON objects representing sensors, we assumed that we already know **what each field means and how to use them**. However, what happens if the only information we have about a device is its internet endpoint and no further details? This lack of shared understanding highlights the **semantic gap problem**. It begins at the **network level** with device discovery: how can an application locate all the devices on a network and determine their endpoints? Without this initial step, communication cannot even begin. Once devices are discovered, the next challenge is for the application to **understand” the devices**, specifically, to determine what sensors or actuators the device offers, the data formats it uses, and the meaning of its properties and fields. One (partial) solution to address this issue is to exploit a fundamental feature of the web: **links**. Just as links in web pages provide a way to navigate and discover content, they can also help applications

explore and understand devices. By embedding links and metadata within APIs, devices can guide applications to discover their capabilities and interpret the data they provide. This approach leverages existing web mechanisms to reduce the semantic gap, enabling more seamless and meaningful interaction between devices and applications.

Summarizing, we interact with an embedded device, potentially located on the other side of the world (like in "My Office"), through a simple web page. This page regularly fetches data from a connected sensor, displaying it on a graph, a remarkable capability achieved with just a few lines of HTML and JavaScript. Now, imagine if our Raspberry Pi didn't provide its data via HTTP, JSON, or WebSockets, but instead relied on a machine-to-machine IoT protocol like ZigBee. In such a case, direct communication from a browser would be impossible. We would be forced to write our application in a lower-level language, like C or Java, and forego web technologies such as URLs, HTML, CSS, and JavaScript. This is the essence of the **Web of Things (WoT)** idea: creating APIs for devices that are universally accessible. By bridging the gap between IoT and the web, WoT leverages widespread web development tools and practices, enabling broader innovation and accessibility in building connected solutions.

## 1.6 Landscape

The following image depicts the system architecture that covers the overall scopes of the course:



At the heart of the architecture is the "World Wide Web," the network of interconnected websites and services where web applications and social media platforms reside. A "Mobile Apps" on a device enable users to access and interact with these web-based services. A "Proxy API" serve as intermediary, translating data between different systems and protocols, effectively bridging the

gap between the web and physical devices. The "Smart Thing" is a physical device equipped with sensors and actuators, allowing them to sense their environment and take action. The "Storage" component refers to a database, which manages and stores data generated by the system, such as sensor readings or user interactions. This system supports diverse interactions, such as a smart thermostat connected to a social media platform, enabling remote control and energy usage notifications, or a sensor-equipped device collecting data and transmitting it to a web application for analysis and visualization. The diagram illustrates the growing phenomenon of the WoT, where everyday objects are becoming increasingly connected to the internet, opening up new opportunities for automation, remote management, and data-driven insights.

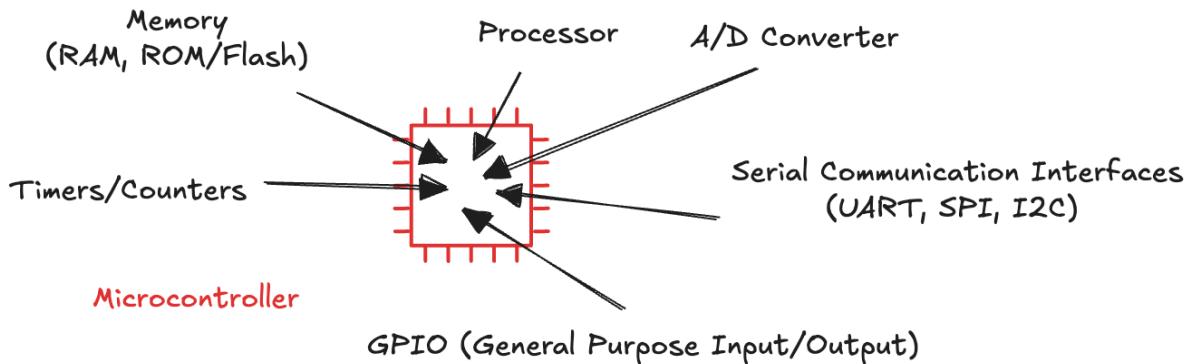
## 1.7 Hands-on Activity

1 - Experiment with the several URLs provided by the API <http://devices.webofthings.io> using Postman. Examine how these URLs are structured, how they differ from one another, and explore the device's endpoints to understand the data provided by each sensor, including its format and structure.

2 - Consider the electronic devices around you: like the appliances in your kitchen, your TV or sound system, the ordering system at a café, or the train notification board at the station and so on. Imagine a world where the services and data offered by these devices follow a unified structure: consistent endpoints, names, and content formats. Using the JSON structure we've introduced, try to design a similar system for these devices. Map out the names they might use and the JSON objects they could return to represent their data and services.

## 2 Edge Devices

In the context of fog computing, an **edge device** refers to a computing unit located at or near the point where data is generated. This close proximity enables **faster data processing** and **minimizes latency**, which is crucial for applications requiring real-time responses. The core component within edge devices is the **microcontroller** a compact, specialized integrated circuit designed to handle specific tasks in embedded systems. Microcontrollers integrate a processor core, memory, and input/output peripherals on a single chip, making them ideal for lightweight, task-focused operations:



Microcontrollers are often combined with **development boards** to simplify prototyping, testing, and developing embedded systems. While a standalone microcontroller chip provides the essential computing capabilities, it typically requires **additional components** and configurations to function effectively in real-world applications. Development boards address this need by providing a **ready-to-use platform** that integrates the microcontroller with other necessary elements, such as LEDs, buttons, voltage regulators, and sometimes even displays, sensors and communication modules (e.g., Wi-Fi or Bluetooth). These features allow developers to **focus on the application logic** rather than building basic hardware. Moreover, development boards are often supported by **software development environments** and **libraries** that simplify coding and reduce development time. For example:

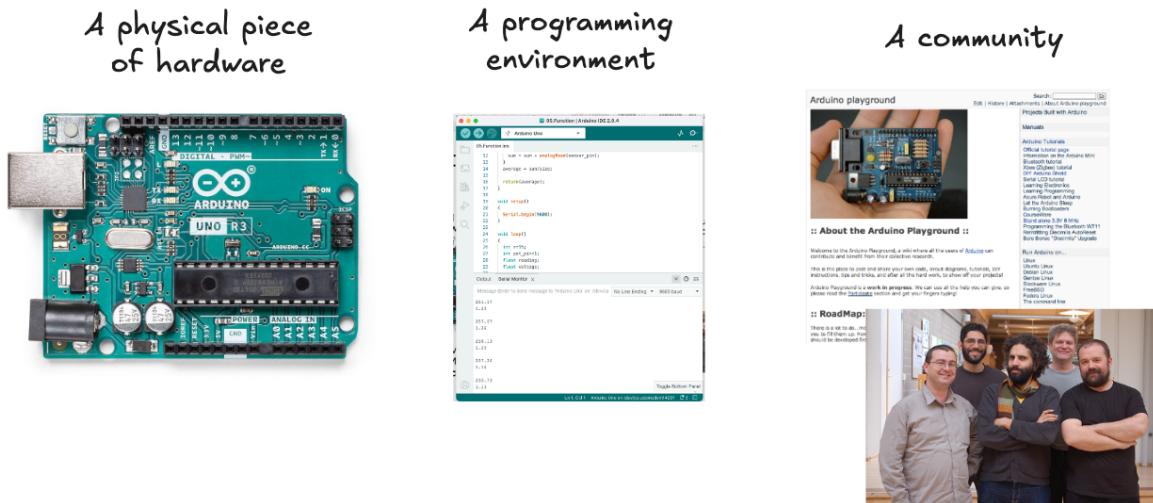
- Arduino boards come with a large collection of libraries for various hardware components,
- STM32 development boards are supported by software tools, easing the configuration of peripherals.

It's important to highlight the difference between a microcontroller and a microprocessor. A microcontroller is a self-contained system designed to perform specific tasks efficiently and repeatedly. The high level of integration reduces the need for external components, making microcontrollers compact, cost-effective, and energy-efficient. Microcontrollers excel at performing a **single dedicated function**, such as controlling a microwave, thermostat, or radio. They typically run one program stored in ROM, repeating it consistently without change. A microprocessor, on the other hand, is primarily a central processing unit (CPU) and lacks integrated components like memory and I/O peripherals. It relies on external modules such as RAM, ROM, storage, and I/O controllers to function, which makes it **more versatile** but also more complex. Microprocessors are typically used in systems requiring flexibility and the ability to handle large, multifaceted programs, such as personal computers, servers, or gaming consoles. The distinction between microcontrollers and microprocessors is becoming less clear, especially with the advent of advanced processors like ARM-based systems. Many ARM processors (e.g., 32-bit Cortex-M series) combine features of both, offering the simplicity of microcontrollers with the

power of microprocessors. These hybrid systems are used in applications that demand both real-time processing and computational power, such as smartphones.

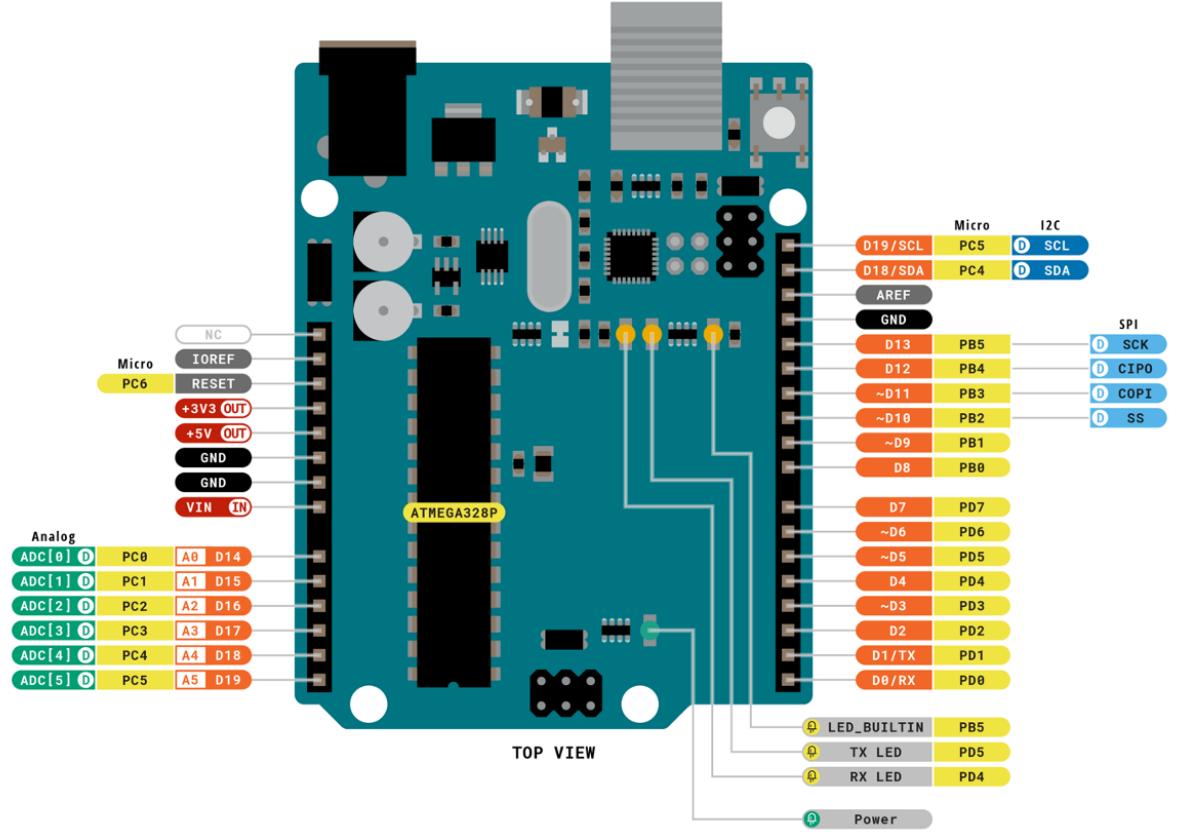
## 2.1 Arduino

Arduino is an **open-source electronics platform** designed to provide accessible and easy-to-use hardware and software for creating electronic projects. It was introduced in 2005 at the Interaction Design Institute in Italy, where it was conceived as an affordable and user-friendly tool for students exploring electronics. Since its inception, Arduino has become a widely recognized platform, empowering individuals to prototype and build innovative devices.

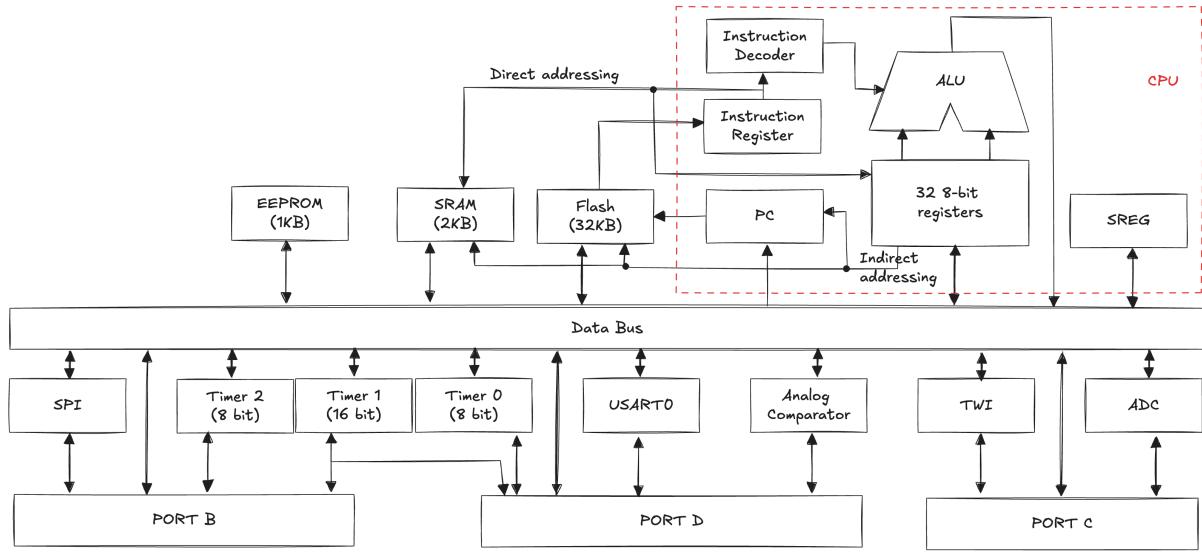


At the core of Arduino is its hardware, a compact printed circuit board that houses a **microcontroller**, alongside essential components to support its operation. The platform also includes **intuitive software**. The Arduino IDE allows users to write, compile, and upload code to the board effortlessly. The programming language, based on C/C++, is straightforward and approachable, even for beginners. A cornerstone of Arduino's success is its **community**, consisting of makers, hobbyists, students, and professionals worldwide. The platform open-source nature encourages collaboration and knowledge sharing, leading to a rich repository of projects, tutorials, and libraries. This openness reflects Arduino philosophy: to make electronics accessible to everyone.

The following image depicts the layout of an **Arduino Uno board** (the simple and basic board) with pins and components:



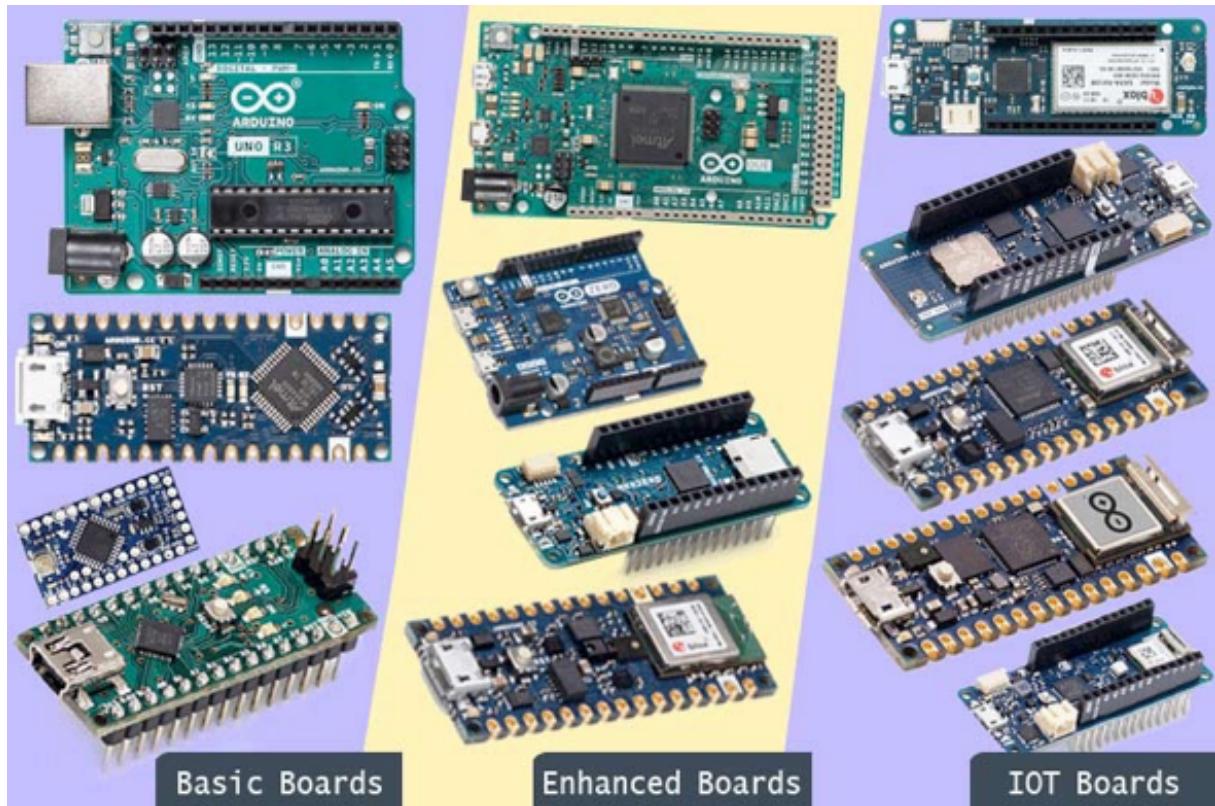
In general, before we can write code for our edge device, we need to understand how our specific microcontroller is structured and functions. This is typically referred to as its architecture. The schematic of the Arduino shows the **ATMEL ATmega328P microcontroller** at the heart of the development board, which handles the execution of programs (or "sketches") uploaded to it. It is a low-power CMOS 8-bit microcontroller based on the Harvard architecture, with physically separate storage and buses for program and data. Note that this is in contrast to the von Neumann architecture which operates with a single storage structure to hold both program and data. While one instruction is being executed, the next instruction is pre-fetched from the program memory. This concept enables instructions to be executed in every clock cycle. A simplified block diagram containing the most pertinent components is the following:



The ATmega328P provides the following significant features. Some memory modules to store programs (32 KBytes flash), to hold run-time variables (2 KBytes SRAM) and to store any data that programs wish to retain after power is cycled (1 KBytes of EEPROM). The microcontroller also includes several general purpose input/output (GPIO) lines: 4 **digital pins** (D0 - D13), some of which can also provide PWM (Pulse Width Modulation) output and 6 **analog input pins** (A0 - A5) for reading analog signals from sensors or other devices. Moreover it has 32 general purpose working registers, three timer/counters with compare modes, internal and external interrupts, an USART, a 2-wire Serial Interface (TWI) port, an SPI serial port, and a 6-channel 10-bit ADC. The 32 8-bit general purpose registers are directly connected to the ALU. Within a single clock cycle, arithmetic operations between general purpose registers or between a register and an immediate are executed. The ALU operations are divided into the three main categories arithmetic, logical, and bit-functions. After an arithmetic operation, the status register (SREG) is updated to reflect information about the result of the operation. Six of the 32 registers can be used as three 16-bit indirect address register pointers for data space addressing, enabling efficient address calculations. One of the these address pointers can also be used as an address pointer for look up tables in flash program memory. Program flow is provided by conditional and unconditional jump and call instructions, able to directly address the whole address space. Most instructions have a single 16-bit word format. Every program memory address contains a 16- or 32-bit instruction. During interrupts and subroutine calls, the return address Program Counter (PC) is stored on the stack. The stack is effectively allocated in the general data SRAM, and, consequently, the stack size is only limited by the total SRAM size and the usage of the SRAM. All user programs must initialize the SP in the reset routine before subroutines or interrupts are executed. The Stack Pointer (SP) is read/write accessible in the I/O space. The data SRAM can easily be accessed through the five different addressing modes

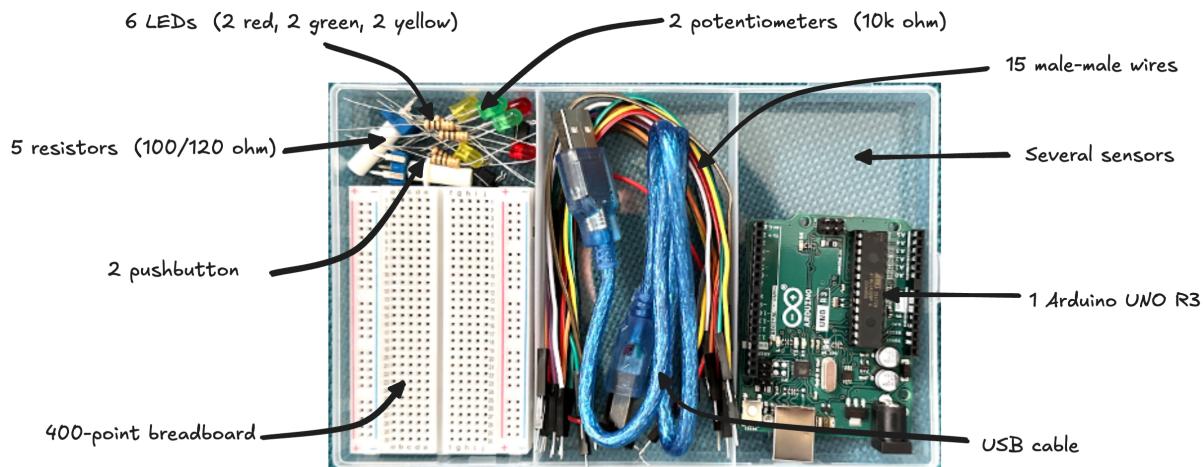
supported in the architecture. All of the peripheral devices are controlled via sets of specific registers, each of which is connected to the 8-bit data bus. Thus, a program interacts with each peripheral by accessing various memory-mapped registers (via C pointers). Additionally, each peripheral device accesses the off-chip pins via one of the three available ports B, C or D. Each port is configurable via memory-mapped registers to allow different functions access to external pins. The flash memory allows the program memory to be changed via either a SPI serial interface, a conventional non-volatile memory programmer, or an on-chip boot loader running on the processor core. The Arduino boot loader uses the USART configured for RS-232 via the USB-to-serial converter chip on the development board. Software in the boot flash section will continue to run while the application flash section is updated. Finally, the board provides also some **pwer pins** (+3.3V, +5V and GND to provide regulated power for external components), some **status leds** to show the power and activity (including TX and RX for data transmission or reception) and a **reset button** to restart the microcontroller program manually.

Arduino boards come in various models, each tailored to different needs, ranging from basic prototypes to advanced projects requiring wireless communication or extensive input/output capabilities:



### 2.1.1 Makers Kit

As student of "Makers", you can use the "Makers Kit", specifically designed to support you in coursework, particularly in introductory of embedded system design. It includes a carefully curated selection of components to enable hands-on learning and experimentation, making it ideal for both classroom activities and individual projects. Here's a detailed overview:

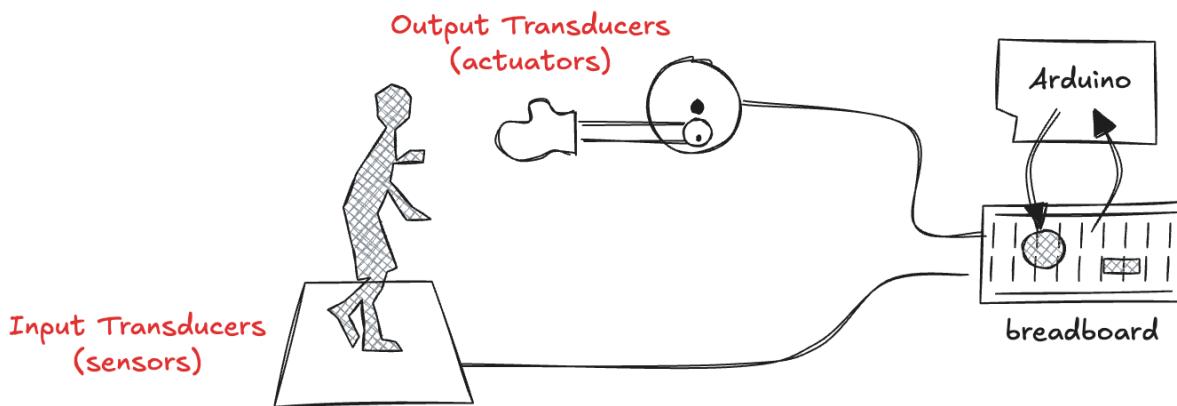


- Arduino UNO R3: a beginner-friendly microcontroller board widely used in educational settings for developing interactive projects
- Several Sensors: these may include basic sensors like temperature, light, or motion detectors, allowing students to explore data collection and environmental interaction
- 6 LEDs (2 red, 2 green, 2 yellow): used for visual feedback in projects, such as signaling or debugging
- 2 Potentiometers (10k ohm): for adjusting values like brightness or resistance in a circuit
- 400-Point Breadboard: a versatile platform for assembling circuits without the need for soldering
- 15 Male-Male Wires: essential for making connections on the breadboard or to the Arduino UNO R3
- 2 Pushbuttons: commonly used for user input in circuits, such as turning on/off devices or triggering actions
- 5 Resistors (100/120 ohm): protect LEDs or other components by limiting the current in the circuit
- USB Cable: enables programming of the Arduino UNO R3 and provides power to it during operation

## 2.2 Programming

Standard programming typically involves developing software for general-purpose computers that have **abundant resources** of processing power, memory, and storage. These systems can handle complex computations and multitasking, where performance, usability, and maintainability are the primary concerns. In standard programming, developers use high-level programming languages and powerful development environments and the software is usually run on an operating system which provides features like memory management, multitasking, and device drivers. On the other hand, embedded programming is focused on writing software for hardware with **limited resources** and should run on a bare-metal setup (without an operating system) or with lightweight operating systems. This requires careful management of hardware resources. Debugging is also more challenging because tools for monitoring may be limited, and developers often rely on specialized equipment. While standard programming may involve developing applications with complex user interfaces (UIs) and handling large datasets, embedded systems typically have minimal or no UIs. Instead, interaction is usually through simple components like LEDs, buttons, or small displays. In some cases, communication with other systems may occur via network protocols or serial communication. The key difference lies in how embedded programming prioritizes resource constraints, real-time processing, and direct hardware interfacing, while standard programming focuses on performance and user experience for general-purpose computing.

In combination with sensors and actuators, the programming style of a microcontroller can be harnessed to implement the concept of **physical computing**: an interactive system that integrates hardware and software to enable devices to sense and respond to the physical environment:



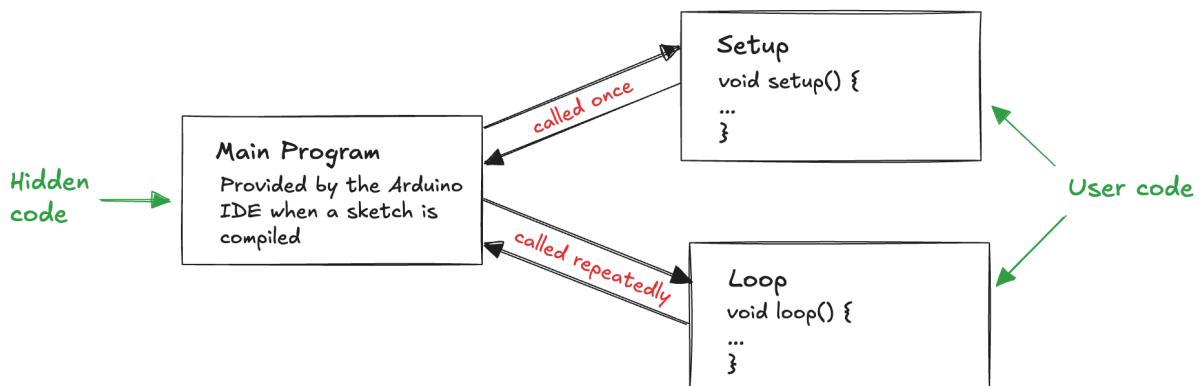
From the perspective of the microcontroller, **inputs** refer to signals or information that are received by the board. These could come from buttons, switches, light sensors, flex sensors, humidity sensors, temperature sensors, and more. **Outputs**, on the other hand, are signals that

leave the board, driving actions such as lighting up LEDs, powering DC motors, controlling servo motors, triggering piezo buzzers, or changing the color of RGB LEDs. Almost every physical computing system incorporates some form of output to respond to the inputs it receives, making interaction with the physical world possible.

To develop a physical computing system, the first step is to **design the circuit**. Begin by determining the electrical requirements of the sensors or actuators we need to use. This includes understanding the voltage, current, and power requirements of each component. Next, identify the analog inputs, such as sensors that provide variable signals (e.g., temperature, light, or humidity sensors). These inputs will typically be connected to the analog pins of the microcontroller. Similarly, identify the digital inputs and outputs, like buttons or switches that act as inputs, and devices like LEDs or motors that serve as outputs. It's important to carefully plan how each component will connect to the microcontroller, ensuring that each part of the circuit is properly powered and grounded.

Once the circuit is designed, we can move on to **writing the code**. It's best to build the code incrementally. Start by getting the simplest piece of functionality working first, such as reading an input or turning on an LED. Once the basic operation works, we can begin to add complexity, step by step, testing our system at each stage to ensure it functions as expected. Throughout the development process, be sure to save and back up our work frequently to avoid losing progress.

In the case of Arduino, the core of a program consists of two C functions: **setup()** and **loop()**. The **setup()** function runs once when the program starts and is used to initialize settings like pin modes and serial communication. The **loop()** function runs continuously after **setup()** and contains the main logic of the program, repeating over and over:



### 2.2.1 Controlling input and output

Controlling pins is fundamental and the **pinMode()** function is used to configure a pin behavior, specifying whether it acts as an input (to read data) or an output (to send data). Once a pin is set as an output, the **digitalWrite()** function allows us to set the pin to **HIGH** (providing +5V, which can turn on a LED or activate a device) or **LOW** (0V, turning a device off). Conversely, if a pin is configured as an input, the **digitalRead()** function can detect its state, determining whether it is receiving a HIGH or LOW signal. For analog operations, **analogRead()** reads the voltage from an analog pin as a value ranging from 0 (0V) to 1023 (5V), enabling the measurement of varying signals like those from a potentiometer or a sensor. To simulate an analog output, the **analogWrite()** function generates a **pulse-width modulation (PWM) signal** (which alternates between HIGH and LOW rapidly, creating the illusion of varying voltage levels. This is especially useful for tasks like dimming leds or controlling motor speed.

### 2.2.2 Timing

Timing functions are essential for controlling the flow of a program. The **delay()** function pauses the execution of the program for a specified number of milliseconds, making it useful for simple timing tasks, such as blinking a led or waiting between actions. However, because **delay()** blocks all other code execution during the pause, it can limit the responsiveness of your program. For more precise and non-blocking timing, the **millis()** function is a better alternative. It returns the number of milliseconds that have elapsed since the program started running. By comparing the current value of **millis()** with a previously recorded value, we can track elapsed time while allowing the rest of code to run concurrently. This makes **millis()** ideal for implementing non-blocking delays.

### 2.2.3 Hello World!

The first Arduino sketch is typically a simple program called "Blink," often referred to as the **hello world** of physical computing. The program makes an onboard led blink, demonstrating the basics of controlling output (see *01.Blink.ino* sketch):

```
// On most Arduino boards, a led is built into pin 13
const int ledPin = 13;

// The setup function runs once when the board starts or resets
void setup() {
    // Configure the LED pin as an output
```

```

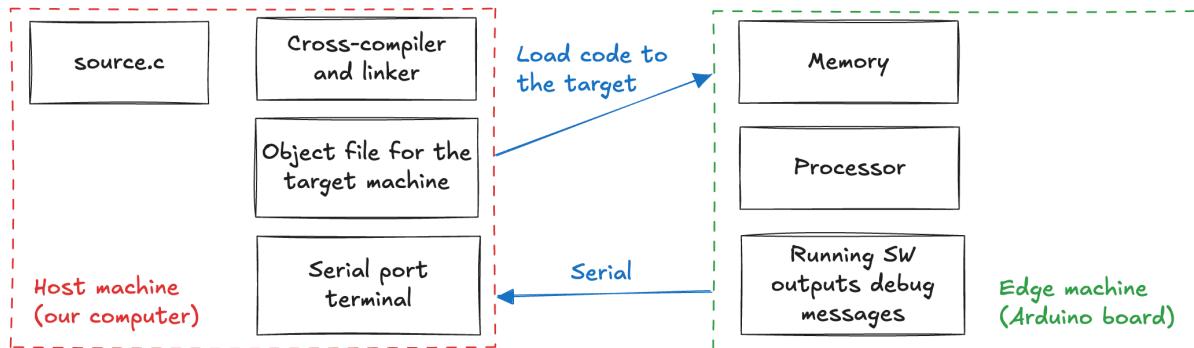
pinMode(ledPin, OUTPUT);
}

// The loop function runs continuously after setup
void loop() {
    // Turn the LED on (set the pin to HIGH voltage)
    digitalWrite(ledPin, HIGH);
    // Wait for 1 second.
    delay(1000);

    // Turn the LED off (set the pin to LOW voltage)
    digitalWrite(ledPin, LOW);
    // Wait for 1 second.
    delay(1000);
}

```

Embedded systems are special, offering special challenges to developers. The system's resources are very limited, making it impossible to perform standard code compilation directly on the device. This is because there is no compiler or linker available to run on these devices. Therefore, **cross-compilation** is required. This means using a different machine (the **host**) to compile the source code into a machine language that is executable on the embedded system (the **target**).



In the case of Arduino, we use our computer as the host machine, the Arduino IDE serves as the cross-compiler, and the compiled code is then transferred to the board via a USB connection. In essence, cross-compilation allows us to leverage the computational power of our computer to generate efficient code for our resource-constrained embedded system. When compiled, the program uses like a thousand bytes of the **available memory** to generate the sequence of instructions needed for blinking. With approximately several tens of thousands bytes of memory on most boards, some space remains unused for more complex programs. This simple exercise introduces key concepts like compiling, uploading, and basic hardware control.

## 2.2.4 Data type, Selection, Repetition and Function

In Arduino programming, we use **standard C data types** to define variables and store values. The most common data types include:

- **Integers**: used to store whole numbers
- **FLOATS**: used to store decimal numbers with floating-point precision

The following examples demonstrates how integer arithmetic works, particularly how truncation (the removal of the decimal portion of a number) occurs when using integer division (see *02.Integers.ino* sketch) and the well-known issue of round-off error that can occur with floating-point arithmetic (see *03.Floatingpoint.ino* sketch). Arduino supports **serial communication** for debugging. We can initialize serial communication using **Serial.begin()**, and functions like **Serial.print()** to send data to the serial monitor:

```
void setup() {  
    int i,j;  
  
    // initialize serial communication  
    Serial.begin(9600);  
  
    // wait for user to open the serial monitor  
    delay(3500);  
  
    // 2/3 is performed using integer division, which results in 0  
    // because 2 and 3 are both integers and the result of their division is truncated  
    // (i.e., the fractional part is discarded).  
    // This result is then multiplied by 4, which gives i = 0.  
    i = (2/3)*4;  
  
    // The variable j is then assigned the value of i + 2, which results in j = 2.  
    j = i + 2;  
  
    // Print the values of i and j to the serial monitor  
    Serial.println("First test");  
    Serial.print(i);  
    Serial.print(" ");  
    Serial.println(j);  
  
    // In this case, 2.0 and 3.0 are floating-point numbers, so the division result  
    // is a floating-point number, yielding approximately 0.6667  
    // This value is then multiplied by 4.0, resulting in i = 2.6667
```

```
// However, because i is declared as an int, it will be truncated when stored, so i
//   = 2
i = (2.0/3.0)*4.0;

// The variable j is then assigned the value of i + 2, which gives j = 4
j = i + 2;

Serial.println("Second test");
Serial.print(i);
Serial.print(" ");
Serial.println(j);
}

void setup() {
    // define the variables
    float w,x,y,z;

    Serial.begin(9600);
    delay(2500);

    // 4.0 / 3.0 gives an approximation of 1.333 ... , which is not exactly 4/3,
    // as floating-point numbers cannot always represent exact decimal values.
    w = 4.0/3.0;

    // w - 1 results in 0.333 ...
    x = w - 1;

    // 3 * x results in 0.999 ... , which should theoretically equal 1,
    // but due to the floating-point approximation, it is just shy of 1.
    y = 3*x;

    // 1 - y results in a small value close to zero, which demonstrates
    // the rounding error introduced in the earlier steps
    z = 1 - y;

    // Print the values of w, x, y, and z to the serial monitor
    // using two-parameter version of Serial.print(), the second parameter
    // specifies the number of digits in value sent to the Serial Monitor

    Serial.println("\nFloating point arithmetic test");
    Serial.println(w,16);
    Serial.println(x,8);
```

```
Serial.println(y,8);
Serial.println(z,8);

// Prints z multiplied by a big number to better visualize the effect of
// the round-off error when scaling up the small floating-point value
Serial.println(z*1.0e7,8);
}
```

Arduino also supports **common control structures** found in C language:

- **Selection (if statement)**: used to make decisions based on conditions.
- **Repetition (loops)**:
  - **For loop**: a loop that repeats a block of code a specified number of times
  - **While loop**: a loop that continues to execute as long as a condition remains true.

The following example (see *04.Repetition.ino* sketch) demonstrates the basics of using a loop for a repetitive task:

```
void setup() {
  Serial.begin(9600);
}

void loop() {
  // define the variables
  int i;

  // Print the numbers from 0 to 9 using a for loop
  for (i=0; i<10; i++) {
    Serial.println(i);
    delay(100);
  }

  Serial.println("for loop over\n");
}
```

**Functions** are a powerful way to organize and reuse code. By encapsulating details of a task into a function, we can break our program into smaller, more manageable blocks. Well-written functions improve code readability and maintainability. As usual, functions can accept input parameters (e.g., values passed to the function to perform tasks) and optionally return an output (e.g., a value resulting from the function's operations).

The following example (see *05.Function.ino* sketch ) demonstrates how to use a function to read an analog sensor (like a potentiometer) multiple times and calculate the average value of those readings, in order to smooth out noise and getting a stable sensor reading:

```
// A function to calculates the average of a number of readings
// from a sensor connected to a specific pin
float average_reading(int sensor_pin, int size) {
    int i;
    float average;
    float sum;

    // Initialize the sum to zero
    sum = 0.0;

    // Read the sensor value 'size' times and add them to the sum
    for (i=1; i<=size; i++) {
        sum = sum + analogRead(sensor_pin);
    }

    // Calculate the average by dividing the sum by the
    // number of readings
    average = sum/size;

    // Return the average
    return(average);
}

void setup() {
    Serial.begin(9600);
}

void loop() {
    // Numer of readings
    int n=15;

    // Pin to which the potentiometer is connected
    int pot_pin=1;

    // The reading from the potentiometer
    float reading;

    // Call the average_reading function to calculate the average reading
    reading = average_reading(pot_pin,n);
```

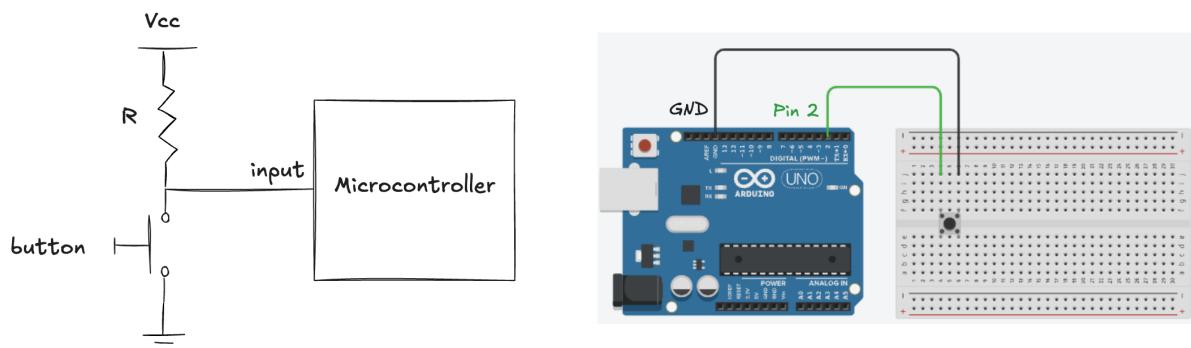
```
// Print the reading and voltage to the serial monitor
Serial.println(reading);
Serial.println();
}
```

## 2.2.5 Interrupt

An interrupt is a mechanism that allows a microcontroller to temporarily stop its current task to handle a more immediate task. It is an essential feature in embedded systems, enabling a program to **respond immediately** to certain events without constantly checking for them in the main loop. They are managed using the **attachInterrupt()** function, which links a specific pin to an **interrupt service routine (ISR)**, a special function that runs when the event occurs. When an interrupt occurs, the currently executing program is paused, the ISR is executed to handle the interrupt and after it is finished, the main program resumes from where it left off. For example, we can use interrupts to detect a button press, even if our main program is busy with other tasks. With **attachInterrupt()**, we specify the interrupt pin, the ISR to execute, and the **trigger mode**, that determines when the interrupt will fire:

- **RISING**: triggers when the pin transitions from LOW to HIGH
- **FALLING**: triggers when the pin transitions from HIGH to LOW
- **CHANGE**: triggers on any change in the pin's state (from HIGH to LOW or vice versa).

Interrupts are particularly useful for **handling time-sensitive events** or for tasks where **polling** (repeatedly checking the status of an event at regular intervals to see if it has occurred) would be inefficient. As an example, consider the problem of **handling a push-button input** to toggle an LED on and off like in the following schematic:



When the button is not pressed, the pin is **pulled HIGH (connected to +5V)** through a pull-up resistor (that limits the current flowing), when the button is pressed, it creates a connection

between the pin and ground, pulling the pin LOW. Many Arduino pins have **built-in internal pull-up resistors**. We need to configure the digital pin as an **INPUT\_PULLUP** in our code in order to enable this internal resistor. In a **polling-based approach**, we would need to continuously check the button state in the main loop, which could lead to missed button presses if the loop is busy with other tasks (see *06.NoInterrupt.ino* sketch):

```
// Define the pin with the LED (the one on the board)
int led_pin = 13;

// define the pin with the button
int button_pin = 2;

// Function to handle the button input and control the LED
void handle_button() {
    // Read the button and set the LED accordingly
    digitalWrite(led_pin, digitalRead(button_pin));
}

// Function to handle other tasks that take time
void handle_other_stuff() {
    // Simulate a delay for a long-running task
    delay(250);
}

void setup() {
    // Set LED pin as output
    pinMode(led_pin, OUTPUT);
    // Set button pin as input with internal pull-up resistor
    pinMode(button_pin, INPUT_PULLUP);
}

void loop() {
    // Check the button and update LED
    handle_button();

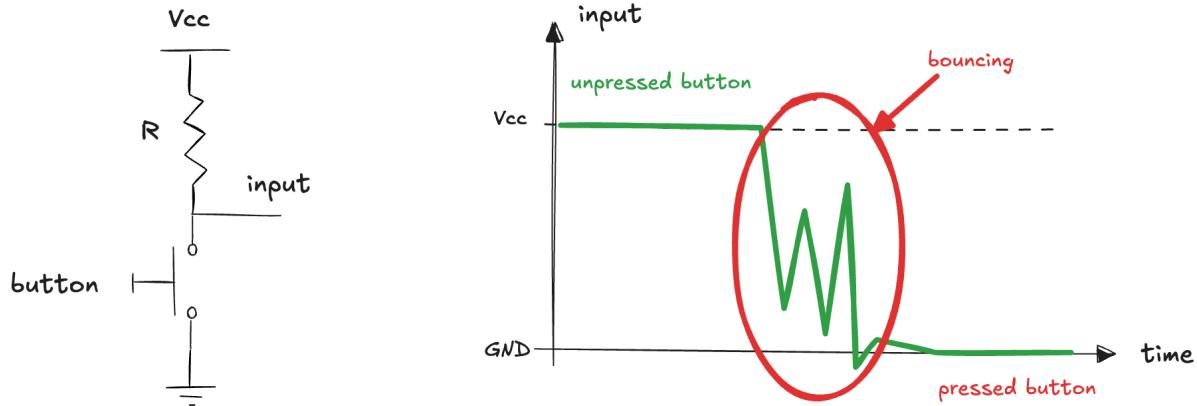
    // Perform other tasks (with delay)
    handle_other_stuff();
}
```

The button state is only checked periodically, and if the button is pressed while the program is inside the `handle_other_stuff()` function, the press might be missed. Additionally, the button input might not be handled quickly enough. We can break up the long tasks into smaller chunks

and handle the button press more frequently. Still, this approach can become inefficient as the time between button presses decreases. By using an interrupt, we can immediately respond to the button press, regardless of what the main loop is doing. The following example demonstrates how to use an interrupt to toggle an LED when a button is pressed (see *07.Interrupt.ino* sketch):

```
void setup() {  
    pinMode(LED, OUTPUT);  
    pinMode(SW, INPUT_PULLUP);  
  
    // Set up an interrupt on button_pin (which corresponds  
    // to INT0 on the Arduino Uno  
    attachInterrupt(INT0, handleSW, CHANGE);  
}  
  
// The loop function call only the handleOtherStuff function  
void loop() {  
    handleOtherStuff();  
}
```

The `attachInterrupt()` function is used to set up an interrupt on `button_pin` (pin 2, which corresponds to `INT0` on the Arduino Uno). The interrupt triggers when the state of the button changes (from `LOW` to `HIGH` or `HIGH` to `LOW`) by using the `CHANGE` mode. When the button state changes (i.e., when the button is pressed or released), the `handle_button()` function is called immediately, regardless of what the main loop is doing. In the `loop()`, the `handle_other_stuff()` function is called, which simulates performing other tasks (e.g., running a long process). This ensures that button presses are detected and processed instantly, even if the program is executing other tasks. However, there is a potential problem, the interrupt might be triggered multiple times for a single button press. This is known as **bouncing**, a common issue with mechanical switches. When a button is pressed, the contacts inside the switch can bounce back and forth before settling into a stable state:



To show the button bouncing problem, we can modify the code to show how the led can rapidly flicker when a button is pressed or released due to the mechanical bouncing (see *08.NoDebouncing.ino* sketch):

```
// Add a variable to count the number of times the button is pressed
// to observe the bouncing problem
volatile int count = 0;

void handle_button() {
    digitalWrite(led_pin, digitalRead(button_pin));

    // Increment the count each time the button is pressed
    count++;

    // Print the count to the serial monitor
    Serial.println(count);
}
```

Notice that the variable `count` is shared between an ISR and the `loop()` function, the program modify this variable in the ISR while the main program can read and act upon it. It is important to declare it as **volatile**, since this specification tells the compiler that the variable can be modified by an external source (the ISR in that case) and should not be optimized, the main program always has to fetch the latest value directly from memory, reflecting any changes made during the interrupt.

When dealing with the mechanical bouncing problem we can use either hardware or software techniques. The **hardware solution** involves the use of external electronic components, such as a resistor-capacitor RC filter. These components smooth out the signal from the button and eliminate oscillations. The primary advantage of this approach is that it offloads the task from the

microcontroller, freeing it for other computations. Additionally, it ensures consistent debouncing without using the microcontroller's processing resources. However, this method requires additional components, which can increase cost and complexity. Moreover, it is less flexible since modifying the timing requires physical changes to the circuit. The **software solution**, on the other hand, relies on programming techniques such as introducing a delay or using timing mechanisms to ignore button state changes until the bouncing has subsided. This approach does not require any extra hardware, making it cost-effective and easy to implement. Furthermore, it is highly flexible, as the debounce timing can be adjusted directly in the code. Despite its simplicity, the software solution has some drawbacks. It adds slight delays to the system's response time and consumes some of the microcontroller's processing cycles, which could affect performance in resource-constrained systems (see *09.Debouncing.ino* sketch):

```
// Timestamp of last valid button press
volatile unsigned long lastDebounceTime = 0;

// Debounce delay in milliseconds
const unsigned long debounceDelay = 50;

// Function to handle the button with debouncing
void handle_button() {
    // Check the time since the last valid button press
    unsigned long currentTime = millis();
    if (currentTime - lastDebounceTime > debounceDelay) {
        digitalWrite(led_pin, digitalRead(button_pin));

        // Print the count to the serial monitor
        count++;
        Serial.println(count);

        // Update the last debounce time
        lastDebounceTime = currentTime;
    }
}
```

In the `handle_button()` function, the time elapsed since the last button press is checked using `millis()`. If the elapsed time exceeds the debounce delay, the button press is considered valid and the led state is toggled only when a valid button press is detected.

## 2.3 Serial communication

To send and receive text and data, we can use the serial communication interface. This allows us to transmit information from the microcontroller to be any serial device. To initiate serial communication, the speed (baud rate) must be specified using the **Serial.begin()** function:

```
Serial.begin(9600);
```

Here, 9600 refers to the **baud rate**, which is the number of symbols transmitted per second. It's essential that both the sending (Arduino) and receiving (PC) sides use the same baud rate; otherwise, the output will be unreadable or completely absent.

### 2.3.1 Send and receive text

Using the serial interface, we can send text or data to be displayed. For instance:

```
Serial.print("The number is ");
Serial.println(number);
```

`Serial.print()` sends text or numbers without moving to a new line, `Serial.println()` sends text or numbers and then moves to the next line. We can also define the format for numerical values, such as decimal, hexadecimal, or binary. For example:

```
Serial.println(number, HEX); // Display the number in hexadecimal
Serial.println(number, BIN); // Display the number in binary
```

**To receive data on an Arduino from a computer**, for example for reacting to commands or data sent from an external source, we can use the same library. To check if data is available, we use the **Serial.available()** function. This returns the number of characters in the serial buffer. For example:

```
if (Serial.available() > 0) {
    // There is data to read
}
```

Once data is available, we can use **Serial.read()** to retrieve the next byte from the buffer. For example:

```
char receivedChar = Serial.read();
Serial.print("You sent: ");
Serial.println(receivedChar);
```

As an example, the following program (see *10.SerialReceive.ino*) controls the blink rate of a led based on numeric input received over the serial connection:

```
// The pin to which the LED is connected
const int led_pin = 13;

// Initial blink rate
int blink_rate = 100;

// Function to controls the LED by turning it on, waiting for the blink_rate duration,
// then turning it off for the same duration.
void blink() {
    digitalWrite(led_pin, HIGH);
    delay(blink_rate);
    digitalWrite(led_pin, LOW);
    delay(blink_rate);
}

void setup() {
    Serial.begin(9600);
    pinMode(ledPin, OUTPUT);
}

void loop() {
    // Check to see if at least one character is available
    if (Serial.available()) {

        // If data is available, reads a character
        char ch = Serial.read();

        // If the character is a numeric ASCII digit ('0' to '9'),
        // convert it to its numeric equivalent using (ch - '0')
        if(ch >= '0' && ch <= '9') {
            blink_rate = (ch - '0') * 100;
        }
    }
}
```

```
// Call the blink function to blink the LED  
blink();  
}
```

When receiving numbers with more than one digit, we need to accumulate characters until we encounter a non-numeric character (e.g., a space or newline). This allows us to construct the full number before processing it. For example (see):

```
// Variable to accumulate the digits of the number  
int value;  
  
void loop() {  
    if (Serial.available()) {  
        char ch = Serial.read();  
  
        // The accumulated number is updated as a decimal value by multiplying the  
        // existing value by 10 and adding the new digit  
        if(ch >= '0' && ch <= '9') {  
            value = (value * 10) + (ch - '0');  
        }  
  
        // When a newline character is received, the accumulated value is used to  
        // calculate the new blinkRate. The value variable is reset to 0  
        // to prepare for the next input.  
        else if (ch == 10) {  
            blink_rate = value * 100;  
            Serial.println(blink_rate);  
            value = 0;  
        }  
    }  
    blink();  
}
```

This process enables the Arduino to interpret and respond to commands or numerical data, making it possible to control its behavior from a computer or other serial device.

### 2.3.2 Send and receive multiple text fields

To transmit multiple pieces of data (*fields*) from an Arduino, such as sensor readings for temperature, humidity, and pressure, it is crucial to define the data clearly and format it in a structured

way. A common method is to use a **comma-separated format** with a unique "header" to indicate the start of the message and a consistent "delimiter" (e.g., commas) to separate the fields. The header must be distinct, ensuring it does not appear within any data fields or as the delimiter. This ensures proper parsing on the receiving side. Once the message is formatted, it can be transmitted using `Serial.print()` to send the data over the serial connection. For example, temperature, humidity, and pressure readings can be sent in a comma-separated format with 'H' as the header, see *11.CommaDelimitedOutput.ino* sketch:

```
void setup() {
    Serial.begin(9600);
}

void loop() {
    // Simulated sensor readings
    float temperature = 25.3;
    float humidity = 60.1;
    float pressure = 1013.2;

    // Send a comma-delimited message with a header
    Serial.print('H');
    Serial.print(",");
    Serial.print(temperature);
    Serial.print(",");
    Serial.print(humidity);
    Serial.print(",");
    Serial.print(pressure);
    Serial.print(",");
    Serial.println();

    // Wait before sending the next message
    delay(100);
}
```

From the receiving side, the data have to be parsed by reading the serial buffer and extracting the fields based on the delimiter. In order to implement an example of this, we need to write code also on the receiving side. As an example, we can exploit the **Processing**, a powerful tool to "talk" with Arduino, particularly for visualizing or processing incoming data. It is based on a simplified version of the Java programming language and provides a simple and intuitive interface for creating interactive graphics and animations. Like Arduino programming, it uses a `setup()` function to initialize the program and a `draw()` function to continuously update the display. Moreover, it is based on an event-driven model, where the program responds to user input

or other events (like data coming from the serial port). The following example demonstrates how to receive and parse the comma-separated data received by Arduino:

```
// imports the Serial class from the processing.serial library
import processing.serial.*;

// Serial port object
Serial myPort;

// Buffer to store received data
String received_data;

// Parsed data
float temperature = 0.0;
float humidity = 0.0;
float pressure = 0.0;

void setup() {
    // Set up the application window
    size(400, 200);

    // Connect to the right port (the one that the Arduino is connected to):
    println(" Connecting to → " + Serial.list()[2]);
    myPort = new Serial(this,Serial.list()[2], 9600);
}

void draw() {
    // Write the received data to the screen
    background(0);
    text("Temperature: " + temperature, 10, 50);
    text("Humidity: " + humidity, 10, 60);
    text("Pressure: " + pressure, 10, 70);
}

// This method is called whenever data is received from the serial port
void serialEvent(Serial p) {

    // Read data until newline character
    received_data = myPort.readStringUntil('\n');

    // If data is received, parse the data by splitting the
    // received string
    if (received_data != null) {
```

```

String[] fields = split(received_data, ',');

// check the header
if (fields[0].equals("H")) {
    temperature = float(fields[1]);
    humidity = float(fields[2]);
    pressure = float(fields[3]);
}
}

}
}

```

To receive a message containing multiple fields on Arduino, such as an identifier for a specific device (e.g., a motor or actuator) and a corresponding value (e.g., speed to set it to), it is necessary to parse the incoming serial data. The message should follow a structured format where each field is separated by a delimiter, such as a comma. The following example, first checks for incoming data using `Serial.available()` to determine if there is data to read. It then reads and parses the message using `Serial.read()`, collecting characters until the message is complete. Once the message is fully received, it is split into fields using the designated delimiter. After parsing the message, the identifier and value are extracted from the fields and processed to perform the corresponding task, such as setting the speed of the specified motor. The following example demonstrates how to receive a message containing multiple numeric fields separated by commas and store the values into an array. The expected format of the message is 12,345,678, where each number is separated by a comma (see `12.SerialReceiveMultipleFields.ino` sketch):

```

// number of expected fields
const int NUMBER_OF_FIELDS = 3;

// current field being received
int field_index = 0;

// Array to hold values
int values[NUMBER_OF_FIELDS];

void setup() {
    Serial.begin(9600);
}

void loop() {
    if( Serial.available() ) {
        char ch = Serial.read();

```

```

// Accumulate digits to build the value of a field
if(ch >= '0' && ch <= '9') {
    values[field_index] = (values[field_index] * 10) + (ch - '0');
}

// Comma is our separator, so move on to the next field
else if (ch == ',') {
    if(field_index < NUMBER_OF_FIELDS-1)
        field_index++;
}

// Any other character ends the acquisition of fields and
// we provide some feedback, then we set the field to zero
// to start collecting the new value
else {
    Serial.print(field_index +1);
    Serial.println(" fields received:");
    for(int i=0; i < field_index; i++) {
        Serial.println(values[i]);
        values[i] = 0;
    }
}

// Ready to start over
field_index = 0;
}
}

```

### 2.3.3 Binary data exchange

In addition to exchanging textual information (formatted as characters), it is also possible to exchange data directly in **binary format**. The following code demonstrates how to send binary data over a serial connection. The **lowByte()** and **highByte()** functions can split a 16-bit integer into two 8-bit segments (low and high bytes). These bytes are sent over the serial connection using **Serial.write()** function (see *13.SendBinary.ino* sketch):

```

// Declare a 16-bit integer variable
int value;

void setup() {

```

```

    Serial.begin(9600);
}

void loop() {
    // send a header character
    Serial.print('H');

    // Generate a random integer
    value = random(599);

    // Send the low and high bytes
    Serial.write(lowByte(value));
    Serial.write(highByte(value));

    delay(1000);
}

```

Binary formats **require fewer bytes** to represent the same information compared to text-based formats. For instance, an integer occupies 2 bytes in binary, while its textual representation (e.g., "12345") requires significantly more bytes. This compactness reduces both transmission time and bandwidth usage. Another advantage is that it **eliminates the need for parsing** numeric values from text, making data processing faster and more efficient. However, binary data has notable drawbacks. It is **not human-readable**, making debugging and manual inspection challenging without specialized tools or decoding scripts. Additionally, platform-specific issues such as endianness (e.g., little-endian vs. big-endian) and differences in data structure interpretation (e.g., identifying integers, floats, or strings) can cause inconsistencies unless both sender and receiver adhere to a predefined protocol. In the provided example, the receiving program must first read the header 'H' to identify the start of a message. It then reads the low and high bytes for each integer and reconstructs the original value. Let's see this in action using Processing:

```

void draw() {
    // Wait until at least 3 bytes are available
    //(header + 2 bytes for the integer)
    if (myPort.available() >= 3) {

        // Check for the header character
        if (myPort.read() == HEADER) {

            // Read the low byte

```

```
int low = myPort.read();

// Read the high byte
int high = myPort.read();

// Reconstruct the 16-bit integer
value = (high << 8) | low;

println("Message received: " + value);
}

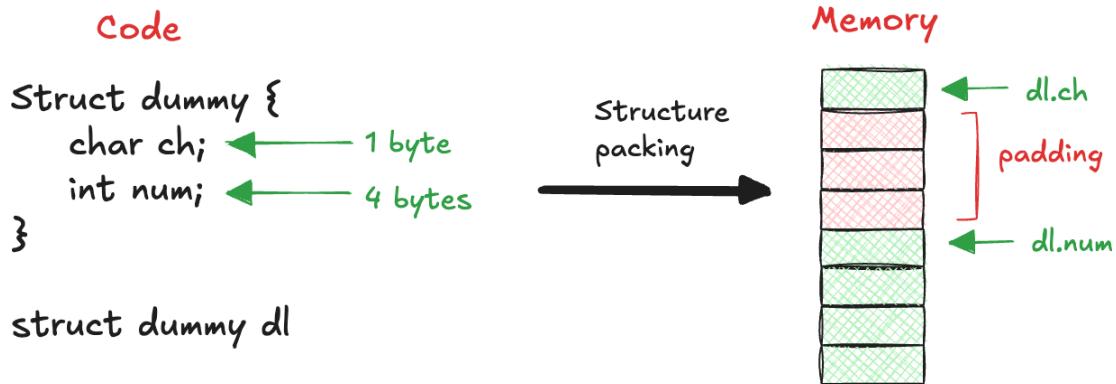
}

}
```

Sending binary data requires **careful planning and coordination** between the sender and receiver to ensure that the data is correctly interpreted. The sender must format the data consistently, while the receiver must parse the data according to the agreed-upon structure. In particular, key considerations include:

- **Variable Size:** ensure that the size of the data being sent matches the size expected by the receiver. Check the programming language's documentation for the exact size of data types (e.g., an int is 2 bytes on Arduino but might be 4 bytes in Processing). One approach can be to use **explicitly sized types** (e.g., uint16\_t in Arduino) and verify the range of values sent does not exceed the maximum value the receiving type can hold to avoid **overflow**.
- **Byte Order (Endianness):** ensure that the bytes within multi-byte values are sent in the order expected by the receiver (little-endian vs. big-endian). A solution can be standardize the byte order between systems, typically by sending data in little-endian order. Also using helper functions (like lowByte() and highByte()) on the sending side and reconstruct values correctly on the receiving side.
- **Synchronization:** the receiver must recognize the start and end of a message. If the receiver begins listening mid-stream, it could interpret bytes incorrectly. A possible solution is to include a header or a delimiter in the message that uniquely identifies the start of valid data (e.g., 'H' as a header). Of course, we need also to avoid the use of values in the message body that could match the header.
- **Structure Packing:** data alignment (see figure) can vary across compilers, leading to mismatches if a structure is packed differently on each side. Compilers may pad data to align it for better performance. A solution is to explicitly define structure packing. For example, in C/C++ the #pragma pack(1) preprocessor statement ensure no padding. In general, it

is better to avoid sending complex structures directly, instead serialize data manually to ensure consistent packing.



- **Flow Control:** if the sender transmits data faster than the receiver can process it, data could be lost or corrupted. A solution is to use a **handshake mechanism** where the receiver signals readiness before the sender transmits more data or choose an appropriate baud rate that allows the receiver to process data at the same rate it is sent.

Sending binary data from an external device to a microcontroller follows similar principles. It's important to ensure that the data format is correctly understood by the microcontroller's firmware. Depending on the system, the binary data might represent control commands, sensor readings, or other structured data that the microcontroller needs to process efficiently. The following code defines a simple communication system between a Processing application and an Arduino using serial communication. The message consists of an header ('|') and a tag ('M') to define the mouse X-coordinate and the mouse Y-coordinate clicked on a Processing window. The tag is used because the message can in principle contain multiple fields, each one identified by a different tag. The windows size is set to 200x400 pixels, so the X-coordinate ranges from 0 to 200 and needs only one byte to be represented, the Y-coordinate ranges from 0 to 400 and needs two bytes to be represented: (see *14.ReceiveBinary.ino* sketch):

```
import processing.serial.*;

Serial myPort;

// Define the header and the tag for the message
public static final char HEADER = '|';
public static final char MOUSE = 'M';

void setup(){
    size(200, 400);
```

```

    println(" Connecting to → " + Serial.list()[2]);
    myPort = new Serial(this,Serial.list()[2], 9600);
}

void draw(){ }

// This callback function is called whenever data is received
// and it is used to echoes the received data to the console
void serialEvent(Serial p) {
    String inString = myPort.readStringUntil('\n');
    if(inString != null) {
        println( inString );
    }
}

// This callback function is called whenever the mouse is pressed
// It gets the x and y coordinates of the mouse and sends them
void mousePressed() {
    int x = mouseX;
    int y = mouseY;
    sendMessage(MOUSE, x, y);
}

// The function sends a header, a tag, x-coordinate as single bytes and
// the y-coordinate as two bytes.
void sendMessage(char tag, int x, int y) {
    myPort.write(H HEADER);
    myPort.write(tag);
    myPort.write(x);
    myPort.write((byte)(y >> 8)); // MSB
    myPort.write((byte)(y & 0xFF)); // LSB
}

```

The receiving part on Arduino side is the following:

```

// Define header, tag, and message length
#define HEADER '|'
#define MOUSE 'M'
#define MESSAGE_BYTES 5

void setup() {
    Serial.begin(9600);

```

```
}

void loop() {
    // Check if there are enough bytes available to read
    if ( Serial.available() >= MESSAGE_BYTES) {

        // Check the header byte
        if( Serial.read() == HEADER) {

            // Read the tag byte
            char tag = Serial.read();

            // Check if the tag is for a mouse message
            if(tag == MOUSE) {

                // Read the X-coordinate (1 byte)
                int x = Serial.read();

                // Read the Y-coordinate (2 byte) and reconstruct the value
                int y = Serial.read() * 256;
                y = y + Serial.read();

                // Send back the received values
                Serial.print("Received mouse msg, x = ");
                Serial.print(x);
                Serial.print(", y ");
                Serial.println(y);
            }

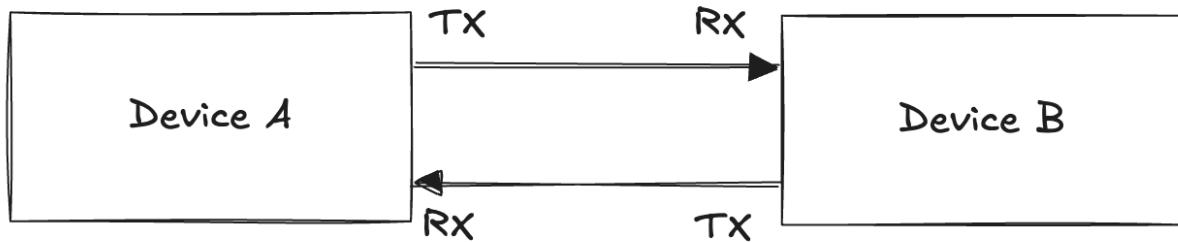
            // If the code gets here, the tag was not recognized.
            // this helps to ignore data that may be incomplete or corrupted.
            else {
                Serial.print("got message with unknown tag ");
                Serial.println(tag);
            }
        }
    }
}
```

In general, the binary approach is particularly useful for transmitting **large amounts of data** or when **speed and efficiency are critical**. However, it is essential to consider the **trade-offs between efficiency and readability** when choosing between binary and text-based formats. Bi-

nary format is recommended for systems where efficiency, fast data processing, and low latency are crucial. For all other cases, the ease of debugging and the human-readability offered by text-based formats typically make them the better choice.

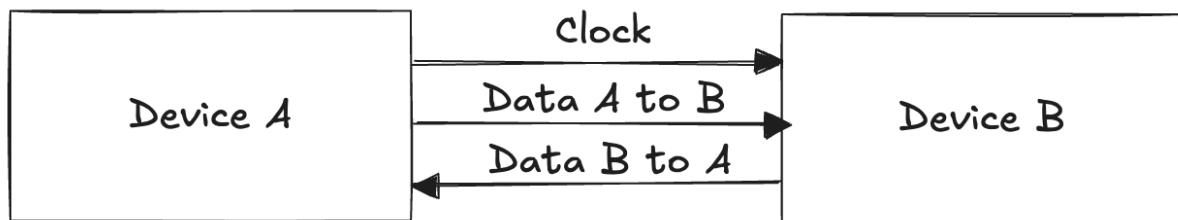
### 2.3.4 Synchronous communication

In the example we have seen so far, based on USB protocol and on the Serial library the communication between the Arduino and the computer is **asynchronous**:



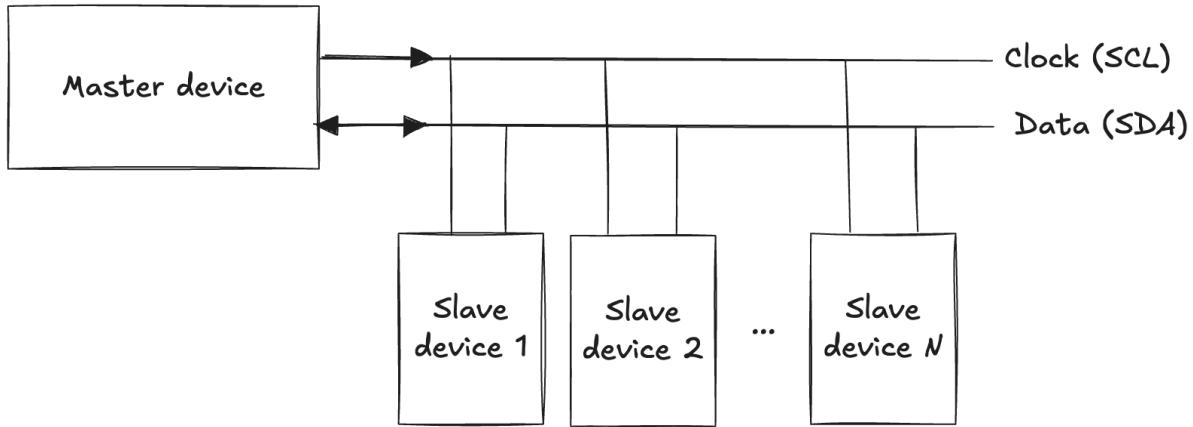
In asynchronous communication, data is sent without a shared clock signal, relying on the sender and receiver to agree on a specific data rate (baud rate) and interpret the data based on their internal timing mechanisms, making it crucial for both devices to have precise internal oscillators. Separate wires are used for transmitting (TX) and receiving (RX) data, allowing bidirectional communication. This approach is simple and effective for many applications, but it can be prone to timing errors, especially over long distances or in noisy environments.

An alternative approach is **synchronous communication**, where devices use a **shared clock** signal to synchronize data transmission. This method eliminates the need for independent timing mechanisms within the devices, as the clock acts as a "conductor" guiding the flow of data.



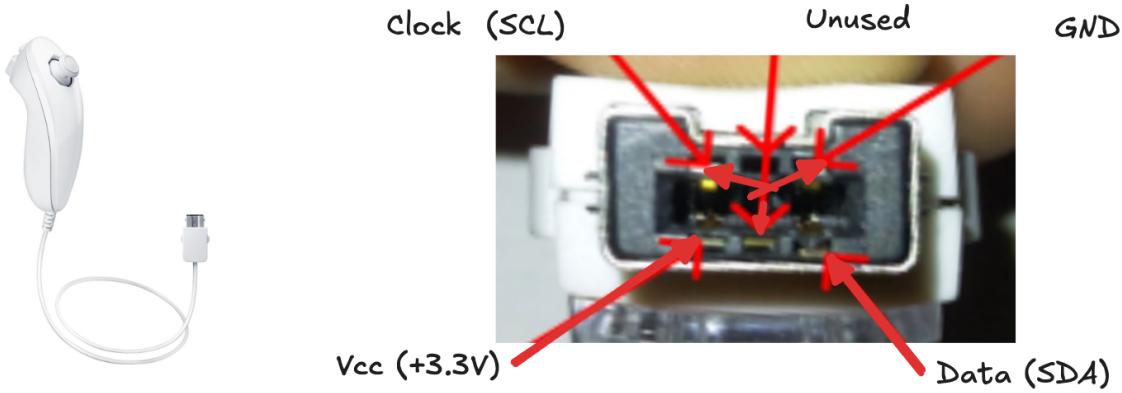
This method ensures that both devices operate in lockstep, sampling data at the correct moments. Synchronous communication offers more reliable data exchange by eliminating timing mismatches and ensuring precise synchronization between devices. However, it requires **additional wiring for the clock signal**, which can increase hardware complexity. The choice between asynchronous and synchronous communication depends on the application requirements for timing precision, hardware constraints, and error tolerance.

The **I2C (Inter-Integrated Circuit)** protocol is an example of synchronous communication. In I2C, devices (master and slaves) communicate over a shared bus consisting of two wires: one for data (**SDA: Serial Data Line**) and one for the clock (**SCL: Serial Clock Line**). It supports multiple devices on the same bus, where each device has a unique address. The master device controls the communication by generating the clock signal and initiating data transfers, while the slave devices respond to commands from the master:

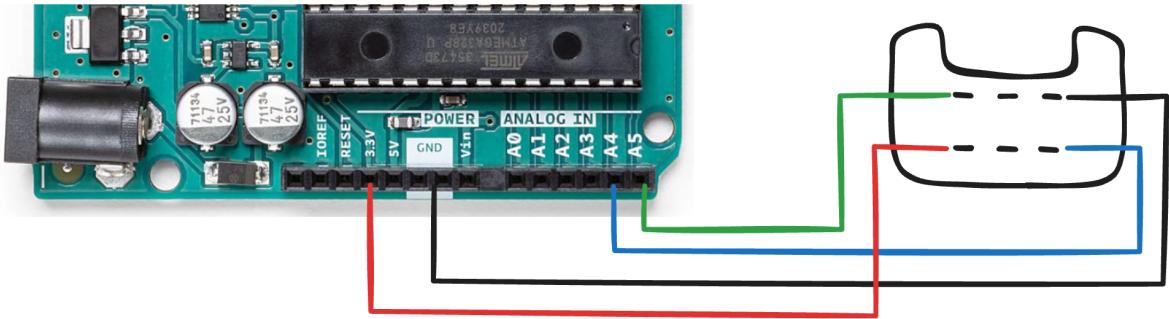


Data is transmitted bit by bit, and each bit is sampled on the clock rising or falling edge (depending on the implementation), this synchronization is what makes I2C a synchronous protocol. When a master device wants to communicate with a slave, it first sends the address of the slave along with a read/write bit. The slave device that matches the address acknowledges the request, and data can then be transferred between the devices. Data is transferred in 8-bit chunks (1 byte at a time), with each byte being transmitted in sync with the clock. After each byte, the receiver sends an acknowledgment (ACK) bit, signaling that the byte was successfully received. The clock-driven nature of I2C ensures that both the master and slave devices stay synchronized, which eliminates the need for start and stop bits, a feature that is common in asynchronous protocols like UART. The data integrity in I2C is guaranteed by the clock signal, which ensures that each bit is transmitted at the correct time. Because of this synchronization, the protocol is reliable and efficient in systems where multiple devices need to communicate with a single master.

As an example, we can interact with the **Wii Nunchuck**, a popular I2C device, to read its accelerometer and joystick data:



Most Arduino boards support I2C communication mapped to specific analog pins: SCL on pin A5 and SDA on pin A4:



Although there are no official datasheets, the functionality of Wii Nunchuck has been **reverse-engineered**, making it accessible to developers. The I2C slave address is 0x52 and the communication is based on an handshake signal to initialize the Nunchuck (send a sequence of two bytes 0x40 and 0x00) and a data request command (send one byte 0x00) to request data. The Nunchuck responds with a 6-byte data packet containing the joystick positions (X byte 0 and Y byte 1), accelerometer data (X byte 2, Y byte 3, Z byte 4) and the button states and some least significant bits of acceleration (byte 5). The data are encoded using a specific scheme to ensure data integrity:

```
data = (reading XOR 0x17) + 0x17
```

Arduino provides the **Wire library**, which abstracts the I2C protocol. The library simplifies the process of sending and receiving data over the I2C bus. The function `Wire.begin()` initializes the bus in master mode. For sending data the `Wire.beginTransmission(address)` starts communication with a specific slave device (identified by its address), `Wire.write(data)` sends data (a single byte, a string, or an array of bytes) to the slave device and `Wire.endTransmission()` ends the transmission and releases the bus. To request data, `Wire.requestFrom(address, quantity)` requests

a specified number of bytes from a slave device, `Wire.read()` reads the received data byte by byte and `Wire.available()` returns the number of bytes available to read. The following example demonstrates how to read data from a Nunchuck and send data to an external device connected on the serial port (see *15.Nunchuck.ino*):

```
// Include the Wire library for I2C communication
#include <Wire.h>

// Define the Nunchuk slave address
#define NUNCHUK_ADDRESS 0x52

// Buffer for data received
static uint8_t nunchuck_buf[6];

// Establishes I2C communication with the nunchuck
static void nunchuck_init() {
    // Join the I2C bus as master
    Wire.begin();

    // Initialize the slave device (nunchuck) on address 0x52
    Wire.beginTransmission(NUNCHUK_ADDRESS);

    // Send handshake signal to initialize the Nunchuk
    Wire.write((uint8_t)0x40);
    Wire.write((uint8_t)0x00);
    delay(100);
    Wire.endTransmission();

    Serial.println("Wii Nunchuk initialized!");
}

static void nunchuck_send_request() {
    Wire.beginTransmission(NUNCHUK_ADDRESS);
    Wire.write((uint8_t)0x00);
    Wire.endTransmission();
}

// A function to decode Nunchuk data
char nunchuk_decode(char reading) {
    return (reading ^ 0x17) + 0x17;
}

// Receive data from the nunchuck
```

```
static int nunchuck_get_data() {
    int cnt=0;

    // Get six bytes of data from device 0x52 (nunchuck).
    Wire.requestFrom (NUNCHUK_ADDRESS, 6);

    // Indicate how many bytes have been received
    while (Wire.available ()) {
        nunchuck_buf[cnt] = nunchuk_decode( Wire.read() );
        cnt++;
    }

    nunchuck_send_request();

    if (cnt >= 5) {
        return 1;
    }

    return 0;
}

int nunchuck_zbutton() { return ((nunchuck_buf[5] >> 0) & 1) ? 0 : 1; }
int nunchuck_cbutton() { return ((nunchuck_buf[5] >> 1) & 1) ? 0 : 1; }
int nunchuck_joyx() { return nunchuck_buf[0]; }
int nunchuck_joyy() { return nunchuck_buf[1]; }
int nunchuck_accelx() { return nunchuck_buf[2]; }
int nunchuck_accely() { return nunchuck_buf[3]; }
int nunchuck_accelz() { return nunchuck_buf[4]; }

void setup() {
    Serial.begin(9600);
    nunchuck_init();
}

void loop() {

    nunchuck_get_data();

    Serial.write(nunchuck_joyy());

    delay(100);
}
```

We can write a Processing program to visualize data received from the Nunchuk connected to the Arduino. It draws a line on the canvas that dynamically updates its position based on the data sent via the serial port:

```
import processing.serial.*;

Serial myPort;

void setup() {
    size(200, 200);
    myPort = new Serial(this, Serial.list()[2], 9600);
}

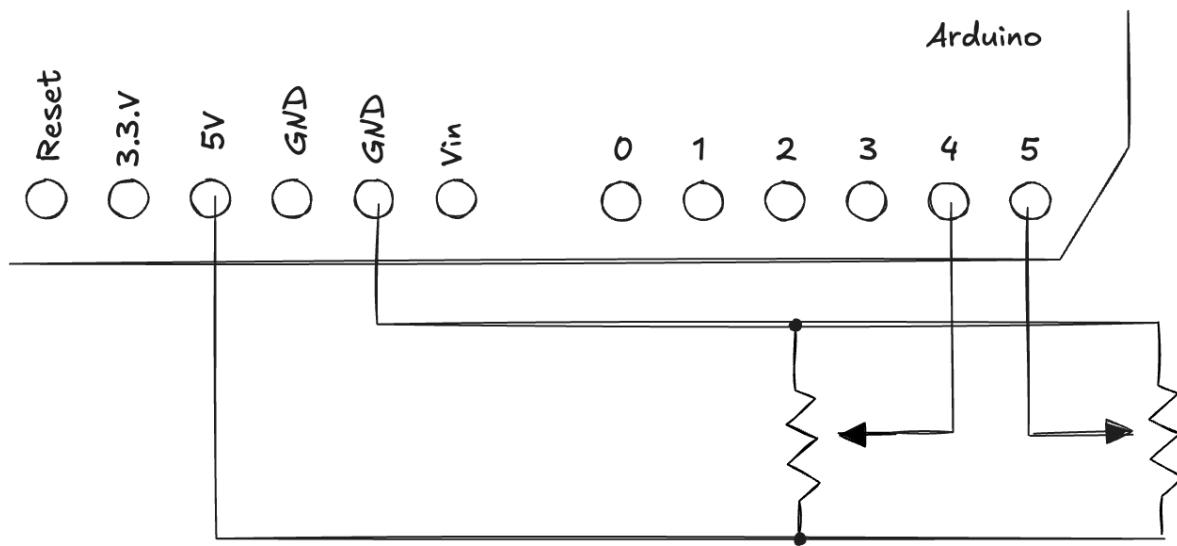
void draw() {
    if (myPort.available() > 0) {
        int value = myPort.read();
        background(255);
        println(value);
        line(0, 226-value, 200, 226-value);
    }
}
```

## 2.4 Hands-on Activity

1 - Create a simple traffic light system. The system should alternate between green, yellow, and red lights, simulating the behavior of a real traffic light. Solution: *16.TrafficLights*

2 - Enhance the traffic light system by adding pedestrian lights and a push button. When the button is pressed, the pedestrian light will turn green, allowing pedestrians to cross safely. Solution: *17.PedestrianTrafficLights*

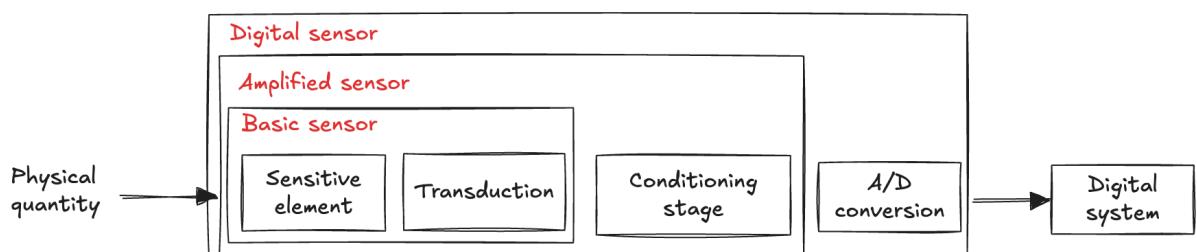
3 - Control the computer mouse cursor using two potentiometers connected to an Arduino, see the next figure. You should use Processing to receive data from the Arduino and move the mouse cursor using the Robot class, which generates native system input events.



