MEAN

* What is MEAN stack

ans → The MEAN stack is a javascript-based technology stack used for building dynamic web applications. It consisting of four main components.

(i) MongoDB → A database
(ii) Express.js → A framework for server
(iii) Angular → A frontend framework for web pages.
(iv) Node.js → A Javascript Environment to run Javascript.

Page no.

* Benefits of full stack development

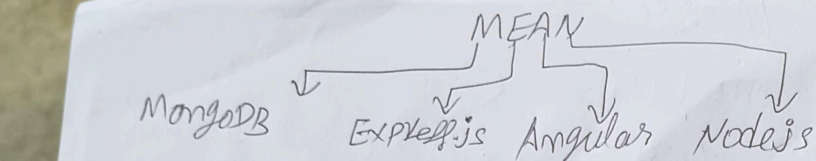
(i) versatility and Efficiency: A full stack developer can handle both frontend and backend tasks, making development more efficient.

(ii) Faster Development and Deployment: Since a single developer or a small team can handle both client-side and server side, projects can be completed faster.

(3) Cost Effective: Hiring a full-stack Developers is more economical than hiring seperate frontend and backend developers.

(4) Better problem-Sloving Skills: Full stack developers have a broader understanding of the entire system, allowing them to troubleshoot and fix issues efficiently.

(5) End-to-End Responsibility: A full stack developer can take complete ownership of a project, from planning and design to deployment and maintenance.



* MongoDB and its Advantages

MongoDB is a NoSQL database that stores data in a flexible, document-oriented format (JSON like BSON documents). Unlike traditional relational database (RDBMS), MongoDB is schema-less, which allows for dynamic and scalable data storage. It is widely used in modern web application, big data processing, and cloud computing.

Advantages of MongoDB

1. Schema flexibility (NoSQL): Unlike SQL database, MongoDB does not require a fixed schema. In this feilds can be added or removed dynamically. suitable for applications with evolving data structures.

(2) High performance: Faster read and write operations due to its document-based structure. No complex joins which improves speed.

(3) Scalability: supports horizontal scalling to handle large data volumes.

(4) Easy to used and JSON like documents: stores data in a BSON (binary json) format, making it easy to integrate with web applications.

(5) Open-Source and cost-Effective: Free to use under the MongoDB Community Edition.

(6) powerful Quesying capabilities: supports rich query operators, including filtering, sorting, and aggregation.

* Express and its advantages

Ans ⇒ Express.js is a fast unopinionated, and minimalist web application framework for Node.js it simplifies the process of building server-side application and APIs by providing a robust set of features.

Key features

(i) Routing: Defines multiple routes for handling HTTP requests (GET, POST, PUT, DELETE).

(ii) middleware: Uses functions to handle requests and responses efficiently.

(iii) Template Engines: Supports rendering dynamic HTML using engines like EJS, Pug and Handlebars.

(iv) Database Integration: Easily integrates with databases like MongoDB.

(v) Error Handling: Centeralized error handling for better debugging.

Advantages of Express.js

(i) Fast and Lightweight: Express.js is minimalistic and runs efficiently on Node.js, making it faster than traditional frameworks like Django or Rails.

(ii) Easy to Learn and use: Simple syntax and minimal boilerplate code make it beginner-friendly.

(iii) Middleware Support: Express has built-in middleware for parsing requests (express.json(), express.urlencoded()) and supports third party middleware

(iv) Large community and Ecosystem: A vast number of open-source modules and middleware are available to extend functionalities.

(v) Full Stack JavaScript: Since Express run on Node.js, it allows development using JavaScript for both frontend & backend.

* Angular and its advantages.

ans → Angular is a TYPEscript-based open-source front-end web application framework developed and maintained by Google. It is used for building single page application and dynamic web application.

Advantages of Angular

(i) Component-Based Architecture: The application is divided into reusable components ~~is divided into~~, making development and maintenance easier.

(ii) Two-way Data Binding: Automatically synchronizes data b/w the model and view.

(iii) Routing: Built-in router module for navigation in SPAs.

(iv) Cross platform Development: Can be used to build web, mobile and desktop applications.

(v) API integration: Allows interation with APIs and backend services.

* Node.js and its advantage

Node.js is an open-source, cross-platform javascript runtime environment that allows developers to run javascript code outside a web browser. It is built on google chrome's V8 engine and it is used to develop scalable server-side and network application.

Advantages of node.js [what are the main use of Node JS in mean stack]

(1) Asynchronous & non-Blocking: Handles multiple requests without waiting for the previous one to complete

(2) Fast Execution: Runs javascript using the V8 engine, making it very fast. It allows JS to run on the server.

(3) Cross-platform: works on windows, Linux and macOS. Provides npm to install dependencies.

(4) Built-in-package Manager (NPM) ~~use an event-driven approach for real time application~~.

MST. Diff b/w centerilised and distributed version control syst
diff bit and GitHub

⑤ microservices and API development: Ideal for RESTful RESTful APIs and GraphQL APIs.

⑥ Scalability: Ideal for building highly scalable application

⑦ Large community support: It is supported by companies like google, Microsoft, paypal and netflix with a large developer community.

⑧ Easy to learn: Javascript is widely used, making Node.js easy to learn for frontend also for backend.

Q2. Illustrate how the MEAN stack components work together to create an application.

ans ⇒ Illustrating how MEAN stack Components work together to create an application.

The MEAN stack is a Javascript-based full stack development framework that includes MongoDB, Express.js, Angular and Node.js. It allows developers to build dynamic and scalable web applications using javascript throughout the stack.

MEAN Stack Components Overview

Components	Description
MongoDB	A NOSQL database that stores data in JSON-like format (BSON)
Express.js	A lightweight Node.js framework for handling HTTP requests, routing and middleware.
Angular	A frontend framework for building single-page applications.
Node.js	A runtime environment that executes Javascript on the server side.

How MEAN Stack Works Together.

Each component in MEAN Stack has a specific role in handling client requests, processing logic, and managing data.

Step-by-Step Workflow

Step1: User interacts with the Angular frontend: The user opens the Angular application in a browser. Angular dynamically loads components, templates and styles. User actions (eg. submitting a form).

Step2: Angular sends requests to express server via http requests: Angular uses HttpClient to send GET, POST, PUT, DELETE requests to Express.js.

Step3: Express.js Handles the HTTP Requests: Express.js receives the HTTP request with MongoDB for data retrival or manipulation. Routes and middleware handle authentication, validation, and error handling.

