COS30049 – Computing Technology Innovation Project

Week 10 - Introduction to Back-End Development

(Lecture – 01)

Ningran Li (Icey)
ningranli@swin.edu.au

Acknowledgement of Country

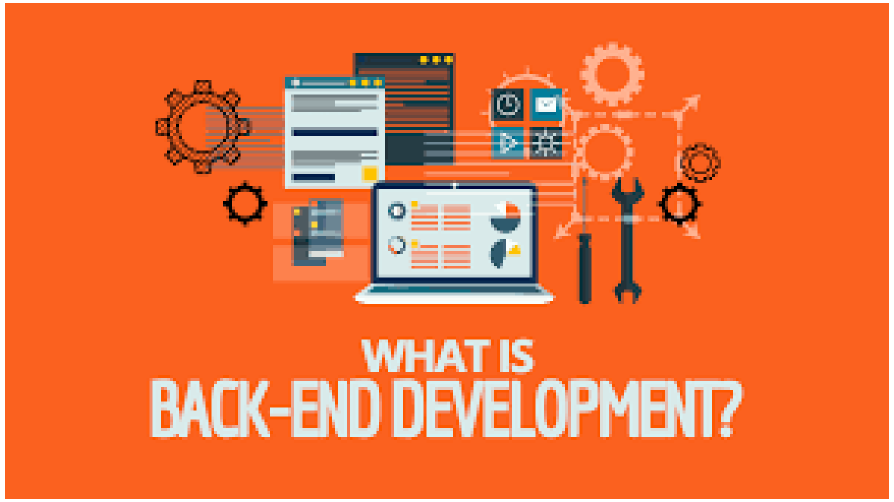
We respectfully acknowledge the Wurundjeri People of the Kulin Nation, who are the Traditional Owners of the land on which Swinburne's Australian campuses are located in Melbourne's east and outer-east, and pay our respect to their Elders past, present and emerging.

We are honoured to recognise our connection to Wurundjeri Country, history, culture, and spirituality through these locations, and strive to ensure that we operate in a manner that respects and honours the Elders and Ancestors of these lands.

We also respectfully acknowledge Swinburne's Aboriginal and Torres Strait Islander staff, students, alumni, partners and visitors.

We also acknowledge and respect the Traditional Owners of lands across Australia, their Elders, Ancestors, cultures, and heritage, and recognise the continuing sovereignties of all Aboriginal and Torres Strait Islander Nations.

- Introduction to Back-End development
- Introduction to HTTP
- Introduction to FastAPI
- Introduction to Swagger & Postman
- Simple Demo



Simple Programmer

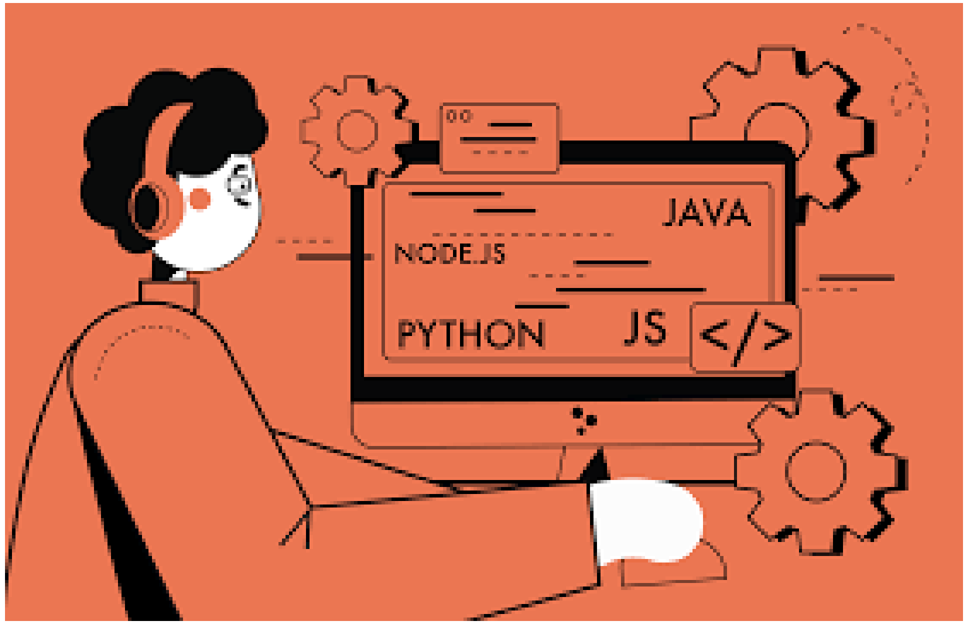
Overview of Back-End development

Definition:

- Back-End development refers to the server-side part of web applications.
- It focuses on the core functionality, databases, server logic, and APIs.

Role in Web Applications:

- Acts as the backbone that supports front-end interactions.
- Manages data storage, security, and business logic.



fiverr blog

Key Responsibilities of Back-End Development

API Development:

- Create APIs（application programming interface） for front-end interaction.
- Ensure smooth data flow between client-side and server-side.

Server-Side Logic:

- Implement business logic and application functionality.
- Handle user authentication, authorization, and session managemer..

Database Management:

- Design, implement, and maintain databases.
- Ensure data integrity and efficient querying.

Orange Developer

Core Technologies in Back-End Development

Programming Languages:

- Popular choices: Python, Java, JavaScript (Node.js), Ruby, PHP.