This technique for controlling mouse is easy to implement and should work with any operating system that can run the Processing application. If we require Arduino to actually appear as a mouse to the computer, we have to emulate the actual USB protocol (**Human Interface Devices (HID)**). Solution: *18.Mouse*

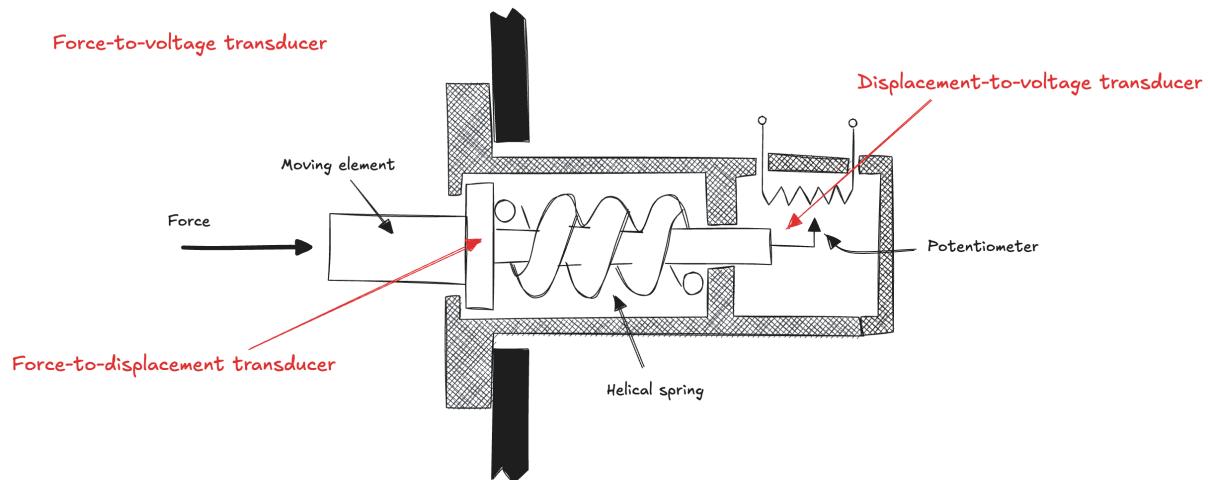
### 3 Sensors

A sensor is a device that detects **physical properties or changes in the environment** (such as temperature, light, motion, pressure, or sound) and converts this information (called **stimulus**) into **an electrical signal** that can be read and processed by an electronic system. The following image presents a simplified block diagram of a digital sensor system, detailing its components and their roles in the measurement process:



A physical quantity, which represents a real-world phenomenon the sensor is designed to measure, such as temperature, pressure, or light intensity. This quantity is detected by a sensitive element, the core component of the sensor that interacts directly with the physical environment. The sensitive element converts the physical input into a measurable electrical signal, a process

known as **transduction**. In some cases, the resulting signal may be too weak for effective processing, necessitating a **conditioning stage**, often involving amplification. This step ensures that the signal reaches an appropriate strength for further handling. The next step is **analog-to-digital conversion (A/D)**, where the analog electrical signal is transformed into a digital format. This conversion is essential for compatibility with digital systems, which interpret and process the data. Finally, the digital system takes over, performing operations such as filtering, calibration, data analysis, or control to make the information usable for applications. As a simple example, we can consider a force transducer setup:



An external force is applied to a moving element, which is typically a piston or a similar component. This force causes the moving element to compress a helical spring. As the spring compresses, it results in the displacement of the moving element. This displacement is then measured by a displacement-to-voltage transducer, which often uses a potentiometer, a type of variable resistor. As the moving element shifts, it alters the resistance of the potentiometer, which in turn changes the output voltage. The change in voltage is directly proportional to the amount of displacement. As a result, the output voltage provides a measure of the force applied to the system.

The electrical signal generated by the sensor depends on its type and the amount of information it needs to convey. Some sensors, such as photoresistors and piezo knock sensors, are made from materials that change their electrical properties in response to physical changes. Other sensors, however, are more advanced electronic modules that include their own microcontroller to process data before transmitting the signal to the microcontroller for further use. All sensors can be broadly classified into two categories: passive and active. A **passive sensor** operates without requiring any external energy source. It directly converts the energy from an external stimulus into an electrical signal, making it self-sufficient in generating an output. For instance, a thermocouple generates a voltage in response to a temperature difference, a photodiode produces

current when exposed to light, and a piezoelectric sensor generates an electric charge under mechanical stress. In contrast, **active sensors** depend on an external power source, known as an excitation signal, for their operation. The sensor modifies this signal to produce the desired output. For example, a thermistor, which is a temperature-sensitive resistor, does not generate an electrical signal by itself. Instead, when an electric current is passed through it, its resistance can be measured by observing the variations in current or voltage across it. These variations, expressed in ohms, correspond to temperature values through a predefined relationship.

### 3.1 Specifications

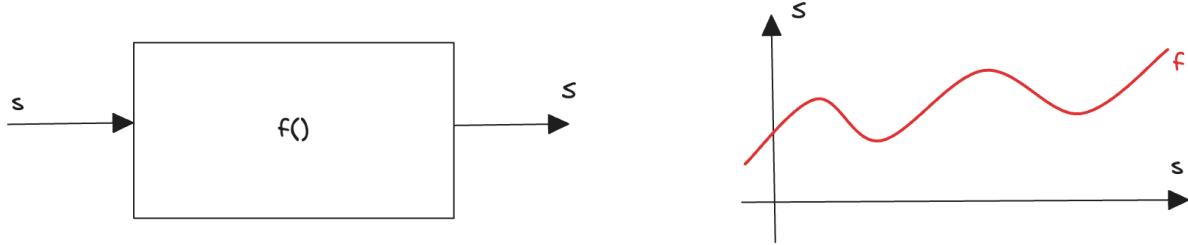
A sensor performance is defined by a set of **key parameters** that characterize its capabilities. These specifications determine how well a sensor can convert real-world signals into electrical signals for further processing or feedback. In this section, we will explore the primary specifications of sensors, focusing on how they impact sensor selection and performance. We will examine concepts like accuracy, response time, linearity, and noise levels, among others. Notice the importance of environmental factors, such as temperature and humidity, that can affect sensor performance in real-world applications. By understanding specifications, we can make **informed decisions** when designing systems that rely on accurate and reliable sensor data.

#### 3.1.1 Transfer function

An ideal output-stimulus relationship exists for every sensor, representing how the output would perfectly correspond to the input if the sensor were manufactured under **ideal conditions**. In such a theoretical scenario, the sensor would always provide an accurate and true representation of the measured stimulus. This ideal relationship between the output and the stimulus is captured by the **transfer function**, which defines how the electrical signal produced by the sensor depends on the stimulus. Mathematically, this can be written as

$$S = f(s)$$

where S is the output signal (one of the characteristics of the output electric signal used by the data acquisition devices as the sensor output, like amplitude, frequency, or phase, depending on the sensor properties) and s is the input stimulus.

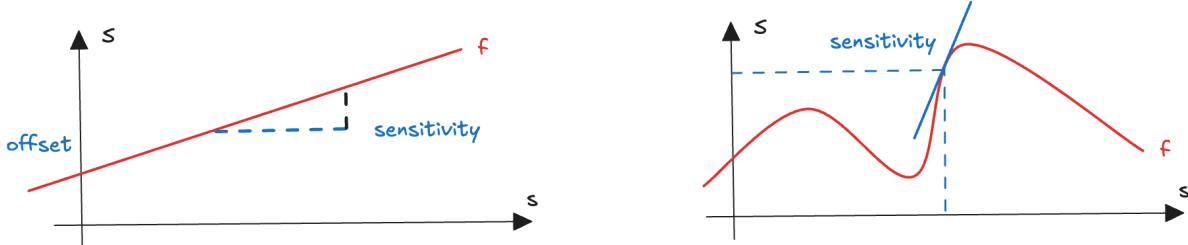


The transfer function might be a **simple linear relationship**, such as

$$S = a + b \cdot s$$

where  $a$  represents the intercept or the output signal at zero input (**offset**), and  $b$  is the slope, also known as the **sensitivity** of the sensor. It reflects how strongly a sensor output reacts to changes in its input. Intuitively, it measures how much the sensor "notices" and responds to variations in the stimulus. A sensor with high sensitivity will produce a significant change in output even for a small change in the input, making it excellent for detecting subtle variations. Conversely, a sensor with low sensitivity requires larger changes in the input to produce noticeable changes in the output, making it less responsive to minor fluctuations. Alternatively, the function may be **non-linear**, taking forms such as logarithmic, exponential, or power functions. For non-linear sensors, sensitivity varies with the input value. At a specific input value, the sensitivity can be defined as the derivative of the output signal with respect to the input:

$$b = \frac{ds}{ds}|_{s_0}$$



In this case the transfer function may still be approximated as **linear over small input ranges**. This approximation leverages the fact that, for sufficiently small changes in the input, the function behaves almost like a straight line. Mathematically, the transfer function can be locally approximated by a linear equation using a first-order Taylor expansion around a specific input value:

$$S \approx f(s_0) + \frac{df}{ds}|_{s_0} \cdot (s - s_0)$$

However, this linear approximation is valid only for small changes in the input, as the function may deviate significantly from linearity for larger variations.

For broader ranges, a **piecewise approximation** involving several linear segments is often used to model the non-linear behavior. The transfer function is divided into segments with inout ranges:

$$[s_0, s_1], [s_1, s_2], \dots, [s_{n-1}, s_n]$$

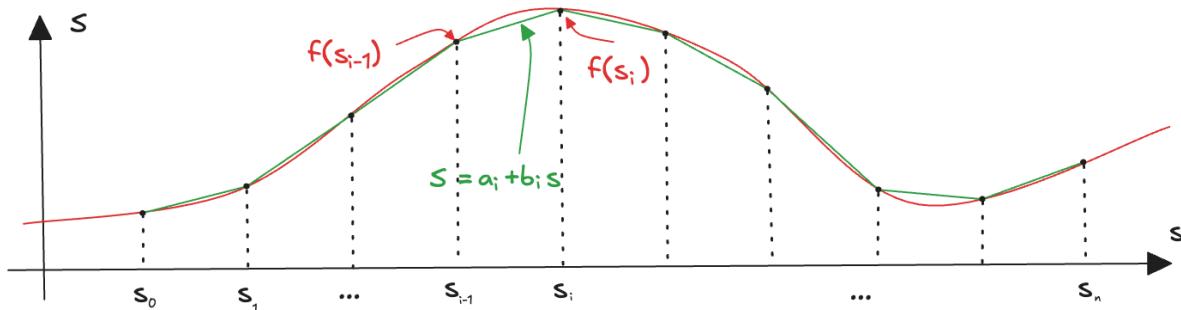
In each segment, the transfer function can be approximated as:

$$S = a_i + b_i \cdot s \text{ for } s \in [s_{i-1}, s_i]$$

To ensure continuity between segments, the values of intercept and sensitivity of each linear segment are determined such that the endpoints of each segment match the actual transfer function value:

$$f(s_{i-1}) = a_i + b_i \cdot s_{i-1}$$

$$f(s_i) = a_i + b_i \cdot s_i$$



This ensures that the piecewise approximation closely follows the original non-linear transfer function across the entire input range. The accuracy of the piecewise approximation improves as the number of segments increases, but at the cost of increased complexity in modeling and computation.

The transfer function can be also **multi-dimensional**, involving multiple input variables. In this case, the transfer function would be a function of multiple variables, such as

$$S = f(s_1, s_2, s_3, \dots)$$

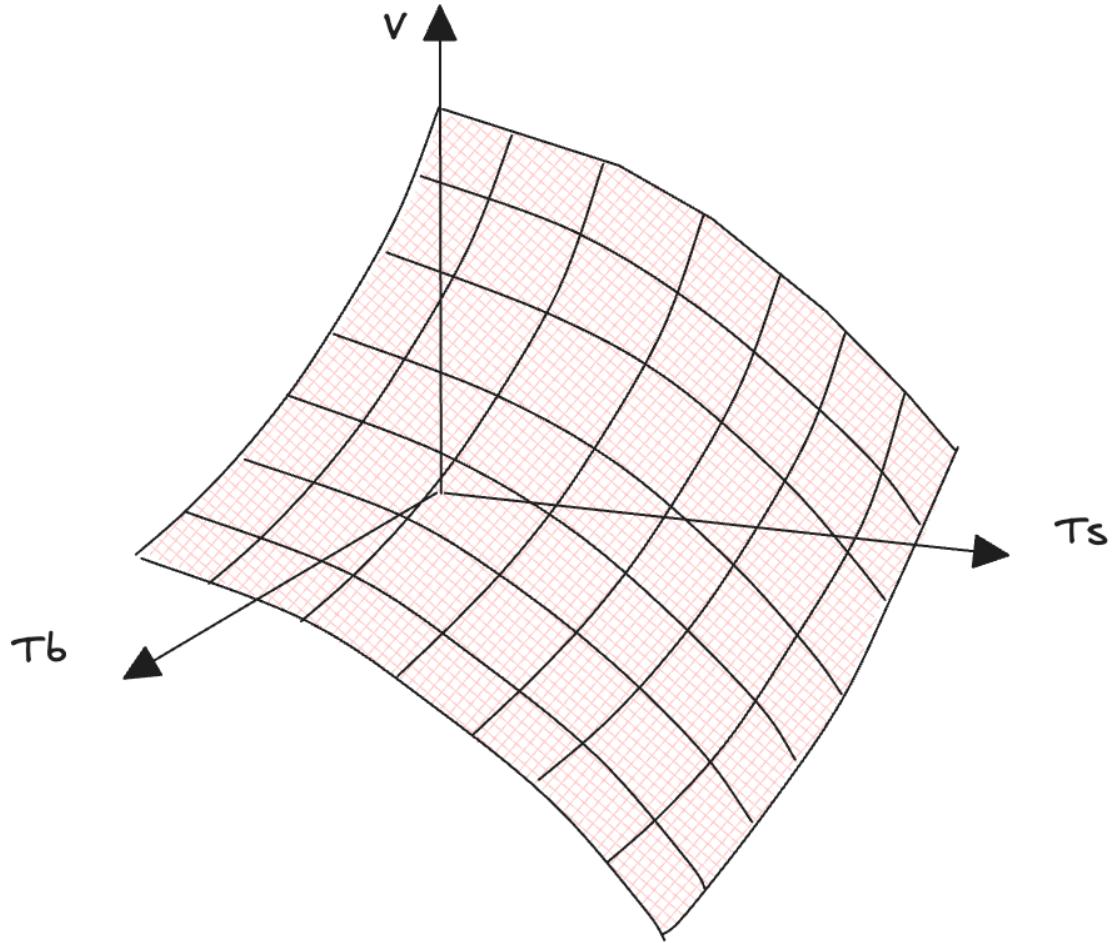
The sensitivity can be defined as the **partial derivative** of the output signal with respect to the particular input variable:

$$b_i = \frac{\partial S}{\partial s_i} \Big|_{s_0}$$

A common example is the transfer function of a **thermal radiation sensor**, where the output voltage V depends on both the absolute temperature of the measured object ( $T_b$ ) and the absolute temperature of the sensor's surface ( $T_s$ ). The relationship is described by the equation:

$$V = G(T_b^4 - T_s^4)$$

Where G is a constant that depends on the sensor's properties. We can show the transfer function as a 3D plot:



This transfer function shows not only non-linearity, represented by the fourth power dependence, but also a dependence on the sensor own surface temperature. To determine the sensitivity to the object temperature, the partial derivative of the output voltage with respect to  $T_b$  is calculated as:

$$b = \frac{\partial V}{\partial T_b} = 4GT_b^3$$

Summarizing, the transfer function is a crucial concept in sensors, as it provides a mathematical model for understanding how the sensor responds to different stimuli. By characterizing the transfer function, engineers can determine several sensor properties, like sensitivity, linearity, and range of operation, enabling them to optimize the sensor performance for specific applications.

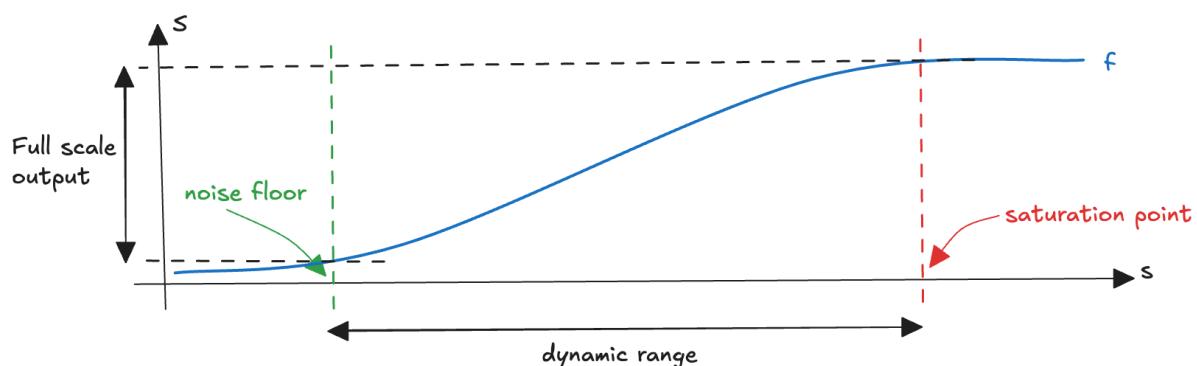
### 3.1.2 Dynamic Range

The **dynamic range** quantifies the sensor ability to **operate effectively across a spectrum of signal intensities** without distortion or loss of information. The upper limit is determined by its **saturation point**, the maximum signal intensity that the sensor can handle before it becomes incapable of producing a valid output. Every sensor has its operating limits. Even if it is considered linear, at some levels of the input stimuli, its output signal no longer will be responsive, a further increase in stimulus does not produce a desirable output. It is said that the sensor exhibits a saturation. Beyond this point, the response plateaus, leading to a loss of detail in high-intensity regions. On the other end, the lower limit is defined by the **noise floor**, the smallest signal the sensor can distinguish from inherent electronic noise, which arises from thermal or electrical fluctuations within the sensor system. The **Full-scale output** is often also specified together with the dynamic range and it is defined as the difference between the electrical signals measured when the maximum and minimum stimulus are applied as input. The full-scale output represents how the dynamic range is mapped to the output signal:

$$DR = s_{max} - s_{min}$$

$$FS = S_{max} - S_{min}$$

Where  $s_{max}$  is the saturation point and  $s_{min}$  is the noise floor and  $S_{max}$  and  $S_{min}$  are the corresponding output signals



A high dynamic range is particularly desirable. For instance, in imaging sensors, a wide dynamic range allows the capture of scenes with both bright and dark areas, preserving details in both shadows and highlights. Similarly, in audio sensors like microphones, a high dynamic range ensures the accurate representation of both faint whispers and loud sounds without distortion. For sensors with very broad and nonlinear response characteristic, the dynamic range is often expressed in **decibels**, which is a logarithmic scale used to express the ratio between two values. The logarithmic nature compresses large ranges, making them more manageable:

$$DR_{dB} = 10 \cdot \log_{10} \left( \frac{s_{max}}{s_{min}} \right)$$

We can use decibels for the output signal as well. If the sensor output spans several orders of magnitude, expressing the output in decibels can help compress the range into a more compact and readable form:

$$S_{dB} = 10 \cdot \log_{10} \left( \frac{S}{S_{ref}} \right)$$

where  $S_{ref}$  is a reference signal, often the smallest measurable signal or the sensor noise floor.

The logarithmic nature of the decibel scale is particularly useful for describing situations where the **human perception** follows a similar logarithmic pattern. The human eye or ear, have dynamic ranges that align well with the logarithmic nature of decibels. For instance, the human ear perceives loudness on a logarithmic scale, so decibels are an intuitive way to describe the perceived loudness of sounds.

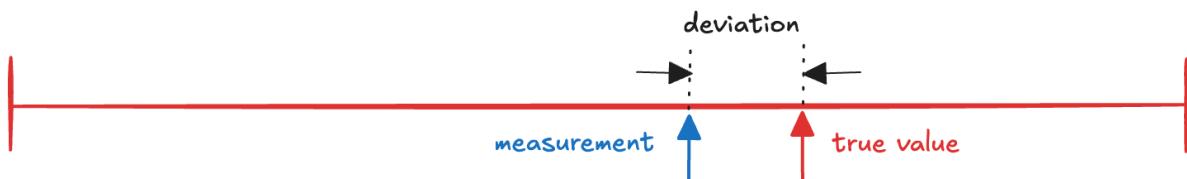
### 3.1.3 Accuracy

A sensor is a real device to measure a physical quantity, and as such, it is subject to **imperfections** and **uncertainties** that can affect its measurements. We define as **deviation** the difference between the value of the physical quantity measured by the sensor and the actual value of the quantity. Mathematically, it can be expressed in terms of absolute error and relative error. The **absolute error** represents the difference between the measured value and the true value:

Absolute Error = Measured Value – True Value

The **Relative error** expresses the deviation of the sensor as a fraction of the true value. It provides insight into how significant the error is in comparison to the scale of the measured quantity:

$$\text{Relative Error} = \frac{\text{Absolute Error}}{\text{True Value}}$$



As an example, consider a displacement sensor that ideally generate 1 mV per 1-mm displacement. Its transfer function is linear with a slope of 1 mV/mm:

$$S[mV] = 1[mV/mm] \cdot s[mm]$$

However, in an experimental setup, a displacement of 10 mm produced an output of 10.5 mV. Converting this value into the displacement using the inverted transfer function:

$$s[\text{mm}] = \frac{S[\text{mV}]}{1[\text{mV/mm}]} = 10.5[\text{mm}]$$

The deviation is then:

$$\text{Deviation} = 10.5[\text{mm}] - 10[\text{mm}] = 0.5[\text{mm}]$$

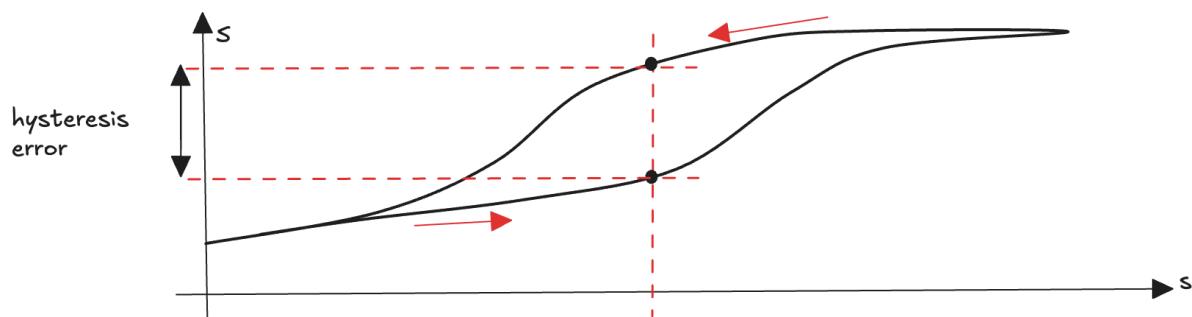
The accuracy of the sensor is defined as the **maximum deviation** between the measured value and the true value of the physical quantity. It refers to **how closely** the output of a sensor matches the true value of the measured quantity. It is one of the most important specifications when evaluating a sensor, as it determines how well the sensor's reading represents the actual physical quantity. In practice, sensor accuracy is not always perfect due to various sources of error, such as calibration inaccuracies, noise, or environmental factors. In many applications, the accuracy of a sensor is described as a percentage of the sensor full-scale reading:

$$\text{Accuracy} = \frac{\text{Max Absolute Error}}{\text{Full-scale Reading}} \times 100\%$$

### 3.1.4 Hysteresis

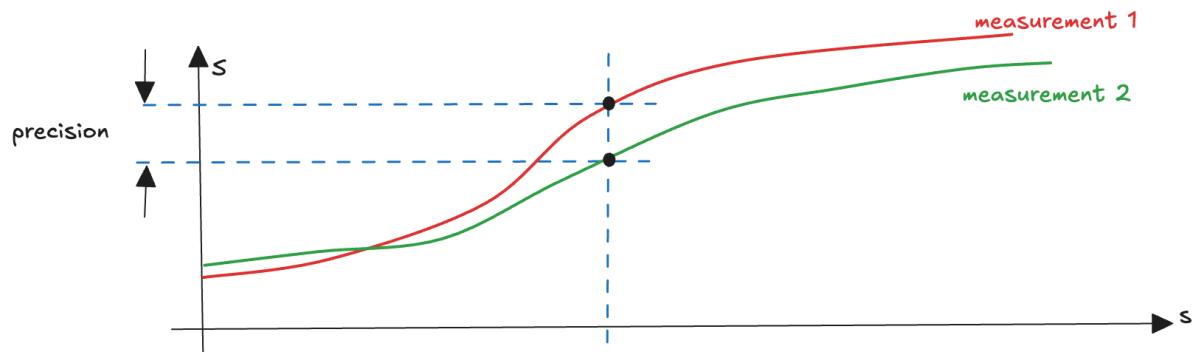
An **hysteresis error** is a deviation of the sensor output at a specified point of the input signal when it is approached from the **opposite directions**. For example a sensor that measures the position of a moving object may produce different output values when the object is moving towards the sensor than when it is moving away from it. It is often caused by mechanical or thermal effects within the sensor that introduce a **memory effect** in the output signal. Hysteresis can be quantified by measuring the difference between the output values when the input signal is increasing and when it is decreasing. The hysteresis error is defined as the **maximum difference** between the two output values:

$$\text{Hysteresis Error} = \max(S_{\text{increasing}} - S_{\text{decreasing}})$$

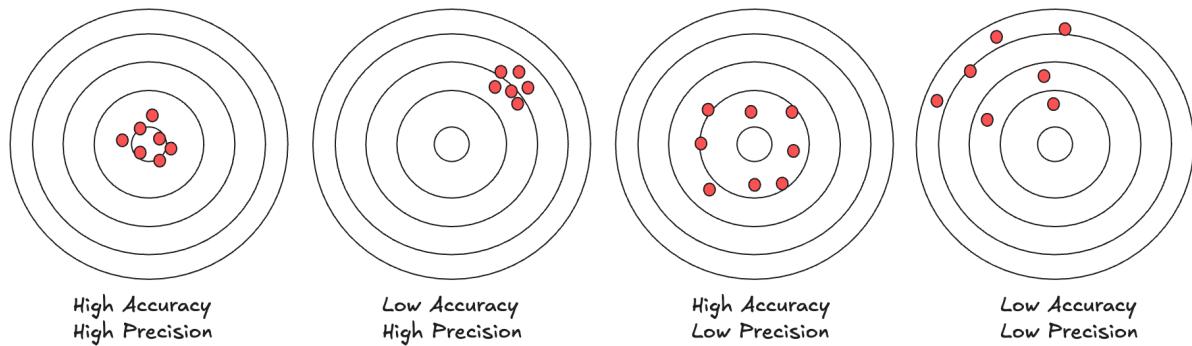


### 3.1.5 Precision

The **precision** of a sensor refers to the ability of a sensor to produce consistent output readings when the same input stimulus is applied multiple times under identical conditions. In simple terms, if you measure the same physical quantity multiple times with the same sensor, the sensor should give you very similar results each time, even if we perform the measurements at different moments. This reflects the **repeatability** of the sensor performance, which is a crucial aspect of its overall reliability.



Accuracy and precision are closely related but distinct concepts. Accuracy refers to how close the sensor output is to the true value of the measured quantity, while precision focuses on the consistency of the sensor output when measuring the same quantity multiple times. A sensor can be accurate but not precise, if it consistently produces the same error in its measurements. Conversely, a sensor can be precise but not accurate if it consistently produces the same output value that is different from the true value:



### 3.1.6 Resolution

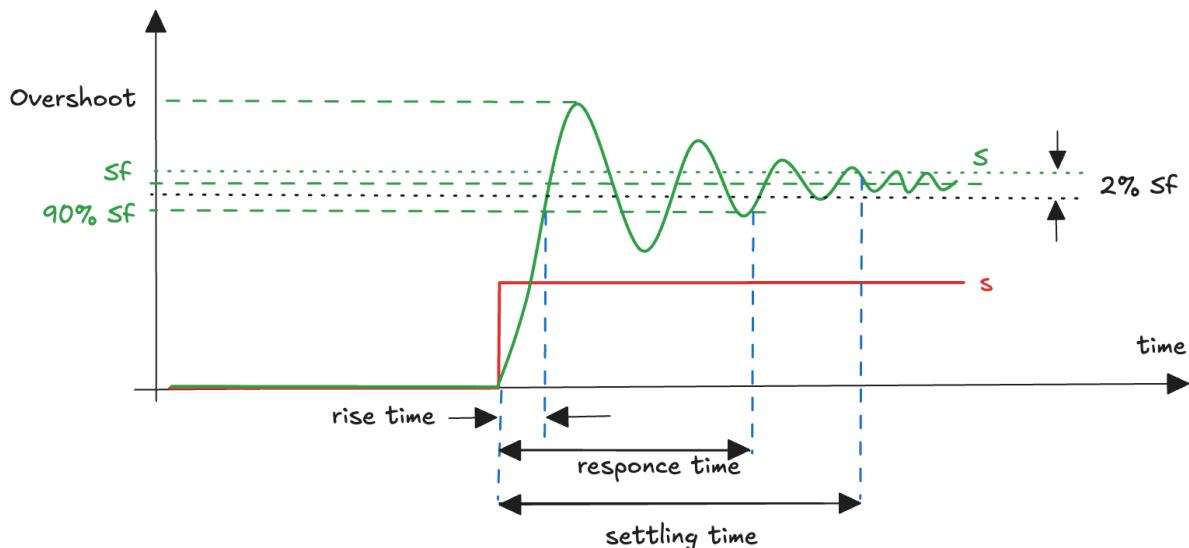
The resolution is **the smallest detectable change** in the measured quantity that the sensor can distinguish. It defines the sensor ability to detect fine details. For analog sensors, the resolution

is influenced by the noise and the precision of the measurement system. In digital sensors, the resolution is typically determined by the **number of bits** in the sensor's analog-to-digital converter (ADC).

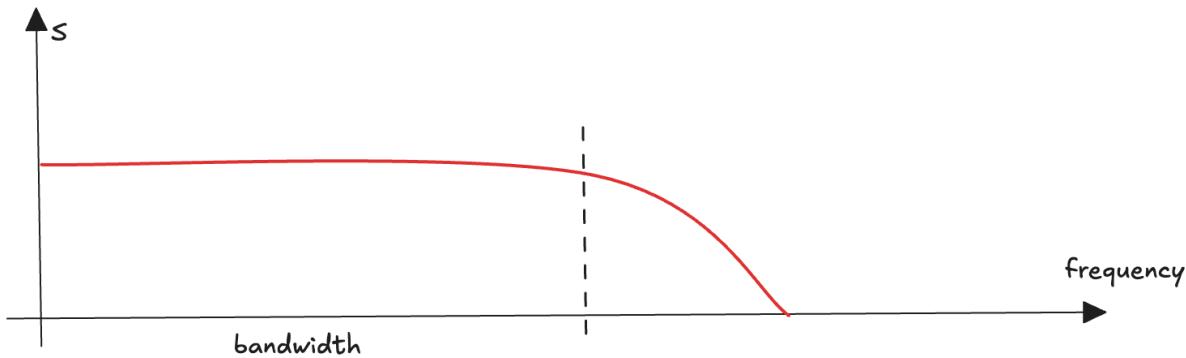
### 3.1.7 Dynamic characteristics

The dynamic characteristics of a sensor refer to **how the sensor responds to changes in the input signal over time**, particularly its ability to track rapidly changing or varying inputs. These characteristics describe how well the sensor can detect, follow, and accurately measure signals that are not constant, such as fluctuating or transient inputs. Key dynamic characteristics include:

- **Rise Time:** the time it takes for the sensor output to go from 0 to an higher value, often 90% of the final output. It focuses only on the initial portion of the transition and measures how quickly the output ramps up to the near-final value.
- **Response Time:** the time it takes for the sensor to initially react to a change in the input signal and reach a significant portion (90%) of the final output value. It includes both the rise phase and potentially part of the stabilization phase, focusing on the overall responsiveness of the sensor to the change.
- **Settling Time:** the time it takes for the sensor to stabilize within a certain range after a step change in input, typically within 2% of the final value. It is about how long it takes for the sensor to stop fluctuating and produce a stable output.
- **Overshoot:** the extent to which the sensor' output exceeds the final value during a transient response before stabilizing



Typically, we characterize the sensor from this point of view using the **frequency response**, which shows the magnitude and phase of the sensor output as a function of the frequency of the input signal. We can define the **bandwidth** of the sensor as the range of frequencies over which the sensor can accurately measure the input signal:



### 3.1.8 Reliability

The **reliability** of a sensor refers to its ability to consistently perform accurate measurements under specified conditions over a defined period. It is typically expressed as the **probability** that the sensor will operate without failure or degradation in performance for a given duration or number of usage cycles. Typically it is expressed using the **mean time between failures (MTBF)**, which is the average time that a sensor can operate without failing.

$$MTBF = \frac{\text{Total Operating Time}}{\text{Number of Failures}}$$

The reliability of a sensor can be influenced by various factors, such as the quality of the sensor components, the manufacturing process, the operating conditions, and the maintenance practices. A reliable sensor is essential for ensuring the accuracy and consistency of the measurements it provides, especially in critical applications where sensor failure can have serious consequences.

## 3.2 Calibration

Calibration is the process of **adjusting and verifying the accuracy** of a sensor by comparing its output to a known reference. It is a critical step in ensuring that the sensor provides accurate and reliable measurements. Calibration is necessary because sensors can exhibit **drift** over time, where their output changes due to factors like aging, temperature variations, or mechanical stress. By calibrating a sensor, we can correct for these changes and ensure that its output remains accurate and consistent. In order to understand the calibration process, let's consider

a simple example. Suppose we have a linear sensor designed to measure a physical quantity, like a force  $F$  and produce an output voltage  $V$ . The relationship between the input force and the output voltage is assumed to be linear:

$$V = m \cdot F + b$$

And suppose that from the datasheet we know that the sensor has a sensitivity of  $m=0.2V/N$  and an offset of  $b=0.4V$ . However, we don't know if in practice the sensor behaves exactly as expected and we want to calibrate it. The calibration process typically involves the following steps:

### 3.2.1 Establishing a reference

The first step is to identify a known reference value for the physical quantity being measured. In the case of our linear sensor we can apply two known forces  $F_1$  and  $F_2$ :

$$F_1 = 10N, F_2 = 20N$$

The reference can be obtained from another calibrated sensor or using a physical standard with a known value. The reference should be traceable to a recognized standard to ensure its accuracy.

### 3.2.2 Exposure to the reference value

The sensor is then exposed to the reference value, and its output is measured. In our case, we can apply the forces  $F_1$  and  $F_2$  and measure the output voltages  $V_1$  and  $V_2$ :

$$V_1 = 2.5V, V_2 = 4.7V$$

Normally, it is not sufficient to measure the output only once, but it is necessary to repeat the measurement several times to account for variability and noise in the sensor output.

Then, we can calculate the difference between the real sensor output and the value of the sensor output expected from its transfer function when exposed to the reference value. From the datasheet, the expected output voltages for the forces  $F_1$  and  $F_2$  should be:

$$eV_1 = 0.2 \cdot 10 + 0.4 = 2.4V \quad eV_2 = 0.2 \cdot 20 + 0.4 = 4.4V$$

The difference is called calibration error.

### 3.2.3 Adjusting the sensor

If the calibration error is significant, the sensor may need to be adjusted to correct for the error. This adjustment can involve changing the sensor settings, recalibrating its internal components,

or applying correction factors to its output. In our case we can recalculate the parameter of its linear transfer function by solving the system of equations:

$$m \cdot 10 + b = 2.5$$

$$m \cdot 20 + b = 4.7$$

That provides the new values of the sensitivity and the offset:

$$m = 0.21V/N, b = 0.3V$$

### 3.2.4 Validation

After adjusting the sensor, the calibration process is repeated to verify that the sensor output now matches the reference value within an acceptable tolerance. The sensor is considered calibrated if the calibration error falls within the specified limits. However, this is a simple example because we take only two points supposing that the sensor is still linear. In practice, calibration is often more complex, involving multiple reference points, non-linear corrections, and detailed analysis of the sensor's behavior. The calibration process is essential for ensuring the accuracy and reliability of sensor measurements, especially in critical applications where precise measurements are required. By calibrating sensors regularly and following best practices, we can maintain the quality and performance of sensor systems over time.

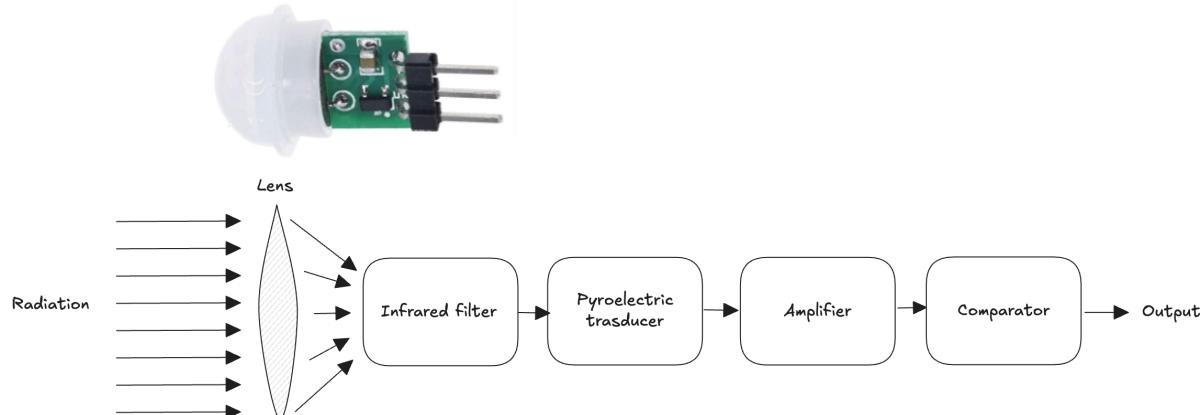
In general, the theory of measurement is a complex field that involves a deep understanding of physics, electronics, and signal processing. The design and implementation of sensor systems require careful consideration of the sensor characteristics, the measurement environment, and the intended application. By understanding the key concepts of sensors, specifications, and calibration, we can develop effective sensor systems that provide accurate, reliable, and consistent measurements for a wide range of applications.

## 3.3 Context Awareness

Sensors are essential for detecting and quantifying various **environmental parameters**, enabling embedded systems to make informed decisions. For example, sensors can regulate room temperature, adjust display brightness, or detect the presence of objects. This task is inherently complex because the environment is a dynamic, multifaceted system with numerous interacting components. By measuring a wide range of environmental parameters, sensors play a key role in achieving what is often referred to as **context awareness**, allowing systems to adapt to and interact intelligently with their surroundings.

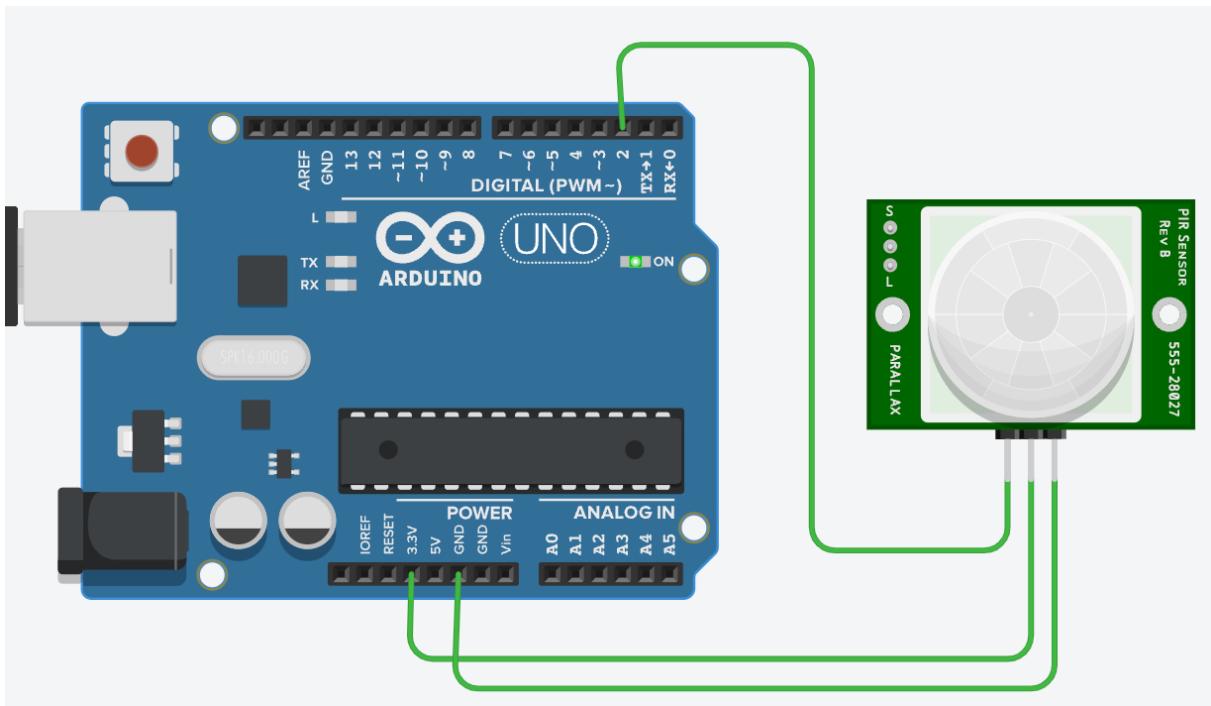
### 3.3.1 Detecting presence

One of the most common applications of sensors is detecting the presence or absence of a person in a room. This is a fundamental task in many systems, from industrial automation to consumer electronics. Sensors used for presence detection are often designed to respond to specific stimuli, such as light, sound, pressure, or motion. The choice of sensor depends on the environmental conditions in which the detection is performed. In this example, we exploit a **Passive Infrared PIR sensor**, which is a device that **captures changes in infrared radiation emitted by objects** in its field of view. When a person enters the sensor range, their body heat causes a change in the infrared radiation detected by the sensor, triggering an output signal. The PIR sensor is sensitive to changes in temperature, making it ideal for detecting the presence of warm-blooded animals, such as humans. In particular we will use the AM312 PIR sensor (see *01.DetectingMotion* folder). The AM312 sensor consists of the following main components:



The **lens** on the outside of the sensor expands the detection area of the sensor by focusing infrared radiation from multiple directions onto the sensor. Then the **infrared filter** allows only IR radiation in the 8–14  $\mu\text{m}$  wavelength range (typical for human body heat) to reach the **pyroelectric transducer**, which exploit the properties of some crystals to generate an electric charge when they are heated or cooled. This **sensitive element** converts the infrared radiation into an electrical signal. The raw signal generated is weak and requires **amplification** and **filtering**. The processed signal is sent to the output pin, which provides a digital output: HIGH (e.g., 3.3V or 5V, depending on the supply voltage) when motion is detected or LOW (0V) when no motion is detected.

In the Makers Kit we have a PIR sensor that we can connect to the Arduino board to detect presence:



```
// define the pin that the sensor is attached to
int pir_pin = 2;

// a variable to count the number of times the sensor has been triggered
int count = 0;

void setup() {
    Serial.begin(9600);
    pinMode(pir_pin, INPUT);
}

void loop() {

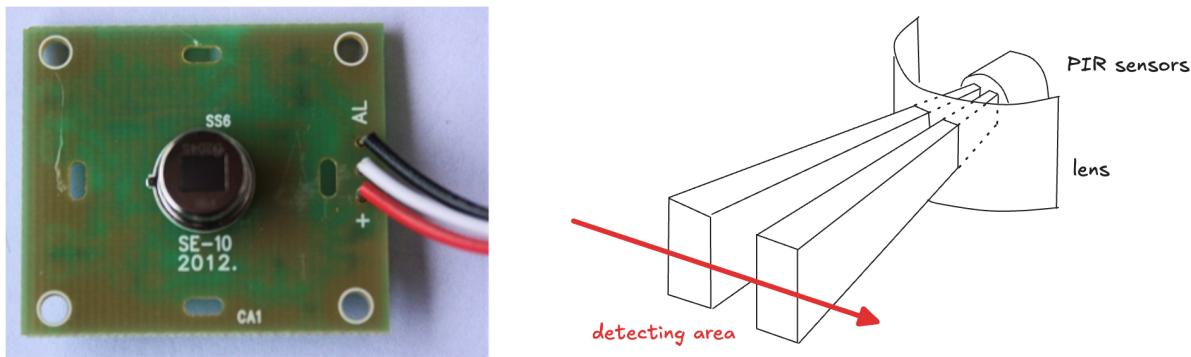
    // detect the output of the sensor
    int pir_val = digitalRead(pir_pin);

    // if the sensor is triggered, increment the count
    if(pir_val == HIGH) {
        count++;

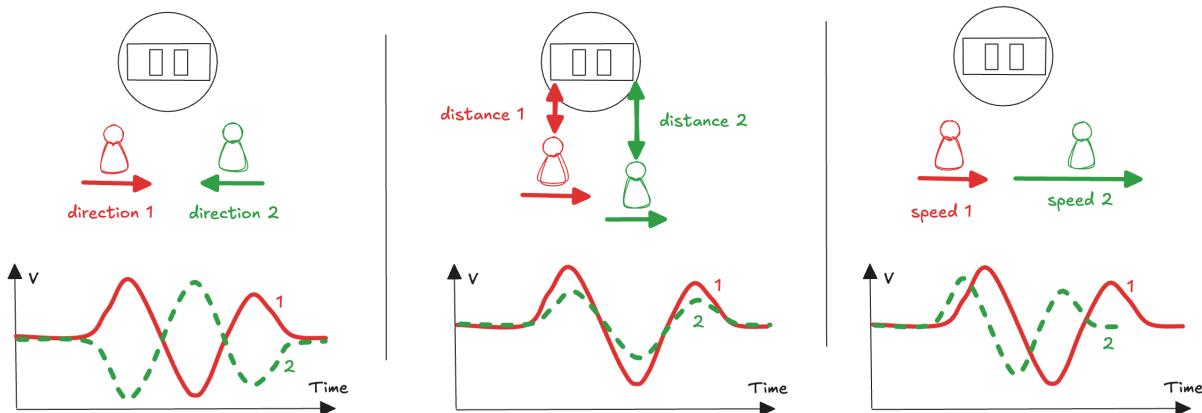
        // send the count to the serial port
        Serial.print("Motion Detected: ");
        Serial.println(count);
    }
}
```

```
// The sensor needs 2 seconds before it can detect motion again
delay(2000);
}
}
```

We can also leverage the analog output of the PIR sensor to gain more detailed insights, not only about presence but also about movement. As an example, consider the following scientific paper "Human Movement Detection and Identification Using Pyroelectric Infrared Sensors" investigates human movement identification using a pair of orthogonally aligned PIR sensors with modified lenses:



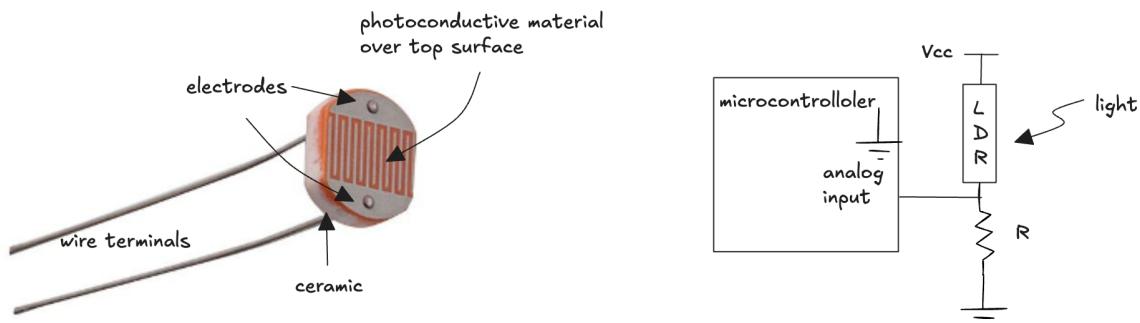
The researcher collect raw data of walking in different directions, at different distances and at different speed levels:



Classification analysis was conducted using machine learning algorithms demonstrated the possibility of classify movement direction, speed, and distance intervals, as well as identifying individuals with high accuracy.

### 3.3.2 Detecting Light

A simple and effective method for detecting light involves using a **Light Dependent Resistor (LDR)**, also known as a **photoresistor**. This sensor operates using semiconductor materials, such as Cadmium Sulfide (CdS), whose electrical resistance changes with varying light intensity. The underlying principle is **photoconductivity**: when light strikes the material, its energy excites electrons from the valence band to the conduction band. This process increases the material's conductivity as light intensity rises, thereby decreasing its resistance. By integrating the LDR into a **voltage divider circuit**, it produces a variable voltage output that can be read by the analog input of a microcontroller. In the Makers Kit we have a LDR sensor that we can connect to the Arduino board to detect light:

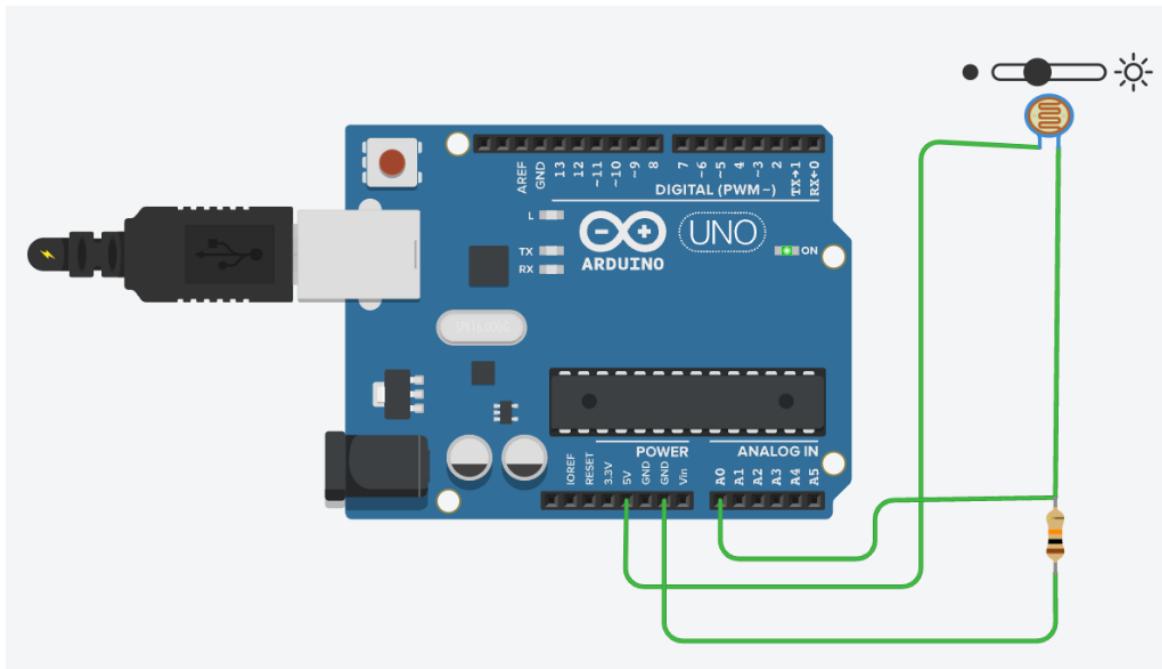


The voltage across either the LDR or the resistor can be determined using the following formula:

$$V_{out} = V_{dd} \cdot \frac{R}{LDR + R}$$

The value of R has to be defined by the user, and it is typically chosen to be in the same order of magnitude as the LDR resistance in the dark. The LDR resistance can vary significantly depending on the light intensity, ranging from a few KOhms in bright light to several MOhms in darkness.

The code for reading the LDR sensor reading is simple as reading from an analog pin (see *02.DetectingLight* sketch):



```
// select the input pin for the photoresistors
const int pin = 0;

// variable to store the value coming from the sensor
int val = 0;

void setup() {
    Serial.begin(9600);
}

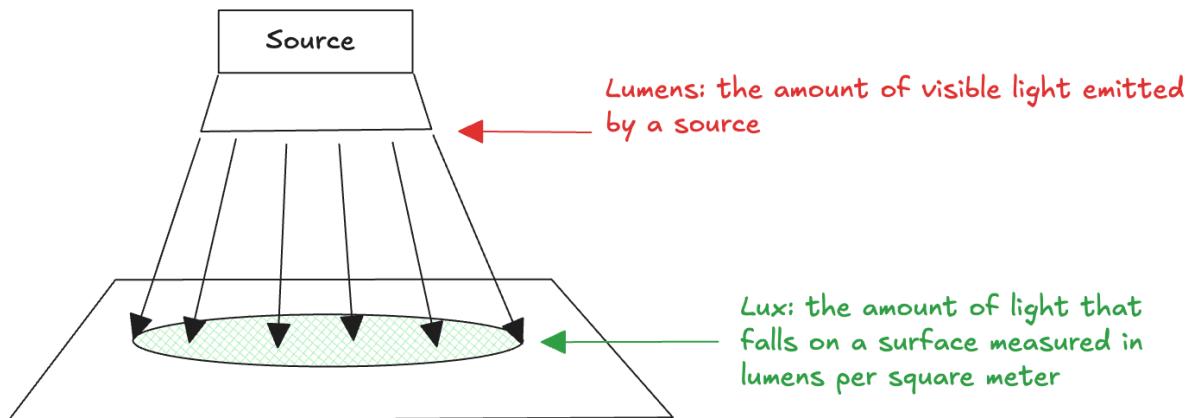
void loop() {
    // read the voltage on the photoresistor
    val = analogRead(pin);

    // print the value to the serial port
    Serial.println(val);
}
```

The LDR is a simple and cost-effective sensor for detecting light levels, making it suitable for applications like automatic lighting control, light intensity measurement, and light-sensitive alarms. However, it has several limitations, since the relationship between light intensity and resistance is **highly non-linear**, which means it can be difficult to directly correlate resistance

values with light intensity without additional calibration or compensation. Another limitation is the **spectral sensitivity**. Photoresistors are typically responsive only to a specific range of wavelengths, usually within the visible spectrum. This makes them unsuitable for measuring light outside this range, such as ultraviolet or infrared light, which is important in some applications that require broader spectrum measurements. Additionally, photoresistors have **slower response times**, which can be problematic in situations where light intensity changes rapidly, as it may fail to track such fluctuations accurately. Finally, another issue is the **sensitivity to temperature**. The resistance of a photoresistor can change with temperature, introducing errors when measuring light in environments where temperatures vary significantly.

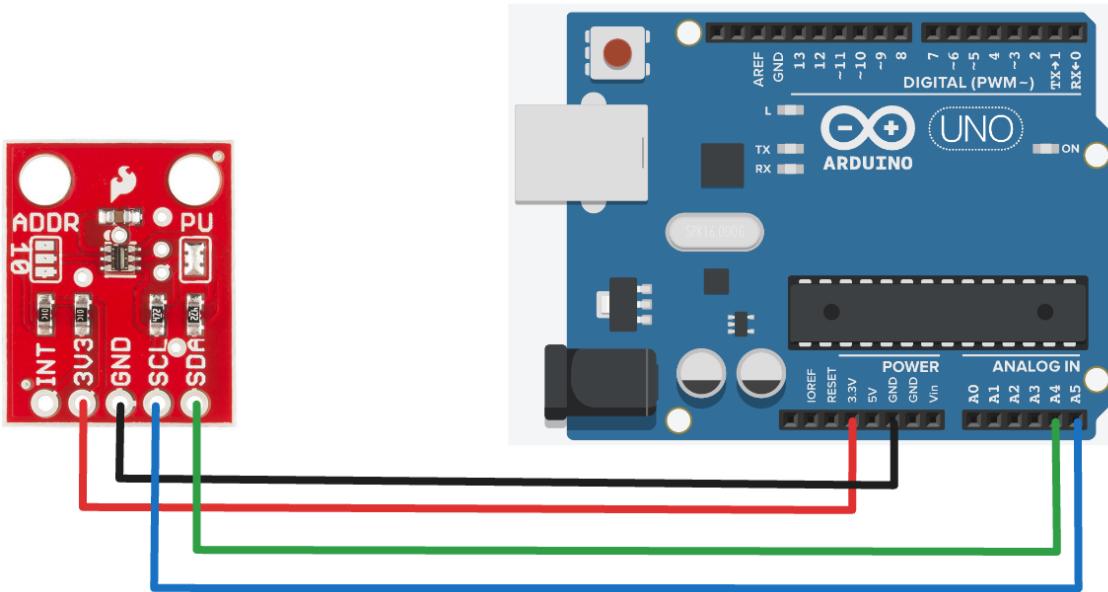
For accurate light measurement in terms of **lumens**, which quantify the total visible light emitted by a source, more specialized sensors, are required. Lumen is often quantified in terms of **lux** when considering its intensity per square meter:



Lux is the unit that describes how bright a light source will appear to the human eye, factoring in both the total light output and the area over which it is distributed. The human eye is capable of perceiving a wide range of light intensities, from as low as 0.0001 lux in starlight, up to over 100,000 lux in direct sunlight. This is a vast range, and it is difficult to build a sensor with the same sensitivity across the entire spectrum, which can be very useful in applications like photography.

In the Makers Kit, we have a TSL2561 sensor. Although it is not a real luxmeter (manufacturer has characterized its output against professional equipment to produce lux approximation equations), it is designed to handle the vast differences in light levels by incorporating features that mimic the characteristics of a camera. For example, the sensor has a **sensitivity setting** that is similar to the ASA film rating, allowing it to adjust its response to light levels. Additionally, it includes an **integration time**, which is akin to a camera's shutter speed, helping to control how long the sensor "exposes" itself to light. By adjusting the sensitivity and integration time, the TSL2561 can effectively measure light in both very dim environments, such as starlight, and

extremely bright conditions, like direct sunlight. It is a digital sensor that communicates with the microcontroller using the **I2C protocol**:



The code to setup and interact with the sensor is provided by the manufacturer with the SparkFunTSL2561 library that can be easily integrated into the Arduino IDE (see the *03.DetectingLight* sketch):

```
// Include the TSL2561 sensor library
#include <SparkFunTSL2561.h>

// Include the I2C library for communication
#include <Wire.h>

// Create an SFE_TSL2561 object to represent the light sensor
SFE_TSL2561 light;

// Global variables to store settings
// Gain setting: 0 = X1 (low gain), 1 = X16 (high gain)
// Integration ("shutter") time in milliseconds
boolean gain;
unsigned int ms;

void setup() {
    Serial.begin(9600);

    // Initialize the TSL2561 sensor.
```

```
// If no I2C address is provided, it uses the default (0x39).
if (!light.begin()) {
    Serial.println("Error: Sensor initialization failed!");
    while (1);
}

// Configure gain
// gain = 0, device is set to low gain (1X)
// gain = 1, device is set to high gain (16X)
gain = 0;

// Configure integration time
// intgration_time = 0, integration will be 13.7ms (short integration time)
// intgration_time = 1, integration will be 101ms (medium integration time)
// intgration_time = 2, integration will be 402ms (long integration time)
// intgration_time = 3, use manual start / stop to perform your own integration
int intgration_time = 2;

// Set the gain and integration time, the library will provide
// the timing for the integration time in ms
light.setTiming(gain, intgration_time, ms);

// Power up the sensor to begin measurements
Serial.println("Powering up the sensor ... ");
light.setPowerUp();
}

void loop() {

    // Read the light sensor data (visible and infrared readings)
    unsigned int data0, data1;
    if (light.getData(data0, data1)) {

        // Print raw sensor data
        Serial.print("Data0 (Visible Light): ");
        Serial.print(data0);
        Serial.print(" Data1 (Infrared Light): ");
        Serial.println(data1);

        // Calculate the lux (light intensity) based on the sensor readings
        double lux;
        boolean good = light.getLux(gain, ms, data0, data1, lux);
    }
}
```

```
// Print the lux value and indicate if the calculation was successful
Serial.print("Lux: ");
Serial.print(lux);
if (good)
    Serial.println(" (Good measurement)");
else
    Serial.println(" (Bad measurement: Sensor may be saturated)");

}

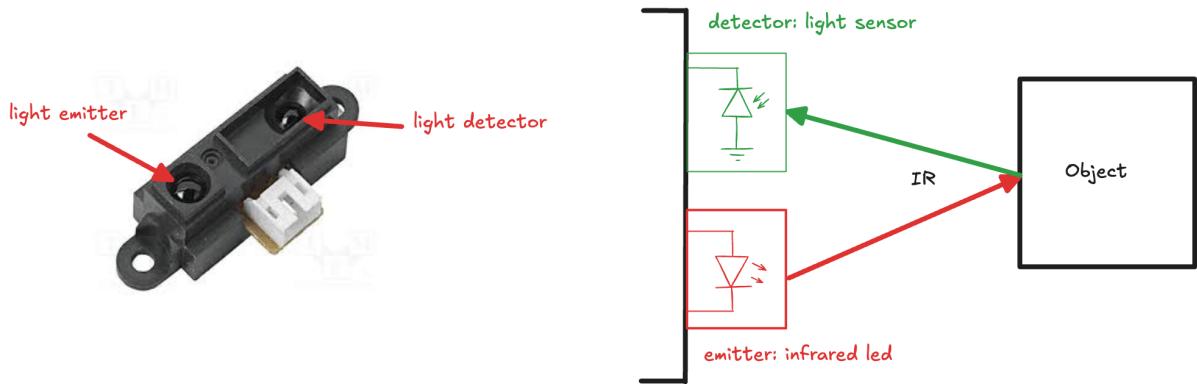
// If data retrieval fails, print the error
else {
    byte error = light.getError();
    switch (error) {
        case 0:
            Serial.println("Success");
            break;
        case 1:
            Serial.println("Data too long for transmit buffer");
            break;
        case 2:
            Serial.println("Received NACK on address (disconnected?)");
            break;
        case 3:
            Serial.println("Received NACK on data");
            break;
        case 4:
            Serial.println("Other error");
            break;
        default:
            Serial.println("Unknown error");
    }
}

// Delay before the next measurement cycle
delay(1000);
}
```

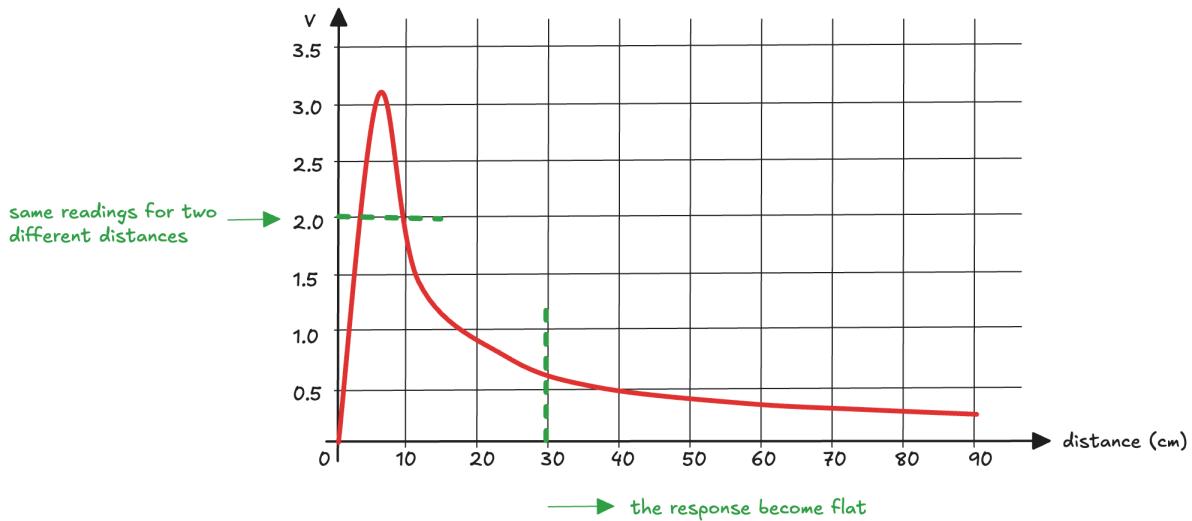
### 3.3.3 Measuring distance

In order to measure the distance of our embedded system from objects in the surrounding environment, we can use a **infrared (IR) sensor**. It works by using a specific light sensor to detect

a select light wavelength in the infrared spectrum. By using a led which produces light at the same wavelength as what the sensor is looking for, it can look at the intensity of the received light. When an object is close to the sensor, the light from the led bounces off the object and into the light sensor. This results in a large jump in the intensity, which can be used to determine the distance of the object from the sensor. Another possibility is to measure the time it takes for the light to bounce back to the sensor, which is the principle behind **Time-of-Flight (ToF)** sensors. These sensors emit a light pulse and measure the time it takes for the pulse to return after reflecting off an object. By knowing the speed of light, the sensor can calculate the distance to the object. In the Makers Kit we have the Sharp GP2Y0A41SK0F sensor:



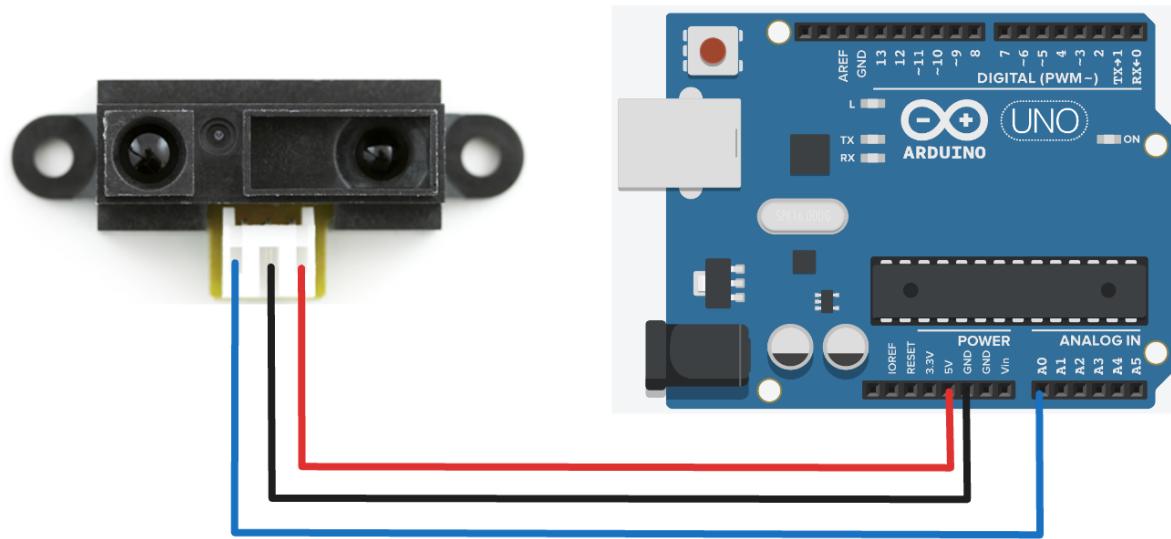
However, the response is non-linear and even worse, it is not monotonic, which means that the output voltage can decrease as the distance increases:



This limitation poses a challenge when using the sensor to measure distance accurately. To address the non-linearity, a **lookup table** can be employed to map the sensor's output to actual distances. The lookup table is created by recording the sensor's output at various known distances and interpolating the values between these points. However, certain distances produce

the same output voltage. This overlap means the sensor cannot differentiate between these distances, limiting its effectiveness for objects closer than 4 cm. Additionally, beyond 30 cm, the sensor's response becomes flat, rendering it unable to detect objects farther than this distance.

We can consider the code to read the output from the analog IR sensor and convert it into a corresponding distance in centimeters, using a lookup table with interpolation (see *04.MeasuringDistance* sketch):



```
// Analog pin connected to the sensor
const int SENSOR_PIN = A0;

// Reference voltage in millivolts (5V for most Arduino boards)
const long REFERENCE_MV = 5000;

// Distance table configuration
const int TABLE_ENTRIES = 12; // Number of entries in the distance table
const int INTERVAL = 250; // Millivolts separating each table entry

// Distance in cm corresponding to each 250 mV interval
static int distanceTable[TABLE_ENTRIES] = {150, 140, 130, 100, 60, 50, 40, 35, 30,
                                         ↓ 25, 20, 15};

// Function to calculate the distance based on the sensor's output voltage
int getDistance(int millivolts) {
    // If the millivolts exceed the last table entry, return the last value
    if (millivolts ≥ INTERVAL * TABLE_ENTRIES) {
        return distanceTable[TABLE_ENTRIES - 1];
    }
}
```

```
}

// Calculate the index of the table entry
int index = millivolts / INTERVAL;

// Calculate the fractional part for interpolation
float fraction = (millivolts % INTERVAL) / (float)INTERVAL;

// Perform linear interpolation between the current and next table entries
return distanceTable[index] - ((distanceTable[index] - distanceTable[index + 1])
→ * fraction);
}

void setup() {
    Serial.begin(9600);
}

void loop() {
    // Read the analog value from the sensor
    int sensorValue = analogRead(SENSOR_PIN);

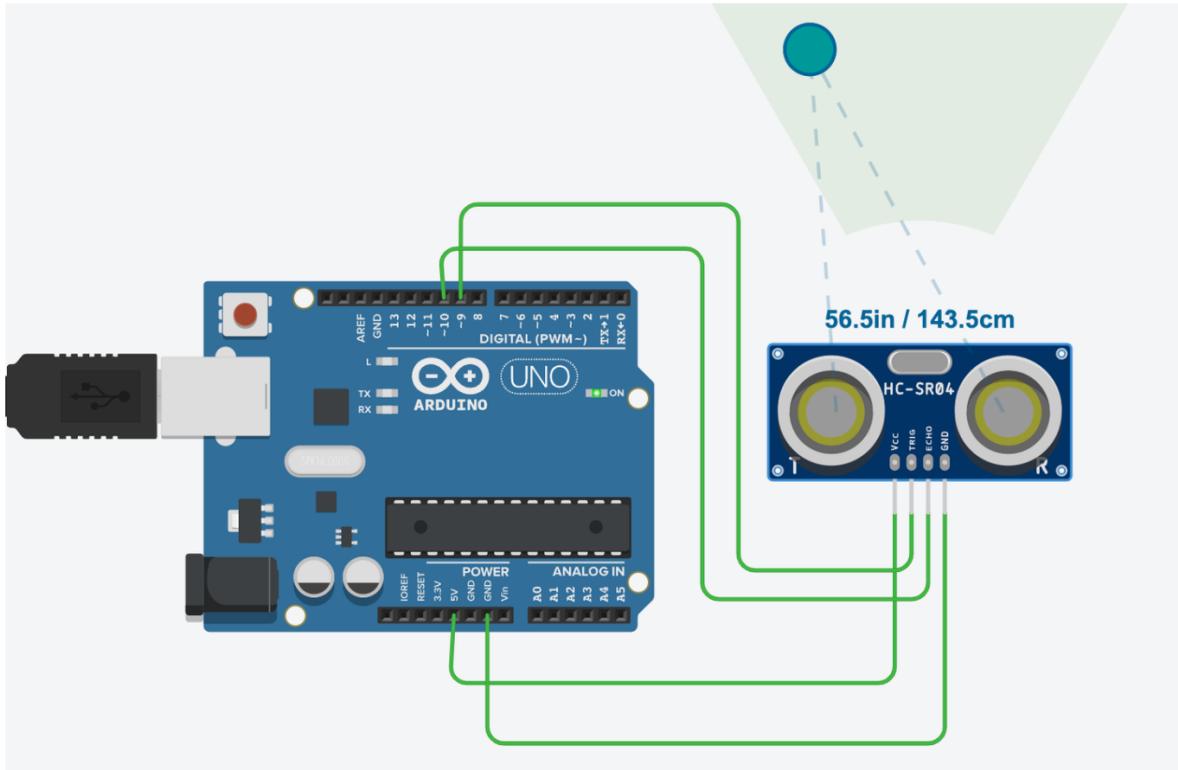
    // Convert the analog value to millivolts
    int millivolts = (sensorValue * REFERENCE_MV) / 1023;

    // Calculate and print the corresponding distance
    int distance = getDistance(millivolts);
    Serial.print(distance);
    Serial.println(" cm");

    // Wait before the next measurement
    delay(100);
}
```

An alternative approach is to use an **ultrasonic sensor**, which determines distance by emitting ultrasonic sound waves and measuring the time it takes for the waves to bounce back after hitting an object. By knowing the speed of sound, the sensor calculates the distance based on the round-trip time of the sound waves. Ultrasonic sensors offer greater accuracy and reliability across a wide range of distances, typically from a few centimeters to several meters. They are particularly suited for applications requiring precise distance measurements, as they perform well in challenging environments such as dusty, foggy, or misty conditions where optical sensors might face difficulties. However, their accuracy can be influenced by factors such as the texture

and angle of the object's surface, which may result in measurement errors. Additionally, their response time is slower compared to infrared sensors, making them less ideal for high-speed applications. In the Makers Kit we have the HC-SR04 sensor, it has two pins for triggering the ultrasonic pulse and receiving the echo:



The microcontroller sends a  $10 \mu\text{s}$  HIGH signal to the Trigger pin to activate the ultrasonic transmitter, which emits 8 cycles of a 40 kHz ultrasonic wave. The sensor waits for the echo to return, to calculate the time we can use the `pulseIn()` function. Finally, the time it takes for the echo to return is proportional to the distance between the sensor and the object, knowing the speed of sound in air, we can calculate the distance:

$$d = \frac{t \cdot v}{2}$$

The division by 2 is necessary because the measured time is for the round trip (to the object and back), so the actual distance is half of that time. This code demonstrates how to read the distance (see *05.MeasuringDistance\_sound* sketch):

```
// Define the pins for the ultrasonic sensor
const int trigPin = 9;
const int echoPin = 10;
```

```
// Define variables for the duration and distance
float duration, distance;

void setup() {
    pinMode(trigPin, OUTPUT);
    pinMode(echoPin, INPUT);
    Serial.begin(9600);
}

void loop() {
    // Send a 10 µs pulse to the trigger pin
    digitalWrite(trigPin, LOW);
    delayMicroseconds(2);
    digitalWrite(trigPin, HIGH);
    delayMicroseconds(10);
    digitalWrite(trigPin, LOW);

    // Measure the duration of the echo pulse
    duration = pulseIn(echoPin, HIGH);

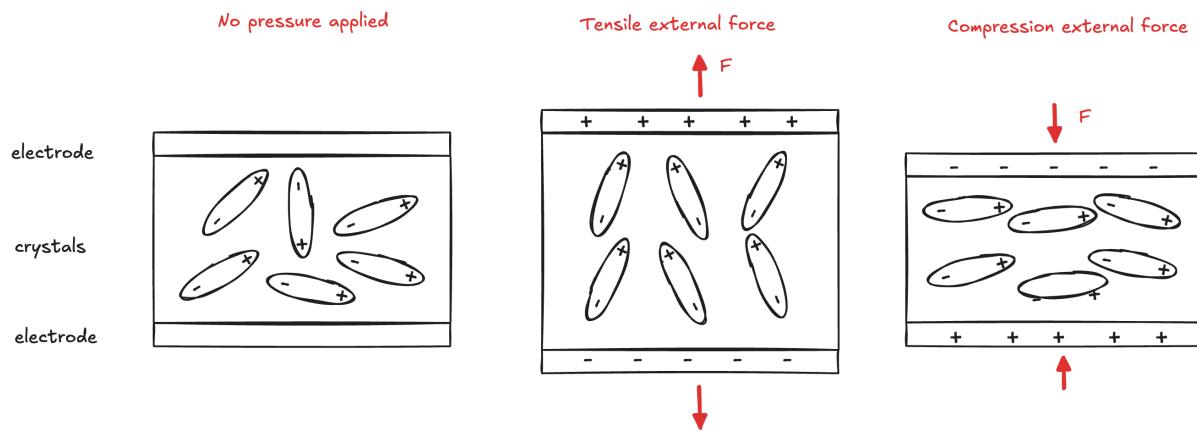
    // Calculate the distance based on the speed of sound
    distance = (duration*.0343)/2;

    // Print the distance to the serial monitor
    Serial.print("Distance: ");
    Serial.println(distance);

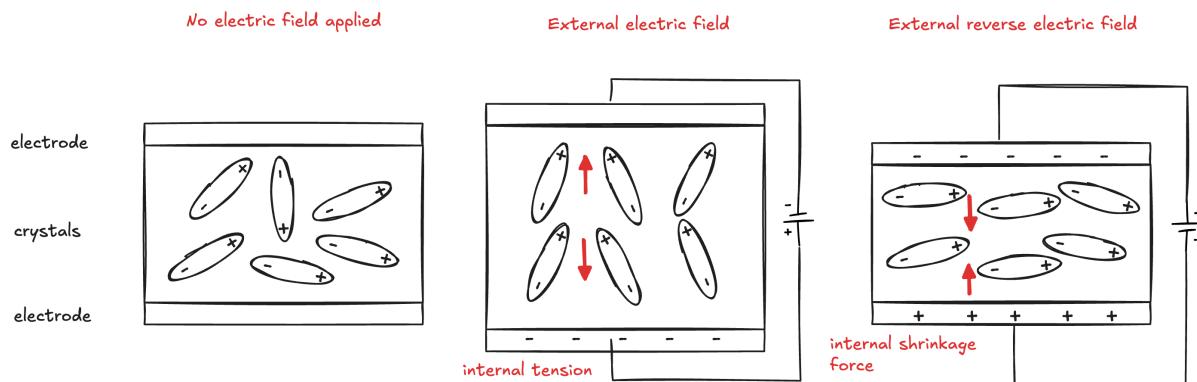
    // Delay before the next measurement
    delay(100);
}
```

### 3.3.4 Detecting vibration

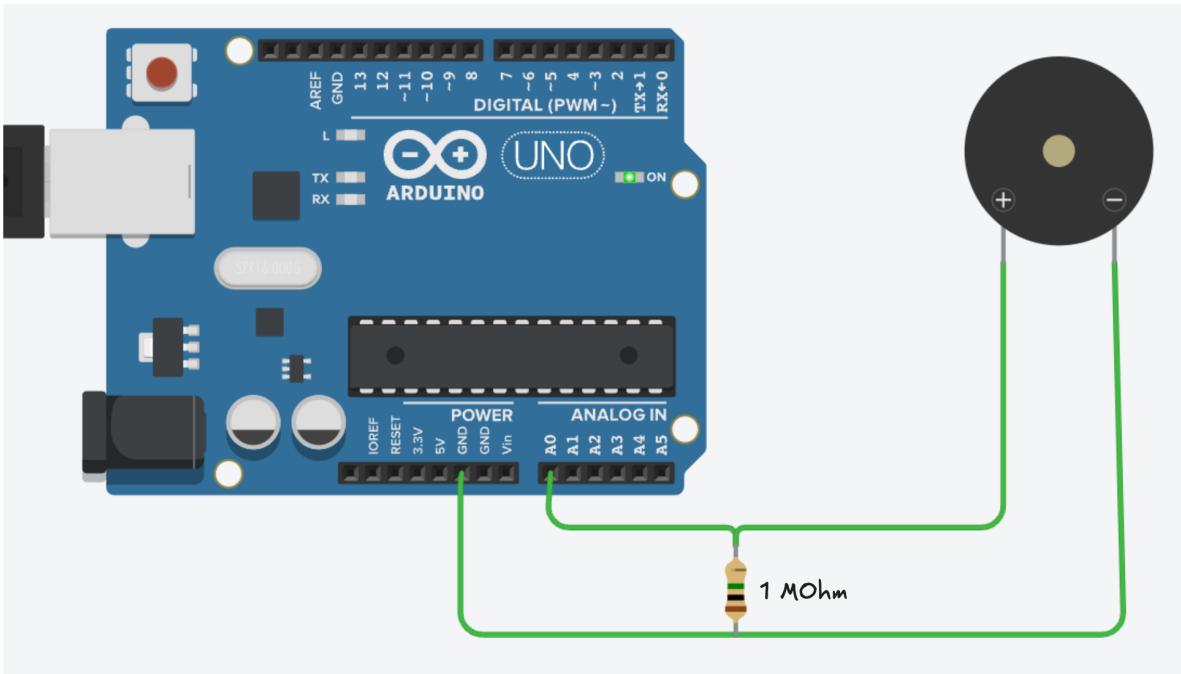
The **piezoelectric effect** is a phenomenon in which certain materials, such as quartz, ceramics, or specific polymers, generate an electric charge when subjected to mechanical stress. When an external force is applied in a specific direction, it causes a redistribution of electrical charges within the material, creating electric polarization. This results in an accumulation of charge on the material surfaces. Once the external force is removed, the material returns to its uncharged state. The polarity of the generated charge is aligned with the direction of the applied force, and the amount of charge produced is directly proportional to the magnitude of the applied stress.