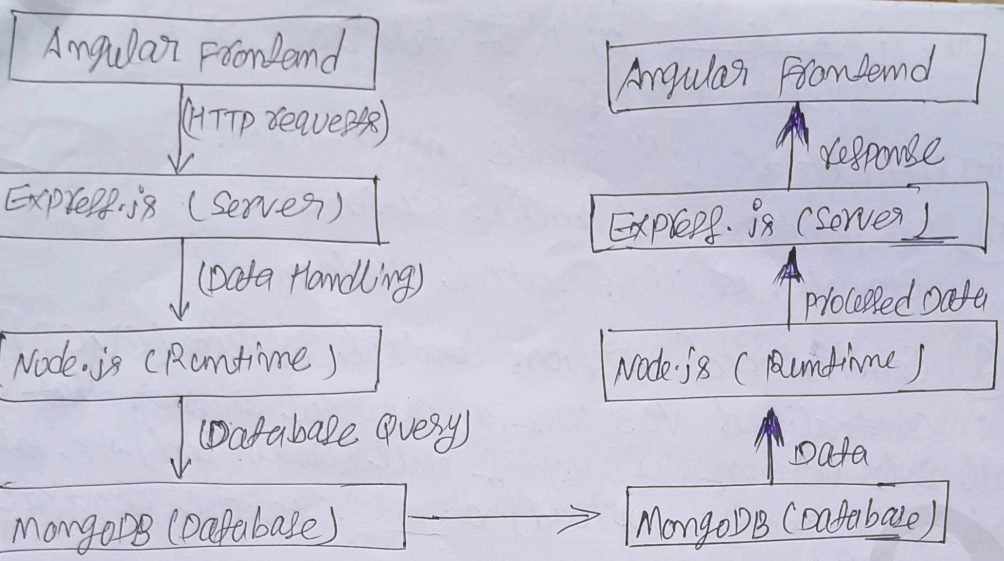
Step 4: Node.js Processes the Reavest: Node.js acts of a server-side runtime to execute Javascript Code.

Step5: Express.js interacts with MongoDB: Express.js uses Mongoose to interact with MongoDB. it Performs CRUD operations ① Create ② Read ③ Update ④ Delete

Step6: MongoDB Store and Retrives Data: MongoDB stores data in collections of JSON-like documents.

Step7: Express.js sends a Response Back to Angular: once MongoDB provides the data, Express.js sends a JSON response to Angular. It includes requested data and success or error message.

Step8: Angular updates the UI: Angular receives the response and updates the UI dynamically. Components display data without requiring a full page reload.



Q3. ~~Illustrate how the MEAN stack components work together.~~

Q3. Explain various types of Version Control System, Explain Diff b/w Git & Github.

ans ⇒ A version control system (VCS) is a software tool that helps developers track changes in code, collaborate on projects, and maintain a history of modifications. There are three main types of version control systems.

1. Local Version Control System (LVCS): A Local VCS is a simple system where all file versions are maintained on a single machine. Developers manually track changes and create backups.

How it works:

- Developers maintain multiple copies of a project at different stages. Changes are stored in separate directories.

pros: (i) Simple and easy to use (ii) Good for personal projects. (iii) No need for an internet

Cons: (i) ~~Simple~~ Risk of data loss if the system crashes
(ii) Difficult to collaborate in a team.
(iii) No centralized backup system.
Example: Revision control system

(2) Centralized version control system (CVCS):
A centralized VCS uses a central server that stores all code versions. Developers pull (Download) the latest code and commit (upload) changes to the central repository.
How it works
(i) A central server manages all versions and history.
(ii) Developers connect to the central repository to check out and commit changes.
(iii) The server ensures data consistency and acts as a single source of truth.

pro s: (i) Easy to manage with a single repository.
(ii) Simplifies collaboration in a team.
(iii) provides a backup of the code in a central location.
Cons: (i) If the central server crashes, all version history may be lost.
(ii) Requires an internet connection to work.
(iii) Developers cannot work offline.
Ex ⇒ Apache Subversion.

3. Distributed
A distribu
a full copy
history
comm

3. Distributed version control system (DVCS):
A distributed VCS allows each developer to have a full copy of the repository, including the entire history of changes. It eliminates reliance on a central server.

How it works:
(i) Every developer has a local copy (clone) of the entire repository.
(ii) Developers can work offline, commit changes locally and later push them to a shared repository.
(iii) git supports branching and merging efficiently.

Pros: (i) Works offline, no need for a constant internet connection.
(ii) No single point of failure (since to local developer has a full backup).
(iii) Faster performance due to local operations.
(iv) supports easy branching and merging.

Cons: (i) Requires more disk space due to multiple copies of the repository.
(ii) learning curve can be steep for beginners.

Example: Git, Bazaar, mercurial.

Comparison of VCS Types

Features	Local VCS	Centralized VCS	Distributed VCS
collaboration	Limited	Good	Excellent
works offline	Yes	No	Yes
Data loss	High	Medium	low
Speed	fast	slower	fast
Branching & Merging	Difficult	Limited	Easy

Difference between Git and Github

Features	Git	Github
Definition	A distributed version control system (VCS)	A cloud-based hosting service for Git repositories
Function	Manages source code versions locally.	provides a remote platform to store and collaborate on Git repositories.
installation	Needs to be installed on a local machine.	No installation required accessed via a web brower.
command-line	uses commands like git init, git commit, git push.	provides a web-based GUI but also supports Git commands.
Hosting	Does not provide any hosting services	Hosts Git repositories on the cloud.
Authentication	No Authentication needed for local use.	Requires an account for accessing private repositories
Repository Type	Local repository	Remote repositories
pull requests	Not available	Allows developer to propose and review changes
primary users	individual developers working locally.	Teams and open-source contributors working remotely.

Assignment - 2

Name : Raushan Kumar
CRN : 2221139
URN : 2203751
Sec : IT - B2

Raushan

Q1. Create a new document/database using MongoDB and perform CURD operation like insert, update read and delete data with example.

ans ⇒ Introduction to MongoDB

MongoDB is a NoSQL database that stores data in a flexible, JSON-like format known as BSON (Binary JSON). Unlike relational database, MongoDB does not use tables, rows or columns. Instead, it organizes data into Collections and documents.

