1. **Explain the mechanism of handling routes in Express.js with an example program.**

Routing
Routing refers to how an server side application responds to a client request to a particular endpoint. This endpoint consists of a URI (a path such as / or /books) and an HTTP method such as GET, POST, PUT, DELETE, etc.

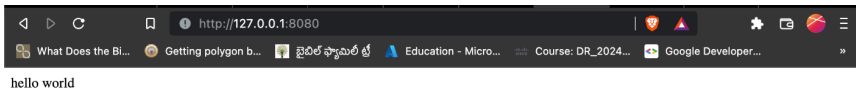
Routing Paths
A routing path is a combination of a request method to define the endpoints at which requests can be made by a client. Route paths can be strings, string patterns, or regular expressions.

You define routing using methods of the Express app object that correspond to HTTP methods; for example, app.get() to handle GET requests and app. post to handle POST requests. For a full list, see app.METHOD. You can also use app.all() to handle all HTTP methods and app.use() to specify middleware as the callback function

Example program

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

app.all((req, res, next) => {
   console.log('Accessing the secret section ...')
   next() // pass control to the next handler
})
// takes all requests
app.get('/', (req, res) => {
   res.send('hello world')
})
// respond with "HOME PAGE" when a GET request is made to the homepage
app.get('/home', (req, res) => {
   res.send('home page')
})
// respond with error page if no route found
app.get('/*', (req, res) => {
   res.send('error 404 page not found')
})
// POST method route
app.post('/', (req, res) => {
   res.send('POST request to the homepage')
})
// start the express server listening on port 8080
app.listen(8080, () => {
   console.log('server started on port 8080')
})
```

```
samireddynani@Nanis-Macbook 1a % npm run start

> 1a@1.0.0 start
> nodemon main

[nodemon] 2.0.20
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node main index.js`
```



2. **Discuss the Express.js process of connecting to MongoDB with Mongoose and perform CRUD operations.**

Refer to this:-

https://javascript.plainenglish.io/connect-mongodb-to-node-using-express-and-mongoose-c405d1158c

MongoDB

MongoDB is a source-available cross-platform document-oriented database program. Classified as a NoSQL database program.

Express

Express.js, or simply Express, is a back end web application framework for Node.js, released as free and open-source software under the MIT License. It is designed for building web applications and APIs.

Mongoose

Mongoose.js connects your MongoDB clusters or collections with your Node.js app. It enables you to create schemas for your documents. Mongoose provides a lot of functionality when creating and working with schemas.

1. Installing Express and Mongoose

Install Express and Mongoose using the command

```
npm install express mongoose --save
```

server.js

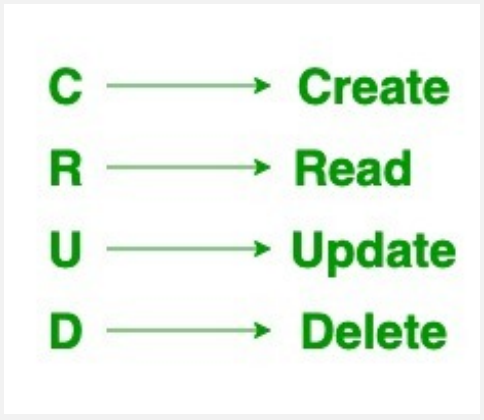
```
const express = require("express");
const mongoose = require("mongoose");
const userModel = require("./models");

app.use(express.json());

mongoose.connect('mongodb://localhost:27017/usersdb', { useNewUrlParser: true });

const db = mongoose.connection;
db.on("error", console.log( "connection error: "));
db.once("open", () => {
   console.log("Connected successfully");
});

app.get("/users", async (req, res) => {
    await userModel.find({}).then(
       (users) => {
       res.send(user);
   }).catch((error) => {
       console.log(error);
   });
});
app.get("/addUser", async (req, res) => {
   const user = new userModel({
       name: "John",
       lucky_number: 7,
   });
   await user.save().then((user) => {
       res.send(user);
   }).catch((error) => {
       console.log(error);
   });
});

app.listen(3000, () => {
   console.log("Server is running at port 3000");
});
```