The inverse piezoelectric effect refers to the phenomenon in which the application of an alternating electric field to a crystal induces mechanical deformation and vibration, also known as the electrostrictive effect. When the frequency of the applied alternating voltage matches the crystal natural frequency (which typically depends on the crystal's size), the amplitude of the mechanical vibrations increases significantly. This phenomenon is referred to as piezoelectric resonance:



This effect can be used in various types of sensors, including pressure sensors, accelerometers, and microphones. They can detect mechanical changes (e.g., pressure, vibrations) and convert them into an electrical signal. As an example, on the Makers kit we have a PKM22EPP-40, a simple buzzer (a piezoelectric device that generates sound by vibrating in response to an electrical signal). We can use it to detect vibrations by reversing the process: instead of applying an electrical signal to generate vibrations, we can detect vibrations by reading the electrical signal generated by the buzzer:



The buzzer generates an electrical signal when it is subjected to mechanical vibrations. However, when the buzzer is not vibrating, the analog pin might be left floating (no connection to either power or ground) and this can lead to unpredictable readings due to noise or interference. The resistor connects the analog pin to ground to ensure that when the buzzer is not generating a signal, the analog pin reads a low voltage. The code to read the output from the buzzer and detect vibrations when the signal exceeds a certain threshold (see *06.DetectingVibration* sketch):

```
// The analog pin connected to the sensor
const int sensorPin = 0;

// Variable to count the number of vibrations detected
int count = 0;

// Threshold value for vibration detection
const int THRESHOLD = 10;

void setup() {
    Serial.begin(9600);
}

void loop() {
    // Read the sensor value
    int sensorValue = analogRead(sensorPin);
```

```
int val = analogRead(sensorPin);

// If the sensor value exceeds the threshold, increment the count
if (val > THRESHOLD){
    count++;

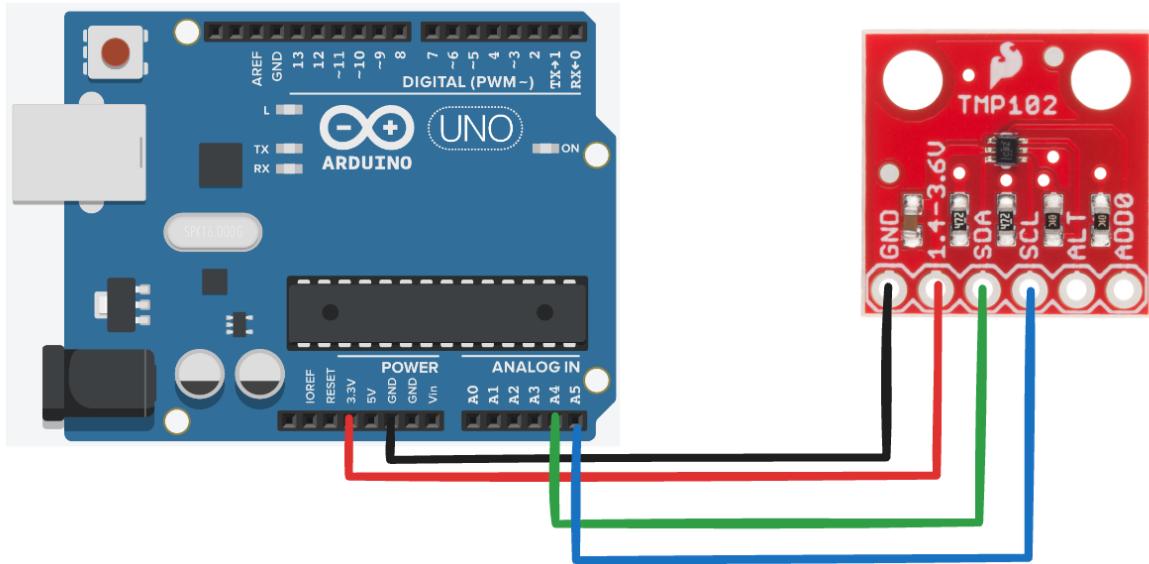
    // Print the number of vibrations detected
    Serial.print("Vibration detected: ");
    Serial.println(count);

    // Delay to prevent multiple detections
    delay(200);
}

}
```

### 3.3.5 Measuring temperature

A **thermal sensor** operates based on the principle that the properties of certain materials change with temperature. Specifically, the bandgap of a semiconductor varies with temperature, and this variation can be harnessed to produce a corresponding voltage or resistance signal. In the Makers kit we have the low-power digital TMP102 sensor. It includes a temperature sensor diode as the primary sensing element, which detects temperature changes. The analog signal from the diode is converted into a digital signal by a 12-bit analog-to-digital converter, providing a resolution of  $0.0625^{\circ}\text{C}$  and an accuracy of  $\pm 0.5^{\circ}\text{C}$ , operating within a temperature range of  $-25^{\circ}\text{C}$  to  $+85^{\circ}\text{C}$ . Control logic manages the sensor's operation, including power modes and communication protocols. The sensor supports I<sup>2</sup>C, SMBus, and two-wire interfaces for easy integration with microcontrollers and other digital systems (see *07.MeasuringTemperature* sketch):



```
#include <Wire.h>

// Define the I2C address of the TMP102 sensor
int tmp102Address = 0x48;

// A function to read the temperature from the TMP102 sensor
float getTemperature() {

    // Request 2 bytes of data
    Wire.requestFrom(tmp102Address, 2);

    // Read the most significant and the least significant bytes
    byte MSB = Wire.read();
    byte LSB = Wire.read();

    // Combine the two bytes into a 12-bit integer
    // The temperature is a 12-bit value, using two's complement
    // for negative values
    int TemperatureSum = ((MSB << 8) | LSB) >> 4;
    float celsius = TemperatureSum * 0.0625;

    // Return the temperature in Celsius
    return celsius;
}
```

```

void setup() {
    Serial.begin(9600);
    Wire.begin();
}

void loop() {

    // Get the temperature in Celsius
    float celsius = getTemperature();

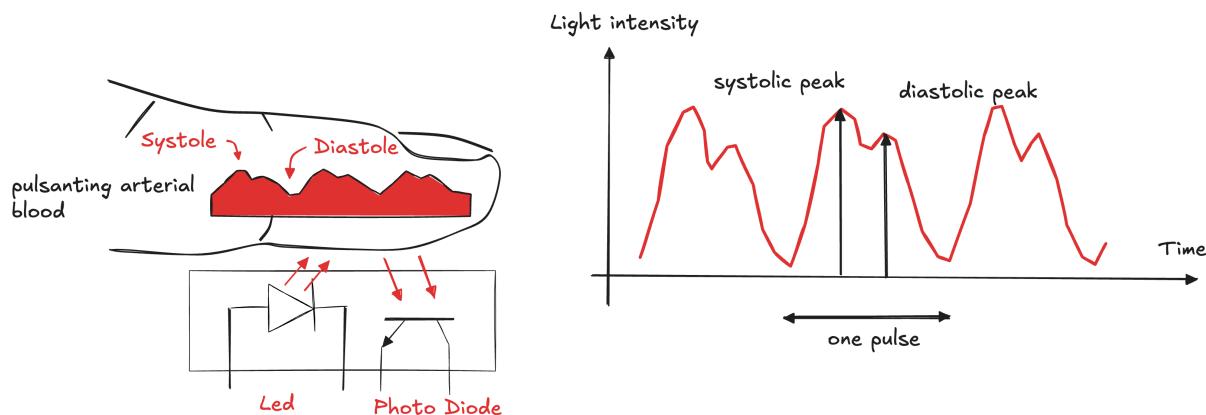
    // Print the temperature to the serial monitor
    Serial.print("Celsius: ");
    Serial.println(celsius);

    // Delay to slow down the output
    delay(200);
}

```

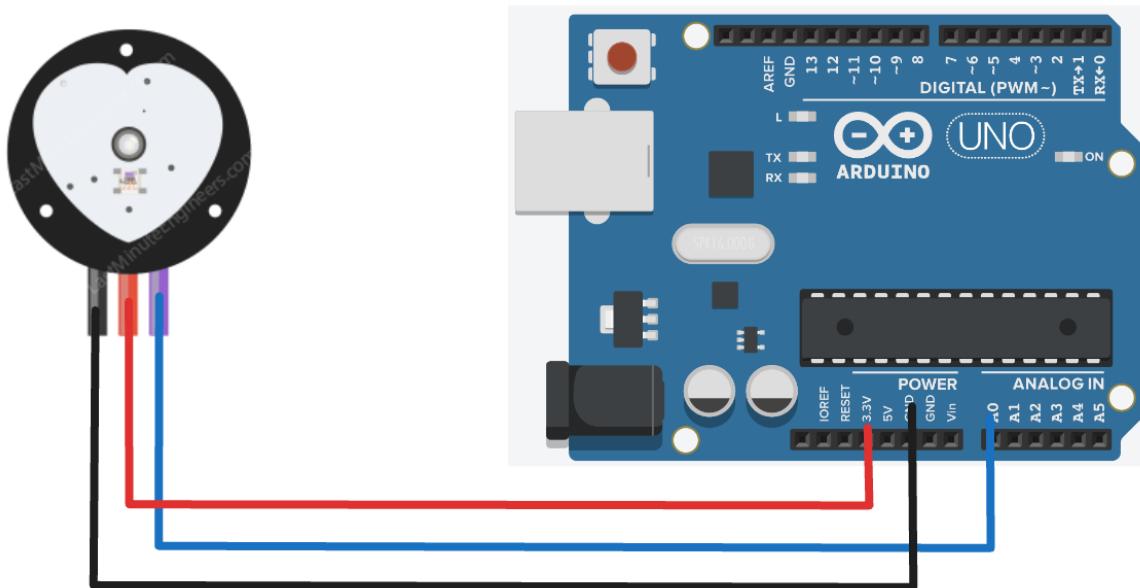
### 3.3.6 Measuring physiological signals

A **photoplethysmograph** (PPG) is a non-invasive optical sensor designed to detect changes in blood volume within the microvascular tissue. It works by emitting light from a source, typically a red or infrared led, onto the skin. As blood absorbs light differently depending on its volume and oxygenation level, the light that is either reflected or transmitted through the tissue varies with each heartbeat. This variation is captured by a photodetector, such as a photodiode, positioned to measure the changes in light intensity:



The signal obtained consists of two main components: a pulsatile signal, which corresponds to the rhythmic blood flow caused by the heartbeats, and a baseline component representing

the steady-state absorption of the surrounding tissue. By processing this signal through amplification and filtering, key physiological parameters such as heart rate, oxygen saturation, and blood pressure trends can be derived. Commonly placed on areas like the fingertip, wrist, or earlobe where blood perfusion is strong, the PPG sensor combines simplicity, efficiency, and reliability, making it a widely used tool in health monitoring and wearable technology. In the Makers kit we have a PulseSensor Kit, a PPG sensor designed to be used with Arduino. We can use it to demonstrate a live visualization of human heartbeat in Arduino Serial Plotter.(see *08.PulseSensor* sketch):



```
// Pin Definitions
const int PulseSensorPin = A0;
const int LEDPin = LED_BUILTIN;

// Variables to store the sensor data and threshold
// to detect a heartbeat (adjust based on your setup)
int Signal = 0;
const int Threshold = 580;

void setup() {
    pinMode(LEDPin, OUTPUT);
    Serial.begin(115200);
}

void loop() {
    // Read the raw signal from the Pulse Sensor
```

```
Signal = analogRead(PulseSensorPin);

// Print the signal value to the Serial Plotter for visualization
Serial.println("Signal: " + String(Signal));

// Check if the signal is above the threshold
if (Signal > Threshold) {
    digitalWrite(LEDPin, HIGH); // Turn on the LED if a heartbeat is detected
} else {
    digitalWrite(LEDPin, LOW); // Turn off the LED otherwise
}

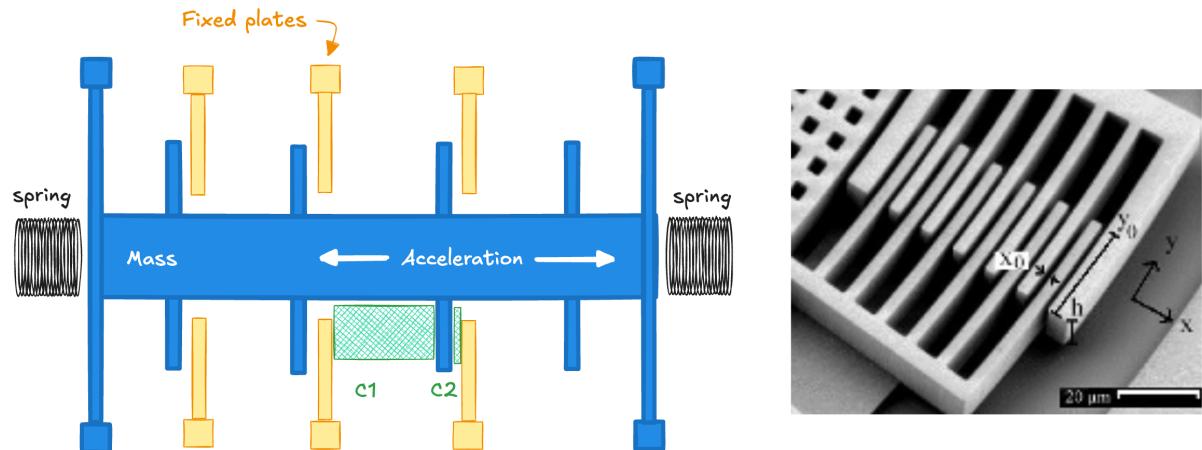
// Add a small delay to stabilize readings (20ms = 50Hz sampling rate)
delay(20);
}
```

### 3.4 Self awareness

The concept of **self-awareness** is fundamental in embedded systems, enabling them to understand their state, monitor performance, and adapt to changing conditions. For instance, a device's **orientation** is critical in applications like navigation, gaming, and virtual reality. By determining its spatial orientation, the device can deliver precise information to users, adjust its behavior dynamically, or seamlessly interact with its environment. In this section we concentrate on how we can support orientation using different source of information and how we can combine them to improve the accuracy of the estimation using **sensor fusion** techniques.

#### 3.4.1 Accelerometers

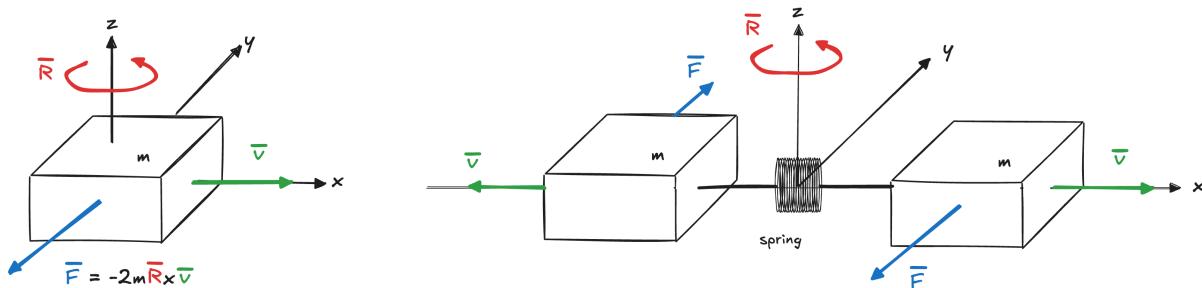
An **accelerometer** measures the total acceleration acting on it, which includes both *dynamic forces from movement* and the **static acceleration caused by gravity**. This allows the device to detect motion as well as orientation relative to the Earth's gravitational field. Internally, an accelerometer consists of tiny capacitive plates. Some of these plates are fixed, while others are attached to microscopic springs, enabling them to move in response to external forces such as gravity, vibrations, or motion. When acceleration occurs, the movable plates shift their position relative to the fixed plates. This movement causes a change in the capacitance, which is the ability of the plates to store electrical charge. By measuring these changes in capacitance, the device can determine the magnitude and direction of the acceleration acting on it.



The measured acceleration is typically expressed in meters per second squared ( $\text{m/s}^2$ ) or in G-forces, where one G-force is equivalent to approximately  $9.81 \text{ m/s}^2$ , the acceleration due to Earth's gravity. Notice the small size of the device, where mechanical elements such as tiny springs, capacitive plates, and moving masses are integrated with electronic circuitry on a single chip, typically fabricated using techniques similar to semiconductor manufacturing. These systems operate at the microscale and this type of device is called **MicroElectroMechanical Systems (MEMS)**. This technology is widely used in applications such as detecting orientation in smartphones, monitoring vibrations in machinery, and enabling motion-sensitive systems in automotive safety features like airbags.

### 3.4.2 Gyroscopes

A **gyroscope** functions by exploiting the **Coriolis effect**, which causes a moving mass to experience a force that deviates its path when subjected to rotational motion. Inside a gyroscope, there is typically a suspended vibrating structure that responds to changes in angular velocity. When the device rotates, the vibrating element is affected by the Coriolis force, resulting in a shift perpendicular to its original vibration.

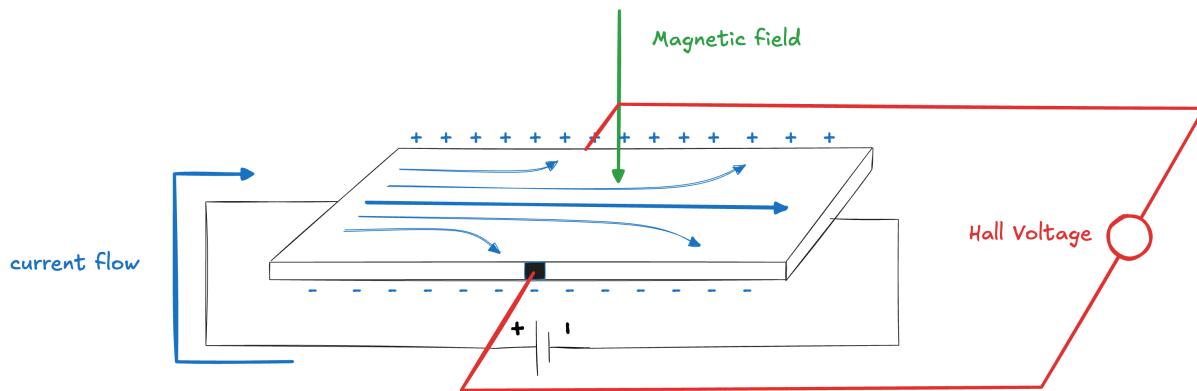


This displacement is minute, but it can be detected with high precision, often through capacitive

or piezoelectric sensors. As the vibrating structure moves, the distance between it and fixed electrodes changes, which in turn alters the capacitance or generates a voltage. This variation is then measured and used to determine the angular velocity of the rotation.

### 3.4.3 Magnetometers

A **magnetometer** measures the strength and direction of a magnetic field using the **Hall Effect**. When a current flows through a thin, conductive plate, the charge carriers move in a straight line from one side of the plate to the other. However, when a magnetic field is applied perpendicular to the direction of the current, it causes the charge carriers to experience a force, known as the **Lorentz force**. This force causes the electrons to be deflected to one side of the plate, while the holes are pushed to the opposite side. As a result, a voltage develops across the width of the plate, perpendicular to both the current and the magnetic field. This voltage is known as the Hall voltage and is proportional to the strength of the magnetic field, allowing it to be used for magnetic field measurements:



This device can be used to detect the Earth's magnetic field, which is relatively constant in direction and strength, providing a reference for orientation.

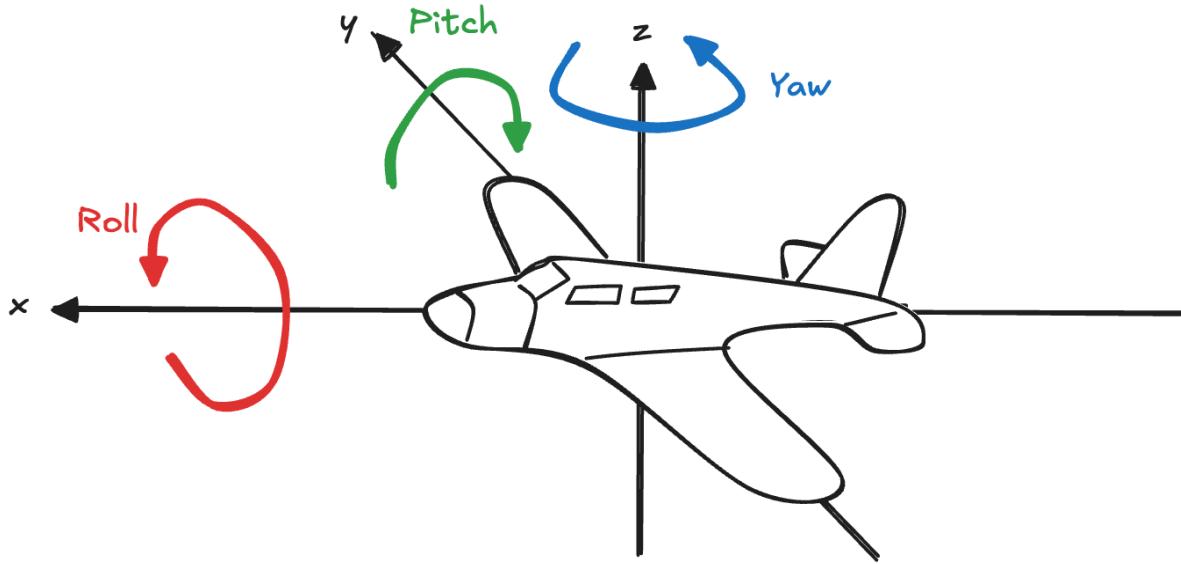
### 3.4.4 Orientation

**Orientation** describes the position of an object in three-dimensional space relative to a reference frame, typically defined using three angles:

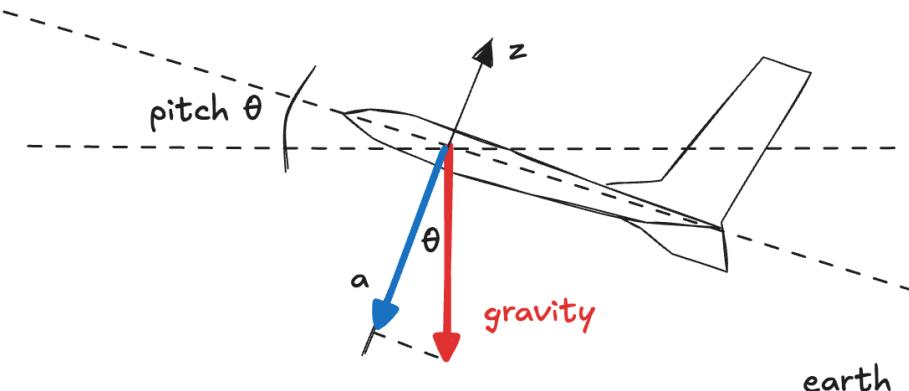
- **Yaw:** The rotation around the **vertical (z-axis)**. It describes how the object turns left or right, similar to a compass direction. For example, a car turning on a flat road is primarily changing its yaw.
- **Pitch:** The rotation around the **lateral (y-axis)**. It describes the up or down tilt of the object, like nodding your head or a plane adjusting its nose up or down during flight.

- **Roll:** The rotation around the **longitudinal (x-axis)**. It describes the tilting motion to the left or right, such as when an airplane banks to turn or a boat rocks sideways.

Together, these angles specify the orientation in 3D space. These angles are often used in applications like navigation, robotics, aerospace, and computer graphics to represent and control the position of an object.



As described, an accelerometer measures acceleration along predefined axes, and this includes both dynamic accelerations (from motion) and the Earth's gravity, which is a constant acceleration of approximately  $9.8 \text{ m/s}^2$ . By leveraging this measurement and in static condition (no motion), we can calculate the orientation of the body relative to the gravitational field. For example, considering only the pitch, with a body perfectly aligned with the horizontal plane, the gravity force is a constant downward force. When the body is tilted, the component of gravity along the z-axis changes according to the angle of tilt:



$$a = -g \cdot \sin(\theta)$$

By measuring the acceleration along the x-axis, we can calculate the pitch angle using the inverse sine function:

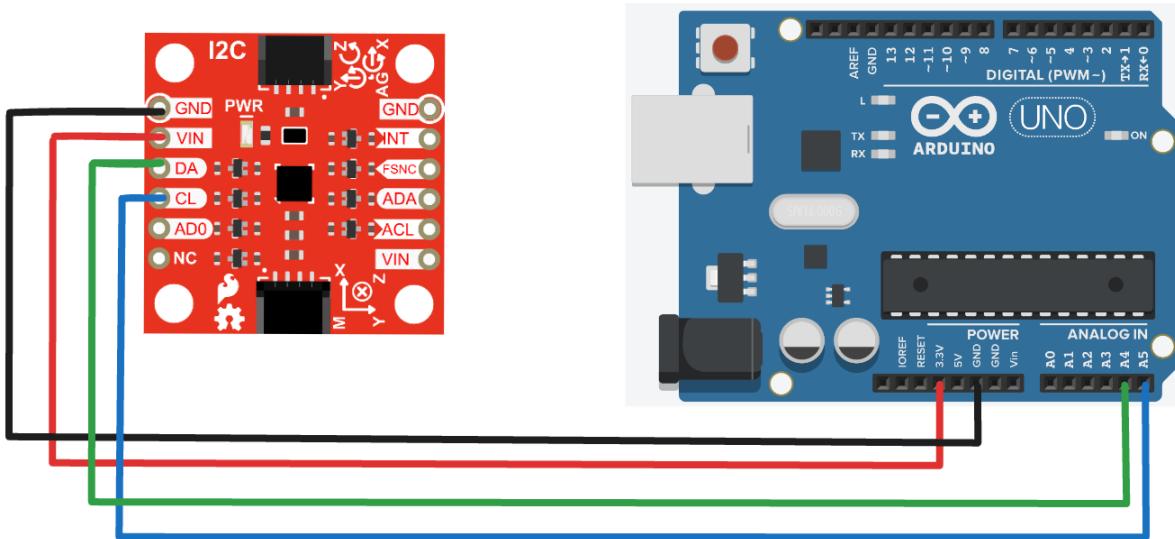
$$\theta = \arcsin\left(\frac{-a}{g}\right)$$

More generally, if the body is not perfectly aligned with the horizontal plane and considering the roll and yaw angles, the orientation can be calculated using the following equations:

$$pitch = \arctan\left(\frac{a_x}{\sqrt{a_y^2 + a_z^2}}\right)$$

$$roll = \arctan\left(\frac{a_y}{\sqrt{a_x^2 + a_z^2}}\right)$$

In the Makers Kit, we have a ICM-20948 sensor featuring a 3-axis accelerometer, a 3-axis gyroscope, and a 3-axis magnetometer, all integrated into a single chip. The ICM-20948 sensor is a **9DOF** (degrees of freedom) IMU (Inertial Measurement Unit). We can use it to calculate the two angles (see *09.Pitch\_Accelerometer* sketch):



We can conduct an experiment to analyze the behavior of the accelerometer when rotated around its y-axis. We held the IMU in one hand and gently rotated first in one direction and then in the opposite direction. The motion is performed carefully, aiming to maintain a smooth and consistent speed throughout the rotation. During the experiment, the pitch angle is recorded by reading the sensor data and sending it to the serial monitor (see *09.Pitch\_Accelerometer*):

```
// Include the sensor library and the I2C library
#include <Wire.h>
#include "ICM_20948.h"

// Create an ICM_20948 object
ICM_20948_I2C myICM;

void setup() {
    Serial.begin(115200);
    Wire.begin();

    // Initialize the ICM-20948
    if (myICM.begin(Wire, 0x68) != ICM_20948_Stat_Ok) {
        Serial.println("IMU initialization failed!");
        while (1);
    }
    Serial.println("IMU initialized successfully!");
}

void loop() {

    // If sensor data ready are available
    if (myICM.dataReady()) {

        // Read accelerometer, gyroscope, and magnetometer data
        myICM.getAGMT();

        // Extract accelerometer data
        float ax = myICM.accX();
        float ay = myICM.accY();
        float az = myICM.accZ();

        // Calculate pitch angle (in degrees)
        float pitch = atan2(-ax, sqrt(ay * ay + az * az)) * 180.0 / PI;

        // Send pitch angle to Serial Monitor
        Serial.print("Pitch: ");
        Serial.println(pitch);
    }

    // Small delay to make output readable
    delay(200);
}
```

The data are collected by a Processing sketch that reads the serial data, appends them to a text file while visualizing the pitch in a 3D box on the screen.

```
import processing.serial.*;
import java.io.BufferedReader;
import java.io.FileWriter;
import java.io.IOException;
import java.text.DecimalFormat;
import java.text.DecimalFormatSymbols;
import java.util.Locale;

// Output filename for storing pitch values
String outFilename = "out.csv";

// Serial object for communication
Serial myPort;

// Variable to store the pitch value
float pitch = 0;

void setup() {
    // Set up the canvas size and 3D renderer
    size(640, 360, P3D);

    // Initialize serial communication with the port
    myPort = new Serial(this, Serial.list()[2], 115200);
    myPort.bufferUntil('\n');
}

void draw() {
    // Clear the screen with a black background
    background(0);

    // Add lighting to the scene
    lights();

    // Apply 3D transformations and rotate the box based on pitch
    pushMatrix();
    translate(width / 2, height / 2, -30); // Move the box to the center of the canvas
    rotateX(radians(pitch)); // Rotate the box on the X axis using the pitch value
    box(100); // Draw the box with a size of 100 pixels
```

```
popMatrix(); // Restore the previous transformation state
}

void serialEvent(Serial myPort) {
    // Read the incoming data until a newline character
    String rpstr = myPort.readStringUntil('\n');

    // Check if the data is not null
    if (rpstr != null) {
        // Split the data into an array using ':' as delimiter
        String[] list = split(rpstr, ":");

        try {
            // Parse the pitch value from the received data
            pitch = float(list[1]);

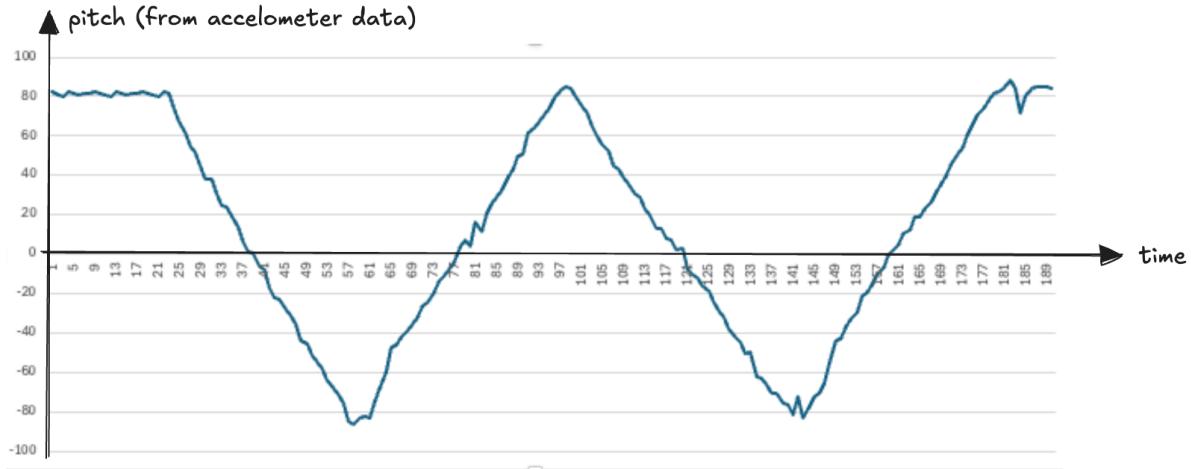
            // Append the formatted output to the file
            appendTextToFile(outFilename, pitch);
        }
        // Catch and ignore any exceptions in case of malformed data
        catch (Exception ex) {}
    }
}

void appendTextToFile(String filename, float pitch) {
    // Set up the locale for countries that use ',' as a decimal separator
    DecimalFormatSymbols symbols = new DecimalFormatSymbols(Locale.ITALY);
    DecimalFormat df = new DecimalFormat("#0.00", symbols);

    // Try to open the file and write the text to it
    try (BufferedWriter out = new BufferedWriter(new FileWriter(dataPath(filename),
        true))) {
        out.write(df.format(pitch)); // Write the pitch to the file
        out.newLine(); // Add a new line after each entry
    }
    catch (IOException e) {
        e.printStackTrace(); // Print any errors that occur while writing the file
    }
}
```

Finally, we can import in a spreadsheet the data collected and plot the pitch angle as a function of time. The plot shows the pitch angle increasing as the sensor is rotated in one direction and

decreasing as it is rotated in the opposite direction. This demonstrate how the IMU detects and reports angular changes when subjected to controlled rotational motion:



The accelerometer is a sensor designed to measure all forces acting on an object, not just gravitational acceleration. This means it detects any external forces, such as vibrations, impacts, or motion-induced forces. As a result, its measurement is **highly sensitive and easily disrupted by even minor disturbances**. For instance, small forces like friction and vibrations from the environment can significantly alter the readings, making it challenging to isolate the specific force of interest (such as gravity or intentional motion).

However, we also have a gyroscope, which measures the **angular velocity** and we can also estimate angular position using gyroscope data. To achieve this, we must compute the **integral of angular velocity over time**. However, continuous integration is not possible in a digital system. Instead, we approximate it by summing a finite number of discrete samples taken at regular intervals:

$$\omega_y(t) = \frac{d}{dt} \text{pitch}$$

$$\$ \text{displaystyle pitch} = \int_0^t \omega_y(t) dt \approx \sum_{i=0}^{n-1} \omega_y(t_i) \cdot T_s$$

We can repeat the experiment using the angular position estimated from gyroscope data (see [10.Pitch\\_Gyroscope](#)):

```
// Include the sensor library and the I2C library
#include <Wire.h>
#include "ICM_20948.h"

// Create an ICM_20948 object
ICM_20948_I2C myIMC;
```

```
// Variable to store the timer for pitch calculation
// usgin integration of gyroscope data
unsigned long timer;

// Variable to store the pitch angle
// initialized to 90 degrees (the first position)
float pitch = 90;

void setup() {
    Serial.begin(115200);
    Wire.begin();

    // Initialize the ICM-20948
    if (myICM.begin(Wire, 0x68) != ICM_20948_Stat_Ok) {
        Serial.println("IMU initialization failed!");
        while (1);
    }
    Serial.println("IMU initialized successfully!");
}

void loop() {

    // If sensor data ready are available
    if (myICM.dataReady()) {

        // Read accelerometer, gyroscope, and magnetometer data
        myICM.getAGMT();

        // Extract gyroscope data
        float vx = myICM.gyrX();
        float vy = myICM.gyrY();
        float vz = myICM.gyrZ();

        // Calculate the pitch angle based on gyroscope Y-axis data
        // This is an integration of the velocity over time to get the angle
        float dt = (double)(millis() - timer) / 1000.0;
        pitch += vy * dt;

        // Update the timer for the next calculation
        timer = millis();

        // Send pitch angle to Serial Monitor
    }
}
```

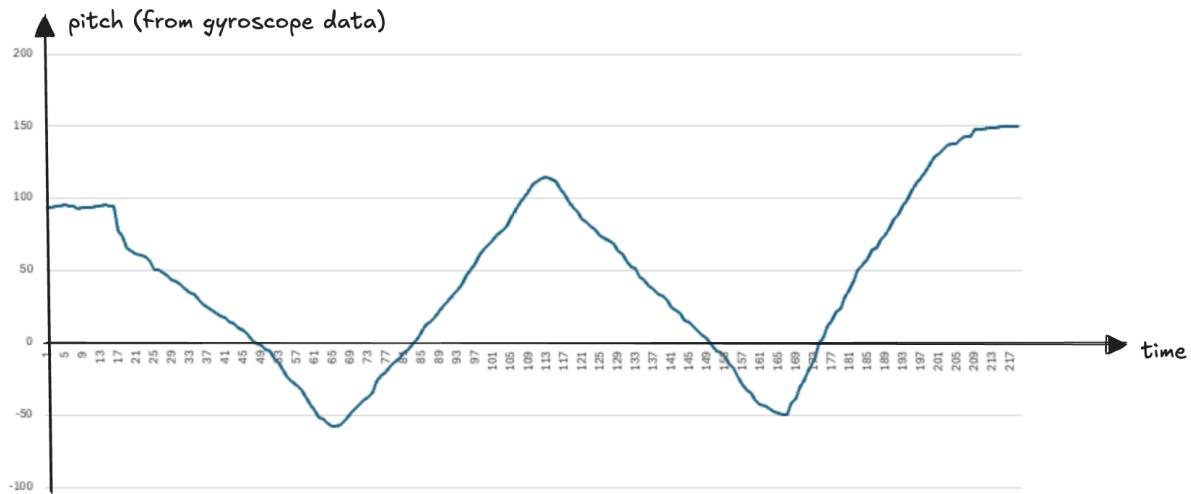
```

    Serial.print("Pitch: ");
    Serial.println(pitch);
}

// Small delay to make output readable
delay(200);
}

```

We can repeat the experiment using the angular position estimated from gyroscope data:



The gyroscope data are much more stable and less sensitive to external disturbances compared to the accelerometer. However, if the angular velocity changes more rapidly than the sampling frequency allows, some variations will go undetected, resulting in an inaccurate integral approximation. Additionally, small inaccuracies in each measurement **accumulate over time**, gradually deviating from the true angular position. This phenomenon, known as **drift**, becomes more pronounced the longer the integration continues.

### 3.4.5 Sensor fusion

Using sensor fusion, accelerometers and gyroscopes complement each other to provide more reliable and accurate data. As mentioned, the accelerometer is sensitive to all forces, including external disturbances like vibrations, which can affect the measurement of orientation. Meanwhile, the gyroscope provides accurate rotational data but tends to drift over time due to the integration process. By **combining** both sensors, we can leverage the gyroscope short-term accuracy and the accelerometer ability to correct drift over time. This approach results in a more

stable and precise estimation of orientation, addressing the individual limitations of each sensor. This can be done using a technique called **complementary filtering**, that combines the two angles, using a weight to control the blend between the accelerometer and gyroscope data. The equation for updating the pitch estimate is:

$$pitch(t) = \alpha \cdot (pitch(t-1) + \omega(t) \cdot T_s) + (1 - \alpha) \cdot \arctan\left(\frac{a_x}{\sqrt{a_y^2 + a_z^2}}\right)$$

The first term represents the predicted angle from the gyroscope. It provides a fast response but can drift over time due to sensor noise. The second term represents the angle based on the accelerometer. It is stable over time but noisy in the short term (e.g., when there is a lot of movement or vibration). The weight determines the balance between the two sources, and its value is usually set empirically, depending on the application. If the system is highly dynamic, a larger value of (e.g., 0.98 or 0.99) will give more weight to the gyroscope, making the system respond quickly to changes. If the system is relatively stable, we can decrease to give more weight to the accelerometer data for more accuracy (see [11.Pitch\\_SensorFusion](#)):

```
// Include the sensor library and the I2C library
#include <Wire.h>
#include "ICM_20948.h"

// Create an ICM_20948 object
ICM_20948_I2C myIMC;

// Complementary filter parameter
const float alpha = 0.8;

// Estimated pitch angle (initialized to 90)
float pitch_acc = 90;
float pitch_gyro = 90;
float pitch = 90;

// Timer variables for delta time calculation
unsigned long timer = 0;

void setup() {
    Serial.begin(115200);
    Wire.begin();

    // Initialize the ICM-20948
    if (myIMC.begin(Wire, 0x68) != ICM_20948_Stat_Ok) {
        Serial.println("IMU initialization failed!");
    }
}
```

```
    while (1);
}

Serial.println("IMU initialized successfully!");

// Initialize time tracking
timer = millis();
}

void loop() {
    // If sensor data is available
    if (myICM.dataReady()) {

        // Read accelerometer, gyroscope, and magnetometer data
        myICM.getAGMT();

        // Extract accelerometer data
        float ax = myICM.accX();
        float ay = myICM.accY();
        float az = myICM.accZ();

        // Compute pitch angle from the accelerometer
        pitch_acc = atan2(-ax, sqrt(ay * ay + az * az)) * 180.0 / PI;

        // Extract gyroscope data
        float vy = myICM.gyrY();

        // Compute time difference ( $\Delta t$ ) in seconds
        float dt = (double)(millis() - timer) / 1000.0;

        // Compute pitch angle from the gyroscope
        pitch_gyro += vy * dt;

        // Compute pitch using the complementary filter
        pitch = alpha * (pitch + vy * dt) + (1 - alpha) * pitch_acc;

        // Update previous time
        timer = millis();

        // Print pitch angle to Serial Monitor
        Serial.print("Pitch: ");
        Serial.print(pitch_acc);
        Serial.print(":");
        Serial.print(pitch_gyro);
```

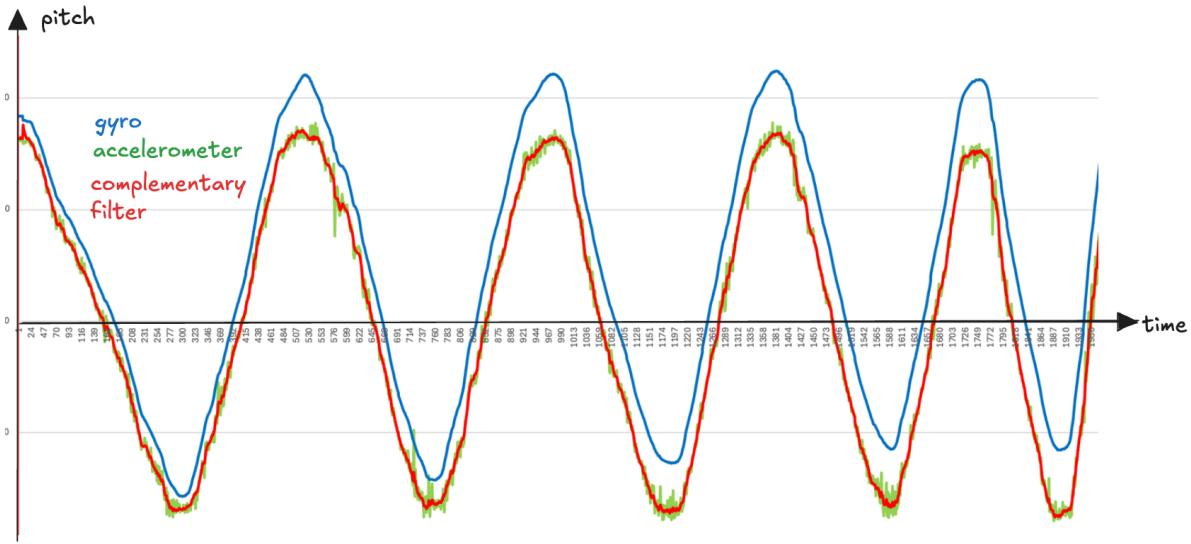
```

    Serial.print(":");
    Serial.println(pitch);
}

// Small delay to make output readable
delay(10); // Reduced delay for smoother angle tracking
}

```

We can repeat the experiment using the sensor fusion approach to estimate the pitch angle and compare it with the accelerometer and gyroscope data:



The graph illustrates how the sensor fusion approach provides a more stable and accurate estimation of the pitch angle compared to using either the accelerometer or gyroscope alone. The accelerometer-based estimate is noisy, while the gyroscope-based estimate tends to drift over time. The complementary filter effectively combines the strengths of both sensors, leveraging the accelerometer's long-term stability and the gyroscope's responsiveness to produce a more reliable orientation estimate. This technique is widely used in applications such as drones, robotics, and virtual reality systems, where precise orientation tracking is essential. A more advanced method for sensor fusion is the **Kalman filter**, an algorithm that estimates the state of a system based on a series of noisy measurements. It recursively updates the state estimate by integrating new measurements with previous predictions, allowing it to handle multiple sensor inputs, compensate for noise and uncertainty, and adapt to changing conditions. However, a detailed discussion of the Kalman filter is beyond the scope of this course.

### 3.5 Hands-on Activity

1 - Adjust the brightness of an LED based on ambient light levels using Arduino **Pulse Width Modulation** (PWM). PWM controls brightness by rapidly switching the LED on and off, varying the proportion of time it stays on (duty cycle). This creates the effect of smooth dimming, as the human eye perceives the average light intensity rather than the rapid flickering. By adjusting the duty cycle in response to ambient light readings, the LED can automatically brighten in the dark and dim in well-lit conditions.

2 - Utilize the pitch and roll data to dynamically rotate a cube in 3D space along the corresponding axes. By continuously updating these values based on sensor readings, the cube will realistically respond to changes in orientation, simulating real-world motion.

3 - Display a real-time chart of heart rate data collected from the heart-rate sensor. Continuously update the graph to visualize heart rate variations over time, allowing for easy monitoring of trends, anomalies, or sudden changes in pulse.

## 4 Javascript

JavaScript was created in 1995 at Netscape Communications, at that time the most popular Internet browser. Initially, it was designed **to add interactivity to web pages**, and it quickly became an essential tool for web development. Over time, JavaScript's capabilities expanded beyond simple client-side scripting, with the introduction of frameworks and libraries, and the development of Node.js, which allowed it to run on servers as well. This made JavaScript **a full-stack language**, growing its popularity even further. Adopted by all major browsers, JavaScript became standardized under the **ECMA specification**, ensuring uniformity and stability across different environments. Interestingly, despite its name, JavaScript has **nothing to do with Java**, the name was largely a marketing decision at the time to capitalize on Java growing popularity.

JavaScript is particularly well-suited for the Web of Things domain due to its versatility, real-time capabilities, and large ecosystem of libraries and tools. For instance, JavaScript can handle both front-end and back-end tasks, making it an ideal choice for handling data from various devices in a unified application. It can interact with hardware through APIs, process data efficiently, and integrate with cloud services, offering seamless communication between connected devices. Additionally, its asynchronous nature, supported by promises and `async/await`, is key in managing multiple IoT devices without blocking operations, making it a powerful tool for the Web of Things.

## 4.1 Basic features

JavaScript supports all the basic functionalities that you would expect from a programming language like C, so much of what you know is still valid. For example, comments in JavaScript are written using the same syntax as in C:

```
// This is a comment
```

Blocks of code are enclosed within curly braces, just like in C. Conditional execution works in the same way, with the basic constructs like "if", "if...else", and "switch...case":

```
if (x > 10) {
    console.log("x is greater than 10");
} else {
    console.log("x is less than or equal to 10");
}
```

To run JavaScript code directly in our browser, such as Google Chrome, we can use the **built-in Developer Tools** on the **Console** tab. This is where we can enter and run JavaScript code using the internal browser JavaScript interpreter.

Loops are also supported, including "while", "do...while", and "for", as well as "for-in" for iterating over objects or arrays:

```
let number = 1;

while (number <= 12) {
    console.log(number);
    number++;
}
```

A fragment of code that produces a value is called an **expression** and a statement, on the other hand, is a **composition of expressions** and ends with a semicolon:

```
// This is an expression that produces a value
let result = 5 * 3;
```

The semicolon used to mark the end of a statement, it is technically optional in many cases, due to a feature called **Automatic Semicolon Insertion**). It allows JavaScript to automatically insert

semicolons where it assumes they should go, so you don't always need to explicitly write them. However, there are cases where ASI might not behave as expected, leading to bugs. To avoid unexpected behavior and make the code clearer to both the browser and other developers, it's generally recommended to always use semicolons to explicitly mark the end of statements.

Variables can be declared using `let`, `const`, or `var` (though `let` and `const` are preferred in modern JavaScript). Functions are also defined similarly to how you would do so in C, allowing us to structure and reuse our code effectively:

```
function add(a, b) {  
    return a + b;  
}
```

So, since you are already familiar with the basics of C, you'll find JavaScript to be quite similar in its structure and functionality.

#### 4.1.1 Variables

Of course there are important differences between C and JavaScript, one of the most notable being the way variables are declared. In C, you would declare a variable by specifying its type, such as int, float, or char. Instead, JavaScript is a **dynamically typed language**, meaning that we don't need to declare the type of a variable explicitly. Instead, the **type is determined by the value assigned to it at runtime**. For example, when we write:

```
let name = value;
```

JavaScript automatically infers the type of `name` based on the type of `value`. So we don't have to declare whether a variable should hold a string, number, or any other type. A key consequence is that **variables can hold values of different types at different times**. For instance, we might assign a string to a variable and later change it to a number:

```
// x is a string  
let x = 'pippo';  
  
// now x is now a number  
x = 5;
```

This flexibility allows for greater ease of use, but it can also lead to **automatic type conversion**, where JavaScript implicitly changes the type of a value in certain operations. This behavior can be **confusing** and lead to unexpected results:

```
console.log('5' - 1); // 4 (string '5' is converted to number)
console.log('5' + 1); // '51' (number 1 is converted to string and concatenated)
console.log('5' * 2); // 10 (string '5' is converted to number and multiplied)
```

For example, in the expression '5' - 1, JavaScript will convert the string '5' to a number and subtract 1, resulting in 4. However, in the expression '5' + 1, JavaScript treats the + operator as a string concatenation operator when one of the operands is a string. As a result, the number 1 is automatically converted to a string, and the result is '51' instead of performing numeric addition. This automatic conversion can sometimes be hard to track, especially when the operations involve different data types.

While dynamic typing provides a lot of flexibility, it can sometimes make the code harder to understand and debug, especially when type conversions happen automatically and unexpectedly. Developers need to be cautious and often explicit about the types they expect in their code to avoid tricky bugs.

#### 4.1.2 Functions

Another important key feature of JavaScript is its support for **first-class functions**. A function, like in C, is a block of code designed to perform a specific task or computation. It takes input, processes it, and produces an output. Functions help in organizing code, making it more reusable and modular. Instead of writing the same code repeatedly, we can define it once inside a function and call it whenever we need it. In JavaScript, a function is typically defined using the "function" keyword and can have parameters to represent the values we pass into the function and a body, which contains the code that defines what the function does. For example, a simple function that adds two numbers could look like this:

```
function add(a, b) {
    return a + b;
}
```

In this example, a and b are parameters, and a + b is the computation the function performs. When the function is called, we provide arguments that correspond to the parameters, like so:

```
let result = add(3, 4); // result is now 7
```

Functions can return values, or they can perform actions without returning anything. If a function doesn't explicitly return a value, it implicitly returns **undefined**.

The specificity of JavaScript is that functions are treated like any other value, such as numbers or strings. A function can be assigned to a variable, passed around, or reassigned just like any other value. For example, we can define a function and assign it to a variable like this:

```
let name = function(parameters) {  
    // body  
};
```

Since functions are just variables, they can be used in any expression and can even be assigned to new values. We could, for example, reassign square to another function or value entirely:

```
name = function(parameters) {  
    // another body;  
};
```

Another interesting feature of functions in JavaScript is **nested scope**. A nested function is simply a function defined inside another function. The inner function is **accessible only** within the outer function and is useful for modularizing code, improving readability, and encapsulating logic that does not need to be exposed globally:

```
function outerFunction(a, b) {  
    console.log("Outer function executing ...");  
  
    function innerFunction(x, y) {  
        return x + y;  
    }  
  
    // Call inner function  
    let result = innerFunction(a, b);  
  
    console.log("Result:", result);  
}  
  
// call the outer function that will call the inner function  
outerFunction(5, 3);
```

JavaScript also supports **arrow functions**, which provide a more concise syntax for defining functions. Arrow functions are particularly useful for short, one-line functions. They are defined using the `=>` syntax and do not require the `function` keyword:

```
// No Parameters → Use empty parentheses:  
() => { console.log("Hello!"); }  
  
// One Parameter → Parentheses are optional:  
value => console.log(value);  
(value) => console.log(value);  
  
// Two or More Parameters → Parentheses are required:  
(a, b) => a + b;
```

They are more concise, reducing **boilerplate code** and making function expressions easier to read and write. When the function consists of a single expression, the curly braces and return keyword can be omitted, leading to a more streamlined syntax:

```
// Implicit return (when there's only one expression)  
let anon = (a, b) => a + b;  
console.log(anon(2, 3)); // 5  
  
// Explicit return using curly braces  
let anon = (a, b) => { return a + b;};
```

Summarizing, functions can be assigned to variables, passed as arguments to other functions, and returned from functions. This flexibility allows for powerful programming paradigms like **functional programming** and **callbacks**.

#### 4.1.3 Equality

We need to pay attention to equality comparisons, as they can behave differently from what we might expect coming from C. In JavaScript, there are two types of equality comparisons: **loose equality** (==) and **strict equality** (===).

The **strict equality operator** ( === ) checks both the type and the value of the operands to ensure they are exactly the same. For example:

```
5 === 5 // true, both are numbers with the same value  
'hello world' === 'hello world' // true, both are strings with the same value  
77 === '77' // false, one is a number, the other is a string  
false === 0 // false, different types (boolean vs. number)
```

The **loose equality operator** (`==`), on the other hand, performs type **coercion**. It tries to convert the operands to a common type before comparing their values. For example:

```
77 == '77' // true, string '77' is coerced to the number 77
false == 0 // true, false is coerced to 0
null == undefined // true, they are considered equal in loose equality
```

While `==` may seem more flexible, the triple equals (`====`) operator is generally preferred because it ensures a more reliable equality check by comparing both the type and the value of the operands. This helps avoid unexpected behavior due to type coercion:

```
// Correct way to check for equality
'123' === 123 // false, different types (string vs. number)
'123' == 123 // true, type coercion happens, '123' is converted to 123

// Best practice: Always use ===
if (value === 5) {
  console.log("The value is exactly 5");
}
```

Using strict equality (`====`) ensures that you're performing a true equality test, avoiding the pitfalls of implicit type conversions.

#### 4.1.4 Const, let and var

In JavaScript, the way variables are declared has evolved, especially with the introduction of **let** and **const** in ES6. Before ES6, the **var** keyword was the only way to declare a variable, but it comes with some quirks that can lead to unexpected behavior. For example, variables declared with **var** are **function-scoped, not block-scoped**. This means they are accessible throughout the function, regardless of where they are declared within a block, like inside a loop or an if statement:

```
function testVar() {
  if (true) {
    // `x` is available throughout the entire function, not just the block
    var x = 10;
  }

  // Outputs 10, even though `x` was declared inside the `if`
```

```
    console.log(x);  
}
```

This can lead to problems if we accidentally override a variable within a block, making the **behavior harder to predict**. Additionally, var is **hoisted**, meaning we can use a variable before it is declared! However, only the declaration is hoisted, not the assignment.

```
// Outputs undefined, due to hoisting  
console.log(y);  
var y = 20;
```

With ES6, let and const were introduced to address some of these issues. let is **block-scoped**, which means it is limited to the block in which it is declared, making it safer for use inside loops or conditionals:

```
function testLet() {  
  if (true) {  
    // `z` is only available within this block  
    let z = 30;  
    console.log(z);  
  }  
  // Error: `z` is not defined outside the block  
  console.log(z);  
}
```

const is also block-scoped but, unlike let, it makes the variable **read-only**. Once a variable is assigned with const, it cannot be reassigned, ensuring that its value remains constant.

```
const name = "John";  
  
// Error: Assignment to constant variable  
name = "Doe";
```

In most cases, it's recommended to use const for variables that will not be reassigned, as it helps prevent accidental changes and improves code clarity. Use let only when the value of a variable is expected to change, such as in loops or when the variable needs to be dynamically updated.

```
// Use const for constants
const pi = 3.14159;

// Use let when the value will change
let radius = 10;

// This is fine since we're using `let`
radius = 15;
```

As for var, it is best avoided in modern JavaScript, as let and const provide better, more predictable scoping behavior.

#### 4.1.5 Arrays

Arrays in JavaScript are similar to arrays in C, but they are more flexible and dynamic. Arrays allow us to **store multiple values** within a single variable and are often used to represent **lists** or **collections** of data. For example, you can store a list of numbers like this:

```
let listOfNumbers = [2, 3, 5, 7, 11, 4];
console.log(listOfNumbers[2]); // Outputs 5
```

In the example, the array contains six elements. Arrays in JavaScript are indexed, meaning each element can be accessed by its index, starting from 0. So, listOfNumbers[2] accesses the third element.

Arrays in JavaScript have various **properties** that provide useful information or functionality. One important property is **length**, which returns the number of elements in the array:

```
console.log(listOfNumbers.length); // Outputs 6
```

When accessing properties of an array, we can use either **dot notation** or **bracket notation**. The bracket notation allows use to use expressions or **dynamic property names**. For instance, we can access the length property like this:

```
console.log(listOfNumbers['length']);
```

Arrays also come with several\*\* built-in methods for manipulating the elements. **For example, we can** add or remove elements at the end\*\* of an array using push() and pop():

```
listOfNumbers.push(13); // Adds 13 to the end of the array  
console.log(listOfNumbers); // Outputs [2, 3, 5, 7, 11, 4, 13]
```

```
listOfNumbers.pop(); // Removes the last element (13)  
console.log(listOfNumbers); // Outputs [2, 3, 5, 7, 11, 4]
```

Similarly, we can **add or remove elements at the start** of an array using unshift() and shift():

```
listOfNumbers.unshift(1); // Adds 1 to the beginning of the array  
console.log(listOfNumbers); // Outputs [1, 2, 3, 5, 7, 11, 4]
```

```
listOfNumbers.shift(); // Removes the first element (1)  
console.log(listOfNumbers); // Outputs [2, 3, 5, 7, 11, 4]
```

We can also **search for elements** in an array using indexOf() and lastIndexOf(). These methods return the index of the first and last occurrence of a specified element, respectively:

```
console.log(listOfNumbers.indexOf(5)); // Outputs 2, the index of 5  
console.log(listOfNumbers.lastIndexOf(7)); // Outputs 3, the last index of 7
```

Another useful method is slice(), which allows you to **get a portion of an array** between two specified indexes, without modifying the original array:

```
let slicedArray = listOfNumbers.slice(1, 4); // Extracts elements from index 1 to 3  
// (4 is exclusive)  
console.log(slicedArray); // Outputs [3, 5, 7]
```

Arrays in JavaScript are **dynamic** and **versatile**, making them an essential tool for storing and managing collections of data. The various properties and methods allow us to perform a wide range of operations, from accessing and modifying elements to searching and slicing arrays.

#### 4.1.6 Strings

Strings in JavaScript are similar to strings in C, but they come with additional features and methods that make them more powerful. Strings are used to represent text data and are enclosed in either single quotes ("") or double quotes ("""). For example:

```
let carName1 = "Volvo XC60";
let carName2 = 'Volvo XC60';
```

Both variables hold the same value, but we can use whichever type of quotes suits our need or preference. Additionally, we can include quotes inside a string, as long as the quotes inside do not match the ones that are enclosing the string. For example:

```
let answer1 = "It's alright"; // Single quote inside a double-quoted string
let answer2 = "He is called 'Johnny'"; // Single quotes inside double quotes
let answer3 = 'He is called "Johnny"'; // Double quotes inside single quotes
```

Strings come with several useful **properties** and methods. For example, the **length** property returns the number of characters in a string:

```
let carName = "Volvo XC60";
console.log(carName.length); // Outputs 12
```

We can also transform a string to **uppercase** or **lowercase**:

```
let greeting = "Hello World";
console.log(greeting.toUpperCase()); // Outputs "HELLO WORLD"
console.log(greeting.toLowerCase()); // Outputs "hello world"
```

If you need to **extract a part of a string**, we can use the **slice()** method, or we can **find the position of a specific character** with **indexOf()** or access a character directly with **charAt()**:

```
let phrase = "Hello World";
console.log(phrase.slice(0, 5)); // Outputs "Hello"
console.log(phrase.indexOf('o')); // Outputs 4 (index of the first 'o')
console.log(phrase.charAt(6)); // Outputs 'W' (character at index 6)
```

The **trim()** method is helpful for **removing any whitespace** from both ends of a string:

```
let message = "Hello World ";
console.log(message.trim()); // Outputs "Hello World" (without spaces at the ends)
```

When comparing strings, JavaScript **compares them based on their value**. For example:

```
let answer1 = "It's alright";
let answer2 = "It's alright";
console.log(answer1 === answer2); // Outputs true
```

This comparison checks both the content and the type of the strings, so it returns true when the strings are identical.

#### 4.1.7 Parameter passing

In JavaScript, primitive values such as numbers, strings, and booleans are **passed by value** when used as function arguments. This means that when a primitive is passed to a function, the function **receives a copy of the value rather than the original**. Any modifications made inside the function do not affect the original variable outside the function:

```
function no_modify_primitives(x) {
  x = 7;
}

let x = 3;
no_modify_primitives(x);

// Outputs 3, because x was passed by value
console.log(x);
```

For arrays (and objects), JavaScript follows a **call-by-sharing** strategy, which is often confused with **pass-by-reference**. When an array is passed to a function, the function receives a reference to the same memory location rather than a copy of the value. This allows the function to modify the contents of the array or object:

```
function modify_array(array) {
  array[0] = 7;
}

let a = [3, 0, 2];

modify_array(a);

// Outputs [7, 0, 2], since the modification affects the original array
console.log(a);
```

However, if a function tries to assign a new array to the parameter, the **reference is broken**, and the function starts working with a new local version. This means that the original array or object remains unchanged outside the function.

```
function no_modify_array(array) {
    // This creates a new array, breaking the reference
    array = [3, 0, 2];
    array[0] = 9;
}

no_modify_array(a);

// Outputs [7, 0, 2], because reassignment inside the function
// does not affect the original array
console.log(a);
```

This behavior is crucial to understand when working with functions that modify arrays or objects, as direct modifications will persist, but reassignment will not affect the original variable.

#### 4.1.8 Optional arguments

JavaScript is **very flexible** when it comes to function arguments. If you pass more arguments than a function expects, the extra ones are simply **ignored**. If you pass fewer than required, the missing arguments are assigned the value **undefined**. This behavior allows functions to handle optional parameters, but it can also lead to unintended bugs if a function is accidentally used incorrectly. A common approach to dealing with optional arguments is to assign default values when an argument is undefined. This allows the function to work even when some parameters are omitted.

```
function power(base, exponent) {
    if (exponent === undefined) exponent = 2;
    let result = 1;
    for (let i = 0; i < exponent; i++) result *= base;
    return result;
}

console.log(power(3, 3)); // Outputs 27 (3^3)
console.log(power(3));   // Outputs 9  (3^2), using the default exponent
```

With the introduction of ES6, JavaScript provides a more concise way to handle optional arguments using default parameters:

```

function power(base, exponent = 2) {
  let result = 1;
  for (let i = 0; i < exponent; i++) result *= base;
  return result;
}

console.log(power(3, 3)); // Outputs 27
console.log(power(3));   // Outputs 9

```

Using default parameters directly in the function definition makes the code cleaner and easier to read while avoiding explicit checks for undefined. This feature is particularly useful when designing functions that can handle various use cases with minimal redundancy.

#### 4.1.9 Recursion

Recursion is a programming technique where a **function calls itself** to solve a problem by breaking it down into smaller subproblems. In JavaScript, recursion is often used when a problem can be naturally divided into simpler instances of itself, such as mathematical calculations, tree traversal, or searching algorithms. A classic example of recursion is computing the power of a number. Instead of using a loop, we can define a function that calculates the exponentiation of a base number by multiplying it with a recursive call to itself with a reduced exponent:

```

function power(base, exponent) {
  if (exponent == undefined) exponent = 2;
  if (exponent == 0) return 1;
  else return base * power(base, exponent - 1);
}

console.log(power(3, 3)); // Outputs 27 (3^3)
console.log(power(3));   // Outputs 9 (3^2, using the default exponent)
console.log(power(2, 5)); // Outputs 32 (2^5)
console.log(power(5, 0)); // Outputs 1 (base case)

```

In this function, the base case is when exponent reaches 0, in which case the function returns 1, as any number raised to the power of zero equals one. If the exponent is greater than zero, the function multiplies base by the result of calling power(base, exponent - 1), gradually decreasing exponent until it reaches the base case.

Each recursive call reduces the problem's complexity until it can be directly solved. This approach eliminates the need for explicit loops, making the code more readable and elegant. How-

ever, recursion should be used carefully, as excessive recursion depth can lead to stack overflow errors if the function calls itself too many times without reaching a termination condition.

#### 4.1.10 Error Handling

Error handling is a crucial aspect of programming, ensuring that programs can **respond to unexpected situations** or faults in a controlled manner. In JavaScript, error handling allows developers to catch and manage errors during runtime, preventing the application from crashing and providing users with meaningful feedback. JavaScript provides a robust system for error handling, relying heavily on the **try**, **catch**, and **finally** keywords to structure error management. The try block contains the code that might throw an error. If an error occurs within this block, JavaScript immediately jumps to the catch block, where the error can be handled. This allows the program to continue running without interruption. The catch block can access the error object, which contains details about the error, such as the message, stack trace, and error type. This makes it possible to handle different types of errors appropriately and provide relevant feedback. For instance, in a situation where we're trying to parse a string into JSON, a potential error could occur if the string is not valid JSON. Using a try...catch block, we can gracefully handle this:

```
try {
    let data = JSON.parse('{"name": "John", age: 30}'); // Syntax error
}
catch (error) {
    console.log("An error occurred:", error.message);
}
```

Additionally, JavaScript provides a finally block, which is optional. Code inside this block runs regardless of whether an error occurred or not. It is often used for cleanup tasks, such as closing files or releasing resources, ensuring that these actions are always performed no matter what.

```
try {
    let result = performRiskyOperation();
}
catch (error) {
    console.error("Error:", error);
}
finally {
    cleanup();
}
```

When an error occurs, JavaScript generates an object containing the details about it. This object has several properties, such as message, name, and stack, which can be used to identify the error and provide useful information to the developer. For example, the message property contains a human-readable description of the error, while the stack property contains a stack trace showing the sequence of function calls that led to the error

```
try {
    lalala; // This will cause a ReferenceError
}
catch (err) {
    alert(err.name); // ReferenceError
    alert(err.message); // lalala is not defined
    alert(err.stack); // ReferenceError: lalala is not defined at (... call stack)

    // Can also show an error as a whole
    // The error is converted to string as "name: message"
    alert(err); // ReferenceError: lalala is not defined
}
```

#### 4.1.11 Modules

Modules are a way **to organize and structure code** into reusable, isolated units that can be easily imported and exported between different parts of an application. The concept of modules helps manage complexity in large codebases by breaking down functionality into smaller, self-contained pieces. Each module can have its own set of variables, functions, and objects, which are scoped to that module and not accessible globally unless explicitly exported. For example, to export a function from a module, you can use the export keyword:

```
// module.js
export function greet(name) {
    return `Hello, ${name}!`;
}
```

Then, in another file, we can import and use the function like this:

```
// app.js
import { greet } from './module.js';

console.log(greet('Alice')); // Outputs: Hello, Alice!
```

In the example, the `greet()` function is exported from `module.js` and imported into `app.js`. The function is then called within the application, demonstrating how different parts of the application can share code through modules.

Modules also provide a way to control what is exposed to other parts of the program. By default, only the exported members of a module are accessible outside of it. This encapsulation helps maintain cleaner and more maintainable code by preventing unwanted side effects from global variables or functions. Additionally, a module can export multiple values at once, or even the default value for a module can be exported, making it easier to import without needing to reference specific exports.

```
// module.js
export const name = 'JavaScript';
export function greet(name) {
  return `Hello, ${name}!`;
}

// app.js
import * as utils from './module.js';

console.log(utils.name); // Outputs: JavaScript
console.log(utils.greet('Alice')); // Outputs: Hello, Alice!
```

JavaScript also supports CommonJS and AMD modules, which are widely used in environments like Node.js and older web applications, respectively. These module systems offer their own ways to import and export code, with CommonJS using `require()` and `module.exports`, and AMD using `define()` and `require()`.

## 4.2 Code examples

In this section, we will explore a range of fundamental algorithms implemented using JavaScript.

### 4.2.1 Factorial of a number

The factorial of a number is the product of all positive integers less than or equal to that number. For example, the factorial of 5 (written as  $5!$ ) is  $5 * 4 * 3 * 2 * 1 = 120$ . We can calculate the factorial of a number using a recursive function in JavaScript:

```

function factorial(x) {
    if (x < 0) throw new Error("Factorial is not defined for negative numbers.");
    if (x === 0 || x === 1) return 1;
    return x * factorial(x - 1);
}

console.log(factorial(15)); // Outputs 1307674368000
console.log(factorial(0)); // Outputs 1
console.log(factorial(1)); // Outputs 1

```

#### 4.2.2 Binary search

Binary search is an efficient algorithm for finding a target value within a sorted array. It works by repeatedly dividing the search interval in half until the target value is found or the interval is empty. Here's an example of a binary search function in JavaScript:

```

function binarySearch(array, target, start = 0, end = array.length - 1) {
    // Base case: not found
    if (start > end) return undefined;

    const mid = Math.floor((start + end) / 2);

    if (array[mid] === target) return mid;
    if (array[mid] < target) return binarySearch(array, target, mid + 1, end);

    return binarySearch(array, target, start, mid - 1);
}

const list = [21, 45, 76, 93, 204, 345, 6654];

console.log(binarySearch(list, 345)); // Outputs: 5
console.log(binarySearch(list, 50)); // Outputs: undefined (not found)
console.log(binarySearch(list, 21)); // Outputs: 0
console.log(binarySearch(list, 6654)); // Outputs: 6

```

#### 4.2.3 Bubble sort

Bubble sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass through the list is

repeated until no swaps are needed, indicating that the list is sorted. Here's an example of a bubble sort function in JavaScript:

```
function swap(array, first_index, second_index){
    let temp = array[first_index];
    array[first_index] = array[second_index];
    array[second_index] = temp;
}

function bubble_sort(array){
    let len = array.length;
    for (let i=0; i<len; i++)
        for (let j=0; j<len-i; j++)
            if (array[j] > array[j+1]) swap(array, j, j+1);
    return array;
}

let a = [3, 0, 2, 5, -1, 4, 1];
console.log(bubbleSort(a)); // [-1, 0, 1, 2, 3, 4, 5]
```

#### 4.2.4 Quicksort

Quicksort is a popular sorting algorithm that uses a divide-and-conquer strategy to sort an array. It works by selecting a pivot element from the array and partitioning the other elements into two sub-arrays according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively. Here's an example of a quicksort function in JavaScript:

```
function quickSort(array) {
    if (array.length <= 1) return array;

    // Take the last element as pivot
    const pivot = array[array.length - 1];

    const left = [];
    const right = [];

    for (let i = 0; i < array.length - 1; i++) {
        if (array[i] <= pivot) {
            left.push(array[i]);
        } else {
            right.push(array[i]);
        }
    }

    return [...left, pivot, ...right];
}
```

```

        }
    }
    return quickSort(left).concat(pivot, quickSort(right));
}

let myArray = [3, 0, 2, 5, -1, 4, 1];
console.log("Original array:", myArray);

let sortedArray = quickSort(myArray);
console.log("Sorted array:", sortedArray);

```

## 4.3 Functional Programming

JavaScript supports **functional programming**, a programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing state and mutable data. Functional programming emphasizes the use of **pure functions**, which produce the same output for the same input and do not have side effects. This approach leads to more predictable and maintainable code, as functions are isolated and do not rely on external state. JavaScript provides several features that support functional programming, such as **higher-order functions** and **closures**.

### 4.3.1 High order functions

Higher-order functions are functions that either **take other functions as arguments, return functions**, or both. They allow for the abstraction of behavior over values, enabling a **more flexible and functional style** of programming. These functions are powerful because they allow us to write more generic, reusable, and concise code. By treating functions as first-class citizens (just like numbers or strings), we can pass them around, return them from other functions, or create specialized behavior dynamically. A higher-order function operates on other functions in a way that allows us to encapsulate specific actions, not just values.

```

function greaterThan(n) {
    return function(m) { return n > m; };
}

let greaterThan10 = greaterThan(10);
console.log(greaterThan10(7)); // true
console.log(greaterThan10(11)); // false

```

```
let greaterThan4 = greaterThan(4);
console.log(greaterThan4(3)); // true
console.log(greaterThan4(5)); // false
```

In the example above, `greaterThan` is a higher-order function. It returns a function that checks if `n` is greater than a given number `m`. The variable `n` is accessible inside the inner function due to lexical scoping, where the inner function "remembers" the environment in which it was created. This ability to return functions and capture variables from their surrounding scope is a key feature of functional programming.

#### 4.3.2 Filter, map and Reduce

JavaScript provides several built-in higher-order functions that are commonly used in functional programming. Three of the most popular ones are `filter`, `map`, and `reduce`. These functions allow us to manipulate arrays in a concise and expressive way, making it easier to work with collections of data.

The `filter` function takes an array and a test function as arguments, and it returns a new array containing all elements that pass the test:

```
function filter(array, test) {
  let passed = [];
  for (let i = 0; i < array.length; i++) {
    if (test(array[i])) passed.push(array[i]);
  }
  return passed;
}

let a = [2, 3, 5, 6];
let test = function(x) { return x ≥ 5; };
let filtered = filter(a, test);
console.log(filtered); // [5, 6]
```

The `map` function transforms each element in an array using a transformation function.

```
function map(array, transform) {
  let mapped = [];
  for (let i = 0; i < array.length; i++) {
```

```

        mapped.push(transform(array[i]));
    }
    return mapped;
}

let a = ['3', '5', '6'];
let transform = function(x) { return Number(x); };
let mapped = map(a, transform);
console.log(mapped); // [3, 5, 6]

```

The **reduce** function combines all elements of an array into a single value, using a combining function and an initial value.

```

function reduce(array, combine, start) {
    let current = start;
    for (let i = 0; i < array.length; i++) {
        current = combine(current, array[i]);
    }
    return current;
}

let a = [3, 5, 6];
let sum = function(x, y) { return x + y; };
let result = reduce(a, sum, 0);
console.log(result); // 14

```

The implementation of filter, map, and reduce for arrays is straightforward and may seem intuitive, but it is often naïve and inefficient. Manually iterating over an array and applying transformations using loops can lead to unnecessary complexity, reduced readability, and potential performance issues. JavaScript arrays already provide built-in `.filter()`, `.map()`, and `.reduce()` methods, which are optimized for performance.

```

let filtered = a.filter(test);
let mapped = a.map(value => value * 2);
let reduced = a.reduce(sum, 0);

```

In general, instead of manually iterating and modifying lists, leveraging language-based methods to ensure better performance and cleaner code.

### 4.3.3 Closures

One of the most powerful features of higher-order functions is the concept of closures. When a function is returned by another function, the inner function **retains access** to the variables of the outer function, even after the outer function has completed execution.

```
function wrapValue(n) {
  let localVariable = n;
  return function() { return localVariable; };
}

let wrap1 = wrapValue(1);
let wrap2 = wrapValue(2);

console.log(wrap1()); // 1
console.log(wrap2()); // 2
```

In this example, `wrapValue` returns a function that retains access to `localVariable` even after the `wrapValue` function has finished executing. This enables a powerful pattern where we can “freeze” specific values for later use. This pattern is useful in **stateful applications**, like managing user sessions, caching results, or handling module states. As an explicative example, we can create a simple counter that maintains its state between calls:

```
function createCounter() {
  let count = 0;
  return function () {
    count++;
    console.log(`Current count: ${count}`);
  };
}

const counter = createCounter();

counter(); // Current count: 1
counter(); // Current count: 2
counter(); // Current count: 3
```

### 4.3.4 Currying

A powerful application of closures is **currying**, a technique that transforms a function from being called as `f(a, b, c)` into a sequence of calls like `f(a)(b)(c)`, or any number of parameters. Let's

explore an example to clarify the concept, followed by practical applications. We'll start by implementing a helper function `curry()` which enables currying for a three-argument function:

```
function curry(f) {
  return function(a) {
    return function(b) {
      return function(c) {
        return f(a, b, c);
      }
    };
  };
}

function sum(a, b, c) {
  return a + b + c;
}

let curriedSum = curry(sum);
console.log(curriedSum(1)(2)(3));
```

The implementation is straightforward: it's just three wrappers. The result of `curry()` is a wrapper `function(a)`. When it is called (like `curriedSum(1)`), the argument is saved in the freezed variable `a`, and a new wrapper is returned (`function(b)`). Then this wrapper is called with the second argument, and again, the argument is saved in the freezed variable `b`, and a new wrapper is returned (`function(c)`). Finally, when the last wrapper is called with the third argument, the original function is called with all three arguments. To understand the benefits of this technique, we need a real-life example. Suppose we have a logging function that formats and outputs some information, like:

```
function log(date, type, message) {
  console.log(date.getHours(), date.getMinutes(), type, message);
}
```

Let's curry the `log` function:

```
let curriedLog = curry(log);
```

After that `log` works normally, but also works in the curried form:

```
log(new Date(), "DEBUG", "Message");
curriedLog(new Date())("DEBUG")("Message");
```

However, by using the curried version, we can create a **specialized** logging function. Instead of repeatedly passing the same argument, we can define a version that automatically includes common parameters, making the code more concise. For example, we can create a function that prepends a timestamp to each message:

```
let logNow = curriedLog(new Date());
logNow("DEBUG")("Message");
```

The function `logNow()` is a version of the `log()` function with fixed first argument, in other words “partially applied function” or “partial” for short. We can go further and make a convenient function for current debug logs:

```
let debugNow = logNow("DEBUG");
debugNow("Message");
```

Higher-order functions are a cornerstone of Javascript programming. By allowing functions to operate on other functions, they enable more modular, reusable, and concise code. Whether we’re using them for mapping, filtering, reducing, or creating closures and curries, higher-order functions offer a powerful toolset for abstracting and composing behaviors in our applications.

## 4.4 Object Oriented Programming

JavaScript is a versatile language that supports multiple programming paradigms, including **object-oriented programming (OOP)**. OOP is a programming paradigm that organizes code into objects, which encapsulate data and behavior. Objects interact with each other through methods, and they can inherit properties and methods from other objects.

### 4.4.1 Objects

Objects are **arbitrary collections of properties** that can be **created directly without requiring a class**. An object is simply a set of key-value pairs, where keys are property names and values can be any type, including numbers, strings, arrays, or even other objects. The syntax is straightforward:

```
let example = { name: 'example', values: [1, 2, 3], length: 3 };
console.log(example);
```

Each property is defined by a name, followed by a colon and its corresponding value, separated by commas. Objects are flexible and can be dynamically modified by **adding or removing properties at runtime**. One important thing to remember is that objects are **reference-based**. Comparing two different objects using `==` or `====` will always return false, even if they have identical properties. This is because they are stored at different memory locations, similar to pointers in languages like C or Java.

```
let example1 = { name: 'example' };
let example2 = { name: 'example' };
console.log(example1 == example2); // false
```

Objects can also contain **methods**, which are properties that hold function values. Inside these functions, the keyword **this** refers to the object itself, allowing access to other properties. For example:

```
let example = {
  name: 'example',
  append: function(x) {
    this.name = this.name + ' ' + x;
  }
};

console.log(example);
example.append('pippo');
console.log(example);
```

JavaScript also provides built-in objects as collections of methods and properties related to specific functionalities. For example, the **Math** object contains mathematical constants and functions (like `Math.sqrt()`, `Math.random()` and `Math.pow()`), while the **Date** object provides methods for working with dates and times. These built-in objects are useful for common tasks and can be accessed directly without the need to create instances. In particular, it is really useful for our IoT applications to work with the **JSON** object, which provides methods for converting objects to JSON strings and vice versa.