- Database: A container for collections, similar to a database in SQL.
- collection: A group of documents, similar to a table in SQL.
- Document: A JSON-like data structure that holds key-value pairs.

1. Setting Up MongoDB:

To work with MongoDB, you need to install MongoDB Community Edition or use MongoDB Atlas;

Starting MongoDB Server (Locally)

(1) Open a terminal or Command prompt.

(2) Run the following command to start the MongoDB Server.

```
mongod
```

(3) Open another terminal and start the MongoDB shell.

mongosh

② Creating a New Database

MongoDB automatically creates a database when you insert data into it. However, you can switch to a database using.

```
use myDatabase
```

If myDatabase does not exist, MongoDB will create it when you insert the first document.

CRUD Operations in MongoDB

CRUD stands for Create, Read, Update, Delete. These are the fundamental operations performed on a database.

A. Create (Insert Data)

To insert data into a collection, we use the insertOne() or insertMany() methods.

Example 1: Insert a single Document

```
db.users.insertOne({
  name: "Rayhan",
  age: 28
});
```

If the users collections does not exist, MongoDB creates it automatically.

Example 2: Insert Multiple Documents

```
db.users.insertMany([{name:"Rayhan", age:25},
                     {name:"Rayhani", age:28}
                    ]);
```

This inserts multiple user documents into the users collections

(B). Read (Retrieve Data)

MongoDB provides the find() method to retrieve documents

Example 3: Retrieve All Documents

```
db.users.find();
```

Example 4: Retrieve a specific Document

```
db.users.findOne({name: "Raufhan"});
```

finds the first document where name is "Raufhan".

Find the first document where name is "Raufhan"

Example 5: Retrieve specific fields

```
db.users.find({}, {name:1, email:1, _id:0});
```

This retrives only name and email fields for all users excluding _id).

Example 6: Query using conditions

```
db.users.fing({age:{$gt:25}});
```

find users where age is greater than 25.

③ Update (Modify data).

MongoDB provides updateOne(), updateMany(), and replaceOne() for Modification.

Example 7: Update a single Document

```
db.users.updateOne(
  {name:"Raufhan"},
  {$set:{age:29, city:"chapra"}}
);
```

Update Raufhan's age and city.

Example 8: Update multiple Documents

```
db.users.updateMany(
  {city:"New York"},
  {$set:{country:"USA"}}
);
```

Adds a country field to all users in New York.

Example: Increment a value

```
db.users.updateOne
  {name:"Raufhan"},
  {$inc:{age:1}});
```

increment Raufhan's age by 1.

(D) Delete (Remove Data)

MongoDB provides deleteOne() and deleteMany() to remove documents.

Example 10: Delete a single Document

```
db.users.deleteOne({name: "Raufhan"});
```

Delete the first document where name is "Raufhan"

Example 11: Delete multiple Documents

```
db.users.deleteMany({city: "Los Angeles"})
```

Deletes all users from Los Angeles.

Example 12: Delete All Documents

```
db.users.deleteMany({})
```

Removes all documents from the users collection.

MSTIS

(8M) Architecture of Mean stack

any ⇒ How to any

(i) introduction (definition of mean) ✓ done

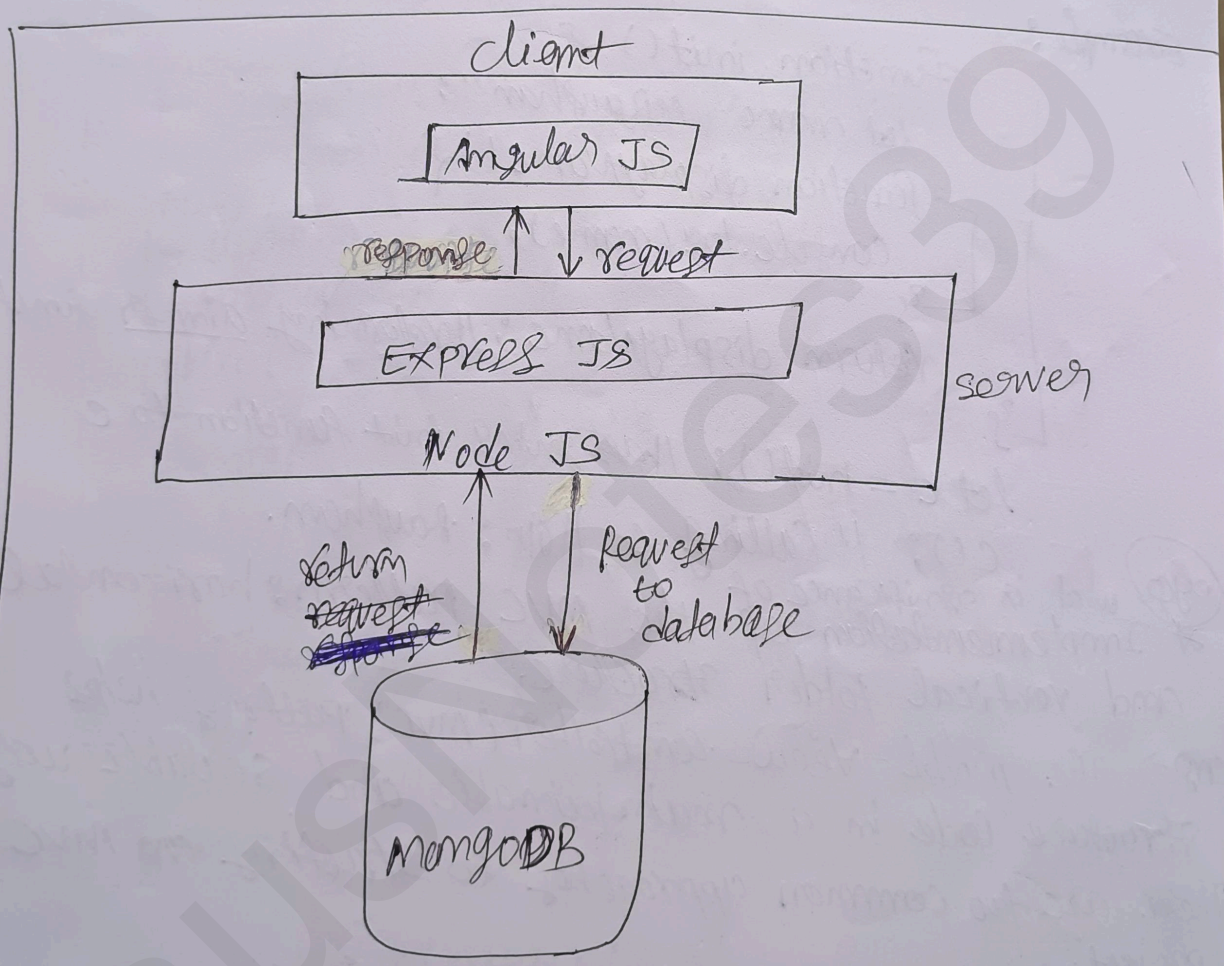
(ii) Mean stack components (explain each separately only defn) ✓

(iii) Architectural diagram (draw diagram) ✓ below

(iv) How MEAN components works together. ✓

(v) Advantages of MEAN stack only ✓

Architecture of MEAN Stack



(*) Explain Javascript closures?
A closure in javascript is a function that retains access to its lexical scope, even when the function is executed outside of its original scope.

