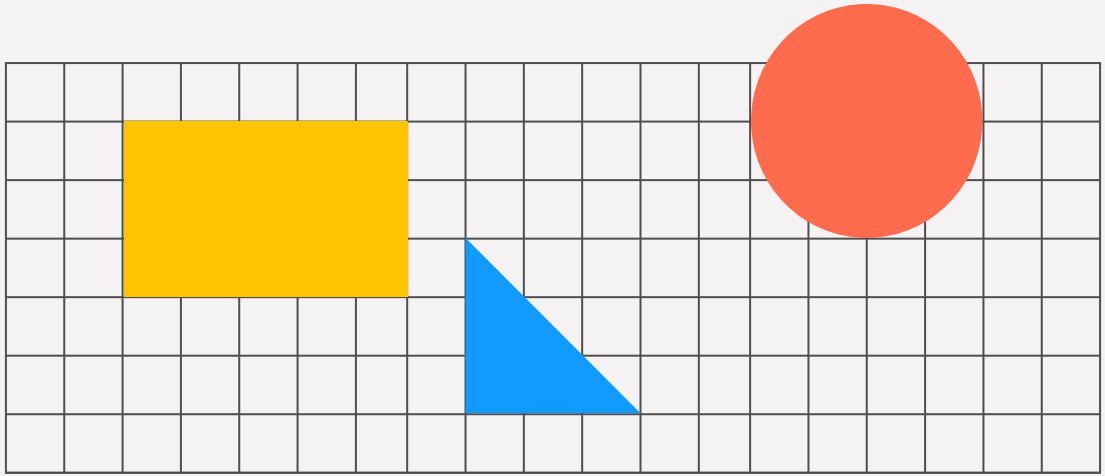
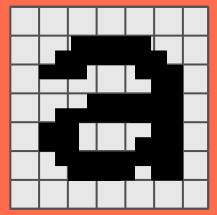
Networks, SWE, and Beyond!

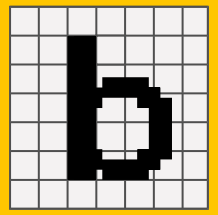
Ali Abrishami



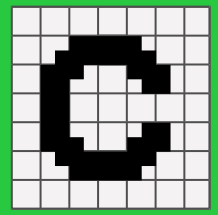
In This Lecture You Will...



Learn the basics of Computer Networks



Learn some SWE Concepts

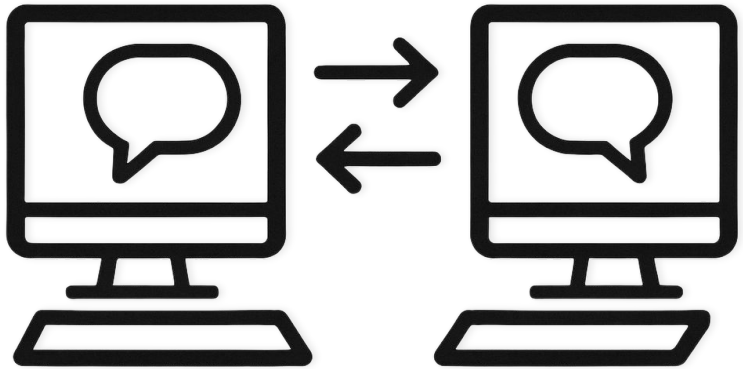


Get to know the concept of frameworks

Computer Networks

The Basics

- Computers connect to each other using networks.
- Think of them as people talking to each other.
- Talking needs language, formalities, and conventions.
- What conventions do computers follow?



Internet Protocols

The conventions of communication

- There are some conventions on how two computers should communicate over the internet.
- There are some operations that need to happen before, after, or as we talk (like shaking hands when we meet).
- These operations are called **protocols**.
- Two important protocols used for sending data are **UDP** (User Datagram Protocol) and **TCP** (Transmission Control Protocol).

The conventions we just talked about are not rules, but rather suggestions (Why?) made by a publication named RFC (formerly known as IETF)

Fun Fact!

Protocol x Protocol

TCP

- Connection-based (needs a handshake)
- Reliable, ensures data is delivered correctly
- Slower due to error checking and confirmation

UDP

- Connectionless (no handshake needed)
- Unreliable, no guarantee of delivery
- Faster, no waiting for acknowledgments

Class Activity Time!

Which one uses TCP? Which one uses UDP?

Online Video Games	File Downloading	Video Streaming
Video Calls and Meetings	Static Websites	Sending Messages

Now That Computers Can Talk, What Do They Say?