#### 4.4.2 Factories

Creating objects directly can be useful in several cases, however it can be cumbersome when we need to create multiple objects with similar properties and methods. In these cases, we can use **factory functions** to create objects based on a template. A factory function is a function that returns an object, allowing us to create multiple instances with shared properties and methods. For example:

```
function createUser(name, age) {
  return {
    name: name,
    age: age,
    greet() {
      console.log(`Hello, my name is ${this.name} and I am ${this.age} years old.`);
    }
  };
}

let user1 = createUser("Alice", 25);
let user2 = createUser("Bob", 30);

user1.greet(); // "Hello, my name is Alice and I am 25 years old."
user2.greet(); // "Hello, my name is Bob and I am 30 years old."
```

Think of a factory as an automated assembly line. Each time it is called, it produces a new object, just like a factory producing identical products with different configurations. This method is useful for encapsulation, allowing internal logic to be hidden while providing a simple interface to create objects.

#### 4.4.3 Constructors

Another way to create objects in JavaScript is by using **constructors**. A constructor is a function that is called with the **new** keyword to create a new object:

```
function User(name, age) {
  this.name = name;
  this.age = age;
  this.greet = function() {
    console.log(`Hello, my name is ${this.name} and I am ${this.age} years old.`);
  };
}
```

```

}

let user1 = new User("Alice", 25);
let user2 = new User("Bob", 30);

user1.greet(); // "Hello, my name is Alice and I am 25 years old."
user2.greet(); // "Hello, my name is Bob and I am 30 years old."

```

When we use `new` with a constructor, JavaScript does several things behind the scenes: it creates a new empty object, binds this to that object, and then runs the constructor function, allowing us to initialize properties and methods for the object. After the constructor function executes, it automatically returns the object created. The main difference with a factory function is that all instances created with a constructor are **linked to a prototype**. The prototype is a fundamental feature of the language **inheritance model**, it is an object from which an object inherits properties and methods. When we access a property or a method on an object, JavaScript looks for it on the object itself, and if it doesn't find it there, it looks up the prototype. This allows objects to share functionality and reduces memory usage by avoiding duplication of methods across all instances of an object. When we use a constructor function, every instance created by that constructor shares a common prototype. The prototype is typically used to define methods or properties that should be available to all instances, ensuring that these methods aren't copied for each instance, but instead are shared across them:

```

function User(name, age) {
  this.name = name;
  this.age = age;
}

// Adding a method to the prototype
User.prototype.greet = function() {
  console.log(`Hello, my name is ${this.name} and I am ${this.age} years old.`);
};

let user1 = new User("Alice", 25);
let user2 = new User("Bob", 30);

// Both user1 and user2 share the same greet method via the prototype.
user1.greet(); // "Hello, my name is Alice and I am 25 years old."
user2.greet(); // "Hello, my name is Bob and I am 30 years old."

```

In the example, `greet()` is defined on the prototype of `User`, which means both instances (`user1`

and user2) share the same greet() method. This results in **memory efficiency** because the method is not recreated for each instance of User. Instead, both objects look up the method on the prototype. Factory functions, instead, don't rely on prototypes. Each time a factory function is called, it creates a completely new object that is not linked to any prototype. This means that each instance of an object created by a factory function is independent, and does not share methods via a prototype. However, factory functions provide more flexibility and control over object creation. We can encapsulate private data and logic within the factory, whereas constructors are more rigid and focus primarily on creating instances with shared prototypes. In summary, constructor functions are ideal when we want multiple instances to share common methods via the prototype, reducing memory usage and enabling inheritance. Factory functions provide more flexibility and are suitable when we need independent objects.

As an example, we can create the support for a linked list in JavaScript using the following code (see *01.LinkedList.js* script). First we create the constructor:

```
function List() {
    this.makeNode = function() { return { data: null, next: null }; };
    this.start = null;
    this.end = null;
}
```

Then we add some methods to the prototype:

```
List.prototype.add = function(data) {
    if (this.start === null) { this.start = this.makeNode(); this.end = this.start; }
    else { this.end.next = this.makeNode(); this.end = this.end.next; };
    this.end.data = data;
};
```

```
List.prototype.delete = function(data) {
    let current = this.start;
    let previous = this.start;
    while (current !== null) {
        if (data === current.data) {
            if (current === this.start) { this.start = current.next; return; }
            if (current === this.end) this.end = previous;
            previous.next = current.next; return;
        }
        previous = current;
        current = current.next;
    }
}
```

```
    }
};
```

```
List.prototype.insertAsFirst = function(d) {
    let temp = List.makeNode();
    temp.next = this.start;
    this.start = temp;
    temp.data = d;
};
```

```
List.prototype.insertAfter = function(t, d) {
    let current = this.start;
    while (current != null) {
        if (current.data == t) {
            let temp = List.makeNode();
            temp.data = d;
            temp.next = current.next;
            if (current == this.end) this.end = temp;
            current.next = temp;
            return;
        }
        current = current.next;
    }
};
```

```
List.prototype.item = function(i) {
    let current = this.start;
    while (current != null) {
        i--;
        if (i == 0) return current;
        current = current.next;
    }
    return null;
};
```

```
List.prototype.each = function(f) {
    let current = this.start;
    while (current != null) {
        f(current);
        current = current.next;
```

```
    }  
};
```

Finally, we can create a new linked list and add some elements:

```
var list = new List();  
  
for(var i=1;i<=10;i++){  
    list.add(i);  
}  
  
list.each(function(item) { console.log(item.data); });  
  
list.delete(4);  
  
list.each(function(item) { console.log(item.data); });
```

#### 4.4.4 Getter and Setter

Getters and Setters are special methods that allow us to define how to retrieve or set the value of an object property in a clean and controlled way. A getter is a method that gets the value of an object property. It allows us to access a property as usual and (behind the scenes) actually invoking a function to perform some logic or computation before retrieving the value:

```
const rectangle = {  
    width: 10,  
    height: 5,  
  
    // Getter for area  
    get area() {  
        return this.width * this.height;  
    }  
};  
  
console.log(rectangle.area); // 50
```

In this example, `area` is a computed property. Since it is defined using a getter, we can access it like a regular property without parentheses, which helps keep the interface clean.

A setter is a method used to set the value of a property. It allows us to control how a property is updated in a **controlled manner**. It's often used when we want to prevent direct manipulation of a properties, or when setting a value requires additional logic (like validation or transformations):

```
const rectangle = {
    _width: 10,
    _height: 5,

    // Getter for area
    get area() {
        return this._width * this._height;
    },

    // Setter for width
    set width(value) {
        if (value <= 0) {
            console.log("Width must be positive.");
        } else {
            this._width = value;
        }
    },

    // Setter for height
    set height(value) {
        if (value <= 0) {
            console.log("Height must be positive.");
        } else {
            this._height = value;
        }
    }
};

rectangle.width = 15; // Valid value, setter is called
rectangle.height = -3; // Invalid value, setter rejects it
console.log(rectangle.area); // 75 (after width is updated)
```

In the example, width and height are private properties, but their values are modified through the setters. The setters ensure that the width and height are positive numbers and provide a controlled way to update the values. Notice that the underscore (\_) before a property name, like `_width` or `_height`, is a **common convention** in JavaScript to indicate that the property is private or intended for internal use only. While JavaScript does not natively support private

properties (though this is changing with the introduction of **private fields** in ES2022, bat for classes), the underscore serves as a visual cue to developers that the property should not be accessed or modified directly from outside the object.

#### 4.4.5 Classes

JavaScript in ES 6 introduces also the support for **class-based object-oriented programming**. A class is a blueprint for creating objects, defining properties and methods that all instances of the class will share. Classes provide a more structured and organized way to create objects, similar to classes in other object-oriented languages like Java or C++. A class is defined using the **class** keyword followed by the name. Inside a class, we can define the **constructor** method, which is called when a new instance of the class is created. The constructor is used to initialize the object's properties. We can also define methods that are shared by all instances of the class. These methods can operate on the properties of the instance and define behaviors specific to that object. Here's an example of a class in JavaScript:

```
class Rectangle {
    constructor(width, height) {
        this.width = width;
        this.height = height;
    }

    getArea() {
        return this.width * this.height;
    }
}

const myRectangle = new Rectangle(10, 5);
console.log(myRectangle.getArea()); // 50
```

Classes in JavaScript also support **inheritance**, allowing one class to inherit properties and methods from another class. This is done using the **extends** keyword, which creates a subclass that inherits from a parent class. The subclass can override or extend the functionality of the parent class:

```
class Shape {
    constructor(name) {
        this.name = name;
    }
}
```

```
describe() {
  return `This is a shape named ${this.name}`;
}

class Circle extends Shape {
  constructor(name, radius) {
    super(name); // Calls the parent class's constructor
    this.radius = radius;
  }

  getArea() {
    return Math.PI * this.radius * this.radius;
  }
}

const circle = new Circle('Circle', 10);
console.log(circle.describe()); // "This is a shape named Circle"
console.log(circle.getArea()); // 314.159
```

Classes also introduce also the concept of **static methods**. These are methods that belong to the class itself, not to instances of the class. Static methods are useful for utility functions or methods that do not depend on instance properties.

```
class MathUtility {
  static add(a, b) {
    return a + b;
  }
}

console.log(MathUtility.add(5, 3)); // 8
```

Classes are essentially **syntactic sugar** over constructor functions, offering a more readable and structured approach to object-oriented programming. While they serve the same purpose as constructor functions, classes provide a more formal and consistent structure. Overall, classes offer a more modern and object-oriented approach, while constructor functions are the older, more flexible method. Despite the differences in syntax and structure, both classes and constructors are used to achieve similar goal, creating objects with shared properties and behaviors. However, classes provide more clarity and support for advanced features like inheritance and method definitions, making them the preferred approach in modern JavaScript.

#### 4.4.6 Polymorphism

Polymorphism is a key concept in object-oriented programming that allows objects of different types to be treated as instances of the same type through a common interface. It describes the ability of different objects to respond to the same method in different ways, depending on their underlying type. Polymorphism provides flexibility and extensibility. It allows developers to write more generalized code that can work with objects of different types, promoting code reuse and reducing redundancy. For example, a function can be written to accept any object that implements a certain method, regardless of the specific class it belongs to. This enables objects of various classes to be treated in a uniform manner, while still allowing each class to implement the method in a way that is appropriate for its own behavior.

```
class Animal {
    speak() {
        console.log("Animal makes a sound");
    }
}

class Dog extends Animal {
    speak() {
        console.log("Dog barks");
    }
}

const myAnimal = new Animal();
const myDog = new Dog();

myAnimal.speak(); // "Animal makes a sound"
myDog.speak(); // "Dog barks"
```

In this example, both `Animal` and `Dog` have a `speak` method, but the behavior of the method differs depending on the object calling it. This is an example of **method overriding**, where the `Dog` class provides a new implementation for the `speak` method.

In JavaScript, polymorphism is not restricted to classes or object-oriented programming. Even without using classes, we can still leverage polymorphism through the use of objects and functions. The concept remains the same: polymorphism allows different types to be treated uniformly by using a common interface, meaning that different objects or functions can exhibit different behaviors while sharing the same method name or function signature. The key is that you can have multiple objects with the same method name, and each object can implement that

method differently. This allows us to interact with different objects in a similar manner while enabling them to execute unique behaviors when needed:

```
const dog = {
  speak: function() {
    console.log("Dog barks");
  }
};

const cat = {
  speak: function() {
    console.log("Cat meows");
  }
};

const bird = {
  speak: function() {
    console.log("Bird chirps");
  }
};

function makeAnimalSpeak(animal) {
  animal.speak();
}

makeAnimalSpeak(dog); // "Dog barks"
makeAnimalSpeak(cat); // "Cat meows"
makeAnimalSpeak(bird); // "Bird chirps"
```

Polymorphism helps achieve **loose coupling** in a system, where components are not tightly dependent on each other. Instead, they rely on a common interface or base class, making the system more flexible and easier to extend. This enables developers to create highly reusable and maintainable code.

## 4.5 Programming paradigms

We have seen that JavaScript supports multiple programming paradigms, including **procedural**, **functional**, and **object-oriented** programming. Each paradigm offers a **different approach** to structuring code and solving problems, and the language flexibility allows developers to choose the best paradigm for a given task.

#### 4.5.1 Procedural Programming

In procedural programming, the focus is on writing sequences of instructions or procedures (functions) that operate on data. This paradigm is based on the concept of the procedure call, where functions are defined to perform specific tasks and are executed in a step-by-step manner. The program is typically organized around a series of actions or instructions that manipulate data. JavaScript, in its early days, was mainly used in a procedural style, with functions being written to execute particular tasks. For example, a JavaScript program might involve defining functions to handle user input, perform calculations, and update the user interface, all without structuring the program around objects or classes. Let's approach a simple problem (calculating the sum of the squares of even numbers in an array) from the procedural approach:

```
function sumOfEvenSquares(numbers) {
    let sum = 0;

    // Loop through the array
    for (let i = 0; i < numbers.length; i++) {
        // Check if the number is even
        if (numbers[i] % 2 === 0) {
            // Add the square of the even number to the sum
            sum += numbers[i] ** 2;
        }
    }

    return sum;
}

const numbers = [1, 2, 3, 4, 5, 6];
console.log(sumOfEvenSquares(numbers)); // Output: 56
```

In the procedural approach, we use a for loop to iterate through the array, check for even numbers, and calculate their squares. The result is accumulated in the sum variable, which is returned at the end.

#### 4.5.2 Object Oriented Programming

On the other hand, object-oriented programming organizes code around the concept of objects, which are instances of classes. These objects hold both data and methods that define their behaviors. OOP encourages the use of inheritance, encapsulation, and polymorphism to model real-world entities and their interactions. JavaScript adopted OOP principles in ES6 with the

introduction of the class syntax, although the language has always allowed object-based programming through the use of object literals and prototypes. Let's rewrite the previous example using an object-oriented approach:

```
class EvenSquaresCalculator {  
    constructor(numbers) {  
        this.numbers = numbers;  
    }  
  
    sumOfEvenSquares() {  
        let sum = 0;  
  
        // Loop through the array of numbers  
        for (let number of this.numbers) {  
            // Check if the number is even  
            if (number % 2 === 0) {  
                // Add the square of the even number to the sum  
                sum += number ** 2;  
            }  
        }  
  
        return sum;  
    }  
}  
  
const calculator = new EvenSquaresCalculator([1, 2, 3, 4, 5, 6]);  
console.log(calculator.sumOfEvenSquares()); // Output: 56
```

In the object-oriented style, we encapsulate the logic in an object that represents the process of summing the squares of even numbers. We define a method inside an object to handle the calculation.

#### 4.5.3 Functional Programming

Functional programming, in contrast, emphasizes functions as the primary building blocks of a program. In this paradigm, functions are first-class citizens, meaning they can be assigned to variables, passed as arguments, and returned from other functions. Functional programming avoids shared state and mutable data, focusing instead on **pure functions** (functions that produce the same output for the same input and have no side effects). JavaScript supports functional programming by allowing anonymous functions, higher-order functions, and closures,

making it easier to write code in a functional style. Let's rewrite the previous example using a functional approach:

```
const sumOfEvenSquares = (numbers) =>
  numbers
    .filter(num => num % 2 === 0)      // Get even numbers
    .map(num => num ** 2)              // Square each even number
    .reduce((acc, num) => acc + num, 0); // Sum the squares

const numbers = [1, 2, 3, 4, 5, 6];
console.log(sumOfEvenSquares(numbers)); // Output: 56
```

In the functional approach, we use filter to extract the even numbers, map to square them, and reduce to sum the squared values. Each of these operations returns a new array, and the data is processed without mutating any state. The functions are small and composed in a declarative way.

#### 4.5.4 Imperative vs Declarative style

In this context, we can also distinguish between **imperative** and **declarative** programming styles. Imperative programming is about **explicitly describing the steps needed to perform a task**. In this paradigm, the programmer writes a sequence of instructions that tell the computer how to do something. This includes specifying the exact control flow (like loops and conditionals) and managing the state of the program at each step. Imperative code is focused on the control of execution, where the programmer takes on the responsibility of managing the state and providing a step-by-step process. For example, in an imperative style, calculating the sum of even numbers in an array might look like this:

```
let numbers = [1, 2, 3, 4, 5, 6];
let sum = 0;

for (let i = 0; i < numbers.length; i++) {
  if (numbers[i] % 2 === 0) {
    sum += numbers[i];
  }
}

console.log(sum); // Output: 12
```

Declarative programming, on the other hand, focuses on **describing what we want to achieve without specifying all the steps involved**. In this style, we express the desired result, leaving the underlying mechanism for achieving it abstracted away. Declarative programming abstracts away the control flow and focuses more on the end goal. This approach is generally more readable and concise, as it removes much of the boilerplate code. The same task of summing the even numbers in an array can be written in a declarative style like this:

```
let numbers = [1, 2, 3, 4, 5, 6];
let sum = numbers.filter(num => num % 2 === 0)
    .reduce((acc, num) => acc + num, 0);

console.log(sum); // Output: 12
```

In JavaScript, all three paradigms (procedural, OOP and function) can be used together, depending on the needs of the application and the preferences of the developer. JavaScript is particularly well-suited to support these paradigms because it is flexible and allows developers to mix and match approaches within the same application. For example, we might write our data handling and transformations in a functional style, while organizing the logic for user interfaces and complex behaviors in an object-oriented manner. In general, the procedural approach is useful for small, straightforward scripts where we just need to process data in a linear fashion. Object-oriented programming shines in scenarios that require modeling complex entities and behaviors, particularly when dealing with large-scale applications. Functional programming, with its emphasis on immutability and pure functions, is ideal for cases that require high levels of concurrency, easy testing. Of course, both style (imperative and declarative) are often used together. For instance, we can write imperative code for certain low-level tasks that require explicit control, while using declarative methods like map, filter, and reduce for data manipulation. Let's consider an example to compare the two styles (see *02.declarative\_imperative.js*). Suppose to have a JSON string representing ancestral data. It stores information about individuals, including their name, sex, birth and death years, and their parents' names:

```
var ancestry_file = '[\n  ' + [
  '{"name": "Carolus Haverbeke", "sex": "m", "born": 1832, "died": 1905, "father":',
  '  ↳ "Carel Haverbeke", "mother": "Maria van Brussel"}',
  '{"name": "Emma de Milliano", "sex": "f", "born": 1876, "died": 1956, "father":',
  '  ↳ "Petrus de Milliano", "mother": "Sophia van Damme"}',
  '{"name": "Maria de Rycke", "sex": "f", "born": 1683, "died": 1724, "father":',
  '  ↳ "Frederik de Rycke", "mother": "Laurentia van Vlaenderen"}',
  '{"name": "Jan van Brussel", "sex": "m", "born": 1714, "died": 1748, "father":',
  '  ↳ "Jacobus van Brussel", "mother": "Joanna van Rooten"}',
```

```

...
'{"name": "Jacobus Bernardus van Brussel", "sex": "m", "born": 1736, "died":
    ↵ 1809, "father": "Jan van Brussel", "mother": "Elisabeth Haverbeke"}'
].join(",\n ") + "\n]";

var ancestry = JSON.parse(ancestry_file);

```

We want to compute the average age at death for males and females separately, demonstrating a declarative programming style:

```

function plus(a,b){ return a + b; }
function age(p) { return p.died - p.born; }
function male(p) { return p.sex == "m"; }
function female(p) { return p.sex == "f"; }

console.log(ancestry.filter(male).map(age).reduce(plus)/ancestry.filter(male).length);
console.log(ancestry.filter(female).map(age).reduce(plus)/ancestry.filter(female).length);

```

We can adopt a more declarative style by using arrow functions:

```

console.log(ancestry.filter(p => p.sex == "m").map(p => p.died - p.born).reduce((a,b)
    => a + b)/ancestry.filter(p => p.sex == "m").length);

```

In contrast, an imperative approach would involve writing a series of loops and conditional statements to achieve the same result:

```

let males = [];
for(let i=0; i<ancestry.length; i++) {
    if(ancestry[i].sex === 'm') males.push(ancestry[i]);
}

let ages = [];
for(let i=0; i<males.length; i++) {
    ages.push(males[i].died - males[i].born);
}

let sum = 0;
for(let i=0; i<ages.length; i++) {
    sum += ages[i];
}

```

```
let average = sum / ages.length;  
console.log(average);
```

## 4.6 More advanced topics

JavaScript is as a versatile and dynamic programming language, it offers several other features and concepts not covered in this section.

**Regular expressions** (RegEx) are one of the more sophisticated tools available in JavaScript. They allow for pattern-based text searching, validation, and manipulation, making them incredibly useful for tasks such as form validation or extracting specific pieces of data from a string. This powerful feature helps developers write concise code for complex text processing, such as matching email formats, phone numbers, or even parsing log files.

Another fundamental feature of JavaScript is its ability to **interact with the browser** through the Document Object Model (DOM). The DOM provides a structured representation of the HTML page, allowing JavaScript to access and modify the content dynamically. Whether it's adding new elements, changing existing ones, or updating the styling in response to user actions, the DOM provides the interface for JavaScript to shape the webpage in real time, contributing to the dynamic and interactive nature of modern websites.

**Event handling** is another key feature in JavaScript, particularly when it comes to making web-pages interactive. Events are actions triggered by users, such as clicking buttons, typing in input fields, or hovering over elements. JavaScript enables developers to listen for these events and execute specific functions in response. By adding event listeners to elements, developers can manage user interactions in a smooth and seamless way, providing feedback or triggering changes on the page without requiring a full page reload. we will consider this feature in the context of API development using Node.js.

**Timers** in JavaScript offer additional flexibility by allowing the execution of functions after a specified delay or at regular intervals. This is particularly useful for tasks such as creating animations, auto-refreshing content, or handling timeouts in asynchronous operations.

Finally, **handling forms** in JavaScript adds another layer of interactivity. Forms are essential for collecting user input, and JavaScript allows developers to validate and manipulate that data before it is submitted to the server. This can include checking for empty fields, ensuring proper formatting, or even providing real-time feedback as users complete forms.

## 5 HTTP

The **World Wide Web** (the vast, interconnected system of information and interactions we rely on daily) operates on a foundation of fundamental protocols that enable seamless global communication. At the core of this system is the Hypertext Transfer Protocol (HTTP), the backbone of modern web communication.

### OSI Model

- 7 - Application layer - HTTP, FTP, DNS
- 6 - Presentation layer - SSL, TLS
- 5 - Session layer - NetBIOS, PPTP
- 4 - Transport layer - TCP, UDP
- 3 - Network layer - IP
- 2 - Data Link layer - PPP, ATM, Ethernet
- 1 - Physical layer - Ethernet, USB, Bluetooth, IEEE802.11

As illustrated in the image, HTTP is an **application-layer protocol** of the OSI model and it is designed for distributed, collaborative, hypermedia information systems. Since its introduction in 1990, it has served as the primary mechanism for data exchange between **clients** (such as web browsers) and **servers** (where websites are hosted).

HTTP is often described as **generic and stateless**, meaning each request is independent and does not retain session information by default. Its versatility comes from its ability to handle different operations through:

- Request methods (e.g., GET for retrieving data, POST for sending data)
- Status codes (indicating the outcome of requests)
- Headers (providing additional metadata about the exchange)

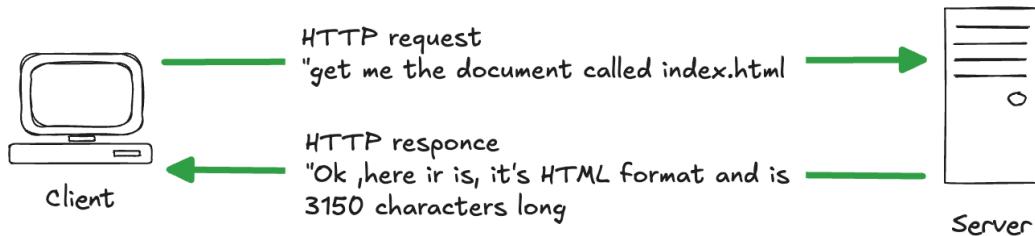
At its foundation, HTTP operates over the TCP/IP protocol suite, leveraging TCP (Transmission Control Protocol) for reliable, ordered data transmission across the internet. TCP, in turn, relies on IP (Internet Protocol) for addressing and routing, ensuring efficient data delivery. Through this architecture, HTTP facilitates the transfer of various forms of content (HTML files, images, API responses, and more) shaping the rich digital experiences we navigate daily. The protocol follows a structured **request-response model**, defining:

- How clients formulate and send requests for resources
- How servers process and respond to these requests

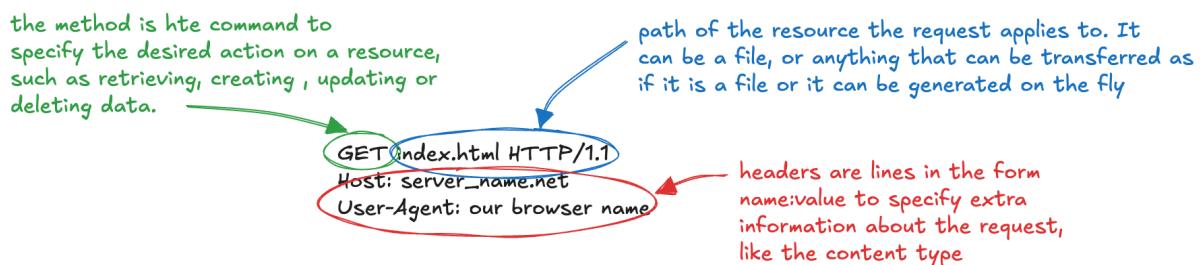
By adhering to this standardized framework, HTTP ensures interoperability, allowing diverse systems to communicate seamlessly. While it typically operates over TCP port 80, it can function on other ports as required. Understanding HTTP is crucial in the Web of Things because it enables seamless communication between interconnected devices, allowing them to exchange data, interact with web services, and integrate with cloud platforms using standardized, web-friendly protocols.

## 5.1 Basic features

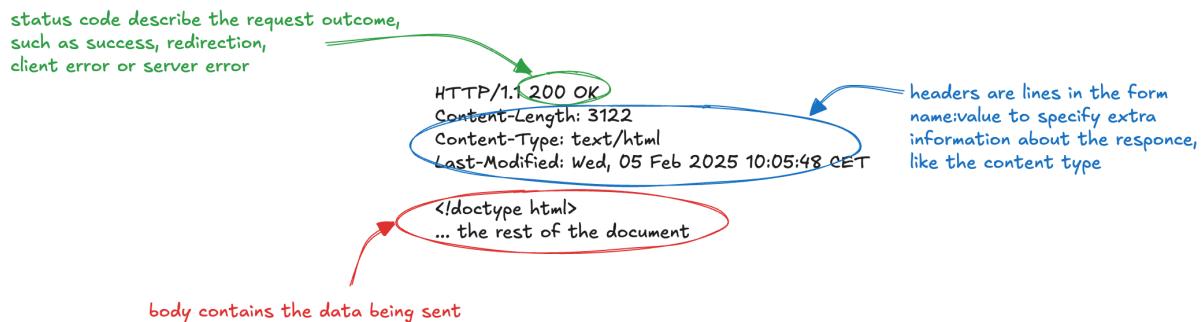
HTTP is base on a **client-server architecture**. A client, such as a web browser, mobile application, or API client, initiates a request, while a server, typically hosting a website or web service, processes the request and returns a response.



The communication begins when the client sends an HTTP request. This request specifies an action using an HTTP method, such as GET for retrieving data or POST for sending data. It also includes a URL that identifies the resource, headers that provide additional metadata like authentication or content type, and, in some cases, a body containing data, particularly for methods like POST or PUT:



Once the server receives the request, it processes the information, retrieves or modifies the requested resource, and prepares a TTP response. This response consists of a status code that indicates the outcome of the request, headers with details such as caching policies or content type, and a body that may contain the requested data, such as an HTML page, a JSON response, or an image:

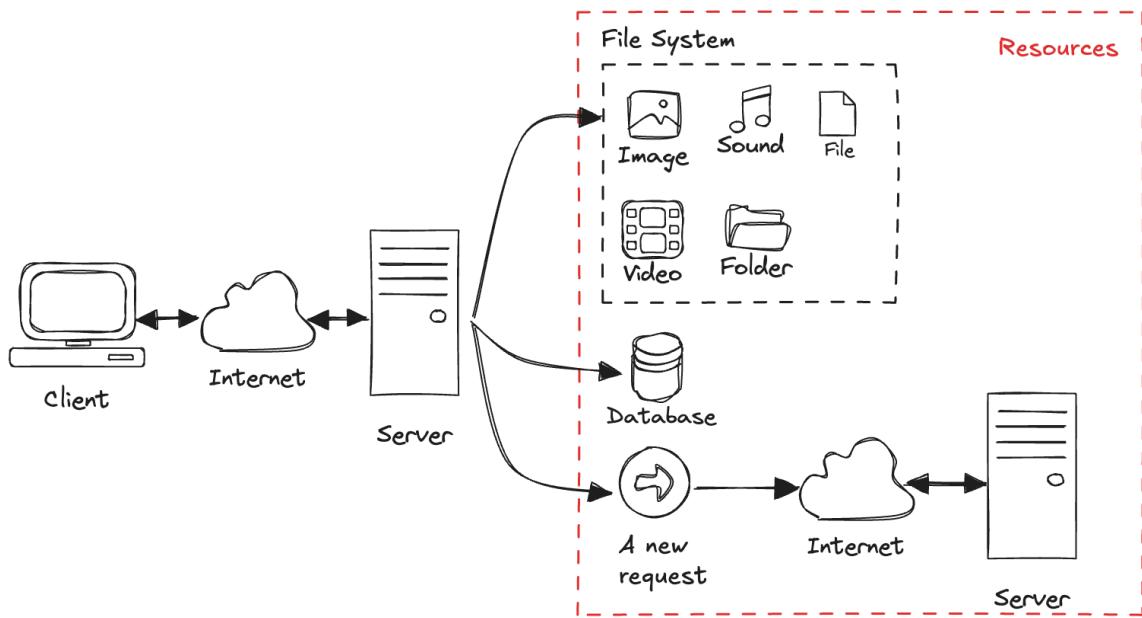


A defining characteristic of the HTTP client-server model is its **statelessness**. Once the server processes the request and sends back a response, the connection is closed, and the client disconnects. The client and server are only aware of each other for the duration of that specific request-response interaction. Moreover, it is a **stateless** communication model, since the server and client recognize each other only during the active request. Once the transaction is complete, both parties forget about the interaction, and no session data is retained between different requests. This design ensures simplicity and scalability but requires additional mechanisms like cookies or tokens for maintaining session state when needed. Another important feature is **media independence**. HTTP can transmit any type of data as long as both the client and server can process it correctly. The only requirement is specifying the content type and ensuring proper interpretation of the data.

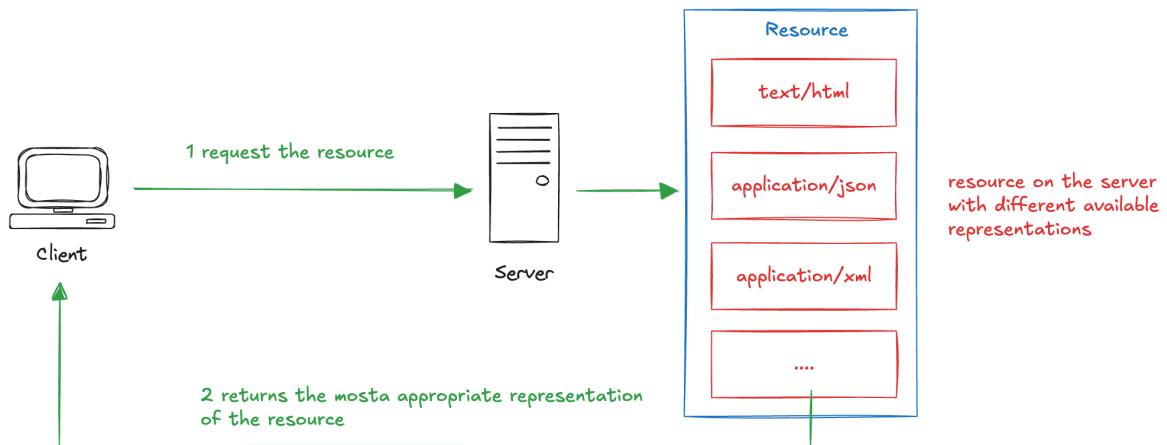
This architecture underpins everything from simple web browsing to complex API-driven applications and cloud services, making it a cornerstone of modern internet communication

### 5.1.1 Resources and URLs

A **resource** is any identifiable piece of data or service accessible on a server. It does not have a predefined nature and can be a document, an image, or any other uniquely identified data that the server can provide or manipulate:



Whether it's a webpage, a video, an image, or a file, each resource needs a **unique address** that distinguishes it from all others on the internet. This need for uniqueness and clarity gave rise to the **Uniform Resource Locator (URL)**. An URL acts as the specific address for a resource on the internet, ensuring that users can find the exact content they are looking for. Just like a physical address helps us locate a home or building in the real world, a URL points to a particular resource hosted on a server. A typical URL consists of several components:



- Protocol:** this part defines the protocol used to access the resource. For example, http, https, or ftp. It tells how to communicate with the server.
- Host:** the host identifies the server where the resource resides, typically in the form of a domain name (e.g., example.com) or an IP address. This ensures that requests are directed to the right server.

3. **Port (optional)**: in some cases, a URL may include a port number, specifying a particular service running on the server.
4. **Path**: this component specifies the exact location of the resource on the server.
5. **Query String (optional)**: this part allows passing additional parameters to the server, often used for dynamic content or search queries.

By combining these elements, a URL provides a precise and universal way to uniquely access resources on the web.

Sometimes you can see the term **Uniform Resource Identifier (URI)**. A URI is a broader concept that encompasses URLs and other identifiers. While a URL is a specific type of URI that includes the network location of a resource, a URI can refer to any unique identifier, such as a URN (Uniform Resource Name) that identifies a resource without specifying its location (like in the case of ISBN numbers for books).

### 5.1.2 Methods

**HTTP Methods** (also known as HTTP Verbs) define the **actions to be performed on a resource**. When a client sends a request to a server, it specifies the desired operation using one of the available verbs. Here are the most commonly used HTTP verbs:

- **GET** method is used to **retrieve a resource** from the server. It is a read-only operation, meaning it does not alter any data on the server. When a client sends a GET request, it expects to receive the resource (e.g., a webpage, an image, or data in JSON format) as a response.
- **POST** method is used to send data to the server to **create a resource**. It typically involves submitting form data or sending data that the server will process, such as creating a new user, submitting a comment, or uploading a file. Unlike GET, POST can change the state of the server.
- **PUT** method is used to **update an existing resource** on the server. When a client sends a PUT request, it typically provides the new data to replace the current state of the resource. If the resource does not already exist, PUT can create a new resource at the specified location.
- **DELETE** method is used to **delete a resource** on the server. When a client sends a DELETE request, the server should remove the specified resource. It is important to note that this requests are typically irreversible, and the resource will be permanently removed from the server.

- **PATCH** method is used to **apply partial updates to a resource**, rather than replacing it entirely as with PUT. This is useful when only specific fields or attributes of a resource need to be updated without modifying the entire resource.
- **HEAD** method is similar to GET, but it **only retrieves the headers of a resource**, not the actual content. This is useful when a client wants to check metadata (such as the content type, length, or modification date) of a resource without downloading the entire resource.
- **OPTIONS** method is used **to retrieve the allowed HTTP methods** (verbs) and other communication options supported by the server for a specific resource. It is often used in Cross-Origin Resource Sharing (CORS) scenarios to determine what actions are permitted on a given resource.

HTTP verbs provide **a standardized way** for clients and servers to communicate and interact with resources. Understanding the appropriate use of each verb is essential for building efficient web applications, where the correct method ensures that operations like retrieving, creating, updating, and deleting resources are performed in a consistent and predictable manner.

### 5.1.3 Status Codes

Status codes are three-digit numbers returned by the server in response to a request. These codes indicate whether the request was successful, redirected, encountered a client-side error, or failed due to a server-side issue. In the context of the WoT, these codes are crucial for ensuring seamless interaction between devices, gateways, and cloud services. They help in managing device state, data exchange, and remote control functionalities. In the following, we will explore some of the most common status codes and their significance:

**1xx - Informational Responses:** indicate that the request has been received and is being processed.

- **100 Continue:** the server acknowledges the initial request and expects the client to continue sending the rest.
- **101 Switching Protocols:** the server agrees to switch protocols as requested by the client, useful for switching to WebSockets.
- **103 Early Hints:** used to provide preliminary responses before the final status code, which can be beneficial for optimizing data retrieval in constrained environments.

**2xx - Success:** indicate that the request was successfully received, understood, and processed.

- **200 OK:** the request was successful, and the response contains the requested data, often used for retrieving sensor readings.

- 201 *Created*: a new resource, such as a settings for a physical device, was successfully created.
- 204 *No Content*: the request was processed, but there is no content to return, which is useful for confirming remote actuation commands.

**3xx - Redirection:** indicate that further action is needed to complete the request.

- 301 *Moved Permanently*: the requested resource has been permanently moved to a new URL, applicable when endpoints change.
- 302 *Found*: the requested resource is temporarily available at a different URL, useful for dynamic device configuration.
- 304 *Not Modified*: the resource has not changed since the last request, reducing network overhead in frequent data exchanges.

**4xx - Client Errors:** indicate an error caused by the client's request.

- 400 *Bad Request*: the server cannot process the request due to malformed syntax, often seen when devices send corrupted data.
- 401 *Unauthorized*: authentication is required to access the resource, ensuring security in interactions.
- 403 *Forbidden*: the client does not have permission to access the requested resource, useful for access control.
- 404 *Not Found*: the requested resource could not be found.
- 429 *Too Many Requests*: the client has sent too many requests in a given timeframe, helping to manage rate limits.

**5xx - Server Errors:** indicate that the server encountered an error while processing the request.

- 500 *Internal Server Error*: a generic error indicating an unexpected issue on the server.
- 502 *Bad Gateway*: the server received an invalid response from an upstream server, impacting cloud-to-edge communications.
- 503 *Service Unavailable*: the server is temporarily unable to handle the request, often due to maintenance or overload.
- 504 *Gateway Timeout*: the server did not receive a timely response from an upstream server, a common issue in constrained networks.

Understanding HTTP status codes is essential for debugging web applications and ensuring proper communication between clients and servers.

### 5.1.4 Headers

Headers are key-value pairs included in requests and responses, providing metadata about the communication. Headers can be categorized into different types based on their function:

- **General Headers:** apply to both requests and responses, providing general information not related to the data transmitted. An example is *Cache-Control* that specifies caching policies to reduce redundant network requests, optimizing data exchanges
- **Request Headers:** provide additional information about the request being made. Some examples are: *Accept* indicates the response content types a client wants to receive, useful for asking specific formats like JSON or XML; *Authorization* carries authentication credentials; *User-Agent* identifies the client making the request; *Content-Type* specifies the format of the data being sent, ensuring interoperability;
- **Response Headers:** provide details about the server's response. Examples are: *Access-Control-Allow-Origin* defines allowed origins for cross-origin resource sharing (CORS), enabling secure API access in distributed networks; *Server* provides details about the server handling the request, useful for diagnostics and debugging; *Content-Length* specifies the size of the response body, helping clients manage memory and bandwidth efficiently.
- **Entity Headers:** contain metadata about the body of the request or response. Examples are: *ETag* provides a unique identifier for a resource version, optimizing caching and reducing redundant data transfers in WoT applications; *Last-Modified* indicates the last modification time of a resource, assisting in conditional requests to minimize unnecessary updates.
- **Security Headers:** enhance security and protect against attacks. Examples are: *Strict-Transport-Security (HSTS)* enforces HTTPS connections, protecting communications from man-in-the-middle attacks; *X-Content-Type-Options* prevents MIME-type sniffing, reducing the risk of data tampering in IoT data exchanges; *X-Frame-Options* protects against clickjacking attacks, ensuring secure UI interactions in web-based IoT dashboards.

A header can also include a **directive**, which is an instruction that determines how requests and responses should be handled, influencing behaviors like caching, security policies, and connection management by setting rules for both clients and servers to follow. Directives are specified using a name followed by an equal sign and a value, with components separated by semicolons. For instance, the *Cache-Control* header can include directives like *max-age* to define the maximum duration a response can be cached, or *no-cache* to indicate that the response should not be stored in the cache.

By leveraging appropriate HTTP headers, developers can enhance the reliability and security of IoT applications. For a comprehensive list of registered HTTP headers, refer to the IANA HTTP Headers Registry. **IANA (Internet Assigned Numbers Authority)** is an organization

responsible for managing key elements of the global internet infrastructure. It oversees the allocation of IP addresses, DNS root zone management, and the registration of protocol parameters, including HTTP headers, status codes, and other internet standards. IANA ensures that these elements are uniquely assigned and consistently maintained to support seamless internet communication.

### 5.1.5 Message Body

The message body is used **to carry the entity representation associated with the request or response**. This means it contains the actual data transmitted by the client or server, such as form data, file uploads, images, and other content. Despite its importance in conveying useful information, the message body is optional, and its inclusion depends on the nature of the HTTP transaction being performed.

HTTP itself is **agnostic** about the format of the message body. It does not impose any specific structure or type on the content; instead, the details are specified by headers like "Content-Type" and "Content-Length." These headers inform the recipient about the nature and size of the data, ensuring that it is processed appropriately regardless of its format. One widely recognized standard for indicating the nature and format of the data is **MIME (Multipurpose Internet Mail Extensions)**. MIME provides a system to identify the type and subtype of the data, which are concatenated with a slash, such as text/plain, text/html, image/jpg, or application/json. This standard supports various **primary types** like application, audio, font, text, image, and video, among others. Further details about MIME types can be found in the official registry. By leveraging these MIME types, both clients and servers can ensure that the transmitted content is interpreted and rendered correctly, maintaining a seamless exchange of information across diverse web technologies.

### 5.1.6 Example

Below is an example illustrating an HTTP request followed by an HTTP response. In this scenario, the client sends a POST request to submit form data, and the server responds with an HTML page confirming the submission. The HTTP request including the method, URL, headers, and body is:

```
POST /submit-form HTTP/1.1
Host: www.example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 27
```

```
name=John+Doe&age=30
```

In the request, the client is submitting form data with two parameters: a name and an age. The "Content-Type" header specifies that the body is URL-encoded form data, while "Content-Length" indicates the size of the message body in bytes. The response from the server might look like this:

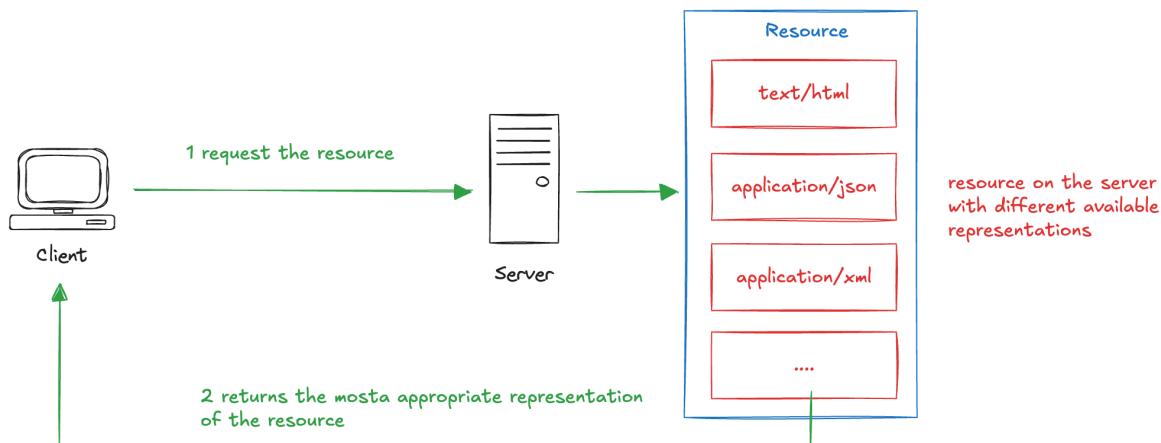
```
HTTP/1.1 200 OK
Date: Wed, 05 Feb 2025 15:30:00 GMT
Server: Apache/2.4.41 (Ubuntu)
Content-Type: text/html; charset=UTF-8
Content-Length: 123

<html>
<head>
  <title>Submission Confirmation</title>
</head>
<body>
  <h1>Form Submitted Successfully</h1>
  <p>Thank you, John Doe. Your submission has been received.</p>
</body>
</html>
```

In the response, the server indicates a successful transaction with a 200 OK status code. It also provides the "Content-Type" header to describe the nature of the returned content (an HTML document) and a "Content-Length" header that tells the client the size of the message body. The body itself contains a simple HTML document that confirms the form submission.

## 5.2 Content Negotiation

Content negotiation is a mechanism that **allows the client and server to agree** on the most appropriate representation of a resource when multiple representations are available:



When a client makes a request, it can include several headers—such as "Accept," "Accept-Language," "Accept-Charset," and "Accept-Encoding"—to indicate its preferences regarding the media type, language, character set, and encoding format of the response. These headers enable the client to inform the server about the formats it supports or prefers, such as "Accept: application/json" for JSON data or "Accept-Language: en-US" for U.S. English content. An important aspect is the use of **preference order**. Clients can express their preferences using quality factors (q-values), which assign a relative weight to each option. The quality factor is a decimal value between 0 and 1, with a higher value indicating a higher preference. For example, a client might include an "Accept" header like this:

```
Accept: text/html;q=0.9, application/json;q=0.8, application/xml;q=0.7
```

In this header, the client is indicating that it prefers HTML content over JSON, and JSON over XML. The order of preference is determined by these q-values, regardless of the order in which the media types are listed in the header. Upon receiving the request, the server examines these headers and **selects the best representation of the requested resource that matches the client preferences**. For instance, if a resource is available in multiple languages, the server can choose the version that aligns with the language specified in the "Accept-Language" header. Here is an example to illustrate this:

```
GET /api/data HTTP/1.1
Host: www.example.com
Accept: text/html;q=0.9, application/json;q=0.8, application/xml;q=0.7
```

In this request, the client expresses a preference for HTML, followed by JSON, and then XML. Assuming the server does not support an HTML representation but can serve either JSON or XML, it will choose JSON because of the higher quality factor. The server's response might look like this:

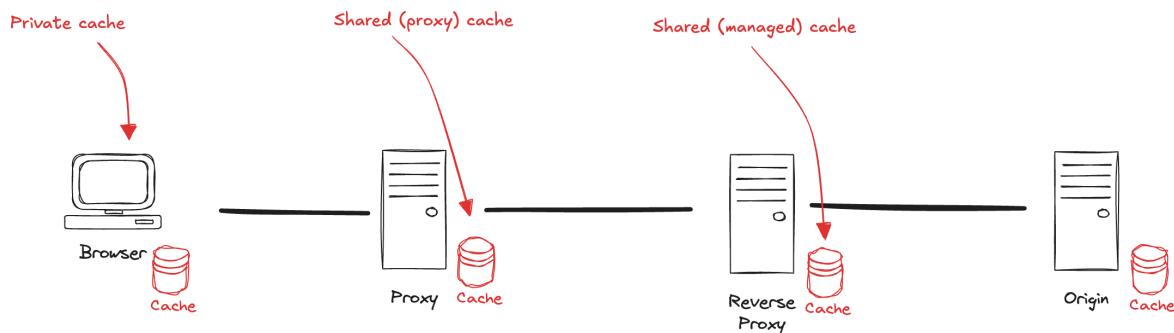
```
HTTP/1.1 200 OK
Date: Wed, 05 Feb 2025 15:45:00 GMT
Server: ExampleServer/1.0
Content-Type: application/json
Content-Length: 85

{
  "status": "success",
  "data": {
    "id": 123,
    "name": "Sample Data",
    "description": "This is an example of content negotiation."
  }
}
```

This negotiation process is critical in ensuring that the content delivered to the client is both compatible and optimal, thereby enhancing user experience. Moreover, the server may also consider factors like its own capabilities and the availability of different resource variants. In cases where there is no suitable match, the server might respond with a 406 Not Acceptable status code, indicating that it cannot provide a representation that meets the client's criteria. Content negotiation, plays an important role in web interactions by facilitating dynamic, flexible content delivery that caters to a diverse range of client requirements and contexts.

### 5.3 Caches

Caching improves performance by **temporarily storing copies of frequently accessed content**, reducing the need for repeated requests to a server. This leads to faster load times and lower bandwidth consumption. The HTTP cache stores a copy of a response associated with a request and reuses it for subsequent requests, offering several advantages. First, since the request doesn't need to reach the origin server, response times improve significantly, especially when the cache is closer to the client. Additionally, reusable responses reduce the processing load on the origin server. The server no longer needs to parse and route the request, query a database, or access the file system to retrieve content. This optimization minimizes server strain and enhances scalability. Caching can be implemented by the client, the server (origin), an intermediate node such as a proxy, or any combination of these:



Caching relies upon three primary concepts: **freshness**, **validation**, and **invalidation**. Freshness is a relative measure between the time the resource was cached and its expiry date. Validation can be used to check to see whether a cached resource is still valid after it expires, or becomes stale. Invalidation is the intention of setting a cache to stale, and it normally happens as a side effect of executing an operation that is not safe.

### 5.3.1 Client-side caching

When a client first retrieves cacheable resources, such as a web page, those resources are stored in the local cache. On subsequent visits, the client loads the cached resources from local storage instead of downloading them again from the origin server. This reduces bandwidth usage and server processing, improving overall performance. Client-side caches, which operate at the client level, are **private** and dedicated to a single user. Since their stored responses aren't shared, they can safely cache personalized content. However, if personalized data were mistakenly stored in a shared cache, other users might access it, leading to potential information leaks. To prevent this, when a response contains personalized content and should be stored only in a private cache, the server must include the *private* directive in the response headers.

```
Cache-Control: private
```

### 5.3.2 Server-side caching

Server-side caching relies on an intermediate node or alternative server to store responses from the origin server, reducing the need for repeated processing. This approach helps lower server load and decreases overall request latency. A common example is a **reverse proxy**, such as NGINX, which sits near the origin server, forwards client requests to the appropriate backend, and returns cached responses when available. This type of cache is typically **shared**, meaning multiple clients can receive the same cached resource. As a result, if one client has recently

requested a resource, others can retrieve it from the cache instead of making an identical request to the origin server, improving efficiency. Performance improvements extend beyond individual clients. No matter where caching is implemented, it provides **network-wide benefits**. When a client retrieves data from a cache, latency decreases, and bandwidth usage is reduced—not just for the client but for all nodes along the entire request/response chain.

```
Cache-Control: public
```

### 5.3.3 Cache-Control headers

The Cache-Control header is used to control caches in clients and intermediate services. Several directives are used to specify the caching behavior. Some of the most common are:

The **no-cache directive** instructs caches to revalidate a stored response with the origin server before serving it to the client, ensuring that outdated content is not used without verification. When sent by a client, it forces the cache to check for an updated version before delivering the response, often triggering a resource reload. Additionally, a server can use the **no-store directive** to prevent the response from being stored in any cache altogether.

```
Cache-Control: no-cache  
Cache-Control: no-store
```

The **max-age directive** allows a server to specify how long a response remains **fresh** in a cache, measured in seconds from the moment it is generated. This directive tells caches how long they can serve the content without revalidating it with the origin server. Importantly, the freshness duration is calculated from the time of generation, not when the response is received, meaning any delay due to network transit or intermediate caching reduces the remaining time. Additionally, a client can use the same directive to indicate that it will accept cached responses that are still within the specified freshness period.

The **max-stale directive** allows a client to specify that it will accept stale responses for a certain number of seconds beyond their expiration. Unlike max-age, which defines how long a response remains fresh, max-stale applies only after the max-age period has expired. This directive is useful when the origin server is temporarily unavailable, allowing the client to retrieve slightly outdated responses instead of failing to load content:

Several other directives, such as **min-fresh**, **must-revalidate**, and **proxy-revalidate**, can be used to further control caching behavior. These directives offer additional flexibility in managing cache lifetimes, maintaining data integrity, and optimizing performance. Additionally, multiple

directives can be included in the header and if conflicting directives are specified, the most restrictive combination will be applied.

There are other headers that can be used to control caching behavior, such as the **Expires header**, which specifies an absolute date and time after which the response is considered stale and should no longer be cached. It indicates when the content is no longer fresh, prompting clients or caches to revalidate or fetch a new copy from the origin server. The value of the header is typically a date in GMT format. It is an older caching mechanism and it has largely been replaced by the **Cache-Control header**, which offers more flexibility and better handling of cache directives.

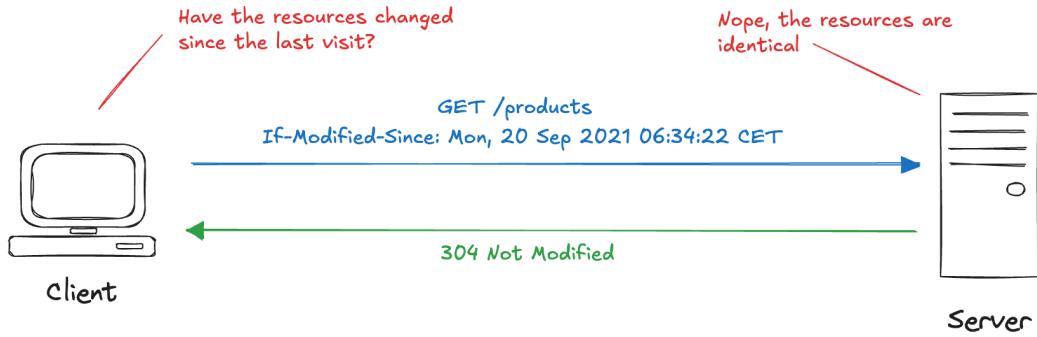
## 5.4 Conditionals

Conditional requests allow a client to **request a resource only if certain conditions are met**, typically to avoid unnecessary data transfer when the resource has not changed. These conditions are specified through headers that compare the current state of the resource with the version stored by the client or cache. Conditional requests improve efficiency by reducing bandwidth usage and speeding up interactions between clients and servers. The most common conditional headers are:

### 5.4.1 Last-Modified and If-Modified-Since headers

The **Last-Modified header** indicates the date and time that the server believes the resource was last modified:

This information can be used as a condition, or validator in a conditional request. When a client makes a request, it can include an **If-Modified-Since header** with the value of the Last-Modified header from a previous response. If the resource has not been modified since that date, the server can respond with a **304 Not Modified** status code, indicating that the client's cached copy is still valid. This process helps reduce unnecessary data transfer and speeds up interactions by avoiding the retransmission of unchanged resources.

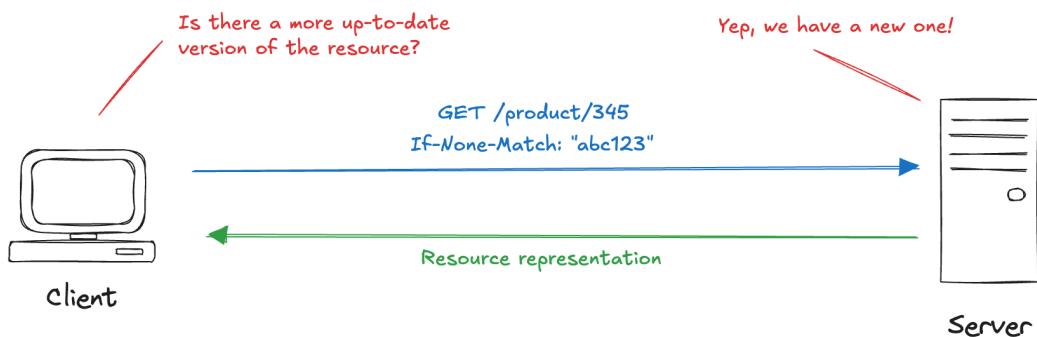


#### 5.4.2 ETag and If-None-Match header

The **ETag (Entity Tag) header** is used to provide a unique identifier for a **specific version** of a resource. It acts as a fingerprint for the content, allowing caches and clients to efficiently determine if the content has changed since the last request. The ETag is typically a hash or string that represents the state of the resource. When a client first requests a resource, the server includes an ETag in the response header:

On subsequent requests, the client sends the ETag value in the **If-None-Match header**.

The server compares the ETag in the request with the current version of the resource. If the ETag matches, it means the content hasn't changed, so the server can return a **304 Not Modified** response, indicating the cached version is still valid. If the ETag doesn't match, the server sends a new version of the resource with a new ETag. This process helps reduce unnecessary data transfers and improves performance by only sending updated content when needed.



## 5.5 Authentication

**Authentication** is the process by which a server verifies the identity of a client before granting access to protected resources. It ensures that only authorized users can access specific content or perform certain actions. We will discuss better this point in the security section. However, it is important to mention that HTTP provides several mechanisms for authentication, each with different levels of security and complexity. Some of the most common authentication methods include:

### 5.5.1 Basic Authentication

This is one of the simplest forms of authentication, where the client sends the username and password in the **Authorization header** as a base64-encoded string. The format is:

A base64-encoded string is a way of **encoding binary data** into an ASCII string format using a set of 64 different printable characters:

- Uppercase letters (A-Z)
- Lowercase letters (a-z)
- Digits (0-9)
- Two special characters, usually + and / in some URL-safe encodings, they may be replaced with - and \_

It is used to encode data that needs to be transmitted over text-based protocols (like HTTP), where binary data may not be suitable. It works by dividing the binary data into chunks of 6 bits (since  $2^6 = 64$ , corresponding to the 64 characters of the alphabet), and then mapping these chunks to the corresponding characters. The result is a string that represents the binary data but is encoded in a text-friendly format.

While easy to implement, Basic Authentication is **not very secure**, as the credentials can be easily decoded and are transmitted in every request.

### 5.5.2 Digest Authentication

Digest Authentication improves security by **hashing the password** before sending it over the network. This prevents the password from being transmitted in clear text. However, it still has limitations in protecting against certain types of attacks and is generally considered less secure than more modern methods.

### 5.5.3 Bearer Token Authentication

Bearer tokens allow clients to authenticate using an access token instead of a password. The token can be scoped to specific actions or resources and have **expiration times**, adding a layer of security. The token is sent in the Authorization header as follows:

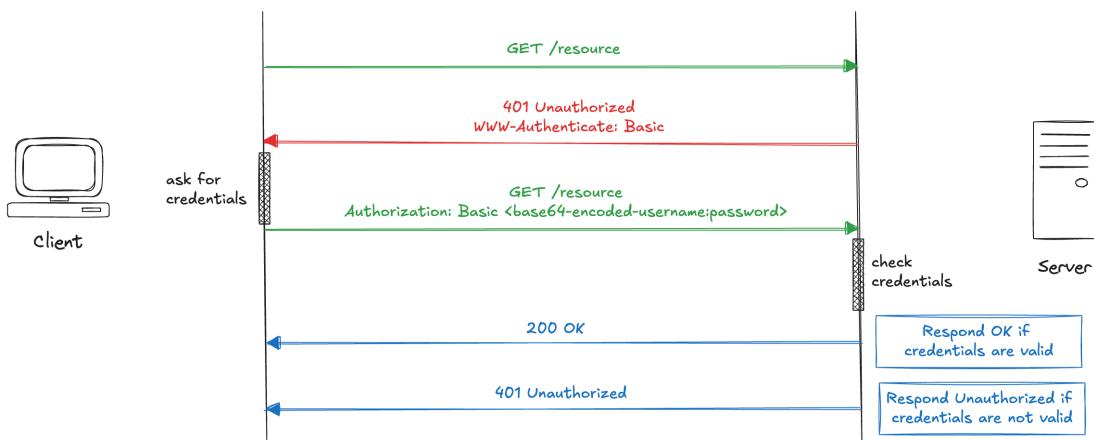
The server uses the token to verify the client's identity and grant or deny access to the requested resource. This method is widely used for web services, as it enables users to **authenticate without exposing their password**. The server doesn't need to know who the client is beforehand. If the client sends a valid bearer token, the server assumes the client is authorized to access the requested resource. However, since bearer tokens are like keys, they must be kept secure.

### 5.5.4 Certificate-Based Authentication

In this method, the client uses a **digital certificate** to authenticate itself to the server. The server verifies the client's certificate, often through a **trusted certificate authority**. This method is more secure than password-based authentication because it involves cryptographic techniques, but it can be more complex to implement and manage.

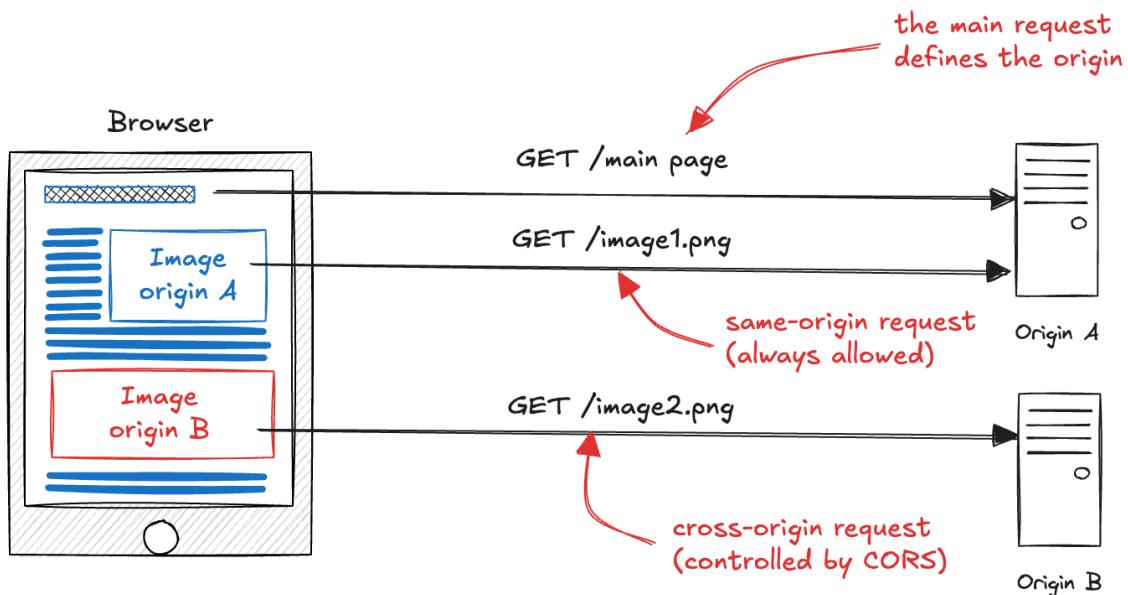
### 5.5.5 Server response

When a protected resource is requested, the server sends an **HTTP 401 Unauthorized** response, prompting the client to authenticate. The server includes a **WWW-Authenticate** header specifying the authentication methods it supports. The client then responds with the appropriate Authorization header containing the credentials or token. If the credentials are valid, the server grants access to the resource.



## 5.6 Cross-Origin Resource Sharing (CORS)

In order to prevent possible security threats, web browsers enforce a **same-origin policy** that restricts how scripts and resources from one origin can interact with resources from another origin. For example, suppose we are browsing a page on "www.wikipedia.org" and try to fetch some data from a different origin, like "www.google.com". Our Browser will abort the request. The policy allows us to load data from the only same origin we are currently browsing. This helps us to protect our privacy. Suppose we were browsing a webpage on "google.com" and made a GET request on "facebook.com". If same origin policy didn't exist, as at that time we were browsing google.com, it could possibly collect our data we were fetching from "facebook.com". However, there are legitimate use cases where an application needs to request resources from a different origin, such as when an app hosted on one domain wants to fetch data from a service on another domain. CORS provides a mechanism for servers to grant permissions for such cross-origin requests.



### 5.6.1 Access-Control-Allow-Origin header

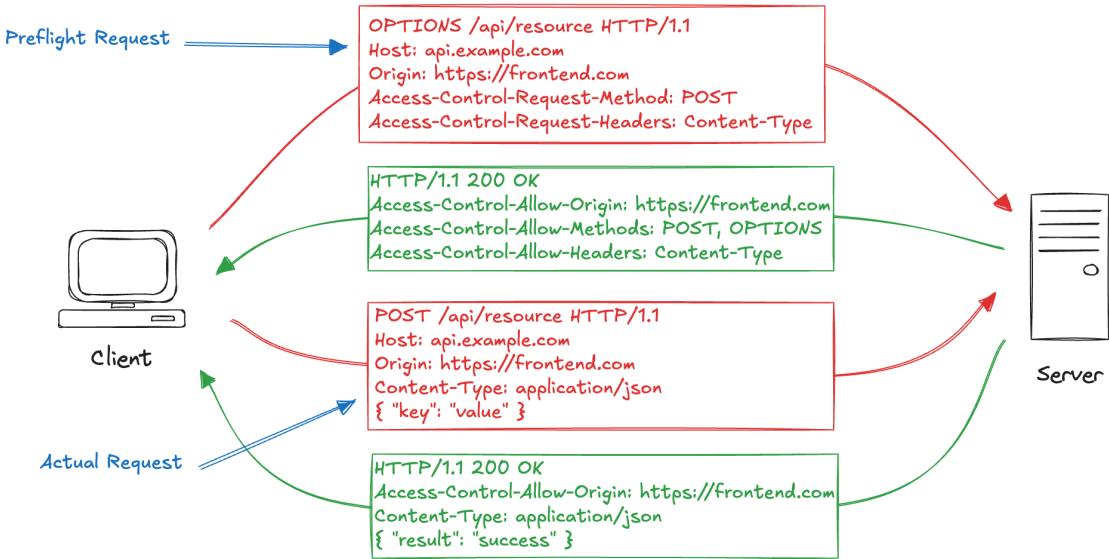
The **Access-Control-Allow-Origin** header is used by servers to specify which origins are permitted to access their resources. It can be set to different values depending on the desired level of access. When set to `*`, it allows requests from any origin, making the resource publicly accessible, which is useful for open APIs or publicly available content. If a specific **origin** is provided, only that origin is allowed to access the resource. Setting the value to `null` indicates that the

resource is not accessible from any origin, effectively blocking all cross-origin requests.

This header is optional. If we have a server and don't want any origin to fetch resources from it, we simply omit the header. By not specifying it, we default to the same-origin policy, where only our own domain can interact with the server and retrieve data. However, this is not always the most common situation. The **Access-Control-Allow-Origin** header allows developers to make exceptions when they know that requests from other domains are safe and expected. This header is always set in the response and controlled by the server that owns the data. The reason this policy is so secure is that it follows a **whitelisting** approach (the opposite of **blacklisting**). With whitelisting, we specify a list of allowed privileges, and everything else is blocked by default.

### 5.6.2 Other CORS headers

CORS involves several other request and response headers that regulate how resources are shared across different origins. On the **request side**, when a client makes a cross-origin request, the browser can send a **preflight request** using the **HTTP OPTIONS method**. This request includes headers such as **Access-Control-Request-Method**, which indicates the method the client intends to use, and **Access-Control-Request-Headers**, which lists any additional headers the client wants to include in the actual request. On the **response side**, the server can specify access rules through several headers. The **Access-Control-Allow-Methods** header specifies the HTTP methods the server permits for cross-origin requests. Similarly, the **Access-Control-Allow-Headers** header lists any custom headers the server allows in cross-origin requests. If a preflight request is sent, the server can also include the **Access-Control-Max-Age** header to indicate how long the browser should cache the preflight response, reducing the need for repeated preflight checks. These headers work together to provide a controlled and secure mechanism for handling cross-origin requests, ensuring that only authorized origins can access protected resources while preventing unauthorized or malicious interactions. This is an example of flow of a CORS request:



CORS is an important mechanism that allows secure cross-origin requests while protecting data and privacy. By enforcing security restrictions based on origins, it prevents potentially harmful interactions between websites and ensures that only authorized domains can access certain resources. It's important for developers to correctly configure CORS headers on the server to enable legitimate cross-origin requests while maintaining security.

## 5.7 JSON

HTTP is indifferent to the type of data it transfers, meaning it can handle any format as long as both the client and server understand how to process it. However, in the context of the Web of Things (WoT), one of the most widely used formats is **JSON (JavaScript Object Notation)**. JSON is a lightweight, text-based format for representing structured data, derived from JavaScript object syntax. It is commonly used for data exchange in web applications, such as transmitting information from a server to a client for display on a web page or sending data from a client to a server for processing. While JSON closely resembles JavaScript object literals, it is language-independent and widely supported across various programming environments, making it a universal choice for structured data exchange in WoT applications.

### 5.7.1 Structure

JSON is represented as a string and must be converted into a native language object (e.g., a JavaScript object) to access its data. This process is called **deserialization**, while the reverse operation (converting a native object into a JSON string for transmission over a network) is known

as **serialization**. JavaScript provides a built-in **JSON object** with methods for performing these conversions, and virtually all modern programming languages offer libraries or built-in functions for parsing JSON strings into native objects and generating JSON from structured data. A JSON string can be stored in a standalone file, typically with a ".json" extension and it is represented by the MIME type **application/json**. The format follows JavaScript object literal syntax, consisting of key-value pairs enclosed in curly braces. Keys are always strings, while values can be strings, numbers, booleans, arrays, objects, or null, with key-value pairs separated by colons and entries separated by commas. Here is an example of a JSON object:

```
{  
  "device": "temperature_sensor",  
  "location": "living room",  
  "temperature": 22.5,  
  "humidity": 45,  
  "status": "active",  
  "last_updated": "2025-02-06T10:15:30Z"  
}
```

JSON's structure enables the efficient representation and transmission of structured data, making it a fundamental format for data exchange in modern web applications and interconnected systems. When loaded into a JavaScript program and parsed into a variable, the data can be accessed using standard JavaScript dot or bracket notation. For example:

```
// Parsing a JSON string into a JavaScript object  
let data = JSON.parse('{"device": "sensor", "temperature": 22.3}');  
  
// Accessing properties of the JavaScript object  
console.log(data.device); // Output: sensor  
console.log(data.temperature); // Output: 22.3  
  
// Serializing a JavaScript object into a JSON string  
let jsonData = JSON.stringify(data); // Output:  
↪ '{"device": "sensor", "temperature": 22.3}'
```

JSON is a text-based format that is **easy for humans to read and write**. Its structure of key-value pairs and arrays, enclosed in curly braces and square brackets, makes it intuitive to understand, even without specialized tools. However, it is also **easy for machines to parse and generate**, since it is language-independent, it can be used across different programming languages and platforms.

### 5.7.2 Arrays

Arrays are ordered collections of values, enclosed within square brackets. Unlike objects, which are key-value pairs, an array contains a list of values that can be of various types, including strings, numbers, booleans, other arrays, objects, or even null. Each item in the array is separated by a comma. One of the main advantages of JSON arrays is that they allow for easy representation of lists or sequences of related data, such as multiple readings from a sensor or a collection of user records. Arrays can hold elements of different types, making them versatile for representing a variety of structured data. Here is an example of a JSON array containing several types of data:

```
{  
  "device": "sensor",  
  "readings": [22.3, 22.4, 22.5, 22.6],  
  "active": true,  
  "locations": ["living room", "bedroom", "kitchen"]  
}
```

In the example, the “readings” key holds an array of numeric values representing temperature readings taken by a sensor, and the “locations” key holds an array of strings, each representing a different room in a house.

Arrays are indexed by zero, and you can access individual elements by referring to their index, like this:

```
console.log(deviceData.readings[0]); // Output: 22.3  
console.log(deviceData.locations[2]); // Output: "kitchen"
```

### 5.7.3 Nested structures

JSON supports the nesting of objects and arrays. This enables the creation of **complex hierarchical data structures**. For instance, a JSON object can contain another JSON object or an array, which in turn may contain objects or other arrays. This allows for easy modeling of real-world data in a structured way. Example of a nested object and array in JSON:

```
{  
  "device": "temperature_sensor",  
  "readings": [  
    { "time": "2025-02-06T10:00:00Z", "value": 22.3 },  
    { "time": "2025-02-06T10:05:00Z", "value": 22.4 },  
    { "time": "2025-02-06T10:10:00Z", "value": 22.5 }  
  ]  
}
```

```
{ "time": "2025-02-06T10:15:00Z", "value": 22.4 }
],
"location": {
  "room": "living room",
  "floor": 1
}
}
```

In this case, the “readings” array contains objects, each representing a reading with multiple properties, such as “time” and “value”, while “location” is a nested object with keys for “room” and “building”. This demonstrates how JSON can represent complex data structures with multiple levels of nesting, making it a versatile format for a wide range of applications.

#### 5.7.4 Schema

A Schema is a tool used for **validating the structure** of JSON data. It provides a way to define the expected structure, types, and constraints for the data that will be transmitted in JSON format. By using a schema, developers can ensure that the data sent or received conforms to a predefined format, making it easier to catch errors and validate input before processing. Here is an example of a schema that validates a simple JSON object representing a temperature sensor:

```
const schema = {
  "$schema": "http://json-schema.org/draft-07/schema#",
  "type": "object",
  "properties": {
    "sensor_id": {
      "type": "string",
      "minLength": 1
    },
    "location": {
      "type": "string",
      "minLength": 1
    },
    "temperature": {
      "type": "number",
      "minimum": -50,
      "maximum": 150
    },
    "humidity": {
      "type": "number",
      "minimum": 0,
      "maximum": 100
    }
}
```

```
    "minimum": 0,
    "maximum": 100
  },
  "timestamp": {
    "type": "string",
    "format": "date-time"
  },
  "required": ["sensor_id", "temperature", "timestamp"],
  "additionalProperties": false
};
```

the "\$schema" field specifies the version of the JSON schema being used, while the "type" field defines the type of the object. The "properties" field lists the expected properties of the object, along with their types and any additional constraints. In this case, the schema specifies that the object should have a "sensorId" property of type "string" and a "temperature" property of type "number". The "required" field indicates that both properties are mandatory. By using schema, it becomes much easier to ensure the integrity of the data being sent. The schema enforces rules like valid temperature ranges, correct timestamps, and the presence of required fields, making it easier to automate data validation, reduce errors, and improve communication between devices and systems. There exists several libraries and tools that can be used to validate JSON data against a schema, for example the AJV library in JavaScript:

```
const Ajv = require('ajv');

// Create an Ajv instance and compile the schema
const ajv = new Ajv();
const validate = ajv.compile(schema);

// Example sensor data to validate
const sensorData = {
  "sensor_id": "sensor123",
  "location": "Living Room",
  "temperature": 200,
  "humidity": 45,
  "timestamp": "2025-02-06T10:15:30Z"
};

// Validate the data
const valid = validate(sensorData);
```

```
if (valid) {  
    console.log("Sensor data is valid!");  
} else {  
    console.log("Sensor data is invalid!");  
    console.log(validate.errors); // Display validation errors  
}
```

In the example, the temperature value is outside the valid range specified in the schema, so the validation fails:

```
[  
  {  
    "keyword": "maximum",  
    "dataPath": ".temperature",  
    "schemaPath": "#/properties/temperature/maximum",  
    "params": {  
      "comparison": " $\leq$ ",  
      "limit": 150,  
      "exclusive": false  
    },  
    "message": "should be  $\leq$  150"  
  }  
]
```

## 5.8 Interacting with HTTP

Interacting with HTTP involves sending requests to servers and processing the responses. This can be done using various programming languages and tools, each offering different levels of abstraction and control. We can consider how the browser interacts with the server when we visit a webpage, then how it can be customized using JavaScript and finally a tool (Postman) which allow a complete control over the construction of a request.

### 5.8.1 From the Browser

The interaction between HTTP and a **browser** is fundamental to how web pages are requested, processed, and rendered. When a user enters a URL into the browser's address bar, the browser sends an HTTP GET request to the specified server to retrieve the corresponding resource, typically an HTML document. This request follows the structure of the HTTP protocol, where the browser asks for a resource and the server responds with the requested data. For example:

```
GET /index.html HTTP/1.1
```

Once the browser receives the HTML content from the server, it begins to parse the HTML code and determine if there are any additional resources referenced within the page, such as images, JavaScript files, CSS stylesheets, or other assets. If the HTML references these external files, the browser will make additional GET requests for each resource **concurrently** (to speed up page loading times) ensuring that the entire page is displayed quickly and efficiently. For instance, if the HTML contains:

```

<script src="app.js"></script>
<link rel="stylesheet" href="style.css">
```

The browser will initiate separate GET requests for "logo.png", "app.js", and "style.css" to fetch these resources and render the complete webpage. In addition to retrieving resources, a browser can also submit data to a server. One of the most common ways to send data is through an HTML form. A form allows users to input information, which can then be sent to the server via a specified method. For example, the following form allows a user to input a name and a message:

```
<form method="GET" action="example/message.html">
  <p>Name: <input type="text" name="name"></p>
  <p>Message: <textarea name="message"></textarea></p>
  <p><button type="submit">Send</button></p>
</form>
```

When the user fills out the form and presses the submit button, the browser will encode the data entered into the form fields into a query string and send it to the server using the GET method. The query string is appended to the URL in the form of key-value pairs:

```
GET /example/message.html?name=Jean&message=Yes HTTP/1.1
```

In this example, the server receives the GET request with parameters (name=Jean and message=Yes), which it can process and use to generate a response, such as saving the message or displaying a confirmation page. If we change the method in the form to POST:

```
<form method="GET" action="example/message.html">
  <p>Name: <input type="text" name="name"></p>
  <p>Message: <textarea name="message"></textarea></p>
  <p><button type="submit">Send</button></p>
</form>
```

The request will use POST, and the query string will be moved to the body of the request instead of being appended to the URL:

```
POST /example/message.html HTTP/1.1
Content-length: 24
Content-type: application/x-www-form-urlencoded

name=Jean&message=Yes
```

An important HTTP convention is the use of GET for requests that do not cause side effects, such as performing a search. GET is considered **safe and idempotent**, meaning it can be applied multiple times without altering the result. In contrast, POST is used for requests that modify something on the server, like posting a message. Unlike GET, POST is neither safe nor idempotent.

### 5.8.2 From Javascript

We can interact with HTTP also from JavaScript. Traditionally, this was done using the **XMLHttpRequest object**, which was introduced by Microsoft in the 1990s when XML was widely used. However, XMLHttpRequest has a poor design as it mixes the concerns of making a request and parsing the response. Consider for example the following code and execute it in the browser console from the "http://eloquentjavascript.net/" page (see *01.HTTPRequest.js*):

```
// Create a new XMLHttpRequest object
let req = new XMLHttpRequest();

// Make a GET request to the server
req.open("GET", "example/data.txt", false);
req.send(null);

// Display the response from the server
console.log(req.status, req.statusText);
console.log(req.getResponseHeader("content-type"));
console.log(req.responseText);
```

This code demonstrates a **synchronous** request to fetch "example/data.txt" resource. However, using synchronous requests is generally discouraged because **it blocks the main thread, freezing** the entire page until the request completes. In order to solve the issue, we can use **asynchronous requests**. This is done by providing a **callback function** that will be executed when the request completes. The following code demonstrates an asynchronous request (see *02.AsyncHTTPRequest.js*):

```
// Create a new XMLHttpRequest object
let req = new XMLHttpRequest();

// Define a callback function to handle the response
function transferComplete(evt) {
    console.log("The transfer is complete.");
}

// Make a GET request to the server, providing the callback function
req.open("GET", "example/data.txt", true);
req.addEventListener("load", transferComplete);
req.send(null);

// Log a message to the console
console.log("I'm here.");
```

In that case, when we call `send()`, the request is **queued to be sent**, allowing the program to continue executing. The browser handles sending the request and receiving the response in the background. However, while the request is still in progress, we cannot access the response until the operation is complete. We must listen for the "load" event on the request object. This event is triggered when the request completes successfully. The event handler receives an event object that contains the response data.

