Week 1 - Basic Terminologies of an App

L1:1: *What is an App ?*

Desktop Apps

Usually standalone

- Editors / Word processors (MS word)
- Web Browser (Chrome, firefox)
- Mail (Outlook, Apple mail, thunderbird)

Often work offline

- Local data storage
- Possible network connection

Software Dev Kits (SDK)

- Custom frameworks
- **OS specific**

Mobile Apps

Targeted at mobile platforms: phones / tablets

- Example: Twitter app, Instagram, amazon, etc...

Constraints

- Limited screen space
- Memory / processing
- Power

Frameworks

- OS specific
- Cross-platform

Network!

- Usually network oriented

Web Apps

The Platform

Works across OS, device: create a common base

Heavily network dependent

- Workarounds for offline processing

L1.2: *Components of an App*

Exmaple: email client

Storage

- Where are the emails stored ?
- How are they stored on the server ? File formats etc...

Compute

- Indexing of emails
- Searching

Presentation

- Display list of mails
- Rendering/display of individual mails

Platform features

Desktop

- Keyboard, mouse, video
- Desktop paradigm - folders, files, documents

Mobile

- Touchscreen
- Voice, tilt, camera interfaces
- Small self-contained apps

Web-based

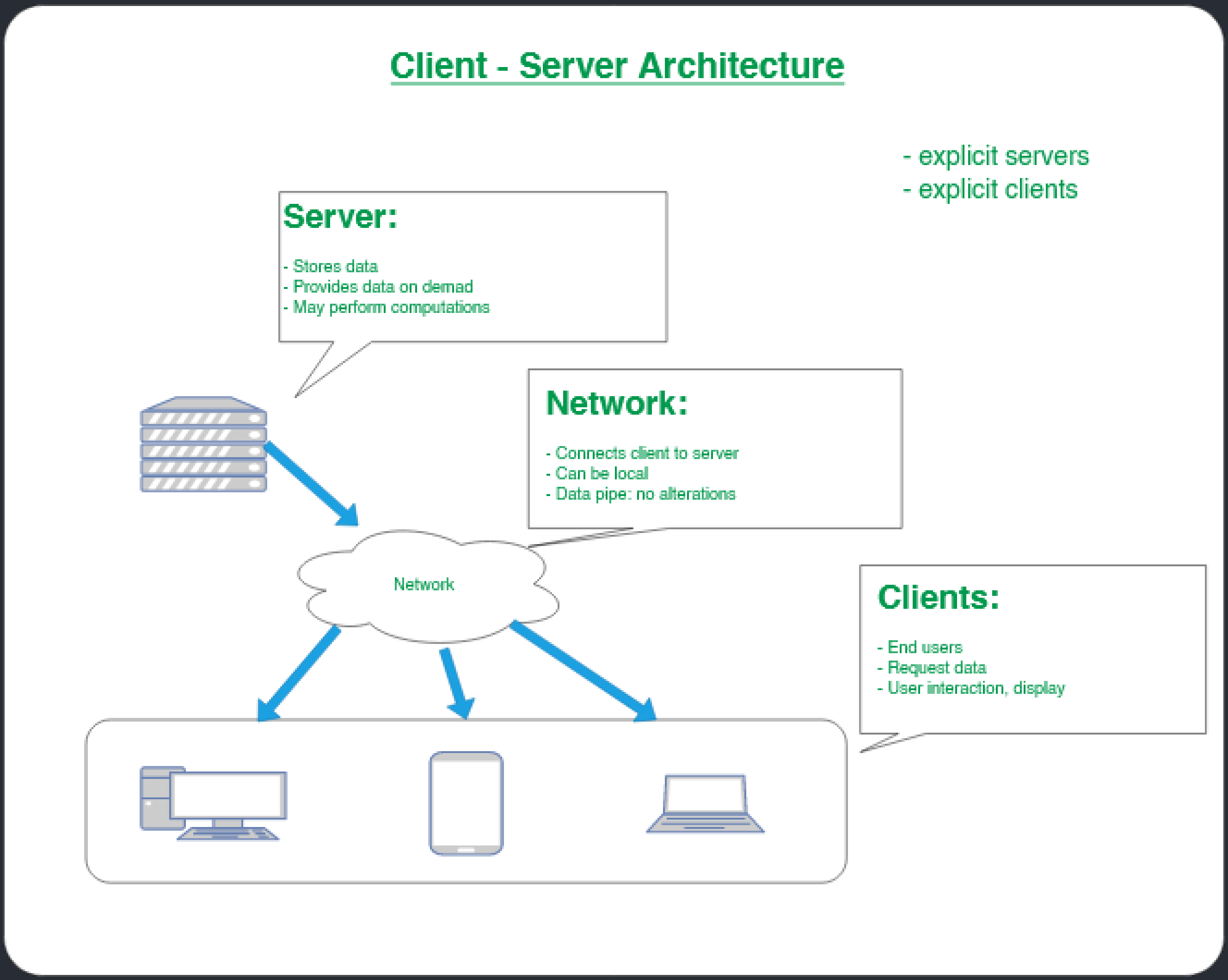
- Datacenter storage - persistent
- Cloud: access anywhere, multi-device

Embedded

- Single function, limited scope
- Example: Watch, Camera (they are acessible separately, but are embedded in mobiles)

L1.3: *Client Server and Peer-to-Peer Architecture*

Client Server Architecture



Explicit differentiation between clients and servers

Local systems:

- both client and serer on same machine - local network / communication
- conceptually still a networked system

Machine clients

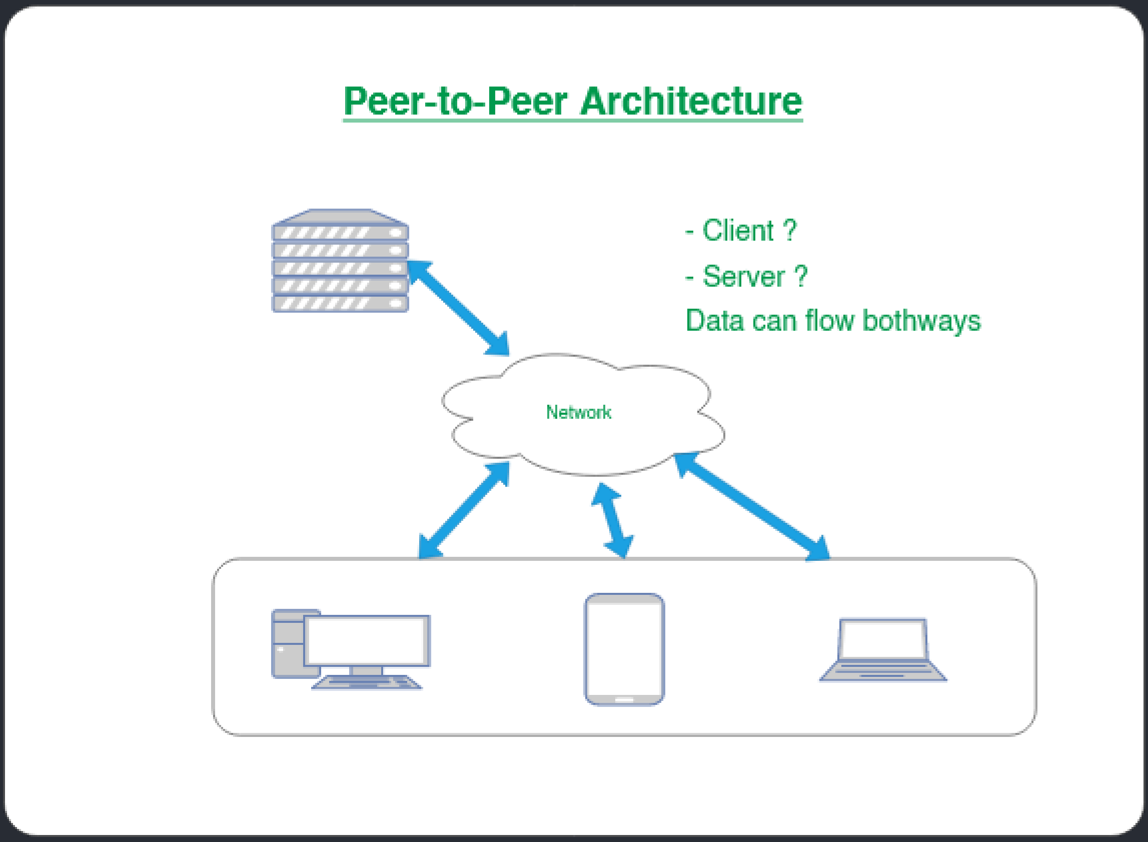
- Eg: Software / antivirus updaters
- Need not have user interaction