- With TCP and UDP, computers can send and receive data reliably or quickly.
- But they still need to agree on what kind of data they’re sending.
- Are they sending a file? A video? A web page?
- Enter application layer protocols, like HTTP, which powers the web.
- And with HTTP comes a powerful idea...

When two computers connect, they often "shake hands" (literally called a handshake) to agree on how they’ll talk before they send any real data!

Fun Fact!

Here Comes HyperText!

- A digital text format that allows users to navigate from one piece of information to another via hyperlinks.
- Can lead to other documents, sections within the same document, or external websites.
- Users can make choices about what to explore next, which fosters a dynamic and interactive experience.
- Unlike traditional text, hypertext offers a non-linear way of accessing information, facilitating easier exploration.
- Can incorporate various types of media, such as images, videos, and audio.

Tim Berners-Lee, the guy who invented hypertext. The first website ever is still up at info.cern.ch

Fun Fact!

Transferring HyperText

- We know what hypertext is and what it does now, but how do we move it across devices?
- Since data integrity is important, **TCP** seems like the way to go.
- But isn't it better to have a nice structure to keep our hypertext in place?
- Let's make a new protocol... over protocol! We call it HyperText Transfer Protocol (literally what it does), or HTTP for short!
- It's a protocol that relies on TCP/IP and has a well-formed structure for requests and responses.

HTTP has different versions! From HTTP/1.1, which laid the groundwork for the modern web, to the faster and more efficient HTTP/2 and the even snappier HTTP/3, each version has introduced cool features to enhance your browsing experience!

Fun Fact!

Request & Response

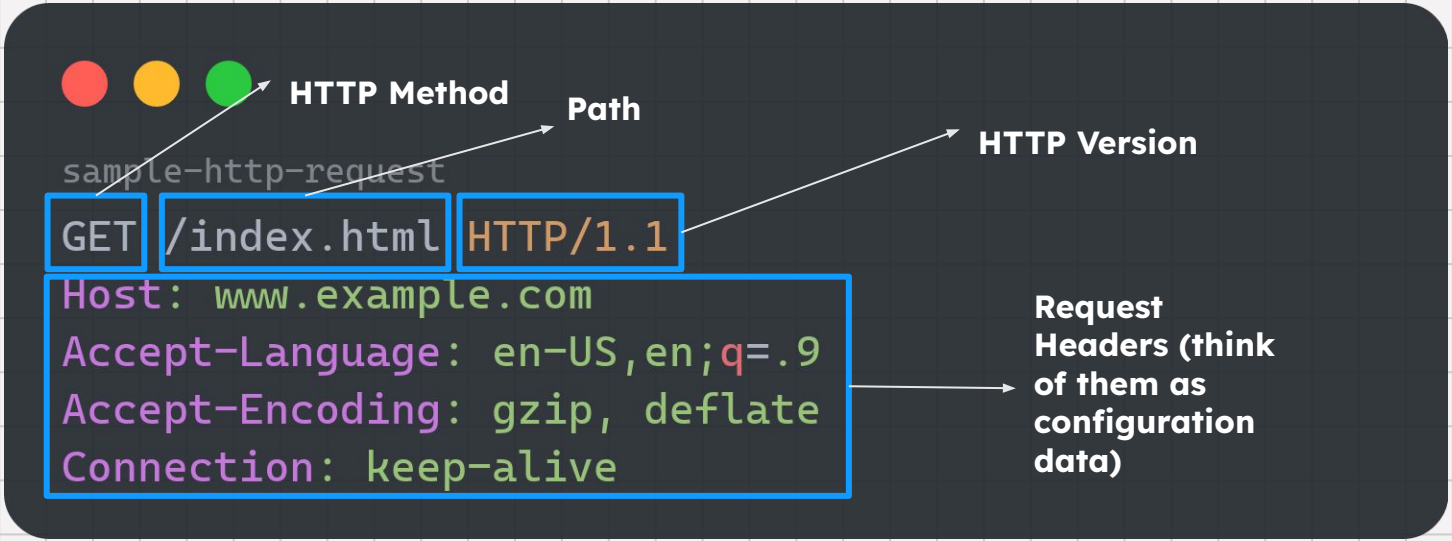
Sample HTTP Request

```
sample-http-request
GET /index.html HTTP/1.1
Host: www.example.com
Accept-Language: en-US,en;q=.9
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

Sample HTTP Response

```
sample-http-response
HTTP/1.1 200 OK
Date: Mon, 27 Sep 2021 12:00:00 GMT
Server: Apache/2.4.41 (Ubuntu)
Last-Modified: Wed, 22 Sep 2021 10:00:00 GMT
Content-Type: text/html; charset=UTF-8
Content-Length: 305

