CLOUDY

ĐỨNG ĐẦU TRÊN NHỮNG ĐÁM MÂY

PREPARED BY

Nguyen Nhu Khanh

ĐƯỢC SOẠN BỞI

Nguyễn Như Khánh

NỘI DUNG:

1. OOP:

Trong TypeScript, OOP (Object-Oriented Programming) cung cấp một số tính năng cho phép chúng ta tạo ra các đối tượng và thực hiện các hành động trên đối tượng đó. Sau đây là một số kiến thức cơ bản về OOP trong TypeScript và ví dụ minh họa cho mỗi cái:

1.1. Lớp (Class):

Là một cấu trúc để định nghĩa các đối tượng với các thuộc tính và phương thức.

Ví dụ:

```
class Person {
    name: string;
    age: number;

    constructor(name: string, age: number) {
      this.name = name;
      this.age = age;
    }

    sayHello() {
       console.log(`Hello, my name is ${this.name} and I am ${this.age} years
old`);
    }
  }

  const person = new Person("John", 30);
  person.sayHello(); // Hello, my name is John and I am 30 years old
```

1.2. Đối tượng (Object):

Là một thực thể của một lớp với các thuộc tính và phương thức.

Ví dụ:

```
class Car {
    make: string;
    model: string;

    constructor(make: string, model: string) {
        this.make = make;
        this.model = model;
    }

    startEngine() {
        console.log(`Starting engine of ${this.make} ${this.model}`);
    }
}

const car = new Car("Toyota", "Corolla");
car.startEngine(); // Starting engine of Toyota Corolla
```

1.3. Đặc tính của OOP:

1.3.1. Kế thừa (Inheritance):

Cho phép một lớp kế thừa các thuộc tính và phương thức của một lớp khác.

Ví dụ:

```
class Animal {
    name: string;

    constructor(name: string) {
        this.name = name;
    }

    speak() {
        console.log(`${this.name} makes a noise.`);
    }
}

class Dog extends Animal {
    constructor(name: string) {
        super(name);
    }

    speak() {
        console.log(`${this.name} barks.`);
    }
}

const dog = new Dog("Max");
dog.speak(); // Max barks.
```

1.3.2. Đóng gói (Encapsulation):

Cho phép giới hạn truy cập đến các thuộc tính và phương thức của một lớp.

Ví dụ:

```
class BankAccount {
    private balance: number;

    constructor(balance: number) {
        this.balance = balance;
    }

    deposit(amount: number) {
        this.balance += amount;
    }

    withdraw(amount: number) {
        if (amount > this.balance) {
            console.log("Insufficient balance");
        } else {
            this.balance -= amount;
        }
    }

    getBalance() {
        return this.balance;
    }
}

const account = new BankAccount(100);
account.deposit(50);
account.withdraw(75);
console.log(account.getBalance()); // 75
```

1.3.3. Đa hình (Polymorphism):

Cho phép một đối tượng có thể thể hiện các hành động khác nhau tùy theo ngữ cảnh.

Ví dụ:

```
class Shape {
    getArea() {
        return 0;
    }
}

class Rectangle extends Shape {
    constructor(private width: number, private height: number) {
        super();
    }

    getArea() {
        return this.width * this.height;
    }
}

class Circle extends Shape {
    constructor(private radius: number) {
        super();
    }

    getArea() {
        return Math.PI * this.radius ** 2;
    }
}

function printArea(shape: Shape) {
    console.log(`The area is ${shape.getArea()}`);
}

const rectangle = new Rectangle(10, 5);
const circle = new Circle(5);

printArea(rectangle); // The area is 50
printArea(circle); // The area is 78.53981633974483
```

1.3.4. Trừu tượng:

Là khả năng tách biệt giữa giao diện (interface) và implement (triển khai). Khi ta thiết kế một đối tượng, ta không cần phải quan tâm đến chi tiết cài đặt bên trong của đối tượng đó, mà chỉ quan tâm đến giao diện của nó.