Frameworks:

- Examples: Django, Spring Boot, Flask, Ruby on Rails.

Databases:

- Types: Relational (e.g., MySQL, PostgreSQL), NoSQL (e.g., MongoDB, Redis).

wiki

Other considerations in Back-End Development

Security:

Developers need to ensure that data is protected from unauthorized access, breaches, and vulnerabilities.

Performance Optimization:

This includes managing server resources efficiently, minimizing latency, and employing techniques like caching and load balancing to handle high traffic.

Scalability:

As web applications grow, back-end systems must be scalable to accommodate increased user demand.

Other considerations in Back-End Development (Cont.)

Integration with Front-End:

This involves developing APIs that facilitate smooth communication between the server and client, managing data flow, and providing the necessary endpoints for front-end functionalities.

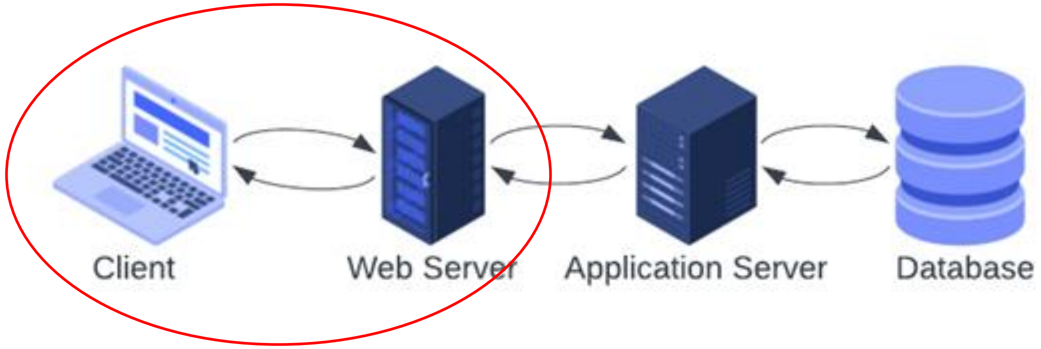
DevOps & Continuous Integration:

Back-End development is increasingly intertwined with DevOps practices, which emphasize continuous integration, deployment, and automated testing.

A Simple Web App Model – Client and Web Server

Client to Web Server:

The client (usually a web browser or a mobile app) initiates communication by sending an HTTP request to the web server. This request can be for various operations such as fetching a web page, submitting a form, or retrieving data via an API endpoint.



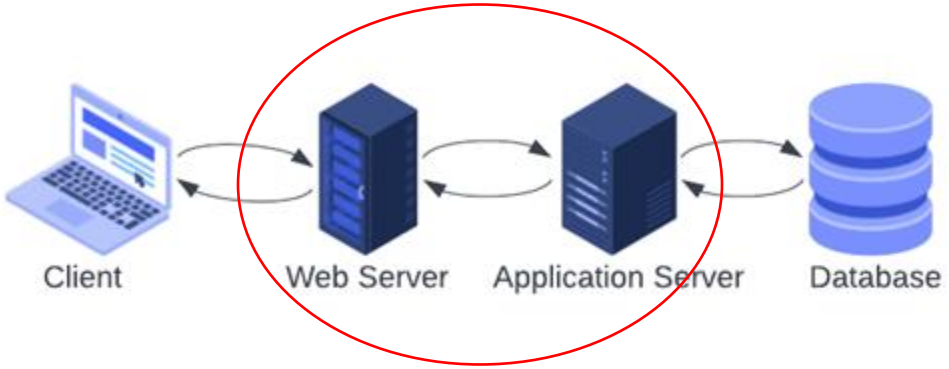
Web Server to Client:

The web server processes the incoming HTTP request, which may involve static content delivery or forwarding the request to the application server. Once the response is ready, the web server sends it back to the client as an HTTP response.

A Simple Web App Model - Web Server and Application Server

Web Server to Application Server:

Upon receiving an HTTP request from the client that requires dynamic processing (e.g., retrieving data from a database or executing business logic), the web server forwards this reque to the appropriate application server.



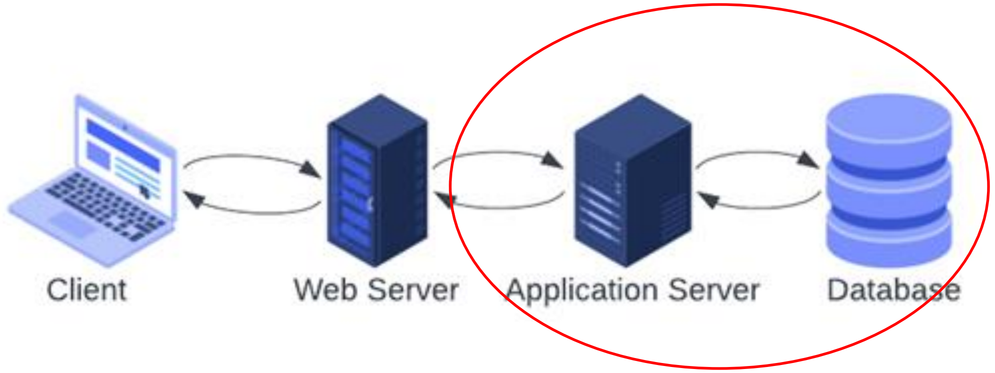
Application Server to Web Server:

The application server processes the request by executing the required business logic, which may involve querying the database or performing computations. After processing, the application server sends the result back to the web server.

