HTTP

Prof. Riccardo Berta

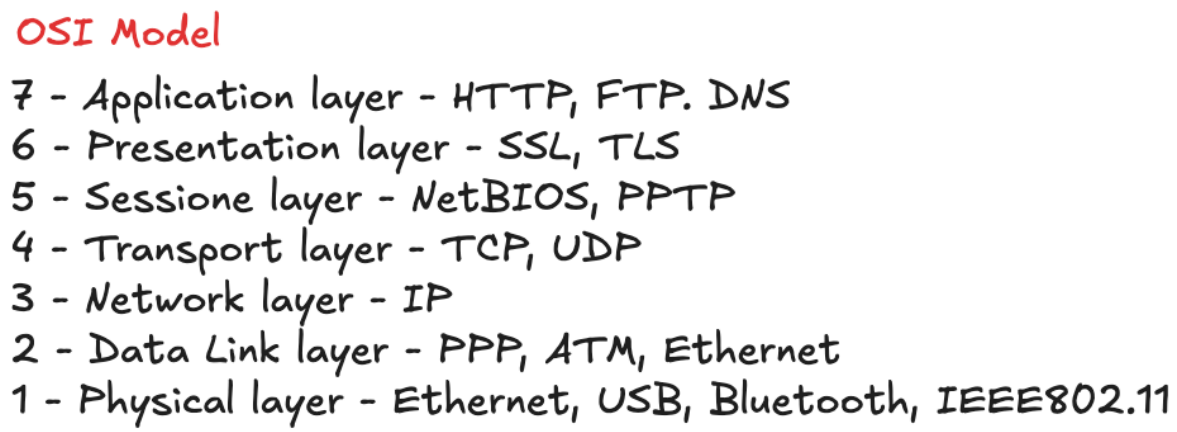
2025-04-04

Contents

1 HTTP **2**
1.1 Basic features . . . 3
1.1.1 Resources and URLs . . . 4
1.1.2 Methods . . . 6
1.1.3 Status Codes . . . 7
1.1.4 Headers . . . 8
1.1.5 Message Body . . . 9
1.1.6 Example . . . 10
1.2 Content Negotiation . . . 11
1.3 Caches . . . 12
1.3.1 Client-side caching . . . 13
1.3.2 Server-side caching . . . 13
1.3.3 Cache-Control headers . . . 14
1.4 Conditionals . . . 15
1.4.1 Last-Modified and If-Modified-Since headers . . . 15
1.4.2 ETag and If-None-Match header . . . 16
1.5 Authentication . . . 16
1.5.1 Basic Authentication . . . 16
1.5.2 Digest Authentication . . . 17
1.5.3 Bearer Token Authentication . . . 17
1.5.4 Certificate-Based Authentication . . . 18
1.5.5 Server response . . . 18
1.6 Cross-Origin Resource Sharing (CORS) . . . 18
1.6.1 Access-Control-Allow-Origin header . . . 19
1.6.2 Other CORS headers . . . 19
1.7 JSON . . . 20
1.7.1 Structure . . . 21
1.7.2 Arrays . . . 22
1.7.3 Nester structures . . . 22
1.7.4 Schema . . . 23
1.8 Interacting with HTTP . . . 24
1.8.1 From the Browser . . . 25
1.8.2 From Javascript . . . 26
1.8.3 Postman . . . 27
1.9 Hands-on Activity . . . 29

1 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.



As illustrated in the image, HTTP is an **application-layer protocol** of the OSI model an 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)

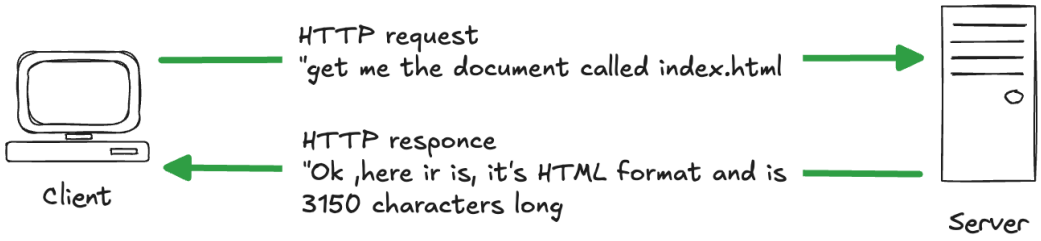
At its foundation, HTTP operates over the TCP/IP protocol suite, leveraging TCP (Transmission Control Protocol) for reliable, ordered data transmission across the internet. TCP, in turn, relies on IP (Internet Protocol) for addressing and routing, ensuring efficient data delivery. Through this architecture, 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