<!doctype html>
<html>
<head>
    <title>Welcome to Example</title>
</head>
<body>
    <h1>Hello, world!</h1>
    <p>Welcome to the example website.</p>
</body>
</html>
```

More on HTTP Requests



HTTP Methods

- **GET**: Retrieve data from a specified resource without modifying it.
- **POST**: Submit data to a server for processing, which can create or update a resource.
- **PUT**: Replace all current representations of a resource with the uploaded content.
- **PATCH**: Apply partial modifications to a resource, updating only specified fields.
- **DELETE**: Remove a specified resource from the server.
- **HEAD**: Similar to GET, but retrieves only the headers, not the body, of the response.
- **OPTIONS**: Describe the communication options for the target resource (more on this later).

Class Activity Time!

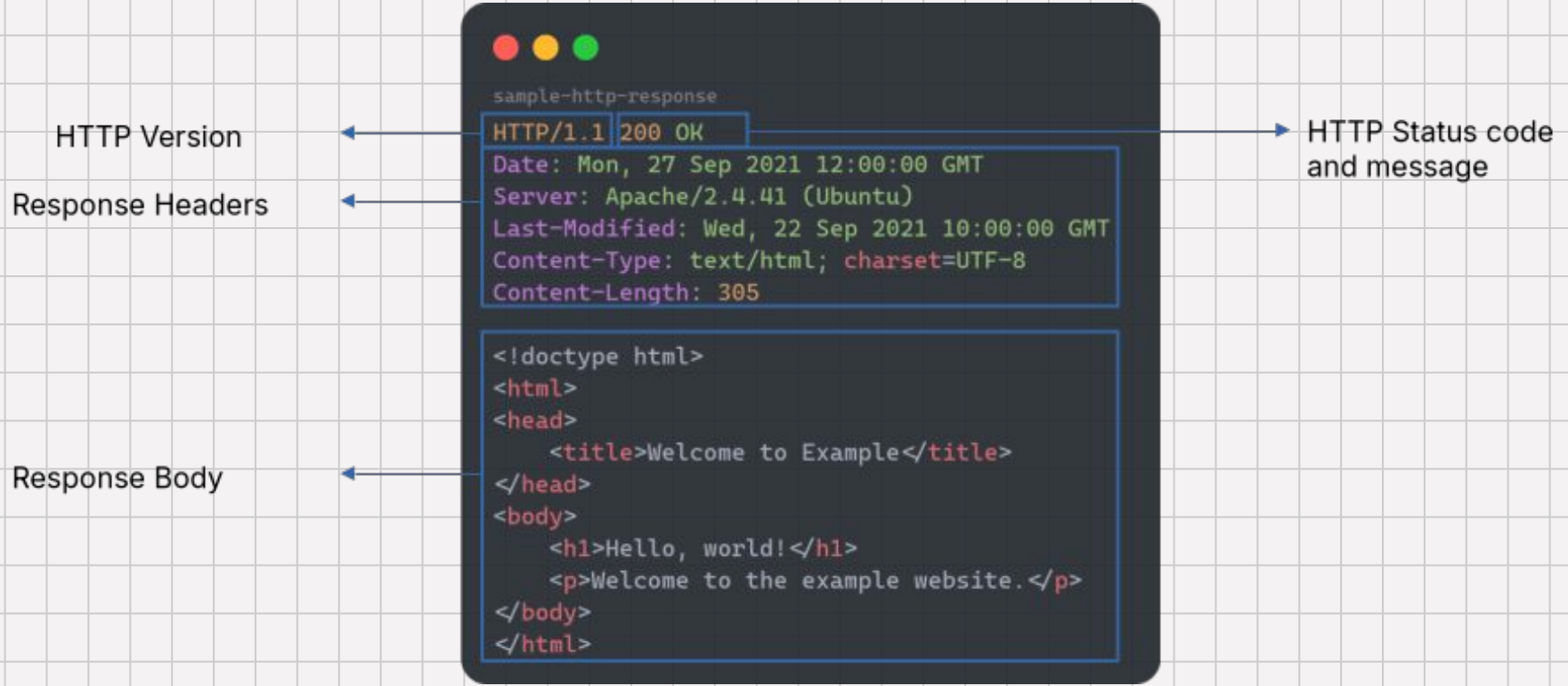
What you're seeing is a sample HTTP POST request.
You may notice that there's an empty line after headers and then comes some data.
That data is called request body.

1. Why is it uncommon for HTTP GET requests to have a body?
2. If it's uncommon for HTTP GET requests to have a body, then how do we handle things like filters in our web applications?

sample-http-request