A Simple Web App Model - Application Server and Database

Application Server to Database:

When an application server needs to store or retrieve data, it sends a query to the database. This interaction is usually performed via a database query language like SQL. The application server might request data retrieval, insertion, updating, or deletion based on the client's request.



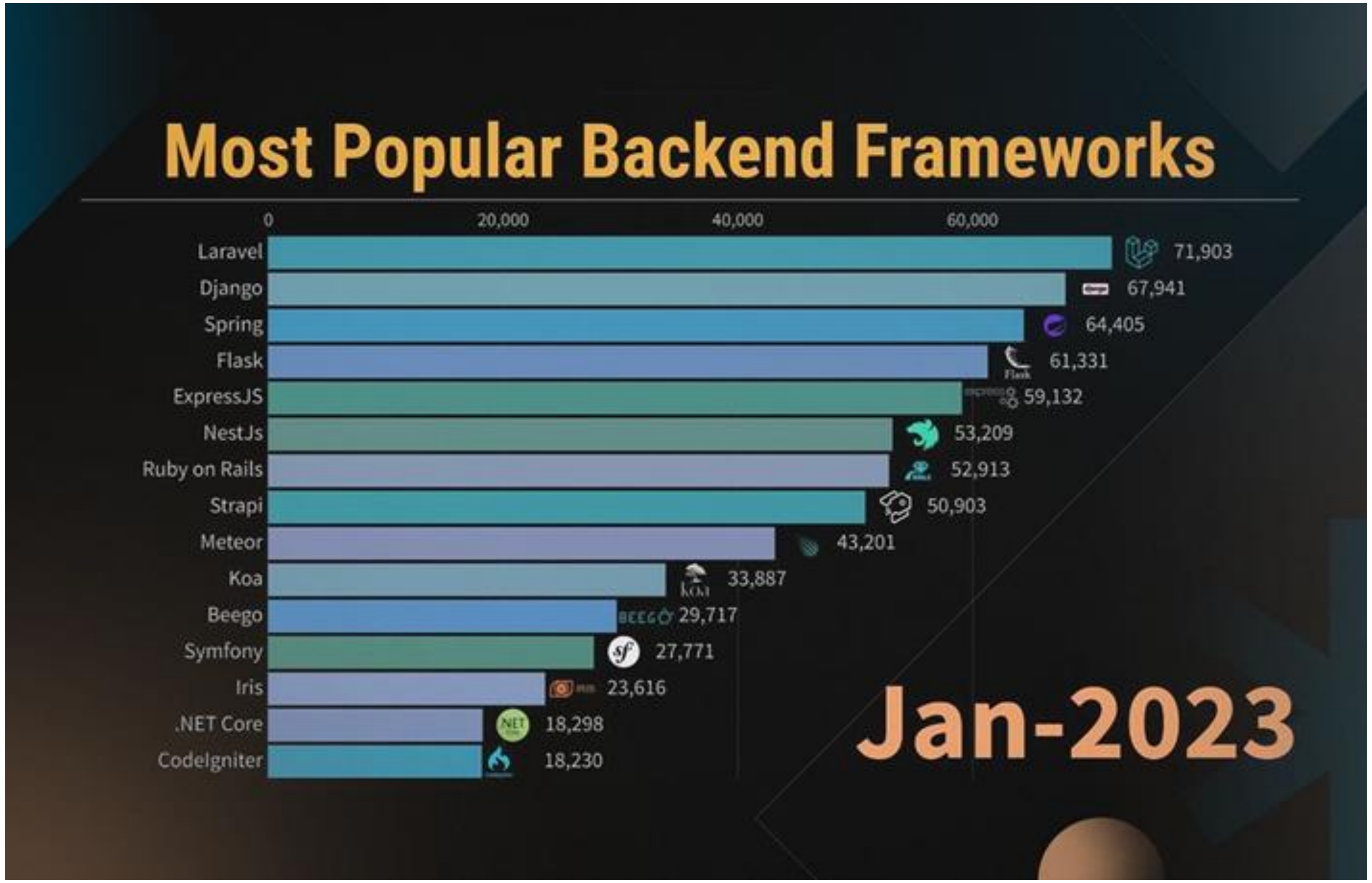
Database to Application Server:

The database processes the query sent by the application server and returns the requested data or confirmation of the executed operation. The application server then uses this data to generate a response for the web server or to further process it according to the application's business logic.

Most popular Backend frameworks

We can choose one of them as the backend framework in this course.

- Laravel
- Django
- Spring
- Flask
- ExpressJS
- …



Acropolium

- Introduction to Back-End development
- Introduction to HTTP
- Introduction to FastAPI
- Introduction to Swagger & Postman
- Simple Demo

The Purpose of HTTP

Definition:

HTTP (Hypertext Transfer Protocol) is a protocol used for transmitting hypertext over the web.

