NYCU GDSC &
Backend Program -1

What is **Backend** Development?

GDSC Backend Outline

- Fundamentals of Backend Development and API Design
- Database Design and Management
- Security and Authentication
- Containerization and Deployment
- System Design and Scalable Architecture
- Final Project

GDSC Backend Outline

- **Fundamentals of Backend Development and API Design**
  - Familiarity with Express.js for building RESTful APIs
  - Designing API routes and middleware for request handling

GDSC Backend Outline

- **Database Design and Management**
  - Choosing between relational and NoSQL databases
  - Executing and optimizing SQL queries
  - Designing database schemas with ORMs (optional)

GDSC Backend Outline

- **Security and Authentication**
  - Implementing JWT, OAuth, and RBAC.
  - Preventing attacks such as XSS, CSRF, and SQL Injection

GDSC Backend Outline

- **Containerization and Deployment**
  - Using Docker for application containerization
  - Building CI/CD pipelines (with GitHub Actions)

GDSC Backend Outline

- **System Design and Scalable Architecture**
  - Understanding microservices architectures
  - Addressing high concurrency, load balancing, and caching.

Backend Introduction

Appier

Software Engineer,
Backend Development

About you

[Minimum qualifications]

- BS/BA degree in Computer Science or related field with 3+ years experience in related industry
- Ability to build web services on Linux.
- Good at any of the listed language: Python / Scala / Go/Node.js.
- Good knowledge of Network API Design (e.g. REST or GraphQL).
- Good understanding of any SQL/NoSQL database (MySQL / MongoDB / Redis / etc.)
- Familiar with git.
- Team player and able to work independently.
- Proactive, good interpersonal and problem-solving skill.

[Preferred qualifications]

- MS degree in Computer Science or related field.
- Good at profiler and debugging tools.
- High performance network service on Linux.
- Design and architect large scale distributed system.
- Design and implement distributed algorithm and data structure.
- Familiar with HTML and Javascript.
- Familiar with Nginx / HAProxy.
- Familiar with operation automation tool (such as Ansible).
- Familiar with continuous integration / continuous deployment
- Familiar with monitoring and alert system (Prometheus / Nagios).
- Familiar with functional programming.
- Familiar with Amazon Web Service or Google Compute Engine.

Traditional Architecture

Frontend and backend are part of the same codebase.

Easier to start but harder to scale.

Mordern Architecture

Frontend-Backend separation architecture

Next.js Fullstack Integration

NEXT.JS

Routes Handlers

Think

What is the purpose of Next.js Routes Handlers in the trend of front-end and back-end separation?

Think

What is the purpose of Next.js Routes Handlers in the trend of front-end and back-end separation?

- **Perfect for prototyping or small projects.**
- Next.js’s getServerSideProps and getStaticProps handle server-side data

Node.js Introduction

Node.js history

Before 2009 (Traditional Server Models)