Variants

- Multiple servers, single queue, multiply queues, load balancing frotends

> **Example:**
>
> - Email
> - Databases
> - WhatsApp / messaging
> - Web browsing

Peer-to-Peer Architecture (distributed model)


Peer-to-Peer Architecture

- Client ?

- Server ?

Data can flow bothways

All peers are considered "equivalent"

- But some peers may be more equal than others...

Error tolerance

- Masters / introducers
- Election / re-selection of masters on failure

Shared information

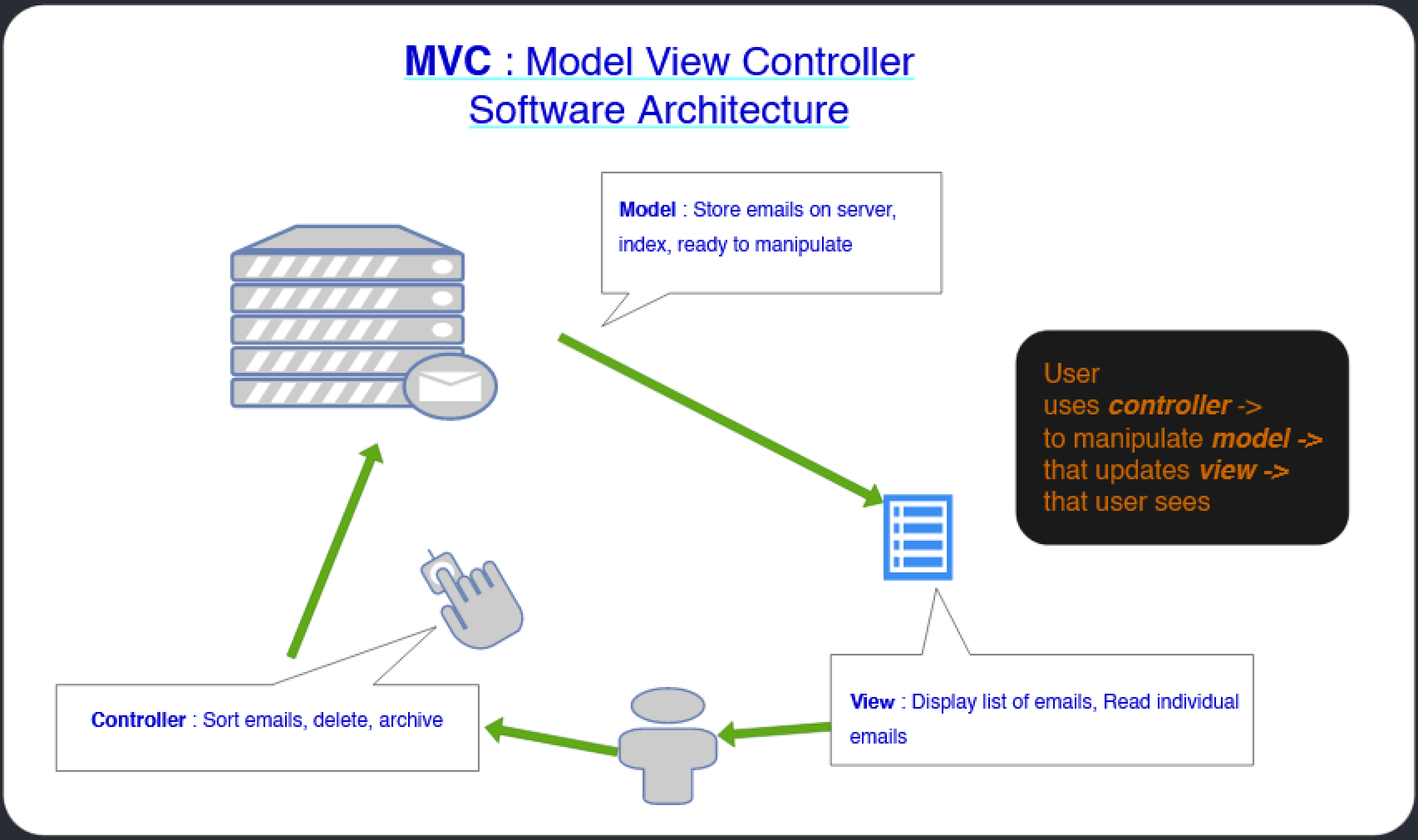
> **Examples:**
>
> - Bittorrent
> - Blockchain-based systems
>
> IPFS, Tahoe (distributed file systems)

L1.4: *Software Architecture Patterns*

What is a design pattern ?

A general, reusable solution to a commonly occuring problem within a given context in software design.

M-V-C paradigm


MVC : Model View Controller
Software Architecture
Model : Store emails on server, index, ready to manipulate
User uses controller -> to manipulate model -> that updates view -> that user sees
View : Display list of emails, Read individual emails
Controller : Sort emails, delete, archive

Model

The model represents the data of the application. It contains the business logic and rules for how the data is stored and manipulated.

View

The view is responsible for displaying the data to the user. It renders the data into a graphical user interface (GUI) that the user can interact with.

Controller

The controller mediates between the model and the view. It receives user input from the view and updates the model accordingly. It executes the business logic of the application.

Example: *To-do application*

Here is an example of how the MVC pattern can be used to build a simple todo list application.

- **Model**: The model would contain a list of todo items. Each todo item would have a title, a description, and a status (e.g., "pending", "in progress", or "complete").
- **View**: The view would be a web page that displays the list of todo items. The user would be able to add, edit, and delete todo items from the view.
- **Controller**: The controller would handle user input from the view. When the user adds a todo item, the controller would add the item to the model. When the user edits a todo item, the controller would update the item in the model. When the user deletes a todo item, the controller would delete the item from the model.