```
C:\Users\mOdEL\Desktop\men_tutorial>node server.js
Server is running at port 3000
Connected successfully
```

Models.js

```
const mongoose = require("mongoose");
const UserSchema = new mongoose.Schema({
   name: {
       type: String,
       required: true,
   },
   lucky_number: {
       type: Number,
       default: 0,
   },
});
const User = mongoose.model("User", UserSchema);
```

CRUD operations



Create Operations –

The create or insert operations are used to insert or add new documents in the collection. If a collection does not exist, then it will create a new collection in the database. You can perform, create operations using the following methods provided by the MongoDB:

Method	Description
db.collection.insertOne()	It is used to insert a single document in the collection.
db.collection.insertMany()	It is used to insert multiple documents in the collection.
db.createCollection()	It is used to create an empty collection.

Read Operations –

The Read operations are used to retrieve documents from the collection, or in other words, read operations are used to query a collection for a document. You can perform read operation using the following method provided by the MongoDB:

Method	Description
db.collection.find()	It is used to retrieve documents from the collection.

Update Operations –

The update operations are used to update or modify the existing document in the collection. You can perform update operations using the following methods provided by the MongoDB:

Method	Description
db.collection.updateOne()	It is used to update a single document in the collection that satisfy the given criteria.
db.collection.updateMany()	It is used to update multiple documents in the collection that satisfy the given criteria.
db.collection.replaceOne()	It is used to replace single document in the collection that satisfy the given criteria.

Delete Operations –

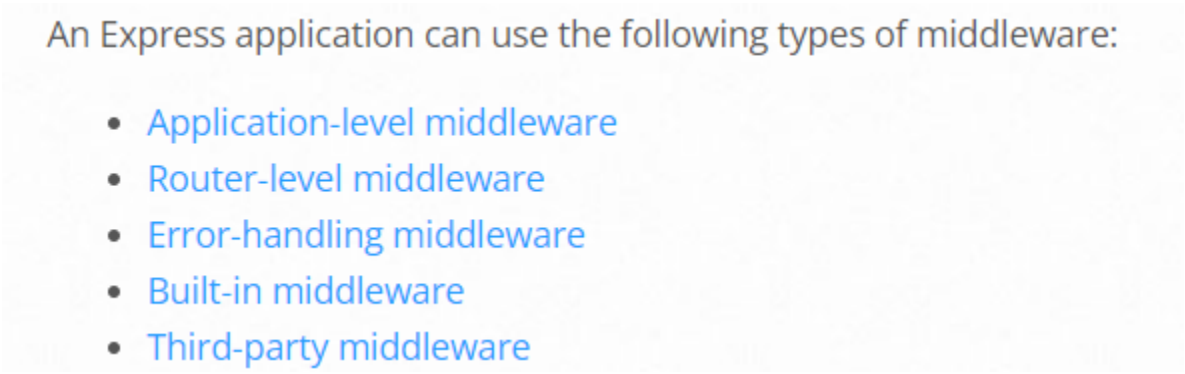
The delete operation are used to delete or remove the documents from a collection. You can perform delete operations using the following methods provided by the MongoDB:

Method	Description
db.collection.deleteOne()	It is used to delete a single document from the collection that satisfy the given criteria.
db.collection.deleteMany()	It is used to delete multiple documents from the collection that satisfy the given criteria.

3. Illustrate different types of Middleware in Express.js with examples.

What is Express Middleware?

Middleware literally means anything you put in the middle of one layer of the software and another. Express middleware are functions that execute during the lifecycle of a request to the Express server.



Application-level middleware

Bind application-level middleware to an instance of the app object by using the `app.use()` and `app.METHOD()` functions, where `METHOD` is the HTTP method of the request that the middleware function handles (such as GET, PUT, or POST) in lowercase.

This example shows a middleware function with no mount path. The function is executed every time the app receives a request.

Router-level middleware

Router-level middleware works in the same way as application-level middleware, except it is bound to an instance of `express.Router()`.