In simple term
A closure is allows a function to remember the variable from its outer function, even after the outer function has finished executing.

Example:

```
function init() {
  let name = "Raufhan";
  function displayName() {
    console.log(name);
  }
  return displayName; // returning inner function
}
let c = init(); // assigning init function to c
c(); // Calling c Op: Raufhan.
```

(pyq) what is significance of Implementation of the MVC pattern: horizontal and vertical folder structure

ans ⇒ The Model-View-Controller (MVC) pattern helps structure code in a maintainable and scalable way.

There are two common approaches to structing an MVC project.

(i) Horizontal folder structure: organises the project by layer (models, views, controllers).

(ii) vertical folder structure: organises the project by feature/module, grouping related models, views, and controllers together.

(1) Horizontal
This is the
medium

① Horizontal Folder Structure (Layered Approach):
This is the most common structure for small to medium-sized applications.

- Significance
  1. Easy to locate files by functions
  2. Suitable for small to medium app.
  3. Promotes separation of concerns by type.
  4. Layered approach

```
project/
├── models/
│   ├── userModel.js        // database interactions
│   └── postModel.js
├── views/
│   ├── index.ejs           // UI templates
│   └── profile.ejs
├── controllers/
│   ├── userController.js   // Handle user request
│   └── postController.js
├── routes/
│   ├── userRoutes.js       // api endpoints
│   └── postRoutes.js
├── public/
│   ├── css/                // static assets
│   └── js/
├── config/
│   └── db.js               // database configuration
├── middleware/             // middleware function
├── utils/                  // utility funct.
├── server.js               // entry point
├── package.json
└── .env
```

② Vertical Folder Structure (Feature-Based Approach):
This is useful for larger applications, where each module has its own Model, View, Controller and Route.

```
project/
├── modules/
│   ├── user/
│   │   ├── userModel.js
│   │   ├── userController.js
│   │   ├── userRoutes.js
│   │   └── views/
│   │       ├── login.ejs
│   │       └── profile.ejs
│   └── post/
│       ├── postModel.js
│       ├── postController.js
│       ├── postRoutes.js
│       └── views/
│           └── post.ejs
├── config/
├── middleware/
├── public/
├── utils/
├── server.js
├── package.json
└── .env
```

- Significance
  - (i) Better for large-scale or modular app.
  - (ii) Improves feature-based development and team collaboration.
  - (iii) Makes refactoring and scaling easier.
  - (iv) Feature Based approach
  - (v) Each module have its own model, view and controller

Q) Rendering views, serving static files.

ans⇒ In express, you can render dynamic views using template engine like EJS, pug and serve static files (css, image, JS) from a public directory.

1. Rendering views with EJS

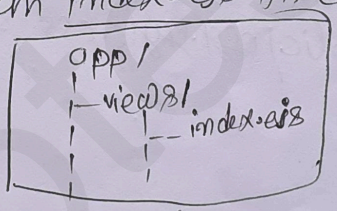
Step1: Install EJS

npm install ejs

Step2: Configure express to use EJS.

```
const express = require('express');
const app = express();
// Set EJS as the view engine
app.set('view engine', 'ejs');
// Define a route to render an ejs view
app.get('/', (req, res) => {
res.render('index', { title: 'Home page', message:
'welcome to express!' });
});
app.listen(3000, () => console.log('Server is running on port 3000'));
```

Step3: Create the views directory: inside your project create a views folder and add an index.ejs file.




```
<!DOCTYPE html>
<html>
<head>
<title><%= title %></title>
<link rel="stylesheet" href="/css/style.css">
</head>
<body>
<h1><%= message %></h1>
</body>
</html>
```

Run your project node app.js

② Serving static files (CSS, JS, image): Static files should be placed inside a public directory

step 1: create the public directory inside your project
- create a folder public/ and add subdirectories

```
public/
├── css/
│   └── style.css
├── images/
│   └── logo.png
└── js/
    └── script.js
```

step 2: Serve static files in express

```
app.use(express.static('public'));
```

step 3: create a css file
public/css/style.css →

style.css
```
body {
    color: blue
}
h1 {color: blaele;
}
```

step 4:
~~step 4~~: Link CSS in ejs : views/index.ejs

```
<link rel='stylesheet' href="/css/style.css" />
```

step 5: Serving images & Javascript

```
<img src="/images/logo.png" />
```
images in public/images/logo

Serving Javascript files /public/JS

```
<script src="/js/script.js"></script>
```

(*) Implementation of MVC pattern:

An MVC framework is a software framework that follows the model-View-Controller (MVC) pattern, which separates the application into three interconnected components: Model, View, and Controller. This separation enhances modularity, maintainability, scalability, and testability. MVC frameworks commonly used in web development to build robust, scalable applications by organizing code efficiently.

1. Understanding MVC

(a) Model: It manages database interaction (CRUD operat). validates data before sending it. sends data to the controller when requested.

Example: Mongoose model for MongoDB)

```
Const mongoose = require('mongoose');
Const userSchema = new mongoose.Schema({
  name: String,
  email: String,
  password: String
});
Const User = mongoose.model('User', userSchema);
module.exports = User;
```

(b) View (User interface): The view responsible for displaying data. It sends UI updates based on the controller's instruction. Its views it can be HTML, CSS, Javascript, or templating engine like EJS, pug.