Other design patterns

- **Model-View-Adapter**
- **Model-View-Presenter**
- **Model-View-Viewmodel**
- **Hierarchical MVC**
- **Presentation-Abstraction-Control**

> Each has it's uses, but fundamentals are very similar

L1.5: *Why the "Web" ?*

Historical background:

Telephone networks ~1890+

- Circuit switched - allow A to talk to B, through complex switching network
- Physical wires tied up for duration of call even if nothing said

Packet switched networks ~1960s

- Wires occupied only when data to be sent - more efficient use
- Data instead of Voice

ARPANet (*Advanced Rsearched Project Agency Network*) - 1969

- Node-to-node network
- Mostly university driven

Others

- **IBM SNA, Digital DECNet, Xerox Ethernet, ...**

Protocol:

- A set of rules that defines how the data packets are formed and placed on wires.
- How to format packets, place them on wire, headers/checksums.
- Each network had its own protocol

"Inter" network

- How to communicate between differnet network protocols ?
- Or replace them with a single "Inter" net protocol.

TCP: *Transmission Control Protocol* ~1983

- Connection-oriented protocol.
- Provides reliable data transfer.
- Used for applications that require guaranteed delivery, such as web browsing and email.

UDP: *User Datagram Protocol* ~1983

- Connectionless protocol.
- Provides no guarantee of data delivery.
- Used for applications that do not require guaranteed delivery, such as streaming media and gaming.

Domain Names ~1985

- Use names instead of IP addresses
- Easy to remember - .com revolution still in the future
- Hireracial structure - find names from the domain(root) node

HyperText ~1989+

- Used to format the text, how it displayed.
- Text documents to be "served"
- Formatting hints inside document to "link" to other documents - HyperText

WWW - *World Wide Web*

- It is a system of interconnected HyperText documents linked by hyperlinks and URLs.

Where are we now ?

Original web limited (Web 1.0)

- Static websites with limited user interaction.
- Served as a platform for information sharing.

Web 2.0 ~2004+

- Interactive websites with user-generated content.
- Served as a platform for collaboration and social networking.

Web 3.0

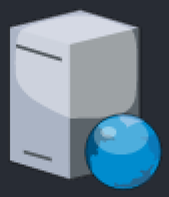
- Decentralized websites with blockchain technology.
- Served as a platform for trustless transactions and ownership.

Feature	Web 1.0	Web 2.0	Web 3.0
User interaction	Low	High	High
Content creation	Low	High	High
Data ownership	Centralized	Centralized	Decentralized
Technology	Static HTML	Dynamic HTML, JavaScript, CSS	Blockchain
Purpose	Information sharing	Collaboration, social networking	Trustless transactions, ownership

---

L1.6: *How does the Web work ?*

Web Server



- Any computer with a network connection
- Software:
-   - Listen for incomig network connections on a fixed port
-   - Respond in specific ways
-   - Opening network connections, ports etc are already known to OS
- Protocol
-   - What should client ask server
-   - How should server respond to client

HTTP

HyperText

- Regular text document
- Cotnains "code" inside that indicate special functions - how to "link" to other documents

HyperText Transfer Protocol

- Largely text based: client send requests, server responds with hypertext document
- HTTP is a stateless protocol.
- It means that each HTTP request is independent of any previous requests. The server does not keep track of any state information about the client, such as what pages the client has visited or what information the client has entered into forms.

L1.7: *Simple Web Server*

Simplest Web Server

- We will create a simple web server with following bash code:
- `simple.server.sh`

```
while true; do
            echo -e "HTTP/1.0 200 OK\n\n $(date)"
            | nc -l localhost 1800;
            done
```

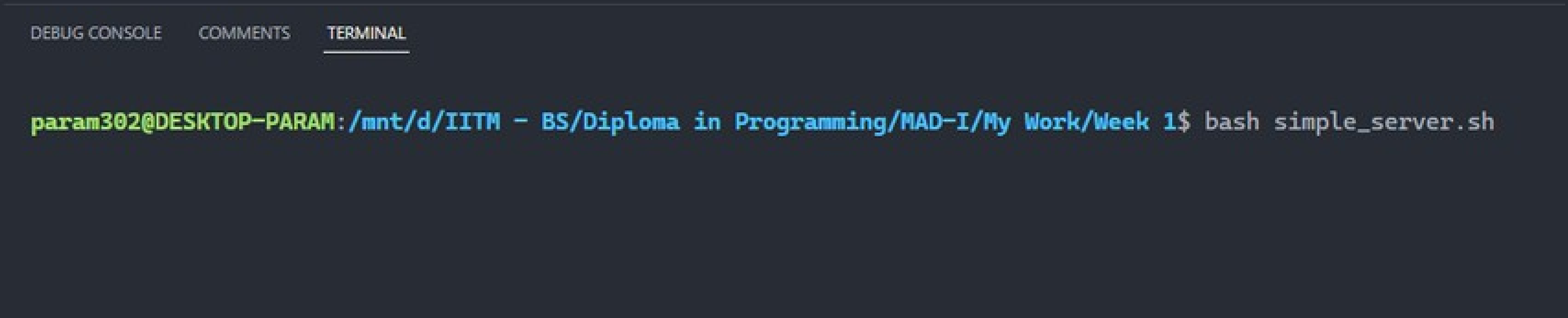
- This Bash script creates a simple HTTP server that listens on port 1800 of the localhost indefinitely. It responds with an HTTP 200 OK response containing the current date and time whenever a connection is made.

1. The script starts an infinite loop using `while true; do`.
2. Within the loop, it uses `echo` to generate an HTTP response message with the status line and current date.
3. The response message is piped (`|`) to `nc` (netcat), which listens on port 1800 and sends the response back to the client. This process continues indefinitely.

- Now, let's run this shell script on a terminal

```
$ bash
            simple_server.sh
```

terminal 1 (server)

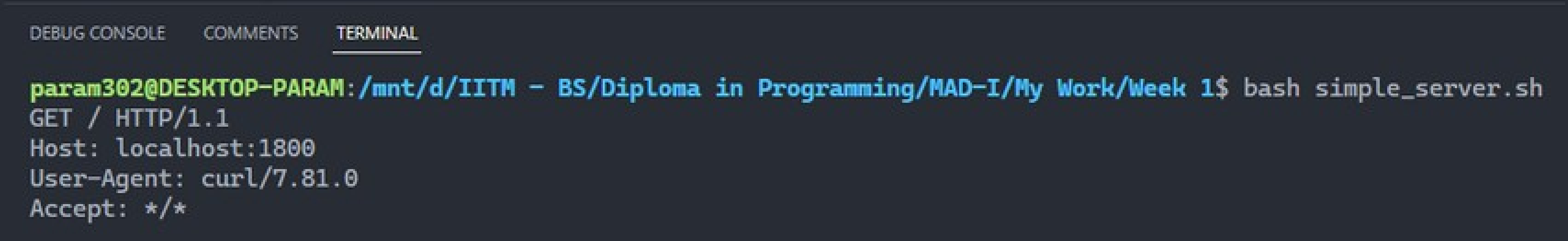


```
param302@DESKTOP-PARAM:/mnt/d/IITM - BS/Diploma in Programming/MAD-I/My Work/Week 1$ bash simple_server.sh
```

- Now, in another terminal, send an HTTP request at 1800 port (localhost) using `curl`