Primary Use:

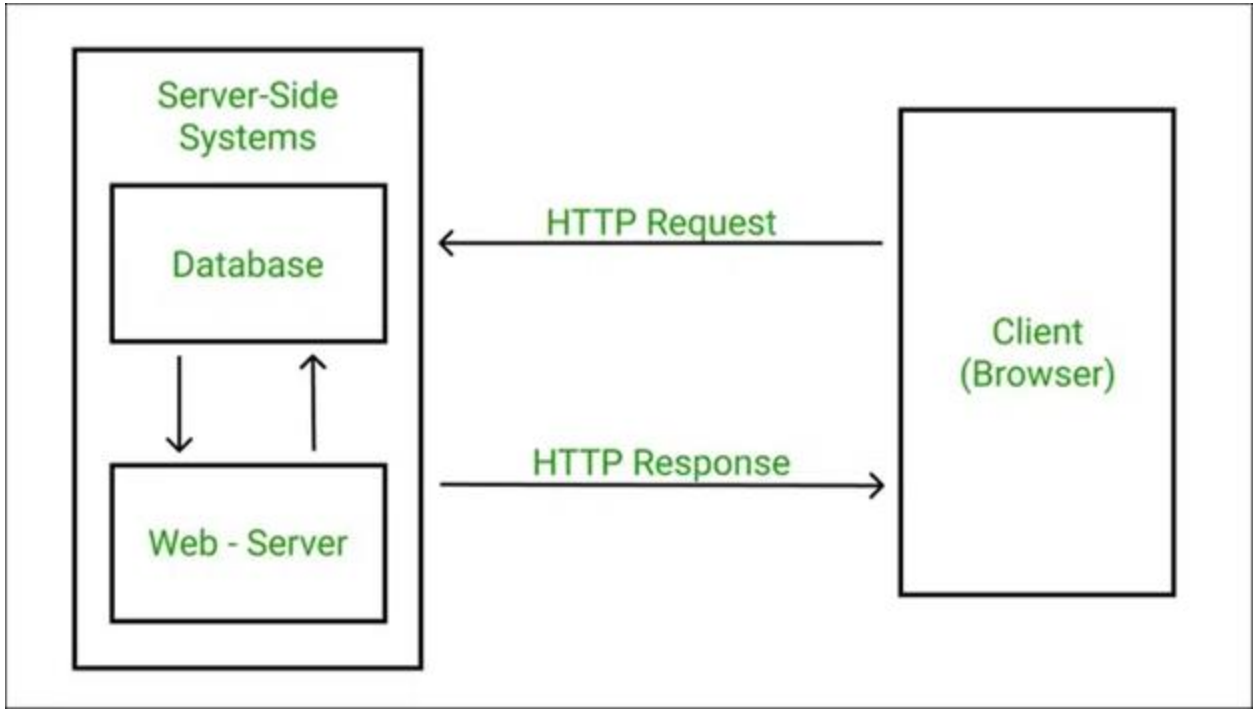
- HTTP is the foundation of data communication on the World Wide Web.
- It allows browsers to fetch resources such as HTML documents, images, and videos from web servers.

Stateless Protocol:

Each HTTP request is independent, meaning the server does not retain any information about previous requests from the same client.

Common Usage:

- Used for sending and receiving data between a client (usually a web browser) and a server.
- HTTP is also the basis for APIs used in web services.



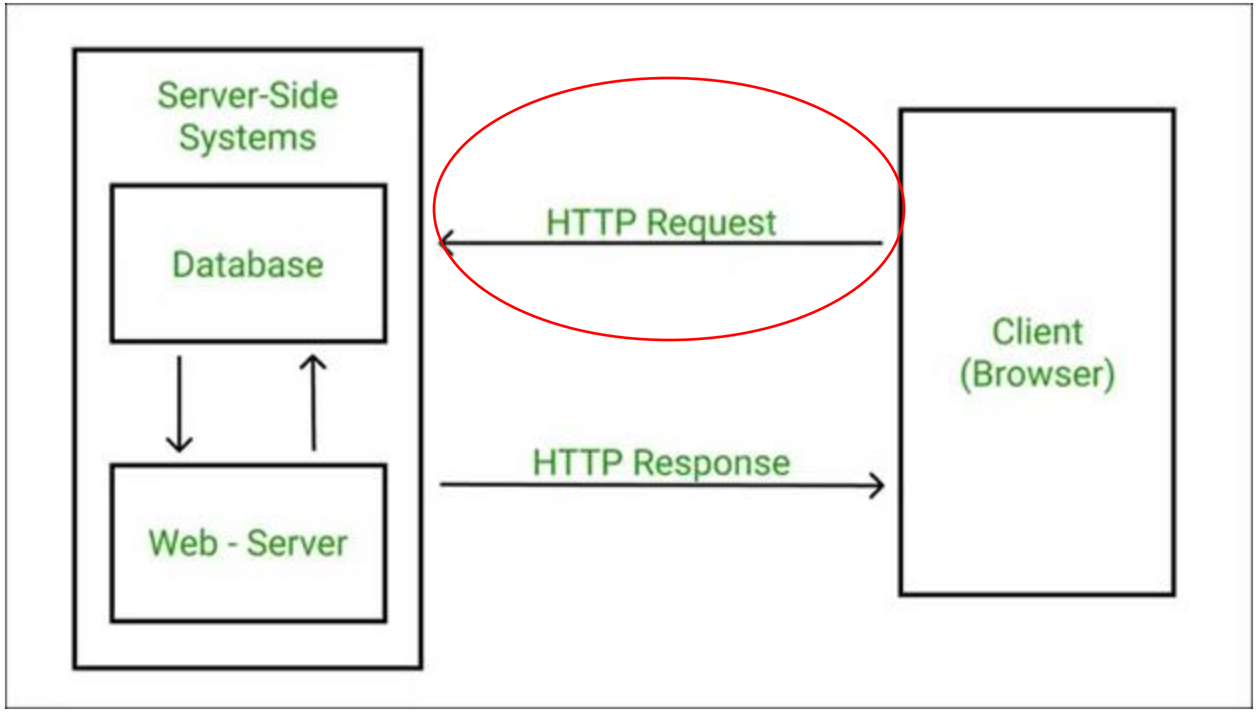
HTTP Request Overview

What is an HTTP Request?:

An HTTP request is sent by the client to request resources or services from a server.