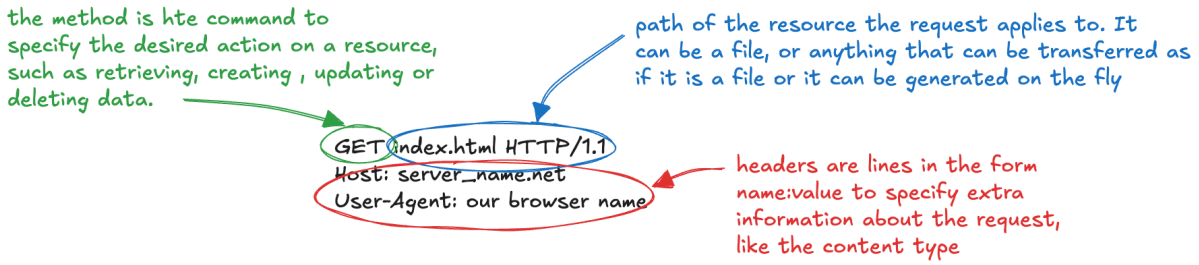
By adhering to this standardized framework, HTTP ensures interoperability, allowing diverse systems to communicate seamlessly. While it typically operates over TCP port 80, it can function on other ports as required. Understanding HTTP is crucial in the Web of Things because it enables seamless communication between interconnected devices, allowing them to exchange data, interact with web services, and integrate with cloud platforms using standardized, web-friendly protocols.

1.1 Basic features

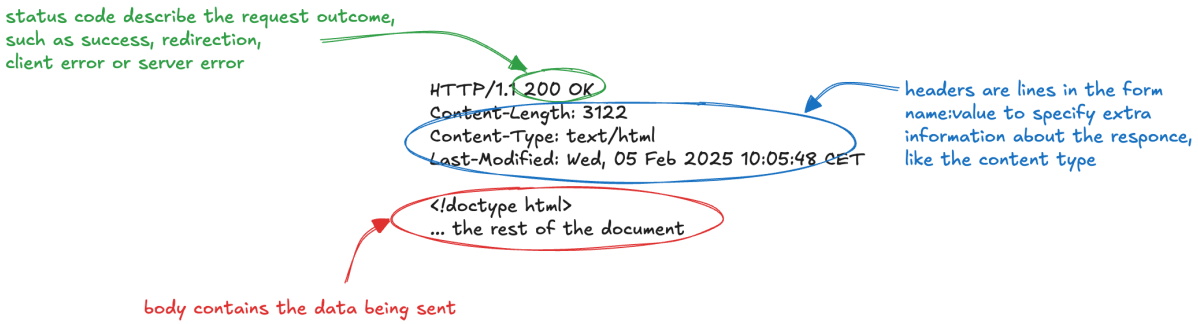
HTTP is base on a **client-server architecture**. A client, such as a web browser, mobile application, or API client, initiates a request, while a server, typically hosting a website or web service, processes the request and returns a response.