Today XMLHttpRequest is outdated and has been replaced by the **Fetch API**, which provides a more modern and flexible way to interact with HTTP resources. The Fetch API is **promise-based**, making it easier to work with asynchronous requests. The following code demonstrates how to use the Fetch API to make a request (see *03.FetchAPI.js*):

```
// Create an async function to use await inside it
async function fetchData() {
    // Initiating a fetch request to get the content of "example/data.txt"
    let response = await fetch("example/data.txt");

    // Logging the HTTP status code, status message and content type of the response
```

```
console.log(response.status, response.statusText);
console.log(response.headers.get("content-type"));

// Reading the response body as text and logging it to the console
let text = await response.text();
console.log(text);

// Call the function
fetchData();
```

The asynchronous style of programming involves wrapping tasks that need to be executed after a request in a function and ensuring that the function is called at the appropriate time, once the request has completed. This approach allows the program to continue running other tasks while waiting for the request to finish. We will consider asynchronous approach in more detail in the context of Node.js.

### 5.8.3 Postman

Postman is a tool for testing and interacting with APIs over HTTP. It offers a user-friendly interface that enables developers to send requests to a server and inspect responses. With Postman, developers can quickly test API endpoints, simulate different request parameters, and analyze status codes, headers, and response data. Postman plays a crucial role in the development and debugging process by allowing developers to make API requests without writing code, making it easier to isolate and resolve issues. By streamlining the interaction with HTTP requests and responses, Postman enhances productivity and accelerates the development cycle.

Postman supports all HTTP request methods, including GET, POST, PUT, DELETE, PATCH, and more, allowing developers to interact seamlessly with different API endpoints. To improve organization and efficiency, Postman lets users group related requests into **collections**, making it easier to manage, reuse, and share API tests. Responses can be **viewed in multiple formats** such as JSON, XML, HTML, or plain text, with detailed insights into status codes, headers, and body content. This comprehensive visualization helps developers quickly analyze results and debug issues effectively.

Postman allows developers to create multiple **environments** and use **environment variables** to store and manage dynamic values such as API keys, base URLs, and user credentials. This eliminates the need to manually update values across multiple requests, improving efficiency and reducing errors. Instead of hardcoding URLs, tokens, or other parameters, these values

can be stored as variables and updated in a single place. This is especially useful when working across different environments (such as development, testing, and production) as users can seamlessly switch between them by selecting the appropriate environment without modifying request URLs or headers. Additionally, environment variables help manage sensitive data, keeping API keys and authentication tokens secure by preventing them from being directly embedded in requests. We can reference a variable using double curly braces in our request:

```
GET {{base_url}}/users
```

Postman allows us write JavaScript code to run **before** and **after** a request. **Pre-request scripts** are useful for modifying request data or setting up conditions, while **tests** validate the response data. As an exmaple, we can set or retrieve variables programmatically:

```
pm.environment.set("auth_token", "your_token_here");  
  
let token = pm.environment.get("auth_token");  
console.log("Current Token:", token);
```

Finally, Postman offers advanced features like **mock servers**, **API documentation**, **monitoring**, and **integration with CI/CD pipelines**. These features enhance collaboration, testing, and automation in the development process. For example we can create mock servers to simulate API responses for testing without actually hitting a live server. This is helpful for front-end developers when the backend is not yet available or we can automate testing through scripts that run assertions on the response. we can validate status codes, response times, data consistency, and more. See *04.Makers.postman\_collection.json* for an example of a Postman collection.

## 5.9 Hands-on Activity

1 - Create a Postman collection to test the API that you can find at the following link <https://jsonplaceholder.typicode.com/>. You have to post some "comments" and get a list of previous "posts".

# 6 NodeJS

Node.js is a powerful way to **run JavaScript code outside of the web browser**. Originally released in 2009, it opened up new possibilities for JavaScript, allowing developers to build server-side applications using a language that was once confined to the client side. At the heart

of Node.js is the V8 engine, the same high-performance JavaScript engine that powers Google Chrome. V8 is built with cutting-edge compiler technologies that make it incredibly fast. In fact, programs written in JavaScript and run with Node.js can achieve performance that rivals C programs, but with a significantly lower development cost and complexity. You can explore more about V8 here: <https://v8.dev>.

Node.js isn't just a JavaScript engine though. It **extends JavaScript with additional APIs** to handle things like the file system, HTTP requests, and binary data—features you wouldn't normally find in browser-based JavaScript.

One of Node's key features is its **event-driven architecture**. This design allows Node.js to handle concurrency through an event loop, rather than relying on traditional multithreading. This means developers don't need to worry about threads and locks, Node.js handles asynchronous operations behind the scenes in a very efficient way. For example, instead of waiting for a file to load from disk (which might take some time), Node lets the program continue running and calls back a function once the file is ready, just like how browser code listens for an "onclick" event. This **non-blocking, asynchronous approach** makes Node.js especially well-suited for applications that need to deal with lots of simultaneous requests, like web servers and real-time apps. The programming model is both scalable and intuitive once we understand the event loop.

Node.js can function as a **full-featured web server**, but it's also a **general-purpose runtime environment**. It includes a REPL (Read-Eval-Print Loop), which is an interactive shell where we can write and test JavaScript code line-by-line. This makes it a great environment not only for building production-ready applications but also for learning and experimenting with JavaScript in real-time.

One of the reasons Node.js has gained so much popularity is its ability to simplify the development of scalable network applications. Traditional server technologies often struggle under the weight of concurrent connections, requiring complex infrastructure and significant hardware. Node.js takes a different approach, using its event-driven, non-blocking architecture to handle a large number of simultaneous connections efficiently, without the need for heavy threading or complex concurrency logic. Major companies have experienced dramatic improvements after switching to Node.js. For instance, LinkedIn migrated its backend from Ruby on Rails to Node.js in 2012. The results were remarkable: server usage was reduced to a factor of 10, and the application ran up to 20 times faster. The switch also enabled the frontend and backend teams to collaborate more easily, using the same language across the stack. PayPal made a similar move in 2013, transitioning from Java to Node.js. Their development cycle sped up, with teams building features almost twice as fast and with fewer people involved. Performance improved significantly as well—requests per second doubled, and the average response time dropped by

35%.

## 6.1 Getting Started

To begin using Node.js, head to the official site. Installation is straightforward on most systems. Once installed, you can launch the interactive shell—called the REPL (Read-Eval-Print Loop)—just by typing “node” in the terminal. This environment is great for experimenting and learning, but it’s not intended for running full applications in production. It does offer a few useful meta-commands such as “.help”, “.exit”, and “.clear”, which can assist while testing small snippets.

### 6.1.1 Writing applications

To write real applications, use any code editor to create a JavaScript file, such as “app.js”, and then run it with the command:

```
node app.js
```

This approach allows you to build structured applications using external libraries and modules, not just quick tests. For development, there are many good options in terms of editors and IDEs. **Visual Studio Code** is a lightweight, powerful editor with great Node.js support and extensions. Other options include IntelliJ IDEA (with its Node.js plugin) and Atom, which can be extended with packages like “atom-debugger” for Node.js debugging support.

### 6.1.2 NPM: Node Package Manager

One of the biggest advantages of using Node.js is its massive ecosystem of reusable code, made easily accessible through a tool called **NPM**, short for **Node Package Manager**. NPM comes automatically installed with Node.js and serves as both a package manager and a public registry for Node modules. The official site hosts **hundreds of thousands of free packages**. These packages range from utilities for handling dates and strings to powerful web frameworks and database connectors. If there’s a problem you’re trying to solve in your Node.js app, chances are someone has already built a package for it.

### 6.1.3 Modules

In Node.js, a **module** is simply a reusable piece of code, like a library in other programming languages. Modules help keep code organized and modular by allowing us to separate func-

tionality into files and folders. Node.js includes a set of **core modules** that are available without any installation. These include modules for basic tasks such as:

- `http` – to create web servers
- `url` – for URL parsing
- `querystring` – to parse query parameters
- `path` – for file path utilities
- `fs` – to interact with the file system
- `util` – for utility functions

To install an external moduel, we use the "npm install" command. For example, to install the "upper-case" module, just run:

```
npm install upper-case
```

This will create a directory called "node\_modules" in our project folder, where the package (and any dependencies it needs) will be stored. Once the package is installed, using it in our code is simple:

```
let uc = require('upper-case');
console.log(uc.upperCase("hello world")); // Output: HELLO WORLD
```

We can also create our own **local modules** to encapsulate and reuse functionality within our project. This is done using the "exports" keyword. Here's a basic example:

```
// myfirstmodule.js

exports.myDateTime = function () {
  return Date();
};
```

Now we can use this custom module in any Node.js file like so:

```
let dt = require('./myfirstmodule');

console.log("Current date and time: " + dt.myDateTime());
```

Notice that when loading a local file module, we must use "./" to indicate that it's in the current directory.

#### 6.1.4 Manifest

Every Node.js project is centered around a special file called "package.json". Think of this file as the "manifest" of our project: it stores essential metadata that identifies the project and outlines how it should behave, what dependencies it needs, and how it can be run or used by others.

The package.json includes **descriptive metadata** such as:

- The project's name
- The current version
- A short description
- The author and license

It also includes **functional metadata** that tells Node.js how to run the app:

- The entry point file (like "index.js")
- A list of dependencies (other modules our code needs)
- Links to source code repositories or bugs reporting tools

To get started, create a new folder for our project and run:

```
npm init
```

This command will ask us a series of questions (like the project name, version, entry file, etc.) and then generate a "package.json" file based on our answers:

```
{
  "name": "test",
  "version": "0.0.1",
  "description": "This is an example",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" & exit 1"
```

```
},
"author": "Riccardo Berta",
"license": "MIT"
}
```

We can then add a dependency to our project, such as:

```
npm install upper-case
```

This installs the "upper-case" module and automatically adds it to the "dependencies" section of package.json:

```
{
  "name": "test",
  "version": "0.0.1",
  "description": "This is an example",
  "main": "index.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1"
  },
  "author": "Riccardo Berta",
  "license": "MIT",
  "dependencies": {
    "upper-case": "^2.0.2"
  }
}
```

In package.json, dependencies are categorized into two main types:

- **dependencies**: these are packages our app needs to run in production. For example, an HTTP server or a database client.
- **devDependencies**: these are packages used during development only, like testing tools, linters, or bundlers. They aren't needed in production.

When distributing our Node.js project, we typically **exclude** the "node\_modules" folder (since it can be huge) and rely on the package.json file instead. Anyone who clones our project can simply run:

```
npm install
```

This command installs all the modules listed under "dependencies" in the package.json file.

### 6.1.5 Version Control

**Semantic Versioning** is a system that uses a three-part version number to convey meaning about the changes in a package. This helps developers manage dependencies in a predictable and reliable way. A version number looks like this:

```
MAJOR.MINOR.PATCH
```

For example, in 1.2.3:

- 1 is the **major** version
- 2 is the **minor** version
- 3 is the **patch** version

A **Major** changes (1.x.x -> 2.0.0) introduce **breaking changes** that may not be backward compatible. A **Minor** changes (1.2.x -> 1.3.0) add new features but **maintain compatibility** with older versions. A **Patch** changes (1.2.3 -> 1.2.4) are **bug fixes or small improvements** that don't affect compatibility.

When we install a package with NPM, the version is recorded in package.json and to provide flexibility in updates, we can use symbols to define **version ranges**:

- `~1.2.3` means to install 1.2.3 or any later **patch** version, like 1.2.4, 1.2.5, etc, but not 1.3.0.  
We use this when we want to accept only safe bug fixes, not new features.
- `1.2.3` means to install 1.2.3 or any **minor** or **patch** updates, like 1.3.0, 1.4.1, etc., but not 2.0.0. We use this when we are OK with new features that don't break compatibility. Be cautious if the library's authors don't strictly follow semantic versioning.
- `*` means to accept **any version**, regardless of major, minor, or patch. We are telling NPM to use whatever is the latest version available. This is risky and **not recommended** for production use.
- `>=1.2.3` and `>1.2.3` means any version **higher/strictly higher** than 1.2.3. It is useful when we want a minimum version (e.g., for a feature introduced in 1.2.3).
- `<=1.2.3` and `<1.2.3` means any version **lower/strictly lower** than 1.2.3. It is useful when we want to avoid breaking changes introduced in newer versions.

Consider the following example:

```
"dependencies": {  
    "express": "^4.17.1",  
    "mongoose": "~6.0.13"  
}
```

This means:

- Accept **any 4.x.x** version of Express package starting from 4.17.1
- Accept **only 6.0.x** versions of Mongoose starting from 6.0.13

## 6.2 Synchronous vs Asynchronous

Events are a foundational aspect of the language and its runtime model. From the beginning, JavaScript has been designed with interactivity in mind, especially for browser environments where user interaction—such as clicks, keyboard input, or mouse movements—drives much of the application behavior. However, events are not limited to user-driven interactions. In server-side environments, a much broader and more diverse range of events come into play. These include network requests, database operations, file system access, timers, and many other asynchronous activities that can occur independently of any user input. Understanding how code execution **flows** is crucial to mastering Node.js. JavaScript, especially in the Node.js environment, relies heavily on **asynchronous programming** to manage tasks efficiently without blocking the system. Let's explore what that really means by comparing three common models.

### 6.2.1 Synchronous, Single Thread

In a traditional synchronous, single-threaded execution model, every instruction in a program is carried out in a strict, **sequential order**. This means that the program handles one task at a time, and it cannot move on to the next task until the current one has been fully completed. There is no parallelism or overlapping of operations, each task waits its turn in the line of execution.

For instance, if a program needs to read two files from disk one after the other, it will begin by reading the first file. The second file read operation will not even start until the first one has completely finished. This approach becomes problematic when **any single operation is slow or involves waiting**, such as accessing the file system, performing a network request, or querying a database. During this waiting period, the entire program is effectively paused, unable to do anything else. As a result, the total time it takes to complete all tasks is simply **the sum of the durations of each task**, including the time spent waiting.

Consider the following JavaScript example using synchronous file reading:

```
let data1 = readFileSync("file1.txt"); // blocks the thread until this is done
let data2 = readFileSync("file2.txt"); // only starts after the first one completes
```

In this code, the second file is not read until the first one is entirely loaded. While this model is easy to understand and debug, especially for beginners, it does not perform well in environments where responsiveness and scalability are important. Applications that need to handle many simultaneous tasks, such as web servers or user interfaces, can become sluggish or unresponsive if built entirely on synchronous, single-threaded logic.

### 6.2.2 Synchronous, Multiple Threads

To enhance the ability of a system to perform multiple tasks concurrently, many programming environments and operating systems make use of multiple threads of execution. In this model, instead of processing every task sequentially within a single thread, **the system creates and manages multiple threads**, each capable of executing a different task. These threads can run simultaneously, leveraging the processing power of multi-core CPUs. When a program follows this approach, tasks that would otherwise block the main thread (such as computationally intensive operations or input/output activities) can be offloaded to separate threads. These threads operate independently, and they may begin **executing in parallel**, without having to wait for the completion of previous tasks. This greatly improves responsiveness and throughput, especially in applications that serve many users or perform many operations at once.

However, because these threads share the same memory space, **coordination between them becomes crucial**. Often, the main thread must eventually pause to wait for the completion of other threads, collect their results, and proceed only after re-establishing a consistent state. This process is known as **synchronization**. Synchronization is essential to ensure that shared resources are accessed in a safe and predictable way, but it also introduces complexity. Developers must be cautious of common pitfalls such as race conditions, deadlocks, and other **concurrency bugs** that are difficult to detect and reproduce.

This model is widely used in languages such as Java and C++, particularly in systems that require high performance under concurrent loads. Examples include multithreaded web servers that handle multiple client requests at once, or desktop applications that maintain a responsive user interface while performing background computations. While powerful, the multi-threaded model demands careful design and discipline to avoid the subtle and often critical issues that arise from improper thread handling.

### 6.2.3 Asynchronous Execution Model

Unlike traditional synchronous models that rely on sequential or multi-threaded execution, Node.js adopts an entirely different strategy: it is built around **asynchronous programming** using an **event-driven architecture**. This means that instead of blocking the execution flow while waiting for slow operations, like file access or network communication, to complete, the program initiates these operations and then immediately moves on to the next line of code.

In practice, when Node.js encounters a time-consuming task (reading a file from disk) it begins that operation and immediately continues executing the rest of the program. Rather than stalling the entire thread until the result is ready, Node.js registers a **callback function** (or, in modern syntax, attaches a **promise**) that will be invoked once the operation completes. When the system finishes the requested task, it executes the callback. For example, consider the following code using Node.js's asynchronous file reading capability:

```
const fs = require('fs');

fs.readFile("file.txt", (err, data) => {
  if (err) throw err;
  console.log(data.toString());
});

// Meanwhile, the app can continue doing other things
```

In this code, the `fs.readFile()` function initiates the file read, but it does not wait for it to finish before moving forward. Instead, the rest of the application remains free to perform other work, such as responding to additional requests or executing unrelated code. When the file has been fully read, the callback function is triggered, and the data is processed.

This asynchronous, non-blocking model is a cornerstone of Node.js and is the reason it can handle thousands of simultaneous I/O operations efficiently, all within a single thread. There is no need to spawn new threads or manage complex synchronization mechanisms, as the callback system is designed to manage concurrency in a lightweight and elegant way.

This model is especially effective in scenarios where applications need to remain highly responsive while performing many slow, I/O-bound tasks, such as serving dynamic web content, handling API requests, or managing large volumes of data coming from databases or network sources. While it may introduce a steeper learning curve compared to straightforward synchronous code, especially in terms of flow control and error handling, the performance benefits and scalability offered by asynchronous programming make it an ideal choice for modern server-side applications.

#### 6.2.4 Callback hell

While the callback approach worked, it often led to a situation known as **callback hell**, where multiple nested callbacks created code that was difficult to read, maintain, and debug. This pattern quickly became problematic as the complexity of logic increased. Consider the following example:

```
getUser(userId, function(err, user) {
  if (err) {
    console.error('Error fetching user');
  } else {
    getOrders(user, function(err, orders) {
      if (err) {
        console.error('Error fetching orders');
      } else {
        getOrderDetails(orders[0], function(err, details) {
          if (err) {
            console.error('Error fetching order details');
          } else {
            console.log('Order details:', details);
          }
        });
      }
    });
  }
});
```

Each function is nested within the previous one, creating deeply indented and hard-to-follow code. This is difficult to manage, especially when the nesting gets deeper or we need to add error handling at multiple levels.

#### 6.2.5 Function Chaining

Function chaining addresses the deep nesting issue by allowing asynchronous operations to be linked in a linear, readable sequence. This is made possible with **Promises**, a core feature in modern JavaScript that represents the eventual completion or failure of an asynchronous task. Promises support methods like `.then()` and `.catch()`, which are specifically designed to be chained.

When a function returns a Promise, you can call `.then()` on it to specify what should happen once the Promise is fulfilled. The `.then()` method itself returns a new Promise, which allows

you to chain another `.then()`, creating a smooth, left-to-right flow of asynchronous logic. If any step in the chain fails, the error can be caught and handled with a `.catch()` block at the end of the chain, without disrupting the rest of the code. For example, instead of nesting callbacks like:

```
getUser(userId, function(err, user) {
  getOrders(user, function(err, orders) {
    getOrderDetails(orders[0], function(err, details) {
      console.log(details);
    });
  });
});
```

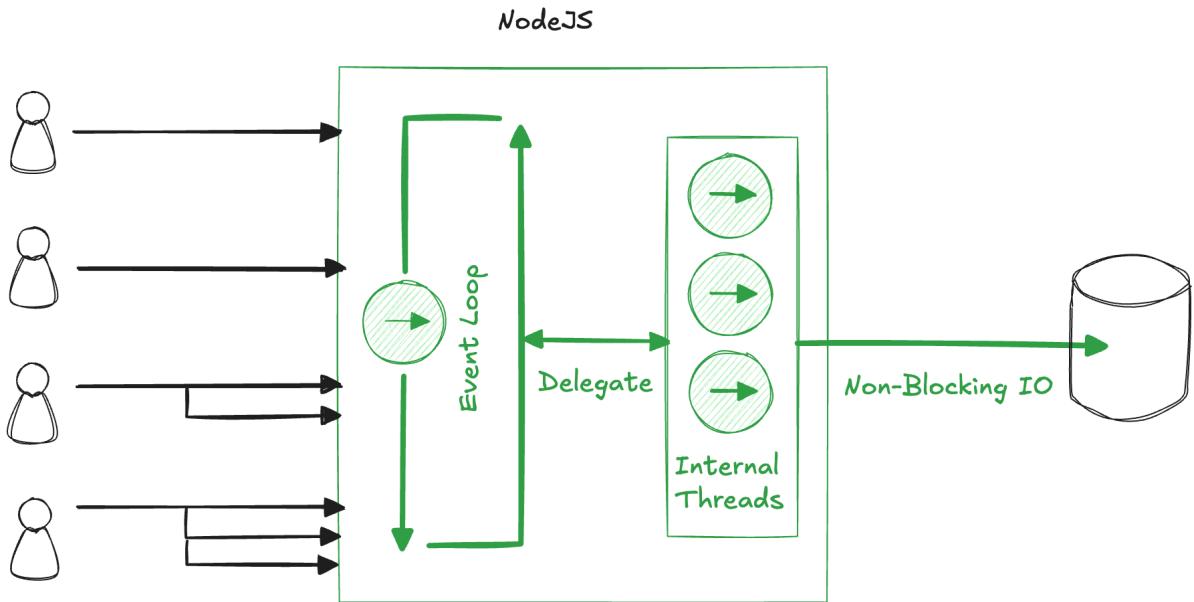
We can refactor the same logic using function chaining and Promises:

```
getUser(userId)
  .then(user => getOrders(user))
  .then(orders => getOrderDetails(orders[0]))
  .then(details => console.log(details))
  .catch(error => console.error('Something went wrong:', error));
```

This structure is much easier to follow and scales better as more steps are added. Each function in the chain returns a Promise, and the next `.then()` waits for that Promise to resolve before continuing. Errors from any point in the chain can be caught and handled in one centralized place using `.catch()`. Function chaining, enabled by Promises, is a major improvement in JavaScript's approach to asynchronous programming, offering a cleaner, more manageable alternative to nested callbacks.

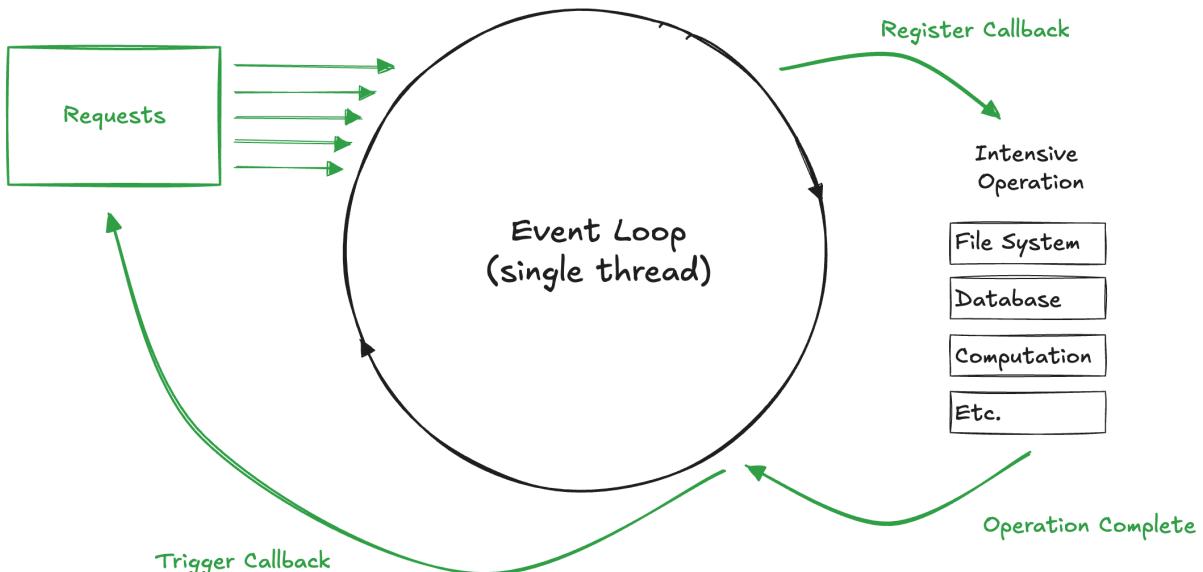
### 6.2.6 Event Loop

To fully understand the asynchronous model adopted by Node.js, we need to explore a key concept at the heart of its runtime: the **Event Loop**. JavaScript is a single-threaded language (it can execute only one instruction at a time) but thanks to the Event Loop, it can handle asynchronous events in a highly efficient, non-blocking way. When a JavaScript program runs in Node.js, there is a single main thread dedicated to executing JavaScript code. This thread is responsible for handling all incoming events (from HTTP requests and timers to file system operations and more). As the program runs, time-consuming tasks (like file access or database queries) are **delegated to the system's underlying non-blocking I/O mechanisms**, such as the pool of internal threads.



These tasks do not block the main thread. Instead, once the task completes, the system emits an event which gets placed in a **task queue**. The Event Loop constantly monitors this queue, and when the main thread is idle, it picks up the next task and executes its associated callback:

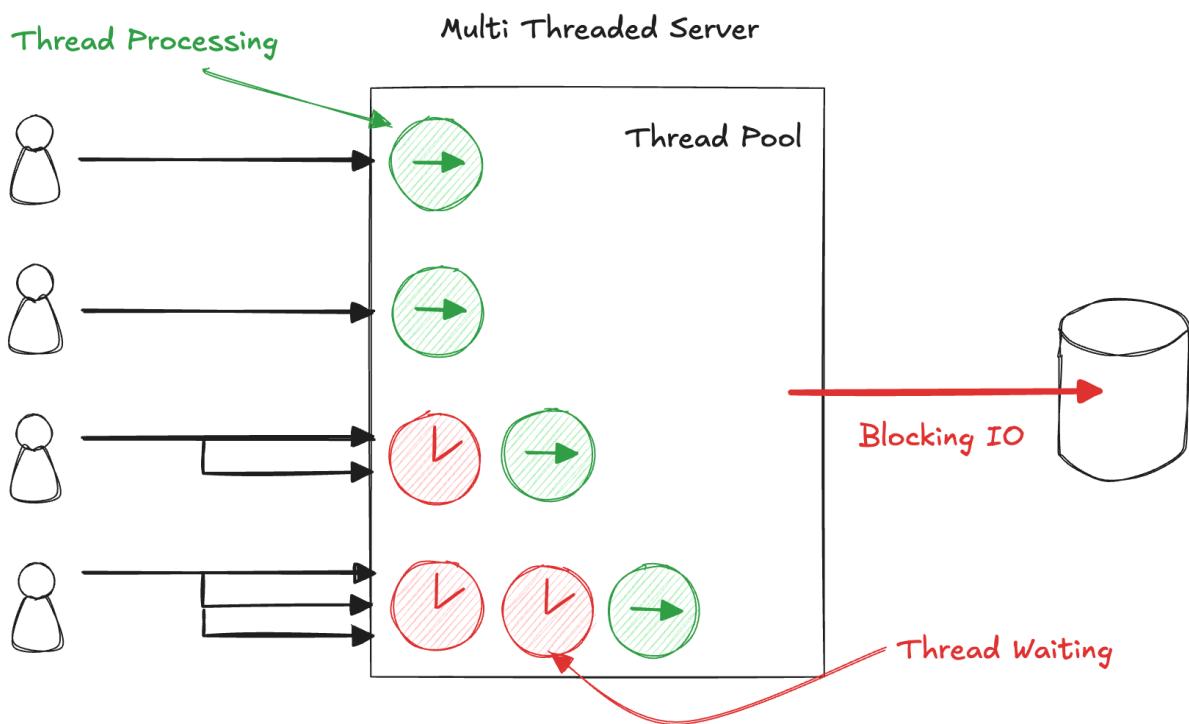
```
while(queue.waitForMessage()){
    queue.processNextMessage();
}
```



Here, multiple requests arrive and are quickly handed off by the single-threaded Event Loop

to the non-blocking I/O system. The actual file or database access happens in the background, managed by internal threads. Once the data is ready, the callback is queued back to the main thread, which processes it when it's ready. This allows the Node.js server to continue handling other requests without being held up by slow I/O operations.

This approach contrasts sharply with the traditional **multi-threaded server** model:



In that model, every incoming request is handled by a separate thread from a thread pool. If a request involves a blocking I/O operation (like reading from disk), the thread must wait: it is effectively paused until the operation completes. This leads to a lot of "thread waiting" time, represented by the red clock icons in the figure, which consumes system resources and limits scalability.

The Node.js model minimizes the number of threads that are left idle. Since the Event Loop does not block while waiting for I/O operations to complete, it can keep the single main thread available for processing many more incoming events, making Node.js particularly well-suited for **I/O-heavy** applications like web servers, chat apps, or real-time data services.

This efficiency comes at the cost of complexity: developers must write code that **never blocks the main thread**, and long computations must be offloaded to worker threads or broken into smaller chunks. But when used appropriately, the Event Loop model offers significant performance and scalability advantages without the pitfalls of thread synchronization, race conditions, or deadlocks typical in multi-threaded environments.

### 6.3 The HTTP Module

In Node.js, the `http` module allows us to create a fully functioning **HTTP server**. Unlike languages like PHP, Java, or C#, which are typically executed inside a separate web server such as Apache or IIS, Node.js can **act as a standalone server itself**. This means it can handle HTTP requests directly without the need for an external server environment. For example, the following code (`01.HelloWorldServer.js`) demonstrates a minimal HTTP server in Node (see `01.HelloWorldServer.js` code):

```
var http = require('http');

http.createServer(function (request, response) {
  response.writeHead(200, {'Content-Type': 'text/plain'});
  response.end('Hello World\n');
}).listen(8124, "127.0.0.1");

console.log('Server running at http://127.0.0.1:8124/');
```

Here, we import the `http` module and assign its functionalities to the `http` object. The `createServer()` method takes a callback function that is triggered every time the server receives a new request. This function accepts two parameters: `request`, which contains details such as the URL and headers, and `response`, which is used to send data back to the client. Finally, the server listens on port 8124 of the localhost.

This demonstrates how Node.js handles events and callbacks. Whenever a request is received (an event), the callback function we've passed is executed. In just six lines, we've created a basic but complete HTTP server. Of course, real-world web servers often need to do more. They typically examine the request method (e.g., GET, POST) to determine the client's intended action, and they parse the requested URL to identify which resource that action applies to.

Node.js can also act as a client to send HTTP requests. The following example (see `02.HelloWorldClient.js` code) shows how to make a simple GET request to download an image:

```
var http = require('http');

var request = http.request({
  hostname: "eloquentjavascript.net",
  path: "/img/cover.jpg",
  method: "GET",
  headers: {Accept: "text/html"}
},
```

```
function(response) {  
    console.log("Server responded with status code: ", response.statusCode);  
}  
  
request.end();
```

In this example, we use `http.request()` to configure the request: we specify the target server, the path, the HTTP method, and the headers. When a response is received, a callback is triggered to handle it. In this case, logging the status code to the console.

To perform secure requests over HTTPS, Node.js provides a separate `https` module with a similar interface, including its own `request()` method.

### 6.3.1 Streams

In many real-world applications, data is not always available all at once. For example, when processing files, receiving input over a network, or handling real-time data feeds, the total size of the data might be large or even unknown at the start. Loading everything into memory at once can be inefficient or even infeasible. To address this, Node.js provides an elegant and efficient solution through **streams**, an abstraction for handling continuous, chunked data.

A **stream** is essentially a sequence of data made available over time. Instead of waiting for the entire dataset to be received or loaded before processing, streams allow you to handle the data in **smaller pieces** (called **chunks**) as it arrives. This streaming approach becomes especially important when working with **large datasets** or **external sources** like file systems, network requests, or `stdin/stdout`, where the total size or duration of the data flow is not known in advance.

We already encountered streams in earlier examples: the `response` object returned by the HTTP server and the `request` object used in `http.request()` are both instances of streams. These are **writable** and **readable** streams, respectively. Node.js stream objects provide a consistent interface for interacting with different types of streaming data. There are four fundamental types of streams in Node.js:

- **Readable** streams: represent sources you can read data from (e.g., file input, HTTP requests).
- **Writable** streams: represent destinations you can write data to (e.g., file output, HTTP responses).
- **Duplex** streams: are both readable and writable (e.g., network sockets).
- **Transform** streams: are duplex streams that can modify or transform the data as it is read and written (e.g., compression or encryption streams).

Each stream is an instance of an **EventEmitter** and emits various events such as data, end, error, and close. For example, to manually transfer data from a readable stream to a writable one, we can do:

```
readable.on('data', (chunk) => {
  writable.write(chunk);
});

readable.on('end', () => {
  writable.end();
});
```

### 6.3.2 Pipe

While this pattern works, Node.js provides a simpler and more elegant approach through the .pipe() method. **Piping** automatically manages the flow of data from a readable stream into a writable one, handling backpressure and cleanup internally. For instance:

```
readableSrc.pipe(writableDest);
```

This line effectively sets up a connection where readableSrc continuously pushes data into writableDest as it becomes available. The .pipe() method greatly simplifies code when dealing with file transfers, proxies, or any situation where data flows from one source to a destination.

Streams provide the building blocks for efficient, modular, and scalable data processing in Node.js applications. They are the backbone of many performance-sensitive operations, allowing developers to write programs that work with data incrementally and without unnecessary memory consumption.

### 6.3.3 Serving a big file

To fully appreciate the power of streams, let's walk through a practical example that demonstrates the dramatic difference they make in terms of **memory consumption** and **efficiency**. Start by generating a large file (see \03.Stream\create.js code):

```
const fs = require('fs');
const file = fs.createWriteStream('./big.file');
for(let i=0; i<= 1e7; i++) {
```

```

    file.write('Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do
    ↵ eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim
    ↵ veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea
    ↵ commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit
    ↵ esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat
    ↵ cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est
    ↵ laborum.\n');
}
file.end();

```

This code produces a file named "big.file", approximately 400 MB in size. We can experiment serving this large file using a method that **does not use streams** (see \03.Stream\server\_1.js)

```

const fs = require('fs');

const server = require('http').createServer();

server.on('request', (req, res) => {
  fs.readFile('./big.file', (err, data) => {
    if (err) throw err;
    res.end(data);
  });
});

server.listen(8000);

```

A request handler reads the entire big.file into memory using fs.readFile(). Only after the full file is loaded is it written to the response. This approach might seem straightforward, but it is **highly inefficient** for large files. When tested, the server started with about 8.7 MB of memory usage. Upon receiving a single request, it jumped to 434.8 MB, meaning the entire file content had to be loaded into memory at once before being sent out. This kind of **memory spike** may be acceptable for small files, but it becomes **untenable for very large files** or high-traffic environments. In fact, trying the same approach with a 2GB file will likely cause the server to run out of memory or become unresponsive. In contrast, we can use a method based on streams (see \03.Stream\server\_2.js):

```

const fs = require('fs');

const server = require('http').createServer();

```

```
server.on('request', (req, res) => {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  const stream = fs.createReadStream('./big.file');
  stream.pipe(res);
});

server.listen(8000);
```

It uses the `fs.createReadStream()` method to serve the same file. Since the HTTP response object is a **writable stream**, we can directly **pipe** the file stream into the response. This setup ensures that **only small chunks of the file** are loaded into memory at any given time. The system reads from the file and writes to the response stream **incrementally**, keeping memory usage low and stable. In this case, even when handling the same 400 MB file, memory usage only rose by about 25 MB, a massive improvement in efficiency.

Now imagine running both versions of the server with a **2GB file**. With `server_1`, the process might crash or become unresponsive, as it tries to load the entire 2GB into memory. But with `server.js` and the stream-based approach, the memory footprint remains minimal, and the server continues to function smoothly, even under load.

This example clearly illustrates the importance of using streams for I/O-heavy operations. Not only do they prevent unnecessary memory consumption, but they also make our application more scalable and responsive under pressure.

## 6.4 Examples

To consolidate the concepts learned so far (including asynchronous programming, streams, and event-driven architecture) we now explore two practical and illustrative Node.js applications: a **File Server** and a **Chat Server**.

These examples serve different purposes but are built on the same foundational principles of the Node.js runtime: efficient handling of concurrent I/O using **non-blocking code** and **event-based logic**. They provide a natural transition from working with the file system and HTTP to managing lower-level TCP socket communication.

- The **File Server** combines HTTP and the file system into a simple interface. It allows remote clients to read, write, and delete files over HTTP using standard methods like GET, POST, and DELETE. By leveraging **streams**, it avoids reading entire files into memory, ensuring efficiency even with very large files.

- The **Chat Server** drops below the HTTP layer and works directly with TCP sockets, enabling real-time communication between multiple clients. It shows how to use Node's **net module** to handle raw socket connections, broadcast messages, and manage disconnections and errors gracefully.

Both projects demonstrate how Node.js scales well without requiring threads, and how its **single-threaded event loop** can effectively manage multiple concurrent connections with minimal resource usage.

These examples are also excellent starting points for building more advanced applications — such as APIs, cloud storage interfaces, multiplayer games, or messaging systems — all while staying true to the Node.js philosophy of lightweight, asynchronous design.

#### 6.4.1 A file Server

We create a simple **file server** using Node.js. This server combines everything we've learned (working with the file system, handling HTTP requests, and streaming data efficiently) to allow remote clients to **read, write, and delete files** over the network (see 04.FileServer.js code).

```
const http = require("http");
const fs = require("fs");
const mime = require("mime");

// An object called methods to store the functions that handle
// the various HTTP methods
const methods = Object.create(null);

http.createServer(function(request, response) {

    // A function to finish the request. It takes an HTTP
    // status code, a body, and optionally a content type as arguments.
    // If the value passed as the body is a readable stream,
    // it will have a pipe method, which is used to forward a readable
    // stream to a writable stream. If not, it is assumed to be either
    // null (no body) or a string and is passed directly
    // to the response's end method.
    function respond(code, body, type){
        if (!type) type = "text/plain";
        response.writeHead(code, {"Content-Type":type});
        if (body && body.pipe) body.pipe(response);
        else response.end(body);
    }
})
```

```
// The respond function is passed to the functions that
// handle the various methods and acts as a callback to finish
// the request.
if(request.method in methods) methods[request.method](urlToPath(request.url),
  ↵ respond, request);
// Returns 405 error responses, which is the
// code used to indicate that a given method isn't handled
// by the server
else respond(405, "Method " + request.method + " not allowed.");

}).listen(8000);

// A function to get a path from the URL in the request,
// it uses Node's built-in "url" module to parse the URL.
// It takes its pathname, which will be something like /file.txt,
// decodes that to get rid of the %20-style escape codes,
// and prefixes a single dot to produce a path relative
// to the current directory.
function urlToPath(url) {
  var path = require("url").parse(url).pathname;
  return "." + decodeURIComponent(path);
};

// A method to manage GET request. It returns a list of files
// when reading a directory, and the file's content when reading
// a regular file.
// One tricky question is what kind of Content-Type header we
// should add? Since these files could be anything, our
// server can't simply return the same type for all of them.
// NPM can help with that: the "mime" package knows the correct
// type for a huge amount of file extensions.
methods.GET = function(path, respond){
  // Because it has to touch the disk, and thus might take a while, fs.stat
  // is asynchronous. When the file does not exist, fs.stat will pass
  // an error code property of "ENOENT" to its callback.
  fs.stat(path, function(error, stats){
    // Codes starting with 4 (such as 404) refers to bad requests
    if( error && error.code=="ENOENT") respond(404, "File not found");
    // Codes starting with 5 (such as 500) refers to a server problem
    else if (error) respond(500, error.toString());
    else if (stats.isDirectory())
      fs.readdir(path, function(error, files){
```

```

                if (error) respond(500, error.toString())
                else respond(200, files.join("\n"));
            });
        else
            respond(200, fs.createReadStream(path), mime.getType(path));
    });
};

methods.POST = function(path, respond, request){
    fs.stat(path, function(error, stats) {
        if( error && error.code=="ENOENT" ) {
            const outStream = fs.createWriteStream(path);
            outStream.on("error", function(error) { respond(500,
                error.toString()); });
            outStream.on("finish", function() { respond(204); });
            // We use pipe to move data from a readable stream to a writable
            // one,
            // in this case from the request to the file
            request.pipe(outStream);
        }
        // Codes starting with 5 (such as 500) refers to a server problem
        else if (error) respond(500, error.toString());
        else respond(400, "The file already exists");
    });
};

// The code to handle a delete requests
methods.DELETE = function(path, respond) {
    fs.stat(path, function(error, stats) {
        // You may be wondering why trying to delete a nonexistent file
        // returns a 204 ("no content") status, rather than an error.
        // When the file that is being deleted is not there, you could say
        // that the request's objective is already fulfilled.
        // The HTTP standard encourages people to make requests idempotent.
        if (error && error.code == "ENOENT") respond(204);
        else if (error) respond(500, error.toString());
        else if (stats.isDirectory()) fs.rmdir(path,
            respondErrorOrNothing(respond));
        else fs.unlink(path, respondErrorOrNothing(respond));
    });
};

function respondErrorOrNothing(respond) {

```

```
    return function(error) {
      if (error) respond(500, error.toString());
      else respond(204);
    };
}

// The code for the PUT request
methods.PUT = function(path, respond, request) {
  const outStream = fs.createWriteStream(path);
  outStream.on("error", function(error) { respond(500,
  ~ error.toString()); });
  outStream.on("finish", function() { respond(204); });
  // We use pipe to move data from a readable stream to a writable one,
  // in this case from the request to the file
  request.pipe(outStream);
};
```

The server acts as a bridge between HTTP and the local file system. It listens for HTTP requests on port 8000 and interprets the **URL path** of each request as a path relative to the server's working directory. This ensures that the server doesn't expose the entire file system, just a local subset of it. Incoming HTTP requests are routed to corresponding handlers for GET, POST, DELETE, and PUT. These handlers interact with the file system asynchronously and use streams wherever possible for efficient data handling. When dealing with files as HTTP resources, different HTTP methods correspond naturally to file operations:

- **GET** is used to **read** a file or list the contents of a directory.
- **POST** and **PUT** are used to **create** or **replace** files.
- **DELETE** is used to **remove** files or directories.

This mapping is a common pattern and it provides a simple interface for interacting with files remotely, for example, to upload data, distribute assets, or manage shared documents.

One of the most important design choices in this server is the use of **streams** when reading from or writing to the file system. For example:

- When a file is read via a GET request, the server uses `fs.createReadStream()` to **stream the file's content** directly into the response object.
- When writing data with POST or PUT, the incoming request object (which is a readable stream) is **piped** into `fs.createWriteStream()`.

This means that large files are **never fully buffered into memory** at any point. Instead, they are transferred chunk by chunk between disk and network, keeping memory usage low and the

system responsive.

We can test this server using Postman:

- **GET** a file: `http://localhost:8000/notes.txt`
- **POST** to create a new file: use `POST http://localhost:8000/newfile.txt` with body content
- **PUT** to overwrite a file
- **DELETE** to remove a file or folder

This project is a great example of how **Node.js's stream-based architecture** allows us to build scalable, efficient servers with minimal resource usage. Would you like to add a section comparing this approach to a synchronous file server or show usage logs from Postman?

#### 6.4.2 A Chat Server

So far, we've looked at how Node.js handles HTTP, file systems, and streams. But Node.js is not limited to HTTP: it also allows for **raw TCP communication** using the `net` module. This capability is especially useful for building custom protocols or lightweight communication systems, such as a **command-line chat server** (see `05.ChatServer.js` code):

```
// Include the net library, that contains all the TPC stuff
const net = require('net');

// Create a TCP server
const chatServer = net.createServer();

// A list of client connections
const clientList = [];

// Add a event listener using the on method whenever a "connection event happens, the
// ↴ listener
// will be called. The event provides a reference to the TPC socket for the new client
chatServer.on('connection', function(client) {
    // decorate the client with an additional property, the name,
    // using the existing properties "remoteAddress" and "remotePort"
    client.name = client.remoteAddress + ':' + client.remotePort;

    // with "write" we can send messages to the new client
    client.write('Hi ' + client.name + '!\\n');

    // add the new client to the list
});
```

```
clientList.push(client);

// another event listener, it's in the scope of the connection callback method, we
↳ need to access the client.
// The event is called "data", it is fired each time client sends some data to the
↳ server.
client.on('data', function(data) { broadcast(data, client) });
});

// A method to send messages to all client, filtering the sender
function broadcast(message, client) {
    for(let i=0; i<clientList.length; i+=1) {
        if(client !== clientList[i]) { clientList[i].write(client.name + " says "
↳ + message); }
    }
}

// The port on which Node is listen to
chatServer.listen(80)
```

The code implements a simple chat server using TCP sockets. WE can test it using a terminal program like telnet or nc (netcat):

```
telnet localhost 9000

nc localhost 9000
```

The server listens for incoming TCP connections. When a client connects, the server assigns them a name (based on their IP and port), sends a greeting, and then starts listening for incoming data using the data event. Whenever a message arrives, the server broadcasts it to all other connected clients. This design demonstrates two key event listeners:

1. A listener for the connection event: triggered whenever a new client connects.
2. A listener for the data event on each client: triggered whenever a client sends a message.

Messages are sent using the .write() method, since TCP sockets are **duplex streams**, meaning they can both read and write. The broadcast function iterates over all clients and sends each message to everyone except the original sender.

However, this initial implementation has a **fatal flaw**: if a client disconnects, the server still tries to write to their socket. This causes errors and can crash the server. We can implement an improved version (see 06.RobustChatServer.js code):

```
const net = require('net');
const chatServer = net.createServer();
const clientList = [];

chatServer.on('connection', function(client) {
    client.name = client.remoteAddress + ':' + client.remotePort
    client.write('Hi ' + client.name + '!\\n');
    console.log(client.name + ' joined');
    clientList.push(client);
    client.on('data', function(data) { broadcast(data, client) });

    // Add a new callback for the "end" event: it is fired
    // when a client disconnect. Now when the next client uses the
    // broadcast call, the disconnected client no longer be in the list
    client.on('end', function() {
        console.log(client.name + ' quit');
        clientList.splice(clientList.indexOf(client), 1);
    });

    // adding a callback method for the "error" event, in order to be
    // sure that any error that occur to clients are logged and the server is no
    // aborted with an exception
    client.on('error', function(e) { console.log(e); });
});

function broadcast(message, client) {
    const cleanup = [];
    for(let i=0; i<clientList.length; i+=1) {
        if(client !== clientList[i]) {
            // adding a check for the write status of the socket during
            // the broadcast call, we can make sure that any sockets that
            // are not available to be written don't cause exception
            if(clientList[i].writable) { clientList[i].write(client.name +
                " says " + message); }
            else {
                // make sure that any non writable socket
                // will be closed and removed by the list.
                // note that we are not removing the client
                // from the list while we are looping through it
                // in order to avoid side effects
                cleanup.push(clientList[i]);
                clientList[i].destroy();
            }
        }
    }
}
```

```
        }
    }
//Remove dead Nodes out of write loop to avoid trashing loop index
for(let i=0; i<cleanup.length; i+=1) {
    clientList.splice(clientList.indexOf(cleanup[i]), 1)
}
}

chatServer.listen(9000)
```

The improved version adds several important features to make the chat server more robust:

- Listening to the end event to detect when a client disconnects and **removing** them from the list.
- Adding a listener for the error event to **gracefully handle socket failures**.
- Checking .writable before writing to a socket to avoid attempting to send data to a dead connection.
- Using a cleanup list to defer removal of broken sockets **after** the main loop, preventing issues with array mutation during iteration.

This kind of defensive programming is essential when working with network code, where failures and disconnects are common.

## 6.5 Express

As applications grow in complexity, managing raw HTTP servers with Node.js can become verbose and error-prone. To address this, the Node.js ecosystem offers **Express**, a minimal and flexible **web application framework** that simplifies the creation of robust web servers and APIs.

Express builds on top of Node's native HTTP module, providing a **more concise and readable syntax**, powerful **routing capabilities**, and an extensible **middleware architecture**. With Express, developers can quickly set up HTTP endpoints, process different kinds of requests, and manage complex flows using a modular and scalable structure. To use Express, it must first be installed via npm:

```
npm install express --save
```

Once installed, a basic "Hello World" web server looks like this:

```
const express = require('express');
const app = express();

app.get('/', function (req, res) {
  res.send('Hello World');
});

const server = app.listen(8081, function () {
  const host = server.address().address;
  const port = server.address().port;
  console.log("Example app listening at http://%s:%s", host, port);
});
```

In this code:

- The `express()` function creates an `app` object that acts as the main server.
- The `app.get()` method sets up a **route** for GET requests on the root URL (/).
- The callback function takes `req` (request) and `res` (response) objects, familiar abstractions used throughout Express for handling HTTP transactions.
- The `res.send()` method sends a simple text response to the client.
- `app.listen()` binds the server to a port and starts listening for requests.

### 6.5.1 Request and Response Objects

Express extends Node's basic `req` and `res` objects with additional utility features. For example:

- `req.query`, `req.params`, and `req.body` to extract data from URLs, routes, and payloads.
- `res.send()`, `res.json()`, and `res.status()` to easily craft responses.

This uniform interface keeps our code clean and expressive, even when dealing with multiple content types or data formats. Express.js is one of the most widely-used frameworks in the Node.js ecosystem. It abstracts the complexity of raw HTTP handling while remaining close to the metal, giving developers fine control over how web servers behave. With powerful routing, intuitive middleware, and a large ecosystem of compatible packages, Express is the go-to choice for building APIs, web apps, and microservices in Node.js.

### 6.5.2 Routing

Routing in Express is **declarative and method-based**: we use functions like `app.get()`, `app.post()`, `app.put()`, and `app.delete()` to define the server's behavior for different HTTP

methods and URLs. This structure creates a **routing table** that determines how requests are handled based on the method and path. Example:

```
app.get('/users', function(req, res) {
  res.send('GET request to /users');
});

app.post('/users', function(req, res) {
  res.send('POST request to /users');
});
```

Each route is matched by method and path, and invokes a callback to respond accordingly.

### 6.5.3 Middleware

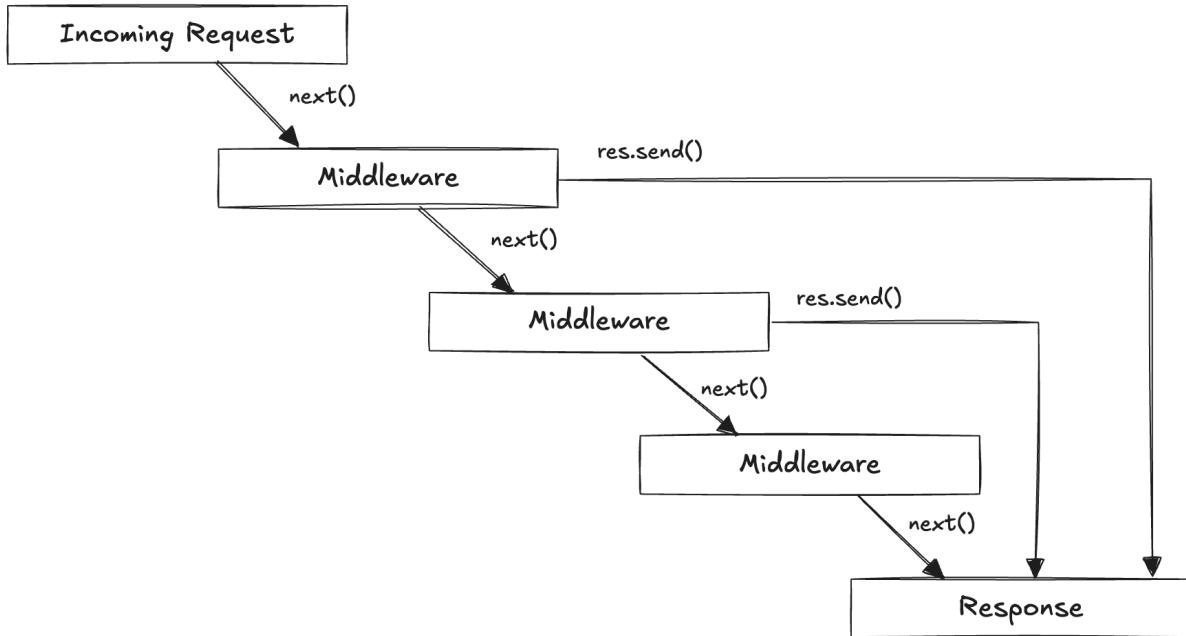
In Express, **middleware** is one of the core concepts that makes the framework flexible, modular, and developer-friendly. Middleware functions are **pieces of code that execute during the life-cycle of an HTTP request**, sitting in between the moment a request arrives and the moment it's handled by a final route like .get() or .post(). A middleware function has access to the request and response objects, and it can either:

- Modify them (e.g., parse a request body),
- Perform actions (e.g., log details, check authentication),
- Or end the request-response cycle by sending a response,
- Or call next() to **pass control to the next middleware** in the chain.

Middleware allows us to **reuse code** for common tasks across multiple routes, keeping our route handlers **clean, focused, and easier to maintain**. They are commonly used for:

- **Logging** every incoming request
- **Authentication and authorization** checks
- **Parsing incoming data**, such as JSON or form submissions
- **Serving static files**
- **Error handling** in a centralized way

Middleware improves both clarity and maintainability by separating responsibilities into modular components:



A request enters the Express app, passes through one or more middleware functions (each optionally transforming or reacting to the request), and eventually reaches the final route handler that sends a response. Each middleware has the opportunity to process or block the request, which gives developers precise control over how HTTP requests are handled.

Middleware is mounted using `app.use()` or directly within route definitions. For example:

```
app.use(express.json()); // parses JSON payloads automatically
```

As an example, we can use the body-parser module to automatically convert the raw request body into a usable JavaScript object:

```
const bodyParser = require('body-parser');

app.post('/send', bodyParser.json(), function(req, res) {
  // req.body is now available thanks to bodyParser
});
```

In this example:

- When a POST request is made to `/send`, the `bodyParser.json()` middleware intercepts the request.
- It reads the raw body, parses the JSON content, and attaches it to `req.body`.

- Only then does the actual route handler function execute, with full access to the parsed data.

This middleware approach avoids duplicating logic for every route and allows the body-parsing behavior to be reused across the entire application, or selectively added to specific routes.

#### 6.5.4 A Simple Twitter Clone

To demonstrate how Express can be used to build real-world web applications, we'll now create a minimal Twitter-like service: a simple app that lets users **post tweets** and **view all submitted tweets**. This example showcases the **core features of Express** — routing, middleware, request parsing, and response formatting — all in under 30 lines of code (see 07.Twitter.js code):

```
// Loading Express, in order to managing HTTP behind the scenes.
const express = require('express');
const bodyParser = require('body-parser');

// Create a server
const app = express();

// Create the array of received tweets
const tweets = [];

// Instead of attaching listener, we can provide methods
// matching HTTP verbs on specific URL requests
app.get('/', function(req, res) { res.send('Welcome to Node Twitter'); })

// A POST route for posting tweets, with a middleware (bodyParser()) to stream
// the post data from the client and turn it in a JavaScript object
// It works only with POST encoded in json
app.post('/send', bodyParser.json(), function(req, res) {
    // express.bodyParser() adds a property to req called body, it
    // contains an object representing the post data
    if (req.body && req.body.tweet) {
        tweets.push(req.body.tweet);
        res.send( { status:"ok", message:"Tweet received" } );
    }
    else { res.send( {status:"nok", message:"No tweet received" } ); }
})

// a GET route to access all the tweets
app.get('/tweets', function(req,res) { res.send(tweets) });
```

```
app.listen(8000);
```

This is the **core functionality**: a route that handles POST requests on /send. It uses bodyParser.json() to automatically parse the request body. If the request contains a tweet field, it adds it to the tweets array and sends a JSON response confirming receipt; and a GET route returning the current array of tweets. Since the data is stored in memory, every new request will reflect the latest state.

To use it:

1. Start the server by running the script.
2. Open a browser and navigate to <http://localhost:8000/tweets> to view the list of tweets.
3. Use Postman to send new tweets with a POST request to <http://localhost:8000/send> and a body like:

```
{  
  "tweet": "ciao come stai?"  
}
```

## 6.6 Why Node.js and IoT?

As we move deeper into the era of the **Internet of Things (IoT)** (where everyday objects become smart and connected) the way we design and build software systems must evolve. IoT applications are characterized by a massive number of devices (sensors, actuators, wearables, gateways) that generate continuous data streams and require low-latency, scalable control systems. In this context, **Node emerges as a highly effective platform** for building IoT backends and services.

### 6.6.1 Scalability and Performance

Node.js is powered by the **Google V8 engine**, which compiles JavaScript into fast machine code. Combined with its **non-blocking, event-driven architecture**, Node.js can handle thousands of concurrent connections efficiently. This is especially important in IoT environments where devices frequently send small payloads of data, often simultaneously, and the backend must remain responsive and scalable.

In contrast to traditional server architectures that spawn new threads per request (consuming memory and CPU), Node.js uses a **single-threaded event loop** and asynchronous callbacks, making it lightweight and ideal for **data-intensive, real-time applications**.

### 6.6.2 Effective Communication and API Design

IoT systems rely heavily on **APIs** for communication between devices, servers, and dashboards. Node excels at building APIs quickly and efficiently thanks to its rich ecosystem of frameworks like **Express**, **Hapi**, and **Restify**. This model is well-suited for IoT, where devices need to send and retrieve data using lightweight protocols over HTTP. A real-world endorsement comes from companies like **LinkedIn**, which rebuilt its mobile backend using Node.js to take advantage of its **speed and concurrency model**.

### 6.6.3 Streaming and Real-Time Data Processing

Many IoT devices produce continuous data streams, from temperature sensors to motion detectors or GPS modules. Node handles **streaming data natively**, using its built-in stream interface. This makes it possible to read, process, transform, and forward data from devices **without buffering or storing it in memory**, enabling fast and efficient **data pipelines**. Node also integrates easily with WebSockets or brokers for **real-time bidirectional communication**, which is crucial for use cases such as remote control, live monitoring, and actuation.

### 6.6.4 Resource Efficiency

IoT environments often involve **constrained devices and edge computing**, small gateways or controllers with limited CPU, memory, or power. Node has a **very low resource footprint** and can run on minimal setups, including single-board computers like the **Raspberry Pi**. This makes it ideal not only for cloud-based processing but also for on-device scripting and edge analytics.

## 6.7 Hands-on Activity

- Improve the file server to create folders. Add support for a method PUT, which should create a directory by calling `fs.mkdir`
- Improve the Twitter application to provide a support for hashtags. The term in the URL path of a GET is the hashtag, the server must provide just the posts containing that word

## 7 REST API

In modern distributed systems, especially those powering the web and the Internet of Things (IoT), effective communication between components is essential. One of the most prominent paradigms for building such systems is the **Representational State Transfer** (REST) architecture. REST is not a protocol but a set of architectural principles that guide the design of scalable, simple, and robust systems based on web standards, particularly HTTP.

REST was formalized by Roy Fielding in his doctoral dissertation and has since become the foundation of most web services. It emphasizes the use of **resources** (identifiable pieces of data) and defines standard operations over these resources using HTTP methods like GET, POST, PUT, and DELETE. Unlike older models such as **Remote Procedure Call (RPC)**, where communication mimics traditional function calls, REST abstracts these interactions in a way that aligns with the design of the World Wide Web itself.

At the heart of REST lies the idea of leveraging a **uniform interface**. This uniformity ensures that clients and servers can interact without prior knowledge beyond the shared understanding of resource URIs and HTTP semantics. RESTful systems follow constraints such as statelessness, cacheability, and a layered system design, which contribute to their performance and scalability.

Each interaction in a RESTful API revolves around the representation of a resource. A resource is an abstract concept — a user, a sensor reading, a transaction — and its representation is typically a document encoded in a standard format like JSON. These representations are self-descriptive, meaning they include metadata that allows clients to understand how to process them without relying on out-of-band information.

A RESTful approach becomes even more powerful when coupled with **hypermedia**, a principle known as HATEOAS (Hypermedia as the Engine of Application State). This allows clients to dynamically discover actions they can take by following links embedded in the responses, much like how users navigate the web.

Today, REST APIs form the backbone of web services provided by companies like Twitter, PayPal, and eBay. They also underpin the emerging **Web of Things (WoT)**, enabling physical devices to be integrated into the web using the same architectural principles. However, REST alone is not always sufficient, especially in real-time scenarios. For these cases, extensions such as WebHooks, long polling, and WebSockets are used to push updates from servers to clients, enabling responsive, event-driven applications.

## 7.1 Remote Procedure Call

Remote Procedure Call, or RPC, is a communication pattern used in distributed systems that enables a client to execute code on a remote server as if it were invoking a local function. The idea is to hide the complexity of the network by encapsulating the function invocation into a message, sending it to a remote endpoint, and then unpacking the result once the server responds. Unlike REST, which is built around resources and HTTP semantics, RPC is centered around procedures and parameter passing.

In a typical RPC interaction, the client and server agree on a set of callable procedures. When the client calls one of these procedures, a message is constructed that includes the procedure name and its arguments. This message is sent over the network to the server. Upon receiving it, the server decodes the request, executes the corresponding function, and returns the result back to the client. From the client's point of view, it is as if a local function has returned a value, even though the execution happened elsewhere.

### 7.1.1 RPC over HTTP: server side

While traditional RPC frameworks define their own transport and message formats, modern web technologies allow RPC to be implemented using HTTP and JSON. This approach is less efficient than binary protocols like gRPC but is very accessible and easy to prototype, especially in JavaScript environments such as Node.js.

Consider a simple use case: a client wants to add two numbers using a function hosted on a remote server. The client sends a message that specifies a method name ("add") and a list of parameters (two numbers). The server reads this request, invokes the corresponding function, and returns the result. To implement this server, we use the Express framework. The server defines a single endpoint, /rpc, which handles all procedure calls. Inside this handler, the server checks the method name provided in the request body and dispatches the call to the appropriate function (see 01.rpc/server.js code):

```
const express = require('express');
const app = express();

app.use(express.json());

// Example RPC method
function add(a, b) {
  return a + b;
}
```

```
// RPC endpoint
app.post('/rpc', (req, res) => {
  const { method, params } = req.body;

  // Validate method and parameters
  if (method === 'add' && Array.isArray(params) && params.length === 2) {
    const result = add(params[0], params[1]);
    res.json({ result });
  }

  // Handle other methods or invalid input
  else {
    res.status(400).json({ error: 'Invalid method or parameters' });
  }
});

// Start the server
app.listen(3000, () => { console.log('RPC server running at
  ↵ http://localhost:3000');});
```

In this code, the function `add()` is a local JavaScript function that simply returns the sum of two numbers. The server expects a POST request with a JSON body that includes the field "method", set to "add", and a "params" array with two numeric values. If the conditions are met, the function is executed and the result is returned as JSON.

### 7.1.2 RPC client

On the client side, we perform the HTTP POST request. The client constructs a JSON object that mirrors a remote procedure call and sends it to the server (see `01.rpc/client.js` code))

```
const http = require('http');

// Client-side code to call the RPC method
function remoteAdd(a, b) {

  // Create a JSON-RPC request
  const data = JSON.stringify({
    method: 'add',
    params: [a, b]
```

```
});

// Set up the HTTP request options
const options = {
  hostname: 'localhost',
  port: 3000,
  path: '/rpc',
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
    'Content-Length': Buffer.byteLength(data)
  }
};

// Make the HTTP request
const req = http.request(options, res => {
  let body = '';

  res.on('data', chunk => {
    body += chunk;
  });

  res.on('end', () => {
    try {
      const parsed = JSON.parse(body);
      if (res.statusCode === 200) { console.log('Result from RPC call:', parsed.result); }
      else { console.error('RPC Error:', parsed.error); }
    }
    catch (err) { console.error('Error parsing response:', err.message); }
  });
});

req.on('error', err => { console.error('Request error:', err.message); });

req.write(data);
req.end();
}

// Example of calling the remote method
remoteAdd(7, 5);
```

This client defines a function `remoteAdd()` which internally sends a POST request to the RPC

server. The request body contains the method name "add" and the parameters "[7, 5]". When the server processes the request and responds, the client prints the result.

### 7.1.3 Limitations

The client and server must both know the exact name of the method and the structure of its parameters. There is no standardized way to describe or discover available methods, and HTTP features such as method semantics, status codes, and caching are underutilized. These limitations lead to tight coupling between the client and server, making the system harder to evolve and less interoperable with other services.

By contrast, REST proposes a different model, where interaction is structured around named resources, standard HTTP methods, and hypermedia-driven navigation. The next chapter will introduce the foundational principles of REST and demonstrate how the same problem—adding two numbers—can be modeled using RESTful design.

## 7.2 Architectural principles

Representational State Transfer, or REST, is a set of architectural principles designed for building distributed systems that scale well and are easy to maintain over time. Although often associated with web APIs, REST is more than a programming technique or communication protocol, it is an architectural style that defines how networked components should interact.

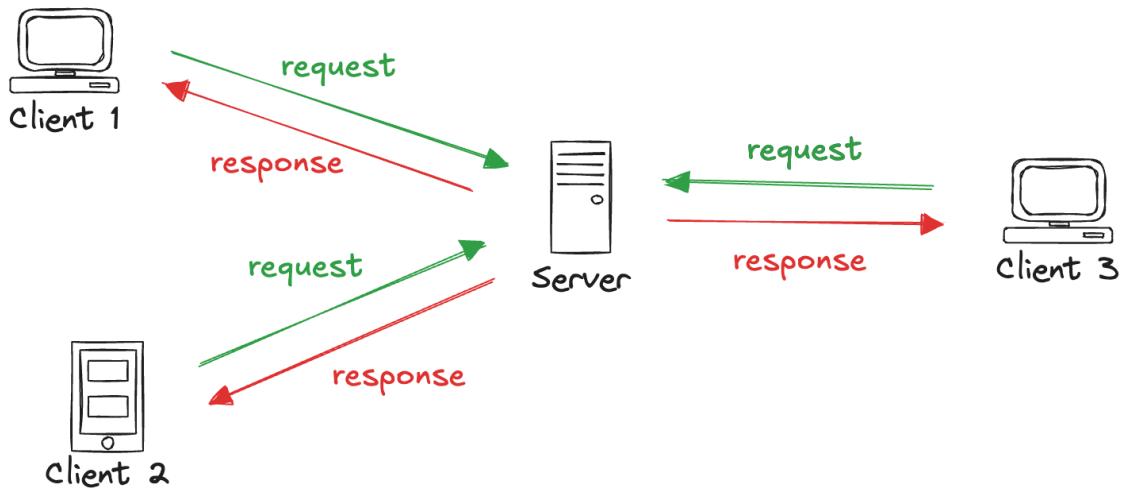
At its core, REST is not a technology but a model: it defines a set of constraints that, if fully respected, result in what is known as a **RESTful** system. Systems that implement these constraints benefit from scalability, reduced latency, simpler interfaces, and independent deployment of components. REST achieves this by advocating a particular way of structuring interactions around resources, standard operations, and stateless communication.

The reason REST has had such a profound impact is that it captures the very nature of the Web itself. The success of the Web is due not to any specific implementation, but to the uniformity, modularity, and simplicity of its architecture. By following the same principles, developers can build applications and services that inherit these same benefits: robustness, flexibility, and universality.

The REST architectural style is defined by five mandatory constraints. Each constraint contributes to the overall architecture by guiding how components interact and what rules they must follow to remain loosely coupled and interoperable.

### 7.2.1 Constraint 1: Client-Server

The client-server constraint defines a clear separation between two roles in a distributed system: the **client**, which initiates requests for services or resources, and the **server**, which processes those requests and returns the appropriate responses. This division creates a boundary of responsibility and establishes a communication model that is both simple and powerful:



In practical terms, every interaction in a RESTful system follows the **request-response pattern**. A client constructs a request that includes a target resource (typically identified by a URI) and sends it over the network. The server receives this request, performs the necessary processing (such as retrieving data, creating or modifying resources) and returns a response that includes the result of the operation, often encoded as a JSON document. This model is the same one used by the World Wide Web, where browsers request web pages and servers respond with HTML content.

The most important effect of the client-server constraint is the **decoupling** of responsibilities. Clients do not need to understand how the server is implemented, how it stores data, or how it makes decisions internally. They only need to know what URIs to call and what format to use for their requests. Similarly, servers are not concerned with the inner workings of clients. They respond to standardized requests and leave the interpretation, display, and user interaction to the client side.

This separation has profound implications for system design. First, it promotes **loose coupling**, which means that clients and servers can evolve independently. A server can change its internal logic, upgrade its storage engine, or migrate to a different platform without requiring any changes on the client side, as long as the interface remains consistent. Likewise, a client appli-

cation can switch between different servers or update its user interface without modifying the server. This flexibility is essential for building scalable, maintainable, and long-lived systems.

Another benefit of the client-server model is the **separation of concerns**. With distinct roles, each side of the communication can specialize in different tasks. The server focuses on data processing, storage, and security, while the client takes care of control logic and presentation. This makes it easier to organize teams, modularize code, and reason about system behavior. It also leads to **portability**, since clients can be implemented on any device or platform, from smartphones and web browsers to embedded sensors, as long as they can send and receive HTTP requests.

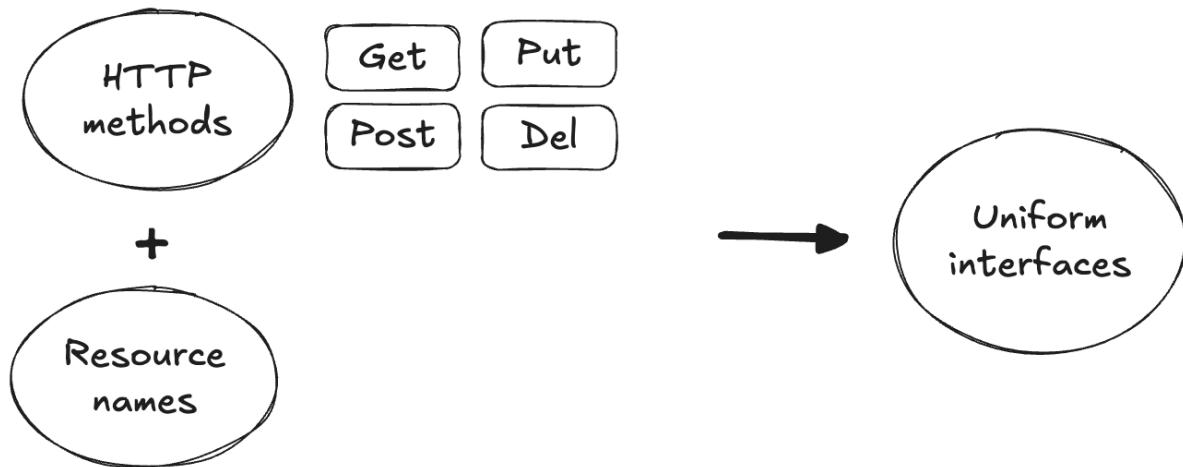
In REST, this architectural constraint is not merely a guideline, it is a structural principle. It defines the nature of the system and determines how all the other constraints (such as statelessness, cacheability, and layering) build upon this foundation. The Web itself is the ultimate proof of its success: billions of independent clients interact with countless servers every day, with no shared assumptions beyond a set of agreed-upon standards and interfaces.

By respecting the client-server boundary, RESTful systems achieve robustness, modularity, and simplicity. Each component can exist and evolve on its own terms, leading to distributed architectures that are resilient to change, open to integration, and ready to scale.

### 7.2.2 Constraint 2: Uniform Interface

The uniform interface constraint defines how components within a distributed system interact and exchange information in a consistent, predictable way. REST mandates that all interactions between clients and servers should conform to a uniform set of rules and conventions. This constraint enables loose coupling between components, which is essential for building systems that are scalable, evolvable, and interoperable.

At its core, the uniform interface simplifies communication. Clients do not need to know the internal logic or structure of a server, nor do they need a specialized protocol to access different services. They only need to understand a small set of standard operations and how to interpret resource representations. In REST, this standard interface is implemented using the semantics of HTTP: clients interact with resources identified by URIs and perform operations on them using well-defined methods such as GET, POST, PUT, and DELETE.



This simplicity is not a limitation but a strength. By constraining the interface, REST promotes **generality**, making it easier to build tools, intermediaries, and libraries that can interact with any RESTful service. It also enhances **extensibility**: since the interface is consistent and self-descriptive, new content types and resource representations can be introduced without changing the communication model. This is one of the reasons why the Web has succeeded as a massively distributed and heterogeneous system.

A uniform interface also plays a crucial role in enabling **automation** and **discovery**. Since the behavior of the system is predictable, clients can be written to interact with servers they have never seen before. This is especially important in the Web of Things, where new devices and services may appear dynamically, often without human intervention. In such an environment, each device should expose its functionality using the same conventions as any other web resource. This allows clients to discover, access, and manipulate devices with minimal configuration or custom logic.

Different HTTP methods correspond to specific actions on resources. A GET request retrieves the representation of a resource. A POST request creates a new resource or triggers a subordinate process. A PUT request updates an existing resource, and a DELETE request removes it. This method-resource matrix is the foundation of the uniform interface. It abstracts application behavior into a small, well-understood vocabulary that can be reused across domains, from web pages and social media APIs to sensors and actuators in smart environments.

The uniform interface also implies that messages exchanged in the system are **self-descriptive**. Each message contains enough information to describe how it should be processed, independently of the context in which it was sent. This means that intermediaries, such as caches, proxies, and gateways, can inspect, route, and even transform messages without needing to understand the application's internal logic. This further enhances the modularity and robustness of RESTful systems.

In summary, the uniform interface constraint is what allows REST to scale across domains, platforms, and use cases. It removes the need for custom protocols or tight integration, enabling openness and broad participation. In REST, uniformity is not about rigidity; it is about creating a **shared language** that all components can understand and extend. Just as the Web allows people to publish and consume information from anywhere in the world, REST allows systems to interact and evolve in an open, consistent, and collaborative manner.

### 7.2.3 Constraint 3: Stateless

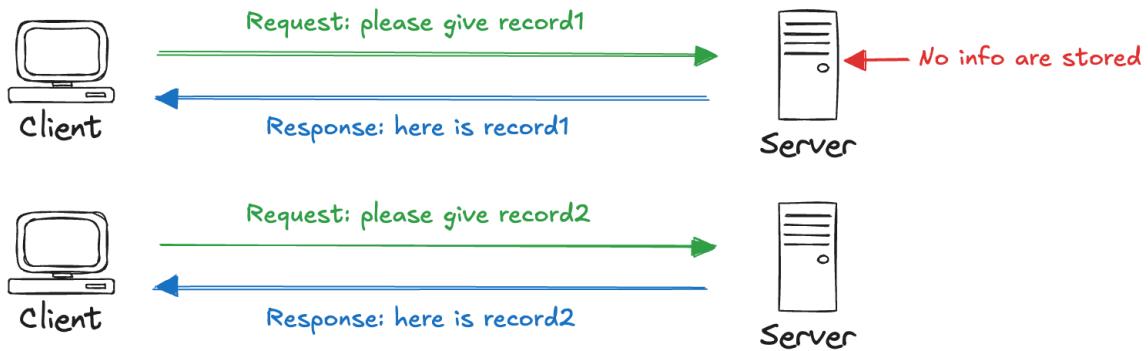
The stateless constraint defines how client-server interactions should be structured. According to this principle, each request from a client to a server **must contain all the information necessary to understand and process that request**. This means that the server does not store any context or session information about the client between different requests. Every client request is treated as an **independent transaction**, with no reliance on information from previous interactions.

This design choice offers several significant advantages. First, it enhances the **visibility** of the system. Since each request is self-contained, developers and system administrators can more easily trace, monitor, and debug client-server communication. Every request reveals its full intent, making logs and monitoring tools more informative and meaningful.

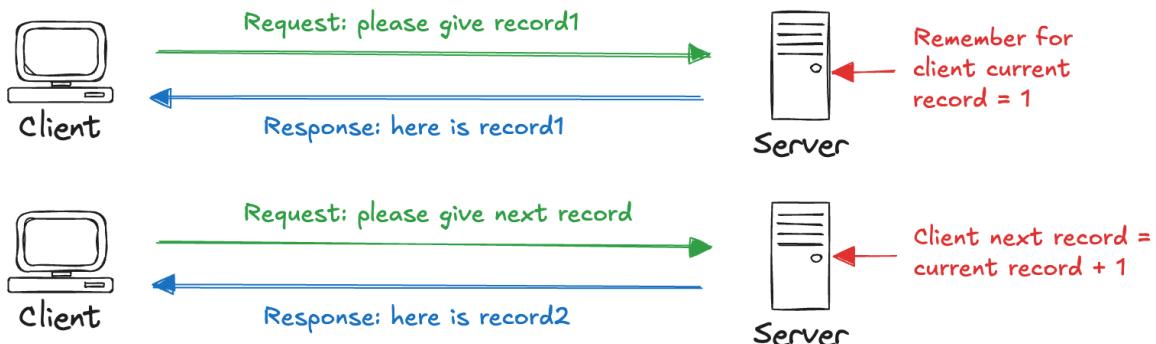
Second, the stateless nature of interactions improves **robustness**. In systems where the server does not retain session information, recovering from network or application failures becomes simpler. A failed server can be restarted without the need to recover or reconstruct lost session data. The client simply continues to send complete, well-formed requests, and the restarted server can resume handling them immediately.

Third, statelessness supports **scalability**. Servers do not need to allocate memory or other resources to maintain client-specific state across multiple requests. As a result, requests can be distributed freely across a pool of servers, allowing for horizontal scaling. Load balancers can route each request to any available instance, ensuring efficient use of resources and enabling systems to handle large volumes of traffic with ease.

To better understand this concept, it is helpful to contrast stateless and stateful interactions. Consider a client asking for a specific resource, such as a data record. In a stateless scenario, the client includes the identifier of the desired record in each request. The server processes each request independently and does not retain any information about what the client requested previously:



In contrast, a stateful server maintains session information. When the client requests a record, the server notes that the client is currently at this position. Later, if the client asks for the "next record", the server uses the stored context to infer the next record. This approach requires the server to store client-specific state, tying the logic of future responses to past interactions:



It is important to note that the stateless constraint **does not prohibit applications or servers from managing state** internally. Rather, it requires that all necessary state be conveyed explicitly in the interactions between components. For example, a web application may allow users to build a shopping cart over time. Even in a stateless RESTful system, this cart can be maintained by the client and sent with each request.

Stateless communication also facilitates other design features. Authentication mechanisms, for instance, often use stateless tokens that include encoded claims about the user's identity and permissions. **Pagination** schemes can be implemented by passing parameters or cursors that specify the desired subset of data. This way, the server does not need to remember where the client left off. Similarly, stateless systems are **more resilient to retry mechanisms**. Since each request contains all required information, it can be safely repeated in case of timeout or failure.

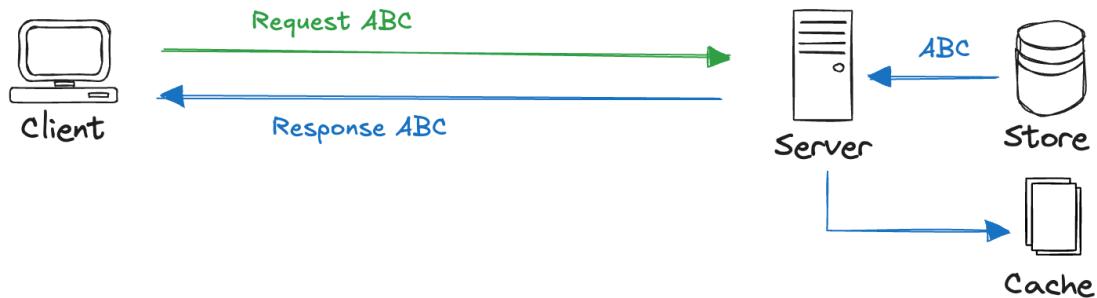
In summary, the stateless constraint enables clean, scalable, and robust interactions in distributed systems. By ensuring that each request is self-contained, statelessness reduces de-

pendencies between client and server, enhances system transparency, and simplifies recovery and scaling strategies. It is a core feature that underpins the success of RESTful architectures and the broader web ecosystem.