Components of an HTTP Request:

- **Request Line:** Includes the HTTP method (e.g., GET, POST), the URL, and the HTTP version.
- **Headers:** Provide additional information about the request, such as the host, user-agent, and accepted response formats.
- **Body:** Contains data sent to the server, typically used with POST or PUT methods.



HTTP Request - Detailed Components

Request Line:

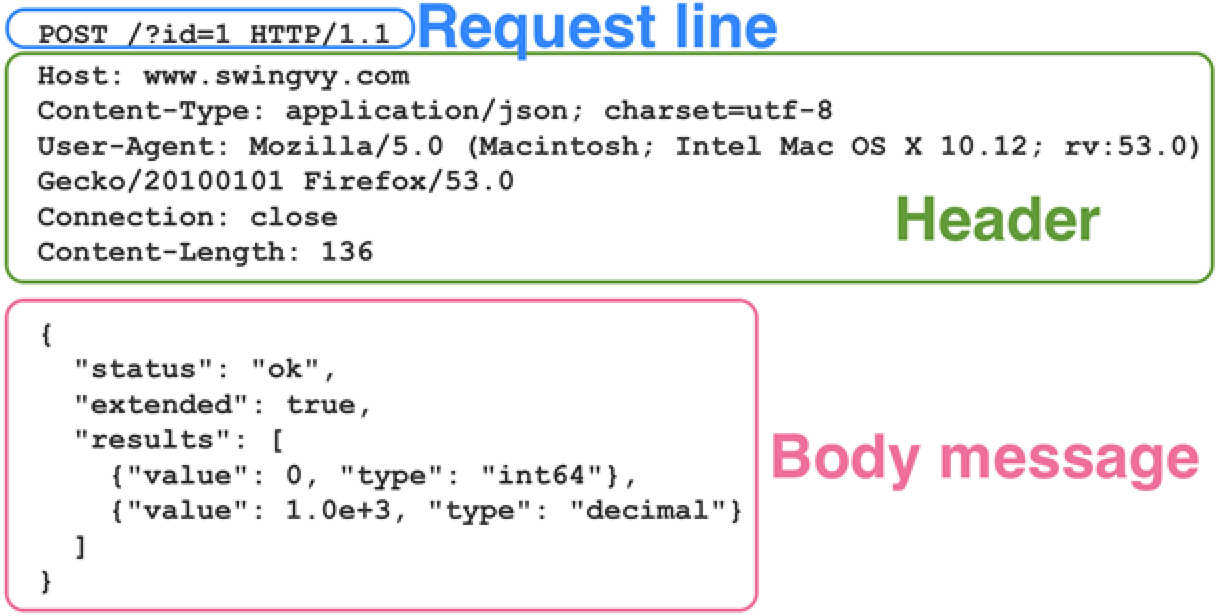
- **Method**: Specifies the action (e.g., GET, POST, PUT, DELETE).
- **URL**: The address of the resource the client is requesting.
- **HTTP Version**: Indicates the version of the HTTP protocol used (e.g., HTTP/1.1).

Headers:

- **Host**: Specifies the domain name of the server.
- **User-Agent**: Identifies the client software (e.g., browser type and version).
- **Accept**: Indicates the content types the client can handle (e.g., text/html).
- **Authorization**: Includes credentials for authentication purposes.

Body:

- Used when data needs to be sent to the server, typically in formats like JSON or form data.



```
POST /?id=1 HTTP/1.1
Host: www.swingvy.com
Content-Type: application/json; charset=utf-8
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.12; rv:53.0)
Gecko/20100101 Firefox/53.0
Connection: close
Content-Length: 136

{
  "status": "ok",
  "extended": true,
  "results": [
    {"value": 0, "type": "int64"},
    {"value": 1.0e+3, "type": "decimal"}
  ]
}
```