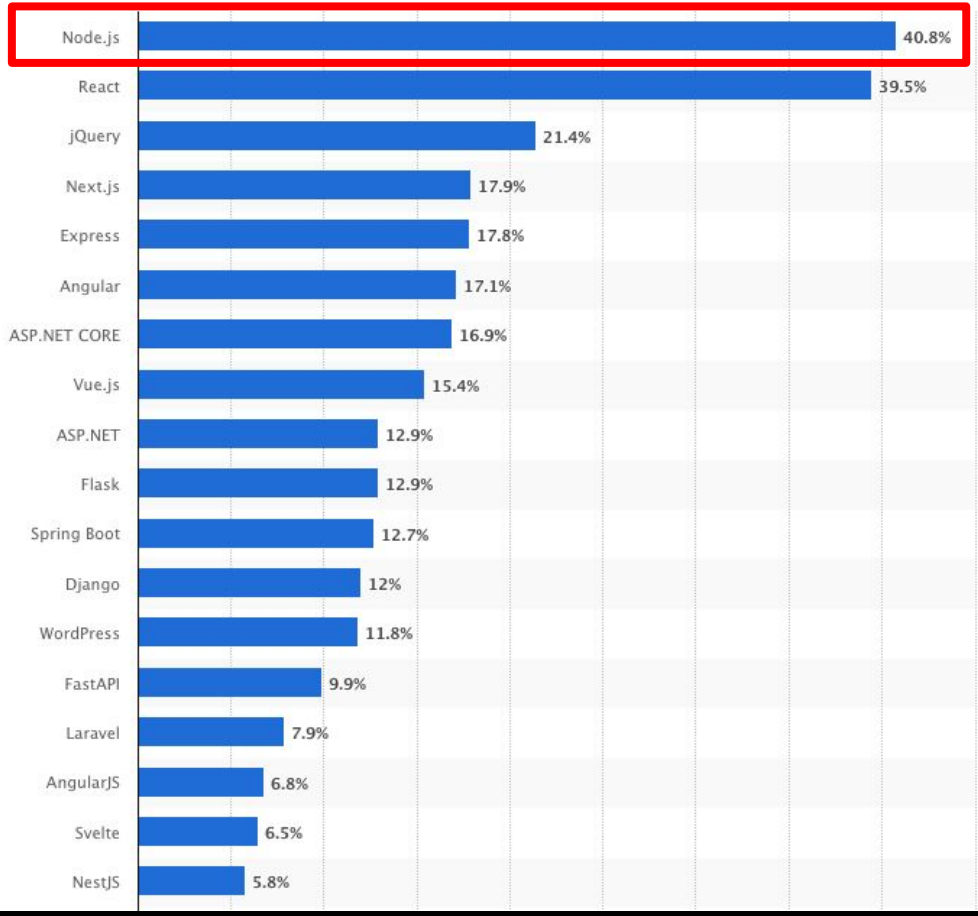
Thread-based servers (Apache) →
Each request creates a new thread/process, consuming high memory.

Issue: Traditional servers struggled with **high concurrency** and **real-time** processing.

After 2009 (Node.js Revolution)

Event-driven, non-blocking I/O →
Single-threaded but handles thousands of connections efficiently.

Most used web frameworks among developers worldwide, as of 2024



DOWNLOAD

PDF + XLS + PNG + PPT +

Source

→ Show sources information
→ Show publisher information
→ Use Ask Statista Research Service

Release date
July 2024

Region
Worldwide

Survey time period
May 19 to June 20, 2024

Number of respondents
48,503 respondents

Special properties
Software developers

Method of interview
Online survey

Supplementary notes

Original question: "Which web frameworks

Node.js Benefit

- High Performance & Fast Execution
- Non-Blocking & Event-Driven Architecture
- Large & Active Community
- Scalable & Lightweight
- Real-Time Capabilities

High Performance & Fast Execution

- Built on **Google V8 Engine**, which compiles JavaScript into machine code for faster execution.
- Non-blocking **asynchronous I/O** allows handling multiple requests simultaneously.

Non-Blocking & Event-Driven Architecture

- Uses **Event Loop** to manage multiple requests without creating multiple threads.
- Ideal for **I/O-heavy applications** like APIs, real-time apps, and streaming services.