Ví dụ:

```
interface Geometry {
    getArea(): number;
    getPerimeter(): number;
}

class Square implements Geometry {
    constructor(private side: number) { }

    getArea(): number {
        return this.side * this.side;
    }

    getPerimeter(): number {
        return 4 * this.side;
    }
}

class Circle implements Geometry {
    constructor(private radius: number) { }

    getArea(): number {
        return Math.PI * this.radius * this.radius;
    }

    getPerimeter(): number {
        return 2 * Math.PI * this.radius;
    }
}
const square = new Square(5);
const circle = new Circle(3);

console.log(square.getArea()); // Kết quả: 25
console.log(square.getPerimeter()); // Kết quả: 20

console.log(circle.getArea()); // Kết quả: 28.274333882308138
console.log(circle.getPerimeter()); // Kết quả: 18.84955592153876
```

1.4. Giao diện (Interface):

Là một bộ khai báo về các thuộc tính và phương thức mà một đối tượng cần phải có.

Ví dụ:

```typescript
interface Shape {
    getArea(): number;
}

class Rectangle implements Shape {
    constructor(private width: number, private height: number) { }

    getArea() {
        return this.width * this.height;
    }
}

class Circle implements Shape {
    constructor(private radius: number) { }

    getArea() {
        return Math.PI * this.radius ** 2;
    }
}

function printArea(shape: Shape) {
    console.log(`The area is ${shape.getArea()}`);
}

const rectangle = new Rectangle(10, 5);
const circle = new Circle(5);

printArea(rectangle); // The area is 50
printArea(circle); // The area is 78.53981633974483
```

2. Design Pattern:

NestJS sử dụng nhiều Design Pattern trong quá trình phát triển ứng dụng, nhưng điểm chung của chúng đều là để giúp cho ứng dụng được thiết kế và triển khai theo cách có hệ thống, dễ bảo trì và mở rộng.

2.1. Dependency Injection Pattern:

Cho phép chúng ta tự động đưa các đối tượng cần thiết vào các lớp, giúp giảm độ phức tạp của việc tạo các đối tượng.

Ví dụ:

```
@Injectable()
class AuthService {
    constructor(private readonly usersService: UsersService) { }

    async validateUser(username: string, password: string): Promise<any> {
        const user = await this.usersService.findOne(username);
        if (user && user.password === password) {
            const { password, ...result } = user;
            return result;
        }
        return null;
    }
}
```

2.2. Decorator Pattern:

Là một cách để thêm thông tin cho một lớp hoặc phương thức bằng cách sử dụng các từ khóa đặc biệt.

Ví dụ:

```
@Controller('cats')
export class CatsController {
    constructor(private catsService: CatsService) { }

    @Get()
    async findAll(): Promise<Cat[]> {
        return this.catsService.findAll();
    }
}
```

2.3. Repository Pattern:

Là một cách để trừu tượng hóa cách truy cập dữ liệu, giúp giảm độ phụ thuộc vào cơ sở dữ liệu và dễ dàng kiểm soát việc truy cập dữ liệu.

Ví dụ:

```
@Injectable()
export class CatsService {
    constructor(
        @InjectRepository(Cat)
        private readonly catRepository: Repository<Cat>,
    ) { }

    async findAll(): Promise<Cat[]> {
        return this.catRepository.find();
    }

    async create(cat: Cat): Promise<void> {
        await this.catRepository.insert(cat);
    }
}
```

2.4. Singleton Pattern:

Là một cách để đảm bảo rằng chỉ có một đối tượng được tạo ra và sử dụng trong toàn bộ ứng dụng.

Ví dụ:

```
@Injectable({ scope: Scope.DEFAULT })
export class AppService {
    private readonly data: string[] = [];

    addData(data: string) {
        this.data.push(data);
    }

    getData() {
        return this.data;
    }
}

@Module({
    providers: [AppService],
    exports: [AppService],
})
export class AppModule { }
```

2.5. Factory Pattern:

Là một cách để tạo ra các đối tượng dựa trên một số thông tin đầu vào và giúp tăng tính linh hoạt của ứng dụng.