```
POST /api/v1/resource HTTP/1.1
Host: example.com
Content-Type: application/json
Authorization: Bearer your_access_token
Content-Length: length

{
    "name": "John Doe",
    "email": "john.doe@example.com",
    "password": "securePassword123"
}
```

More on HTTP Responses



Important status codes!

Status Code	Status Message	Meaning
200	OK	The request has succeeded.
201	Created	A new resource has been created.
202	Accepted	Request has been accepted for processing.
204	No Content	There is no content to be shown.

Important status codes! (cont.)

Status Code	Status Message	Meaning
400	Bad Request	The request in malformed, invalid, etc.
401	Unauthorized	Authentication is required, but there is none.
403	Forbidden	Client cannot access the resource.
404	Not Found	The requested resource does not exist.

Important status codes! (cont.)

Status Code	Status Message	Meaning
500	Internal Server Error	The server encountered an unexpected condition that prevented it from fulfilling the request
502	Bad Gateway	The server, while acting as a gateway or proxy, received an invalid response from the upstream server.

Class Activity Time!

What's the difference between these two status codes?

401

403

Hint: Think of your home's front door and a cookie jar inside of your house when you were 5!

Software Architecture

“The rules of software architecture are independent of every other variable” – Robert. C. Martin -

Software Architecture is about:

Structure, not tools

Long-term maintenance

Decoupling and clarity*

Guiding change, not chasing trends

Separation of concerns*

Separation of concerns

- It’s important to divide our system into parts, each with a clear and distinct responsibility.
- Each part does one thing well.
- This reduces overlap and entanglement.
- It also makes testing, debugging, and scaling easier.
- Remember: “Confusion comes from mixing responsibilities.”

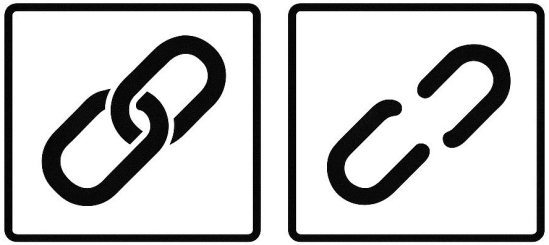
Class Activity Time!