```
const router = express.Router()
```

Load router-level middleware by using the `router.use()` and `router.METHOD()` functions.

Error-handling middleware

Define error-handling middleware functions in the same way as other middleware functions, except with four arguments instead of three, specifically with the signature `(err, req, res, next)`):

Built-in middleware

Express has the following built-in middleware functions:

- express.static serves static assets such as HTML files, images, and so on.
- express.json parses incoming requests with JSON payloads. **NOTE: Available with Express 4.16.0+**
- express.urlencoded parses incoming requests with URL-encoded payloads. **NOTE: Available with Express 4.16.0+**

Third-party middleware

Use third-party middleware to add functionality to Express apps.

Install the Node.js module for the required functionality, then load it in your app at the application level or at the router level.

The following example illustrates installing and loading the cookie-parsing middleware function `cookie-parser`.

```
$ npm install cookie-parser
```

```
const express = require('express')
const app = express()
const router = express.Router()
// Application-level middleware
app.use((req, res, next) => {
  console.log('Time:', Date.now())
  next()
})
// Router-level middleware
router.use((req, res, next) => {
    console.log('Time:', Date.now())
    next()
  })
  app.use('/', router)

//   error handler
  app.use((err, req, res, next) => {
    console.error(err.stack)
    res.status(500).send('Something broke!')
  })
//   Third-party middleware
const cookieParser = require('cookie-parser')
// load the cookie-parsing middleware
app.use(cookieParser())
```

4. Explain session management using Cookies in Express.js with an example program.

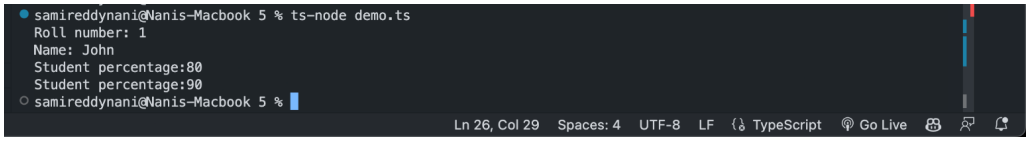
Session cookies: Session cookies are the temporary cookies that mainly generated on the server-side.The main use of these cookies to track all the request information that has been made by the client overall particular session. The session is stored for a temporary time when the user closes the browser session automatically destroys it. In this article, we will be using external file storage in order to store session cookies.

```
var express = require('express');
var cookieParser = require('cookie-parser');
var app = express();
app.use(cookieParser());
app.get('/cookieset',function(req, res){
res.cookie('cookie_name', 'cookie_value');
res.cookie('College', 'Aditya');
res.cookie('Branch', 'Cse');

res.status(200).send('Cookie is set');
});
app.get('/cookieget', function(req, res) {
 res.status(200).send(req.cookies);
});
app.get('/', function (req, res) {
 res.status(200).send('Welcome to Aditya');
});
var server = app.listen(8000, function () {
 var host = server.address().address;
 var port = server.address().port;
 console.log('Server listening at http://%s:%s', host, port);
});
```