Example in EJS:

```
<!DOCTYPE html>
<html>
<head>
  <title> User List Title </title>
</head>
<body>
    <h1> Users </h1>
    <ul>
      <% users.forEach(user => { %>
       <li> <%= user.name %> -
          <%= user.email %>
       </li> <% }) %>
    </ul>
</body>
</html>
```

(c) Controller: Controller is an intermediary b/w model and view, it is responsible for handling request and response, it receives user requests, process them and interact with Model using controller methods and updates view based on Model data.

Example:

```
const User = require("../models/User");
// fetch all users
exports.getAllUsers = async(req, res) => {
  const users = await User.find();
  res.render('index', users);
};
exports.addUser = async (req, res) => {
const {name, email, password} = req.body;
const newUser = new User({name, email, password});
await newUser.save(); res.redirect("/users");
```

Advantages of MVC

(i) Code Reusability: Separation of concerns allows for better code organization and reusability.

(ii) Scalability: Large applications can be easily manage and expanded.

(iii) Maintainability: Each layer (Model, view, controller) can be updated independently.

(iv) Parallel Development: Teams can work on different parts of the application simultaneously.

(v) Testability: Unit testing is easier since the Model is separate from UI (views).

(6 Marks) Discuss the process of implementing of MVC pattern. Explain the horizontal and vertical folder structure with example.

AngularJS

* What are modules in AngularJS?

ans → In angular js a module is like a container that holds different parts of your application - such as controllers, services, directives, filters etc.

why use modules

(i) To separate concerns (keep controllers, services organized)
(ii) To make the app scalable
(iii) To support code reuse
(iv) To keep everything modular.

Syntax of creating a module

```
let app = angular.module("myApp", []);
```

myApp → is the module name
[] → means this module has no dependencies

Ex →

```
<!DOCTYPE html>
<html ng-app="myApp">
<head>
  <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js"></script>
  <script>
    // define a module
    let app = angular.module("myApp", []);
    // create a controller inside the module
    app.controller("myctrl", function($scope) {
        $scope.name = "Raushan";
    });
  </script>
</head>
<body>
```

If connect the controller to a part of the page

```
<div ng-controller="myCntrl">
  Hello {{name}}!
</div>
</body>
</html>
```

(pgQ) explain two way data-binding, provide an exp of how it can be implemented in AngularJS

* What is TWO-way Data Binding?

Two-way data binding in AngularJS means automatic synchronization of data between the model and the view (UI). If you update the data in the model, it reflects in the view, and if you change the input in the view, it automatically updates the model. This is one of the most powerful features of AngularJS that makes it easy to build dynamic and interactive web application. AngularJS uses the ng-model directive for two way binding. Any changes made to the input field update the associated model value and vice-versa.

```
<!DOCTYPE html>
<html ng-app="myAPP">
<head>
  <title> Two-way data Binding </title>
  <script src="angular.min.js"></script>
</head>
<body ng-controller="mainController">
  <p> Enter your name : </p>
  <input type="text" ng-model="name" />
  <p> Hello, {{name}}! </p>
<Bcript>
let app = angular.module("myAPP", []);
```

app.controller('MainController', function($scope) {
$scope.name = "Raufhan";
});
</script>
</body>
</html>

ng-app = initializing the AngularJS app
ng-controller = mainController defining scope
ng-model = creates a two-way databinding b/w input and $scope.name

* what is a Directives?

A directives is a marker on a DOM element (such as an attribute, element name, comment, or CSS class) that tells AngularJS's HTML compiler $compiler to attach a specified behavior to that DOM element or even transform the DOM element and its children.

Types of Directives in AngularJS

(i) Built-in-Directives: provided by AngularJS itself.
ng-app: Defines root element of an angular JS app.
(ii) ng-model: Binds the value of HTML controls
ng-bind: Replaces the innerHTML of an element with the value of given expression.
ng-repeat: Instantiates a template once per item from a collection.
ng-if: Removes or recreates a portion of the DOM tree.
ng-click: Specifies custom behavior when an ele is clicked

```
<!DOCTYPE html>
<html ng-app="myApp">
<head>
<script src="angular.min.js"></script></head>
```

```
<body ng-Controller="myCtrl">
  <input type="text" ng-model="name">
  <p>Hello, <span ng-bind="name"></span></p>
<ul>
  <li ng-repeat="item in items">{{item}}</li>
</ul>
<button ng-click="addItem()"> Add Item</button>
<script>
  let app = angular.module('myApp', []);
app.Controller('myCtrl', function($scope){
  $scope.name = "Raufhan";
  $scope.items = ['Apple', 'Banana', 'Orange'];
  $scope.addItem = function(){
      $scope.items.push("New Item");
  };
});
</script>
</body>
</html>
```

② Custom directives: you can create your own directive using the directive() function.

~~Synatx~~ syntax;

```
app.directive('directiveName', function(){
  return{
    restrict: 'E' or 'A' or 'C' or 'M',
    template: 'HTML content or template URL',
    link: function(scope, element, attrs){
      // behavior
    }
  };
});
```

restrict options

'A' → Attribute (e.g. <div my-directive>)

'E' → Element (<my-directive></my-directive>

'C' → class (

'M' → Comment (rarely used)

```
<!DOCTYPE html>
<html ng-app="myApp">
<script src="angular.min.js"></script>
</head>
<body>
<hello-directive></hello-directive>
<script>
let app=angular.module('myApp',[]);
app.directive('helloDirective', function(){
    return {
    restrict :'E',
    template :'<h1>Hello from custom directive!</h1>',
    };
});
</script>
</body>
</html>
```

* Structuring and Bootstrapping AngularJS App

(i) Basic project structure for AngularJS

```
angularJS-app/
├── index.html
├── app/
│   ├── app.module.js
│   ├── app.routes.js
│   ├── controllers/
│   │   └── main.controller.js
│   ├── services/
│   │   └── data.service.js
│   ├── view/
│   │   └── home.html
│   └── css/
│       └── style.css
├── js/
│   └── lib/
│       └── angular.min.js
```

what is Bootstrapping in AngularJS

ans ⇒ Bootstrapping in angularJS is the process of initializing or starting an angularJS application. It tells AngularJS which module to load and where to start the application on the HTML page.

There are two types of Bootstrapping

1. Automatic Bootstrapping:
   - How: using the ng-app directive
   - what happens: AngularJS automatically scans the HTML page when loaded, finds the ng-app directive, and bootstraps the application using the specified module.