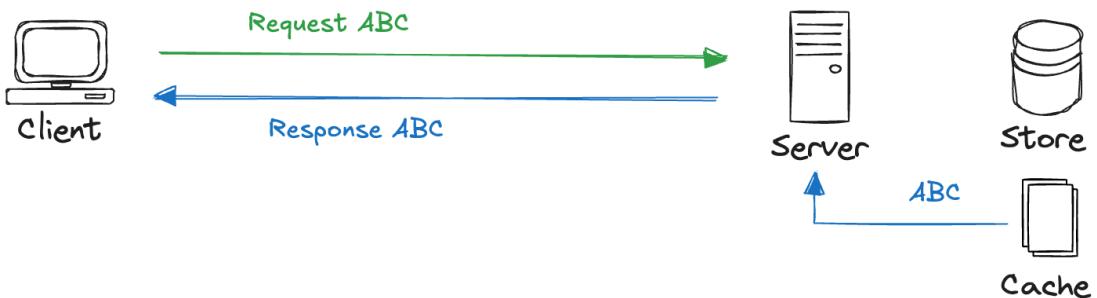
#### 7.2.4 Constraint 4: Cacheable

The cacheable allows responses from the server to be **explicitly marked as cacheable**, permitting clients and intermediaries such as proxies to store them locally and reuse them when appropriate. The idea behind caching is straightforward yet powerful: avoid redundant data transfers by reusing previously retrieved responses.

1 - A new request arrives. Response is created, stored in cache and sent to client



2 - Same request arrives within a short duration. Response is fetched from the cache and sent to client.



In a RESTful architecture, caching is treated as an **architectural feature, not just an implementation detail**. When a server responds to a client request, it can include metadata that instructs the client or an intermediary whether the response may be cached and under what conditions it can be reused. This metadata typically includes HTTP headers like Cache-Control, Expires, or entity tags like ETag. These headers define the rules that govern the lifetime of a cached response and the conditions under which it must be revalidated with the server.

Allowing clients and intermediaries to store data locally has immediate performance advantages. By serving repeated requests from a nearby cache rather than contacting the origin server,

RESTful systems **reduce response time** and **conserve bandwidth**. This is particularly beneficial in high-latency environments or when dealing with mobile or embedded devices. Moreover, caching helps to **alleviate the load on backend servers**. When thousands of users request the same resource, serving them from distributed caches rather than repeatedly computing or retrieving the resource from a central point greatly improves system scalability and availability.

However, the benefits of caching must be balanced against the **risk of serving stale or outdated data**. To maintain data consistency, RESTful systems rely on **caching policies** that control how long data can be reused and when it must be refreshed. These policies can define expiration times or specify that a client must verify with the server whether the cached version is still valid. For instance, a client might send a conditional request using the If-Modified-Since or If-None-Match headers. If the resource has not changed, the server can return a lightweight 304 Not Modified response, thus saving resources and avoiding unnecessary data transfer.

The impact of the cacheable constraint is visible in several areas. From the user's perspective, **applications feel faster** because data is often retrieved from a local cache. From the server's viewpoint, fewer incoming requests translate into **lower processing demands** and improved scalability. From a network perspective, reduced traffic **lowers the chance of congestion** and improves overall efficiency.

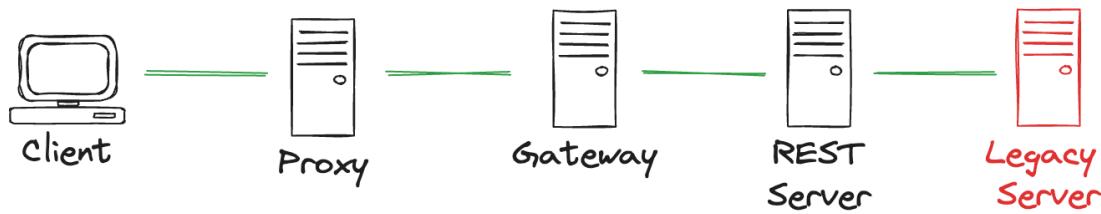
Caching is particularly important in the design of modern web applications and distributed systems. It underpins technologies such as **content delivery networks**, which replicate popular resources across geographically distributed caches, and browser caches, which store static assets like images, scripts, or style sheets locally. In RESTful systems, the cacheable constraint extends this principle to dynamic data and API responses, contributing to a better, faster, and more reliable user experience.

In conclusion, the cacheable constraint reinforces REST's emphasis on statelessness, efficiency, and independence between components. By delegating the responsibility of reusing responses to clients and intermediaries, and by defining clear rules for freshness and invalidation, RESTful systems become more robust and performant. This constraint is a cornerstone of web architecture and remains a key factor in the scalability and success of the modern Internet.

### 7.2.5 Constraint 5: Layered System

The Layered System constraint refers to the architectural principle of **organizing system components into hierarchical layers**. Each layer serves a specific function and interacts only with its immediate neighbors, without needing to know the details of components beyond its adjacent layers. This approach promotes a **clean separation of concerns** and enhances the **modularity** of distributed systems.

One of the main advantages of the layered system is the ability to introduce intermediate components such as **proxies**, **gateways**, and **caches**, which act as mediators between clients and servers. These intermediaries can be used to improve performance, manage security, balance loads, or support legacy systems, all without changing the way clients interact with the system. From the client's perspective, there is **no visible difference** between interacting with the final server and interacting with an intermediary. This abstraction makes the system more flexible and easier to evolve.



An illustrative example is the use of **distributed caches**. These caches can store frequently accessed data at various locations around the globe. When a client requests a resource, the request may be fulfilled by a nearby cache instead of the origin server, leading to faster response times and reduced load on the server infrastructure. The client does not need to know whether the data comes from the server or a cache; it only cares about receiving the correct response.

Another important application of the layered system is in **encapsulating legacy systems**. A RESTful interface can be placed in front of an older backend that uses proprietary or outdated communication protocols. This wrapper acts as a translator between the modern RESTful API and the legacy logic, allowing clients to interact with the old system through a standardized, web-friendly interface.

By enabling such abstractions, the layered system constraint makes it possible to build large, scalable applications in which components can be developed, deployed, and maintained independently. It also enhances security by allowing sensitive operations such as authentication or rate-limiting to be handled at intermediary layers, shielding backend servers from unnecessary complexity and potential threats.

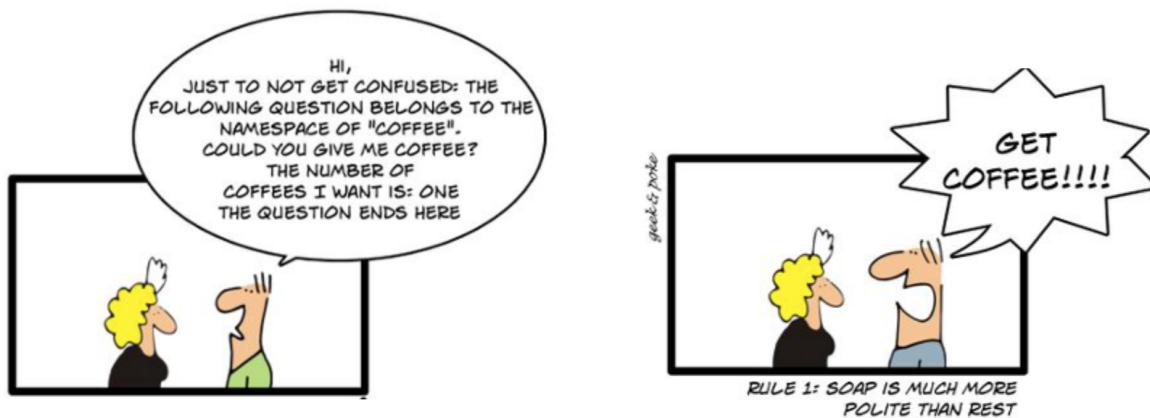
In summary, the layered system constraint allows RESTful systems to be composed of modular, replaceable parts. It supports scalability by allowing load distribution, improves performance through mechanisms like caching, and ensures flexibility by enabling the integration of legacy technologies behind modern interfaces. This principle contributes significantly to the robustness and longevity of web architectures built according to REST.

### 7.2.6 Comparison with SOAP

The Simple Object Access Protocol (SOAP) is a protocol developed by Microsoft in 1998 to facilitate structured and reliable communication between distributed systems. At the time, the primary focus was on enabling RPC style communication over the Internet using XML, a format that was gaining traction for its ability to represent complex data in a human and machine readable way.

SOAP operates primarily over HTTP but is **transport-neutral**, meaning it can also function over other protocols. SOAP messages are strictly formatted as XML documents and follow a specific structure composed of an envelope, a header, and a body. The body of the message contains the actual data or command to be executed, and the header may carry metadata such as authentication or transaction controls. SOAP was widely adopted in the early 2000s, particularly in enterprise systems, because of its ability to provide advanced features such as built-in security, transaction management, and formal interface definitions through WSDL (Web Services Description Language).

Despite these strengths, SOAP has gradually fallen out of favor for most web applications due to its **verbosity**, **rigidity**, and **high implementation complexity**. As web technologies evolved, developers began to seek lighter and more flexible alternatives for building web services. This shift led to the emergence of REST, since it does not define a protocol but rather a set of constraints that, when followed, enable scalable and loosely coupled systems. Instead of wrapping data in envelopes and using XML schemas as SOAP does, RESTful systems treat data as resources and exploit completely the HTTP semantic:



The major conceptual differences lies in **how each approach treats HTTP**. In SOAP systems, HTTP is simply a transport mechanism. The semantics of the operations are embedded within the SOAP body and understood through WSDL contracts. REST, on the other hand, embraces HTTP as its foundation, using its methods directly to express operations and relying on URI

patterns to identify resources. This makes REST more aligned with the native architecture of the Web and simplifies development by eliminating the need for additional messaging protocols or service descriptions.

REST gained popularity throughout the 2000s, especially as companies like Amazon, Google, Twitter, and Facebook began offering web services that followed RESTful principles. Its simplicity, use of standard HTTP methods, and compatibility with JSON instead of XML made REST highly attractive for building web and mobile applications. Developers appreciated not having to generate WSDL documents or manage SOAP envelopes; instead, they could design APIs that were intuitive, fast, and directly accessible from any browser or HTTP client. The rise of smartphones and the proliferation of connected devices further accelerated the shift toward REST, as these environments required efficient data exchange and low-power communication models.

Today, while SOAP continues to be used in specific domains such as enterprise integration, financial transactions, and governmental systems, REST has become the dominant paradigm for API design. The historical shift from SOAP to REST reflects a broader evolution in software architecture: from rigid, standardized contracts designed for reliability to flexible, web-native systems optimized for speed, usability, and scalability.

### 7.3 Resources

In RESTful architecture, the fundamental building block is the **resource**. A resource can be any piece of data or conceptual entity that an application manages: a sensor in a smart home or a user profile, a document, or even an abstract process. This abstraction shifts the development mindset away from traditional Remote Procedure Call (RPC) mechanisms toward an interaction model based on resources and representations.

RESTful systems treat everything (documents, images, users, sensors, services, or even abstract concepts) as resources. These resources are exposed via URIs and are manipulated using standard HTTP methods. This design promotes **loose coupling** between client and server components and leverages the existing infrastructure of the Web to build scalable, interoperable, and evolvable systems.

The design of those resources should follow a few principles: addressability, manipulation through representations, self descriptiveness, and hateoas.

#### 7.3.1 Principle 1: Addressable Resources

A resource is an **abstraction** that represents a piece of information or capability. It is not the data itself, but rather a conceptual mapping to a set of entities or services. For example, a temperature

sensor does not expose its internal state directly, but instead provides an interface to retrieve a current temperature reading or configure settings. The principle of Addressable Resources asserts that each of these entities must be **uniquely identifiable** and **accessible** via a **Uniform Resource Identifier (URI)**. This principle is foundational in enabling scalable, discoverable, and interoperable web services.

The structure of a URI adheres to a general syntax:

```
`<scheme> ":" <authority><path> [ "?" query ] [ "#" fragment ]`
```

This structure allows for hierarchical organization and scoping of resources. For example:

- `http://gateway.api.com/devices/TV/` identifies a television device.
- `https://kitchen:3000/fridge/` points to a fridge in a kitchen-based local server.

In the context of the Web of Things (WoT), every device must have a root URL that corresponds to its network location. These URLs are not arbitrary; they should reflect the logical structure and relationships of the physical or virtual entities they represent.

Resources are often structured hierarchically using path segments. This allows developers to model relationships, containment, or grouping:

- `http://devices.webofthings.io/24/sensors` might represent all sensors on device ID 24.
- `http://192.168.44.12/building4/devices/` could list all devices within a specific building.

The hierarchy encoded in the URI can reveal **spatial**, **organizational**, or **functional relationships** among Things, which is particularly relevant in IoT environments.

While there are no strict rules governing the semantics of resource identifiers, several **best practices** help make APIs more intuitive and maintainable:

- **Use Descriptive Names:** choose terms that carry semantic meaning to humans, improving readability and usability. For instance, `/temperature`, `/sensors`, or `/status` is much more understandable than vague identifiers like `/a1b2c3`.
- **Avoid Verbs in URIs:** URIs should name resources, **not actions**. Actions are communicated through HTTP methods. For example:
  - Bad: `/garagedoor/openDoor`
  - Good: `/garagedoor/status`, the GET method retrieves the status, and a PUT may change it.
- **Use Plurals for Collections:** when referring to a collection or aggregate of similar resources, the plural form should be used:

- /sensors for a collection of sensor entities
- /students for a list of student resources

These conventions ensure consistency and aid both human and machine consumers in interpreting API semantics effectively.

### 7.3.2 Principle 2: Manipulation through Representations

A resource is an abstract concept, it is not a physical object or data file, but a virtual entity that represents some information or functionality within the system. What clients interact with are not the resources themselves, but rather their **representations**. A representation is a **concrete and tangible instance of a resource's state** at a given time, **encoded in a particular format** suitable for transmission over a network.

These representations are standardized using **media types**, more commonly known as MIME types. MIME types describe how a resource is formatted and enable clients and servers to correctly interpret the data being exchanged. For example, an image encoded in PNG format is labeled as "image/png", and an audio file in MP3 format is labeled as "audio/mp3". The complete list of official MIME types is maintained by the Internet Assigned Numbers Authority (IANA), ensuring consistency and reliability across different platforms and implementations.

RESTful systems take advantage of HTTP **content negotiation mechanism** to determine the most appropriate representation format for a given request. A client can specify its preferred formats using HTTP headers such as "Accept", and the server will respond with the best available representation that meets those preferences. This mechanism increases flexibility by allowing the same resource to be available in multiple formats, depending on the needs of the client and the capabilities of the server.

In the context of the Web of Things, the selection of a representation format becomes even more critical due to constraints such as limited memory, low processing power, and battery efficiency. Currently, JSON is the most widely adopted format, since it is lightweight, portable, self-contained, and easy to parse, making it ideal for constrained environments. However, because it is still a text-based format, it may not be optimal for devices that require more compact data transmissions.

To address this, alternative formats such as **MessagePack** have been proposed. MessagePack is a **binary serialization format** that retains the structure and semantics of JSON while providing a smaller and faster encoding. It is recently an officially recognized MIME type (2024) and it shows promise for scenarios where bandwidth and storage are critical concerns.

Best practices in RESTful WoT systems dictate that all devices and services must support **JSON as the default representation format** to ensure baseline compatibility. Additionally, they **may**

**offer HTML-based representations, particularly when a human interface is required**, such as in dashboards or configuration panels.

Ultimately, the concept of representations reinforces the separation between the abstract nature of resources and their concrete instances.

### 7.3.3 Principle 3: Self-descriptiveness

Each message exchanged between client and server should be self-descriptive. This means that the message must contain all the information necessary to understand and correctly process the request or the response, without relying on prior knowledge or out-of-band information. By embedding meaning into the message itself (through standardized structures, metadata, and semantics) REST promotes interoperability, reliability, and the evolution of distributed systems.

In practical terms, self-descriptive messages in REST rely on the use of standard HTTP methods (GET, POST, PUT, DELETE) each with **well-defined semantics**. These correspond to the **basic CRUD operations**: Create, Read, Update, and Delete. By adopting these verbs consistently, systems can expose predictable interfaces, enabling clients to interact with resources without knowing implementation details. This design significantly contributes to **loose coupling**: clients only need to understand and support the semantics of these operations, and not how a server internally manages its state or resources.

This principle maps particularly well to the Web of Things, where devices usually expose simple, atomic functionalities. For instance, a sensor might provide its current value via a GET request, while an actuator like a smart bulb may be controlled through a PUT request that modifies its state. Because devices are limited in capability and often operate in constrained environments, the predictability and simplicity of self-descriptive messages are crucial for achieving interoperability and ease of integration.

HTTP **status codes** are another key aspect of this principle. These codes (such as 200 OK, 404 Not Found, 500 Internal Server Error) provide a lightweight and standardized mechanism to inform clients about the outcome of their requests. Clients can **understand the result of an operation without any prior agreement** with the server, thus enhancing visibility and error handling in distributed applications.

To ensure that RESTful systems respect this principle, a number of guidelines should be followed:

- Every device or service must support the core HTTP verbs in a way consistent with their semantics. For example, a GET on a sensor returns its reading, a POST might create a new configuration or rule, a PUT updates an existing resource, and DELETE removes it.

- HTTP status codes must be implemented correctly, covering both success (2xx series) and various error conditions (4xx and 5xx series).
- Devices must support a GET request on their root URL, so that clients can always obtain metadata or descriptive information about the resource.

By making every message self-contained and understandable in isolation, REST enables scalable, distributed systems that are easier to evolve and debug. This principle is not just about adhering to HTTP standards, it is about designing messages and interfaces that are **explicit**, transparent, and robust in the face of the heterogeneity and dynamism typical of web-scale systems.

#### 7.3.4 Principle 4: HATEOAS

Hypermedia as the Engine of Application State (HATEOAS) ensures that clients interact with a REST API dynamically through **hyperlinks** embedded in the resource representations, without requiring out-of-band information or hardcoded workflows. This principle transforms the interaction between clients and servers into a discoverable, navigable experience, like to browsing the human-readable Web.

At the heart of HATEOAS lie two key ideas:

- **Hypermedia** refers to the use of links (and sometimes forms or embedded controls) within the representations of resources to indicate potential actions or navigational paths. These links guide the client through available operations and transitions without needing prior knowledge of the API's structure.
- **Application state** refers to a step in a process or workflow in which the client resides, the set of available links reflects the current state and the valid next steps.

To implement HATEOAS effectively, **every possible application state is treated as a RESTful resource** with its own unique URI. For instance, the current state of a smart device (like the color or status of an LED) can be accessed through a specific URL. A client can retrieve a JSON representation of that state, which also includes links to other operations: such as turning the LED on or off, changing its brightness, or querying related sensors. These navigational affordances are crucial in both human-driven interactions and machine-driven automation.

An essential characteristic of HATEOAS is that it decouples clients from the internal logic of the server. The client does not need to know in advance which URIs to invoke for future steps, it simply follows the links provided in the current response. This means that **the server fully controls the application flow and exposes possible transitions**, dynamically adapting the API surface depending on context. For example, suppose to querying a bank account resource:

```
GET /accounts/12345 HTTP/1.1
Host: api.example.com
```

The server might return different follow-up links depending on the account balance: an account with positive balance might offer options to withdraw, deposit, or close the account:

```
{
  "account": {
    "account_number": 12345,
    "balance": {
      "amount": 1000.00,
      "currency": "USD"
    },
    "links": {
      "deposit": {
        "href": "/accounts/12345/deposits",
        "method": "POST"
      },
      "withdraw": {
        "href": "/accounts/12345/withdrawals",
        "method": "POST"
      },
      "transfer": {
        "href": "/accounts/12345/transfers",
        "method": "POST"
      }
    }
  }
}
```

whereas an overdrawn account might only allow deposits:

```
{
  "account": {
    "account_number": 12345,
    "balance": {
      "amount": -25.00,
      "currency": "USD"
    },
    "links": {
      "deposit": {
```

```
        "href": "/accounts/12345/deposits",
        "method": "POST"
    }
}
}
```

This **dynamic discovery model** is especially relevant for the Web of Things. In WoT systems, devices often change their capabilities or available operations over time. With HATEOAS, clients can discover related resources (such as actuators, rules, or sensor readings) without relying on a fixed sitemap. Human users can browse resources intuitively, while machine clients can crawl or traverse them programmatically, enabling scalable automation and dynamic reconfiguration of IoT environments.

Ultimately, HATEOAS embodies the Web's ethos of **navigation by exploration**. By embedding state-transition logic within resource representations, it creates **self-documenting APIs** that naturally evolve. Clients can interact with them without version-specific manuals, hardcoded paths, or brittle assumptions: just by following the links. In this way, HATEOAS not only enhances loose coupling and resilience, but also makes RESTful APIs more flexible, extensible, and aligned with the distributed nature of the Web itself.

## 7.4 Richardson Maturity Model

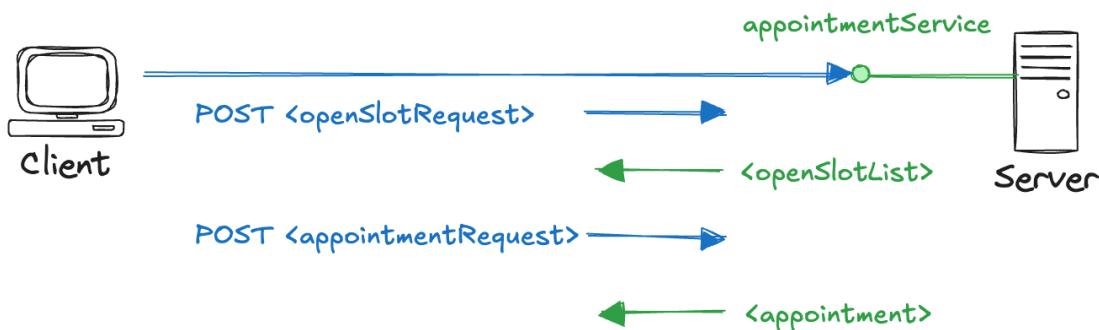
The Richardson Maturity Model is a conceptual framework that evaluates the maturity of a web service's RESTful design. Proposed by Leonard Richardson, it breaks down the journey toward achieving a **fully RESTful architecture** into four progressive levels. Each level introduces a deeper commitment to the REST architectural style and brings the implementation closer to what is often called the "Glory of REST".

### 7.4.1 Level 0: The Swamp of POX

At the base of the model lies Level 0, sometimes pejoratively described as the "Swamp of POX" (Plain Old XML). This level represents systems that **use HTTP merely as a transport mechanism for remote procedure calls (RPC)**, with no regard for RESTful principles. The interface typically consists of **a single URI** that serves as an entry point for all operations, and **HTTP methods are often ignored** in favor of tunneling every interaction through POST. This approach negates the semantics of HTTP, often ignoring its native capabilities like caching, content nego-

tiation, and status codes. Systems at this level are tightly coupled and hard to scale or evolve independently.

As a running example, we consider a simple web service that allows users to get an appointment from a doctor. In a level 0 implementation, the service exposes a single endpoint, such as /appointmentService, and all operations are performed using POST requests. The request body contains XML or JSON data that specifies the action to be taken (e.g., "getAppointment", "cancelAppointment"). The server processes the request and returns a response, but the client has no way of knowing what operations are available or how to interact with the service beyond the initial POST:



Here a simple implementation of a level 0 service in Node.js (see 02.level0/server.js code):

```

const express = require('express');
const app = express();

app.use(express.json());

// Simulated database
const openSlots = [
  { id: 1, time: "2025-05-12T10:00:00" },
  { id: 2, time: "2025-05-12T11:00:00" }
];

// Simulated appointment storage
const appointments = [];

// RPC-like API for appointment service
app.post('/appointmentService', (req, res) => {

  const { method, payload } = req.body;

  // Retrieve available slots
  if (method === 'getAvailableSlots') {
    res.json(openSlots);
  } else if (method === 'makeAppointment') {
    const appointment = {
      id: Date.now(),
      time: payload.time,
      patientName: payload.patientName
    };
    appointments.push(appointment);
    res.json(appointment);
  }
});
  
```

```
if (method === 'openSlotRequest') {
  res.json({ openSlotList: openSlots });
}

// Handle appointment requests
else if (method === 'appointmentRequest') {
  const { slotId, user } = payload;

  const slot = openSlots.find(s => s.id === slotId);
  if (!slot) { return res.status(400).json({ error: "Slot not available" }); }

  // Simulate appointment creation
  const appointment = {
    id: appointments.length + 1,
    slot,
    user
  };

  // Add appointment to storage
  appointments.push(appointment);

  // Reply with appointment details
  res.json({ appointment });
}

// Handle unknown methods
else { res.status(400).json({ error: 'Unknown method' }); }
});

// Start the appointment service server
app.listen(3000, () => {console.log('Level 0 RPC API running on
  ↵ http://localhost:3000');});
```

And here is a simple client that consumes the service:

```
POST http://localhost:3000/appointmentService
Content-Type: application/json
{
  "method": "openSlotRequest",
  "payload": {}
}
```

```
POST http://localhost:3000/appointmentService
Content-Type: application/json
{
  "method": "appointmentRequest",
  "payload": {
    "slotId": 1,
    "user": "Alice"
  }
}
```

The Level 0 is limited. All interactions are funneled through a single endpoint, typically using the POST method, regardless of the nature of the operation. This means that even simple read operations, such as retrieving a list of available appointment slots, are treated the same way as state-changing operations like creating an appointment. As a result, the **API does not benefit from HTTP's semantics**, such as using GET for safe, idempotent reads, or DELETE for resource removal.

Additionally, the approach **lacks resource orientation**. Instead of exposing identifiable resources like /appointments/123, it hides the underlying domain behind abstract method calls such as "appointmentRequest", which must be known and understood in advance by the client. This **tight coupling** between client and server **reduces the discoverability and evolvability of the API**. Clients cannot easily understand or adapt to new functionality without prior knowledge, and the messages are not self-descriptive.

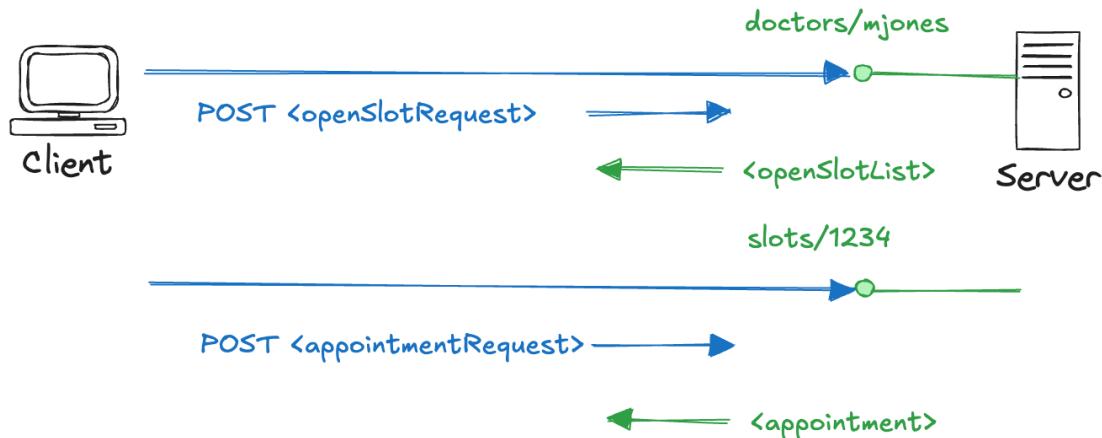
The absence of distinct URIs also means the system **cannot leverage the benefits of HTTP features such as caching, conditional requests, and proper status codes**. Intermediaries like proxies or CDNs are less effective, and observability suffers because logging, monitoring, and debugging are more difficult when every action passes through the same generic endpoint.

Ultimately, this RPC-style API design sacrifices scalability, flexibility, and long-term maintainability for the convenience of a simplistic interface, making it harder to extend or integrate in complex, distributed environments.

#### 7.4.2 Level 1: Resources

Level 1 introduces the concept of **resources**. Rather than exposing a single service endpoint, the system starts organizing its capabilities around uniquely identifiable resources, each with its own URI. This decomposition helps tackle complexity by enabling a divide-and-conquer strategy: instead of one monolithic service, the API is structured as a set of addressable entities. For example, /students, /courses, and /grades become distinct endpoints. While this level

brings clarity and a modular structure to the API, it still often relies on a single HTTP verb like POST for all operations, which limits its RESTfulness. We can modify the previous example to use resources:



Here the modified server code (see 03.level1/server.js code):

```
// A resource for each doctor
app.post('/doctors/:doctorId', (req, res) => {
  const doctorId = req.params.doctorId;
  const slots = openSlots[doctorId] || [];
  return res.json({ openSlotList: slots });
});

// A resource for each slot
app.post('/slots/:slotId', (req, res) => {
  const slotId = parseInt(req.params.slotId);
  const user = req.body.user;
  const slot = Object.values(openSlots).flat().find(s => s.id === slotId);
  if (!slot) { return res.status(404).json({ error: "Slot not found" }); }

  const appointment = {
    id: appointments.length + 1,
    slot,
    user
  };

  appointments.push(appointment);
  return res.json({ appointment });
});
```

The improvement from Level 0 to Level 1 lies in **introducing resource-specific URIs**, allowing

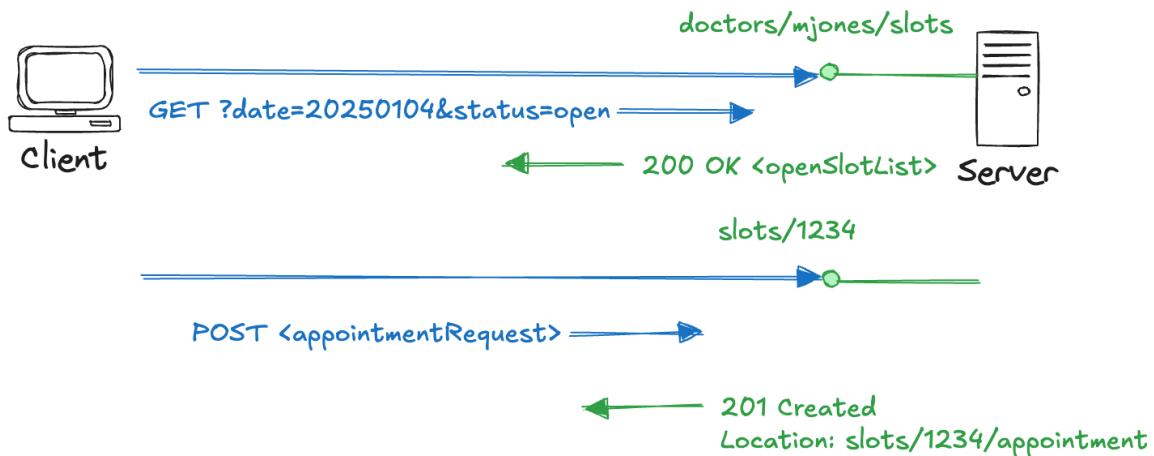
the API to identify and target individual entities like doctors or slots, rather than routing all actions through a single generic endpoint. Now the client can interact with the API using URIs:

```
POST http://localhost:3000/doctors/1
Content-Type: application/json
{}

POST http://localhost:3000/slots/2
Content-Type: application/json
{
  "user": "Alice"
}
```

#### 7.4.3 Level 2: HTTP Verbs

At Level 2, the service begins to **fully embrace the semantics of HTTP** by using its standard verbs (GET, POST, PUT, DELETE) appropriately across resources. These verbs correspond naturally to CRUD operations (Create, Read, Update, Delete). A GET request retrieves a resource, POST creates a new one, PUT updates an existing resource, and DELETE removes it. This approach standardizes interaction patterns, eliminates unnecessary variation, and enables clients to operate with predictable behaviors. Level 2 significantly improves scalability, interoperability, and ease of use because it adheres to well-known web standards. The example can be modified to use the HTTP verbs:



Here the modified server code (see 04.level2/server.js code):

```
// The slots for a doctor using GET
app.get('/doctors/:doctorId/slots', (req, res) => {
  const { doctorId } = req.params;
  const slots = openSlots[doctorId]
  res.status(200).json({ openSlotList: slots });
});
```

The improvement from Level 1 to Level 2 lies in the proper use of HTTP methods to express the semantics of operations. While Level 1 introduced resource-oriented URIs, it still relied exclusively on POST for all actions, treating every operation as a remote procedure. In contrast, Level 2 aligns with RESTful principles by using different HTTP verbs, such as GET for retrieving data and POST for creating new resources, making the API more expressive, predictable, and aligned with standard web architecture. Now the client can interact with the API using the appropriate HTTP methods:

```
GET http://localhost:3000/doctors/1/slots
Accept: application/json

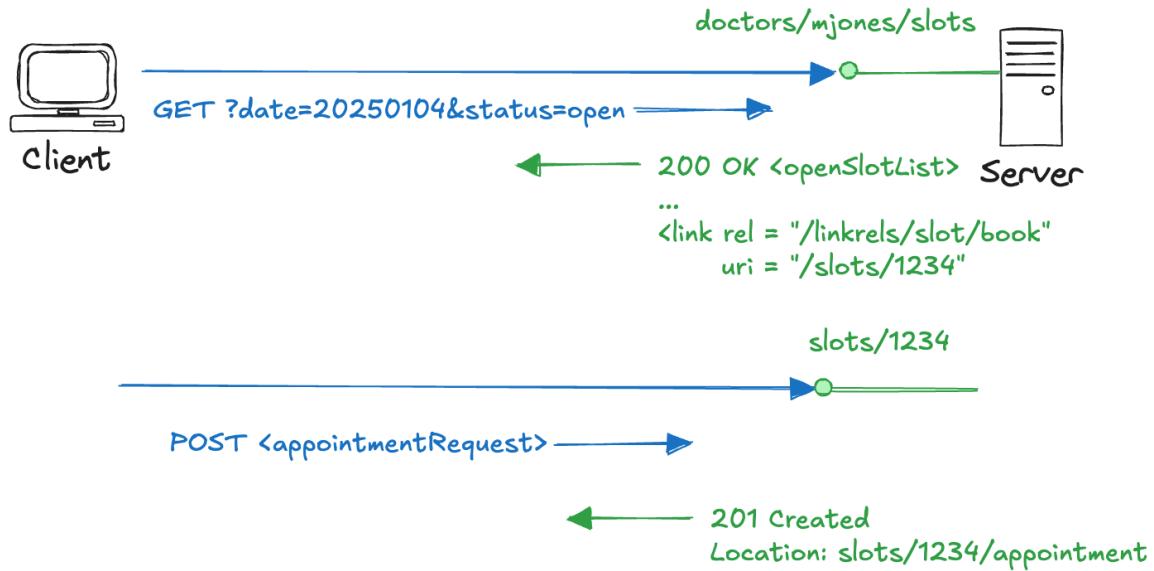
POST http://localhost:3000/slots/1
Content-Type: application/json
{
  "user": "Alice"
}
```

This shift improves clarity for clients, who can now infer the nature of the operation from the HTTP verb itself, and enables better support for HTTP features like caching, idempotency, and status code conventions. Filtering through query parameters becomes more natural and standardized, enhancing the discoverability of resources. Additionally, responses can now include meaningful status codes (e.g., 200 OK, 201 Created) and headers like Location, helping clients understand the outcome of their requests without relying solely on custom response formats.

#### 7.4.4 Level 3: Hypermedia Controls (HATEOAS)

The final level, Level 3, introduces **Hypermedia as the Engine of Application State** (HATEOAS), a cornerstone of REST as defined by Roy Fielding. Here, the server provides not only the data but also the **available actions** a client can take next, in the form of hyperlinks embedded in the response. These hypermedia controls guide clients through the application workflow without relying on out-of-band knowledge. This means clients can **dynamically**

**discover** the API's capabilities by following links rather than hardcoding endpoint URIs and expected actions. This level supports better decoupling, adaptability, and resilience to change. It also improves the API's discoverability, making it more self-documenting. The example can be modified to use hypermedia controls:



Here the modified server code (see 05.level3/server.js code):

```

app.get('/doctors/:doctorId/slots', (req, res) => {
  const { doctorId } = req.params;
  const slots = openSlots[doctorId]

  const enrichedSlots = slots.map(slot => ({
    id: slot.id,
    time: slot.time,
    _links: {
      book: {
        href: `/slots/${slot.id}`,
        rel: "Make appointment",
        method: "POST"
      }
    }
  }));
  res.status(200).json({ openSlotList: enrichedSlots });
});
  
```

The improvement from Level 2 to Level 3 lies in the introduction of hypermedia controls, which

make the API truly RESTful by enabling clients to navigate the application dynamically through links provided in the server's responses. While Level 2 uses proper HTTP verbs and resource-oriented URIs, it still requires clients to hardcode knowledge of those URIs and workflows. With Level 3, the server guides the client by embedding links that describe available actions, such as how to book an appointment for a slot. This decouples the client from the internal URI structure and allows the API to evolve more easily, since clients follow links rather than constructing paths manually. It also improves discoverability, self-documentation, and flexibility, as each resource response includes affordances for the next available operations, making interactions more intuitive and state-driven.

#### 7.4.5 The Glory of REST

Summarizing the Richardson Maturity Model, REST offers a powerful architectural style for building scalable and maintainable web services. Its strength lies in a layered approach to simplifying complexity and improving interoperability.

- At **Level 1**, REST addresses complexity through **divide and conquer**: a large service endpoint is broken down into multiple distinct resources. This separation allows developers to manage and understand the system more easily by isolating concerns and structuring the API around meaningful entities.
- **Level 2** advances this structure by introducing a standard set of HTTP verbs (such as GET, POST, PUT, DELETE), enabling the system to **handle similar situations in the same way**. This reduces variability, encourages consistency, and simplifies client-server interactions by applying a uniform interface to all resources.
- **Level 3** brings in the concept of **discoverability** through hypermedia. By embedding links and actions within responses, RESTful systems become **self-documenting**, guiding clients through available operations without requiring external documentation.

Finally, REST realizes the Web's original vision by treating HTTP not merely as a transport mechanism but as an **application layer protocol**. It allows the creation of APIs for Things (e.g., IoT devices) by reusing the proven architecture of the Web, integrating them as first-class citizens within the existing Web ecosystem.

### 7.5 Real-World Examples

Understanding REST architecture in theory is essential, but examining how it is applied in real-world APIs is what truly brings its principles to life. We explore three widely-used web APIs to illustrate how REST constraints and best practices are adopted in practical implementations.

Each of these APIs serves a different domain: **Twitter (now X)** manages social interactions and user-generated content, **eBay** provides access to online marketplaces, and **PayPal** supports secure payment and billing operations. Despite their differing scopes and technical histories, all three APIs expose HTTP-based interfaces that allow developers to interact with their underlying services in a structured, programmatic way.

By analyzing these APIs, we highlight the diversity of REST implementations in the wild and reveal the spectrum between partial and full adoption of the REST architectural style. These examples help clarify what it means for an API to be **RESTful** in practice and what trade-offs different providers make between strict adherence to REST and practical integration needs.

### 7.5.1 The X (Twitter) API

The X API (formerly Twitter API) is a compelling and instructive example of a RESTful API implemented at scale. It demonstrates how the REST architectural style can support a high-performance, distributed, and developer-friendly communication interface for a global social media platform. By exposing resources such as tweets, users, likes, and followers via clearly structured URIs and standard HTTP methods, X API embodies the foundational REST constraints and principles:

<https://docs.x.com/x-api/introduction#x-api-v2>

At its core, the X API is organized around addressable resources. Each tweet, user, or list is treated as a uniquely identifiable resource, which can be accessed through a stable and descriptive URL. For example, the endpoint:

<https://api.x.com/2/tweets?ids=1234567890>

allows clients to retrieve the representation of a specific tweet. This aligns directly with the REST principle of addressable Resources, where each resource must have a unique identifier, often implemented using a URI. Representations of these resources are returned in JSON, including fields like id, text, lang, author or embedded user objects, which are representations of associated resources:

```
{  
  "id": "1212092628029698048",  
  "text": "This is the tweet text",  
  "edit_history_tweet_ids": [  
    "1212092628029698048"  
  ],
```

```
"author_id": "2244994945",
"lang": "en",
"in_reply_to_user_id": "2244994945",
"public_metrics": {
    "retweet_count": 7,
    "reply_count": 3,
    "like_count": 38,
    "quote_count": 1
},
"created_at": "2019-12-31T19:26:16.000Z",
"attachments": {
    "media_keys": [
        "16_1211797899316740096"
    ]
},
"possibly_sensitive": false,
...
}
```

The X API makes use of standard HTTP verbs to implement actions over resources, illustrating the principle of self-descriptive messages. This facilitates loose coupling between client and server: developers can interact with the API by understanding the semantics of HTTP without needing to know the server's internal structure.

Although the current version of the X API (v2) partially implements hypermedia controls, the HATEOAS constraint is not fully embraced. While responses occasionally include embedded links and resource identifiers that facilitate discoverability, the API does not yet provide dynamic navigation of state transitions. This means that while the API is rich and consistent, clients still need to have prior knowledge of endpoint structures and workflows, rather than discovering them at runtime.

### 7.5.2 eBay API

The **eBay Trading API** in the XML-based version as found at

<https://developer.ebay.com/Devzone/XML/docs/Reference/eBay/index.html>

is not a good example of a RESTful API. It exemplifies a more traditional RPC-style architecture over HTTP, which falls under Level 0 of the Richardson Maturity Model. Instead of exposing addressable resources with URIs, the API relies on a single endpoint:

<https://api.ebay.com/ws/api.dll>

and invokes different actions via XML payloads that include a call name to tunneling remote procedure calls through HTTP. There are no meaningful or descriptive URIs representing resources. Everything is addressed by the same base URL, violating the principle of addressable Resources in REST. All operations, whether reading, creating, or deleting data, are sent via POST. The HTTP method does not reflect the semantic meaning of the operation, which breaks the self-descriptive Messages principle. Instead of using standard HTTP methods to convey intent, it uses internal call names like AddItem, GetItem, EndItem, encoded in the body. This is a classic sign of RPC, not REST. While not disqualifying per se, the use of large XML schemas and SOAP-like semantics adds verbosity and complexity. Finally, there are no links or affordances in the response payloads to guide the client to next available actions or resources, meaning it lacks hypermedia controls entirely.

Instead, ehe **eBay Buy Browse API**:

[https://developer.ebay.com/api-docs/buy/browse/resources/item\\_summary/methods/search](https://developer.ebay.com/api-docs/buy/browse/resources/item_summary/methods/search)

is a significant departure from the older XML-based Trading API, and it aligns much more closely with REST principles. The Buy Browse API is a Level 2 RESTful API (and nearly Level 3), using proper resource URIs and HTTP verbs, and returning JSON. It reflects a clear effort by eBay to adopt modern RESTful design and improve developer experience.

### 7.5.3 PayPal API

The PayPal API, as documented at

[https://developer.paypal.com/docs/api/\]\(https://developer.paypal.com/docs/api/\)](https://developer.paypal.com/docs/api/)

provides a robust and modern implementation of REST principles. Designed to handle a wide range of payment-related operations (including orders, payments, subscriptions, and identity verification) the API exemplifies a well-structured, secure, and developer-friendly RESTful service. It serves as a valuable real-world case study of how REST can be applied to a complex, security-critical domain like online payments.

Rather than exposing actions or procedures, the API defines addressable entities such as

`/v2/checkout/orders`

`/v1/payments/payment`

`/v1/billing/plans`

Each URI uniquely identifies a specific resource and reflects its place in the hierarchy of the system, fulfilling the REST constraint of addressable resources. For instance, the URL

<https://api.paypal.com/v2/checkout/orders/5O190127TN364715T>

identifies a specific order and allows it to be retrieved, updated, or captured using the corresponding HTTP verbs. Furthermore, responses include standard HTTP status codes, which support clear and consistent error handling across client applications. The representation format used is exclusively JSON, allowing for compact, machine-readable payloads that can easily be parsed by a wide range of client applications. Another strength of the PayPal API is its implementation of HATEOAS: each response includes a links array that details available next actions. This makes the API more discoverable, as clients can dynamically adapt to the workflow by following links rather than hardcoding endpoint logic. For example, after creating a payment order, the client receives a link that directs the user to approve the payment in their browser. Once approved, another link allows the server to capture the payment. Finally, PayPal provides **sandbox environments**, detailed **OpenAPI-based documentation**, and interactive examples that guide developers through common workflows. This contributes to a strong developer experience and encourages best practices for secure API integration. The PayPal REST API is a mature and well-engineered example of a level 3 RESTful API according to the Richardson Maturity Model.

#### 7.5.4 Comparison

A comparative table that evaluates the **Twitter (X)**, **eBay**, and **PayPal** APIs against the key **REST constraints** and the **Richardson Maturity Model** levels:

REST Principle / Feature	Twitter (X API v2)	eBay Trading API (XML)	eBay Buy Browse API	PayPal REST API
<b>Resource-Oriented URIs</b>	Yes – /2/users/:id, /2/tweets/:id	No – Single endpoint, action in body	Yes – /item_summary/search	Yes – /checkout/orders
<b>HTTP Methods Used Properly</b>	Yes – GET, POST, DELETE	No – Only POST used for all actions	Yes – GET for search	Yes – GET, POST, PATCH, DELETE

REST Principle / Feature	Twitter (X API v2)	eBay Trading API (XML)	eBay Buy Browse API	PayPal REST API
<b>Standard Representations</b>	JSON (compact and standard)	XML (heavy, legacy)	JSON	JSON with content negotiation
<b>Stateless Interactions</b>	Yes – Each request includes all context	Nominally stateless, but RPC-like	Yes	Yes
<b>Self-descriptive Messages</b>	Uses HTTP methods and status codes	Custom call names in XML payload	Clear use of methods and codes	Rich status codes and headers
<b>Cacheable Responses</b>	Yes – Supports HTTP caching	No HTTP-level caching semantics	Yes – Cacheable search queries	Yes – Caching headers for some resources
<b>Hypermedia Controls</b>	Partial – Some resource IDs, but limited links	None	Partial – No full HATEOAS	Full HATEOAS with action links
<b>Richardson Level</b>	<b>Level 2 (2.5)</b>	<b>Level 0</b>	<b>Level 2</b>	<b>Level 3</b>

**PayPal API** is the most complete implementation of REST, reaching **Level 3** with **HATEOAS**, proper resource modeling, full use of HTTP semantics, and hypermedia-driven workflows. **Twitter (X) API** offers a solid **Level 2.5** implementation—modern, JSON-based, and well-structured, but lacking full hypermedia support. **eBay Buy Browse API** is a clean, REST-aligned API at **Level 2**, offering structured JSON endpoints and standard HTTP usage, but also stops short of full HATEOAS. **eBay Trading API** is a classic **Level 0 RPC API**, using HTTP merely as a transport layer for remote procedure calls with XML payloads.

## 7.6 Beyond REST: the Real-Time

Traditional RESTful APIs, built upon the **request-response paradigm** of HTTP, function well in many web applications, including much of the WoT where clients explicitly retrieve sensor

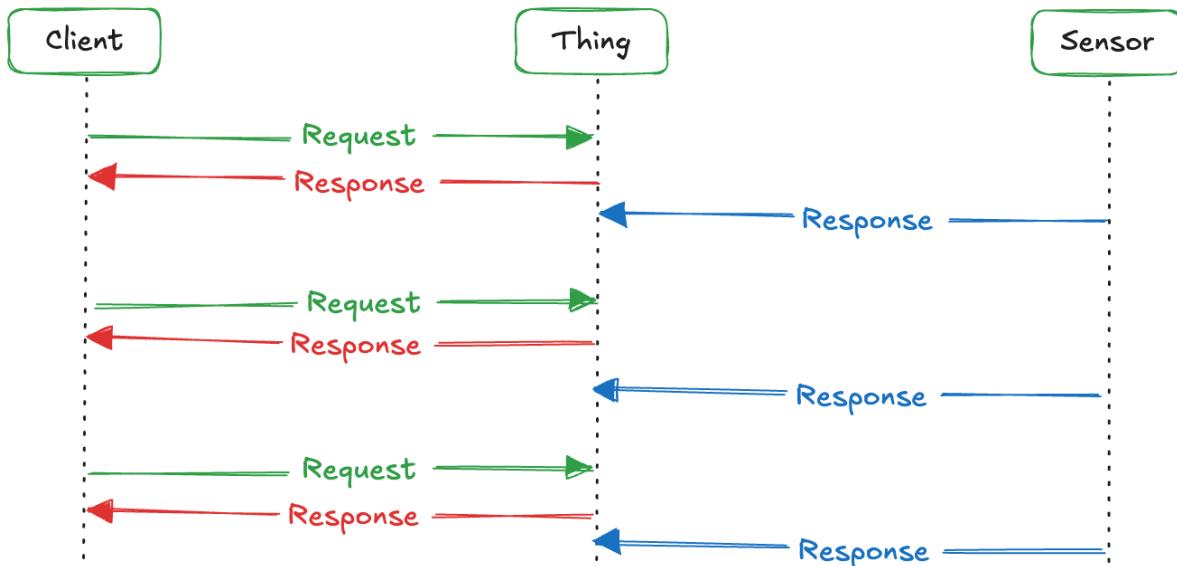
data or control actuators. However, this model becomes increasingly inadequate as WoT evolves toward more **dynamic**, **interactive**, and **responsive** environments. In particular, **real-time communication** is essential when devices must notify clients asynchronously as events occur.

In the WoT context, the classical **client-initiated communication** is effective only for scenarios in which the client periodically polls a device for updated information, such as reading a temperature sensor from a mobile app. But this **polling strategy** is inefficient, leading to excessive network traffic and unnecessary energy consumption, especially for battery-operated or low-power devices. It is also fundamentally unsuitable for **event-driven use cases** that require low-latency, on-demand delivery of information. Imagine a fire alarm or a security camera that needs to immediately alert users or systems in response to an anomaly: waiting for the client to initiate a request would introduce unacceptable delays.

To enable truly reactive behaviors in such scenarios, the WoT must move beyond REST's strict request-response model and embrace **asynchronous, server-initiated communication**. This shift allows devices to **push** data and alerts directly to subscribers when specific conditions are met, rather than waiting to be polled. Implementing this push-based logic requires integration with **publish/subscribe patterns**.

### 7.6.1 Basic Polling

Basic polling is a strategy used to **simulate real-time behavior** by having clients repeatedly send HTTP GET requests to a Thing at regular intervals:



Each request asks for the latest data, such as a sensor reading, and receives a response contain-

ing the current value. The server exposes a resource (e.g., /temperature) that represents the sensor's state (see 06.polling/server.js code):

```
const express = require('express');

const app = express();
const port = 3000;

// Simulated sensor value
let sensorValue = 0;

// REST endpoint to get sensor value
app.get('/sensor', (req, res) => {
  res.json({ value: sensorValue });
});

app.listen(port, () => { console.log(`Web Thing listening at
  ↳ http://localhost:${port}`); });
```

The client can poll this resource every few seconds to get the latest reading (see 06.polling/-client.js code):

```
const http = require('http');

function pollSensor() {
  const options = {
    hostname: 'localhost',
    port: 3000,
    path: '/sensor',
    method: 'GET',
  };

  const req = http.request(options, res => {
    let data = '';
    res.on('data', chunk => data += chunk);
    res.on('end', () => {
      try {
        const parsed = JSON.parse(data);
        console.log(`Polled Sensor Value: ${parsed.value}`);
      }
      catch (err) { console.error('Error parsing response:', err.message); }
    });
  });
}
```

```
});  
  
req.on('error', err => { console.error('Request error:', err.message); });  
  
req.end();  
}  
  
// Poll every 2 seconds  
setInterval(pollSensor, 2000);
```

Finally, we simulate a sensor update value every 5 seconds (see /polling/server.js code):

```
setInterval(() => {  
    sensorValue = Math.floor(Math.random() * 100);  
    console.log(`Sensor updated: ${sensorValue}`);  
, 5000);
```

While this approach is **simple to implement** and compatible with RESTful architectures, it has significant limitations that make it **inefficient** and **unsuitable for scalable or energy-sensitive applications**.

The primary inefficiency of basic polling lies in the **large number of redundant HTTP calls** it generates. Many of these requests yield no new information, especially when the underlying data has not changed since the last poll. This constant stream of HTTP traffic **does not scale well** with an increasing number of clients and may quickly overwhelm the server.

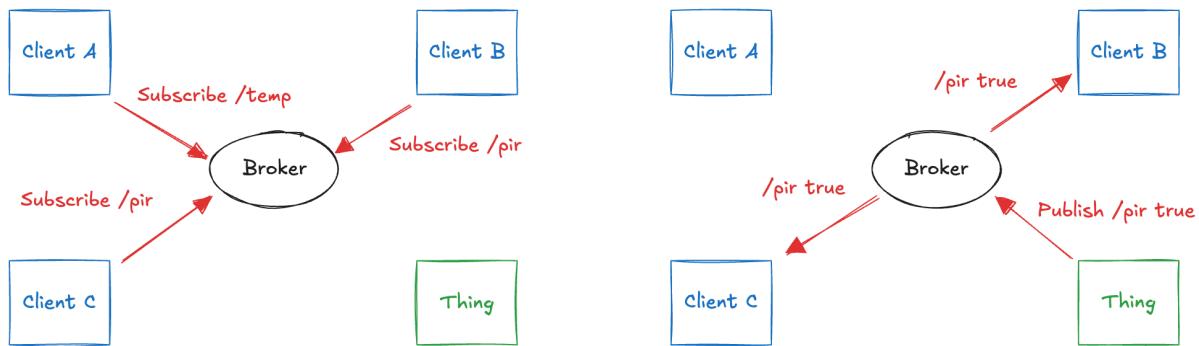
Moreover, basic polling is particularly **problematic for battery-powered devices**, which must minimize energy consumption and data transmission. The repeated HTTP requests not only waste network resources but also reduce battery life by forcing unnecessary communication cycles.

The client continuously issues requests to the Web Thing, which queries the underlying sensor. Even if no significant event has occurred, the server responds with the same data, leading to wasted effort on both sides. This design fails to efficiently address dynamic, event-driven scenarios and highlights the need for more advanced mechanisms.

### 7.6.2 Publish/Subscribe pattern

In the context of the WoT, real-time responsiveness is essential to support applications that need to react promptly to dynamic conditions, such as motion detection, alarm triggering, or

sensor value thresholds. The traditional REST model, which relies on a client-server request/response interaction, falls short when it comes to efficiently managing these asynchronous and event-driven scenarios. To overcome this limitation, the **publish/subscribe (Pub/Sub) model** emerges as a powerful architectural pattern. The pattern introduces a fundamental decoupling between data **producers** (**publishers**) and data **consumers** (**subscribers**). Instead of clients repeatedly polling for new data, publishers send messages to a central component called a **broker**, which is responsible for routing and distributing these messages to the appropriate subscribers. Subscribers express their interest in specific types of events or data (e.g., subscribing to temperature or motion updates), and the broker ensures that they receive relevant updates as soon as they are published:

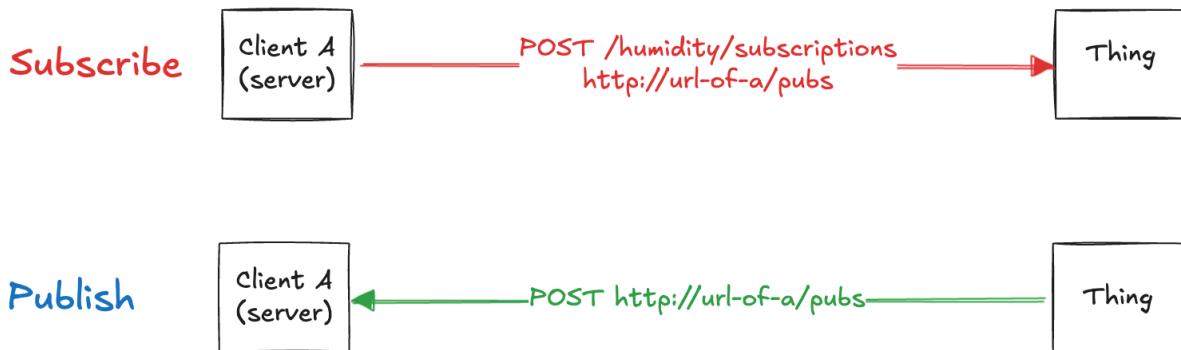


This mechanism supports highly scalable and dynamic environments, where clients may come and go, and publishers may emit a wide variety of data. The analogy to a chat room is particularly apt: some channels are public, others are private, and communication can be one-to-one or one-to-many, depending on the nature of the subscriptions.

To implement this pattern on the Web, several technologies can be leveraged. Among the most popular are **WebHooks**, **Long Polling**, and **WebSockets**, each with different trade-offs in terms of latency, complexity, and compatibility. These technologies enable the broker to push data to clients as soon as it becomes available, providing a foundation for reactive and responsive WoT applications.

### 7.6.3 WebHooks

**WebHooks** are a simple and effective way to implement server-to-server communication in an event-driven architecture. Unlike basic polling, where the client repeatedly requests updates, WebHooks reverse the flow: the server (the Thing) actively sends updates to a client-provided URL when something relevant occurs:



This approach treats every entity (both the device and the client) not only **as a consumer** but also as **a potential server**. The client exposes an endpoint and subscribes by telling the Thing where to send updates (e.g., a URL). When the event occurs, the Thing sends an HTTP POST request with the relevant data directly to that endpoint. This model is widely used across the web, for example, by PayPal to notify other systems when a payment is processed.

The main limitation is that the subscriber (client) must be accessible from the public internet. This makes it difficult to use in scenarios involving private networks, mobile apps, or battery-powered IoT devices.

We can write a simple publisher that take care of registering the subscriber and sending the updates (see 07.webhook/publisher.js code):

```
const express = require('express');
const http = require('http');

const app = express();
app.use(express.json());
const port = 3000;

const subscribers = [];

// Endpoint to subscribe to events
app.post('/subscribe', (req, res) => {
  const { url } = req.body;
  if (!url) { return res.status(400).json({ error: 'Missing URL in request body' }); }
  subscribers.push(url);
  res.status(201).json({ message: 'Subscribed successfully' });
});

// Internal function to push events to all subscribers
```

```

function publishEvent(value) {
  const data = JSON.stringify(value);

  // Send the event to all subscribers
  subscribers.forEach(subscriberURL => {

    // Parse the URL to get the hostname and port
    const { hostname, port, pathname } = new URL(subscriberURL);

    // Create an HTTP request to the subscriber
    const options = {
      hostname: hostname,
      port: parseInt(port, 10),
      path: pathname,
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
        'Content-Length': data.length
      }
    };

    // Send the request to the subscriber
    const req = http.request(options, res => { console.log(`Event sent to
      ${subscriberURL} - Status: ${res.statusCode}`); });
    req.on('error', err => { console.error(`Error sending to ${subscriberURL}:`,
      err.message); });
    req.write(data);
    req.end();
  });

  app.listen(port, () => { console.log(`Thing server running at
    http://localhost:${port}`); });
}

```

We simulate a sensor update value every 5 seconds and when the event occurs, we send an HTTP POST request to the subscriber URL (see 07.webhook/publisher.js code):

```

setInterval(() => {

  // Simulate a sensor event
  const event = {
    sensor: 'humidity',

```

```
    value: Math.floor(Math.random() * 100),
    timestamp: new Date().toISOString()
};

// Log the event and publish it to subscribers
console.log('Publishing event:', event);
publishEvent(event);
}, 5000);
```

An a simple subscriber that receives the updates (see 07.webhook/subscriber.js code):

```
const express = require('express');
const http = require('http');
const app = express();
const port = 5001;

app.use(express.json());

// Webhook endpoint to receive events
app.post('/webhook', (req, res) => {
  console.log('Received event:', req.body);
  res.sendStatus(204);
});

// Server to listen for incoming events
app.listen(port, () => {
  console.log(`Subscriber listening on http://localhost:${port}/webhook`);

  // Automatically subscribe to the Thing
  const data = JSON.stringify({ url: `http://localhost:${port}/webhook` });

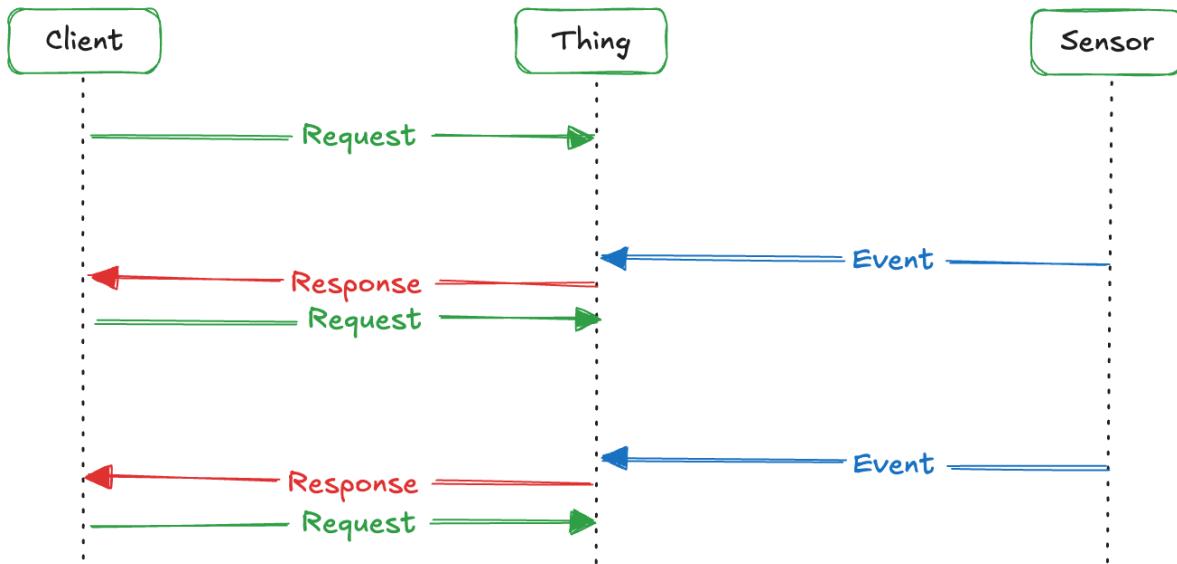
  const options = {
    hostname: 'localhost',
    port: 3000,
    path: '/subscribe',
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
      'Content-Length': data.length
    }
};
```

```
const req = http.request(options, res => { console.log(`Subscribed to Thing -`  
` Status: ${res.statusCode}`);});  
req.on('error', err => { console.error('Error subscribing:', err.message); });  
req.write(data);  
req.end();  
});
```

This example shows how WebHooks can mimic a lightweight Pub/Sub mechanism using only HTTP. It's simple and compatible with REST principles, but remember: it requires the client to be accessible over the network, which is not always practical in WoT scenarios.

#### 7.6.4 Long Polling

**Long polling** is a technique that improves on basic polling by keeping the client's HTTP request **open** until the server has data available to return. This reduces unnecessary traffic and latency, allowing near real-time communication between devices. When the client sends a request, the server holds it open and responds only when an event occurs. The client then immediately opens a new request to wait for the next event.



It is a practical **workaround** to achieve near real-time communication over HTTP, but it comes with **notable limitations** that must be considered, especially in the context of the Web of Things. One major issue is the **connection overhead**: each client maintains an open connection while waiting for a server response, which consumes server resources such as memory and sockets. As the number of connected clients grows, this can lead to scalability challenges and performance

degradation. Another limitation lies in the **reconnection cycle**. After each event is delivered, the client must immediately re-establish a new connection. Although this reduces idle requests compared to basic polling, it still introduces unnecessary network traffic and overhead due to repeated connection setups and teardowns.

Here a simple implementation of a long polling server (see 08.longpolling/server.js code):

```
const express = require('express');
const app = express();
const port = 3000;

// The latest event to be sent to clients
let latestEvent = null;

// The list of clients waiting for events
let waitingClients = [];

// Long polling endpoint
app.get('/events', (req, res) => {

    // Save the response object to respond later
    waitingClients.push(res);

    // Timeout after 25 seconds to avoid broken connections
    req.setTimeout(25000, () => {
        const index = waitingClients.indexOf(res);
        if (index !== -1) waitingClients.splice(index, 1);
        res.status(204).end(); // No Content
    });
});

app.listen(port, () => { console.log(`Thing running on http://localhost:${port}`); });
```

We simulate a new hevent every 5 seconds and send it as a JSON response to all clients waiting for updates. After responding, we clear the list of waiting clients (see 08.longpolling/server.js code):

```
setInterval(() => {

    // Simulate a sensor event
    latestEvent = {
        sensor: 'humidity',
```

```

    value: Math.floor(Math.random() * 100),
    timestamp: new Date().toISOString()
};

console.log('New event:', latestEvent);

// Respond to all waiting clients
waitingClients.forEach(res => res.json(latestEvent));
waitingClients = []; // Clear list
}, 5000);

```

This client repeatedly sends GET requests to /events on the server. If a new event is received (status 200 with data), it logs the event and immediately polls again. If no event is received or a timeout occurs, it retries (see 08.longpolling/client.js code):

```

const http = require('http');

// Create a polling request
function poll() {
  const options = {
    hostname: 'localhost',
    port: 3000,
    path: '/events',
    method: 'GET'
  };

  // Send the request to the server
  const req = http.request(options, res => {
    let data = '';

    res.on('data', chunk => data += chunk);

    // Handle the end of the response
    res.on('end', () => {
      if (res.statusCode === 200 && data) {
        const event = JSON.parse(data);
        console.log('Received event:', event);
      }
      else { console.log('No new event (timeout).'); }

      // Immediately reconnect after receiving response
      poll();
    });
  });
}

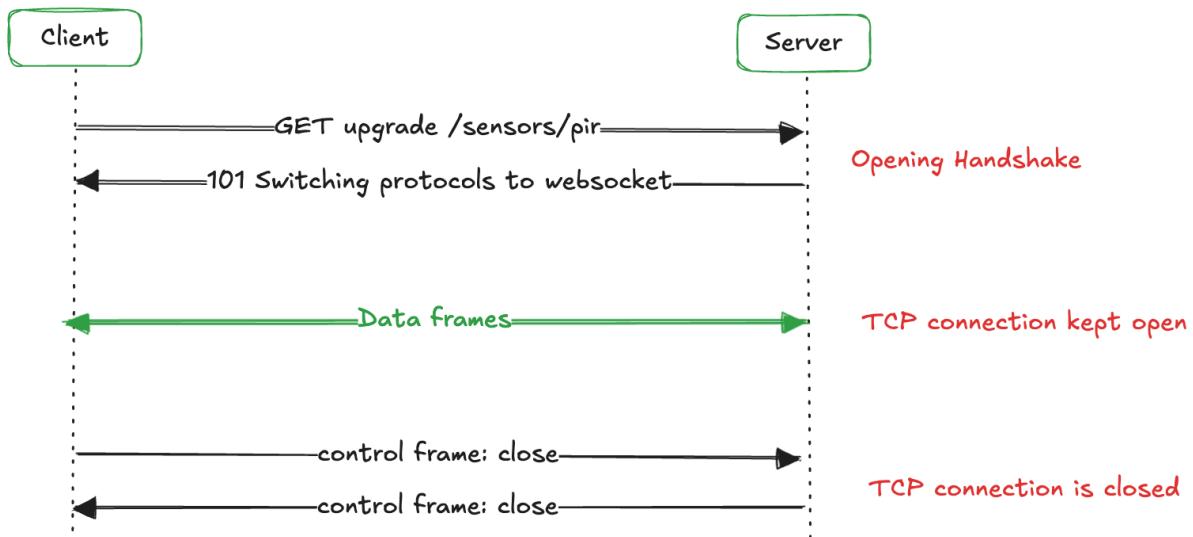
```

```
});  
  
req.on('error', err => {  
  console.error('Polling error:', err.message);  
  // Retry after delay  
  setTimeout(poll, 2000);  
});  
  
req.end();  
}  
  
// Start polling loop  
poll();
```

### 7.6.5 WebSockets

WebSockets represent a modern, **efficient solution** for real-time, bidirectional communication between clients and servers. Unlike long polling, which simulates reactivity by repeatedly opening and closing HTTP connections, WebSockets **establish a persistent TCP connection** that remains open, allowing data to flow in both directions as events occur. This makes WebSockets particularly well-suited for applications in the Web of Things, where devices must frequently send and receive timely updates.

The WebSocket protocol begins with a standard HTTP request containing an **upgrade header**. If the server supports WebSockets, it responds with a **101 Switching Protocols** status code. Once the handshake completes, the connection transitions from HTTP to a **full-duplex TCP channel**. From this point forward, both the client and server can asynchronously push messages to each other using WebSocket frames instead of traditional HTTP requests and responses. The communication overhead is minimal: each frame requires only a 2-byte header, a substantial improvement over HTTP's verbose structure:



This lightweight, always-on channel is ideal for scenarios where latency, responsiveness, and resource efficiency are critical, such as controlling actuators, receiving sensor updates, or broadcasting events to multiple devices. WebSockets overcome the limitations of polling and long polling by providing true push-based communication with minimal delay and network overhead.

NodeJS provides a built-in WebSocket library called `ws` that simplifies the implementation of WebSocket servers and clients. The library allows developers to create real-time applications with ease, enabling them to handle multiple connections, broadcast messages, and manage events efficiently. Here's a simple example of a WebSocket server and client using the `ws` library (see `09.websocket/server.js` code):

```

const WebSocket = require('ws');

// Create a WebSocket server
const wss = new WebSocket.Server({ port: 3000 });

// Array to hold connected clients
let clients = [];

// Handle WebSocket connections
wss.on('connection', ws => {
  console.log('Client connected');

  // Add the new client to the array
  clients.push(ws);
}
  
```

```
// Manage closing connections
ws.on('close', () => {
  console.log('Client disconnected');
  clients = clients.filter(client => client !== ws);
});
});
```

As before, the code simulates an event every 5 seconds and broadcasts it as a JSON message to all connected WebSocket clients that are still open (see 09.websocket/server.js code)::

```
setInterval(() => {

  // Simulate a sensor event
  const event = {
    sensor: 'humidity',
    value: Math.floor(Math.random() * 100),
    timestamp: new Date().toISOString()
  };
  console.log('Broadcasting:', event);

  // Send the event to all connected clients
  const message = JSON.stringify(event);
  clients.forEach(ws => {
    if (ws.readyState === WebSocket.OPEN) { ws.send(message); }
  });
}, 5000);
```

The client code creates a WebSocket that connects to the server, logs when it's connected or disconnected, and prints any received sensor event. It also handles and logs connection errors (see 09.websocket/client.js code):

```
const WebSocket = require('ws');

// Create a WebSocket client
const ws = new WebSocket('ws://localhost:3000');

// Handle WebSocket events
ws.on('open', () => { console.log('Connected to WebSocket server'); });

ws.on('message', data => {
```

```
const event = JSON.parse(data);
console.log('Received event:', event);
});

ws.on('close', () => { console.log('Disconnected from server'); });

ws.on('error', err => { console.error('WebSocket error:', err.message); });
```

## 7.7 Hands-on Activity

Design and implement (using NodeJS and Express) a RESTful API to manage a university course. You have to design resources (like students, exams, grades, etc.), routes (taking into account hierarchy, like /students/id/exams), representations (the JSON body of each resource). We don't have any database technology until now, so, please keeps data in memory using arrays.

## 8 WoT Proxy

The implementation of a Web of Things Proxy involves a structured and progressive design process aimed at integrating physical devices (sensors and actuators) into the web ecosystem. This process transforms low-level embedded devices into first-class web citizens, accessible and controllable via standardized interfaces. The implementation path proceeds through **five tightly interconnected design dimensions**, each responsible for a different layer of abstraction and functionality.

The first step is the **Integration Strategy**, where a decision is made on how the physical devices, referred to as Things, will connect to the Internet and the Web. This choice depends on the capabilities of the devices themselves. Some can implement web protocols like HTTP or Web-Sockets natively, while others, due to power or computational constraints, require the support of an intermediary such as a gateway or a cloud platform. The selected integration pattern (direct, gateway-based, or cloud-based) defines how the Thing exposes its services to the web.

Once connectivity is defined, **Resource Design** organizes the Thing functionalities into logical services. These are modeled as REST resources, mapping the physical components into a hierarchical structure that mirrors their digital representation. The design must consider what resources are exposed, how they are grouped, and the semantic meaning associated with each endpoint.

After resources are identified, **Representation Design** focuses on the format in which data is exchanged. Since RESTful architectures are agnostic to representation, the server can support multiple formats such as JSON, HTML, or the compact binary format MessagePack. This design step enables content negotiation and ensures compatibility with a wide range of clients, including those operating in constrained environments.

The **Interface Design** extends the expressivity of the API beyond simple data retrieval. It introduces support for HTTP verbs like PUT and POST, enabling interaction with actuators and other mutable resources. Additionally, it defines error handling mechanisms and status codes, and may introduce asynchronous communication models using WebSockets to support real-time updates from sensors.

Lastly, **Resource Linking Design** establishes relationships among resources, allowing clients to navigate the Thing API through hypermedia links. This enriches the interface by making it self-descriptive and discoverable, aligning with the REST constraint of HATEOAS (Hypermedia as the Engine of Application State).

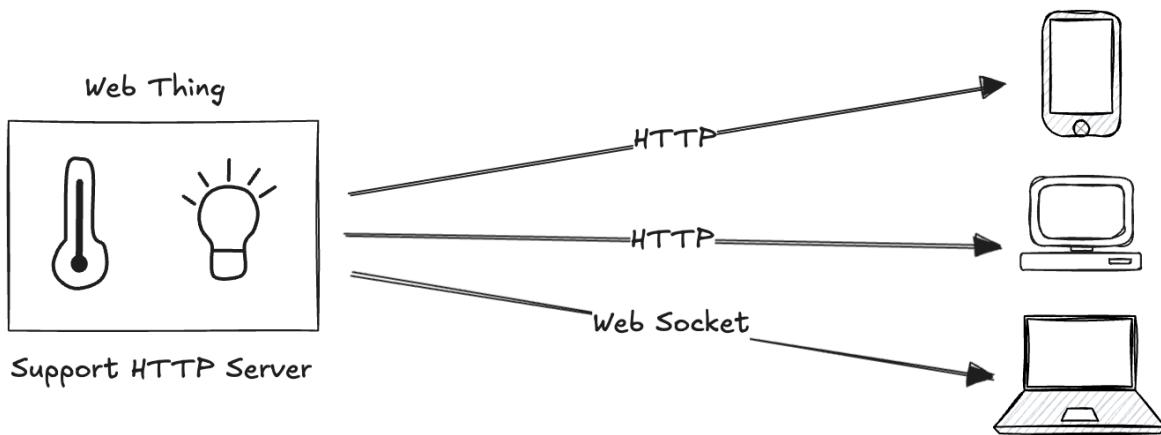
Together, these design steps constitute the foundation of a scalable and interoperable WoT architecture. They are not isolated phases but interdependent layers that must be coherently aligned to transform embedded devices into services integrated seamlessly with the Web.

## 8.1 Integration Strategy

The integration strategy establishes how the API of a physical Thing is exposed on the Web, based on its networking capabilities and hardware constraints. It determines whether the Thing can serve requests directly, needs support from a gateway, or relies on cloud infrastructure. This decision shapes the overall architecture of a Web of Things system.

### 8.1.1 Direct Integration Pattern

In the direct integration pattern, the Thing is **capable of implementing Web protocols on its own**. It exposes a Web API directly over HTTP or WebSockets and handles requests without intermediaries. This is the cleanest and most interoperable setup, suitable for devices that can afford the necessary memory, processing power, and energy consumption:

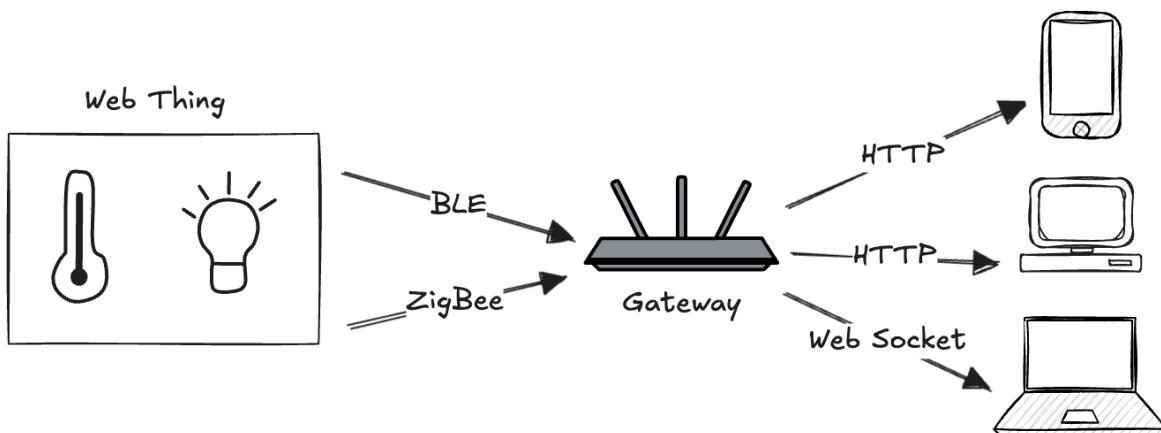


For example, a Raspberry Pi equipped with a temperature sensor runs an Express.js server. It exposes a RESTful API where clients can perform "GET /temperature" to retrieve current readings, or "POST /log" to store data remotely. The Pi handles HTTP and WebSocket requests natively and serves responses using JSON or HTML.

This approach is ideal in local networks where latency must be minimized, or where the Thing must interact with other Web components directly. However, it assumes reliable connectivity and sufficient system resources.

### 8.1.2 Gateway Integration Pattern

Many Things (such as small microcontrollers) cannot support HTTP or TCP/IP stacks due to **hardware limitations**. In these cases, a gateway device is introduced to act as a **bridge**. It communicates with the constrained Things using low-power protocols (like Bluetooth, ZigBee, or even serial) and exposes their data through a unified Web API:



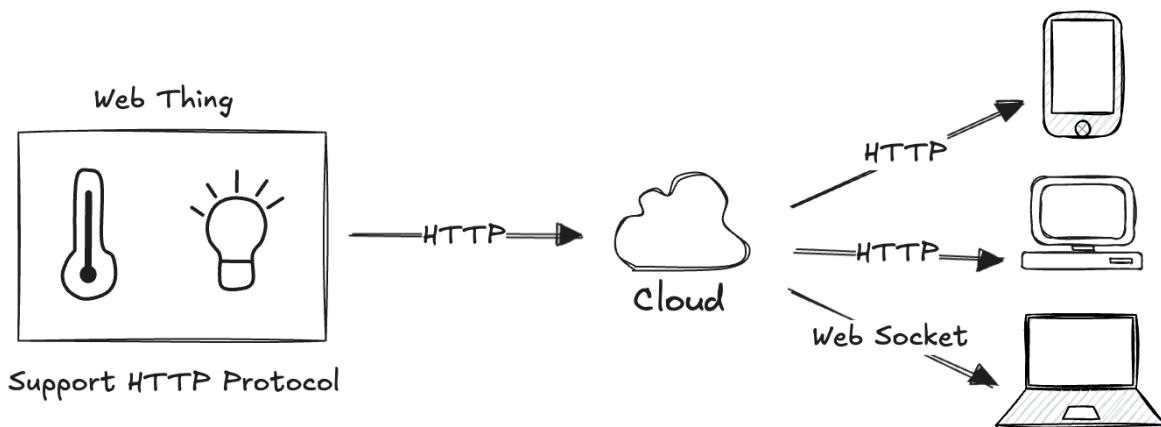
For Example, several Arduino boards equipped with sensors communicate via serial over USB

with a Node.js-based gateway running on a laptop or embedded controller. The gateway collects sensor values and exposes them as REST resources (e.g., "GET /sensors/light", "PUT /actuators/led") on an HTTP interface. Clients only interact with the gateway, which abstracts the complexity of the underlying devices.

This pattern also allows for protocol translation, caching, or security filtering and is a key enabler of so-called **fog computing architectures**, where intermediate devices process data near the edge of the network.

### 8.1.3 Cloud Integration Pattern

When devices **can connect to the Internet** but **lack sufficient capabilities** to expose a full Web API, cloud integration becomes a viable alternative. Devices push data to a cloud platform using lightweight protocols such as MQTT or HTTPS, and the cloud platform exposes this data through REST APIs to client applications:



For Example, a battery-powered ESP32 board collects environmental data and periodically sends it to Measurify (<https://measurify.org/>), a cloud platform designed for Things. Measurify stores the measurements and offers a RESTful API to access them. Clients can retrieve current and historical data using calls like "GET /things/id/measurements".

This strategy is particularly **suitable for large-scale deployments** where scalability, persistence, and global accessibility are priorities. It offloads the processing burden from constrained devices and allows for powerful analytics or dashboards to be built on top of the exposed APIs.