```
D:\mstd>node cooki.js
Server listening at http://:::8000
```

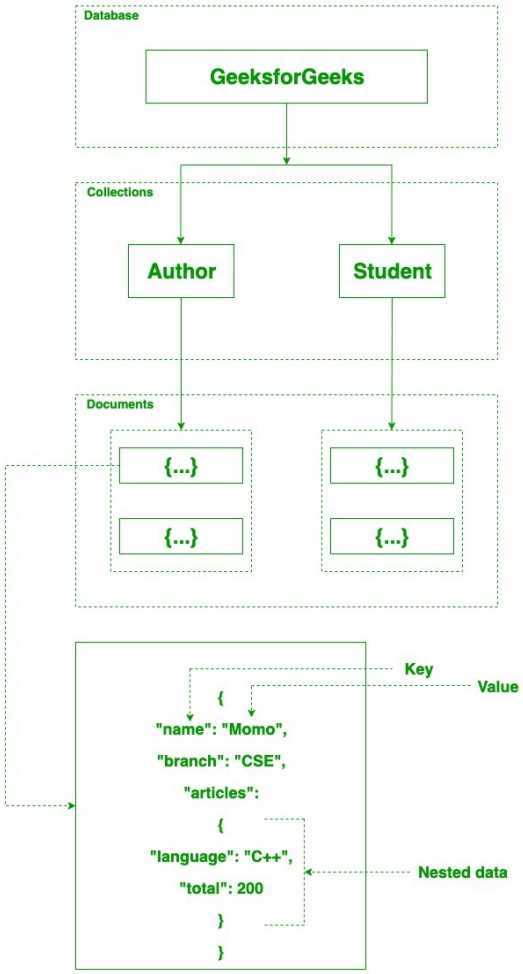
5. Make use of class, constructor, access modifiers, properties and methods in Typescript with an example program.

```
class Student{
    rollNo: number;
    public name: string;
    private percentage: number;
    constructor(rollNo: number, name: string, percentage: number){
        this.rollNo = rollNo;
        this.name = name;
        this.percentage = percentage;
    }
    display(){
        console.log("Roll number: "+this.rollNo + "\nName: " + this.name);
    }
    displayPercentage(){
        console.log("Student percentage:"+this.percentage);
    }
    updatePercentage(percentage: number){
        this.percentage = percentage;
    }

}

let student = new Student(1, "John", 80);
student.display();
student.displayPercentage();
student.updatePercentage(90);
student.displayPercentage();
```



6. Explain MongoDB structure and architecture with neat diagrams.

Structure:



Database

It is also called the physical container for data. Every database has its own set of files existing on the file system. In a single MongoDB server, there are multiple databases present. A database contains multiple collections

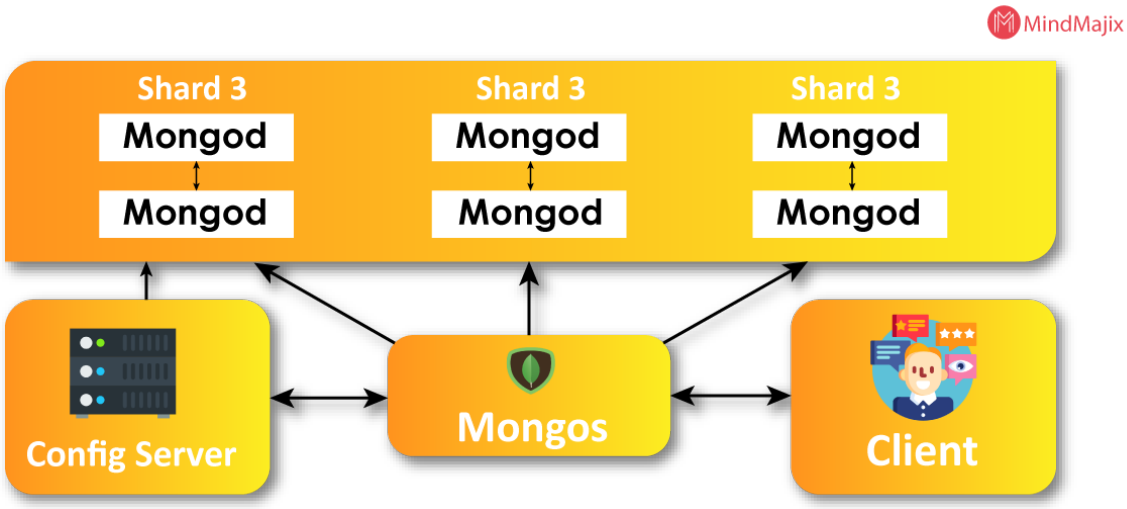
Collection

The collection consists of various documents from different fields. All the collections reside within one database. In collections no schemas are present.

Document

The set of key values is assigned to the document which is in turn associated with dynamic schemas. The benefit of using these schemas is that a document may not have to possess the same fields whereas they can have different data types.

Architecture



MongoDB Sharding is a technique to distribute the data across multinode. Using Sharding MongoDB supports large datasets and delivers high throughput operation. It uses horizontal scaling to distribute the data on multiple systems and depending upon the requirement add the nodes as well.

7. Summarize how to create and use modules in Typescript with an example program.

→ A module is a way to create a group of related variables, functions, classes, and interfaces, etc. It executes in the local scope, not in the global scope. In other words, the variables, functions, classes, and interfaces declared in a module cannot be accessible outside the module directly. We can create a module by using the export keyword and can use in other modules by using the import keyword

Let’s create an external module with some functions

→ create a new .ts file named demoModule.ts and write the following code in that file