Why does separation of concerns matter a lot?

Can you separate things... too much?

Decoupling

- **Decoupling** means designing parts of a system to work independently.
- Loose connections = Fewer side effects
- **Easier** to change, test and scale.
- Encourages **reusable** components.
- You can change one piece without breaking the rest.

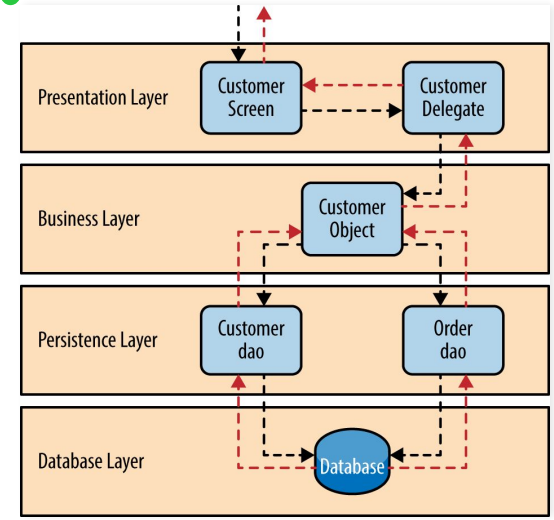


Architectural Patterns

- It costs us a lot of time and money to figure out a project structure for the software we are going to develop.
- When you think about it, the project structure is the architecture of our software.
- It needs to be clean, maintainable, stable, and scalable.
- Luckily, more experienced researchers and software engineers have already put some patterns out there for us to use.
- It’s important for us to adopt the already existing and working patterns, rather than creating our own ones. This reduces development costs significantly.

The Fundamentals of Architectural Patterns

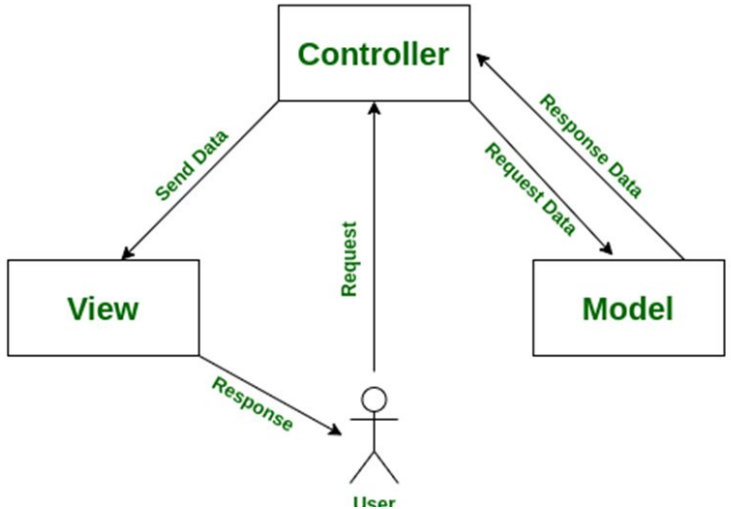
- **Separation of Concerns**: Organizes functionality for better maintainability.
- **Decoupling Components**: Allows for easier updates with loose coupling.
- **Single Responsibility Principle**: Each module has one clear purpose, simplifying design.
- **Continuous Refactoring**: Promotes iterative improvements to adapt to requirements.



Source: O'Reilly's Software Architecture Patterns (Book)

Model-View-Controller

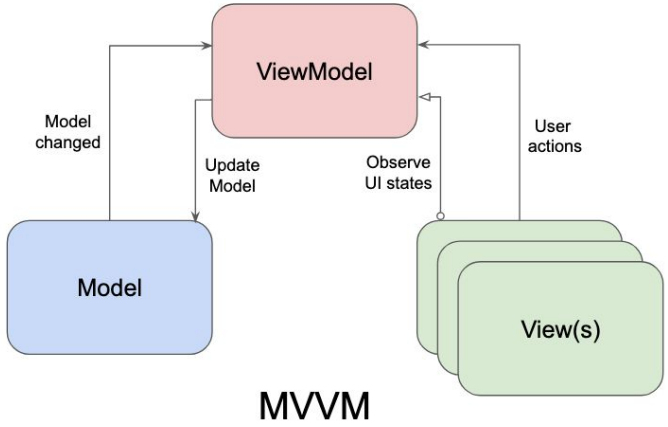
- Divides application into **Model** (data), **View** (UI), and **Controller** (logic).
- User interactions trigger Controller actions that update the Model and refresh the View.
- Enhances **modularity**, making the application easier to maintain and test.
- Commonly used in web applications and desktop software for structured user interfaces.



Source: Stackoverflow

Model-View-ViewModel

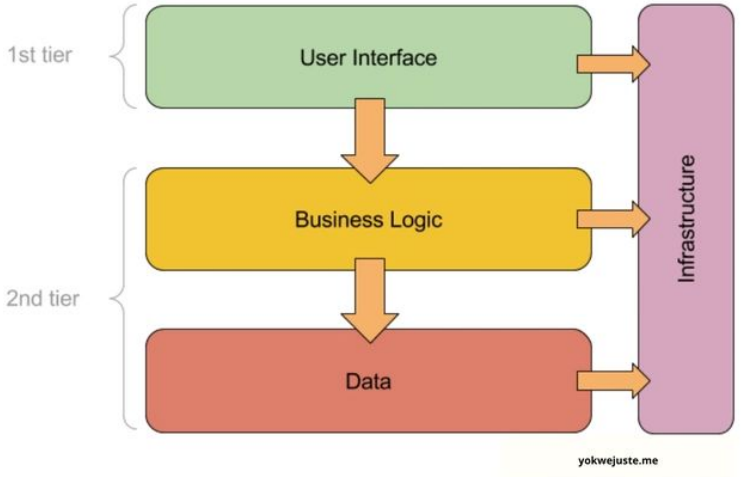
- Divides an application into **Model** (data), **View** (UI), and **ViewModel** (binds data and presentation logic).