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:

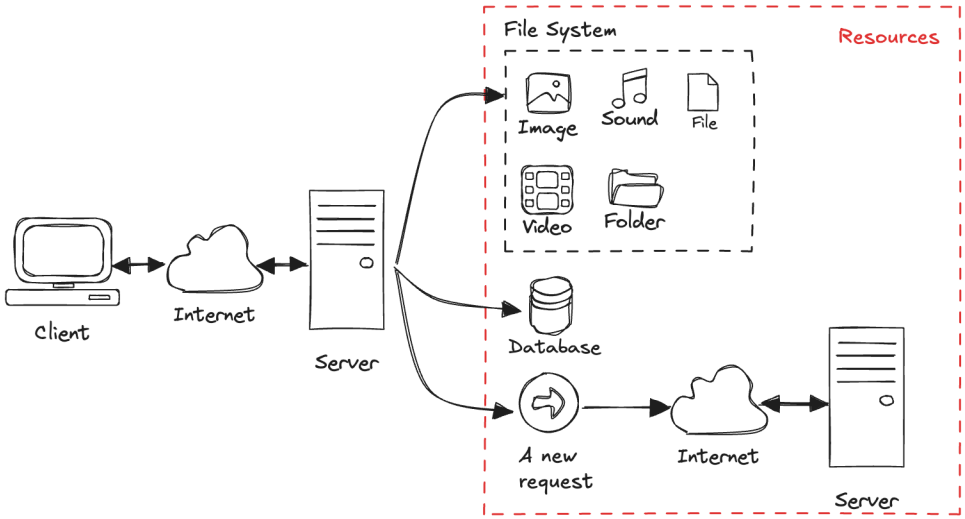


A defining characteristic of the HTTP client-server model is its **statelessness**. Once the server processes the request and sends back a response, the connection is closed, and the client disconnects. The client and server are only aware of each other for the duration of that specific request-response interaction. Moreovere, 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.

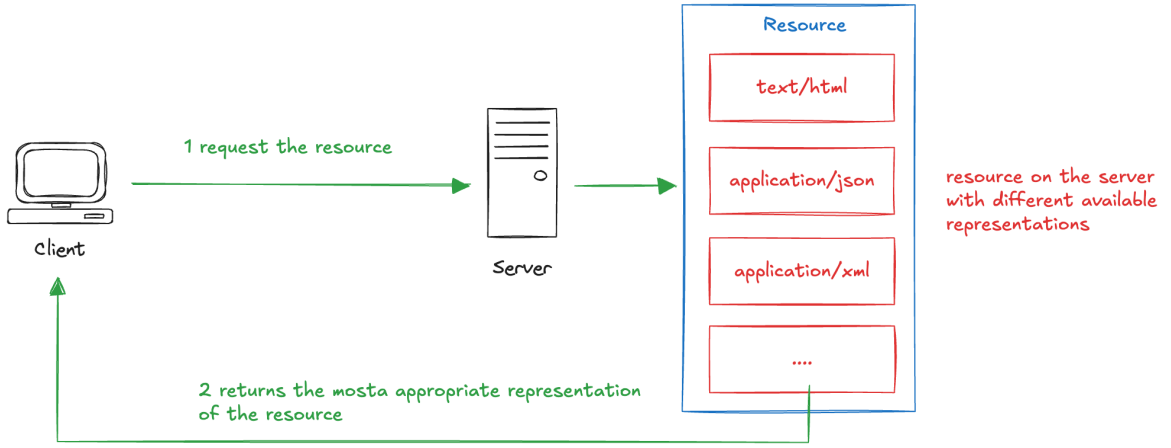
This architecture underpins everything from simple web browsing to complex API-driven applications and cloud services, making it a cornerstone of modern internet communication

1.1.1 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:



1. **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.
2. **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.

Somethimes 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).

1.1.2 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.

1.1.3 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.

1.1.4 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 trasmitted. 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 example 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.

1.1.5 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.

1.1.6 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:

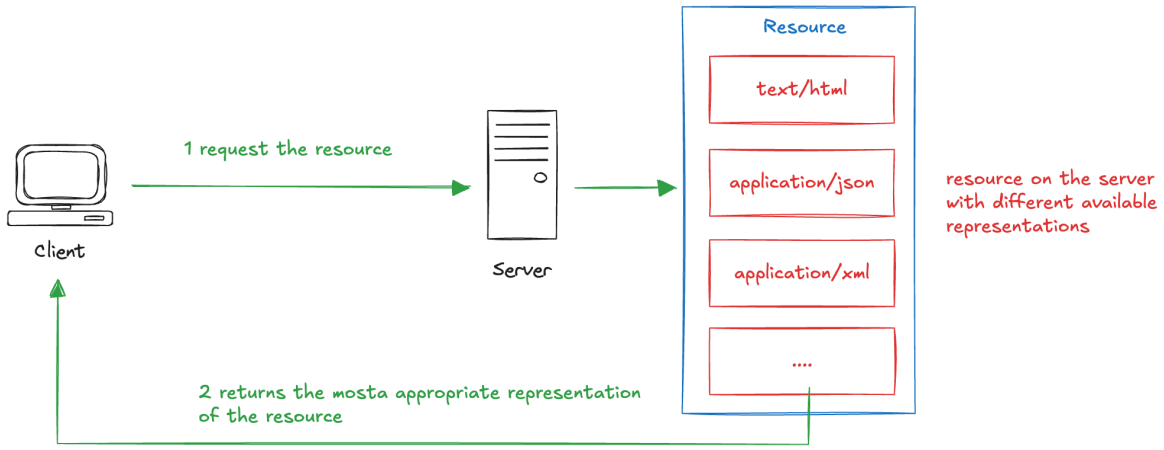
```
HTTP/1.1 200 OK
Date: Wed, 05 Feb 2025 15:30:00 GMT
Server: Apache/2.4.41 (Ubuntu)
Content-Type: text/html; charset=UTF-8
Content-Length: 123

<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.

1.2 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:

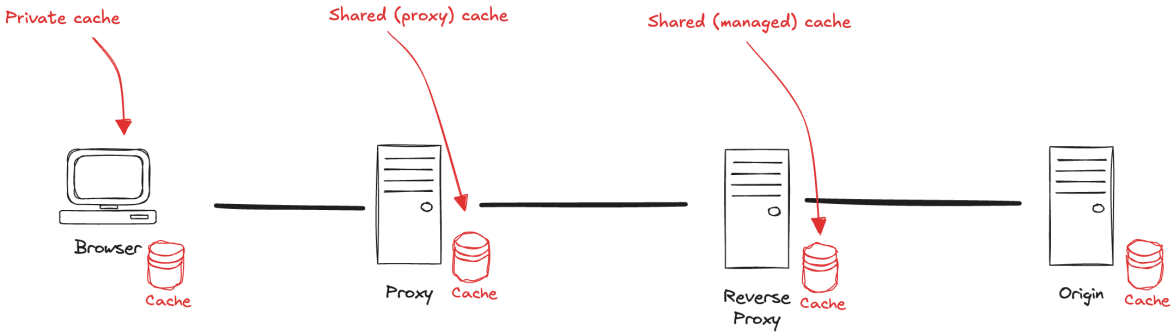
```
HTTP/1.1 200 OK
Date: Wed, 05 Feb 2025 15:45:00 GMT
Server: ExampleServer/1.0
Content-Type: application/json
Content-Length: 85

{
  "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.

1.3 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.

1.3.1 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.

```
1  Cache-Control: private
```