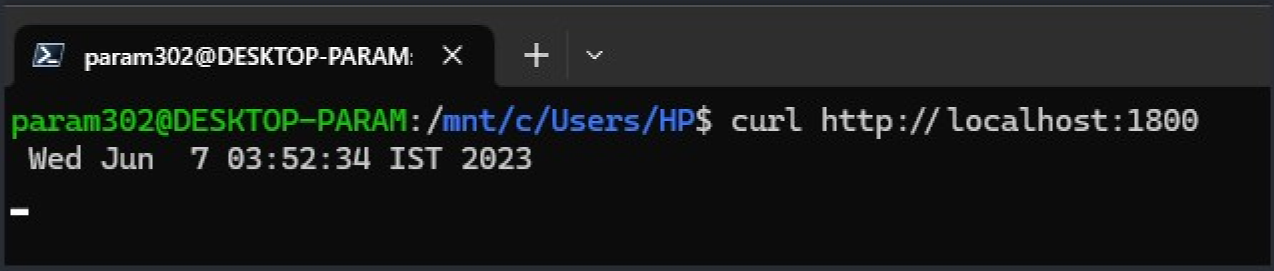
```
$ curl
          http://localhost:1800
```

terminal 1 (server)



```
param302@DESKTOP-PARAM:/mnt/d/IITM - BS/Diploma in Programming/MAD-I/My Work/Week 1$ bash simple_server.sh
GET / HTTP/1.1
Host: localhost:1800
User-Agent: curl/7.81.0
Accept: */*
```

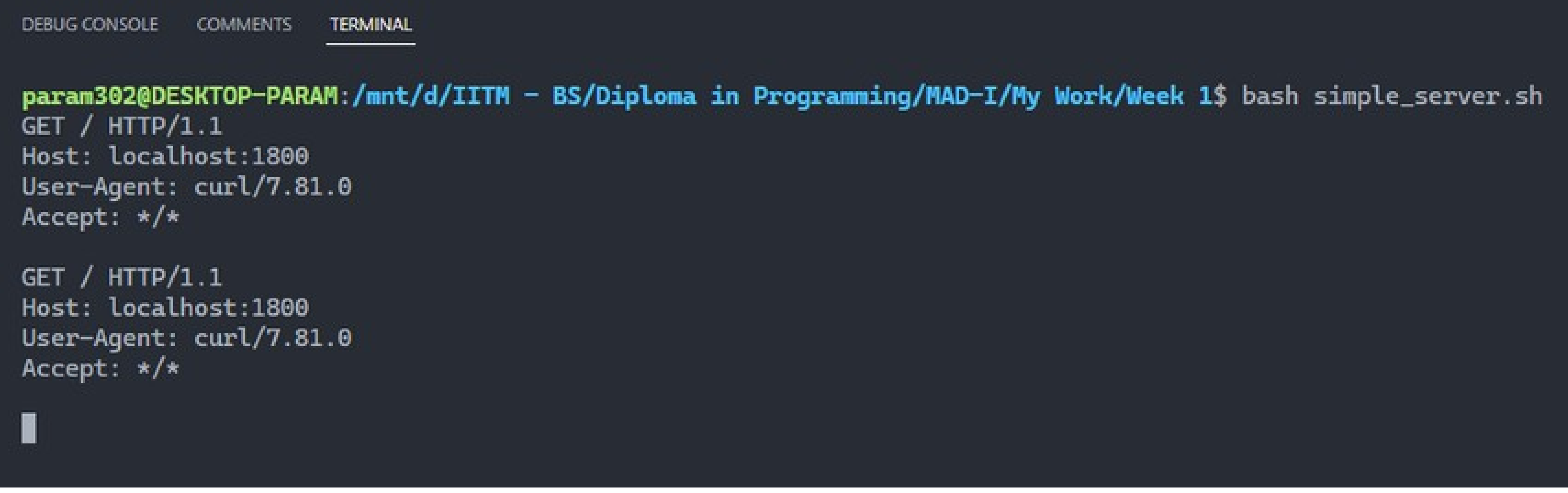
terminal 2 (request)



```
param302@DESKTOP-PARAM:/mnt/c/Users/HP$ curl http:// localhost:1800
 Wed Jun  7 03:52:34 IST 2023
```

- Whenever we try to send an HTTP request, the request will be shown in the server (terminal 1)

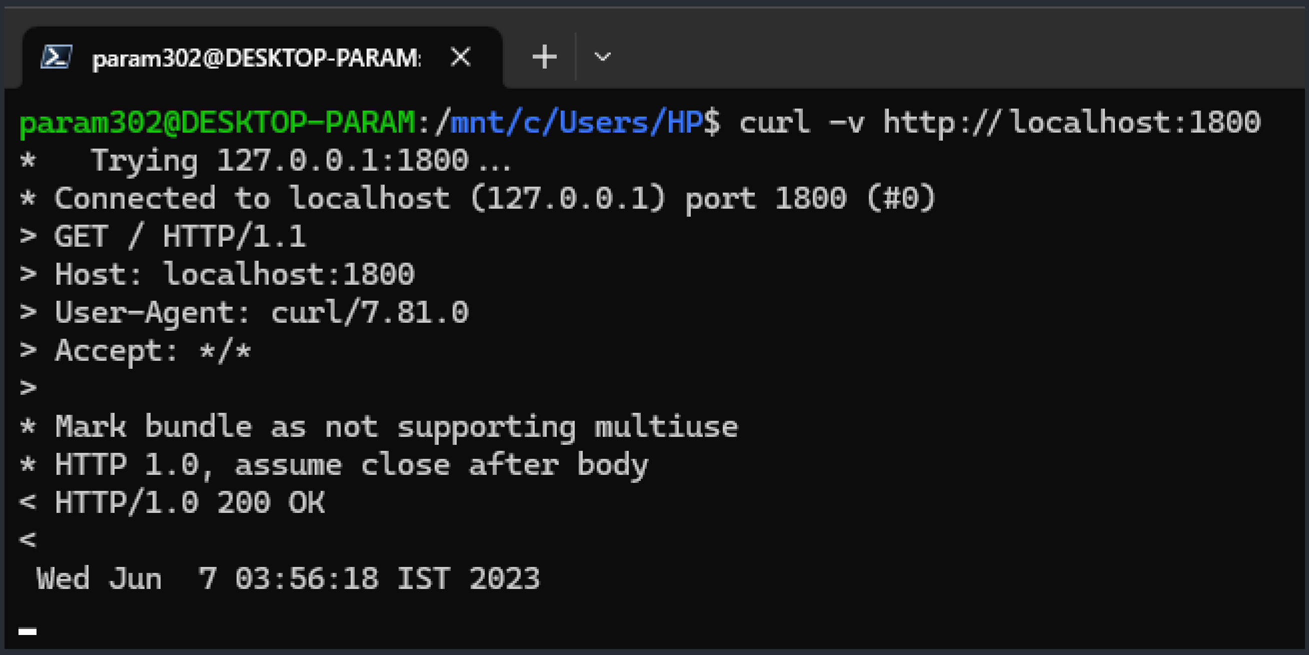
terminal 1 (server)



```
param302@DESKTOP-PARAM:/mnt/d/IITM - BS/Diploma in Programming/MAD-I/My Work/Week 1$ bash simple_server.sh
GET / HTTP/1.1
Host: localhost:1800
User-Agent: curl/7.81.0
Accept: */*

GET / HTTP/1.1
Host: localhost:1800
User-Agent: curl/7.81.0
Accept: */*
```