### 8.1.4 Design decision

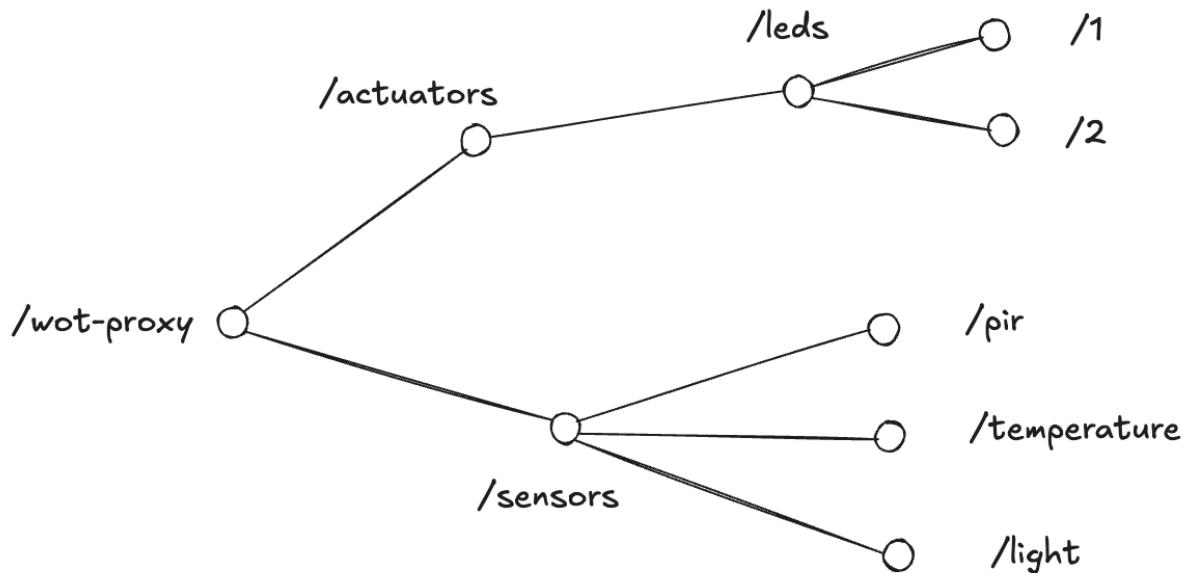
Each strategy offers a different level of abstraction and system complexity. While direct integration is optimal for small, self-contained systems, gateway and cloud approaches enable the

incorporation of highly constrained or widely distributed devices into the Web of Things ecosystem. The choice depends on the application requirements and the capabilities of the deployed hardware.

For our Web Proxy implementation, we will use the **Gateway Integration Pattern**. This allows us to abstract the complexity of the underlying devices and expose a unified Web API for clients to interact with. The gateway will handle communication with the constrained Things and provide a seamless interface for clients to access their data and functionalities.

## 8.2 Resource Design

Resource Design is the process of organizing and exposing the capabilities of a physical Thing as web-accessible REST resources. This step is foundational in bridging the physical and digital realms, allowing sensors and actuators embedded in a device to be accessed through structured URLs. It begins by identifying the physical elements of the system (such as LEDs, temperature sensors, or motion detectors) and mapping each of them into the corresponding logical entities under a coherent URI hierarchy. In the case of our WoT Proxy, we will focus on a simple architecture that includes sensors and actuators:



The entry point of the REST API is typically a **root URL**, such as "http://wot-proxy", which serves as the base path for all resources. From there, the API branches into two main categories: **sensors** and **actuators**. This categorization reflects the distinct roles that physical components play in an IoT system: sensors collect data from the environment, while actuators modify it.

Under the "/sensors" path, different sensing components are exposed as sub-resources. For

example, a passive infrared sensor (PIR) might be available at "/sensors/pir", a temperature sensor at "/sensors/temperature", and a light sensor at "/sensors/light". These endpoints return data (typically in JSON format) that represents the current state of each physical sensor.

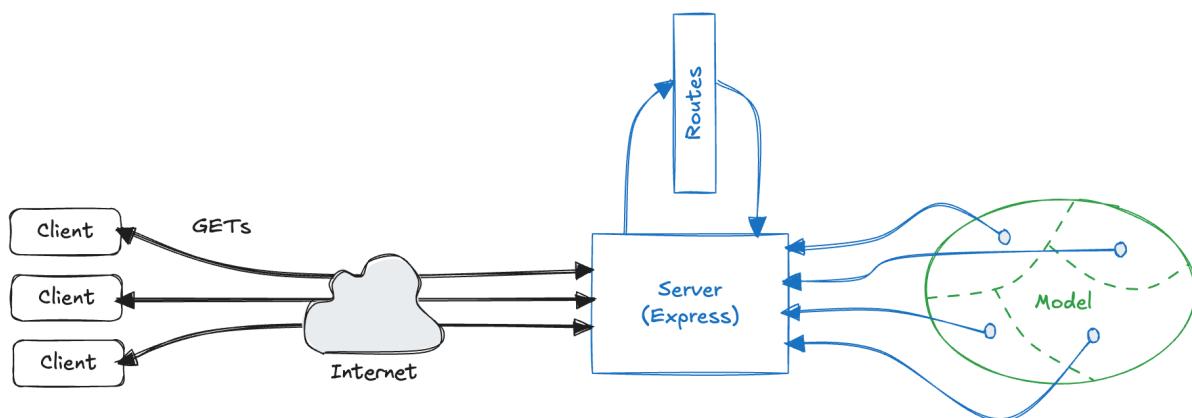
Similarly, the "/actuators" path groups controllable elements, such as LEDs, which are exposed under "/actuators/leds". If multiple instances of the same actuator exist, they can be identified using additional paths, such as "/actuators/leds/1" and "/actuators/leds/2".

The design benefits from a clean and hierarchical structure, which enhances **discoverability** and **navigability** of the API. Moreover, it **reflects the physical layout** of the Thing in a logical, web-friendly way. This structure allows developers and client applications to interact with complex embedded systems using simple and intuitive web requests.

In summary, Resource Design transforms a physical device components into a well-organized tree of web resources. Each element becomes addressable, manipulable, and observable through standard web protocols, thereby enabling true interoperability in the Web of Things.

### 8.3 Simple Thing

We can start the implementation of our WoT Proxy. This project structure should be organized in a **modular** and **clean** way according to the core responsibilities of the system. The structure has to separate the configuration of the resources, the routing, and server logic, which is a best practice for maintainability and scalability:



This architecture enables future extensions, such as adding new resources or supporting additional representation formats, with minimal restructuring (see 01.Simple Thing code):

### 8.3.1 Entry point

The **wot-server.js** file is the main entry point of the proxy. It initializes the server, loads a resource model (that we need to design), and ties together all components into a running system. It is the central coordination script for the application:

```
// wot-server.js

// Load the http server and the model
const httpServer = require('./servers/http');
const resources = require('./resources/model');

// Start the HTTP server by invoking listen() on the Express application
const server = httpServer.listen(resources.iot.port, function () {
    // Once the server is started the callback is invoked
    console.info('Your WoT API is up and running on port %s', resources.iot.port);
});
```

### 8.3.2 Servers

The **servers/** folder contains the server code, which is responsible for handling incoming requests and routing them to the appropriate handlers. It also manages the server's lifecycle, including starting and stopping the server. At the moment, we have only a single HTTP server, but we can add more servers in the future if needed. In particular, the **http.js** file wraps an HTTP server using the Express framework:

```
// /servers/http.js

// Requires the Express framework, your routes, and the model
const express = require('express');
const actuatorsRoutes = require('../routes/actuators');
const sensorRoutes = require('../routes/sensors');

// Creates an application with the Express framework
// this wraps an HTTP server
const app = express();

// Binds your routes to the Express application
// bind them to /pi/actuators/ ... and /pi/sensors/ ...
app.use('/iot/actuators', actuatorsRoutes);
```

```

app.use('/iot/sensors', sensorRoutes);

// Create a default route for /pi
app.get('/iot', function (req, res) {
  res.send('This is the WoT API!')
});

// We export router to make it accessible for "requirers" of this file
module.exports = app;

```

### 8.3.3 Resources

The **resources/** folder defines the resource model of the system. The configuration file **resources.json** declares the hierarchy and identifiers of the available sensors and actuators. This file functions as a digital twin of the physical setup, describing the device tree in a structured format. It plays a crucial role in separating the **hardware logic** (i.e., interaction with real sensors) from the **web interface logic** (i.e., how sensor values are exposed via HTTP routes):

```
{
  "iot": {
    "name": "MyWoT",
    "description": "A simple WoT API",
    "port": 8484,
    "sensors": {
      "temperature": {
        "name": "Temperature Sensor",
        "description": "An ambient temperature sensor.",
        "unit": "celsius",
        "value": 0
      },
      "light": {
        "name": "Light Sensor",
        "description": "An ambient light sensor.",
        "unit": "%",
        "value": 0
      },
      "pir": {
        "name": "Passive Infrared",
        "description": "A passive infrared sensor. When 'true' someone is present.",
        "value": true
      }
    }
}
```

```
        },
        "actuators": {
            "leds": {
                "1": {
                    "name": "LED 1",
                    "value": false
                },
                "2": {
                    "name": "LED 2",
                    "value": false
                }
            }
        }
    }
}
```

The **model.js** module loads and manages the configuration defined in resources.json. It may offer helper functions to access and update the state of individual resources in memory:

```
// /resources/model.js

// We load the entire data model from other files (here only resources.json)
const resources = require('./resources.json');

// We export the resources to make it accessible for "requirers" of this file
module.exports = resources;
```

The model serves as an **in-memory representation** of the state of the Thing. It is loaded into the application via `model.js`, which provides functions to read or update values programmatically. We need to write code to interact with the real hardware, updating the model when a new value is read. At the same time, route handlers \*\* do not interact with the hardware directly. **Instead, they simply read from or write to the model. This design ensures that** real-world hardware is abstracted away from the web layer\*\*. In this way we expose a consistent and predictable API, regardless of whether the device is running in simulation mode or connected to real sensors.

### 8.3.4 Routes

The **routes/** folder contains the **route definitions** for the Express.js application:

- `sensors.js` handles the routes under `/sensors`.

- `actuators.js` manages the HTTP endpoints under `/actuators`.

These modules define the URL structure and link the external API to internal logic, following the REST design principles.

```
// /routes/sensors.js

const express = require('express');

// We require and instantiate an Express Router to define the path to our resources
const router = express.Router();

// We require the model
const resources = require('../..//resources/model');

// Create a new route for a GET request on all sensors and attach a callback function
router.route('/').get(function (req, res, next) {
    // Reply with the sensor model when this route is selected
    res.send(resources.iot.sensors);
});

// This route serves the passive infrared sensor
router.route('/pir').get(function (req, res, next) {
    res.send(resources.iot.sensors.pir);
});

// This routes serve the temperature sensor
router.route('/temperature').get(function (req, res, next) {
    res.send(resources.iot.sensors.temperature);
});

// This routes serve the light sensor
router.route('/light').get(function (req, res, next) {
    res.send(resources.iot.sensors.light);
});

// We export router to make it accessible for "requirers" of this file
module.exports = router;

// /routes/actuators.js

const express = require('express');
```

```
// We require and instantiate an Express Router to define the path to our resources
const router = express.Router();

// We require the model
const resources = require('../..../resources/model');

// Create a new route for a GET request
router.route('/').get(function (req, res, next) {
    // Reply with the actuators model when this route is selected
    res.send(resources.iot.actuators);
});

// This route serves a list of LEDs
router.route('/leds').get(function (req, res, next) {
    res.send(resources.iot.actuators.leds);
});

// With :id we inject a variable in the path which will be the LED number
router.route('/leds/:id').get(function (req, res, next) {
    // The path variables are accessible via req.params.id
    // we use this to select the right object in our model and return it
    res.send(resources.iot.actuators.leds[req.params.id]);
});

// We export router to make it accessible for "requirers" of this file
module.exports = router;
```

### 8.3.5 Benefits

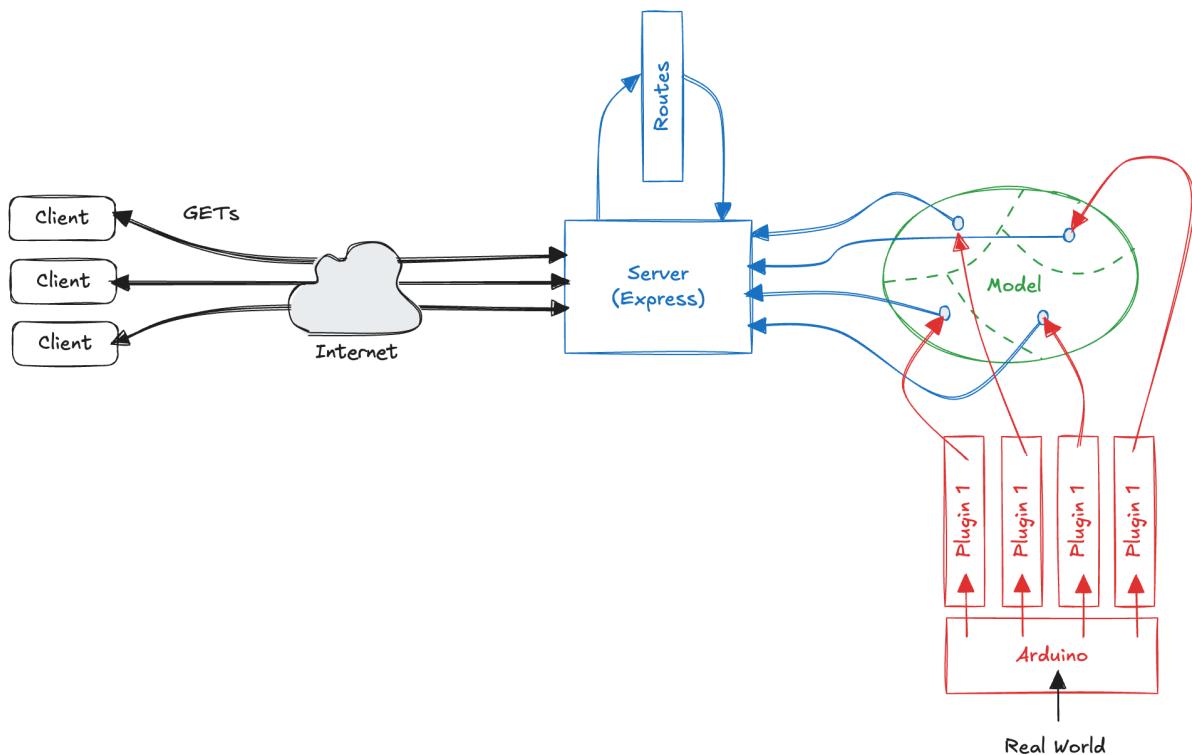
This architectural approach offers several key advantages that contribute to the robustness and flexibility of the WoT Proxy:

- **Separation of concerns:** Hardware drivers and web routes are logically and physically independent, allowing each to evolve without interfering with the other.
- **Simulation support:** Simulated sensors can update the model exactly as real sensors would, enabling development and testing even in the absence of physical hardware.
- **Maintainability:** Changes made to the hardware layer (e.g., swapping or reconfiguring sensors) do not require modifications to the routing logic, and vice versa.
- **Testability:** Since the model operates as a shared state, it can be programmatically tested and manipulated without needing to access real devices.

In summary, "resources.json" acts as a **digital twin** of the physical device, serving as the central point of integration between real-time sensor data and web-based interaction in the WoT Proxy architecture.

## 8.4 Bind the sensors

At this stage, the WoT Proxy architecture successfully exposes the digital model of sensors and actuators via RESTful routes. However, what is still missing is the **connection to the real world**: the actual sensor values remain static as defined in resources.json. Binding the sensors means **making the resource model dynamic** by linking it to physical or simulated hardware, so that the values returned by the API reflect the current state of the real environment. This is accomplished by implementing and integrating **sensor plugins**. Each plugin is responsible for reading data from one specific sensor (e.g., temperature, light, or PIR) and updating the shared resource model accordingly (see 02.Bind sensors code):



### 8.4.1 Plugin Structure

A plugin is a standalone module (e.g., /plugins/tempPlugin.js) that **encapsulates the logic required to read from the sensor and update the corresponding field in the model**. To ensure

flexibility and reusability, each plugin follows a common structure and exposes at least the following functions:

- `start()`: activates the plugin, starting the loop or callback that fetches sensor values and writes them into the model.
- `stop()`: stops the sensor reading, useful for cleanup or shutdown.
- `connectHardware()`: initializes communication with the physical device and configures it.
- `simulate()`: provides simulated values, useful for development without access to real hardware.

Plugins are typically stored in the `plugins/` folder, with separate files for each sensor or actuator. For example, we can consider the temperature sensor plugin (`tempPlugin.js`):

```
// /plugins/temperature.js

const resources = require('../resources/model');

let interval;
const model = resources.iot.sensors.temperature;
const pluginName = resources.iot.sensors.temperature.name;
const unit = resources.iot.sensors.temperature.unit;
let localParams = {'simulate': true, 'frequency': 2000};

// Starts the plugin, should be accessible from other
// files so we export them
exports.start = function (params) {
    localParams = params;
    if (localParams.simulate) { simulate(); }
    else { connectHardware(); }
};

// Stop the plugin, should be accessible from other
// files so we export them
exports.stop = function () {
    clearInterval(interval);
    console.info(`%s plugin stopped!`, pluginName);
};

// Require and connect the actual hardware driver and configure it
function connectHardware() {
    let arduino = require('../hardware/arduino');
```

```

interval = setInterval(function () { model.value = arduino.temperature; },
  ↵ localParams.frequency);
  console.info('Hardware %s sensor started!', pluginName);
};

// Allows the plugin to be in simulation mode. This is very useful when developing
// or when you want to test your code on a device with no sensors connected, such as
  ↵ your laptop
function simulate() {
  interval = setInterval(function () { model.value += 1; showValue(); },
  ↵ localParams.frequency);
  console.info('Simulated %s sensor started!', pluginName);
};

function showValue() { console.info('%s value = %s %s', pluginName, model.value,
  ↵ unit); };

```

Other files in the **plugins/** folder are similar, each implementing the specific logic for their respective sensors or actuators. The plugins are loaded and started in the main entry point (**wot-server.js**) when the server is initialized:

```

// wot-server.js

// Load the http server and the model
const httpServer = require('./servers/http');
const resources = require('./resources/model');

// Require all the sensor plugins we need
const ledsPlugin = require('./plugins/ledsPlugin');
const pirPlugin = require('./plugins/pirPlugin');
const tempPlugin = require('./plugins/tempPlugin');
const lightPlugin = require('./plugins/lightPlugin');

// Start them with a parameter object. Here we start them on a
// laptop so we activate the simulation function
ledsPlugin.start({'simulate': true, 'frequency': 2000});
pirPlugin.start({'simulate': true, 'frequency': 1000});
tempPlugin.start({'simulate': true, 'frequency': 1000});
lightPlugin.start({'simulate': true, 'frequency': 1000});

// Start the HTTP server by invoking listen() on the Express application
const server = httpServer.listen(resources.iot.port, function () {

```

```
// Once the server is started the callback is invoked
console.info('Your WoT API is up and running on port %s', resources.iot.port);
});
```

Notice that the start() function supports **two operational modes**, depending on whether the plugin should interact with **real hardware** or operate in **simulation**. This dual-mode architecture is **essential for development, testing, and deployment** in diverse environments.

#### 8.4.2 Real Hardware

In real-hardware operation mode, the plugin needs to connect to the physical device and start reading values. This is done **separating the plugin logic from the hardware-specific code**. The plugin itself does not need to know how to connect to the hardware; it simply calls the connectHardware() function, which is implemented in a separate module. This module handles the actual connection and data retrieval from the sensor.

Under the **/hardware** folder we have the code implements a complete **hardware-software bridge** between an Arduino board (acting as a real-world sensor and actuator interface) and the proxy server. This part can be adapted to any other hardware platform, such as Raspberry Pi, ESP32 without affecting the rest of the code.

In particular, the **hardware.ino** code is an Arduino sketch that runs on the board. It handles real sensor data acquisition. It continuously reads values from a temperature sensor via I2C, a passive infrared (PIR) motion detector via digital input, and a light sensor via analog input. Every second, the Arduino sends a structured data string over the serial interface in the format:

```
H;<temperature>;<motion>;<light>;
```

```
// hardware.ino

#include <Wire.h>

const char HEADER = 'H';
int tmp102Address = 0x48;
int led1Pin = 6;
int led2Pin = 7;
int pirPin = 2;
int lightPin = 0;
```

```
void setup() {
    Serial.begin(9600);

    Wire.begin();

    pinMode(led1Pin, OUTPUT);
    pinMode(led2Pin, OUTPUT);
    pinMode(pirPin, INPUT);
}

void loop() {
    float temperature = getTemperature();
    int motion = getMovement();
    float light = getLight();

    Serial.print(H HEADER);
    Serial.print(";");
    Serial.print(temperature);
    Serial.print(";");
    Serial.print(motion);
    Serial.print(";");
    Serial.print(light);
    Serial.println(";");
}

delay(1000);
}

float getLight() {
    return analogRead(lightPin);
}

int getMovement() {
    int pirVal = digitalRead(pirPin);
    if(pirVal == LOW) return 1;
    else return 0;
}

float getTemperature(){
    Wire.requestFrom(tmp102Address, 2);
    byte MSB = Wire.read();
    byte LSB = Wire.read();
    int TemperatureSum = ((MSB << 8) | LSB) >> 4;
    float celsius = TemperatureSum*0.0625;
```

```
    return celsius;
}
```

On the Node.js side, the **arduino.js** module uses the serialport to **read and parse incoming serial data**. It maintains a shared values object containing the most recent sensor values, extracted from each incoming line sent by the Arduino. The module exports this object, making it accessible to other parts of the application such as sensor plugins, which can update the RESTful resource model accordingly:

```
// arduino.js

const serialport = require('serialport');
const readline = require('@serialport/parser-readline')
const portName = '/dev/cu.usbmodem1421';

const values = { temperature:"0",
                light: "0",
                movement: "0"
            }

const sp = new serialport(portName, {
    baudRate: 9600,
    dataBits: 8,
    parity: 'none',
    stopBits: 1,
    flowControl: false,
});

const parser = new readline();
sp.pipe(parser);

parser.on('data', function(input) {
    var res = input.split(":");

    if(res[0] == 'H'){
        values.temperature = res[1];
        values.movement = res[2];
        values.light = res[3];
    }
});

module.exports = values;
```

This approach allows sensor data to flow from the Arduino into the WoT Proxy without the server needing to poll or query for updates. Instead, it simply uses the latest values stored in memory.

The overall architecture is clean and modular. The Arduino handles low-level sensing and control, while the Node.js application exposes those values over the web through a structured API. **This design separates concerns effectively**, ensuring that the physical device logic and the web logic remain independent and easy to maintain.

## 8.5 Representation Design

In a RESTful architecture, the same resource can be represented in different formats depending on the client's needs or capabilities. REST itself is **agnostic** to the specific data format used in the communication between client and server. This principle enables a Web of Things system to support **multiple representations** of the same resource, enhancing interoperability.

### 8.5.1 Supported formats

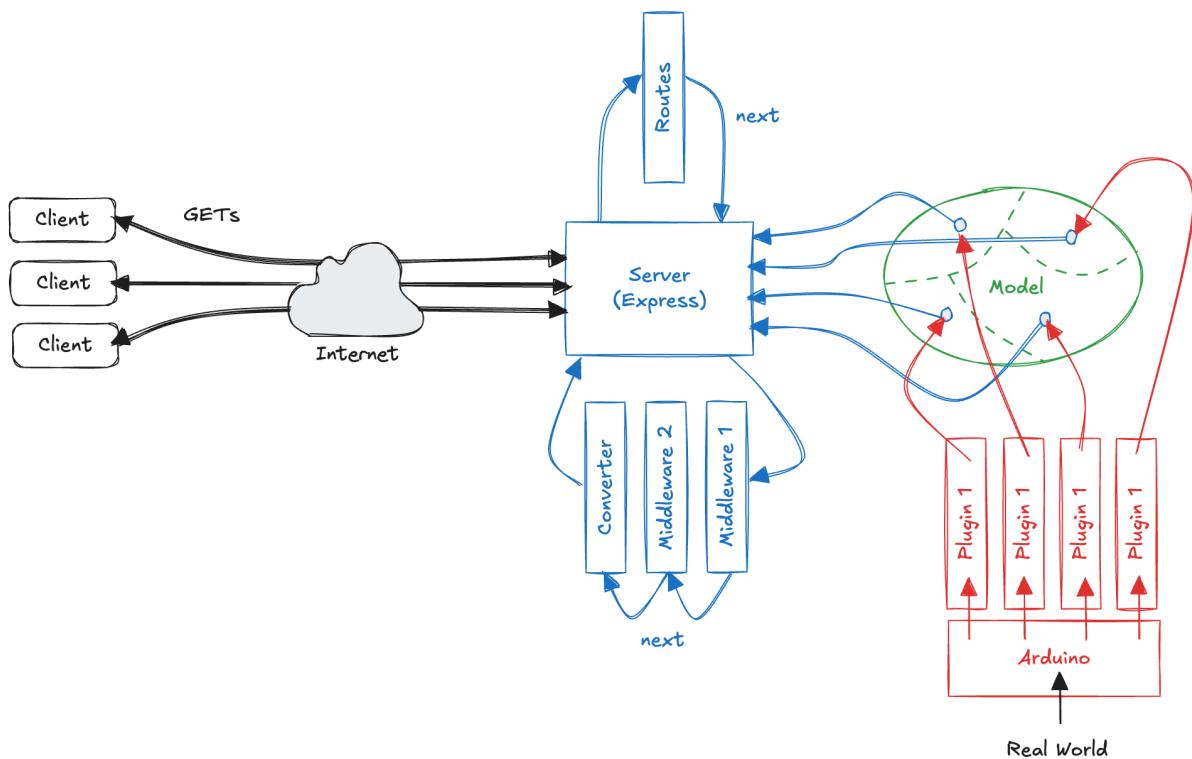
The default and most widely used representation is **JSON**, which ensures compatibility across systems and platforms due to its ubiquity and simplicity. JSON is essential to guarantee that clients (from web dashboards to mobile apps) can parse and use the data without requiring additional tooling. However, JSON is not the only useful representation.

Adding support for **HTML** representation allows human users to navigate and explore the API through a browser. Instead of seeing a raw JSON object, they can interact with a structured and readable HTML view. This **improves developer experience** and facilitates testing and debugging. For this purpose, a transformation library like "json2html" can be used to convert the model's JSON objects into an HTML representation dynamically, without hardcoding HTML templates.

In more constrained environments (such as embedded devices with limited memory or bandwidth) **MessagePack** becomes a valuable alternative. It is a **binary serialization format** that maps cleanly to JSON but produces more compact payloads. This efficiency makes it especially suited for low-power IoT scenarios, where network usage needs to be minimized. On the server side, the "msgpack5" library can be used to support this representation seamlessly.

### 8.5.2 Middleware implementation

To handle these multiple formats in a clean and extensible way, the server architecture relies on the **middleware pattern**. Middleware functions in Express are chained components that process the incoming request and outgoing response objects. Each middleware has access to the full context of the request and can either handle the response or pass control to the next function using "next()". In the context of representation design, a custom middleware can be introduced to examine the "Accept" header of the request and select the appropriate representation (e.g., "application/json", "text/html", or "application/x-msgpack"). The selected format determines how the resource is encoded before it is sent in the response:



This approach keeps route handlers focused solely on retrieving the right resource, while **the responsibility for formatting the response is entirely delegated to the middleware layer**. This design adheres to the principle of separation of concerns and improves the maintainability and scalability of the codebase.

To implement representation design in practice (see 03.Multiple Representations code), we install the required libraries via NPM ("node-json2html" package to generate readable HTML from JSON and "msgpack5" to encode into a compact binary format). A custom middleware is then implemented in **middleware/converter.js**:

```
// /middlewares/converter.js

// Require the two modules and instantiate a MessagePack encoder
const msgpack = require('msgpack5')();
const json2html = require('node-json2html');

const encode = msgpack.encode;

// In Express, a middleware is usually a function returning a function
module.exports = function() {
  return function (req, res, next) {
    console.info('Representation converter middleware called!');

    // Check if the previous middleware left a result in req.result
    if (req.result) {

      // Read the request header and check if the client requested HTML
      if (req.accepts('html')) {
        console.info('HTML representation selected!');
        // If HTML was requested, use json2html to transform the JSON into simple HTML
        const transform = {'tag': 'div', 'html': '${name} : ${value}'};
        res.send(json2html.transform(req.result, transform));
        return;
      }

      // Read the request header and check if the client requested MessagePack
      if (req.accepts('application/x-msgpack')) {
        console.info('MessagePack representation selected!');
        // Encode the JSON into MessagePack using the encoder
        res.type('application/x-msgpack');
        res.send(encode(req.result));
        return;
      }

      // For all other formats, default to JSON
      console.info('Defaulting to JSON representation!');
      res.send(req.result);
      return;
    }

    // If no result was present in req.result, there's not much you can do, so call
    // the next middleware
  }
}
```

```
    else { next(); }
}
};
```

This module inspects each request headers to determine the desired format. It accesses the resource to be returned via a shared property, typically assigned to "req.result", and then serializes it into the appropriate format before sending the response.

To activate this middleware, existing routes (sensors and actuators) must be slightly refactored. Instead of sending the response directly using "res.send(resource)", the resource is assigned to "req.result", and the route handler ends with a call to "next()". This signals Express to pass control to the next middleware, in this case the representation converter:

```
// /routes/sensors.js

const express = require('express');

const router = express.Router();
const resources = require('../resources/model');

router.route('/').get(function (req, res, next) {
  req.result = resources.iot.sensors;
  next();
});

// This route serves the passive infrared sensor
router.route('/pir').get(function (req, res, next) {
  req.result = resources.iot.sensors.pir;
  next();
});

// This routes serve the temperature sensor
router.route('/temperature').get(function (req, res, next) {
  req.result = resources.iot.sensors.temperature;
  next();
});

// This routes serve the light sensor
router.route('/light').get(function (req, res, next) {
  req.result = resources.iot.sensors.light;
  next();
});
```

```
module.exports = router;
```

Finally, the middleware must be registered in the main HTTP server file using "app.use()" to add it into the middleware chain. It should be included **after the route definitions** so it processes only finalized req.result objects:

```
// /servers/http.js

const express = require('express');
const actuatorsRoutes = require('../routes/actuators');
const sensorRoutes = require('../routes/sensors');
const resources = require('../resources/model');

// Requires the converter
const converter = require('../middleware/converter');

// Creates an application with the Express framework
// this wraps an HTTP server
const app = express();

// Binds your routes to the Express application
// bind them to /pi/actuators/ ... and /pi/sensors/ ...
app.use('/iot/actuators', actuatorsRoutes);
app.use('/iot/sensors', sensorRoutes);

// Create a default route for /iot
app.get('/iot', function (req, res) {
  res.send('This is the WoT API!')
});

// Add the converter to the chain
// As the converter middleware responds to the client
// make sure you add it last, after app.get('iot') or it
// will simply bypass any other middleware!
app.use(converter());

module.exports = app;
```

Once this setup is complete, the server becomes capable of responding in three different representations: JSON (application/json), MessagePack (application/x-msgpack), and HTML (text/html). **This enhances both machine-to-machine and human-friendly interactions**, while

also supporting **bandwidth-sensitive applications**. The design is modular and extensible, allowing future representations to be added without modifying the route logic.

Supporting multiple representations makes the WoT Proxy more versatile. Clients with different capabilities and constraints can access the same underlying data through a common API, tailored to their specific context. Whether it's a browser, a low-power sensor node, or a mobile app, each can request the most appropriate format without changing the application logic.

## 8.6 Interface Design

The interface design of the system focuses on enabling interaction with connected devices through the API, with a gradual evolution from simple data retrieval to **full actuator control**. The initial version of the system supported only basic **GET** operations to retrieve sensor data. While functional, this limited the interaction to read-only operations. To support more dynamic applications, including device control and real-time updates, additional HTTP verbs and mechanisms have been introduced.

### 8.6.1 Adding PUT Support

To enable data submission and state modification, the system should be extended to support the **PUT** method. This capability allows clients to send data to the server in a structured format which can be used for tasks such as configuring system parameters or triggering events (see 04.POST support code). In particular, we add the PUT method implementation to enable external clients to update the state of a specific LED actuator by sending a JSON payload with a "value" field:

```
// /routes/actuators.js
...
// Callback for a PUT request on an LED
router.route('/leds/:id').put(function(req, res, next) {
  var selectedLed = resources.iot.actuators.leds[req.params.id];

  // Update the value of the selected LED in the model
  if(req.body.value == undefined) {
    req.result = { error: "missing value" };
  }
  else {
    selectedLed.value = req.body.value;
    req.result = selectedLed;
```

```
    }
    next();
});
...
...
```

However, updating the model alone is not sufficient, this change must also be **reflected in the physical world**. One option would be to **directly invoke the plugin** to modify the actuator's value. However, this would make the plugin code **dependent** on the structure of the model, **violating the principle of separation of concerns**. Instead, we should implement a **notification mechanism** that alerts the plugins whenever the model is updated. This way, plugins can autonomously react and update the hardware, without the server code needing to manage how this is done. We can implement this notification mechanism using the **Observer pattern**. The model will maintain a list of observers (in this case, the plugins) and notify them whenever a change occurs. Each plugin will then decide how to handle the update based on its own logic.

### 8.6.2 Observer Pattern

The Observer Pattern is a design pattern in which an object (the **subject**) maintains a list of dependents (called **observers**) and notifies them automatically of any state changes. This is useful when multiple components need to stay in sync **without tightly coupling** them. In Node, the node-observer package provides a simple way to implement this pattern.

When the API updates the state of an actuator (e.g., turns an LED on or off), we want the hardware-related plugin to be notified automatically and react accordingly, **without embedding hardware logic into the web-related code**.

We need to subscribe the plugin to listen for state changes of the model:

```
// /plugins/ledsPlugin.js

const resources = require('../resources/model');
const observer = require("node-observer");

const model1 = resources.iot.actuators.leds['1'];
const model2 = resources.iot.actuators.leds['2'];

let interval;
const pluginName = "Leds"
let localParams = {'simulate': false, 'frequency': 2000};
```

```
exports.start = function (params) {
    localParams = params;

    // Observe the model for the LEDs
    observe(model1);
    observe(model2);

    if (localParams.simulate) { simulate(); }
    else { connectHardware(); }
};

exports.stop = function () {
    clearInterval(interval);
    console.info('%s plugin stopped!', pluginName);
};

function observe(what) {
    observer.subscribe(this, what.name, function(who, data) { switchOnOff(what.name,
        data); });
};

function switchOnOff(name, value) {
    console.info('Change detected by plugin for %s = %s', name, value);
    if (!localParams.simulate) {
        const arduino = require('../hardware/arduino');
        arduino.send(name, value);
        console.info('Changed value of %s to %s', name, value);
    }
};

function connectHardware() { console.info('Hardware %s actuator started!', pluginName); }

function simulate() {
    interval = setInterval(function () {
        // Switch value on a regular basis
        if (model1.value) { model1.value = false; }
        else { model1.value = true; }
    }, localParams.frequency);
    console.info('Simulated %s actuator started!', pluginName);
};
```

Then we need to emit an update when the model changes:

```
// /routes/actuators.js

const express = require('express');
const observer = require("node-observer");

const router = express.Router();
const resources = require('../resources/model');

router.route('/').get(function (req, res, next) {
  req.result = resources.iot.sensors;
  next();
});

router.route('/leds').get(function (req, res, next) {
  req.result = resources.iot.actuators.leds;
  next();
});

router.route('/leds/:pippo').get(function (req, res, next) {
  req.result = resources.iot.actuators.leds[req.params.id];
  next();
});

// Callback for a PUT request on an LED
router.route('/leds/:id').put(function(req, res, next) {
  var selectedLed = resources.iot.actuators.leds[req.params.id];
  // Update the value of the selected LED in the model
  if(req.body.value == undefined) {
    req.result = { error: "missing value" };
  }
  else {
    selectedLed.value = req.body.value;
    req.result = selectedLed;

    // Send information to observers
    observer.send(this, selectedLed.name, req.body.value);
  }
  next();
});

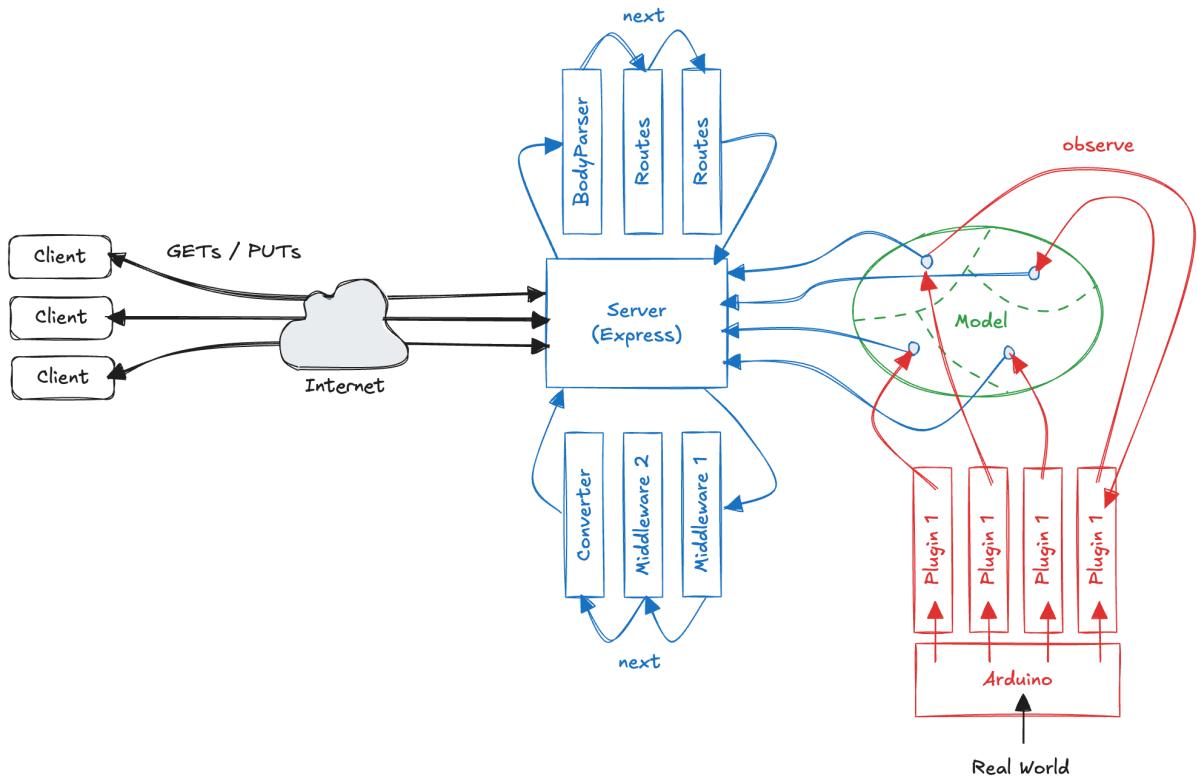
module.exports = router;
```

### 8.6.3 Body Parsing Middleware

To fully support verbs like PUT and POST, we need to handle the payload of incoming HTTP requests. The WoT server supports JSON payloads for actuator control. For example, the state of an LED can be updated by sending the following structure:

```
{
  "value": true
}
```

A **body parser middleware** can be introduced:



This middleware intercepts the HTTP request at the beginning of the request lifecycle to extract and parse the request body, typically in JSON format:

```
const express = require('express');
const actuatorsRoutes = require('../routes/actuators');
const sensorRoutes = require('../routes/sensors');
const resources = require('../resources/model');
const converter = require('../middleware/converter');
```

```
// Requires the body parser
const bodyParser = require('body-parser');

const app = express();

// Add the bodyParser to the chain
// As the bodyParser middleware gets information from
// the request useful for other middleware make sure you add it first
app.use(bodyParser.json());

app.use('/iot/actuators', actuatorsRoutes);
app.use('/iot/sensors', sensorRoutes);

app.get('/iot', function (req, res) {
  res.send('This is the WoT API!')
});

app.use(converter());

module.exports = app;
```

#### 8.6.4 Bidirectional Serial Communication

Finally we need to update the Arduino code in order to support **bidirectional interaction**: while the device continues to send sensor readings to the host, it now also listens for incoming characters via the serial port and uses them to control the state of two LEDs, enabling remote actuation directly from the server:

```
#include <Wire.h>

const char HEADER = 'H';
int tmp102Address = 0x48;
int led1Pin = 6;
int led2Pin = 7;
int pirPin = 2;
int lightPin = 0;

void setup() {
  Serial.begin(9600);
```

```
Wire.begin();

pinMode(led1Pin, OUTPUT);
pinMode(led2Pin, OUTPUT);
pinMode(pirPin, INPUT);
}

void loop() {
    float temperature = getTemperature();
    int motion = getMovement();
    float light = getLight();

    Serial.print(H HEADER);
    Serial.print(";");
    Serial.print(temperature);
    Serial.print(";");
    Serial.print(motion);
    Serial.print(";");
    Serial.print(light);
    Serial.println(";");
}

if (Serial.available() ) {
    char ch = Serial.read();
    if(ch == '0') setLed(1,0);
    else if(ch == '1') setLed(1,1);
    else if(ch == '2') setLed(2,0);
    else if(ch == '3') setLed(2,1);
}

delay(1000);
}

void setLed(int led, int value) {
    if(led == 1 && value == 1)
        digitalWrite(led1Pin,HIGH);

    if(led == 1 && value == 0)
        digitalWrite(led1Pin,LOW);

    if(led == 2 && value == 1)
        digitalWrite(led2Pin,HIGH);

    if(led == 2 && value == 0)
```

```

    digitalWrite(led2Pin,LOW);
}

float getLight() {
    return analogRead(lightPin);
}

int getMovement() {
    int pirVal = digitalRead(pirPin);
    if(pirVal == LOW) return 1;
    else return 0;
}

float getTemperature() {
    Wire.requestFrom(tmp102Address,2);
    byte MSB = Wire.read();
    byte LSB = Wire.read();
    int TemperatureSum = ((MSB << 8) | LSB) >> 4;
    float celsius = TemperatureSum*0.0625;
    return celsius;
}

```

The arduino.js has to expose the possibility to send commands to the Arduino board. We snippet introduce a **send function**, which is used to transmit data from the application to the connected device. This function ensures proper formatting and delivery of messages, enabling seamless communication between components of the system:

```

const serialport = require('serialport');
const readline = require('@serialport/parser-readline')
const portName = '/dev/cu.usbmodem1421';

const values = { temperature:"0",
                light: "0",
                movement: "0",
                send: function(led, data){
                    if(led=="LED 1" && data==false) sp.write("0");
                    if(led=="LED 1" && data==true) sp.write("1");
                    if(led=="LED 2" && data==false) sp.write("2");
                    if(led=="LED 2" && data==true) sp.write("3");
                }
}

```

```
const sp = new serialport(portName, {
  baudRate: 9600,
  dataBits: 8,
  parity: 'none',
  stopBits: 1,
  flowControl: false,
});

const parser = new readline();
sp.pipe(parser);

parser.on('data', function(input) {
  var res = input.split(":");

  if(res[0] == 'H') {
    values.temperature = res[1];
    values.movement = res[2];
    values.light = res[3];
  }
});

module.exports = values;
```

## 8.7 Real-time data

The current implementation of the WoT Proxy allows clients to retrieve sensor data and control actuators through a RESTful API. However, this model is inherently **polling-based**, meaning that clients must periodically query the server to obtain updated information. This approach can lead to inefficiencies, increased latency, and unnecessary network traffic. To address these limitations, we can introduce **real-time data updates** using WebSockets. This technology enables a persistent connection between the client and server, allowing for **bidirectional communication**. Clients can subscribe to specific events or resources and receive updates as soon as they occur, without needing to poll the server continuously.

### 8.7.1 WebSocket Server

The WebSocket server is implemented using the lightweight ws library, chosen for its simplicity and efficiency. A dedicated listener is defined in the `websockets.js` module, which attaches a WebSocket server to the existing Express HTTP server (see 05.WebSocket code):

```
// /servers/websocket.js

const WebSocketServer = require('ws').Server;
const resources = require('../resources/model');
const observer = require("node-observer");

exports.listen = function(server) {

    // Create a WebSockets server by passing it the Express server
    const wss = new WebSocketServer({server: server});
    console.info('WebSocket server started ...');

    // Triggered after a protocol upgrade when the client connected
    wss.on('connection', function (ws, req) {
        let url = req.url;
        console.info(url);

        // Register an observer corresponding to the resource in the protocol upgrade URL
        try {
            observer.subscribe(this, selectResouce(url).name, function(who, data) {
                ws.send(JSON.stringify(selectResouce(url).value), function () {});
            });
        }
    });

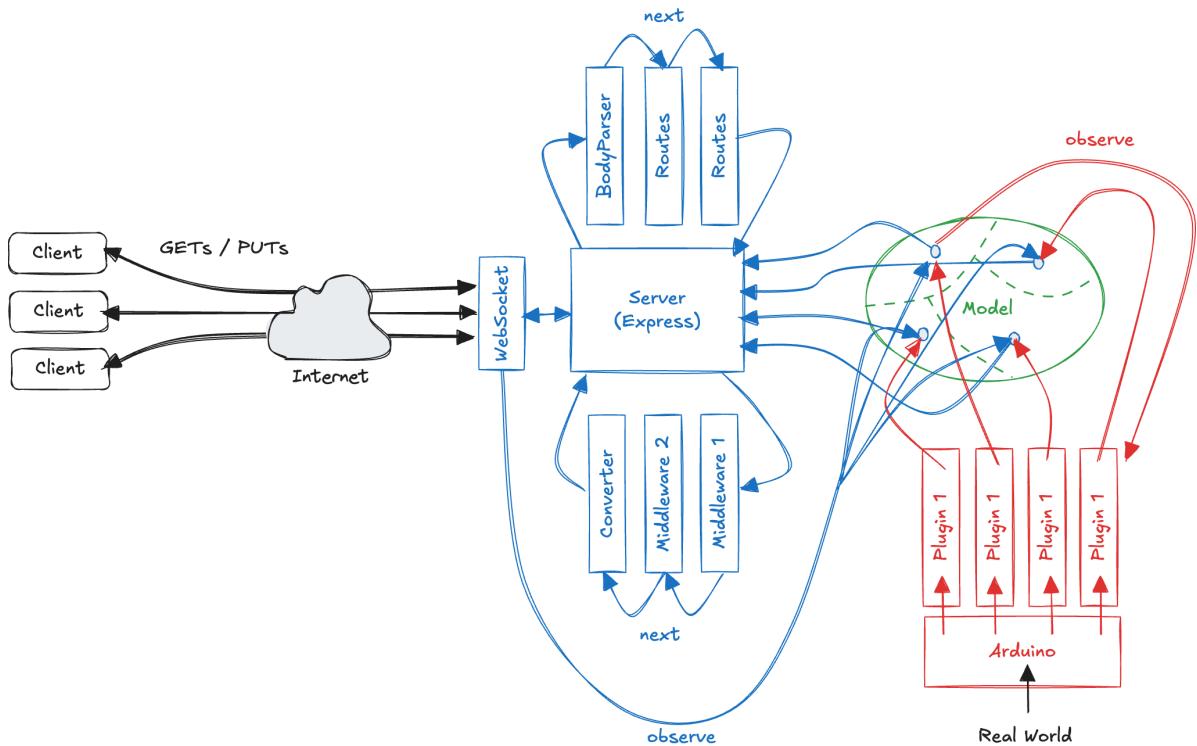
    // Intercept errors (e.g., malformed/unsupported URLs)
    catch (e) { console.log('Unable to observe %s resource!', url); console.log(e); };
});

// This function takes a request URL and returns the corresponding resource
function selectResouce(url) {
    let parts = url.split('/');
    parts.shift();
    let result = resources;
    for (let i = 0; i < parts.length; i++) { result = result[parts[i]]; }
    return result;
}
```

### 8.7.2 Observer changes

When a client initiates a WebSocket connection, the server inspects the URL path of the upgrade request to identify the target resource within the shared data model. Using the **observer pattern**, the server can dynamically subscribe to changes in the target resource and send updates to the client.

**tern**, each WebSocket connection subscribes to updates on the corresponding resource. When a change occurs (such as a new temperature reading from a sensor plugin) the WebSocket server pushes the updated value to all connected clients in real time.



Of course the sensors plugins must be updated to emit the new value whenever a change occurs. For example, the temperature plugin can be modified to emit an event whenever a new value is read from the hardware:

```
// /plugins/tempPlugin.js

const resources = require('../resources/model');
const observer = require("node-observer");

let interval;
const model = resources.iot.sensors.temperature;
const pluginName = resources.iot.sensors.temperature.name;
const unit = resources.iot.sensors.temperature.unit;
let localParams = {'simulate': true, 'frequency': 2000};

// Starts the plugin, should be accessible from other
// files so we export them
exports.start = function (params) {
```

```
localParams = params;
if (localParams.simulate) { simulate(); }
else { connectHardware(); }
};

// Stop the plugin, should be accessible from other
// files so we export them
exports.stop = function () {
    clearInterval(interval);
    console.info('%s plugin stopped!', pluginName);
};

// Require and connect the actual hardware driver and configure it
function connectHardware() {
    var arduino = require('../hardware/arduino');

    interval = setInterval(function () {
        model.value = arduino.temperature;
        observer.send(this, model.name, model.value);
    }, localParams.frequency);
    console.info('Hardware %s sensor started!', pluginName);
};

// Allows the plugin to be in simulation mode. This is very useful when developing
// or when you want to test your code on a device with no sensors connected, such as
// your laptop
function simulate() {
    interval = setInterval(function () {
        model.value += 1;
        observer.send(this, model.name, model.value);
        showValue();
    }, localParams.frequency);
    console.info('Simulated %s sensor started!', pluginName);
};

function showValue() { console.info('%s value = %s %s', pluginName, model.value,
    unit); };
```

### 8.7.3 HTTP and WebSocket integration

Finally, to integrate the WebSocket server, the main server script wot-server.js is updated to call the listen function exported from websockets.js, passing the already-running HTTP server. This

ensures that WebSocket and HTTP clients share the same port and infrastructure:

```
// Load the http server, the websocket server and the model
const httpServer = require('./servers/http');
const resources = require('./resources/model');
const wsServer = require('./servers/websockets');

// Require all the sensor plugins we need
const ledsPlugin = require('./plugins/ledsPlugin');
const pirPlugin = require('./plugins/pirPlugin');
const tempPlugin = require('./plugins/tempPlugin');
const lightPlugin = require('./plugins/lightPlugin');

// Start them with a parameter object. Here we start them on a
// laptop so we activate the simulation function
ledsPlugin.start({'simulate': true, 'frequency': 1000});
pirPlugin.start({'simulate': true, 'frequency': 1000});
tempPlugin.start({'simulate': true, 'frequency': 1000});
lightPlugin.start({'simulate': true, 'frequency': 1000});

// Start the HTTP server by invoking listen() on the Express application
const server = httpServer.listen(resources.iot.port, function () {
    console.info('Your WoT Pi is up and running on port %s', resources.iot.port);
});

// Start the WebSocket server by passing it the HTTP server
wsServer.listen(server);
```

#### 8.7.4 WebSocket Client

Clients can now open a WebSocket connection (e.g., using /iot/sensors/temperature) and receive updates without polling. A sample HTML client CN demonstrate how the client subscribes and handles incoming updates:

```
!-- /public/websocketsClient.js

<!DOCTYPE html>
<html lang="en">
    <body>
        <script>
            function subscribeToWs(url, msg) {
```

```
// Create a new WebSocket connection
var socket = new WebSocket(url);

// Define the event handlers for the WebSocket connection
socket.onmessage = function (event) { console.log(event.data); };

socket.onerror = function (error) {
    console.log('An error occurred while trying to connect to a
← Websocket!');
    console.log(error);
};

socket.onopen = function (event) { if (msg) { socket.send(msg); } };
}

// Subscribe to the WebSocket server
subscribeToWs('ws://localhost:8484/iot/sensors/temperature');

</script>
</body>
</html>
```

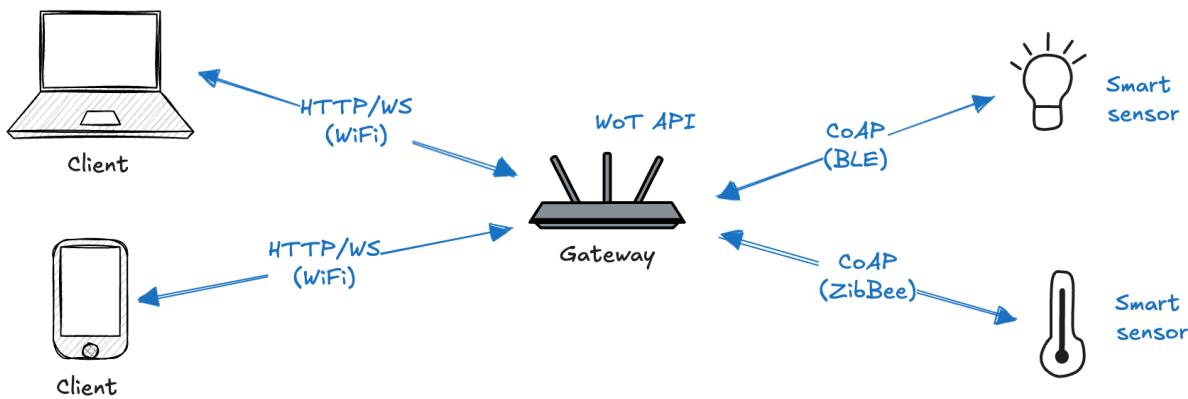
This enhancement shifts the WoT Proxy architecture from purely request-response interactions to a hybrid model supporting both synchronous queries and asynchronous notifications, making it suitable for reactive IoT applications such as live dashboards or event-driven automation.

## 8.8 Integrate other devices

The WoT Proxy currently supports a single device, but the architecture is designed to be extensible. We can easily add support for multiple devices by creating additional resource models and plugins. Each device can have its own set of sensors and actuators, and the WoT Proxy can expose them under different paths in the API. Moreover, devices can be connected to the proxy using different protocols, depending on their capabilities. This flexibility allows the WoT Proxy to act as a **universal gateway** for various devices, regardless of their underlying technology.

Many real-world battery-powered devices and resource-constrained microcontrollers cannot connect over Wi-Fi or Ethernet and instead rely on **low-power communication protocols** such as **Bluetooth Low Energy (BLE)** or **ZigBee**.

In that case, the proxy can act as a gateway, and speaks to the constrained device using a protocol it understands (e.g., **Constrained Application Protocol (CoAP)**), and translates the data and commands into the RESTful API that clients can consume using HTTP or WebSockets. This pattern provides a number of benefits. It decouples the physical limitations of the device from the requirements of web protocols, making the integration scalable and robust:



In addition to protocol translation, the proxy can perform additional roles. It can enrich security by adding authentication or encryption layers that the device cannot handle. It can buffer data, acting as temporary storage in case of intermittent connectivity, a key feature of what is often called **fog computing**. It can also expose semantic metadata for devices that lack the capability to describe themselves, enabling automatic discovery and integration.

In the WoT Proxy architecture, the gateway is responsible for managing these low-level interactions, abstracting them behind standard REST resources. From the perspective of a web client, there is no difference between interacting with a natively web-enabled Thing and one that is bridged through a gateway. The uniformity of the API and the abstraction of protocol details are what enable true interoperability across the heterogeneous landscape of IoT devices.

### 8.8.1 CoAP device

The **Constrained Application Protocol (CoAP)** is a lightweight, REST-based protocol specifically designed for resource-constrained devices and low-power communication scenarios. It operates over **UDP** instead of **TCP** and is ideally suited for device-to-device communication across low-power wireless networks such as ZigBee. Unlike HTTP, it has a **minimal footprint**, making it feasible to implement even on microcontrollers like the Arduino Uno. To experiment with CoAP, we implement a simple CoAP device using Node (see 06.CoAP code) to simulate a CO<sub>2</sub> sensor. The device exposes a single resource, /co2, which returns the current CO<sub>2</sub> level in parts per million (ppm). The CoAP server is implemented using the `coap` module:

```
// /devices/coap.js

// Require the Node.js CoAP module you installed
const coap = require('coap');
const port = 5683;

// Create a CoAP server and bind it to the callback function
coap.createServer(function (req, res) {

    console.info('CoAP device got a request for %s', req.url);

    // We only serve JSON, so we reply with a 4.06 (= HTTP 406: Not acceptable)
    if (req.headers['Accept'] != 'application/json') {
        res.code = '4.06';
        return res.end();
    }

    // Handle the different resources requested
    switch (req.url) {
        // This is a simulated CO2 resource; generate a random value for it and respond
        case "/co2":
            respond(res, {'co2': Math.floor(Math.random() * 1000)});
            break;
        default:
            respond(res);
    }
}).listen(port, function () {
    // Start the CoAP server on port 5683 (CoAP's default port)
    console.log("CoAP server started on port %s", port)
});

// Send the JSON content back or reply with a 4.04 (= HTTP 404: Not found)
function respond(res, content) {
    if (content) {
        res.setOption('Content-Type', 'application/json');
        res.code = '2.05';
        res.end(JSON.stringify(content));
    } else {
        res.code = '4.04';
        res.end();
    }
};
```

This demonstrates how to run a CoAP server on a resource-limited device, simulating such behavior using Node.js.

### 8.8.2 Gateway

To make CoAP device accessible from standard web clients, we can proxy it through our HTTP-capable WoT Proxy, following the **Gateway Integration Pattern**. The resource model is extended in resources.json to describe the new CoAP device:

```
{  
  "iot": {  
    "name": "MyWoT",  
    "description": "A simple WoT API",  
    "port": 8484,  
    "sensors": {  
      "temperature": {  
        "name": "Temperature Sensor",  
        "description": "An ambient temperature sensor.",  
        "unit": "celsius",  
        "value": 0  
      },  
      "light": {  
        "name": "Light Sensor",  
        "description": "An ambient light sensor.",  
        "unit": "%",  
        "value": 0  
      },  
      "pir": {  
        "name": "Passive Infrared",  
        "description": "A passive infrared sensor. When 'true' someone is present.",  
        "value": true  
      },  
      "co2": {  
        "name": "Co2 Sensor",  
        "description": "An ambient CO2 sensor",  
        "unit": "ppm",  
        "value": 0  
      }  
    },  
    "actuators": {  
      "leds": {  
        "1": {  
          "name": "LED 1",  
          "description": "A red LED",  
          "unit": "none",  
          "value": false  
        }  
      }  
    }  
  }  
}
```

```
        "name": "LED 1",
        "value": false
    },
    "2": {
        "name": "LED 2",
        "value": false
    }
}
}
}
```

Then, a dedicated plugin (`coapPlugin.js`) is implemented to communicate with the CoAP endpoint and update the shared data model accordingly. This plugin typically uses the `coap` module to issue CoAP requests to the target device and parse the responses:

```
// /plugins/coapPlugin.js

const resources = require('../resources/model');

const model = resources.iot.sensors.co2;
const pluginName = resources.iot.sensors.co2.name;
const unit = resources.iot.sensors.co2.unit;

let interval, pollInterval;
let localParams = {'simulate': false, 'frequency': 5000};

function connectHardware() {

    // Require the CoAP and BL library, a Buffer helper
    const coap = require('coap');
    const bl = require('bl');

    // Create a sensor object and give it a read function
    var sensor = {
        // The read function wraps a coap over UDP request with the enclosed parameters;
        // replace localhost with the IP of the machine you're simulating the CoAP
        // device from (e.g., your laptop)
        read: function () {
            coap
                .request({
                    host: 'localhost',
```

```
    port: 5683,
    pathname: '/co2',
    options: {'Accept': 'application/json'}
  })
  // When CoAP device sends the result, the on response event is triggered
  .on('response', function (res) {
    console.info('CoAP response code', res.code);
    if (res.code != 205)
      console.log("Error while contacting CoAP service: %s", res.code);
    // Fetch the results and update the model
    res.pipe(bl(function (err, data) {
      var json = JSON.parse(data);
      model.value = json.co2;
      showValue();
    }));
  })
  .end();
}

// Poll the CoAP device for new CO2 readings on a regular basis
pollInterval = setInterval(function () {
  sensor.read();
}, localParams.frequency);
};

exports.start = function (params, app) {
  localParams = params;

  if (params.simulate) {
    simulate();
  } else {
    connectHardware();
  }
};

exports.stop = function () {
  if (params.simulate) {
    clearInterval(interval);
  } else {
    clearInterval(pollInterval);
  }
  console.info('%s plugin stopped!', pluginName);
```

```
};

function simulate() {
  interval = setInterval(function () {
    model.value = Math.floor(Math.random() * 1000);
    showValue();
  }, localParams.frequency);
  console.info('Simulated %s sensor started!', pluginName);
};

function showValue() {
  console.info('CO2 Level: %s ppm', model.value);
};
```

Next, the server's route definitions in sensors.js are updated to include endpoints that expose the CoAP device data over HTTP:

```
// /routes/sensors.js

var express = require('express');

var router = express.Router();
var resources = require('../resources/model');

router.route('/').get(function (req, res, next) {
  req.result = resources.iot.sensors;
  next();
});

router.route('/pir').get(function (req, res, next) {
  req.result = resources.iot.sensors.pir;
  next();
});

router.route('/temperature').get(function (req, res, next) {
  req.result = resources.iot.sensors.temperature;
  next();
});

router.route('/light').get(function (req, res, next) {
  req.result = resources.iot.sensors.light;
  next();
```

```
});

// This routes serve the co2 sensor
router.route('/co2').get(function (req, res, next) {
  req.result = resources.iot.sensors.co2;
  next();
});

module.exports = router;
```

Finally, the new plugin is loaded and started from the main entry point wot-server.js, just like the other hardware plugins:

```
// /wot-server.js

// Load the http server, the websocket server and the model
const httpServer = require('./servers/http');
const resources = require('./resources/model');
const wsServer = require('./servers/websockets');

// Require all the sensor plugins we need
const ledsPlugin = require('./plugins/ledsPlugin');
const pirPlugin = require('./plugins/pirPlugin');
const tempPlugin = require('./plugins/tempPlugin');
const lightPlugin = require('./plugins/lightPlugin');
const coapPlugin = require('./plugins/coapPlugin');

// Start them with a parameter object. Here we start them on a
// laptop so we activate the simulation function
ledsPlugin.start({simulate: true, 'frequency': 1000});
pirPlugin.start({simulate: true, 'frequency': 1000});
tempPlugin.start({simulate: true, 'frequency': 1000});
lightPlugin.start({simulate: true, 'frequency': 1000});
coapPlugin.start({simulate: false, 'frequency': 1000});

const server = httpServer.listen(resources.iot.port, function () {
  console.info('Your WoT Pi is up and running on port %s', resources.iot.port);
});

// Websockets server
wsServer.listen(server);
```

Once this setup is complete, the CoAP device becomes accessible through the WoT Proxy via HTTP, making it transparent to clients whether the underlying device uses HTTP, CoAP, or any other protocol. This abstraction not only simplifies client development but also enables heterogeneous devices to be integrated into the same web-based application logic, reinforcing the WoT Proxy's goal of unifying the Web of Things under a consistent API.

## 8.9 Test-Driven Development

As our applications grow, so does the need to ensure that each part works as intended. One of the most reliable strategies for writing robust and maintainable code is **Test-Driven Development**, or **TDD**. TDD is not just about catching bugs, it's a disciplined workflow that encourages better design and more confidence in your codebase. It is a methodology where tests are written **before** the code they're meant to verify. The process usually follows three simple steps:

1. **Write a test** for a specific behavior or feature.
2. **Run the test** and watch it fail (since the feature hasn't been implemented yet).
3. **Write the minimal code** necessary to make the test pass.
4. **Refactor** the code while ensuring the test still passes.
5. Repeat.

This approach helps us think more clearly about the behavior we expect from our code and ensures that each new feature or change is verifiable and isolated.

Beyond correctness, unit tests offer secondary benefits. They act as living documentation, help prevent regressions during future code changes, and promote modular, decoupled code that is easier to maintain and extend.

### 8.9.1 Using assert

Node.js comes with a built-in module called `assert`, which allows developers to perform simple tests by checking that values are equal, defined, or follow certain conditions. If an assertion fails, an exception is thrown — which signals that the test did not pass. Here's a basic example of using `assert`:

```
const assert = require('assert');

assert.strictEqual(1 + 1, 2); // Passes
assert.strictEqual('hello'.toUpperCase(), 'HELLO'); // Passes
assert.strictEqual([1, 2].length, 3); // Fails and throws an error
```

While assert is great for quick and simple testing, real-world projects benefit from more powerful testing frameworks to provide a more structured test suites, better reporting, and advanced mocking capabilities.

### 8.9.2 Unit tests

Unit testing is the practice of writing code that **validates the behavior of individual units** of functionality in a program, typically a single function or module. These tests are written by developers and are meant to verify that a specific portion of code performs as expected in a variety of scenarios. The idea is to **isolate the smallest testable parts** of an application and ensure that each performs correctly, both in normal and edge cases.

A famous quote by Edsger Dijkstra reminds us of a core limitation of testing: "Program testing can be used to show the presence of bugs, but never to show their absence".

Historically, developers often performed unit tests **manually**, by running the code and checking outputs. However, this ad-hoc approach is not structured, not repeatable, and not scalable. Without **automation**, tests are often skipped, poorly documented, or inconsistently applied across the codebase.

To illustrate the concept, consider a simple function:

```
function sum(numbers) {  
    let sum = 0;  
    for (let i = 0; i < numbers.length; i++) sum += numbers[i];  
    return sum;  
}
```

A corresponding unit test might look like this:

```
function testSum() {  
    if (sum([1, 2]) !== 3) throw new Error("1+2 ≠ 3");  
    if (sum([-2]) !== -2) throw new Error("-2 ≠ -2");  
    if (sum([]) !== 0) throw new Error("0 ≠ 0");  
}
```

This test suite checks whether the function correctly sums an array of numbers under different conditions. If any check fails, an error is thrown, signaling that something is wrong with the implementation.

There are a few widely accepted rules of thumb when it comes to unit testing. **All significant logic in the codebase should be covered**. This means that all objects should be tested, and

non-trivial methods (those performing more than simple assignments) should be covered by dedicated tests. **Routine code like simple getters/setters or private helper methods may be omitted** unless they contain meaningful logic. **Before pushing code changes into the version control system, all unit tests should pass**, ensuring that no regressions are introduced.

The advantages of unit testing are substantial. It leads to a **measurable reduction in the number of bugs**, as many errors are caught during development before they ever reach QA or production. It also promotes **better software design**, since writing testable code often leads to cleaner, more modular architecture. Additionally, **tests serve as live documentation**, illustrating how components are intended to behave and interact. From a business perspective, unit testing **reduces the cost of changes and refactoring**, providing developers with the confidence to evolve codebases safely.

An important concept in unit testing is **mocking**. Often, the unit under test depends on other complex components such as databases, APIs, or other services. To **keep the unit test isolated and fast**, these dependencies are replaced with mocks (**fake implementations that simulate the behavior of real objects**). Mocks allow the test to focus solely on the logic of the unit itself, without being affected by the state or availability of external systems.

Modern development heavily relies on testing frameworks to write, run, and manage unit tests efficiently. Popular tools include **QUnit**, **Jasmine**, and **Mocha**. These frameworks offer a structured syntax, support for assertions, setup and teardown hooks, and integration with CI/CD pipelines.

Altogether, unit testing forms the foundation of modern software quality assurance. By enforcing discipline early in the development lifecycle, it leads to more robust, maintainable, and resilient systems.

### 8.9.3 Mocha

**Mocha** is a widely used JavaScript testing framework designed for writing and executing unit tests in Node.js applications. It provides a **simple and expressive syntax** for defining and organizing test suites and individual test cases, making it easier to ensure code correctness throughout development.

A basic Mocha test follows a structured format that uses three core components: **describe**, **it**, and **assert**. In the example below, we test two basic properties of a string (see 07.Testing Examples/01.Basic code):

```
let assert = require('assert');

describe('Basic Mocha String Test', function () {
  it('should return number of characters in a string', function () {
    assert.equal("Hello".length, 5);
  });

  it('should return first character of the string', function () {
    assert.equal("Hello".charAt(0), 'H');
  });
});
```

The function **describe()** is used to **group related tests under a common name**, providing semantic structure to the test suite. It takes two parameters: a string describing the functionality under test, and a function that contains one or more test cases.

Inside the describe block, **each it() function defines an individual test case**. The first argument is a human-readable description of what the test verifies, and the second is a function containing the test logic.

Assertions can be written using the assert module. The function **assert.equal(actual, expected)** checks whether the actual result matches the expected one. If the values differ, the test fails and Mocha will report the failure.

To use Mocha in a Node.js project, we need to install it as a **development dependency** (a package that is only needed during the development phase, but not during production):

```
npm install --save-dev mocha
```

These dependencies are listed under the "**devDependencies**" section in the package.json file:

```
{
  "name": "01",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "mocha"
  },
  "author": "",
  "license": "ISC",
```

```
"devDependencies": {  
    "mocha": "5.0.1"  
}  
}
```

Notice the **test script** section, which allows us to run the tests using the command "npm test". This command will execute Mocha and run all test files in the project:

```
npm test
```

By convention, Mocha looks for test files in the **test/** directory, so we should place all our test files there (for example, `test/test.js`). Each test file should use Mocha's syntax with `describe`, `it`, and assertions.

Mocha will execute tests, displaying a **summary of passed and failed tests** in the terminal. This setup allows us to **continuously verify that your code behaves as expected during development**. For example, the following output shows that one test passed and one failed:

```
Basic Mocha String Test  
  1) should return number of characters in a string  
     ✓ should return first character of the string  
  
  1) passing (2ms)  
  1) failing  
  
  1) Basic Mocha String Test  
      should return number of characters in a string:  
  
      AssertionError [ERR_ASSERTION]: 5 == 6  
      + expected - actual  
  
      -5  
      +6  
  
      at Context.<anonymous> (test/test.js:4:16)  
      at process.processImmediate (node:internal/timers:485:21)
```

Several information is provided in the output. In the example, we can see that the failure is due to an incorrect expected value in the test case that checks the length of the string "Hello". The test expected 6 characters, but the actual result was 5, leading to an assertion error. This

demonstrates how Mocha clearly reports the status of each test, helping developers quickly identify and correct issues in their code. This setup allows us to **continuously verify that your code behaves as expected during development**.

#### 8.9.4 Function Testing

The key idea is that every function is responsible for a clearly defined operation. To verify its correctness, a test must call that function with the necessary input values and compare the actual result with the expected one using assertions.

As a practical example, consider the function "isValidUserId", which checks whether a given user ID is valid. This function takes two parameters: a user ID and a list of valid users. It returns true if the user ID is found in the list, and false otherwise (see 07.Testing Examples/02.TestingFunction code):

```
exports.isValidUserId = function(userList, user) {
  if(!userList) return false;
  return userList.indexOf(user) >= 0;
}
```

The test file should be placed in the test/ directory:

```
const assert = require('assert');
const loginController = require('../controllers/login.controller');

describe('LoginController', function () {
  describe('isValidUserId', function(){
    it('should return true if valid user id', function(){
      const isValid = loginController.isValidUserId(['abc123','xyz321'], 'abc123')
      assert.equal(isValid, true);
    });
    it('should return false if invalid user id', function(){
      const isValid = loginController.isValidUserId(['abc123','xyz321'], 'abc1234')
      assert.equal(isValid, false);
    });
    it('should return false if missing user id', function(){
      const isValid = loginController.isValidUserId(['abc123','xyz321'], null)
      assert.equal(isValid, false);
    });
    it('should return false if missing user list', function(){
      const isValid = loginController.isValidUserId(null, 'xy32')
    });
  });
});
```

```

    assert.equal(isValid, false);
  });
});
});

```

Each test case provides different input scenarios (valid ID, invalid ID, null ID and missing user list) and uses assertions to confirm whether the function behaves as expected in each case. This method demonstrates a clear testing pattern: **define the input, call the function, and use assertions to verify the output**. This approach helps ensure correctness, catch regressions, and improve overall software reliability.

### 8.9.5 Testing asynchronous function

When dealing with **asynchronous code** (such as functions that perform delayed operations or interact with external resources) it is essential to inform Mocha when the test is complete. Otherwise, Mocha might finish the test prematurely, before the callback is invoked. In the following example (see 07.Testing Examples/03.AsyncCallback), the function isValidUserIdAsync simulates an asynchronous check by calling a callback inside a setTimeout:

```

exports.isValidUserIdAsync = function(userList, user, callback) {
  setTimeout(function(){ callback(userList.indexOf(user) >= 0 ), 5);
}

```

To test this function properly in Mocha, the it block must accept a done parameter and call it only after the asynchronous operation finishes. This signals to Mocha that the test is complete:

```

const assert = require('assert');
const loginController = require('../controllers/login.controller');

describe('LoginController', function () {
  describe('isValidUserIdAsync', function(){
    it('should return true if valid user id', function(done){
      loginController.isValidUserIdAsync(['abc123','xyz321'], 'abc123',
      → function(isValid){
        assert.equal(isValid, true);
        // Notifies Mocha that the async test is finished
        done();
      });
    });
  });
})

```

```
});  
});
```

### 8.9.6 Hooks

Hooks are special functions that allow **to run setup and teardown logic before or after tests**. They are particularly useful when we need to prepare a consistent environment before each test runs, or clean up afterward to avoid interference between tests. In particular, **beforeEach()** is used to run code before each test case, typically to set up a known state, mock data, or initialize components. Conversely, **afterEach()** runs after each test case and is commonly used to clean up resources, reset state, or clear side effects that occurred during the test. For example, consider the following code to be tested (see 07.Testing Examples/04.Hooks code):

```
let userList;  
  
exports.loadUserList = function(users) {  
    userList = users;  
}  
  
exports.isValidUserId = function(user) {  
    return userList.indexOf(user) >= 0;  
}  
  
exports.isValidUserIdAsync = function(user, callback) {  
    setTimeout(function(){ callback(userList.indexOf(user) >= 0) }, 1);  
}
```

We can use **beforeEach()** to guarantee that the `users` array is freshly initialized for every test. **afterEach** ensures that the state is cleared after each run. This pattern improves code clarity, avoids duplication, and enhances test reliability:

```
const assert = require('assert');  
const loginController = require('../controllers/login.controller');  
  
beforeEach('Setting up the userList', function(){  
    console.log('beforeEach');  
    loginController.loadUserList(['abc123', 'xyz321']);  
});
```

```
describe('LoginController', function () {
  describe('isValidUserId', function(){
    it('should return true if valid user id', function(){
      const isValid = loginController.isValidUserId('abc123')
      assert.equal(isValid, true);
    });

    it('should return false if invalid user id', function(){
      const isValid = loginController.isValidUserId('abc1234')
      assert.equal(isValid, false);
    });
  });

  describe('isValidUserIdAsync', function(){
    it('should return true if valid user id', function(done){
      loginController.isValidUserIdAsync('abc123',
        function(isValid){ assert.equal(isValid, true); done(); });
    });
  });
});
```

### 8.9.7 Chai

Mocha is often used alongside other libraries, such as **Chai** for **more expressive assertions** or Sinon for mocking. Combined, these tools provide a powerful and flexible environment for building confidence in your code through systematic and automated testing. As an example, we can apply the **Chai assertion library** that provides a more natural and expressive syntax for writing test validations. It can be seamlessly integrated with any JavaScript testing framework such as Mocha and is widely appreciated for its readable, human-friendly style. Chai supports multiple assertion styles:

- **assert**: similar to Node's built-in assertion library
- **expect**: fluent, chainable assertions (`expect(value).to.equal(expected)`)
- **should**: extends object prototypes for chainable assertions (`value.should.equal(expected)`)

For example, the test for the previous login controller can be rewritten using Chai's expect style:

```
const assert = require('assert');
const loginController = require('../controllers/login.controller');

const expect = require('chai').expect;
const should = require('chai').should();

beforeEach('Setting up the userList', function(){
    loginController.loadUserList(['abc123', 'xyz321']);
});

describe('LoginController', function () {
    describe('isValidUserId', function(){

        it('should return true if valid user id', function(){
            const isValid = loginController.isValidUserId('abc123')
            //assert.equal(isValid, true);
            expect(isValid).to.be.true;
        });

        it('should return false if invalid user id', function(){
            const isValid = loginController.isValidUserId('abc1234')
            //assert.equal(isValid, false);
            isValid.should.equal(false);
        });
    });
});

describe('isValidUserIdAsync', function(){

    it('should return true if valid user id', function(done){
        loginController.isValidUserIdAsync('abc123',
            function(isValid){
                //assert.equal(isValid, true);
                isValid.should.equal(true);
                done();
            });
    });
});
});
```

The test code demonstrates how to use both **expect** and **should** styles:

```
expect(isValid).to.be.true;
isValid.should.equal(false);
```

The tests are more readable than traditional assert syntax and clearly describe the intent. Another example is shown in the following test (see 07.Testing Examples/06.ChaiStyle code):

```
const should = require('chai').should();

describe('Object Test', function(){
  it('should have property name', function(){
    const car = { name:'Compass', Maker:'Jeep' }
    car.should.have.property('name');
  });

  it('should have property name with value Compass', function(){
    const car = { name:'Compass', Maker:'Jeep' }
    car.should.have.property('name').equal('Compass');
  });

  it('should compare objects', function(){
    const car = {name:'Compass', Maker:'Jeep'}
    const car1 = {name:'Compass', Maker:'Jeep'}
    car.should.deep.equal(car1);
  });
});
```

Here, the should interface is used to validate object properties:

```
car.should.have.property('name').equal('Compass');
```

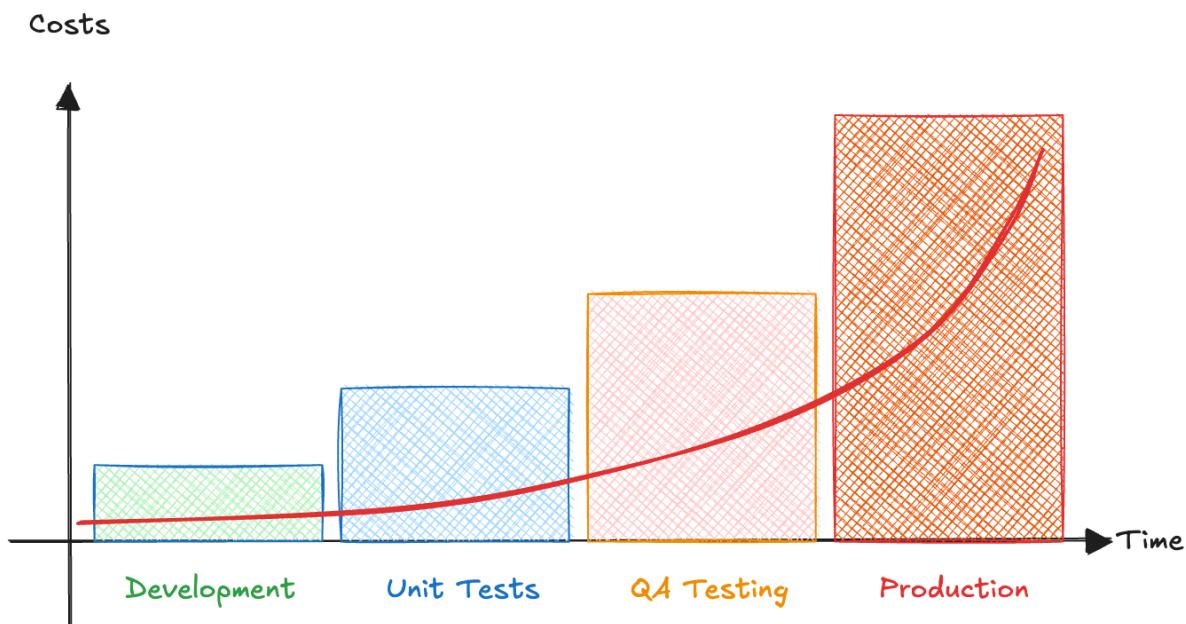
It also includes deep equality checks for object comparisons:

```
car.should.deep.equal(car1);
```

This makes the test output more informative and aligns closely with how developers reason about data structures. In conclusion, Chai enriches the developer experience by transforming traditional assertions into self-descriptive, fluent expressions. Whether we're writing simple unit tests or advanced asynchronous checks, Chai helps ensure our tests communicate exactly what they are validating.