Ví dụ:

```
export interface Pizza {
    name: string;
    price: number;
}

export class MargheritaPizza implements Pizza {
    name = 'Margherita';
    price = 5;
}

export class PepperoniPizza implements Pizza {
    name = 'Pepperoni';
    price = 6;
}

export class PizzaFactory {
    createPizza(type: string): Pizza {
        if (type === 'Margherita') {
            return new MargheritaPizza();
        } else if (type === 'Pepperoni') {
            return new PepperoniPizza();
        }
        throw new Error(`Unknown pizza type "${type}"`);
    }
}
```

2.6. Adapter Pattern:

Là một cách để chuyển đổi các giao diện khác nhau thành một giao diện thống nhất, giúp cho các đối tượng có thể tương tác với nhau một cách dễ dàng hơn.

Ví dụ:

```
interface PaymentProvider {
    charge(amount: number): Promise<void>;
}

class StripePaymentProvider implements PaymentProvider {
    charge(amount: number): Promise<void> {
        // Charge the payment using the Stripe API
        return Promise.resolve();
    }
}

class PaypalPaymentProvider {
    chargePayment(amount: number): void {
        // Charge the payment using the PayPal API
    }
}

class PaypalPaymentAdapter implements PaymentProvider {
    constructor(private paypalPaymentProvider: PaypalPaymentProvider) { }

    charge(amount: number): Promise<void> {
        this.paypalPaymentProvider.chargePayment(amount);
        return Promise.resolve();
    }
}
```

3. Các kiến thức cơ bản của NodeJS:

3.1. Asynchronous Programming:

Node.js sử dụng phong cách lập trình không đồng bộ (asynchronous) để tối đa hóa việc sử dụng tài nguyên và tăng hiệu suất. Việc sử dụng callback và Promise là hai phương thức phổ biến để xử lý các hoạt động không đồng bộ.

3.2. Module System:

Node.js cung cấp một hệ thống module để quản lý các đoạn mã, giúp tái sử dụng và phân tách chúng. Bạn có thể sử dụng các module có sẵn hoặc tự tạo các module riêng cho ứng dụng của mình.

3.3. File System:

Node.js cho phép truy cập vào hệ thống tệp để đọc và ghi các tệp tin. Bạn có thể sử dụng module fs để thao tác với các tệp tin, thư mục và quản lý chúng.

3.4. Networking:

Node.js cho phép xây dựng các ứng dụng mạng bằng cách sử dụng các giao thức như HTTP, HTTPS, TCP và UDP. Bạn có thể sử dụng module net để tạo các máy chủ mạng và kết nối mạng.

3.5. Stream:

Node.js hỗ trợ các stream để xử lý các tệp tin hoặc dữ liệu lớn, giúp giảm thiểu bộ nhớ và tăng hiệu suất của ứng dụng.

3.6. Child Processes:

Node.js cho phép tạo ra các tiến trình con để xử lý các công việc phức tạp hoặc chạy các lệnh hệ thống bên ngoài. Bạn có thể sử dụng module child_process để tạo các tiến trình con.

4. Tôi có một bài toán: Làm thế nào để tính quảng đường đi ngắn nhất trong Google Map. Thuật toán ở đây là gì? Cách giải quyết như nào vậy?

Trả lời:

Để tính toán quảng đường đi ngắn nhất trong Google Map, chúng ta cần sử dụng thuật toán Dijkstra. Thuật toán Dijkstra là một thuật toán tìm kiếm đường đi ngắn nhất trong đồ thị có trọng số không âm. Nó được sử dụng rộng rãi trong các ứng dụng GPS, các ứng dụng định tuyến mạng và các ứng dụng tối ưu hóa khác.

