**Q1: Build a utility App in TypeScript to manage employee salary calculations for a company. The company has a policy of giving performance-based bonuses. Your task is to implement these calculations using arrow functions wherever applicable. Define an array of employee objects. Each object should have: name (string), baseSalary (number), performanceRating (5 or 10). Based on performance rating, make a function to compute the final salary by applying the bonus to the base salary.**

ChatGPT said:

Here's a complete TypeScript utility app that:

1. Defines an Employee type.
2. Creates an array of employee objects.
3. Uses arrow functions to calculate performance-based bonuses and final salaries.

You can run this in a TypeScript environment like Node.js (with ts-node) or in a frontend framework like Angular or React.

✅ TypeScript Code:

ts

CopyEdit

// Define the Employee type

type Employee = {

name: string;

baseSalary: number;

performanceRating: 5 | 10; // Can only be 5 or 10

};

// Sample array of employees

const employees: Employee[] = [

{ name: "Alice", baseSalary: 50000, performanceRating: 10 },

{ name: "Bob", baseSalary: 60000, performanceRating: 5 },

{ name: "Charlie", baseSalary: 45000, performanceRating: 10 },

];

// Arrow function to calculate bonus percentage based on performance

const getBonusPercentage = (rating: 5 | 10): number =>

rating === 10 ? 0.2 : 0.1; // 20% for rating 10, 10% for rating 5

// Arrow function to compute final salary

const calculateFinalSalary = (employee: Employee): number =>

employee.baseSalary + employee.baseSalary \* getBonusPercentage(employee.performanceRating);

// Display each employee's final salary

employees.forEach(emp => {

const finalSalary = calculateFinalSalary(emp);

console.log(`${emp.name}'s final salary is ₹${finalSalary.toFixed(2)}`);

});

💡 Output Example:

pgsql

CopyEdit

Alice's final salary is ₹60000.00

Bob's final salary is ₹66000.00

Charlie's final salary is ₹54000.00

Would you like a version of this app with input from the user (via prompt or HTML form)?

You said:

**Q2: Develop a web application that uses AJAX to perform dynamic search functionality from a product catalog**

ChatGPT said:

Here’s a simple web application that uses AJAX to perform dynamic search on a product catalog using HTML, JavaScript, and a basic Flask backend with a JSON product list.

🔧 Project Structure

sql

CopyEdit

ajax-search-app/

├── static/

│ └── script.js

├── templates/

│ └── index.html

└── app.py

🧠 Step-by-Step Explanation

1. app.py (Flask Backend)

python

CopyEdit

from flask import Flask, render\_template, request, jsonify

app = Flask(\_\_name\_\_)

# Sample product catalog

PRODUCTS = [

{"id": 1, "name": "Apple iPhone 14"},

{"id": 2, "name": "Samsung Galaxy S22"},

{"id": 3, "name": "Google Pixel 7"},

{"id": 4, "name": "Sony WH-1000XM5"},

{"id": 5, "name": "Apple MacBook Air"},

]

@app.route("/")

def home():

return render\_template("index.html")

@app.route("/search")

def search():

query = request.args.get("q", "").lower()

results = [p for p in PRODUCTS if query in p["name"].lower()]

return jsonify(results)

if \_\_name\_\_ == "\_\_main\_\_":

app.run(debug=True)

2. templates/index.html

html

CopyEdit

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Product Search</title>

<script src="/static/script.js" defer></script>

<style>

body { font-family: Arial; margin: 2em; }

input { width: 300px; padding: 8px; }

ul { list-style-type: none; padding: 0; }

li { padding: 6px 0; }

</style>

</head>

<body>

<h2>Search Products</h2>

<input type="text" id="searchInput" placeholder="Start typing...">

<ul id="resultList"></ul>

</body>

</html>

3. static/script.js

javascript

CopyEdit

document.addEventListener("DOMContentLoaded", () => {

const input = document.getElementById("searchInput");

const resultList = document.getElementById("resultList");

input.addEventListener("input", () => {

const query = input.value;

fetch(`/search?q=${encodeURIComponent(query)}`)

.then(res => res.json())

.then(data => {

resultList.innerHTML = "";

if (data.length === 0) {

resultList.innerHTML = "<li>No products found</li>";

} else {

data.forEach(product => {

const li = document.createElement("li");

li.textContent = product.name;

resultList.appendChild(li);

});

}

});

});

});

▶️ How to Run

1. Save the files in a folder as shown above.
2. Run the app with:

bash

CopyEdit

python app.py

1. Open http://127.0.0.1:5000 in your browser.
2. Start typing in the search box to see results dynamically.

Would you like this app extended with TypeScript or connected to a real MongoDB backend?