Large & Active Community

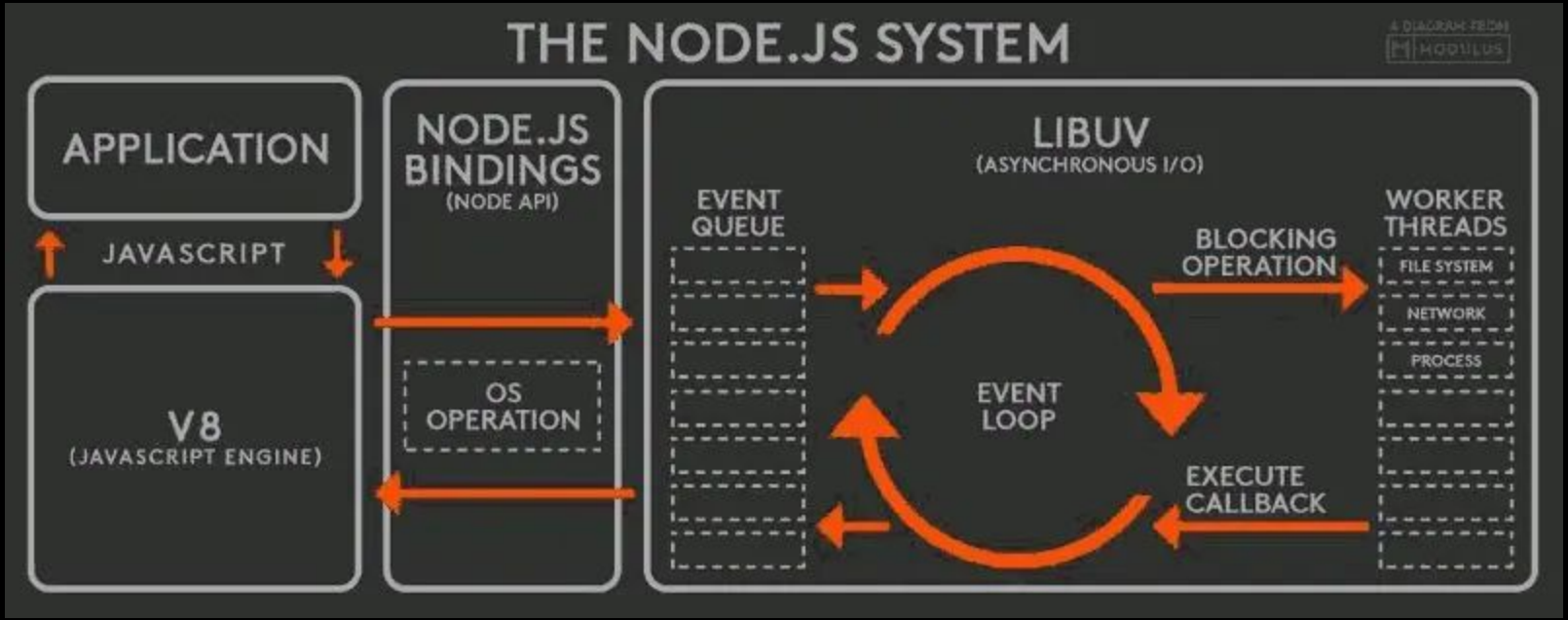
- Rich ecosystem with **millions of open-source libraries** via npm (Node Package Manager).
- Strong support from developers and large companies like Netflix, PayPal, and LinkedIn.

Scalable & Lightweight

- Suitable for **microservices architecture**, allowing independent service scaling.
- Can handle a large number of concurrent connections efficiently.

Real-Time Capabilities

- Built-in support for **WebSockets** for real-time communication.
- Perfect for applications like gaming, live streaming, and collaborative tools.


THE NODE.JS SYSTEM
A DIAGRAM FROM
MODULUS
APPLICATION
JAVASCRIPT
V8
(JAVASCRIPT ENGINE)
NODE.JS
BINDINGS
(NODE API)
OS
OPERATION
LIBUV
(ASYNCHRONOUS I/O)
EVENT
QUEUE
EVENT
LOOP
BLOCKING
OPERATION
EXECUTE
CALLBACK
WORKER
THREADS
FILE SYSTEM
NETWORK
PROCESS

Express.js Introduction

Fast, unopinionated, minimalist web framework for Node.js

Express.js core feature

- Routing
- Middleware
- Request and Response Handling

Routing

Node.js

```
const http = require('http');

const server = http.createServer((req, res) => {
    if (req.method === 'GET' && req.url === '/users') {
        res.writeHead(200, { 'Content-Type': 'text/plain' });
        res.end('Fetching all users');
    } else if (req.method === 'POST' && req.url === '/users') {
        res.writeHead(200, { 'Content-Type': 'text/plain' });
        res.end('Creating a new user');
    } else if (req.method === 'GET' && req.url.startsWith('/users/')) {
        const userId = req.url.split('/')[2];
        res.writeHead(200, { 'Content-Type': 'text/plain' });
        res.end(`Fetching user with ID: ${userId}`);
    } else {
        res.writeHead(404);
        res.end('Not Found');
    }
});

server.listen(3000, () => console.log('Server running on port 3000'));
```

Express.js

```
const express = require('express');
const app = express();

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

app.post('/users', (req, res) => {
    res.send('Creating a new user');
});

app.get('/users/:id', (req, res) => {
    res.send(`Fetching user with ID: ${req.params.id}`);
});

app.listen(3000, () => console.log('Server running on port 3000'));
```

Express allows handling different URLs and HTTP methods separately.

Middleware

Express.js

```
const express = require('express');
const app = express();

const authMiddleware = (req, res, next) => {
    if (req.query.auth === 'secret') {
        next();
    } else {
        res.status(401).send('Unauthorized');
    }
};

app.get('/protected', authMiddleware, (req, res) => {
    res.send('You have access to the protected route');
});

app.listen(3000, () => console.log('Server running on port 3000'));
```

Middleware functions can be used for logging, authentication, error handling, and more.