Để giải quyết bài toán này trong TypeScript, ta có thể sử dụng thư viện **googlemaps** để lấy dữ liệu địa lý của các điểm cần tính toán và tính toán quãng đường sử dụng thuật toán Dijkstra. Ví dụ sau sử dụng googlemaps để lấy dữ liệu về các điểm và tính toán quãng đường đi ngắn nhất giữa chúng:

```
import * as googlemaps from '@google/maps';
import { dijkstra } from './dijkstra';

const googleMapsClient = googlemaps.createClient({
    key: 'YOUR_API_KEY_HERE',
});

// Lấy tọa độ của các điểm trên bản đồ
async function getCoordinates(addresses: string[]):
Promise<googlemaps.LatLng[]> {
    const results = await Promise.all(
        addresses.map((address) =>
            googleMapsClient.geocode({
                address,
            }).asPromise()
        )
    );

    return results.map((result) => result.json.results[0].geometry.location);
}
```

```
// Tính toán quãng đường đi ngắn nhất sử dụng thuật toán Dijkstra
function calculateShortestDistance(
    startAddress: string,
    endAddress: string,
    intermediateAddresses: string[]
): number {
    const addresses = [startAddress, ...intermediateAddresses, endAddress];

    // Lấy tọa độ của các điểm trên bản đồ
    const coordinates = await getCoordinates(addresses);

    // Tính toán khoảng cách giữa các điểm
    const distances: number[][] = [];

    for (let i = 0; i < coordinates.length; i++) {
        distances[i] = [];

        for (let j = 0; j < coordinates.length; j++) {
            if (i === j) {
                distances[i][j] = 0;
            } else {
                const distance = googleMapsClient.distanceMatrix({
                    origins: [coordinates[i]],
                    destinations: [coordinates[j]],
                });

                distances[i][j] =
distance.json.rows[0].elements[0].distance.value;
            }
        }
    }

    // Tính toán quãng đường đi ngắn nhất sử dụng thuật toán Dijkstra
    const shortestDistance = dijkstra(distances, 0, distances.length - 1);

    return shortestDistance;
}
```

Trong ví dụ trên, ta sử dụng thư viện googlemaps để lấy tọa độ của các điểm trên bản đồ và tính toán khoảng cách giữa chúng. Sau đó, ta sử dụng hàm dijkstra để tính toán quãng đường đi ngắn nhất giữa các điểm. Hàm dijkstra có thể được triển khai như sau:

```
function dijkstra(graph: number[][], startNode: number, endNode: number):
number {
    const numNodes = graph.length;
    const visited = new Array(numNodes).fill(false);
    const distances = new Array(numNodes).fill(Infinity);
    distances[startNode] = 0;

    for (let i = 0; i < numNodes - 1; i++) {
        const current = minDistance(distances, visited);
        visited[current] = true;

        for (let j = 0; j < numNodes; j++) {
            if (!visited[j] && graph[current][j] !== 0 && distances[current]
!== Infinity) {
                const distance = distances[current] + graph[current][j];

                if (distance < distances[j]) {
                    distances[j] = distance;
                }
            }
        }
    }

    return distances[endNode];
}

function minDistance(distances: number[], visited: boolean[]): number {
    let min = Infinity;
    let minIndex = -1;

    for (let i = 0; i < distances.length; i++) {
        if (!visited[i] && distances[i] <= min) {
            min = distances[i];
            minIndex = i;
        }
    }

    return minIndex;
}
```

Trong hàm dijkstra, ta sử dụng một mảng distances để lưu trữ khoảng cách từ điểm đầu tiên đến các điểm khác. Ban đầu, ta gán giá trị Infinity cho tất cả các phần tử của mảng distances, trừ điểm đầu tiên có giá trị 0. Sau đó, ta sử dụng vòng lặp để duyệt qua tất cả các điểm và tính toán khoảng cách từ điểm đầu tiên đến các điểm khác. Trong vòng lặp, ta sử dụng hàm minDistance để tìm kiếm điểm có khoảng cách nhỏ nhất từ điểm đầu tiên, và sau đó cập nhật khoảng cách

từ các điểm khác đến điểm đó. Cuối cùng, ta trả về khoảng cách từ điểm đầu tiên đến điểm cuối cùng.

Với thuật toán Dijkstra và thư viện googlemaps, chúng ta có thể tính toán quãng đường đi ngắn nhất giữa các điểm trong Google Map.

5. Tôi có một bài toán: Làm thế nào để xuất một triệu bản ghi trong Mysql mà không tốn quá nhiều thời gian trong Nestjs!

Để xuất một triệu bản ghi trong MySQL mà không tốn quá nhiều thời gian, ta có thể sử dụng pagination và streaming data. Điều này có thể được triển khai trong NestJS bằng cách sử dụng nestjs-query và mysql2 package.