You said:

**Q3: Create a registration form using AngularJS with the following fields Name , Email , Password Use ng-model to bind all input fields. Use ng-submit to handle form submission. The Password must be exactly 6 digitsusing ng-pattern.**

ChatGPT said:

Here is a simple AngularJS registration form that:

* Uses ng-model to bind Name, Email, and Password fields.
* Uses ng-submit to handle form submission.
* Validates that the password is exactly 6 digits using ng-pattern.

✅ Full HTML + AngularJS Code

html

CopyEdit

<!DOCTYPE html>

<html ng-app="registrationApp">

<head>

<meta charset="UTF-8">

<title>AngularJS Registration Form</title>

<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js"></script>

<style>

form { max-width: 300px; margin: 2em auto; font-family: Arial; }

input, button { display: block; margin-top: 10px; padding: 8px; width: 100%; }

.error { color: red; font-size: 0.9em; }

</style>

</head>

<body ng-controller="FormController">

<form name="regForm" ng-submit="submitForm()" novalidate>

<h2>Register</h2>

<!-- Name -->

<input type="text" name="name" ng-model="user.name" placeholder="Name" required>

<span class="error" ng-show="regForm.name.$touched && regForm.name.$invalid">Name is required.</span>

<!-- Email -->

<input type="email" name="email" ng-model="user.email" placeholder="Email" required>

<span class="error" ng-show="regForm.email.$touched && regForm.email.$invalid">Valid email is required.</span>

<!-- Password -->

<input type="password" name="password" ng-model="user.password" placeholder="6-digit Password"

ng-pattern="/^\d{6}$/" required>

<span class="error" ng-show="regForm.password.$touched && regForm.password.$error.pattern">

Password must be exactly 6 digits.

</span>

<!-- Submit -->

<button type="submit" ng-disabled="regForm.$invalid">Register</button>

</form>

<div ng-if="submitted">

<h3>Registration Successful</h3>

<p><strong>Name:</strong> {{ user.name }}</p>

<p><strong>Email:</strong> {{ user.email }}</p>

</div>

<script>

angular.module("registrationApp", [])

.controller("FormController", function($scope) {

$scope.user = {};

$scope.submitted = false;

$scope.submitForm = function () {

if ($scope.regForm.$valid) {

$scope.submitted = true;

}

};

});

</script>

</body>

</html>

🔍 Features:

* Uses ng-model for two-way data binding.
* ng-submit triggers submitForm() on valid form.
* Password field uses ng-pattern="/^\d{6}$/“ to allow only exactly six digits.
* Basic form validation with live error messages.

Would you like this upgraded to Angular (2+) with TypeScript?

You said:

**Q4: implement inheritance concepts. Make a base class vehicle make two derived classes at least (e.g car, bicycle etc) override the method of base class in derived classes.**

ChatGPT said:

Here's a TypeScript example that demonstrates inheritance and method overriding:

* Vehicle is the base class.
* Car and Bicycle are derived classes.
* Each derived class overrides the describe() method of Vehicle.

✅ TypeScript Code

ts

CopyEdit

// Base class

class Vehicle {

brand: string;

constructor(brand: string) {

this.brand = brand;

}

describe(): void {

console.log(`This is a vehicle from brand: ${this.brand}`);

}

}

// Derived class: Car

class Car extends Vehicle {

numberOfDoors: number;

constructor(brand: string, numberOfDoors: number) {

super(brand);

this.numberOfDoors = numberOfDoors;

}

// Override describe()

describe(): void {

console.log(`This is a car from ${this.brand} with ${this.numberOfDoors} doors.`);

}

}

// Derived class: Bicycle

class Bicycle extends Vehicle {

hasGear: boolean;

constructor(brand: string, hasGear: boolean) {

super(brand);

this.hasGear = hasGear;

}

// Override describe()

describe(): void {

const gearInfo = this.hasGear ? "has gears" : "does not have gears";

console.log(`This is a bicycle from ${this.brand} and it ${gearInfo}.`);

}

}

// Create instances and call describe()

const car1 = new Car("Toyota", 4);

const bike1 = new Bicycle("Hercules", true);

car1.describe(); // Output: This is a car from Toyota with 4 doors.

bike1.describe(); // Output: This is a bicycle from Hercules and it has gears.

🧠 Key Concepts:

* super() is used to call the parent class constructor.
* The describe() method is overridden in both Car and Bicycle to provide class-specific output.

Would you like this integrated into a simple HTML page to take input and show results interactively?

**Q5: Implementation of library system using typescript Create module of book transaction users, Use import and export statement , Error handling**

ChatGPT said:

Here's a complete implementation of a Library Management System using TypeScript modules, import/export, and error handling.

✅ Objective

Implement:

* A module for Book
* A module for User
* A main file to simulate book transactions like issuing and returning a book.
* Use try/catch blocks for error handling.

📁 Folder Structure

pgsql

CopyEdit

library-system/

├── book.ts

├── user.ts

└── main.ts

You can compile and run this with Node + TypeScript (tsc compiler).