- The View binds to the ViewModel, which communicates with the Model to retrieve and manipulate data.
- Two-way data binding allows automatic synchronization between the V & VM.
- Promotes separation of concerns, easier unit testing, and better maintainability, especially in rich client applications.



Source: Dashlane

Layered Architecture

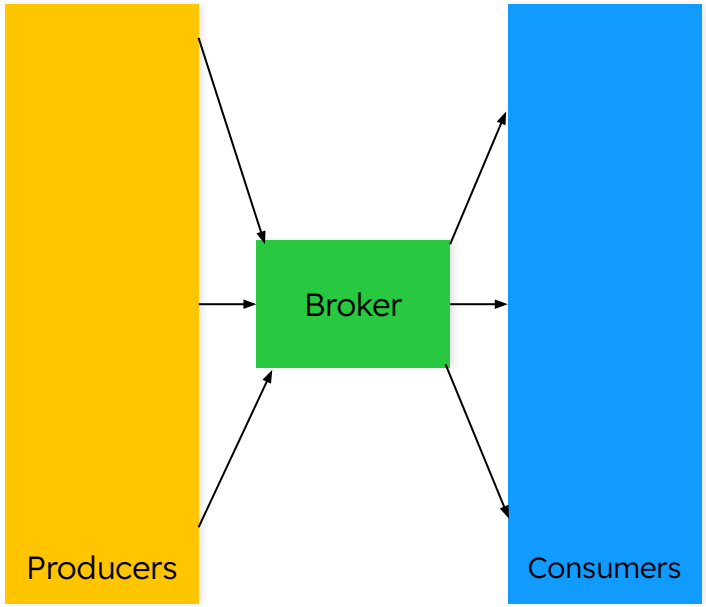
- Separates a system into layers to keep concerns isolated and maintainable.
- Ensures higher layers depend on lower layers' abstractions, not their details.
- Improves testability by isolating logic from infrastructure and UI.
- Reduces complexity and makes large systems easier to evolve safely.



Source: Medium

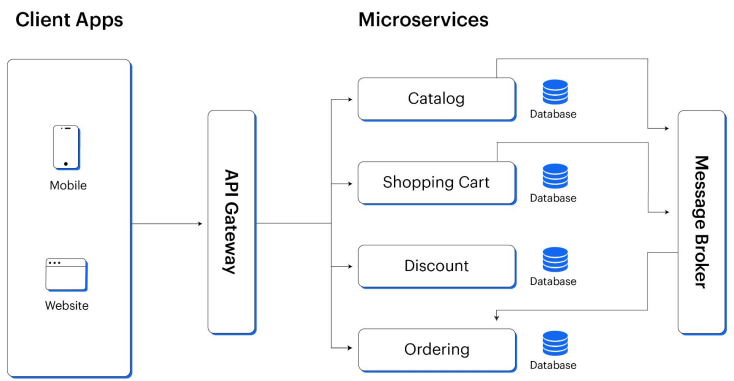
Event-driven Architecture

- Systems react to events instead of executing rigid, linear flows.
- Components stay loosely coupled by communicating through event messages.
- Scaling becomes easier because events can be processed asynchronously.
- System behavior becomes more resilient and extensible as new consumers can listen without changing existing code.



Microservices Architecture

- Breaks a system into small, independently deployable services around clear boundaries.
- Each service owns its own data and logic, reducing shared-state complexity.
- Enables teams to develop, scale, and release services without coordination bottlenecks.
- Improves fault isolation since failures are contained within individual services.

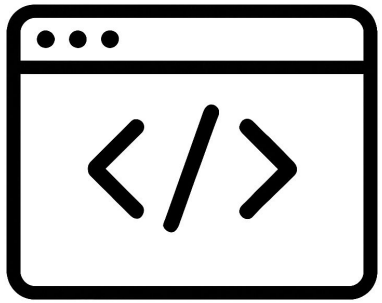


Source: Webandcrafts

Web Frameworks And Their Uses

This is where the fun begins.

- Your website needs somewhere to store all the content and data.
- You can’t do this in a client’s browser.
- You have to store your data and your content on a server.
- But then, you have to program that server to do what you want.

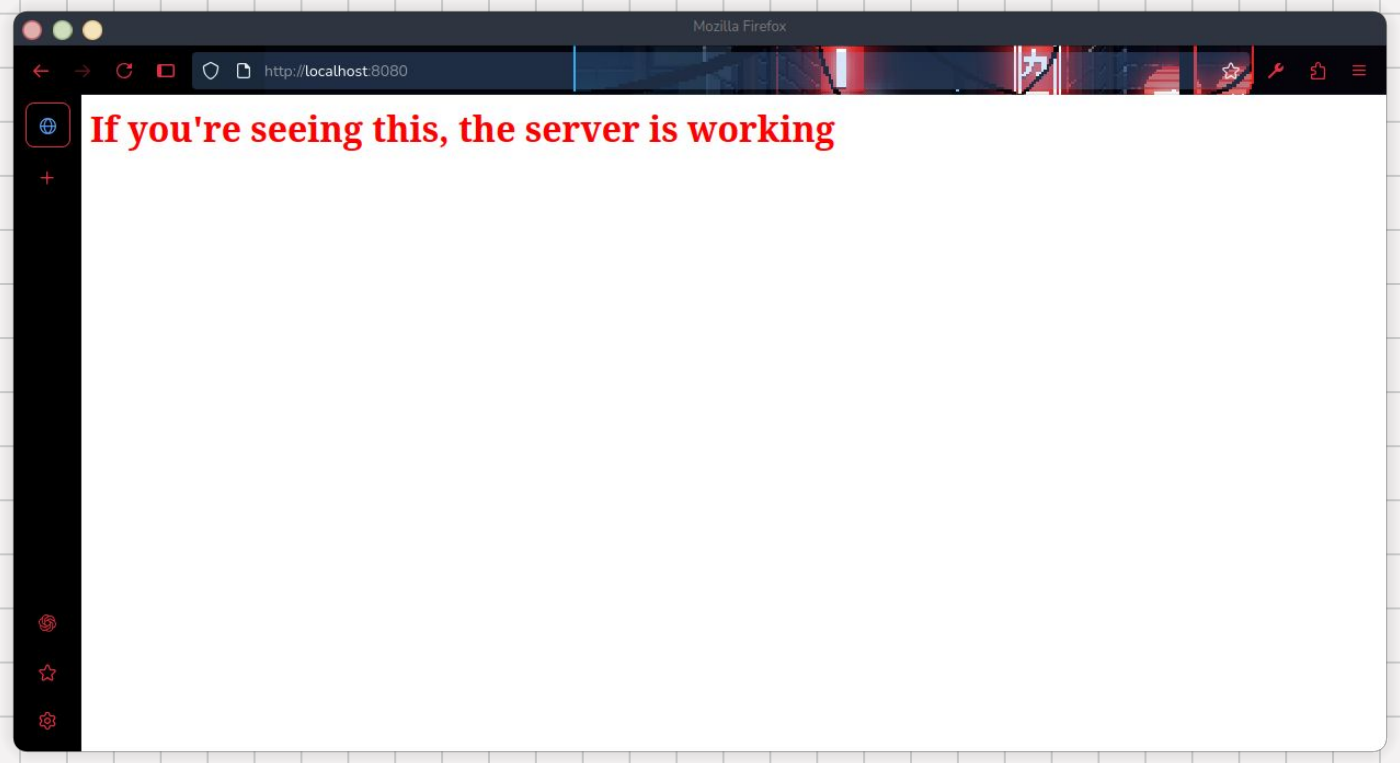


You COULD Use Socket Programming

serversocket.py