Request and Response Handling

Node.js

```
const http = require('http');

const server = http.createServer((req, res) => {
    if (req.method === 'POST' && req.url === '/login') {
        let body = '';

        req.on('data', chunk => {
            body += chunk.toString();
        });

        req.on('end', () => {
            try {
                const { username, password } = JSON.parse(body);

                if (username === 'admin' && password === 'password') {
                    res.writeHead(200, { 'Content-Type': 'application/json' });
                    res.end(JSON.stringify({ message: 'Login successful' }));
                } else {
                    res.writeHead(401, { 'Content-Type': 'application/json' });
                    res.end(JSON.stringify({ message: 'Invalid credentials' }));
                }
            } catch (error) {
                res.writeHead(400, { 'Content-Type': 'application/json' });
                res.end(JSON.stringify({ message: 'Invalid JSON' }));
            }
        });
    } else {
        res.writeHead(404);
        res.end('Not Found');
    }
});

server.listen(3000, () => console.log('Server running on port 3000'));
```

Express.js

```
const express = require('express');
const app = express();

app.use(express.json());

app.post('/login', (req, res) => {
    const { username, password } = req.body;

    if (username === 'admin' && password === 'password') {
        res.json({ message: 'Login successful' });
    } else {
        res.status(401).json({ message: 'Invalid credentials' });
    }
});

app.listen(3000, () => console.log('Server running on port 3000'));
```

Express provides req and res objects to manage data.

Nice tools for Node.js Backend development

NVM

- Easily install and switch between different Node.js versions
- Manage project-specific Node.js environments

PM2

- Production process manager for Node.js applications
- Keeps applications alive forever and handles automatic restarts
- Load balancing and performance monitoring

Nodemon

- Monitors changes in your source code and automatically restarts server
- Speeds up the development process

Postman

- Testing APIs with various HTTP methods
- Creating and saving request collections

Try your Fisrt api

1. Install Node.js and npm

```
node -v
npm -v
```

Try your Fisrt api

2. Create a New Project

```
mkdir express-demo

cd express-demo

npm init -y
```

Try your Fisrt api

3. Install Express

```
npm install express
```

Try your Fisrt api

4. Create server.js

```
const express = require('express');
const app = express();
const port = 3000;

app.get('/', (req, res) => {
    res.send('Hello, Express!');
});

app.listen(port, () => {
    console.log(`Server is running at
http://localhost:${port}`);
});
```

Try your Fisrt api

5. Create server.js

node server.js

Open a browser and go to `http://localhost:3000`, you should see:

Hello, Express!

Try your Fisrt api

6. Add api routes

```
app.get('/api/users', (req, res) =>
{   res.json([
        { id: 1, name: 'Alice' },
        { id: 2, name: 'Bob' }
    ]);
});
```

Test `http://localhost:3000/api/users`, it should return JSON data.

Homework

1. Final Project Thinking
2. Building a Simple Todo List RESTful API
3. Topic Discussion Prepare

Discussion- SGary

1. **RESTful API Design Principles**
   a. What is REST?
   b. Six principles of RESTful API design

2. **Best Practices for API Endpoints**
   a. How to design clear URL paths? (nouns vs. verbs, `/users/{id}` vs `/getUser`)
   b. How to handle versioning? (`/v1/` vs `/v2/`)

Discussion- Joanne

1. **HTTP Methods & Status Codes**
   a. Appropriate use cases for GET, POST, PUT, DELETE, PATCH
   b. Common HTTP status codes (200, 201, 204, 400, 401, 403, 404, 500)

2. **Error Handling & Response Format**
   a. How to standardize error formats? (e.g., `{ "error": "Invalid request", "code": 400 }`)
   b. What is API pagination? (Limit, Offset, Cursor-based pagination)

Discussion- Kai

1. **Authentication vs Authorization Basics**
    a. Authentication (JWT, OAuth) vs Authorization (RBAC, ABAC)
    b. Why do APIs need authentication?

2. **What is CORS?**

Discussion- Max

1. **API Security Best Practices**
    a. Preventing SQL Injection, XSS, CSRF

2. **What is Database Normalization?**