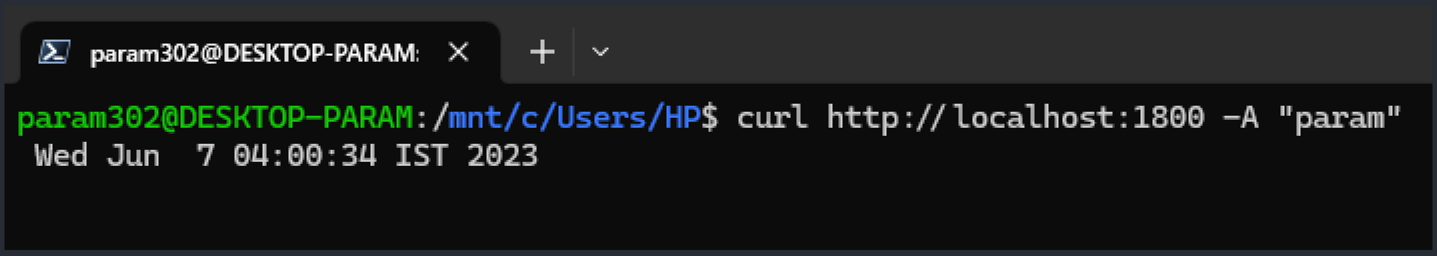
terminal 2 (another request)



```
param302@DESKTOP-PARAM:/mnt/c/Users/HP$ curl -v http://localhost:1800
*   Trying 127.0.0.1:1800...
* Connected to localhost (127.0.0.1) port 1800 (#0)
> GET / HTTP/1.1
> Host: localhost:1800
> User-Agent: curl/7.81.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
<
 Wed Jun  7 03:56:18 IST 2023
```

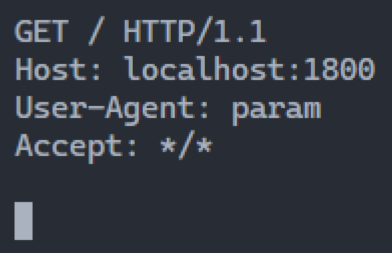
- here `-v` stands for `verbose`, i.e. more detailed info.
- We can set our custom `User-Agent` name using `-A` command while sending a request

terminal 2 (request)



```
param302@DESKTOP-PARAM:/mnt/c/Users/HP$ curl http://localhost:1800 -A "param"
 Wed Jun  7 04:00:34 IST 2023
```

terminal 1 (server)

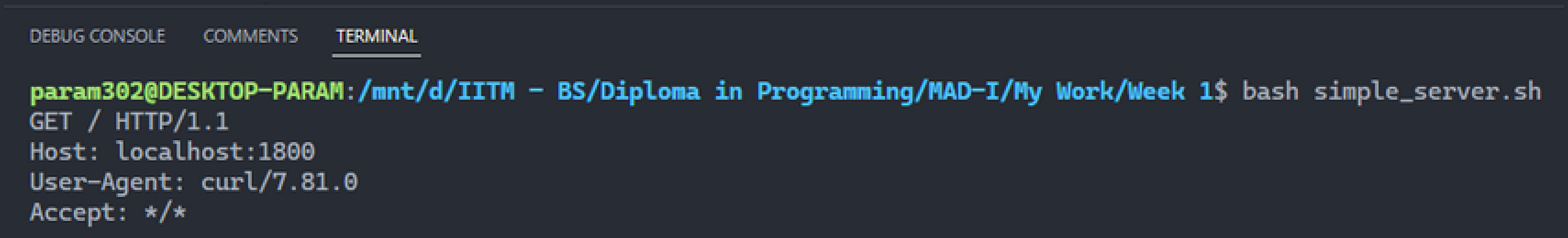


```
GET / HTTP/1.1
Host: localhost:1800
User-Agent: param
Accept: */*
```

General Web server

- Listen on a fixed port
- On incoming request, run some code and return a result
    - Standard headers to be sent as part of result
    - Output can be text or other format - MIME (Multipurpose Internet Mail Extensions)

A Typical request



```
param302@DESKTOP-PARAM:/mnt/d/IITM - BS/Diploma in Programming/MAD-I/My Work/Week 1$ bash simple_server.sh
GET / HTTP/1.1
Host: localhost:1800
User-Agent: curl/7.81.0
Accept: */*
```

- `GET` shows it's a GET request, we are requesting information from the server.
- `HOST` is the request URL, it's localhost at port 1800
- `User-Agent` tells us the agent name who is requesting it, by default it is `curl/version` we modified it above using `-A` command.
- `Accept` indicates MIME types that the client is willing to accept

Loopback Address

- A loopback address is a special IP address that is used to test the network stack of a computer.
- It is not a real address, and packets sent to a loopback address are not sent over the network.
- Instead, they are looped back to the computer that sent them.

Protocol	Loopback Address
IPv4	127.0.0.1
IPv6	::1

- Loopback addresses are useful for testing network connectivity and for debugging network problems.

  > - For example, you can use a loopback address to test whether your computer is able to resolve DNS names and to test whether your computer is able to connect to other computers on the network.

CGI: *Common Gateway Interface*

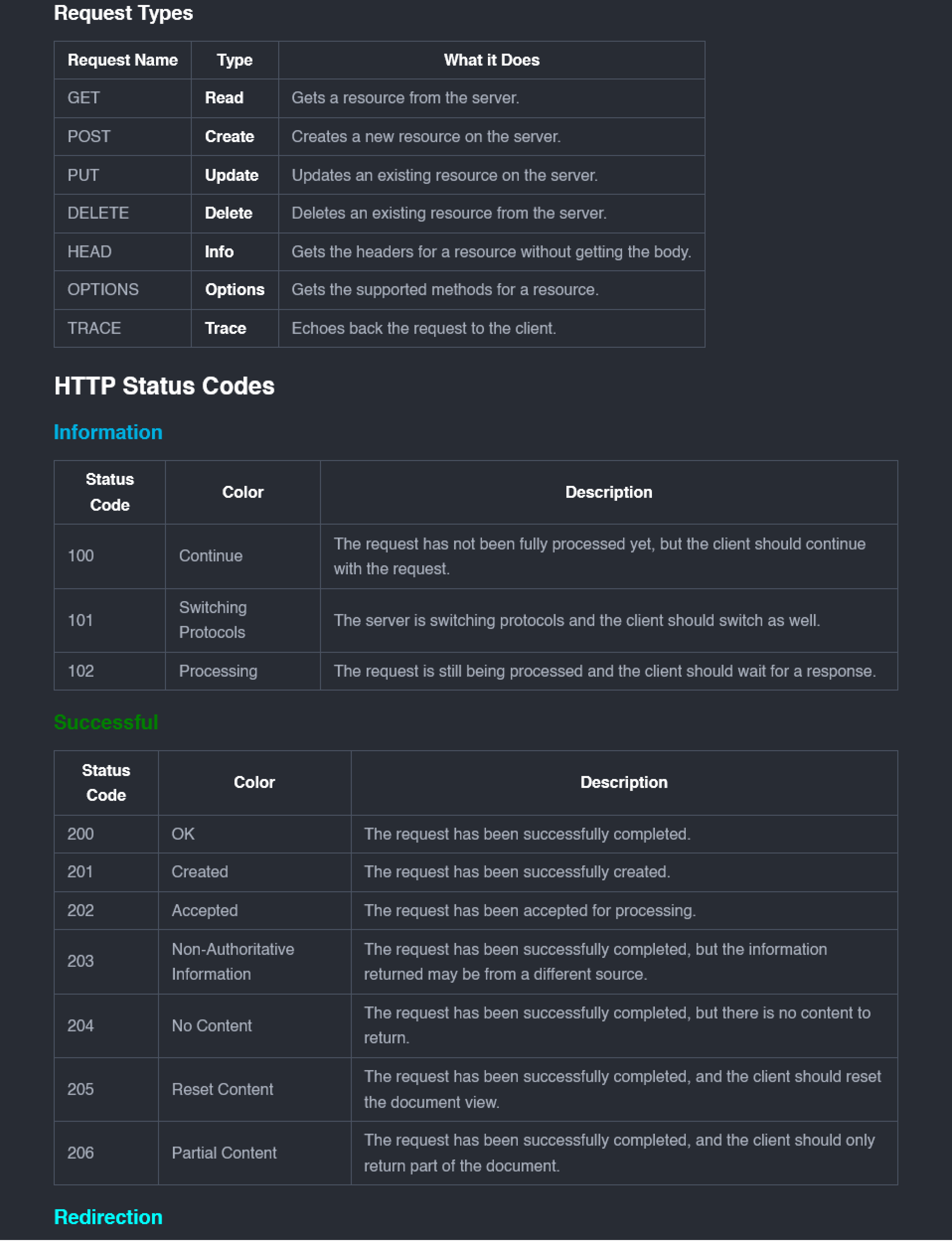
- It is a standard interface that allows web servers to execute external programs, such as scripts, to generate dynamic content.
- CGI scripts are executed on the server-side.
- CGI scripts can be used to do a variety of things, such as:
  - Generate dynamic content, such as web pages, images, or videos
  - Process user input, such as form submissions
  - Access databases
  - Communicate with other web services