1.3.2 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
```

1.3.3 Cache-Control headers

The Cache-Control header is used to control caches in clients and intermediate services. Several directives are used to specifiy 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.

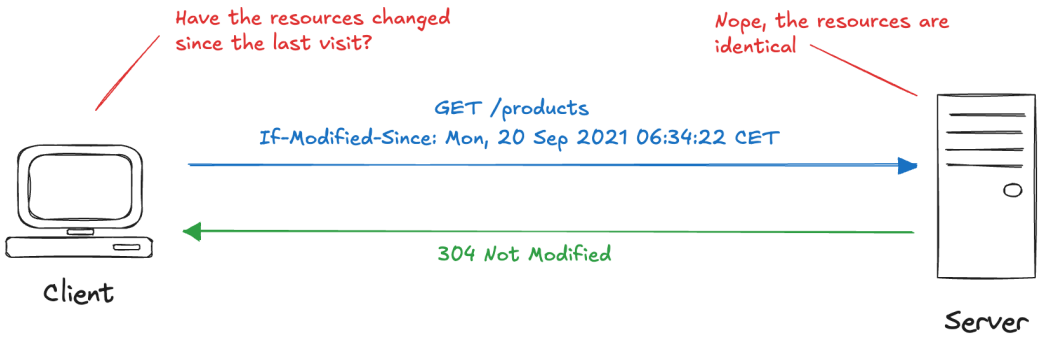
1.4 Conditionals

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:

1.4.1 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.

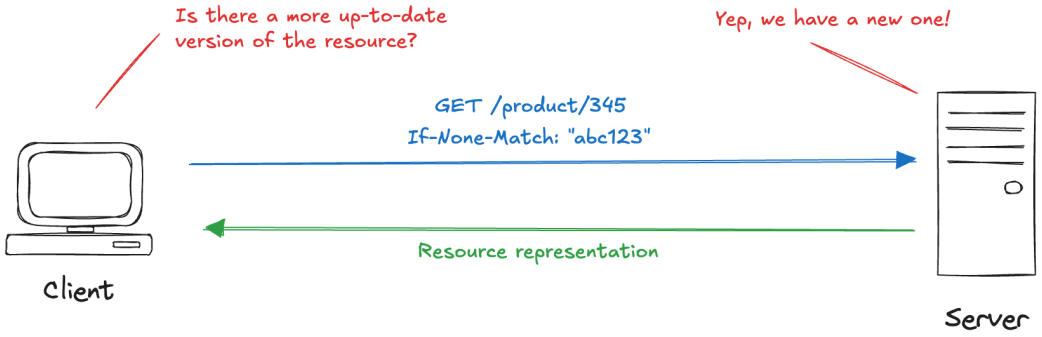


1.4.2 ETag and If-None-Match header

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.



1.5 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:

1.5.1 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.

1.5.2 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.

1.5.3 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:

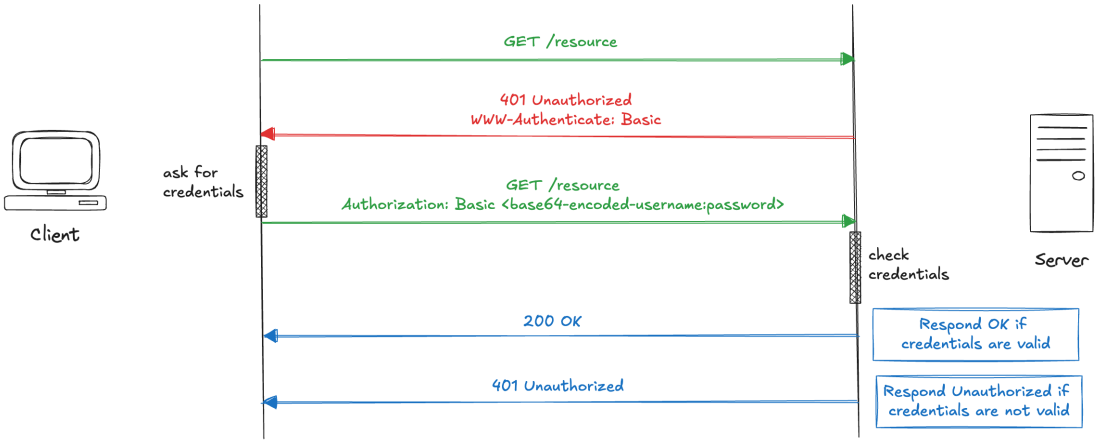
The server uses the token to verify the client's identity and grant or deny access to the requested resource. This method is widely used for web services, as it enables users to **authenticate without exposing their password**. The server doesn't need to know who the client is beforehand. If the client 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.

1.5.4 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.

1.5.5 Server response

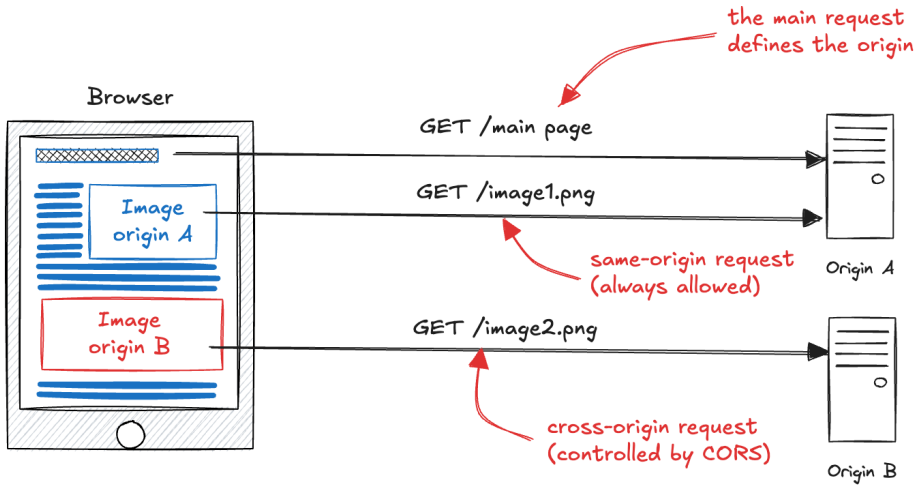
When a protected resource is requested, the server sends an **HTTP 401 Unauthorized** response, prompting the client to authenticate. The server includes a **WWW-Authenticate header** specifying the authentication methods it supports. The client then responds with the appropriate Authorization header containing the credentials or token. If the credentials are valid, the server grants access to the resource.



1.6 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 is 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 do-