Example:

```
<!DOCTYPE html>
<html ng-app="myApp">
  <script src="angular.min.js"></script>
  <script>
     angular.module('myApp',[]);
  </script>
</head>
<body>
   <h1> AngularJS App </h1>
</body>
</html>
```

2. manual Bootstrapping:

How: using angular.bootstrap() function

what happens: you manually initilize the angularJS application in javascript after the DOM is fully loaded.

```
<!DOCTYPE html>
<html>
<head>
  <script src="angular.min.js"></script>
  <script>
    let app = angular.module("myApp", []);
    angular.element(document).ready(function() {
    angular.bootstrap(document, ['myApp']);
    });
  </script>
</head>
<body>
  <h1>Manual Bootstrap Example</h1>
</body>
</html>
```

* MVC pattern in AngularJS

AngularJS is follows the MVC (Model View Controller) design pattern which seperates the concerns of an application into three major components.

1. Model : Represents the data or business logic of the application
2. View : Represents the user interface (UI). what the user sees and interacts with.
3. Controller : Acts as an intermediary b/w the Model and View, handling the logic and user interaction.

In angularJS these components are mapped as

- Model : $Scope or service/factory
- View : HTML with AngularJS directives like ng-model, ng-bind, ng-repeat. etc.
- Controller : Javascript code where logic, data processing and event handling occur.

1. Model in AngularJS : In AngularJS, the Model is typically represented by the data in your application and how it is managed or manipulated. It is usually stored in a scope or retrived from service/factory.

The Model is a part of the $scope object in a controller or inside a service/factory.

~~The Model is a part of the $scope object in a controller or inside a service/factory.~~

Ex:-
```
angular.module('myApp', [])
.controller('MainController', function($scope){
  $scope.message = "Welcome to Angular MVC";
});
```
Here $scope.message is part of the Model in the controller.

2. View in AngularJS : The View in AngularJS is simply the HTML, augmented with AngularJS directives that connect it to the Controller. It displays data and reacts to changes in the Model.

```
<!DOCTYPE html>
<html ng-app="myApp">
<head>
<script src="angular.min.js"></script>
</head>
<body ng-controller="MainController">
  <h1>{{message}}</h1>
  <input ng-model="userInput" />
  <button ng-click="addMessage(userInput)">
    Add Message
  </button>
<ul>
  <li ng-repeat="msg in messages">{{msg}}</li></ul>
```

in this example

{{message}} and {{msg}} are bindings from the Model to the view.

ng-model binds the input field to the scope

ng-repeat loops through an array and displays the data

(5) Controller in Angular JS

The Controller in AngularJS is where the logic resides. It is responsible for managing data (model), handling user input and updating view. It acts as an intermediary b/w the Model and view.

```
angular.module('myApp', [])
.controller('MainController', function ($scope, DataService) {
  // Model
  $scope.message = "welcome to AngularJS MVC";
  // Controller
  $scope.messages = DataService.getData();

  $scope.addMessage = function (message) {
    DataService.addData(message);
    $scope.messages = DataService.getData();
  };
});
```

* Routing in Angular JS:

Routing in AngularJS allows you to create single page Application where the view changes dynamically without reloading the entire page. The ngRoute module is used to implement routing in AngularJS

How Routing works in AngularJS

1. Define Routes: you specify which URL path should correspond to which Controller and view.

2. Update URL: When the user clicks on a link, the URL updates and AngularJS loads the corresponding view dynamically

③ Dynamic Views: Based on the URL, AngularJS loads the appropriate view without reloading the entire page.

④ Controller Association: Each view has an associated controller that manages its logic.

Key Components of AngularJS Routing

① ngRoute module: This is the core module that enables routing in AngularJS.

② $routeProvider: This service is used to define the routes.

③ ng-view: This directive is used to load the view dynamically.

Example of Routing in AngularJS

index.html

[PYQ. How to structuring an AngularJS web application also include dir structure]

```
<!DOCTYPE html>
<html ng-app="myApp">
<head>
  <script src="angular.min.js"></script>
  <script src="angular-route.min.js"></script>
  <script src="app.js"></script>
</head>
<body> <div>
  <ul>
    <li><a href="#!/">Home</a></li>
    <li><a href="#!/about">About</a></li>
  </ul>
</div>
// This is where the view will be loaded dynamically
<div ng-view></div>
</body>
</html>
```

app.js ⇒ let app =
angular.module('myApp', ['ngRoute']); // Include the ngRoute module

app
.config(function($routeProvider)
{ $routeProvider
.when('/', {
templateUrl: 'home.html', // Template for Home route
Controller: 'HomeController' // Controller for Home route
})
.when('/about', {
templateUrl: 'about.html'; // Template for About route
Controller: 'AboutController' // Controller for about route
})
.otherwise({
redirectTo: '/'. // Default route if no match
}));

app.controller('HomeController', function($scope){
$scope.message = "welcome to the About page!";
});

Home view (home.html)
<h2> Home page </he>
 {{message}}

About view (about.html)
<h2> About page </he>
 {{message}}

* Services in AngularJS: Services in AngularJS are singleton object that are used to organize and share code across your application. They provide a way to encapsulate reusable logic like data fetching, validation, business rules, and more.

Why use services?

- To separate concerns (keep controllers thin)
- To reuse logic across multiple controllers/directives
- To perform data operations like HTTP calls
- To maintain application state

Built-in AngularJS services

AngularJS provides many built-in services such as:

Service	purpose
$http	for Ajax requests
$route	for routing info
$timeout	for delying function execution
$interval	Repeatedly call functions with time gaps
$location	URL manipulation and navigation.

*$http service: Used to make Http requests (Get, post, put, delete)

```
$http.get("https://api.example.com/data").
then(function(response){
    $scope.data = response.data;
});
```

*$route service: provides current route information like path, parameters, etc. let app = angular.module('myApp', ['ngRoute']);

To use $route, you must include the ngRoute module.

```
app.controller('RouteCtrl', function($scope, $route){
    console.log($route.current.templateUrl); // logs current template
});
```

* $timeout service: used to delay function execution, like JavaScript's setTimeout.

$timeout(function(){
$scope.message = "Updated after 3 seconds!!!";
}, 3000); // Delay in milliseconds

* interval service : Used to run a function repeatedly at fixed time intervals like setInterval.