### 8.9.8 Bug cost

In software development, one of the most critical factors influencing project success is how early bugs are detected and fixed. **The cost of fixing a bug increases exponentially the later it is discovered in the software lifecycle:**



During the **development stage**, bugs can typically be fixed by the developer who wrote the code, with full access to the logic, context, and dependencies. At this point, code is still fresh in memory, tests are small and fast, and no user-facing consequences are involved. A correction here may take just minutes.

When bugs are caught in the **unit testing phase**, the cost is slightly higher. Although the code is still under the developer's control, more structured effort is required to write tests, interpret failures, and ensure the fix maintains existing functionality. Nevertheless, the investment in unit testing at this stage still yields high returns by avoiding downstream issues.

As we move into **QA testing**, bugs become more expensive. QA engineers must report issues, developers must analyze logs or reproduce behaviors from less direct feedback, and fixes must go through review, integration, and retesting cycles. Coordination overhead and communication delays add to the cost.

Finally, **bugs discovered in production represent the most critical and costly failures**. At this point, the issue may be affecting real users, causing system outages, data loss, or security breaches. Fixing these errors requires emergency patches, hotfix deployments, and in some cases, legal or reputational damage control. Moreover, debugging in a production environ-

ment is often harder due to limited observability and increased pressure to resolve the problem quickly.

This escalating cost highlights **the importance of early error detection** through practices such as automated testing, code reviews, static analysis, and continuous integration. Investing in these techniques up front minimizes long-term risk and dramatically reduces the total cost of maintenance and bug fixing throughout the project lifecycle.

### 8.9.9 Testing the WoT Proxy

Testing the WoT Proxy API is a critical step to ensure that all RESTful endpoints behave as expected. This involves writing test scripts for routes such as /temperature or /measurement, validating both structure and response content (see 08. Test WoT Proxy code).

To begin, a **test/ folder** should be added to our project structure. Inside it, we can create files like temperature-test.js to test the temperature sensor route:

```
const server = require('../servers/http');
const chai = require('chai');
const chaiHttp = require('chai-http');
const should = chai.should();

chai.use(chaiHttp);

// Temperature /GET route
describe('/GET temperature', () => {
  it('it should GET the temperature', async () => {
    const res = await chai.request(server).get('/iot/sensors/temperature');
    res.should.have.status(404);
    res.body.should.be.a('object');
  });
});
```

These files typically use Mocha as the test runner and Chai (with **chai-http**) to simulate **HTTP requests**. In package.json, we add a test script entry:

```
"scripts": {
  "prod": "node ./wot-server.js -prod",
  "test": "nyc mocha"
},
```

In addition to writing tests, it is essential to track **test coverage**, a metric that quantifies how much of our codebase is exercised when the tests run. Higher coverage generally correlates with fewer undetected bugs. Coverage can be assessed using tools like **nyc**, which provide metrics such as:

- the percentage of executed code statements
- the percentage of covered functions or branches

We can install the tool via:

```
npm install -g nyc
```

Once configured, we can run the tests using **nyc**

```
nyc npm test
```

This will generate a **detailed report** showing which parts of the code were covered and which were not, helping you identify testing gaps. Through this structured approach, the WoT Proxy API can be tested, maintained, and confidently evolved with minimized risk of regressions:

File	%Stmts	%Branch	%Funcs	%Lines	Uncovered Line #s
All files	61.84	33.33	25	61.84	
middleware	57.14	33.33	100	57.14	
converter.js	57.14	33.33	100	57.14	... 31,35,36,37,43
resources	100	100	100	100	
model.js	100	100	100	100	
routes	51.28	100	11.11	51.28	
actuators.js	45	100	0	45	... 24,27,28,31,33
sensors.js	57.89	100	20	57.89	... 13,22,23,28,29
servers	92.86	100	0	92.86	
http.js	92.86	100	0	92.86	22

File	%Stmts	%Branch	%Funcs	%Lines	Uncovered Line #s
All files	61.84	33.33	25	61.84	
middleware	57.14	33.33	100	57.14	
converter.js	57.14	33.33	100	57.14	26-43
resources	100	100	100	100	

model.js	100	100	100	100	
routes	51.28	100	11.11	51.28	
actuators.js	45	100	0	45	8-9,13-14,18-19,24-33
sensors.js	57.89	100	20	57.89	7-8,12-13,22-23,28-29
servers	92.85	100	0	92.85	
http.js	92.85	100	0	92.85	22

## 8.10 Hands-on

Modify the model, the routes, etc. in order to add the support for three new sensors (e.g. humidity, wind and pressure)

Modify the model, the routes, etc. in order to add the support for a new actuators (e.g. a sprinkler) and provide a web page connected using Web Socket to the WoT Proxy in order to show a real-time temperature graph

Add test for the light route

## 9 Persisting Data

The WoT proxy acts as an **intermediary** that exposes physical devices (such as sensors and actuators) through standardized web interfaces. These interfaces typically follow RESTful principles or lightweight protocols like CoAP, enabling seamless access and interoperability. While much of the focus in WoT systems is on **real-time interactions** and **device integration**, a critical requirement in real-world deployments is the ability to **persist data**. Persistence refers to the system's ability to **store data beyond the lifetime of a single session or execution**, thus ensuring that information can be **recovered**, **analyzed**, or **acted upon** at a later time. This includes, for example, logging temperature readings over time, tracking the history of actuator commands, or maintaining configuration metadata.

The current WoT proxy implementation uses a hierarchical object to represent its internal data model. This **in-memory structure** reflects the current state of the system, including sensor values and actuator statuses. However, because the model resides entirely in memory, it is **inherently volatile**. When the proxy is restarted, all data is lost unless explicit mechanisms are introduced to persist this information in a durable manner.

Understanding how to implement persistence in a Node.js application is therefore essential. Persistence can be approached in multiple ways, ranging from **simple file-based storage** to the use of more **sophisticated and scalable database systems**.

Node.js, thanks to its large ecosystem and community, supports a wide range of database systems. Among the most popular choices are **PostgreSQL**, **MySQL**, **Redis**, **SQLite**, and **MongoDB**. Each database offers distinct advantages, and the decision to use one over another depends on various factors. These include the time required to become productive with the tool, the learning curve involved, the performance characteristics of the database engine, ease of replication and backup, as well as the cost of deployment and the level of support offered by the user community.

Importantly, **there is no universally “best” database**. Each use case may benefit from a different approach. For example, applications requiring complex relational queries may prefer SQL-based systems like PostgreSQL, while those dealing with flexible, document-oriented data structures might opt for MongoDB. What matters most is selecting a solution aligned with the technical constraints and development needs of the project.

## 9.1 MongoDB

**MongoDB** stores data in a flexible, JSON-like format called **BSON** (Binary JSON) and allows for easy representation of complex data types and relationships, making it particularly well-suited for applications that require rapid development and fast changing data requirements.

It is an alternative to **traditional relational database** systems and is part of a family of **NoSQL databases** that have emerged to address the limitations of older, schema-bound, table-based models. While relational databases remain powerful and widely used, their underlying architecture were originally designed with strong assumptions about **fixed schemas**, **centralized deployment**, and **vertically scaled infrastructures**. These assumptions do not always align well with the demands of contemporary applications, particularly in the context of large-scale distributed systems or **rapidly evolving data models**. For instance, schema evolution in SQL databases typically requires altering table definitions and migrating data, which can be costly and error-prone in production environments.

MongoDB addresses several of these limitations through its **schema-flexible model**. MongoDB can store documents with **heterogeneous structures**, allowing developers to **adapt data formats over time** without the need for rigid schema declarations. This approach is especially useful in applications where data requirements are dynamic or partially unstructured, such as sensor readings, user-generated content, or log data. Additionally, because MongoDB documents are structurally compatible with JSON, they integrate seamlessly with JavaScript-based environments, including Node.js. This reduces the impedance mismatch between application code and persistent storage.

Moreover, **SQL joins** (while expressive and mathematically elegant) can be **difficult to manage**

in systems where performance, latency, and simplicity are key concerns. Although relational joins are powerful, they often introduce a cognitive overhead for developers and can lead to performance issues when used extensively across large datasets. While modern relational systems have improved in scalability and flexibility, many were originally optimized for the hardware and workloads of previous decades.

That said, MongoDB also introduces some trade-offs. The absence of enforced schemas means that data consistency must often be handled at the application level. Querying capabilities, while substantial, may lack some of the expressiveness and optimization opportunities of advanced SQL engines. Complex joins, for instance, are not natively supported in the same way as in relational databases and must be implemented via **embedding** (nesting documents within documents) or **linking** (storing references and resolving them in application logic or using aggregation pipelines).

### 9.1.1 Collection, document, and field

In MongoDB, we define **collection** a group of documents that share a similar purpose or structure. For example, we might have a collection for users, another for products, and another for orders. Each **document** in a collection is a JSON-like object contains a list of fields. Each **field** is a key-value pair, where the key is a string and the value can be a variety of data types, including strings, numbers, arrays, or even nested documents.

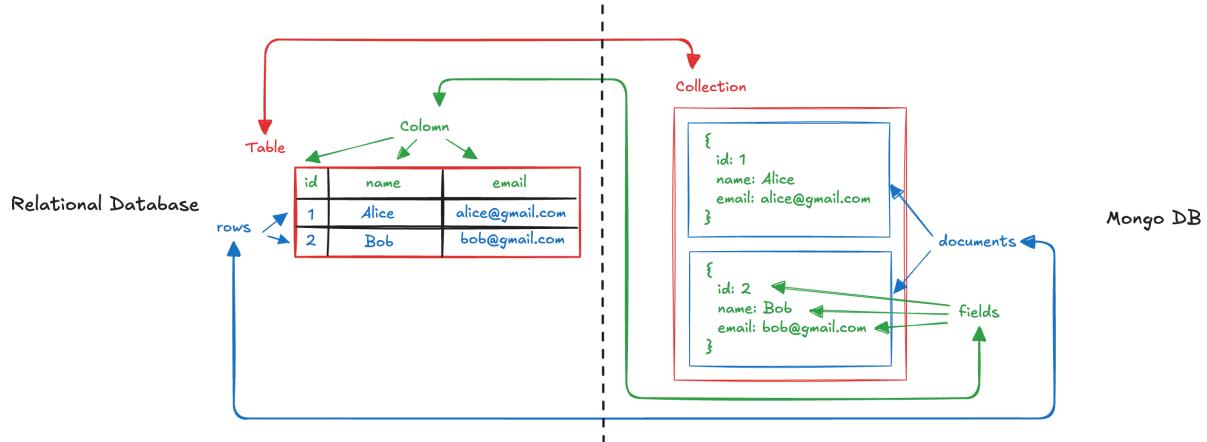
To understand **how MongoDB stores data**, it helps to first compare it with relational databases, which are often the default in traditional applications. Even if you've never worked with a database before, you're probably familiar with the idea of a spreadsheet: you have a table made of rows and columns. In a relational database, data is organized very similarly to a spreadsheet. You define a table, where each row represents an individual record (like one user, one product, or one sensor reading), and each column defines a specific type of information (such as name, date, or temperature). In a relational databases, the structure of data is **strictly defined at the table level**. When creating a table, developers must define in advance the names and types of all columns, and every row stored in the table must comply with this predefined schema.

MongoDB, by contrast, takes a **document-oriented approach**. Instead of enforcing a common schema for all records in a collection, MongoDB allows each document to define its own fields independently. This means that documents within the same collection can have different structures and may contain different sets of keys. The database does not impose any global constraint on the shape of the data. This behavior is commonly described as **schema-less** or **schema-flexible**.

To better understand how MongoDB structures data, it is useful to draw parallels with relational

databases by mapping its core components to more familiar concepts:

- A **collection** is like a table in a relational database, it groups similar pieces of data together.
- A **document** is like a row in a table, it represents one item, like a user or a temperature reading.
- A **field** is like a column—it holds a single piece of information, like a name or a value.



This schema flexibility allows MongoDB to store data that **evolves over time or varies between devices or users**. For example, in a sensor network, one device may record only temperature, while another may include humidity and air pressure. In MongoDB, these can be stored side-by-side in the same collection without the need to update a schema or perform table migrations.

To work with MongoDB, the first step is installing the server and a shell interface. The official MongoDB documentation provides installation instructions across various platforms. Then we can run it manually:

```
mongod --config /opt/homebrew/etc/mongod.conf
```

We need a graphical client to visualize the data. The official **MongoDB Compass client** is a good choice, but there are many other options available. A very popular graphical client, **Studio 3T**, is also available and can simplify exploration and querying. Interaction can take place through its built-in shell, which uses a JavaScript-like syntax. Basic commands include:

```
// Display available methods
db.help()

// Show statistics about the current database
db.stats()
```

```
// List all existing collections
db.getCollectionNames()
```

In MongoDB, we don't explicitly create a database with a separate command (like in SQL). Instead, a database is **created implicitly the first time we store data in it**.

```
use myNewDatabase
```

At this point, myNewDatabase is selected as our current database, but it doesn't actually exist yet. Also **collections are created implicitly**. There is no need to define them in advance. When we insert a document into a collection that does not yet exist, MongoDB automatically creates it. For example, inserting a sensor reading can be done as simply as:

```
db.sensors.insertOne({
  type: "temperature",
  unit: "celsius",
  value: 22
})
```

Once we do this, MongoDB will create: the database myNewDatabase, the collection sensors and the first document inside it. Because MongoDB collections are schema-less, we can insert documents with different structures into the same collection. For example, after inserting a temperature reading, we could insert a heart rate reading with a completely different set of fields:

```
db.sensors.insert({
  name: 'HeartRate',
  device: 'Pulsimeter'
})
```

MongoDB accepts this without complaint, and **both documents will now coexist in the sensors collection**, despite having different fields. This **dynamic behavior** and **flexibility** supports rapid prototyping and development, but it also requires discipline at the application level and it is recommended to **enforce structure at the application level**, for instance using libraries like Mongoose.

### 9.1.2 Arrays and Embedded Documents

Unlike traditional relational databases where **each column has a single scalar type** (e.g., INTEGER, VARCHAR), MongoDB fields can store not only simple values, but also **complex structures** like **arrays** and **embedded documents** as native data types within documents. This allows for the direct modeling of complex, hierarchical, or relational data structures in a way that is both **intuitive** and **efficient**, without the need for traditional relational joins or link tables.

In relational databases, **many-to-many** or **many-to-one** relationships are typically represented using **join tables**. In MongoDB, the same kinds of relationships can be modeled more directly by storing arrays of values or references inside documents. For example, consider a situation where a measurement is recorded by multiple devices. Instead of representing this using a join table, we can store an array of ObjectIds referencing the devices:

```
db.measurements.insert({
  _id: ObjectId('4d85c7039ab0fd70a117d733'),
  value: 76,
  devices: [
    ObjectId('6272386cdf012d4f4ba31b3b'),
    ObjectId('6272386cdf012d4f4ba31b3c')
  ]
})
```

Here, the devices field is **an array of references** to device documents. In addition to arrays, MongoDB supports **embedded documents**, which are nested within other documents. This is particularly useful when related data is always accessed together:

```
db.measurements.insert({
  value: 96,
  user: {
    name: 'Pio',
    surname: 'Blu'
  }
})
```

Here, the user field is a **sub-document** that contains its own fields. It is especially useful in cases where the nested information is **not reused elsewhere** and is always accessed together. This flexibility enables the modeling of hierarchical or nested data directly within a single document, without the need for foreign keys or joins.

We can **combine** both concepts to create **arrays of embedded documents**. This is useful in scenarios such as storing a list of sensor readings, comments, locations, or historical states directly within a parent document:

```
db.devices.insert({  
  device: 'thermometer-1',  
  readings: [  
    { timestamp: '2024-01-01T12:00:00Z', value: 22.1 },  
    { timestamp: '2024-01-01T12:05:00Z', value: 22.3 }  
  ]  
})
```

This structure improves read performance and simplifies data modeling when the embedded elements are tightly related to the parent and are not queried independently.

### 9.1.3 Find documents

Once data is stored in a collection, the most basic and frequently used operation is **reading it back**. This is done using the `find()` method. In its simplest form, calling `find()` with no parameters returns all documents in the current collection. For example, we have previously inserted a temperature reading into a collection called `sensors`, we can retrieve it with:

```
db.sensors.find()
```

We obtain a JSON list with the data we just inserted:

```
[  
  {  
    "_id" : ObjectId("682c46eef6228fe221ea7e0f"),  
    "type" : "temperature",  
    "unit" : "celsius",  
    "value" : NumberInt(22)  
  },  
  {  
    "_id" : ObjectId("682c7925ccb76ce603ea7e0f"),  
    "name" : "HeartRate",  
    "device" : "Pulsimeter"  
  }  
]
```

### 9.1.4 ObjectId

The result of the `find` operation contains not only the fields we inserted (name, unit, value), but also \*\*a special field named `_id`. **This field is automatically added by MongoDB if not explicitly provided. Its purpose is to serve as the primary key for the document\*\***, uniquely identifying it within the collection. Every document must have a unique `_id`. MongoDB will raise an error if two documents share the same value for this field.

By default, the value of `_id` is of type **ObjectId**, a special 12-byte identifier generated by MongoDB that encodes a timestamp and some random bits. While we can manually assign a value to `_id`, it is generally best to let MongoDB manage it unless we have a specific reason to control it.

### 9.1.5 Indexes

An important aspect of `_id` is that it is **automatically indexed**. This means that searching for documents by `_id` is very **efficient**. We can verify the presence of this index using:

```
db.sensors.getIndexes()
```

This command lists all indexes associated with the collection, and we will see an entry for the `_id` field:

```
[  
  {  
    "v" : 2.0,  
    "key" : {  
      "_id" : 1.0  
    },  
    "name" : "_id_"  
  }  
]
```

Indexes function much like indexes in traditional relational databases. They are critical for **enhancing query performance**, particularly for large collections, by allowing MongoDB to locate documents without scanning every entry in the collection. When we query a collection without an index, MongoDB performs a **collection scan**, it examines every document to check whether it matches the query criteria. This is inefficient for large datasets. An index creates a **special data structure**, typically a **B-tree**, a balanced tree data structure that stores the indexed

field values in **sorted order**, allowing efficient retrieval. Indexes are created using the `createIndex()` method and removed using `dropIndex()`. For example:

```
db.devices.createIndex({ name: 1 })
```

This creates an ascending index on the `name` field. To remove the same index:

```
db.devices.dropIndex({ name: 1 })
```

The value `1` indicates ascending order; using `-1` would create a descending index instead. We can enforce **uniqueness** on a field by specifying the `unique` option. This ensures that MongoDB will reject any insert or update that would cause duplicate values in the indexed field:

```
db.devices.createIndex({ name: 1 }, { unique: true })
```

MongoDB allows the creation of **compound indexes**, which index **multiple fields in a single structure**. These are helpful for queries that involve filtering or sorting on more than one field.

```
db.devices.createIndex({ name: 1, values: -1 })
```

MongoDB supports **indexing nested fields** (using dot notation) and fields that contain arrays:

```
db.devices.createIndex({ 'user.name': 1 })
```

This creates an index on the `name` field of an embedded `user` object. If a field contains an array, MongoDB automatically creates **a separate index entry for each element of the array**. This allows us to efficiently query documents where any array element matches a condition. MongoDB offers **specialized index types** for specific use cases, including:

- **Geospatial Indexes** for querying location data (`2dsphere`)
- **Text Indexes** for full-text search on strings
- **Hashed Indexes** for sharding on a hashed key

These advanced indexes are beyond the scope of this chapter but are covered in detail in the MongoDB documentation.

### 9.1.6 Query Selectors

It is often necessary to retrieve **only a subset of documents** from a collection based on certain **conditions**. This is done using **query selectors**, which serve a similar role to the **WHERE clause** in SQL. These selectors are passed as arguments to the `find()` function (and other operations such as `count()`, `update()`, and `delete()`), and are expressed as JavaScript objects that define **criteria a document must meet** in order to be included in the result set. MongoDB evaluates each document against the query conditions and returns only the ones that match. To experiment with query selectors, we can begin by loading some sample documents into a collection, which include multiple devices with various attributes like type, weight, tags, and values:

```
db.devices.insert({
  name: 'Device 1',
  timestamp: new Date(1992, 2, 13, 7, 47),
  tags: ['tag_a', 'tag_b'],
  weight: 600,
  type: 'm',
  values: 63
});

db.devices.insert({
  name: 'Device 2',
  timestamp: new Date(1991, 0, 24, 13, 0),
  tags: ['tag_a', 'tag_c'],
  weight: 450,
  type: 'f',
  values: 43
});

db.devices.insert({
  name: 'Device 3',
  timestamp: new Date(1973, 1, 9, 22, 10),
  tags: ['tag_d', 'tag_e'],
  weight: 984,
  type: 'm',
  values: 182
});

db.devices.insert({
  name: 'Device 4',
  timestamp: new Date(1979, 7, 18, 18, 44),
  tags: ['tag_f'],
```

```
        weight: 575,
        type: 'm',
        values: 99
    });

db.devices.insert({
    name: 'Device 5',
    timestamp: new Date(1985, 6, 4, 2, 1),
    tags:['tag_f', 'tag_a', 'tag_g'],
    weight:550,
    type:'f',
    values:80
});

db.devices.insert({
    name:'Device 6',
    timestamp: new Date(1998, 2, 7, 8, 30),
    tags: ['tag_h', 'tag_i'],
    weight: 733,
    type: 'f',
    values: 40
});

db.devices.insert({
    name:'Device 7',
    timestamp: new Date(1997, 6, 1, 10, 42),
    tags: ['tag_c', 'tag_i'],
    weight: 690,
    type: 'm',
    values: 39
});

db.devices.insert({
    name: 'Device 8',
    timestamp: new Date(2005, 4, 3, 0, 57),
    tags: ['tag_f', 'tag_l'],
    weight: 421,
    type: 'm',
    values: 2
});

db.devices.insert({
    name: 'Device 9',
```

```
        timestamp: new Date(2001, 9, 8, 14, 53),
        tags: ['tag_f', 'tag_m'],
        weight: 601,
        type: 'f',
        values: 33
    });

db.devices.insert({
    name: 'Device 1',
    timestamp: new Date(1997, 2, 1, 5, 3),
    tags: ['tag_f', 'tag_m'],
    weight: 650,
    type: 'm',
    values: 54
});

db.devices.insert({
    name: 'Device 1',
    timestamp: new Date(1999, 11, 20, 16, 15),
    tags: ['tag_c', 'tag_a'],
    weight: 540,
    type: 'f'
});

db.devices.insert({
    name: 'Device 1',
    timestamp: new Date(1976, 6, 18, 18, 18),
    tags: ['tag_c', 'tag_m'], weight: 704,
    type: 'm',
    values: 165
});
```

The most basic query checks for **equality between a field and a value**. For example:

```
db.devices.find({ type: 'm' })
```

This will return all documents where the type field is exactly equal to 'm'. We can specify **multiple field-value pairs** in the query object. This creates an **implicit AND condition**, meaning all conditions must be met simultaneously:

```
db.devices.find({ type: 'm', weight: 650 })
```

This retrieves documents where type is 'm' and weight is 700. MongoDB allows us to **query embedded fields** using dot notation:

```
db.measurements.find({ 'user.name': 'Pio' })
```

MongoDB provides a set of **special operators** for more complex comparisons. These include:

- **\$lt**: less than
- **\$lte**: less than or equal
- **\$gt**: greater than
- **\$gte**: greater than or equal
- **\$ne**: not equal

As an example, to find all devices of type "m" whose weight is greater than 700:

```
db.devices.find({ type: 'm', weight: { $gt: 700 } })
```

Or to find all devices where type is not "f" and weight is less than or equal to 700:

```
db.devices.find({ type: { $ne: 'f' }, weight: { $lte: 700 } })
```

We can also query for the **presence or absence of a field** using the **\$exists** operator:

```
db.devices.find({ values: { $exists: false } })
```

This returns all documents where the values field is missing. If we want to **match any value from a list**, use the **\$in** operator. It acts like **a logical OR within a field**:

```
db.devices.find({ tags: { $in: ['tag_a', 'tag_c'] } })
```

This retrieves documents where the tags array contains either 'tag\_a' or 'tag\_c'. MongoDB also allows **combining multiple conditions using \$or**, which is explicitly defined as an array of sub-conditions:

```
db.devices.find({
  type: 'f',
  $or: [
    { tags: 'tag_c' },
    { weight: { $lt: 600 } }
  ]
})
```

This query returns all documents where the type is 'f', and either the tags field contains 'tag\_c' or the weight is less than 600. By mastering these query selectors, developers gain precise control over which documents are retrieved from a collection, enabling both **simple filters** and **more advanced search logic**.

The same query selectors used in `find()` can also be applied to other operations:

```
// Remove all documents where tags includes 'obsolete'
db.devices.remove({ tags: 'obsolete' })

// Count how many documents contain a particular field
db.devices.count({ type: 'sensor' })
```

This consistency allows developers to use a unified syntax across different types of operations.

### 9.1.7 Projection

When retrieving documents, we may not always need all the fields. MongoDB allows us to **specify which fields to include or exclude in the result set using projection**. This is done by passing a second argument, which is an object that specifies the desired fields. For example, if we want to retrieve only the name field from our documents:

```
db.devices.find({}, { name: 1 })
```

By default, MongoDB always includes the `_id` field. If we wish to exclude it, we must do so explicitly:

```
db.devices.find({}, { name: 1, _id: 0 })
```

It is important to note that a projection in MongoDB must be **either inclusive or exclusive**, meaning we can choose to specify only the fields to include (e.g., `{ name: 1, weight: 1 }`) or only

the fields to exclude (e.g., `{ name: 0, unit: 0 }`), but we **cannot mix inclusion and exclusion** in the same projection \*\*with the exception of the `_id` field\*\*. The `_id` field can be excluded (using `_id: 0`) in an otherwise inclusive projection.

### 9.1.8 Lazy Evaluation

When we perform a query in MongoDB using `find()`, the database does **not immediately return the entire result set**. Instead, it returns a **cursor**, a pointer to the result set that **fetches documents lazily**, i.e., only as they are needed. This design provides both **memory efficiency** and **performance benefits**, especially when dealing with large datasets.

Lazy evaluation means that MongoDB does not load all matching documents into memory immediately after you issue a query. Instead, **documents are retrieved from the server in batches** as the application or shell begins to iterate over the results. This helps avoid wasting resources on documents that may never be used.

In practice, this behavior is **often hidden in the shell**, because the shell automatically iterates over the cursor to display a sample of the results. For example, when we type:

```
db.devices.find()
```

the shell appears to instantly return all results. In reality, it is fetching documents on-demand and only a subset at first. In the shell, we can simulate explicit cursor iteration:

```
const cursor = db.devices.find();

while (cursor.hasNext()) {
    printjson(cursor.next());
}
```

This loop retrieves documents one at a time, allowing us to observe the **lazy evaluation as documents are fetched incrementally** from the server. MongoDB by default retrieves documents in batches (typically 101 documents for the first batch and 4.096 for subsequent ones). We can control this behavior using the `.batchSize(n)` method:

```
db.devices.find().batchSize(10)
```

This can be useful in applications where precise control over network traffic and memory usage is required.

### 9.1.9 Ordering and Pagination

To order query results, MongoDB provides the `sort()` method, which takes a document specifying the fields to sort by and the direction (1 for ascending order, and -1 for descending order):

```
db.devices.find().sort({ weight: -1 })
```

For example, this sort results by descending weight. To sort by name (ascending) and values (descending):

```
db.devices.find().sort({ name: 1, values: -1 })
```

For applications that display results in **pages** (e.g., web interfaces or dashboards), MongoDB supports pagination via the `limit()` and `skip()` cursor methods:

```
db.devices.find().sort({ weight: -1 }).limit(2).skip(1)
```

This example retrieves the second and third heaviest devices by first sorting the documents in descending order by weight, then skipping the first one and limiting the output to the next two. Pagination is an essential feature for scalable user interfaces and APIs, especially when working with large collections. It is always recommended to use pagination in production applications rather than returning entire datasets.

Lazy evaluation is particularly important in ordering and pagination, as it ensures that MongoDB retrieves and processes only the subset of documents needed for the current page, rather than loading the entire sorted dataset into memory—making queries more efficient and scalable, especially on large collections.

### 9.1.10 Update documents

Updating documents is a fundamental operation that allows us to **modify existing data within a collection**. The basic idea is to locate the documents to modify and then apply specific changes to them. While the syntax appears simple, understanding how it works in different scenarios is essential to avoid unintended behavior. Mongo requires the use of **atomic update operators**, such as `set` \*, `**inc`, and `$unset`. These operators **explicitly define which parts of a document should be changed**. For example, to `updateOne()` method is used to update a single document:

```
db.devices.updateOne(  
  { name: 'Device 2' },  
  { $set: { weight: 590 } }  
)
```

This command finds the first document where name is "Device 2" and updates only the weight field, leaving all other fields unchanged. The `$set` operator is used to specify the field to be updated and its new value. If the `*inc` operator increments or decrements numeric fields:

```
db.devices.updateOne(  
  { name: 'Device 3' },  
  { $inc: { values: -2 } }  
)
```

The `$unset` operator removes a field from the document:

```
db.devices.updateOne(  
  { name: 'Device 3' },  
  { $unset: { obsoleteField: "" } }  
)
```

To **update all documents** that match a query, use `updateMany()` instead of `updateOne()`:

```
db.devices.updateMany(  
  { type: 'f' },  
  { $set: { obsoleteField: "obsolete" } }  
)
```

MongoDB also supports **upserts**, which update the matching document if found, or insert a new one if not:

```
db.devices.updateOne(  
  { name: 'Device 15' },  
  { $set: { hits: 1 } },  
  { upsert: true }  
)
```

If "Device 15" exists, its `hits` field is updated. If not, a new document is inserted with name and `hits`.

### 9.1.11 Aggregation

Aggregation is the process of analyzing and transforming data to produce computed results, often by summarizing, filtering, or reshaping multiple documents. It is like SQL's GROUP BY, or COUNT() clauses. Aggregation is especially useful when we want to group values from multiple documents, calculate statistical metrics (like sums or averages), or prepare data for reports and dashboards.

The aggregation pipeline is the most powerful and flexible approach. It uses a series of transformation stages, each operating on the output of the previous one, to progressively reshape and analyze the data:

```
db.devices.aggregate([
  // stages go here
])
```

Each stage is defined as a JSON object and uses operators like *match* \*, \*\*group, *project* \*, \*, \*\*sort, and others. The first stage is often \$match, which filters documents based on a condition (similar to find()):

```
{ $match: { weight: { $gt: 700 } } }
```

The next stage could be \$group, which allows us to group documents by a specific field and apply aggregation functions such as \$sum, \$avg, \$max, or \$min:

```
{
  $group: {
    _id: null,
    total: { $sum: "$values" }
  }
}
```

Putting these together:

```
db.devices.aggregate([
  { $match: { weight: { $gt: 700 } } },
  { $group: { _id: null, total: { $sum: "$values" } } }
])
```

This example filters devices with weight > 700 and computes the total of their values field. Aggregation pipelines are **extremely versatile** and support dozens of stages for tasks like projection, array unwinding, joining collections (*lookup*), and reshaping documents (*project*, *\$addFields*, etc.).

MongoDB also provides some **built-in helper methods** for **common aggregation tasks**:

- `estimatedDocumentCount()` gives an approximate count of all documents in a collection.
- `countDocuments()` returns an exact count, optionally with a filter.
- `distinct(field)` returns all the unique values for a specified field.

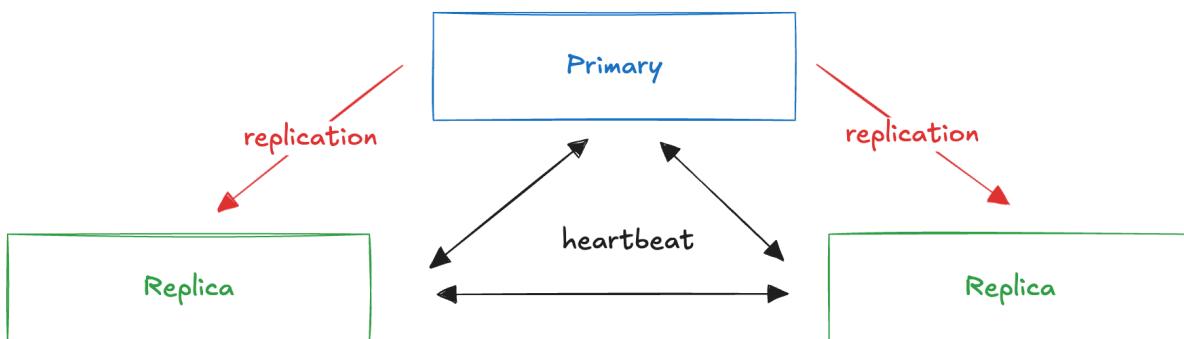
These methods are simpler and faster for basic aggregation but lack the composability of pipelines.

### 9.1.12 Replication and Sharding

To support **large-scale**, **reliable**, and **fault-tolerant** applications, MongoDB provides built-in mechanisms for replication and sharding.

A **replica set** is a group of MongoDB servers that maintain the same dataset:

- a replica set typically includes one primary and two or more secondary nodes.
- all writes and strongly consistent reads are handled by the primary node.
- The secondary nodes receive data updates asynchronously via replication and can serve read-only queries.



Key Benefits are **high availability** (if the primary fails, the system automatically promotes a secondary to become the new primary via an election process), **performance optimization** (long-running or read-intensive queries can be offloaded to secondaries, freeing the primary for write operations) and **durability** (data is maintained across multiple nodes, reducing the risk of data loss). However, reading from secondaries introduces a potential for **slightly stale data**, depending on replication lag.

**Sharding** is a method for **horizontal scaling**. It works by partitioning data across multiple machines (called shards), each holding only a subset of the total data. This allows to handle datasets that exceed the capacity of a single machine, both in terms of storage and query throughput:

- Data is divided based on a **shard key**, such as a user ID or geographic region.
- A **router** (mongos) directs queries to the appropriate shard.
- A config server stores metadata about the **distribution of data** across shards.

The **CAP theorem (Consistency, Availability, Partition tolerance)** states that in a distributed system, we can only **guarantee two out of the three properties** at any given time:

- **Consistency**: every read receives the most recent write or an error.
- **Availability**: every request receives a (non-error) response, even if it's not the most recent.
- **Partition Tolerance**: the system continues to operate even if some nodes cannot communicate due to a network partition.

In practice, partition tolerance is unavoidable, so systems must choose between **consistency** and **availability**

### 9.1.13 Transactions

A transaction is a sequence of one or more operations that are **executed as a single, atomic unit of work**. This means that either all operations in the transaction are successfully applied, or none are, ensuring the database remains in a consistent state even in the event of errors or interruptions. The **ACID** properties (Atomicity, Consistency, Isolation, Durability) allows applications to safely modify multiple elements:

- **Atomicity**: a transaction is treated as a single unit. Either all the operations within the transaction succeed together, or none are applied at all.
- **Consistency**: the database is always transitioned from one valid state to another. A transaction cannot leave the database in an invalid state.
- **Isolation**: concurrent transactions are executed as if they were run sequentially. Intermediate results are invisible to other operations.
- **Durability**: once a transaction is committed, its effects persist, even in the event of a crash or power failure.

MongoDB supports **atomic updates on individual documents**. This means that if we modify a single document (including any of its embedded fields or arrays) the entire operation is guaranteed to be atomic. Since MongoDB supports rich document structures, many relationships

(e.g., a blog post and its comments) can be embedded within a single document. This often eliminates the need for multi-document transactions in many real-world applications.

However, sometimes related data must be updated across multiple documents, collections, or even databases. MongoDB supports full ACID transactions across multiple documents, collections, and even databases, starting from version 4.0 (for replica sets) and 4.2 (for sharded clusters). Single-document operations remain atomic by default, making MongoDB suitable for many use cases without the overhead of full transactions:

```
const session = db.getMongo().startSession();

session.startTransaction();

try {
    session.getDatabase('store').orders.insertOne({ item: 'Book', qty: 1 });
    session.getDatabase('store').inventory.updateOne(
        { item: 'Book' },
        { $inc: { stock: -1 } }
    );
    session.commitTransaction();
}

catch (e) {
    session.abortTransaction();
    throw e;
}

finally {
    session.endSession();
}
```

This transaction inserts a new order and decrements inventory, ensuring that both changes either succeed together or not at all. Use transactions when changes across multiple documents must be applied together, and always prefer simpler atomic operations when they suffice.

### 9.1.14 Profile and Backup

To maintain, monitor, and safeguard MongoDB databases, developers have access to several built-in tools and commands. These enable us to inspect the database's internal state, monitor performance, and manage backups and data exports.

MongoDB provides internal commands to obtain **basic statistics** about a database or collection.

```
// Get information about the current database
db.stats()

// Retrieve collection-specific information.
db.collection.stats()
```

The output includes: number of documents, total size on disk, number and size of indexes, and storage engine details. These statistics help **diagnose performance issues** and **monitor collection growth over time**.

MongoDB includes a **database profiler**, which logs detailed information about operations executed against the database. We can configure the profiler using:

```
// 0: Profiler is off (default).
// 1: Profiler logs slow operations (longer than 100ms by default).
// 2: Profiler logs all operations, useful for debugging.

db.setProfilingLevel(level)
```

Collected profile entries are stored in the system collection:

```
db.system.profile.find()
```

This lets us to inspect\*\* which queries are running, how long they take, how many documents were scanned, **and** how many were returned\*\*, etc. which are very useful information for optimization and troubleshooting.

Finally, MongoDB provides two core utilities for **creating** and **restoring** backups. **Mongodump** creates a binary backup of our databases and collections as BSON files:

```
mongodump --db mydb --collection devices
```

**Mongorestore** restores from a mongodump backup. Useful for disaster recovery, migration, or replication seeding:

```
mongorestore --db mydb dump/mydb
```

These tools are ideal for structured backups and full recovery workflows. For exporting data into formats like CSV or JSON (rather than binary BSON), we can use the **Mongoexport** tool:

```
mongoexport --db mydb --collection devices --csv --fields name,weight
```

This is helpful for reporting, analytics, or migration to external systems.

## 9.2 How to model data

Designing a good **data model** is one of the most important aspects of building scalable and maintainable applications. MongoDB is designed as a **direct alternative to relational databases**, offering a different set of trade-offs optimized for **flexibility, scalability, and developer productivity**. Understanding when, why and how to use MongoDB helps in choosing the right tool for a given problem.

### 9.2.1 Embedded Documents or Referencing?

A common decision is whether to **embed related data directly in a document or reference it in another collection**.

Embedded documents are ideal for closely related data that is always accessed together. A classic example is storing a list of addresses directly inside a user document. This avoids the need for joins and improves read performance. However, documents have a size limit (16 MB), so large or unbounded data (e.g., long logs, millions of comments) should not be embedded. Embedding is generally preferred for **smaller subdocuments** that are **tightly coupled** with the parent. It often eliminates the need to perform joins, which can improve performance and reduce complexity.

Referencing is more appropriate when the related data is **large or shared** across multiple documents, the subdocuments need to be accessed **independently**, and there is frequent write activity that would affect embedded arrays. In such cases, storing references (typically using ObjectIds) allows **decoupling of data** and better **scalability**.

### 9.2.2 Few or Many Collections?

Because MongoDB does not enforce a schema, it is technically possible to store all documents in a **single collection**, even mixing different types. This is strongly **discouraged**.

A more sustainable and logical approach is to **design collections similarly to relational tables**. If a concept would naturally be a table in a relational database, it probably should be a collection in MongoDB.

The exception is for join tables in many-to-many or one-to-many relationships, which can often be replaced with arrays or embedded subdocuments.

Consider a blog application. Should we:

- Create a posts collection and a separate comments collection?
- Or embed an array of comments directly inside each post?

In most real-world scenarios, separating the collections is cleaner and more flexible, especially if comments need their own metadata, moderation workflow, or indexing. However, for performance-sensitive use cases, we may embed a few of the most recent comments inside each post to improve display efficiency, while keeping the full set in a separate collection. This **hybrid approach** provides the best of both worlds: fast access to recent data, and scalable management of the full dataset.

A useful rule of thumb when modeling data in is: **keep together data that you want to retrieve together**. Embedding is appropriate when the data is queried as a unit, while referencing is preferred when data is large, shared, or independent.

### 9.2.3 Schema-on-read or schema-on-write?

One of defining features is that it is **schema-less**, we are not required to predefine the structure of documents in a collection. This is more flexible than traditional relational database tables. **Documents can evolve over time**, which makes MongoDB particularly well-suited to applications where:

- The schema is rapidly evolving (e.g., agile development, startups).
- Occasional structural changes are acceptable or even expected.
- Objects need to be stored “as-is” without extensive mapping logic.

This flexibility reduces the friction when using MongoDB in object-oriented programming (OOP) environments. You can serialize an object to JSON and store it directly in the database, with no need for additional property or type mappings. As an example, a logging database benefits from schema-less design: each log entry might contain different fields depending on its source, and flexibility is more important than strict uniformity.

However, it is important to note that most real-world applications still involve **structured data**. A schema-less model\*\* doesn't mean schema-free, **it means** schema-on-read\*\* rather than

**schema-on-write.** Developers should still enforce consistency at the application level when necessary.

#### 9.2.4 Write Performance and Data Durability?

MongoDB gives developers **fine-grained control** over the trade-off between write performance and data durability: we can configure write operations to **return immediately** without waiting for the data to be written to disk. This improves throughput in **write-intensive scenarios** but may **risk durability** in case of failure.

To prioritize **write performance** over durability:

```
db.collection.insertOne(  
  { name: "fastWrite", timestamp: new Date() },  
  { writeConcern: { w: 1, j: false } }  
)
```

This operation will return as soon as the primary node receives the write in memory (w: 1), before it's written to disk or replicated (j: false). It is really fast, but at the risk of data loss on crash.

To ensure **durability**, you can use:

```
db.collection.insertOne(  
  { name: "fastWrite", timestamp: new Date() },  
  { writeConcern: { w: "majority", j: true } }  
)
```

This ensures the write is committed to the majority of replica set members (w: "majority") and written to disk (j: true).

MongoDB supports **capped collections**, which are fixed-size collections designed for high-throughput data ingestion. When the size limit is reached, the oldest documents are automatically removed:

```
db.collection.insertOne(  
  { name: "fastWrite", timestamp: new Date() },  
  { writeConcern: { w: 1, j: false } }  
)
```

```
db.createCollection('logs', { capped: true, size: 1048576 })
```

This is ideal for logs or event streams where only the most recent data is relevant.

### 9.3 Object-Relational Mapping (ORM)

When it comes to interacting with a database from Node.js, developers typically follow one of two main approaches.

The first is to **use the native query language of the database**, such as SQL for relational databases or the MongoDB query language for document stores. This approach provides the most control and efficiency, especially for developers already familiar with the database's query syntax.

The second approach relies on **Object Relational Mapping (ORM)**. ORMs abstract away the underlying database structure by allowing developers to manipulate data as if they were working with regular objects. This model is especially beneficial for those who prefer to think in terms of object-oriented code rather than low-level query semantics. However, this abstraction comes at a cost: ORMs introduce a translation layer that maps object structures to database representations, which can lead to less efficient queries and slightly **slower performance** compared to raw queries.

Despite these drawbacks, ORMs significantly improve **developer productivity**, particularly in projects where rapid prototyping or database switching is expected. Unless one is deeply experienced with the native query syntax of a given database, adopting an ORM is often a safer and more sustainable choice. It also improves code readability and reduces the likelihood of query-related errors in complex applications.

The Node.js ecosystem offers a rich set of ORM libraries available through NPM. For MongoDB, one of the most widely used and robust options is **Mongoose**, which provides a straightforward interface for **schema definition** and **data manipulation**. Other ORMs such as Waterline, Bookshelf, Objection, and Sequelize target different types of databases and follow slightly different design philosophies. Choosing the right ORM depends on the type of database in use, the development style of the team, and the expected scale and complexity of the application.

#### 9.3.1 Mongoose

Mongoose offers a robust and structured approach to modeling data. It sits on top of the native MongoDB driver and enhances it by introducing **application-level schemas, built-in validation**,

**idation**, and **powerful querying tools**. While MongoDB itself is schema-less and highly flexible, Mongoose introduces a layer of structure that helps developers write more predictable and maintainable code.

One of the main reasons to use Mongoose is its ability to **reconcile the flexibility of NoSQL databases with the structural expectations of modern application logic**. It allows developers to define schemas that describe the shape and constraints of the documents in a collection, similar to how tables and fields are defined in relational databases. These schemas support features like **default values**, **type enforcement**, **validation rules**, and even **middleware functions** that can run before or after database operations. Additionally, Mongoose **handles type casting automatically**, converting JavaScript types into MongoDB-compatible formats.

This structured approach is particularly valuable in applications that require well-defined and consistent data models (such as RESTful APIs, microservices, or IoT platforms), where it is important to ensure that data entering and leaving the system **adheres to expected formats**. By combining MongoDB's flexibility with Mongoose's structure, developers gain both rapid prototyping capabilities and production-level data integrity.

To add Mongoose to our Node.js project:

```
npm install mongoose
```

Mongoose uses the native MongoDB connection string format. A typical connection pattern looks like:

```
const mongoose = require('mongoose');

mongoose.connect('mongodb://localhost:27017/wotserver-prod');

mongoose.connection.on('error', (err) => {
  console.error('Database connection error: ' + err.message);
});
```

In most applications, the connection is **initialized once at startup**, and then reused throughout the lifetime of the app. Avoid opening and closing the connection on every request. We typically place this connection logic in a module like `database.js`, which is then imported where needed.

### 9.3.2 Schemas

In Mongoose, a schema defines the **shape** and **structure** of the documents that will be stored in a collection. While MongoDB itself is schema-less, Mongoose **enforces structure at the ap-**

**lication level** by allowing developers to explicitly declare the fields and types each document should contain. A schema acts as a blueprint: it tells Mongoose **what fields to expect, what types those fields should be, whether any are required, and what default values (if any) should be used**. To define a schema, we can use the `mongoose.Schema` constructor. Here an example of a schema to store measurements from sensors:

```
const measurementSchema = new mongoose.Schema({
  type: { type: String, required: true },
  value: { type: Number },
  timestamp: { type: Date, default: Date.now }
});
```

In this schema, each key (type, value, timestamp) defines a property that will appear in the documents. Each property is cast to a specific SchemaType. Common SchemaTypes include:

- String
- Number
- Date
- Boolean
- Buffer
- Mixed (an arbitrary value)
- ObjectId (used to reference other documents)
- Array
- Decimal128
- Map

Each type can be customized with options such as required, default, enum, min, max, or custom validation logic. To use the schema in the application, it must be **converted into a model**. This is done with the `mongoose.model()` function, which takes two arguments: the name of the model (which becomes the name of the collection, pluralized), and the schema definition:

```
module.exports = mongoose.model('Measurement', measurementSchema);
```

Once exported, the model can be used to create, query, and update documents in the corresponding MongoDB collection using the full Mongoose API.

### 9.3.3 Persiste and fetch data

Once a schema and model are defined, we can create and persist new documents and later retrieve them using flexible query parameters. To save data to the database, we create an instance

of a Mongoose model and call `.save()` on it: For example, the measurement model can be used to store a sensor readings:

```
const Measurement = mongoose.model('Measurement');

mearure = new Measurement({
  type: "temperature",
  value: 20
})

mearure.save();
```

This creates a new instance of the Measurement model, populates it with data, and saves it to the database. Mongoose automatically handles the conversion of JavaScript types to their MongoDB equivalents, including ObjectId for references. The `save()` method returns a promise, allowing us to handle success or failure using `.then()` and `.catch()`.

To fetch data, we can use the model's static methods. Mongoose provides a rich set of query helpers that allow us to filter, sort, and paginate results easily. For example, to find only the most recent 10 measurements of temperature:

```
const results = await Measurement.find({ type: 'temperature' })
  .sort({ timestamp: -1 }) // most recent first
  .limit(10);             // limit to 10 results

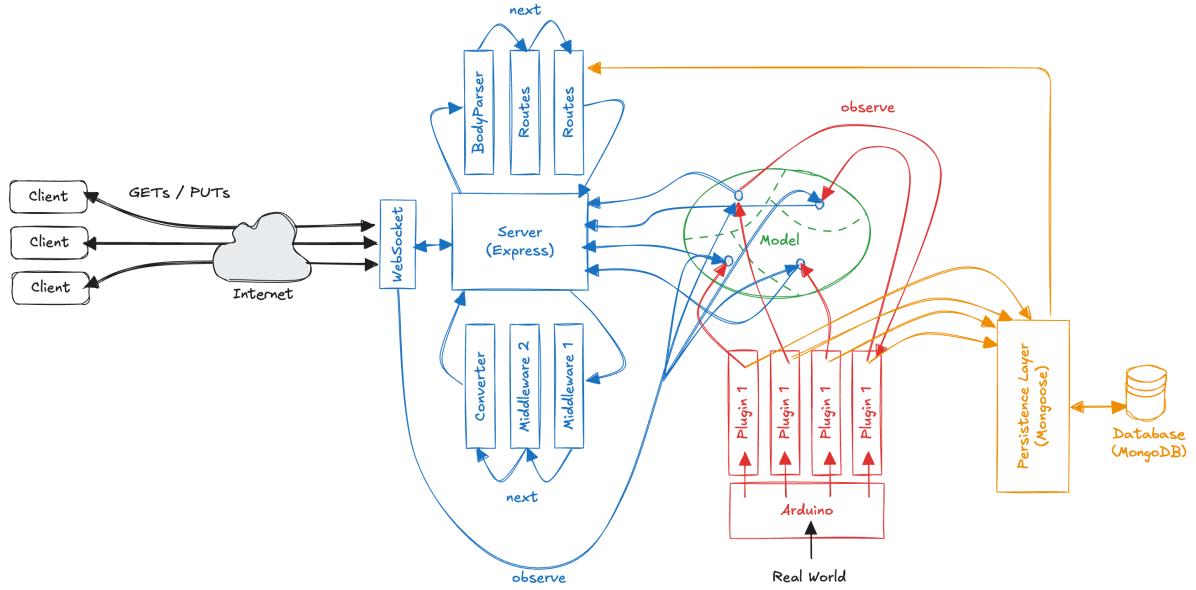
console.log('Latest temperature measurements:');
results.forEach(doc => {
  console.log(` ${doc.timestamp.toISOString()} → ${doc.value}`);
});
```

This code performs a **filtered**, **sorted**, and **paginated** query, then logs each result in a readable format. It's ideal for showing recent sensor data in a dashboard or logging context.

## 9.4 WoT Proxy with persistence

In the WoT Fog Server architecture, database support can be added through a clear separation between persistence logic and the data modeling layer. The MongoDB database serves as the **underlying persistent storage**, holding sensor measurements over time. Mongoose acts as the **persistence layer**, providing schema enforcement, validation, and structured access to the

database. It connects the model component to the MongoDB instance, allowing the application to interact with the database in a structured and consistent way:



This setup abstracts direct database access away from the server logic and plugins, enabling centralized data management and making it easier to apply validation, track changes, and enforce consistency. Overall, the architecture demonstrates a clean use of ORM-based persistence, where the model is the focal point for both runtime operations and durable storage via Mongoose (see code/01. Persistent WoT Server code).

We add a models folder to the src directory, which contains the Measurement schema and a list of measurement types (e.g. temperature and light):

```
// /models/measurement.js

const mongoose = require('mongoose');

const measurementSchema = new mongoose.Schema({
  type: { type: String, required: true },
  value: { type: String },
  timestamp: { type: Date, default: Date.now }
});

measurementSchema.index({ type: 1 });
measurementSchema.plugin(require('mongoose-paginate'));

module.exports = mongoose.model('Measurement', measurementSchema);
```

```
// /models/measurementTypes.js

const MeasurementTypes = Object.freeze({
  temperature: "temperature [Celsius degree]",
  light: "light [%]"
});

module.exports = MeasurementTypes;
```

Notice that we use the popular “mongoose-paginate” plugin that simplify pagination support to Mongoose models. It handles the process of retrieving documents in paged batches, which is essential when dealing with large datasets in APIs or web applications.

We add a database.js file to contain the connection logic to MongoDB. This file is responsible for establishing a connection to the database and exporting the Measurement model for use in other parts of the application:

```
// /database.js

const mongoose = require('mongoose');
const mongoosePaginate = require('mongoose-paginate');

mongoose.Promise = global.Promise;
mongoose.connect(process.env.DATABASE);

mongoose.connection.on('error', (err) => {
  console.error('Database connection error ' + err.message);
});

mongoosePaginate.paginate.options = {
  lean: false,
};

require('./models/measurementSchema');
```

The plugin should be modified to use the Measurement model to store values on the database:

```
// /plugins/temperature.js

const resources = require('../resources/model');
const observer = require("node-observer");
```

```
const mongoose = require('mongoose');
const MeasurementTypes = require('../models/measurementTypes');
const Measurement = mongoose.model('Measurement');

let interval;
const model = resources.iot.sensors.temperature;
const pluginName = resources.iot.sensors.temperature.name;
const unit = resources.iot.sensors.temperature.unit;
let localParams = {'simulate': true, 'frequency': 2000};

// Starts the plugin, should be accessible from other
// files so we export them
exports.start = function (params) {
  localParams = params;
  if (localParams.simulate) { simulate(); }
  else { connectHardware(); }
};

// Stop the plugin, should be accessible from other
// files so we export them
exports.stop = function () {
  clearInterval(interval);
  console.info(`%s plugin stopped!`, pluginName);
};

// Require and connect the actual hardware driver and configure it
function connectHardware() {
  var arduino = require('../hardware/arduino');
  interval = setInterval(function () {
    model.value = arduino.temperature;
    (new Measurement({ type:MeasurementTypes.temperature, value: model.value
      })).save();
  }, localParams.frequency);
  console.info(`Hardware %s sensor started!`, pluginName);
};

// Allows the plugin to be in simulation mode. This is very useful when developing
// or when you want to test your code on a device with no sensors connected, such as
// your laptop
function simulate() {
  interval = setInterval(function () {
    model.value += 1;
    observer.send(this, model.name, model.value);
  });
}
```

```

    (new Measurement({ type:MeasurementTypes.temperature, value: model.value
  ↵ }).save();
  showValue();
}, localParams.frequency);
console.info('Simulated %s sensor started!', pluginName);
};

function showValue() { console.info('%s value = %s %s', pluginName, model.value,
  ↵ unit); };

```

Then we add a new route to handle incoming requests for accessing stored measurements. This route will use the Measurement model to query the database and return the results in a paginated format:

```

// /routes/measurements.js

const express = require('express');
const router = express.Router();
const mongoose = require('mongoose');
const Measurement = mongoose.model('Measurement');

router.route('/').get(async function (req, res, next) {
  try {
    if (!req.query.page) req.query.page = '1';
    if (!req.query.limit) req.query.limit = '10';
    if (req.query.limit > 100) req.query.limit = '100';
    if (!req.query.filter) req.query.filter = '{}';
    if (!req.query.sort) req.query.sort = "{ \"timestamp\": \"desc\" }";
    if (!req.query.select) req.query.select = '{}';
    if (req.query.filter.startsWith("[")) { req.query.filter = "{ \"$or\": " +
      ↵ req.query.filter + " }" };
    const filter = JSON.parse(req.query.filter);
    const options = {
      select: JSON.parse(req.query.select),
      sort: JSON.parse(req.query.sort),
      page: parseInt(req.query.page),
      limit: parseInt(req.query.limit)
    }
    const measurements = await Measurement.paginate(filter, options);
    req.result = measurements;
    next();
  }
}

```

```

    catch (err) { return res.status(500).json({ status: 500, message: err.message })
  ↵  };
};

module.exports = router;

```

Notice that we use the query parameter of the request to filter the measurements and define the pagination.

Finally, we require the database.js file in the entry point of the application to ensure that the database connection is established when the server starts:

```

// Load the database, the http server, the websocket server and the model
const db = require('./database');
const httpServer = require('./servers/http');
const resources = require('./resources/model');
const wsServer = require('./servers/websockets');

// Require all the sensor plugins we need
const ledsPlugin = require('./plugins/ledsPlugin');
const pirPlugin = require('./plugins/pirPlugin');
const tempPlugin = require('./plugins/tempPlugin');
const lightPlugin = require('./plugins/lightPlugin');

// Start them with a parameter object. Here we start them on a
// laptop so we activate the simulation function
ledsPlugin.start({simulate: true, 'frequency': 1000});
pirPlugin.start({simulate: true, 'frequency': 1000});
tempPlugin.start({simulate: true, 'frequency': 1000});
lightPlugin.start({simulate: true, 'frequency': 1000});

const server = httpServer.listen(resources.iot.port, function () {
  console.info('Your WoT Pi is up and running on port %s', resources.iot.port);
});

// Websockets server
wsServer.listen(server);

```

A simple request to the /measurements endpoint will return a paginated list of measurements stored in the database:

```
GET http:{{base_url}}/iot/measurements?filter={"type":"temperature [Celsius  
→ degree]"}&limit=2&page=6
```

The response will include the total number of documents, the current page, and the number of documents per page, along with the actual measurement data. This allows us to easily navigate through large datasets without overwhelming the client or server:

```
{  
  "docs": [  
    {  
      "_id": "682de6356a75fb61029c6c09",  
      "type": "temperature [Celsius degree]",  
      "value": "2",  
      "timestamp": "2025-05-21T14:41:57.222Z",  
      "__v": 0  
    },  
    {  
      "_id": "682de6346a75fb61029c6c07",  
      "type": "temperature [Celsius degree]",  
      "value": "1",  
      "timestamp": "2025-05-21T14:41:56.224Z",  
      "__v": 0  
    }  
,  
    "total": 12,  
    "limit": 2,  
    "page": 6,  
    "pages": 6  
}
```

## 9.5 Hands-on Activity

Create a user collection and populate it with several user documents, each containing fields such as name, surname, phone, address, gender, and birthDate. Practice using different query selectors with the find method to retrieve specific subsets of users—for example, all female users or users under 40 years old. Experiment with additional MongoDB operations like count and remove to explore document statistics and deletion.

Extend the WoT Proxy by defining a new schema for log entries. Each time the API receives a request, a new log document should be created and stored in the database. Add a new route that exposes the logs, making it possible to retrieve them via an HTTP request.

## 10 Security

In the **early stages** of IoT deployment, most systems were conceived as **isolated clusters of devices** operating within **closed environments**. These environments, often termed **Intranets of Things**, were typically confined to specific domains, such as a single building, factory, or enterprise. Within such contexts, devices communicate internally, shielded from the external internet, and access is restricted to a small set of authorized users or systems. This model is still prevalent in many mission-critical settings, such as security infrastructures in financial institutions or production lines in large-scale industrial facilities, where exposure to the wider internet may introduce unacceptable risks.

The WoT idea extends IoT by introducing a layer of abstraction that allows devices and services to be accessed and managed using familiar web technologies. We move **beyond isolated intranets** and begin connecting devices, applications, and services on a global scale. WoT promotes **openness** and **interoperability** by defining standard ways to describe and interact with devices over the Web. This shift allows individual silos to be integrated, enabling seamless communication between heterogeneous systems. Rather than treating each domain as an isolated island, the WoT architecture provides a framework through which devices can be made discoverable, queryable, and composable—across networks and organizational boundaries.

This transition **from intranet to internet** is not driven by technological curiosity alone. It reflects a deliberate effort to **unlock the value of interconnected systems**. On one hand, there are clear cases where keeping IoT systems isolated is necessary, such as when dealing with proprietary or sensitive infrastructures that must be protected against cyber threats. On the other hand, the ability to expose certain types of information, particularly public or non-sensitive data, can lead to significant social and economic benefits. A compelling example can be found in government initiatives like **data.gov**, where open access to real-time environmental data (such as traffic conditions, air quality, or weather patterns) supports a wide variety of services and research efforts. Imagine a distributed network of sensors monitoring a volcano or rainforest. Making that data publicly available allows developers, researchers, and policymakers around the world to benefit from it, fostering innovation in domains ranging from disaster response to climate science.

In this scenario, **ensuring the security of the data** handled by the proxy is essential. Data must remain **confidential during transmission**, **must not be altered either accidentally or maliciously**, and **must originate from and be delivered to trusted parties**. Moreover, system administrators must be able to **track and verify the actions** performed within the system for reasons of **accountability**.

Node.js does not enforce any security practices by default. Developers must proactively integrate **protection mechanisms**, especially when building applications intended to operate within

open or untrusted environments.

Developing a **secure WoT proxy** involves addressing a variety of threats. These range from external attacks, such as unauthorized access over the network, to internal issues, such as the incorrect storage of sensitive data. Security concerns must be addressed throughout the entire lifecycle of a request, from the moment it is received by the proxy, through its validation and processing, to the storage or forwarding of its associated data. This includes implementing secure **authentication mechanisms, managing user privileges, sanitizing all external inputs, encrypting communication channels, and keeping audit records** for all significant events.

## 10.1 The Share Layer

As the WoT expands beyond private networks into public and semi-public spaces, the challenge shifts from simply connecting devices to managing access in a nuanced and context-aware fashion. This complexity is often referred to as the **share layer**, a conceptual boundary that defines **how, when, and under what conditions** devices, data, and services should be **made available** to users.

To illustrate the need for such a layer, consider the hotel example we have introduced in the first chapter. The connected hotel provides devices in each room (such as smart lights, thermostats, or entertainment systems) all accessible via a mobile application or web interface. However, the correct behavior of the system depends on **enforcing a very specific access policy**: only the guest currently checked into a room should be able to control those devices, and only for the duration of their stay. Any lapse in this access control (such as allowing a former guest or unauthorized user to interact with those devices) could lead to privacy violations or even physical security risks.

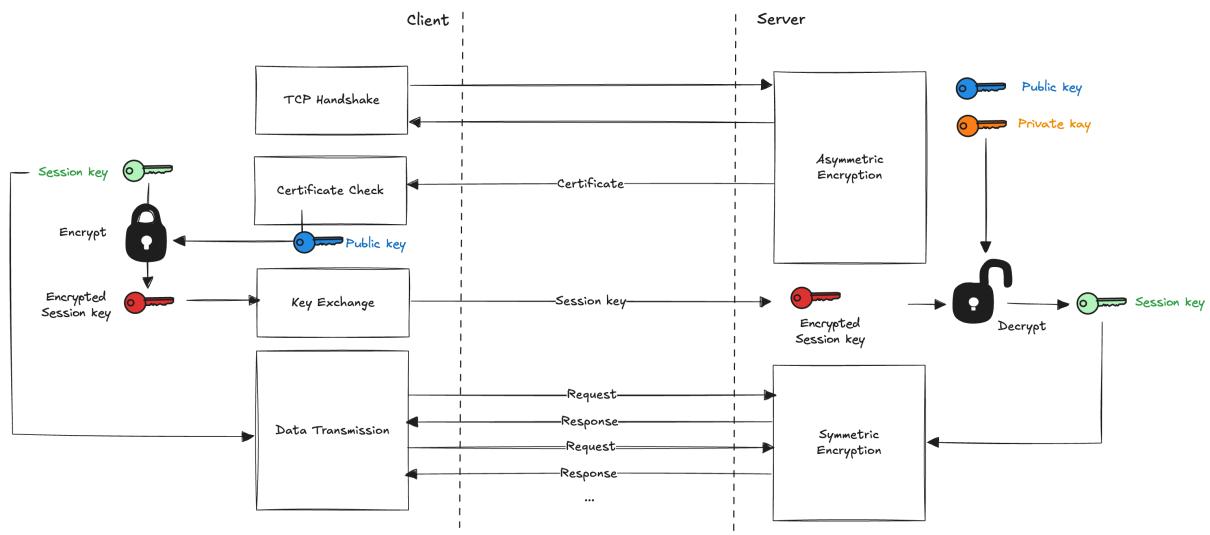
This example reflects a broader shift in public infrastructure, which is becoming not only digitally enabled but increasingly pervasive. Sensors, actuators, and services are embedded in everyday environments (from transportation systems to urban infrastructure) creating **vast networks of interconnected components**. These systems often span multiple administrative domains and serve diverse user groups. Consequently, access cannot be governed by static rules alone. It must be **dynamic, contextual, and secure by design**.

To support this evolution, we must learn to build systems that are not only **functionally robust**, but also **securely deployable** and capable of **enforcing access controls** at multiple levels. The technical architecture must ensure that different users, such as citizens, employees, or guests, are able to access or share data and services selectively and in a safe and controlled manner. This includes enforcing **temporal constraints** (access allowed only during certain periods), **scoping**

**access to specific resources** (such as a room's thermostat or a building's front door), and **ensuring that all operations follow predefined behavioral rules**.

This requirement raises important questions. How can we technically guarantee that only a well-defined subset of users can access a given resource? How can we restrict this access to only a narrow window of time, or to a particular geographic location? How can we ensure that data sharing occurs without compromising the privacy or integrity of the system as a whole?

Addressing these questions will require a blend of **authentication, authorization, session management, and policy enforcement mechanisms**. Even in the simplest interaction (such as a user adjusting the temperature of a smart heater via a mobile app), several critical security challenges emerge. These challenges form the foundation for understanding how attacks can compromise the system and how a secure design must address each of them explicitly:



### 10.1.1 Problem 1: Confidentiality

The first problem arises from the openness of the network itself. Whenever data is sent from a client device (like a mobile app) to a smart appliance, there is a\*\* risk that the message could be intercepted in transit. **This is especially true if the communication is not encrypted\*\*** or if secure channels like TLS are not properly configured. An attacker capable of **sniffing network packets** can eavesdrop on this communication, capturing sensitive commands or user data. The attacker might not even need to understand the full content of the message; simply knowing the structure or observing patterns might be enough to reconstruct behavior or launch further attacks. Therefore, ensuring **confidentiality** of the communication is the first challenge to address.

### 10.1.2 Problem 2: authenticity

The second problem involves **authenticity**. From the user's perspective, it is crucial **to be certain that the device they are interacting with is indeed the intended one**. If the system is vulnerable to **spoofing**, a malicious actor could pose as the heater and deceive the mobile app into sharing private data or accepting invalid responses. This sort of impersonation attack can lead to credential theft or unauthorized data collection. **Trust in the identity** of connected devices is therefore essential. This can be achieved through mechanisms such as mutual TLS authentication, public key infrastructure, or digital certificates that verify the origin of each endpoint.

### 10.1.3 Problem 3: authorization

The third issue mirrors the second but from the opposite perspective. The device itself must be able to **verify that the request it receives genuinely comes from an authorized user**. In the heater example, it must ask whether the request actually came from Lena, and more importantly, whether Lena has permission to execute that specific action. This is the problem of **authorization**, where the system needs to enforce strict control over what operations each authenticated user is allowed to perform. Without proper checks, any user (or even any attacker who manages to forge a request) might be able to manipulate the heater, causing both comfort and security concerns.

Together, these three problems (confidentiality of the communication, authenticity of the devices, and authorization of the actions) form the triad of challenges that must be addressed in every secure WoT application. Only by systematically addressing each of these aspects can we ensure that WoT proxies behave as intended in complex, real-world environments.

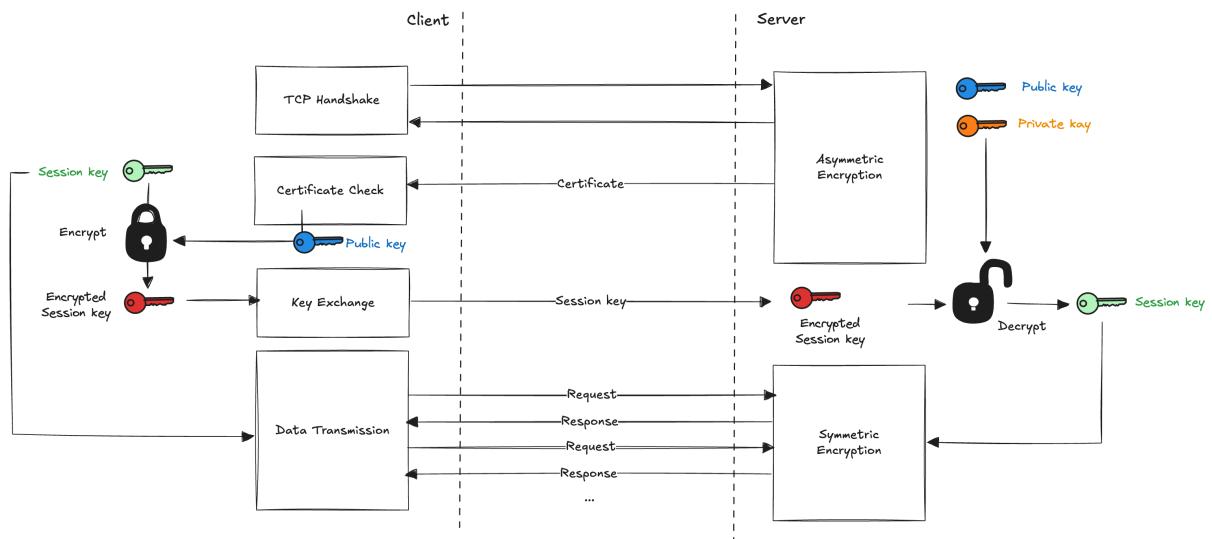
### 10.1.4 Encryption

At the heart of any secure system lies **encryption**. Without it, all other security mechanisms are ineffective. Any message sent over a network is **vulnerable to interception**. Attackers equipped with packet sniffers can capture unencrypted communications, gaining insight not only into the content of those messages but also into the structure and logic of the system itself. In such a case, even well-designed authentication and authorization layers can be undermined. Encryption is therefore not optional: it is the foundational requirement upon which every other security layer must be built.

There are two main types of encryption: symmetric and asymmetric. In **symmetric encryption**, both the sender and the receiver **share a single secret key**. This key is used to encode the message before transmission and to decode it upon receipt. The main advantage of this approach is

performance, symmetric encryption algorithms are typically fast and require few computational resources, making them suitable for constrained devices. However, symmetric encryption introduces a significant challenge: **how to share the secret key securely** in the first place. If the key is exposed during this exchange, the security of the entire system collapses.

**Asymmetric encryption** offers a different model that avoids the key exchange problem. Instead of using a single key, it relies on a **pair of keys**: a **public key** and a **private key**. These keys are mathematically related, but knowledge of the public key does not allow an attacker to derive the private key. This allows the **public key to be freely distributed**, while the **private key remains securely stored** on the device. When a mobile app wants to send a command to the heater, it encrypts the message using the heater's public key. Once encrypted, the message becomes unintelligible to anyone except the heater, which can decrypt it using its own private key:



This model offers strong guarantees: even if an attacker intercepts the encrypted message in transit, they cannot decipher its content without access to the corresponding private key. In practice, most secure communications, such as those using **HTTPS**, combine both symmetric and asymmetric encryption. The asymmetric portion is used to securely exchange a temporary symmetric key, which is then used for the actual data transfer. This **hybrid approach** balances the performance benefits of symmetric encryption with the safety of public-key mechanisms.

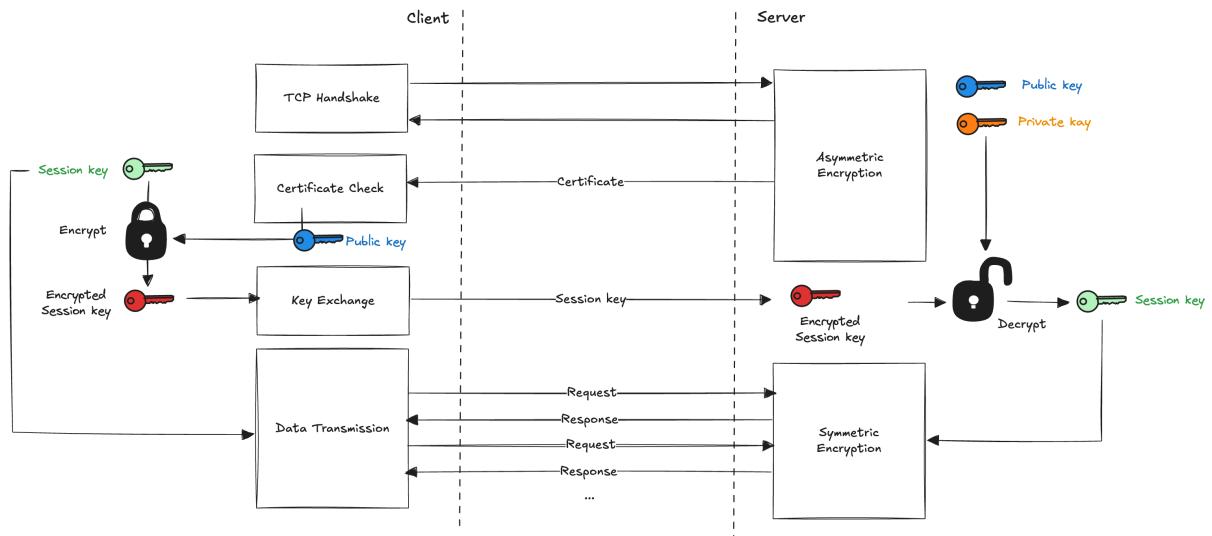
## 10.2 Transport Layer Security (TLS)

**Transport Layer Security (TLS)** is the protocol behind the "S" in **HTTPS**, and it ensures that communication between client and server remains private and secure. Before TLS became dominant, its predecessor Secure Socket Layer (SSL) was used extensively. However, **vulnerabilities** such as the infamous **POODLE** (Padding Oracle On Downgraded Legacy Encryption) attack

led to the deprecation of SSL around 2014, and TLS has since become the standard for encrypted web traffic.

TLS plays two crucial roles in secure communication. The first is **authentication**: the protocol ensures that the client is genuinely communicating with the intended server, and not with an impersonator or a malicious intermediary. The second role is **encryption**: once the server's identity has been verified, TLS guarantees that all data exchanged between client and server is encrypted. This means that even if an attacker intercepts the data packets, they will be unable to read or tamper with their content. Together, these two roles form the backbone of secure data exchange over the internet.

The process of establishing a secure TLS session involves a series of steps, typically referred to as the **TLS handshake**. It begins when the client contacts the server and declares the protocols and encryption algorithms it supports. This negotiation is conceptually similar to how clients and servers agree on content types during HTTP communication. The server then responds by sending its **digital certificate**, which contains its public key. This certificate is issued and signed by a **trusted Certificate Authority (CA)**, allowing the client to verify the server's identity against a list of known and trusted authorities. Once the client accepts the certificate as valid, it generates a session key, encrypts it using the server's public key, and sends it back. Only the server can decrypt this key, because only it holds the corresponding private key. From this point forward, both client and server use the **shared session key** to encrypt and decrypt the messages exchanged during their session:



The success of this mechanism depends heavily on correct configuration and key management. In the context of a WoT proxy, using TLS means configuring our server to support HTTPS, generating and installing valid certificates, and ensuring that all API endpoints are exposed over secure connections only.

### 10.2.1 Certificate

To generate a **certificate** we can use tool like the **OpenSSL command-line tool**. It allows us to create both the private key and the certificate that will be used by the server. The command typically used is:

```
openssl req -x509 -newkey rsa:2048 -keyout privateKey.pem -out caCert.pem -days 1095
↪ -sha256
```

This command creates a new **RSA key pair** (where RSA means Rivest-Shamir-Adleman, a widely used cryptosystem, based on the difficulty of factoring large composite numbers into their prime factors, which is computationally infeasible for sufficiently large key sizes) with a size of 2048 bits. The private key is saved in a file named **privateKey.pem**, which **must be kept secret**, as it allows the server to decrypt incoming messages and prove its identity. The certificate, saved in **caCert.pem**, contains the **public key** and **additional metadata**. It is signed using **SHA-256** (Secure Hash Algorithm 256-bit, a cryptographic hash function) and remains valid for 1095 days (approximately three years). During certificate creation, we will be prompted to enter details such as country, organization, and common name (typically the domain name of the server). These fields are embedded into the certificate and help identify it during the handshake phase.

Once these two files are created, the Node.js server can be configured to use them by importing the **https module** and reading the certificate files from disk. This replaces the standard `http.createServer()` call with `https.createServer()`, passing in the key and certificate as options. Let's modify the WoT Proxy code in order to use HTTPS:

```
// /server/http.js

const https = require('https');
const fs = require('fs');

// Read the certificate and the private key
const cert_file = './resources/caCert.pem'; // The certificate
const key_file = './resources/privateKey.pem'; // The private key
const passphrase = 'pippo'; // The password of the private key

const config = { key: fs.readFileSync(key_file),
                 cert: fs.readFileSync(cert_file),
                 passphrase: passphrase
               };
```

```
const app = express();

app.use(bodyParser.json());

app.use('/iot/actuators', actuatorsRoutes);
app.use('/iot/sensors', sensorRoutes);
app.use('/iot/measurements', measurementRoutes);

app.get('/iot', function (req, res) { res.send('This is the WoT API!') });

app.use(converter());

// Create the server using the certificate and the private key
const server = https.createServer(config, app);

module.exports = server;
```

The same approach applies to WebSocket servers, which must be upgraded to run over **WSS (WebSocket Secure)** rather than plain WS. These changes ensure that all communication to and from the server is encrypted using TLS. This is done just by passing to the WebSocket constructor the HTTPS server instance instead of the HTTP one. The WebSocket server will then automatically inherit the TLS configuration from the HTTPS server, ensuring that all WebSocket connections are also encrypted.

### 10.2.2 Authorities

After the server has been launched with HTTPS enabled, clients connecting to it will attempt to verify the certificate. Because the certificate is **self-signed** and not issued by a recognized **Certificate Authority**, clients will display a **warning**. This is expected behavior during local development and testing. It confirms that the server is correctly serving content over HTTPS, even though **it cannot be fully trusted** by the browser. While encryption ensures that data transmitted between devices remains confidential, it does not by itself guarantee the identity of the parties involved. This is the key limitation of self-signed certificates: although **they can encrypt data effectively, they do not establish trust**. In a secure communication system, trust must be rooted in verifiable identity. Without that verification, there is no assurance that the server a client connects to is truly the intended one. In other words, the **chain of trust** is broken.

This becomes a serious issue on the public Internet, where attackers can easily impersonate services and intercept traffic. For example, a malicious actor could pose as a WoT device, lure a

user into initiating a connection, and then intercept sensitive information or manipulate device behavior. In such cases, even if the data is encrypted, the attacker could still cause harm simply by being mistaken for a legitimate device. This kind of impersonation cannot be prevented using self-signed certificates, because the client has no way to verify the origin of the certificate.

In contrast, certificates issued by trusted **Certificate Authorities (CAs)** provide a solution to this problem. CAs are organizations that validate the identity of entities and issue certificates that can be verified using a public trust infrastructure. When a browser or client connects to a server presenting a CA-issued certificate, **it checks whether the certificate was signed by a trusted authority**. If the signature is valid and the certificate matches the expected identity, the client proceeds with the connection. If not, it raises a warning or blocks the communication altogether. Obtaining a certificate from a trusted CA often involves a **verification process**, which can vary in rigor. Some well-known CAs (such as DigiCert, Sectigo, or GlobalSign) offer **paid services** where the applicant's identity is verified before a certificate is issued. This verification may include domain ownership checks, organizational validation, or even manual review. However, the process is not free, and the cost can be prohibitive for small-scale projects or individual developers. The business of issuing certificates has, unfortunately, led to a market where not all authorities apply the same standards. Some low-cost CAs have been known to issue certificates with minimal validation, increasing the risk of misissued certificates and weakening the overall security model. This variability is why choosing a reputable CA is essential when deploying services to the public web.

### 10.2.3 Let's Encrypt

To address the need for secure certificates that are freely available, **Let's Encrypt** was established as a nonprofit initiative. It provides **domain-validated certificates at no cost**, helping **democratize access to HTTPS and TLS security**. Let's Encrypt is supported by a broad coalition of major web actors and is now widely trusted by modern browsers. Through **automated tools** like **Certbot**, developers can request, install, and renew certificates with minimal effort, making it an ideal choice for securing the WoT Proxy in production.

By using a CA-issued certificate from a reputable provider, developers ensure that their WoT proxies are both **encrypted** and **identifiable**. This enables clients to verify the identity of the server they are connecting to and builds the foundation for trust in the system.

## 10.3 Authentication and Access Control