Tiếp theo, ta cần tạo một service để thực hiện truy vấn và xuất dữ liệu. Trong ví dụ này, ta sẽ tạo một service có tên là ExportService:

```
import { Injectable } from '@nestjs/common';
import { QueryService } from '@nestjs-query/core';
import { mysql2 } from '@nestjs-query/query-graphql';
import { Connection } from 'mysql2';
import { createWriteStream } from 'fs';
import { promisify } from 'util';

@Injectable()
export class ExportService {
    constructor(private readonly queryService: QueryService<Entity>, private
readonly connection: Connection) { }

    async exportData(): Promise<void> {
        const stream = createWriteStream('export.csv');

        const cursor = await
this.queryService.stream(mysql2(this.connection));

        cursor.pipe(stream);

        await promisify(stream.on.bind(stream))('finish');
    }
}
```

Trong service, ta sử dụng QueryService của nestjs-query để lấy dữ liệu từ cơ sở dữ liệu MySQL. Để xuất dữ liệu, ta sử dụng phương thức stream của QueryService để lấy dữ liệu dưới dạng stream và sử dụng createWriteStream để ghi dữ liệu vào tệp CSV. Cuối cùng, ta sử dụng promisify để đợi việc ghi dữ liệu hoàn thành.

Sau đó, ta có thể sử dụng ExportService để xuất dữ liệu. Ví dụ:

```
import { Controller, Get } from '@nestjs/common';
import { Connection } from 'mysql2';
import { ExportService } from './export.service';

@Controller()
export class ExportController {
    constructor(private readonly exportService: ExportService, private
readonly connection: Connection) { }

    @Get('/export')
    async export(): Promise<void> {
        await this.exportService.exportData();
    }
}
```

Ở đây, ta tạo một endpoint /export để xuất dữ liệu. Khi truy cập endpoint này, NestJS sẽ gọi phương thức export của ExportController, và từ đó sẽ gọi phương thức exportData của ExportService để thực hiện việc xuất dữ liệu.

Với việc sử dụng pagination và streaming data, chúng ta có thể xuất một triệu bản ghi trong MySQL mà không tốn quá nhiều thời gian trong NestJS.

6. Nestjs bảo mật hệ thống bằng cách nào?

Nestjs cung cấp nhiều tính năng bảo mật để giúp bảo vệ hệ thống như sau:

6.1. Xác thực:

Nestjs hỗ trợ xác thực thông qua JWT (JSON Web Token), Passport.js và một số middleware khác. Các công nghệ xác thực này giúp đảm bảo rằng người dùng chỉ có thể truy cập vào các tài nguyên được ủy quyền.

6.2. Phân quyền:

Nestjs cung cấp khả năng phân quyền dựa trên vai trò người dùng và quyền truy cập. Việc phân quyền giúp đảm bảo rằng người dùng chỉ có thể truy cập vào các tài nguyên mà họ được phép.

6.3. Bảo vệ chống lại tấn công CSRF:

Nestjs hỗ trợ chống lại tấn công CSRF (Cross-Site Request Forgery) bằng cách sử dụng middleware như csurf.

6.4. Bảo vệ chống lại tấn công XSS:

Nestjs có thể được cấu hình để tự động xóa các đoạn mã độc (malicious script) được chèn vào trong request bằng cách sử dụng thư viện sanitize-html.

6.5. Bảo vệ chống lại tấn công SQL injection:

Nestjs hỗ trợ sử dụng ORM (Object-Relational Mapping) và Query Builder để thực hiện các truy vấn CSDL an toàn hơn, giúp đảm bảo rằng không có mã độc được chèn vào các truy vấn SQL.

6.6. Bảo vệ chống lại tấn công brute-force:

Nestjs hỗ trợ sử dụng các middleware và thư viện như express-rate-limit để giới hạn số lần yêu cầu đến API trong một khoảng thời gian nhất định. Điều này giúp giảm thiểu khả năng bị tấn công bằng cách sử dụng brute-force để đoán mật khẩu.

6.7. HTTPS:

Nestjs hỗ trợ sử dụng HTTPS để mã hóa dữ liệu được truyền giữa máy khách và máy chủ. Việc sử dụng HTTPS giúp đảm bảo rằng dữ liệu không bị lộ thông qua các gói tin truyền qua mạng.