```
export class DemoModule {
 constructor() {
   console.log('DemoModule constructor');
 }
 areaOfCircle(radius: number): number {
   return Math.PI * radius * radius;
 }
   areaOfRectangle(length: number, breadth: number): number {
   return length * breadth;
   }
}
```

→ Now create another file **main.ts** in the same directory and place the below code

```
import * as DemoModuleAll from "./DemoModule";
let demoModule = new DemoModuleFileAll.DemoModule();
console.log(demoModule.areaOfCircle(10));
console.log(demoModule.areaOfRectangle(10, 20));
```

→ run

```
samireddynani@Nanis-Macbook 6 % ts-node main.ts
DemoModule constructor
314.1592653589793
200
```

8. Discuss CRUD operations on documents using MongoDB.

What is CRUD in MongoDB?

For full content refer:- https://www.mongodb.com/basics/crud

9. Discuss components and modules in Angular.js with an example program.

Creating a component

Components are the most basic UI building block of an Angular app. An Angular app contains a tree of Angular components.

The best way to create a component is with the Angular CLI. You can also create a component manually.

Creating a component using the Angular CLI

To create a component using the Angular CLI:

1. From a terminal window, navigate to the directory containing your application.
2. Run the **`ng generate component <component-name>`** command, where `<component-name>` is the name of your new component.

By default, this command creates the following:

- A directory named after the component
- A component file, `<component-name>.component.ts`
- A template file, `<component-name>.component.html`
- A CSS file, `<component-name>.component.css`
- A testing specification file, `<component-name>.component.spec.ts`

Where `<component-name>` is the name of your component.

ng generate component component1

Output in termainalt::

CREATE src/app/component1/component1.component.css (0 bytes)

CREATE src/app/component1/component1.component.html (25 bytes)

CREATE src/app/component1/component1.component.spec.ts (627 bytes)

CREATE src/app/component1/component1.component.ts (218 bytes)

UPDATE src/app/app.module.ts (412 bytes)

In the Component.ts file the default code is

```
import { Component } from '@angular/core';

@Component({

 selector: 'app-component1',

 templateUrl: './component1.component.html',

 styleUrls: ['./component1.component.css']

})

export class Component1Component {};