main wants to fetch data from a service on another domain. CORS provides a mechanism for servers to grant permissions for such cross-origin requests.



1.6.1 Access-Control-Allow-Origin header

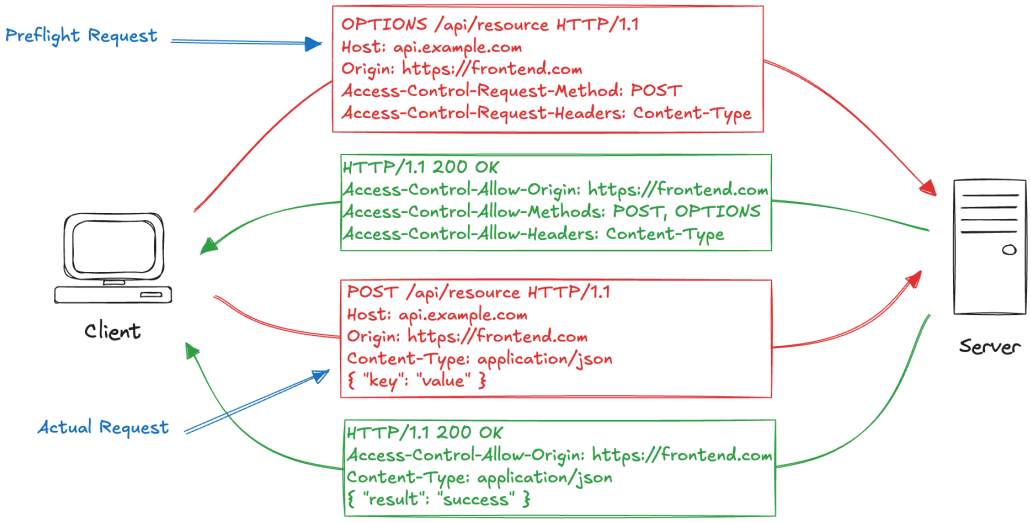
The **Access-Control-Allow-Origin** header is used by servers to specify which origins are permitted to access their resources. It can be set to different values depending on the desired level of access. When set to *****, it allows requests from any origin, making the resource publicly accessible, which 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.

1.6.2 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 sends 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 exmaple 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.

1.7 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 com-

monly 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.

1.7.1 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.

1.7.2 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 wou 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"
```

1.7.3 Nester 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": "2025-02-06T10:15:00Z", "value": 22.4 }
    ],
    "location": {
      "room": "living room",
      "floor": 1
    }
```

```
11    }
```

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.

1.7.4 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
      },
      "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": "<=",
      "limit": 150,
      "exclusive": false
    },
    "message": "should be <= 150"
  }
]
```

1.8 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.

1.8.1 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:

```
<img src="logo.png" alt="Logo">
<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.

1.8.2 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.

1.8.3 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.

1.9 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”.