7. GIT:

Dưới đây là một số câu lệnh Git thông dụng thường được sử dụng:

git init : Khởi tạo một repository mới.

git clone : Sao chép một repository đã tồn tại từ một nguồn khác.

git add : Thêm các file đã chỉnh sửa hoặc mới tạo vào vùng staging.

git commit : Lưu các thay đổi đã được thêm vào vùng staging và tạo một phiên bản mới.

git status : Hiển thị trạng thái hiện tại của repository.

git branch : Tạo, xóa hoặc liệt kê các nhánh hiện có.

git checkout : Chuyển sang một nhánh khác hoặc khôi phục các thay đổi từ một commit cụ thể.

git merge : Hợp nhất các nhánh khác vào nhánh hiện tại.

git pull : Lấy các thay đổi mới nhất từ một repository từ xa và hợp nhất chúng vào nhánh hiện tại.

git push : Đẩy các thay đổi của bạn lên repository từ xa.

git log : Liệt kê các commit đã được tạo ra trên repository.

git remote : Liên kết repository với một repository từ xa.

git fetch : Lấy các thay đổi mới nhất từ một repository từ xa, nhưng không hợp nhất chúng vào nhánh hiện tại.

git diff : Hiển thị sự khác biệt giữa các commit hoặc giữa các phiên bản của các file.

git config : Cấu hình Git cho một repository hoặc toàn bộ hệ thống.

git tag : Gắn một nhãn trên một commit cụ thể.

git stash : Lưu các thay đổi tạm thời để có thể chuyển sang một nhánh khác mà không cần commit.

git revert : Tạo một commit mới để hoàn tác một commit trước đó.

git reset : Trở lại một commit hoặc một trạng thái trước đó của repository.

git rebase : Thay đổi lịch sử commit bằng cách áp dụng các thay đổi của một nhánh khác vào một nhánh khác.

git cherry-pick : Chọn các commit từ một nhánh khác và áp dụng chúng vào nhánh hiện tại.

git submodule : Thêm một repository khác vào như một thư mục con trong repository hiện tại.

git bisect : Tìm kiếm một commit gây ra lỗi bằng cách sử dụng phương pháp tìm kiếm nhị phân.

git blame : Hiển thị lịch sử chỉnh sửa của một file, bao gồm tên người chỉnh sửa, thời gian và commit ID.

8. MySQL:

Dưới đây là một số lệnh thường được sử dụng trong MySQL:

SHOW DATABASES; : Hiển thị danh sách các cơ sở dữ liệu.

USE <database_name>;: Chọn cơ sở dữ liệu cần sử dụng.

SHOW TABLES;: Hiển thị danh sách các bảng trong cơ sở dữ liệu đang được sử dụng.

DESCRIBE <table_name>; : Hiển thị thông tin về các cột trong bảng.

SELECT <column_name(s)> FROM <table_name> WHERE <condition>; : Lấy dữ liệu từ bảng với điều kiện cụ thể.

INSERT INTO <table_name> (<column_name(s)>) VALUES (<value(s)>); : Chèn một bản ghi mới vào bảng.

UPDATE <table_name> SET <column_name> = <value> WHERE <condition>; : Cập nhật dữ liệu trong bảng.

DELETE FROM <table_name> WHERE <condition>; : Xóa dữ liệu khỏi bảng.

CREATE TABLE <table_name> (<column_name> <data_type> <constraint(s)>); : Tạo một bảng mới trong cơ sở dữ liệu.

ALTER TABLE <table_name> ADD <column_name> <data_type> <constraint(s)>; : Thêm một cột mới vào bảng.

DROP TABLE <table_name>; : Xóa một bảng từ cơ sở dữ liệu.

CREATE DATABASE <database_name>; : Tạo một cơ sở dữ liệu mới.

ALTER DATABASE <database_name> CHARACTER SET <character_set>; : Thay đổi bộ ký tự của cơ sở dữ liệu.

DROP DATABASE <database_name>; : Xóa một cơ sở dữ liệu.

ALTER TABLE <table_name> MODIFY <column_name> <new_data_type> <constraint(s)>; : Thay đổi kiểu dữ liệu của một cột trong bảng.

TRUNCATE TABLE <table_name>; : Xóa toàn bộ dữ liệu trong bảng nhưng không xóa bảng.

SHOW COLUMNS FROM <table_name>; : Hiển thị thông tin chi tiết về các cột trong bảng.