}
```

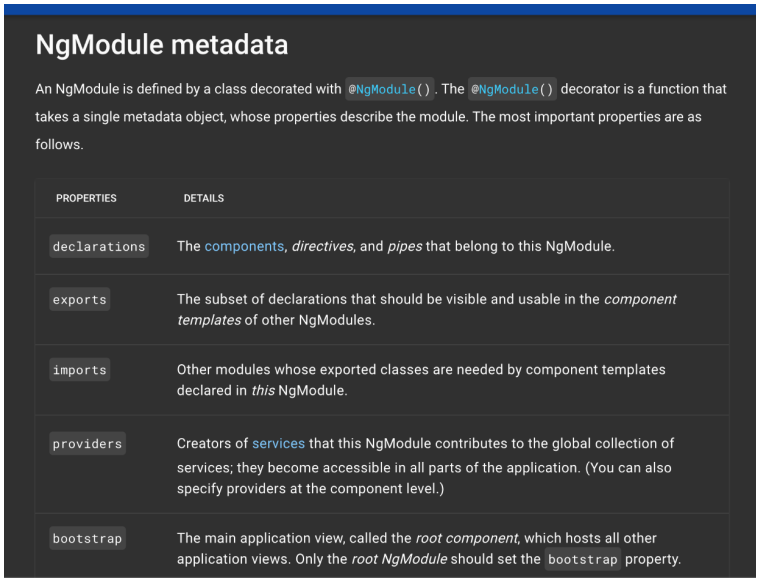
Angular module items:-

Angular applications are modular and Angular has its own modularity system called *NgModules*. NgModules are containers for a cohesive block of code dedicated to an application domain. They can contain components, service providers, and other code files whose scope is defined by the containing NgModule. They can import functionality that is exported from other NgModules, and export selected functionality for use by other NgModules.

Every Angular application has at least one NgModule class, the *root module*, which is conventionally named `AppModule` and resides in a file named `app.module.ts` You launch your application by *bootstrapping* the root NgModule.

app.module.t

```
import { NgModule } from '@angular/core';

import { BrowserModule } from '@angular/platform-browser';

@NgModule({

 imports:      [ BrowserModule ],

 providers:    [ Logger ],

 declarations: [ AppComponent ],

 exports:      [ AppComponent ],

 bootstrap:    [ AppComponent ]

})

export class AppModule { }
```



NgModule metadata

An NgModule is defined by a class decorated with `@NgModule()`. The `@NgModule()` decorator is a function that takes a single metadata object, whose properties describe the module. The most important properties are as follows.

PROPERTIES	DETAILS
`declarations`	The components, *directives*, and *pipes* that belong to this NgModule.
`exports`	The subset of declarations that should be visible and usable in the *component templates* of other NgModules.
`imports`	Other modules whose exported classes are needed by component templates declared in *this* NgModule.
`providers`	Creators of services that this NgModule contributes to the global collection of services; they become accessible in all parts of the application. (You can also specify providers at the component level.)
`bootstrap`	The main application view, called the *root component*, which hosts all other application views. Only the *root NgModule* should set the `bootstrap` property.

10. Illustrate property and attribute binding in Angular.js with an example program.

Property Binding: Similar to Java, variables defined in the parent class can be inherited by the child class which is templates in this case. The only difference between Interpolation and Property binding is that we should not store non-string values in variables while using interpolation. So if we have to store Boolean or other data types then use Property Binding. In simple words, we bind a property of a DOM element to a field which is a defined property in our component TypeScript code.

Attribute binding Angular helps you set values for attributes directly. With attribute binding, you can improve accessibility, style your application dynamically, and manage multiple CSS classes or styles simultaneously

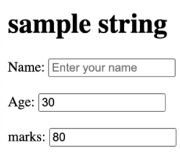
App.component.html

```
<h1>sample string</h1>
<label>Name: </label><input type="text" [attr.placeholder]="namePlaceHolder" />
<br /><br />
<label>Age: </label><input type="text" value="{{ defaultAge }}" /><br /><br />
<label>marks: </label><input type="text" [value]="defaultMarks" />
```

App.component.ts

```
import { Component } from '@angular/core';


@Component({
 selector: 'app-root',
 templateUrl: './app.component.html',
 styleUrls: ['./app.component.css']
})
export class AppComponent {
 title = 'demo';
 defaultName = 'userName';
 defaultAge = 30;
 namePlaceHolder = 'Enter your name';
 defaultMarks = 80;
}
```

Output:



11. Explain structural directives in Angular.js with an example program.

https://www.geeksforgeeks.org/structural-directives-in-angular/

12. Apply Dependency Injection in Angular.js with an example program