$interval(function(){
$scope.time = new Date().toLocaleTimeString();
}, 1000);

* $location service : Used to read or change the URL in the browser

app.controller("LocationCtrl", function($scope, $location){
$scope.currentPath = $location.path(); // Get current path
$location.path("/newRoute"); // Navigate to new route
});

Creating a custom Service : AngularJS provides multiple ways to create custom services.

1. service() function
2. factory() function
3. provider() function

1. Using .service() :

app.service("mathService", function(){
this.add = function(a, b){
return a+b;
};
}); // use it in a controller

app.controller("MainCtrl", function($scope, mathService){
$scope.sum = mathService.add(10, 20);
});

(2.) using .factory()

```
app.factory('DataService', function(){
    let data = {};
    data.getMessage = function(){
    return "Hello from factory";
    };
    return data;
});
```

use in controller

```
app.controller('MainCtrl', function($scope, DataService){
    $scope.message = DataService.getMessage();
});
```

(3.) using .provider():

```
app.provider('GreetService', function(){
    let greet = "Hello";
    this.setGreeting = function(text){
        greet = text;
    };
    this.$get = function(){
    return {
        greetUser: function(name){
            return greet + ", " + name;
        }
    };
});
```

configure in .config;

```
app.config(function(GreetServiceProvider){
    GreetServiceProvider.setGreeting("Welcome");
});
```

use in controller

```
app.controller('MainCtrl', function($scope, GreetService){
    $scope.greeting = GreetService.greetUser("Rushan");
});
```

* Managing Authentication in AngularJS

Authentication in AngularJS involves verifying the user's identity and managing access to protected routes or resources. While AngularJS is a frontend framework, it typically works in coordination with a backend API that handles actual authentication.
(e.g. token based, sessions)

Example: AngularJS - Token-Based Authentication

1. Login Form (HTML)

```
<div ng-controller="LoginCtrl">
 <form ng-submit="login()">
  <input type="text" ng-model="user.username" />
  <input type="password" ng-model="user.password" />
  <button type="submit">Login</button>
 </form>
</div>
```

(2) Login Controller

```
app.controller('LoginCtrl', function($scope, $http, $window){
    $scope.user = {};
    $scope.login = function(){
     $http.post('/api/login', $scope.user)
     .then(function(response){
     $window.localStorage.setItem('token', response.data.
                                  token);
     alert('login successfull');
     }, function(error){
        alert('login failed');});});
```

③ HTTP interceptor to Attach Token

```
app.factory('AuthInterceptor', function($window) {
  return {
    request: function(config) {
      let token = $window.localStorage.getItem('token');
      if (token)
      { config.headers.Authorization = 'Bearer ' + token;
      } return config;
    }
  };
app.config(function($httpProvider) {
  $httpProvider.interceptors.push('AuthInterceptor');
});
```

④ Protecting Routes

```
app.config(function($routeProvider) {
  $routeProvider
  .when('/dashboard', {
  templateUrl: 'dashboard.html',
  controller: 'DashboardCtrl',
  resolve: {
  auth: function($q, $window) {
    let token = $window.localStorage.getItem('token');
    if (token) {
      return true;
    } else { return $q.reject('Not Authenticated');
    }}}});
```

Angular

Creating angular app with TYPESCRIPT and working with Angular Components [How the components of simple Angular app fit and work together]

So,": prerequisites:

1. Node.js and npm installed.
2. Angular.CLI Installed globally.

`npm install -g @angular/cli`

Step1: create a New Angular project

Run the following command to create a new Angular project

`ng new MYAPP`

Step2: project structure overview: After cd MyApp ~~After MYAPP~~ you'll see

```
MYAPP/
├─ src/
│   ├─ app/
│   │   ├─ app.component.ts   // Main component typescript file
│   │   ├─ app.component.html // HTML template
│   │   ├─ app.component.css  // Component styles
│   │   └─ app.module.ts      // Root module
│   ├─ assets/
│   └─ environment/           // environment variables
├─ angular.json               // cli config
├─ package.json               // NPM config
└─ tsConfig.json              // Typescript config
```

Step3: Run the Ang

`ng serve`

Step4: Create new

Step 3: Run the Angular App:

`ng serve` This starts the development server at http://localhost:4200

Step 4: Create a component using Type script create a new component

`ng generate component hell-world`

it creates:

src/app/hello-world/

- hello-world.component.ts
- hello-world.component.html
- hello-world.component.css
- hello-world.component.spec.ts

Modify hello-world.component.ts:

```
import {Component} from "@angular/core";
@Component({
  selector: 'app-hello-world',
  templateUrl: './hello-world.component.html',
  styleUrls: ['./hello-world.component.css']
})
export class HelloWorldComponent {
  message: string = "Hello from angular with TS";
}
```

Update hello-world.component.html:

```
<h2> {{message}} </h2>
```

Step 5: Add Component to App
Edit app.component.html:

<app-hello-world></app-hello-world>

(245)

(*) Building a single page application with Angular: Adding navigation, building modular app, binding HTML content, routing parameters, working with forms and handling submitted data,

ans ⇒ Here's a step by step guide on building a single page Application with Angular,

Step1: Set up Angular Project

```
npm install -g @angular/cli
ng new angular-spa-demo
cd angular-spa-demo
ng serve
```

→ This will run the development at localhost:4200

Step 2: Add Navigation and Routing (a) Create Component

```
ng generate component home
ng generate component about
ng generate Component Contact
```

(b) Set up Routing: In app-routing.module.ts

```
import {NgModule} from '@angular/core';
import {RouterModule, Routes} from '@angular/router';
import {HomeComponent} from './home/home.component';
import {AboutComponent} from './about/about.component';
import {ContactComponent} from './contact/contact.component';
```

```
const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'about', component: AboutComponent },
  { path: 'contact/:id', component: ContactComponent }
];
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

(c) Add navigation Links in app.component.html

```
<nav>
  <a routerLink="">Home</a>
  <a routerLink="/about">About</a>
  <a [routerLink]="['/contact', 1]">
    Contact (with ID)</a>
</nav>
<router-outlet></router-outlet>
```

Fill here would be the example of how to design SPA in Ang

Step 3: Building a Modular app

(a). Create a feature Module

ng generate module products (b) Create productList component

ng generate component products/product-list

(c) Add route in app-routing.module.ts:

```
{ path: 'products', loadChildren: () => import('./products/products.module').then(m => m.ProductsModule) }
```

In products/products.module.ts:

```
import { NgModule } from '@angular/common';
import { CommonModule } from '@angular/router';
import { ProductListComponent } from './product-list/product-list.component';
```