L1.8: *Protocol*

- Both sides agree on how to talk.
- Server expects **requests**
  - Nature of request
  - Nature of client
- Client expects **responses**
  - Ask server for something
  - Convey what you can accept
  - Read result and process

HyperText Transfer Protocol: (HTTP)

- Primarily text based
- Requests specified as "GET", "POST", "PUT" etc...
  - Headers can be used to convey acceptable response types, languages, encoding
  - Which host to connect to
- Response headers
  - convey message type, data
  - cache information
  - status codes



Request Types

Request Name	Type	What it Does
GET	**Read**	Gets a resource from the server.
POST	**Create**	Creates a new resource on the server.
PUT	**Update**	Updates an existing resource on the server.
DELETE	**Delete**	Deletes an existing resource from the server.
HEAD	**Info**	Gets the headers for a resource without getting the body.
OPTIONS	**Options**	Gets the supported methods for a resource.
TRACE	**Trace**	Echoes back the request to the client.

HTTP Status Codes

Information

Status Code	Color	Description
100	Continue	The request has not been fully processed yet, but the client should continue with the request.
101	Switching Protocols	The server is switching protocols and the client should switch as well.
102	Processing	The request is still being processed and the client should wait for a response.

Successful

Status Code	Color	Description
200	OK	The request has been successfully completed.
201	Created	The request has been successfully created.
202	Accepted	The request has been accepted for processing.
203	Non-Authoritative Information	The request has been successfully completed, but the information returned may be from a different source.
204	No Content	The request has been successfully completed, but there is no content to return.
205	Reset Content	The request has been successfully completed, and the client should reset the document view.
206	Partial Content	The request has been successfully completed, and the client should only return part of the document.

Redirection

Status Code	Color	Description
303	See Other	The client should make a new request to the URI specified in the Location header field.
304	Not Modified	The requested resource has not been modified since the last request.
307	Temporary Redirect	The client should make a new request to the URI specified in the Location header field, but only temporarily.
308	Permanent Redirect	The client should make a new request to the URI specified in the Location header field, and this redirect should be treated as permanent.

Client Error

Status Code	Color	Description
411	Length Required	The request did not specify the length of the message-body.
412	Precondition Failed	The request failed due to a precondition.
413	Payload Too Large	The request entity is larger than the server is willing or able to process.
414	URI Too Long	The URI requested is longer than the server is willing to interpret.
415	Unsupported Media Type	The request entity has a media type that the server is not willing or able to handle.
416	Range Not Satisfiable	The requested range cannot be satisfied.
417	Expectation Failed	The server cannot meet the expectation given in the Expect request-header field.

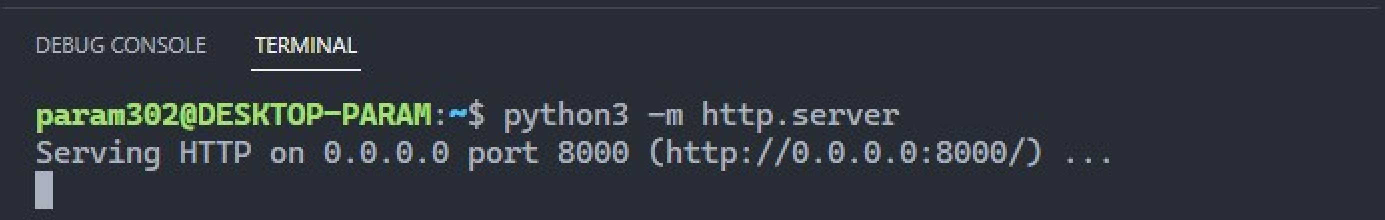
Server Error

Status Code	Color	Description
500	Internal Server Error	The server encountered an unexpected condition which prevented it from fulfilling the request.
501	Not Implemented	The server does not support the functionality required to fulfill the request.
502	Bad Gateway	The server received an invalid response from an upstream server.
503	Service Unavailable	The server is currently unavailable due to maintenance.
504	Gateway Timeout	The server did not receive a timely response from an upstream server.
505	HTTP Version Not Supported	The server does not support the HTTP version used in the request.

Another Web Server

- Let's host a web server using Python's built-in HTTP server module.

```
$ python3 -m http.server
```

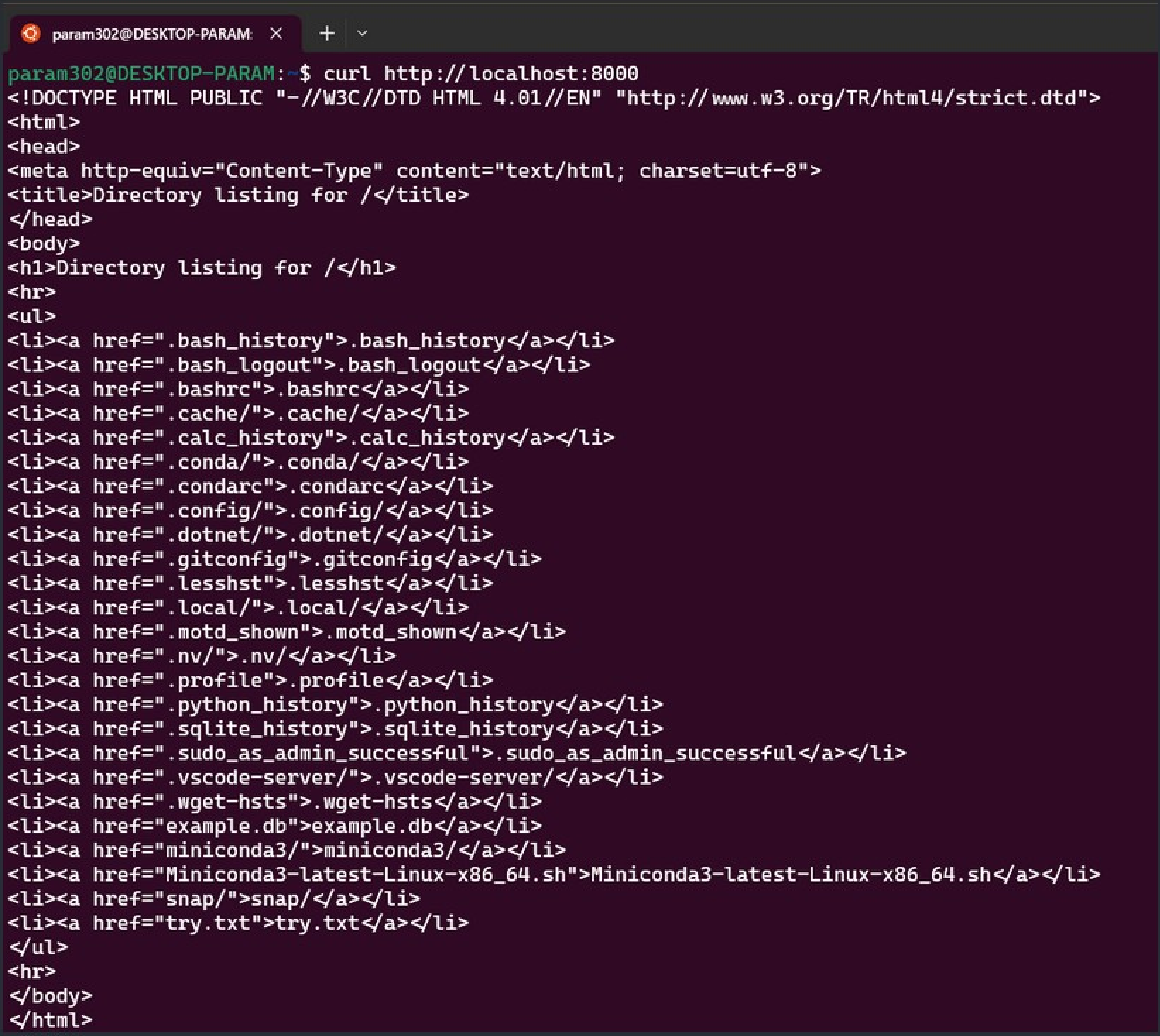


```
param302@DESKTOP-PARAM:~$ python3 -m http.server
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
```