SHOW INDEX FROM <table_name>; : Hiển thị danh sách các chỉ mục trong bảng.

CREATE INDEX <index_name> ON <table_name> (<column_name>); : Tạo một chỉ mục mới cho một cột trong bảng.

CREATE USER '<username>'@'<hostname>' IDENTIFIED BY '<password>'; : Tạo một người dùng mới và gán mật khẩu cho người dùng đó.

GRANT <privileges> ON <database_name>.<table_name> TO '<username>'@'<hostname>'; : Cấp quyền cho người dùng trên cơ sở dữ liệu hoặc bảng cụ thể.

REVOKE <privileges> ON <database_name>.<table_name> FROM '<username>'@'<hostname>'; : Thu hồi quyền của người dùng đối với cơ sở dữ liệu hoặc bảng cụ thể.

SHOW GRANTS FOR '<username>'@'<hostname>';: Hiển thị danh sách các quyền được cấp cho người dùng đó trên toàn hệ thống MySQL.

SET PASSWORD FOR '<username>'@'<hostname>' = '<password>'; : Thay đổi mật khẩu của người dùng.

- ***INNER JOIN***: Trả về các hàng từ bảng 1 và bảng 2 khi có sự khớp giữa các giá trị khóa của hai bảng. Nó chỉ trả về các hàng có giá trị khóa khớp trong cả hai bảng. Cú pháp của lệnh INNER JOIN như sau:

  SELECT *
  FROM table1
  INNER JOIN table2
  ON table1.column = table2.column;
- ***LEFT JOIN:*** Trả về tất cả các hàng từ bảng trái (table1) và các hàng khớp từ bảng phải (table2). Nếu không có hàng nào khớp trong bảng phải, thì giá trị các cột của bảng phải được điền bằng NULL. Cú pháp của lệnh LEFT JOIN như sau:
  SELECT *
  FROM table1
  LEFT JOIN table2
  ON table1.column = table2.column;
- ***RIGHT JOIN:*** Tương tự như LEFT JOIN, nhưng trả về tất cả các hàng từ bảng phải (table2) và các hàng khớp từ bảng trái (table1). Nếu không có hàng nào khớp trong bảng trái, giá trị các cột của bảng trái được điền bằng NULL.
  SELECT *
  FROM table1
  RIGHT JOIN table2
  ON table1.column = table2.column;
- ***FULL OUTER JOIN*:** Trả về tất cả các hàng từ cả hai bảng, bao gồm cả các hàng không khớp. Nếu không có hàng khớp, các giá trị của cột trong bảng thiếu sẽ được điền bằng NULL.
  SELECT *
  FROM table1
  FULL OUTER JOIN table2
  ON table1.column = table2.column;

- ***CROSS JOIN*:** Trả về tất cả các kết hợp của các hàng từ bảng 1 và bảng 2.
  SELECT *
  FROM table1
  CROSS JOIN table2;
- **SELF JOIN:** Trả về các hàng từ bảng được ghép nối với chính nó, cho phép bạn truy vấn các cột trong cùng một bảng với nhau.
  SELECT *
  FROM table1 t1
  JOIN table1 t2
  ON t1.column = t2.column;

9. TypeORM:

TypeORM là một ORM (Object-Relational Mapping) cho Node.js và TypeScript, cho phép bạn tương tác với các cơ sở dữ liệu quan hệ bằng cách sử dụng các đối tượng JavaScript hoặc TypeScript thay vì viết các truy vấn SQL trực tiếp. TypeORM hỗ trợ các cơ sở dữ liệu phổ biến như PostgreSQL, MySQL, MariaDB, SQLite, MS SQL Server, Oracle, MongoDB và CockroachDB.

Một số tính năng chính của TypeORM bao gồm:

- Tạo và quản lý cơ sở dữ liệu.
- Tạo và quản lý bảng và các chỉ mục.
- Tạo và quản lý các quan hệ giữa các bảng.
- Thực hiện các thao tác CRUD (Create, Read, Update, Delete) với cơ sở dữ liệu.
- Sử dụng migrations để đồng bộ hóa cơ sở dữ liệu giữa các phiên bản ứng dụng khác nhau.
- Hỗ trợ caching và transaction management.