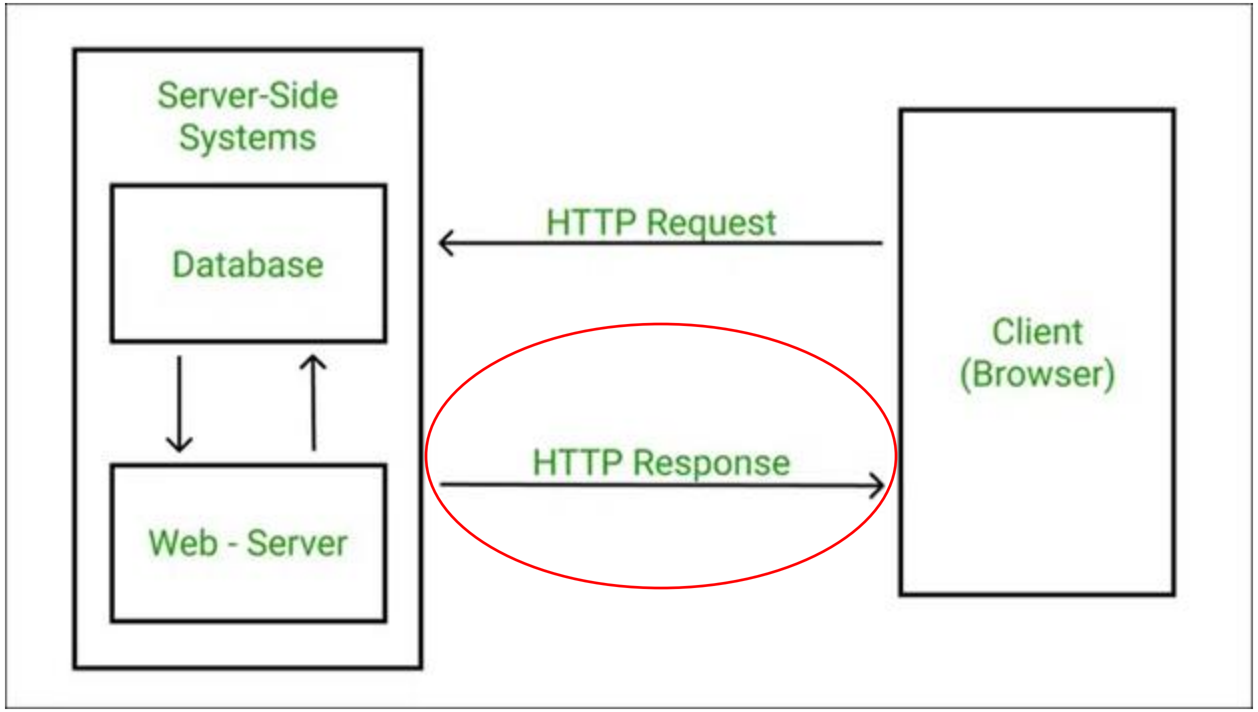
HTTP Response – Overview

What is an HTTP Response?:

An HTTP response is sent by the server in reply to an HTTP request, containing the requested resource or an error message.

Components of an HTTP Response:

- **Status Line:** Includes the HTTP version, status code, and a reason phrase.
- **Headers**: Provide additional information about the response, such as content type and length.
- **Body**: Contains the actual content requested by the client, such as HTML, JSON, or an image.



HTTP Response – Detailed Components

Status Line:

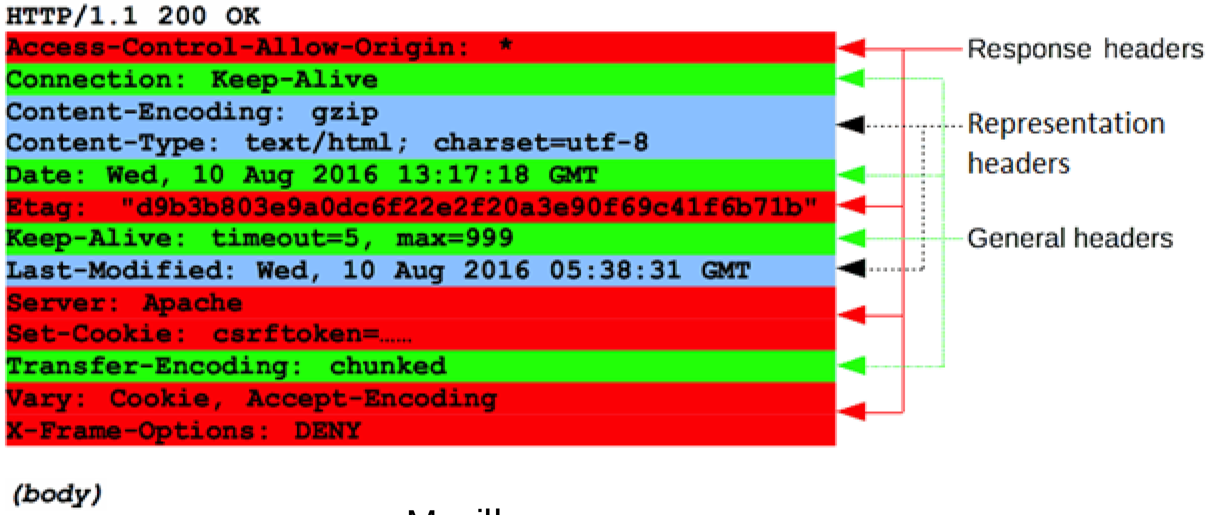
- **HTTP Version**: Indicates the HTTP version used (e.g., HTTP/1.1).
- **Status Code**: A three-digit code indicating the result of the request (e.g., 200 OK, 404 Not Found).
- **Reason Phrase**: A textual description of the status code.

Headers:Content-Type:

- Specifies the media type of the response (e.g., text/html, application/json).
- **Content-Length**: Indicates the size of the response body in bytes.
- **Server**: Provides information about the server software.

Body:

- Contains the resource requested by the client or an error message. This could be an HTML page, JSON data, or a file.



```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
Connection: Keep-Alive
Content-Encoding: gzip
Content-Type: text/html; charset=utf-8
Date: Wed, 10 Aug 2016 13:17:18 GMT
Etag: "d9b3b803e9a0dc6f22e2f20a3e90f69c41f6b71b"
Keep-Alive: timeout=5, max=999
Last-Modified: Wed, 10 Aug 2016 05:38:31 GMT
Server: Apache
Set-Cookie: csrftoken=......
Transfer-Encoding: chunked
Vary: Cookie, Accept-Encoding
X-Frame-Options: DENY

(body)
```

Mozilla

What is an API endpoint & API route?