```
import socket

listener = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
listener.bind(('0.0.0.0', 8080))

HTTP_RESPONSE_TEMPLATE = """
HTTP/1.1 200 OK
Host: http://0.0.0.0:8080

<h1 style="color: red">{content}</h1>
"""

listener.listen(5)

while True:
    client_socket, client_address = listener.accept()
    data = HTTP_RESPONSE_TEMPLATE.strip().format(
            content="If you're seeing this, the server is working")
    client_socket.send(data.encode('utf-8'))
    client_socket.close()
```

And It Does Work...

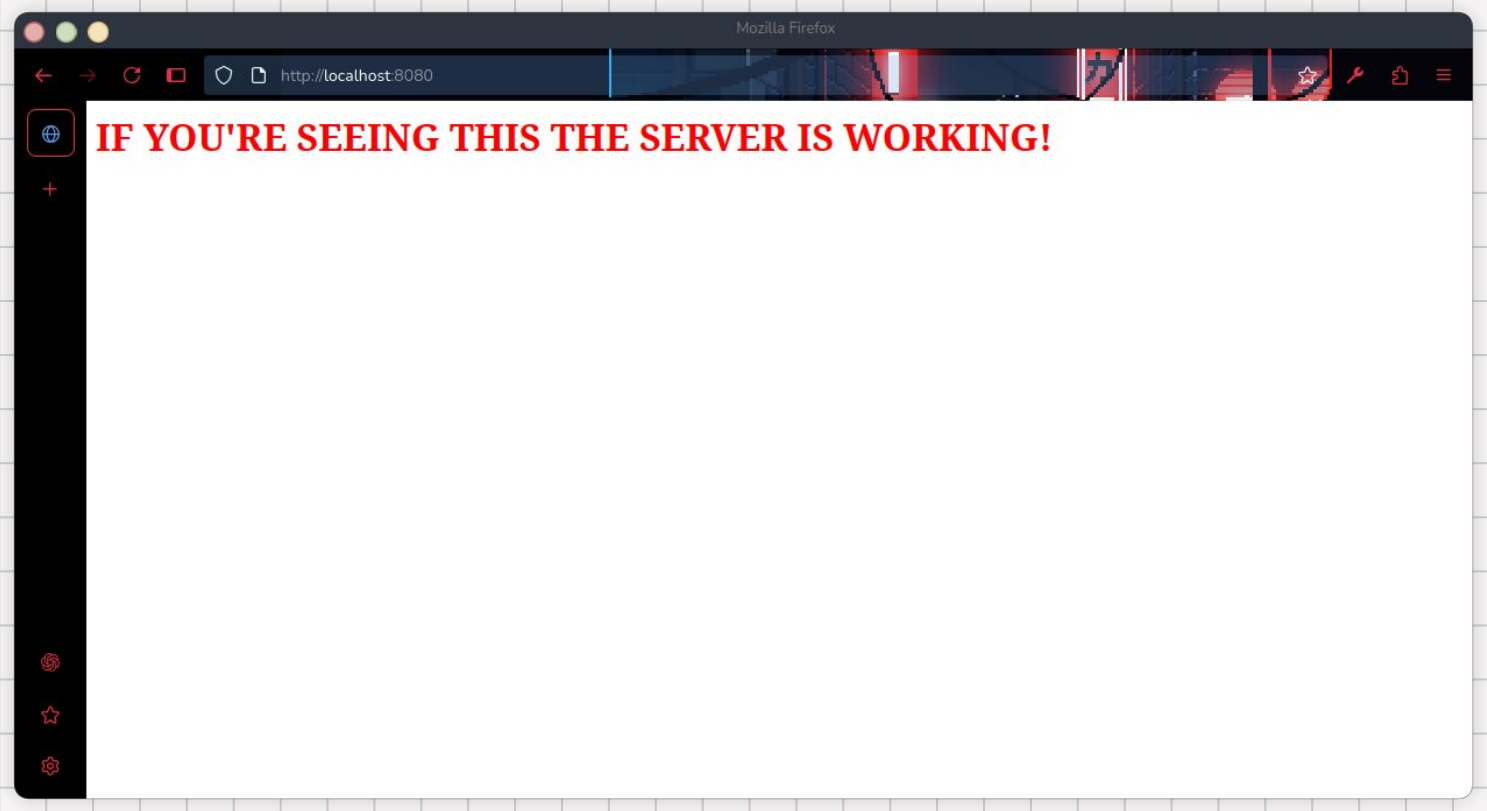


And it COULD get a bit easier...

httpserver.py

```python
from http.server import HTTPServer, BaseHTTPRequestHandler

content = b'<h1 style="color: red">IF YOU\'RE SEEING THIS THE SERVER IS WORKING!</h1>'

class SimpleHandler(BaseHTTPRequestHandler):
    def do_GET(self):
        self.send_response(200)
        self.end_headers()
        self.wfile.write(content)

HTTPServer(('', 8080), SimpleHandler).serve_forever()
```

And It Does Work...

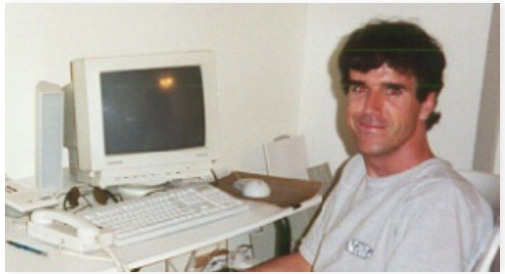


It even comes with it’s own logger!

```
floaterm(1/1)
mani  python ./httpserver.py                                    django-sut
127.0.0.1 - - [21/Jun/2025 16:15:00] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [21/Jun/2025 16:15:01] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [21/Jun/2025 16:15:01] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [21/Jun/2025 16:15:01] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [21/Jun/2025 16:15:01] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [21/Jun/2025 16:15:01] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [21/Jun/2025 16:15:01] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [21/Jun/2025 16:15:01] "GET / HTTP/1.1" 200 -
```

BUT...

- Writing everything from scratch does give you more control over your program.
- But it's **hard**, **time consuming**, and **risky**.
- *"Never reinvent the wheel!"*
- Frameworks come in handy here. They give you some tools and a structure to work with. Like someone has already solved most of the challenges for you.
- Frameworks are like Legos - they provide the tools, rules, and structure you need to build your own application.



Terry A. Davis, was the master of reinventing the wheel! He made his own programming language named "Holy C", then created an entire OS named "TempleOS" in it!

Fun Fact!

Here Comes Django!

A batteries-included framework!

- It makes things easier for you!
- Comes with a vast toolset!
- Has built-in middlewares, ORM, authentication and more!
- Easily integrates with all your other python tools and packages!
- Is used in many big projects such as Gmail and Spotify!

Django is not the only web framework there! There are tons of other frameworks such as ExpressJS, NestJS, DotNet, RocketRS, etc.

Can you name some more?

Fun Fact!