Để sử dụng TypeORM, chúng ta có thể định nghĩa các entity (đối tượng) để tương ứng với các bảng trong cơ sở dữ liệu, sử dụng decorators để chỉ định các thuộc tính và quan hệ của chúng với các entity khác, và sử dụng các phương thức của TypeORM để thực hiện các truy vấn và thao tác với cơ sở dữ liệu.

10. Bảng so sánh giữa microservice, Redis và Network:

	Microservice	**Redis**	**Network**
Định nghĩa	Microservice là một kiến trúc phần mềm phân tán, trong đó các tính năng của	Redis là một hệ thống lưu trữ dữ liệu trong bộ nhớ được sử dụng để	Network là một hệ thống liên kết các thiết bị điện tử như máy tính, điện thoại,

	ứng dụng được phân chia thành các dịch vụ nhỏ độc lập với nhau.	lưu trữ, truy vấn và phân tích các dữ liệu có cấu trúc.	thiết bị IoT với nhau để trao đổi dữ liệu và thông tin.
Mục đích	Tách biệt các tính năng của ứng dụng thành các dịch vụ nhỏ hơn, dễ dàng quản lý và phát triển.	Lưu trữ và truy vấn dữ liệu trong bộ nhớ nhanh chóng và hiệu quả hơn so với hệ thống lưu trữ truyền thống.	Cho phép các thiết bị điện tử trao đổi dữ liệu và thông tin với nhau.
Các đặc tính	Độc lập với nhau, linh hoạt, phát triển dễ dàng, có thể scale lên và xuống dễ dàng.	Nhanh chóng và hiệu quả trong việc lưu trữ và truy vấn dữ liệu.	Đáp ứng các yêu cầu trao đổi dữ liệu và thông tin của các thiết bị điện tử.
Các đặc tính	Độc lập với nhau, linh hoạt, phát triển dễ dàng, có thể scale lên và xuống dễ dàng.	Nhanh chóng và hiệu quả trong việc lưu trữ và truy vấn dữ liệu.	Đáp ứng các yêu cầu trao đổi dữ liệu và thông tin của các thiết bị điện tử.
Ứng dụng	Sử dụng trong các hệ thống phức tạp, đòi hỏi độ tin cậy cao và dễ dàng quản lý, phát triển.	Sử dụng trong các ứng dụng cần lưu trữ và truy vấn dữ liệu với tốc độ cao.	Sử dụng trong các hệ thống mạng, bao gồm cả mạng cục bộ và mạng toàn cầu, để cho phép các thiết bị trao đổi dữ liệu với nhau.
Công nghệ	Sử dụng các công nghệ như Docker, Kubernetes, và các ngôn ngữ lập trình như Java, Python, Go.	Sử dụng các công nghệ như C/C++, Python, và Ruby.	Sử dụng các công nghệ như Ethernet, TCP/IP, DNS, và HTTP.
Ưu điểm	Độc lập, linh hoạt, dễ quản lý và phát triển, có khả năng	Nhanh chóng và hiệu quả trong việc lưu trữ và truy vấn dữ liệu, hỗ trợ các	Cho phép các thiết bị điện tử trao đổi dữ liệu và thông tin với nhau, tạo ra một môi

	scale lên và xuống dễ dàng.	tính năng như cache, publish/subscribe, và có thể tích hợp với nhiều ngôn ngữ lập trình khác nhau.	trường mạng linh hoạt và phát triển được.
Nhược điểm	Đòi hỏi các kỹ năng kiến trúc phần mềm và phát triển phần mềm chuyên nghiệp.	Không thể lưu trữ các dữ liệu lớn hoặc không có cấu trúc.	Đôi khi có thể gặp vấn đề về hiệu suất hoặc bảo mật.
Ví dụ	Netflix, Uber, Amazon, và Google đều sử dụng kiến trúc microservice.	Redis được sử dụng trong các ứng dụng cần lưu trữ và truy vấn dữ liệu nhanh, chẳng hạn như các trang web thương mại điện tử hoặc ứng dụng trò chơi trực tuyến.	Mạng cục bộ trong một văn phòng hoặc mạng toàn cầu như Internet.