API endpoint:

An API endpoint is a specific URL or URI (Uniform Resource Identifier) where an API can be accessed by a client application.

Consider an API for a user management system. An endpoint in this API might be **https://api.example.com/users/12345**, where accessing this URL could return information about the user with **ID 12345.**

API route:

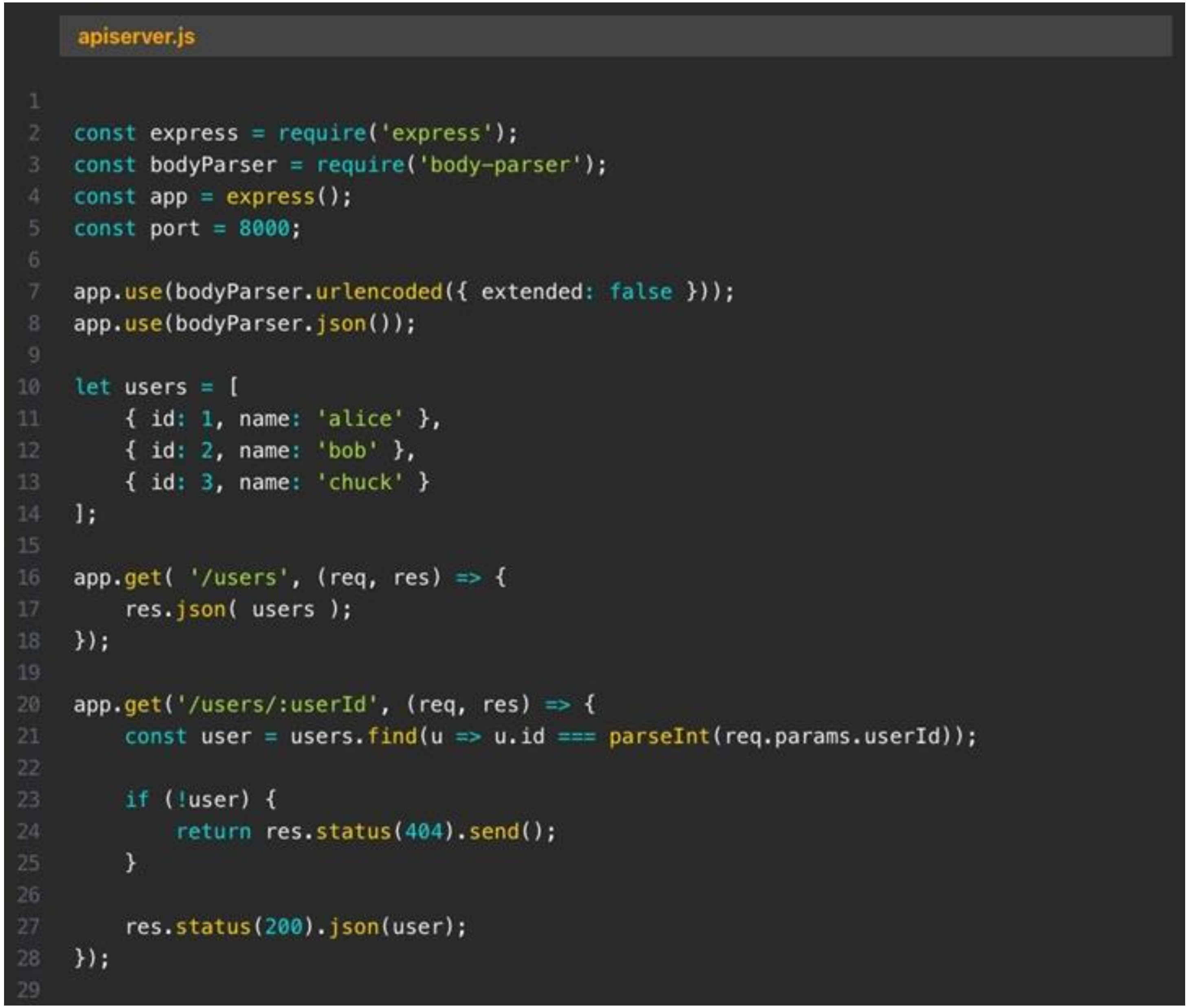
An API route is the definition of paths and methods (like GET, POST, PUT, DELETE) in an API. It's a broader concept that includes the path, the method, and often the logic that gets executed when that path and method are used.

In the same user management system, a route might be defined as **GET /users/:userId**, where **:userId** is a variable part of the path, and GET is the method. This route defines how the application will handle GET requests for any user ID.

Confused? Let’s look at an example.

An endpoint exists at **http://localhost:8000/users**. This returns a collection of all users in the data store.

There exists an endpoint at **http://localhost:8000/users/{userId},** where {userId} represents a unique identifier for a single user. This is sometimes called the singleton object.