- Now, let's make a request to the server using `curl`.

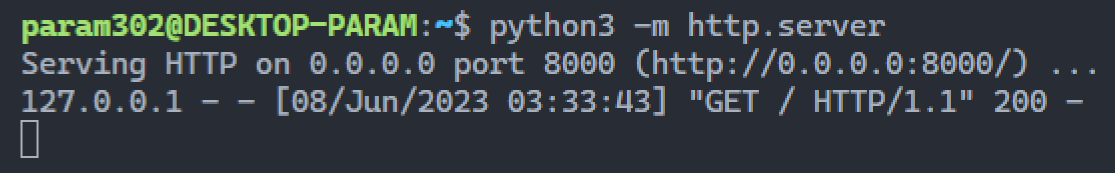
```
$ curl
          http://localhost:8000
```

- We will get our response something like this:



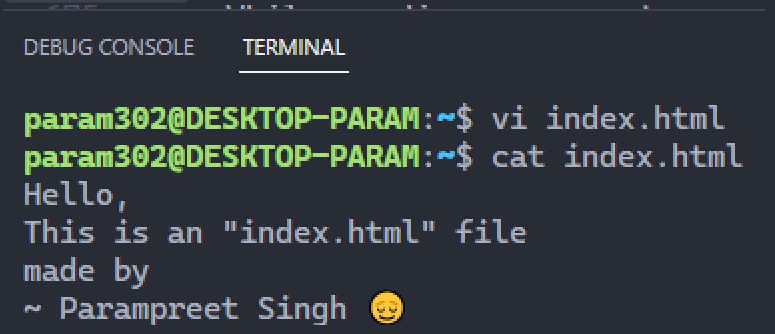
```
param302@DESKTOP-PARAM:~$ curl http://localhost:8000
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Directory listing for /</title>
</head>
<body>
<h1>Directory listing for /</h1>
<hr>
<ul>
<li><a href=".bash_history">.bash_history</a></li>
<li><a href=".bash_logout">.bash_logout</a></li>
<li><a href=".bashrc">.bashrc</a></li>
<li><a href=".cache/">.cache/</a></li>
<li><a href=".calc_history">.calc_history</a></li>
<li><a href=".conda/">.conda/</a></li>
<li><a href=".condarc">.condarc</a></li>
<li><a href=".config/">.config/</a></li>
<li><a href=".dotnet/">.dotnet/</a></li>
<li><a href=".gitconfig">.gitconfig</a></li>
<li><a href=".lesshst">.lesshst</a></li>
<li><a href=".local/">.local/</a></li>
<li><a href=".motd_shown">.motd_shown</a></li>
<li><a href=".nv/">.nv/</a></li>
<li><a href=".profile">.profile</a></li>
<li><a href=".python_history">.python_history</a></li>
<li><a href=".sqlite_history">.sqlite_history</a></li>
<li><a href=".sudo_as_admin_successful">.sudo_as_admin_successful</a></li>
<li><a href=".vscode-server/">.vscode-server/</a></li>
<li><a href=".wget-hsts">.wget-hsts</a></li>
<li><a href="example.db">example.db</a></li>
<li><a href="miniconda3/">miniconda3/</a></li>
<li><a href="Miniconda3-latest-Linux-x86_64.sh">Miniconda3-latest-Linux-x86_64.sh</a></li>
<li><a href="snap/">snap/</a></li>
<li><a href="try.txt">try.txt</a></li>
</ul>
<hr>
</body>
</html>
```

- While sending a request, server is also showing what request is being made.



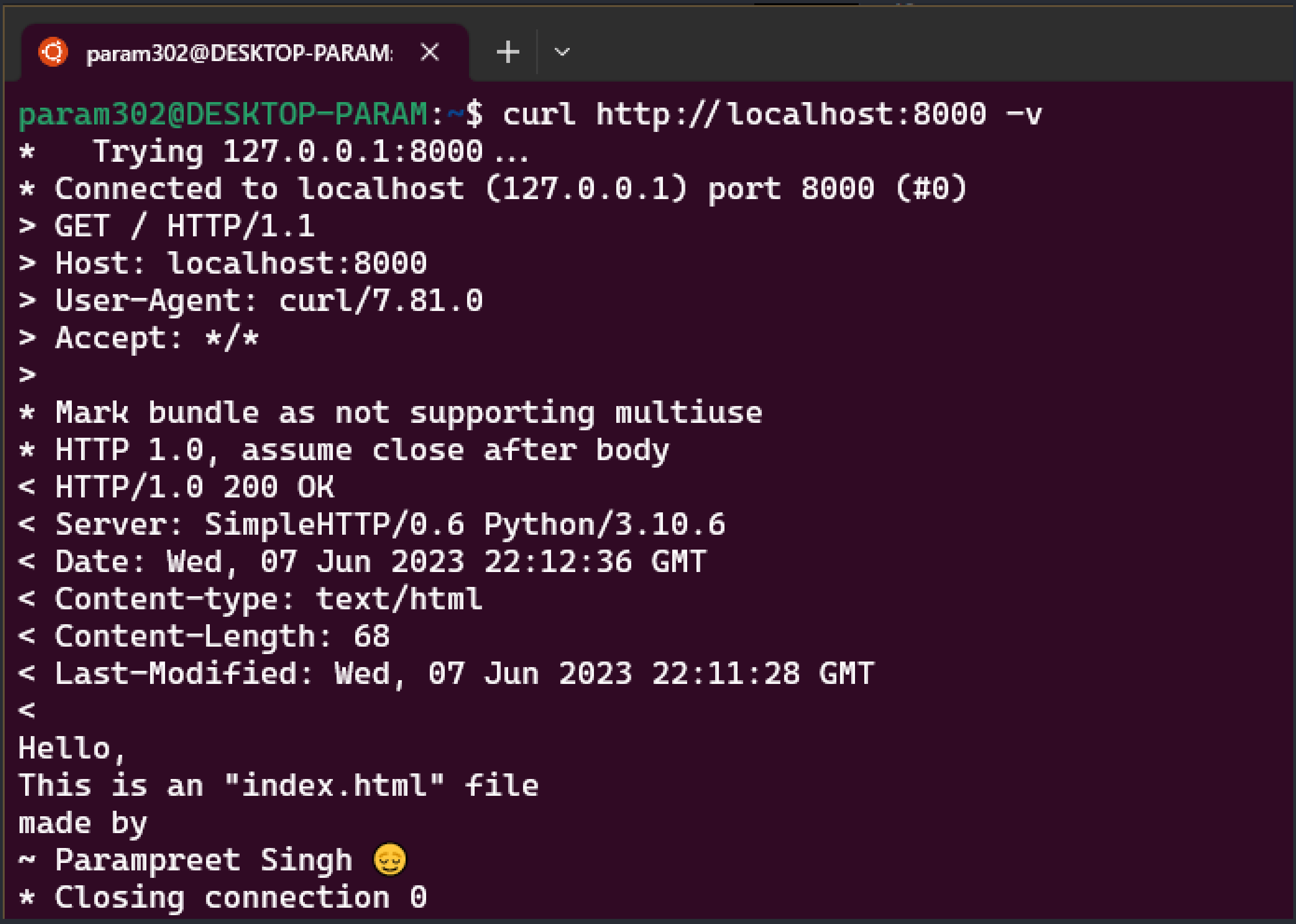
```
param302@DESKTOP-PARAM:~$ python3 -m http.server
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
127.0.0.1 - - [08/Jun/2023 03:33:43] "GET / HTTP/1.1" 200 -
```