```
const routes: Routes = [
  { path: '', component: ProductListComponent }
];
@NgModule({
  declarations: [ProductListComponent],
  imports: [
    CommonModule,
    RouterModule.forChild(routes)
  ]
})
export class ProductModule { }
```

Step 4: Binding HTML content

in home.component.ts:

```
export class HomeComponent
{ title = 'Welcome to Angular SPA';
  imageUrl: 'https://angular.io/assets/images/logos/angular/angular.png';
}
```

In home.component.html

```
<h2>{{title}}</h2>
<img [src]="imageUrl" >
```

Step 5: Routing parameters

In contact.component.ts:

```
import { ActivatedRoute } from '@angular/router';
export class ContactComponent {
  contactId: string;
  constructor(private route: ActivatedRoute) {
    this.contactId = this.route.snapshot.paramMap.get('id');
  }
}
```

In contact.component.html

```
<p>contact ID from URL: {{contactId}}</p>
```

Steps: ① Import forms in app.module

Step 6: Working with forms

(a) Import FormsModule

In app.module.ts:

```
import { FormsModule } from '@angular/forms';
@NgModule({
  imports: [FormsModule]
});
```

(b) Create form in HTML

In about.component.ts

```
export class AboutComponent {
  username = '';
  onSubmit() {
    alert('Form submitted!
      Name: ${this.username}');
  }
}
```

In about.component.html

```
<form (ngSubmit)="onSubmit()">
  <label for="username">Name</label>
  <input [(ngModel)]="username" name="username">
  <button type="submit">Submit</button>
</form>
```

Step 7: Handling submitted data: In the onSubmit() method above, the form data is accessed using two-way data binding [(ngModel)] and displayed via alert. In real apps, you would send this data to a server using Angular's HttpClient service.

* Services in Angular: Services are used to organize and share data or logic across multiple components. They are commonly used for.
  - Fetching data from API.
  - Business logic

Creating service

`ng generate service data` this will create a component called data.service.ts

In `data.service.ts`

```
import { injectable } from "@angular/core";
@injectable({
  providedIn: 'root'
})
export class DataService {
  getMessage()
  { return "Hell from data service";
}}
```

using service in a component

In `home.component.ts`

```
import { Component, OnInit } from "@angular/core";
import { DataService } from "../data.service";
@Component({
  selector: 'app-home',
  template: "<h2>{{message}}</h2>"
})
export class HomeComponent implements OnInit {
  message: String = "";
  constructor(private dataService: DataService) { }
  ngOnInit()
  { this.message = this.dataService.getMessage();
}}
```

Explanation: Data service provides a method getMessage() HomeComponent injects the service using dependency injection and calls the method in ngOnInit() to assign the message.

* Understanding Event Driven programming in Javascript

ans=> Event-Driven programming is a style of coding in Javascript where the flow of program is controlled by events.

An event is something that happens in your app - like
A user clicks a button.
A key is preled.

Instead of running everything from top to bottom seaverdially, Javascript waits for events ~~from top to bottom~~ to happen and then responds to them by running specific functions (called event handlers or callbacks).

Example:

```
<button onclick="sayHI()"> click me </button>
<script>
    function sayHI(){
      alert("HI Roushan!");
    }
</script>
```

when user clicks the button (event). Javascript calls the SayHI() function (Callback) And shows an alert with the meffage.

(P9C) Write a Node.js script that sets up a basic web server using the http module. The server should respond with "Hello, world" when accessed at the root URL.

ans ⇒

```
// Load http module to create http server
const http = require('http');
// Create server
const server = http.createServer((req, res) => {
  if (req.url === '/')
  { // Set the response header
    res.writeHead(200, {'Content-Type': 'text/plain'});
    // Send the "Hello world!" response
    res.end('Hello, world');
  }
  else
  { // Set response with a 404 (Not found)
    res.writeHead(404, {'Content-Type': 'text/plain'});
    res.end('Not found');
  }
}); // Set the server to listen on port 3000
server.listen(3000, () => {
console.log('Server running at http://localhost:3000/');
});
```

(P9G) Differentiate b/w various data models used for designing in MongoDB.

(I) Embedded Data Model: In embedded data model, related data is stored within a single document.

Advantages:

(i) Faster Reads: All related data in one document no need for joins or multiple query.

(ii) Good performance: since no complex joins are used, it leads to save time and performance very high.

Example:

```
{
  "_id": 1,
  "Name": "Raupham",
  "Age": 20,
  "Address": [
    {"city": "chapra"},
    {"country": "india"}
  ]
}
```

embedded data

(2) Referenced Data Model: In this model, data is stored in separate documents and linked using references (like: _id). This is similar to foreign keys in SQL.

Advantages:

(i) Data Redundancy: Avoids data duplication.

(ii) Smaller Documents: Keep individual documents smaller and clean

(3) Example: Book Doc

```
{
  "_id": 1,
  "Book_Name": "ABCD",
  "publication": "def",
  "Author": 2 ;  // reference to Author collection
}
```

Author collection document

```
{
  "id": 2,
  "Name": "Raupham",
  "Age": 25,
  "Gender": "Male"
}
```

(3) Flat Data Model: A flat data model stores all data in a single, non-nested document, without embedding or referencing other documents.

Advantages: (i) Simple structure: Easy to understand, query and manage

(ii) Fast Query performance: No nested documents or complex structure to process

(iii) No nested objects, no arrays of subarrays. Everything is directly accessible.

Ex =>

```
{ "_id" : 101,
"name" : "Raufhan",
"age" : 30,
"email" : "raufhan@gmail.com",
"phone" : 43123333,
"address" : "London, England",
"gender" : "Male",
"profession" : "Engineer",
}
```

(a) Normalized Data Model : Normalized data model separates data into multiple collections, and reference document to each other using (_id).

Advantages: (i) Avoid Redundancy: stores data in separate collections to avoid duplication.

(ii) Scalable: Better for large-scale application.

Example:

Collection : 1 Student

```
{ "_id" : 101,
"name" : "Raufhan",
"email" : "raufhan@gmail.com",
}
```

Collection : 2 Courses

```
{ "_id" : 102,
"course_name" : "IT",
"duration" : "4 years",
}
```

Collection : 3 Enrollement

```
{ "_id" : 103,
"student_id" : 101,
"course" : 102
}
```

reference

reference