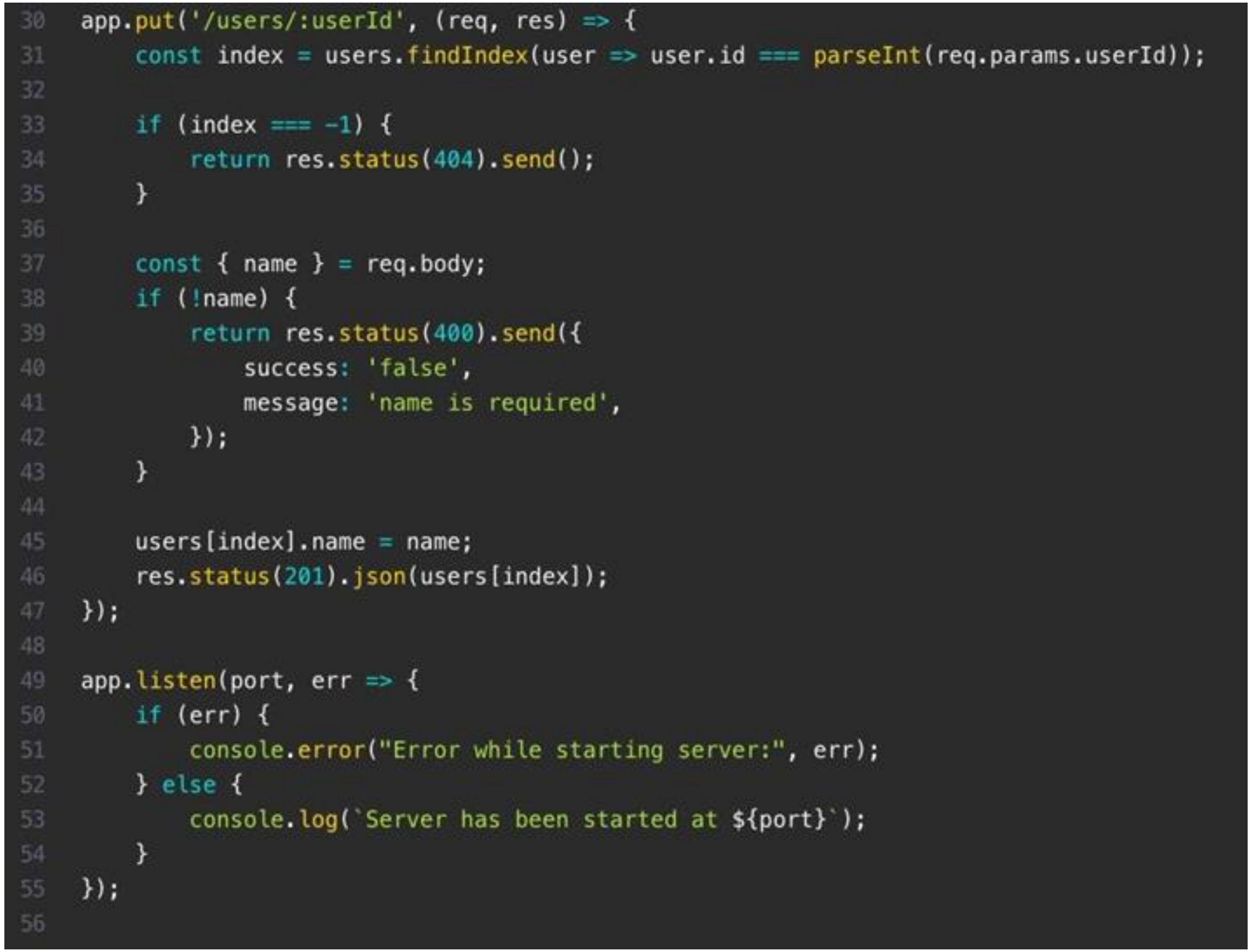


apiserver.js

```

const express = require('express');
const bodyParser = require('body-parser');
const app = express();
const port = 8000;

app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());

let users = [
    { id: 1, name: 'alice' },
    { id: 2, name: 'bob' },
    { id: 3, name: 'chuck' }
];

app.get( '/users', (req, res) => {
    res.json( users );
});

app.get('/users/:userId', (req, res) => {
    const user = users.find(u => u.id === parseInt(req.params.userId));

    if (!user) {
        return res.status(404).send();
    }

    res.status(200).json(user);
});

```

Confused? Let’s look at an example (Cont.)

A route exists for the singleton endpoint for **PUT /users/:userId** that updates a single user.



```
app.put('/users/:userId', (req, res) => {
    const index = users.findIndex(user => user.id === parseInt(req.params.userId));

    if (index === -1) {
        return res.status(404).send();
    }

    const { name } = req.body;
    if (!name) {
        return res.status(400).send({
            success: 'false',
            message: 'name is required',
        });
    }

    users[index].name = name;
    res.status(201).json(users[index]);
});

app.listen(port, err => {
    if (err) {
        console.error("Error while starting server:", err);
    } else {
        console.log(`Server has been started at ${port}`);
    }
});
```

Confused? Let’s look at an example (Cont.)

It’s a very small but distinct difference. You can clearly see that while there is a single API endpoint at **http://localhost:8000/users/{userId}**, there are two routes that serve it. They have entirely different code execution paths. If the API supported adding users (POST) or deleting users (DELETE), there could be even more routes to the same API endpoint.

See the difference? While an API endpoint may represent a single path, the code execution behind it is mapped to the action methods of the service (GET, POST, PUT, DELETE, etc). As such, a potential vulnerability in the code for one route may be very different than another. But they are tied to the same endpoint.

Thank you !

SWINBURNE UNIVERSITY OF TECHNOLOGY