- Now, let's create an `index.html` file in the server and make request again.
- `index.html` file content:



```
param302@DESKTOP-PARAM:~$ vi index.html
param302@DESKTOP-PARAM:~$ cat index.html
Hello,
This is an "index.html" file
made by
~ Parampreet Singh 😔
```

- Now, let's make a request to the server with verbose flag.



```
param302@DESKTOP-PARAM:~$ curl http://localhost:8000 -v
*   Trying 127.0.0.1:8000...
* Connected to localhost (127.0.0.1) port 8000 (#0)
> GET / HTTP/1.1
> Host: localhost:8000
> User-Agent: curl/7.81.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
< Server: SimpleHTTP/0.6 Python/3.10.6
< Date: Wed, 07 Jun 2023 22:12:36 GMT
< Content-type: text/html
< Content-Length: 68
< Last-Modified: Wed, 07 Jun 2023 22:11:28 GMT
<
Hello,
This is an "index.html" file
made by
~ Parampreet Singh 😔
* Closing connection 0
```

- As we can see, now only the content of `index.html` file we get as response.
- Because, `index.html` file is typically set as the default or entry point file for web servers to serve when a user requests a specific directory or domain.

L1.9: *Performance*

Latency

- Latency is the time it takes for a request to travel from the client to the server and back.
- Speed of light is 3×10^8 m/s. in vacuum, and 2×10^8 m/s in copper cable.
-
    - Approximates to $5ms$ per 1000 km.

Example:

- Data center is 200km away
-
    - One way request takes $10ms$
-
    - Round trip will take $20ms$
- which is max 50 requests per second, not good enough.

Response size

- Response size is the amount of data that is sent from the server to the client.

Example:

- Response = 1KB of text (HTML, CSS, JS)
- Network connection = 100 Mbps $=$ 10 MByte/s
- which approximates to $10,000$ requests per second $(100 Mbps * 1)$.
- Server will crash if more than $10,000$ requests per second are made.

Screencast 1.1: *How to serve HTML files on LAN?*

- Let's create a HTML file named `index.html` with some sample content.

```
<!DOCTYPE html>
            <html>
            <head>
            <meta charset="UTF-8"
                />
            <title>A Sample Webpage</title>
            <meta name="description" content="This is an example html page" />
            </head>

            <body>
            <header>
            <h1>Introduction</h1>
            <p>This is an introduction part of this document.</p>
            </header>
            <main>
            <p>This is main section of this document.</p>
            </main>
            <footer>
            <p>This is the footer.</p>
            </footer>
            </body>
            </html>
```

- Now we can either open it in any browser directly by clicking on the file in system's file explorer.
- Or, We can host a Local server - LAN server to serve this file.
- To host a LAN server, we can use Python's built-in HTTP server module.
- In terminal, we will write the below command to host a server.

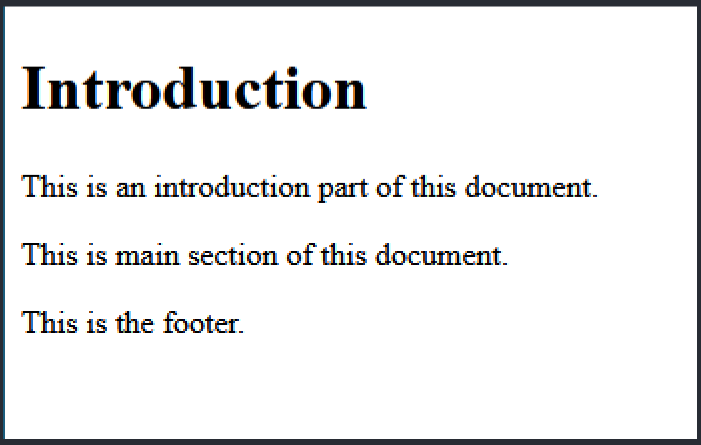
```
python -m http.server
```

- This will (by default) host a server on `8000` port.
- We can access it using any IP address followed by `8000` port or just with localhost.

```
http://localhost:8000
```

```
127.0.0.1:8000
```

- For the above sample, the HTML will be rendered as:



Introduction

This is an introduction part of this document.

This is main section of this document.

This is the footer.

Extra

Total Ports in TCP are: $65,535$ $(0xffff)$

1. There are 65,535 port numbers available in total for communication between devices in TCP.
2. The port numbers are divided into 3 categories:

- **Well-known ports (0–1023)**: These ports are reserved for well-known services, such as HTTP (port 80) and SMTP (port 25).
- **Registered ports (1024–49151)**: These ports are available for registration by organizations that want to use them for their own services.
- **Dynamic/private ports (49152–65535)**: These ports are available for anyone to use.

IPv4 vs IPv6

Feature	IPv4	IPv6
Address length	32 bits	128 bits
Address space	4.29×10^9 $(4.3$ billion addresses)	3.4×10^{38} $(340$ undecillion addresses)
Header size	20–60 bytes	40 bytes
Header options	Yes	No
Fragmentation	Requires intermediate routers	End-to-end
Security	Less secure	More secure
Format	Decimal form $(0 - 255)$	Hexadecimal form $(0 - 65,535)$
Example	`192.168.1.1` divided into 4 octets: $192, 168, 1$ and 1	`2001:0db8:85a3:0000:0000:8a2e:0370:7334` divided into 8 groups of 4 hexadecimal digits: $2001, 0db8, 85a3, 0000, 0000, 8a2e, 0370$ and 7334