📚 book.ts – Book Module

ts

CopyEdit

export class Book {

constructor(

public id: number,

public title: string,

public author: string,

public isIssued: boolean = false

) {}

issue(): void {

if (this.isIssued) {

throw new Error(`Book "${this.title}" is already issued.`);

}

this.isIssued = true;

}

returnBook(): void {

if (!this.isIssued) {

throw new Error(`Book "${this.title}" was not issued.`);

}

this.isIssued = false;

}

}

👤 user.ts – User Module

ts

CopyEdit

import { Book } from "./book";

export class User {

public issuedBooks: Book[] = [];

constructor(public id: number, public name: string) {}

issueBook(book: Book): void {

try {

book.issue();

this.issuedBooks.push(book);

console.log(`${this.name} issued "${book.title}".`);

} catch (error: any) {

console.error(`Error issuing book: ${error.message}`);

}

}

returnBook(book: Book): void {

try {

book.returnBook();

this.issuedBooks = this.issuedBooks.filter(b => b.id !== book.id);

console.log(`${this.name} returned "${book.title}".`);

} catch (error: any) {

console.error(`Error returning book: ${error.message}`);

}

}

}

🧪 main.ts – Entry Point

ts

CopyEdit

import { Book } from "./book";

import { User } from "./user";

// Sample books

const book1 = new Book(1, "1984", "George Orwell");

const book2 = new Book(2, "To Kill a Mockingbird", "Harper Lee");

// Sample user

const user1 = new User(101, "Alice");

// Transactions

user1.issueBook(book1); // Alice issued "1984".

user1.issueBook(book1); // Error: already issued

user1.returnBook(book2); // Error: book wasn't issued

user1.returnBook(book1); // Alice returned "1984".

🔧 Compile & Run

1. Install TypeScript if not already:

bash

CopyEdit

npm install -g typescript

1. Compile:

bash

CopyEdit

tsc main.ts book.ts user.ts

1. Run:

bash

CopyEdit

node main.js

💡 Error Handling Highlights:

* Catches duplicate issue attempts.
* Catches return attempts on non-issued books.
* Error messages are shown using try/catch.

Would you like this extended with more features like due dates, penalties, or a database connection?

You said:

**Q6: multi-level inheritance question**

ChatGPT said:

Here’s a TypeScript example demonstrating multi-level inheritance — a concept where a class inherits from a class that itself inherits from another class.

✅ Scenario: Machine → Vehicle → Car

* Machine is the base class.
* Vehicle extends Machine.
* Car extends Vehicle.

Each class adds or overrides behavior.

🚘 TypeScript Code (Multi-level Inheritance)

ts

CopyEdit

// Base class

class Machine {

type: string;

constructor(type: string) {

this.type = type;

}

start(): void {

console.log(`${this.type} is starting...`);

}

}

// First derived class

class Vehicle extends Machine {

brand: string;

constructor(type: string, brand: string) {

super(type); // Call Machine's constructor

this.brand = brand;

}

move(): void {

console.log(`The ${this.brand} vehicle is moving.`);

}

}

// Second derived class

class Car extends Vehicle {

model: string;

constructor(type: string, brand: string, model: string) {

super(type, brand); // Call Vehicle's constructor

this.model = model;

}

drive(): void {

console.log(`Driving a ${this.brand} ${this.model}.`);

}

// Override method

start(): void {

console.log(`Starting the car: ${this.brand} ${this.model}`);

}

}

// 🔧 Testing

const myCar = new Car("Car", "Honda", "Civic");

myCar.start(); // Overridden method from Machine

myCar.move(); // Inherited from Vehicle

myCar.drive(); // Defined in Car

🧠 Key Concepts Shown:

* super() is used to call constructors of the parent class.
* Methods from the base class can be overridden (start()).
* All levels of methods and properties are accessible from the final subclass (Car).

Would you like a diagram or ASCII illustration of this inheritance hierarchy?

You said:

**Q6: Interface -shape, circle**

ChatGPT said:

Here's an implementation of an Interface in TypeScript to demonstrate Shape and Circle:

* Shape Interface: Defines the basic structure for any geometric shape.
* Circle Class: Implements the Shape Interface, representing a circle with properties like radius.

🚀 TypeScript Code

ts

CopyEdit

// Shape Interface

interface Shape {

area(): number;

perimeter(): number;

}

// Circle Class implementing Shape

class Circle implements Shape {

constructor(public radius: number) {}

// Implementing the area method from Shape interface

area(): number {

return Math.PI \* Math.pow(this.radius, 2);

}

// Implementing the perimeter method from Shape interface

perimeter(): number {

return 2 \* Math.PI \* this.radius;

}

}

// Test the Circle class

const myCircle = new Circle(5);

console.log(`Area of Circle: ${myCircle.area()}`); // Area of Circle: 78.53981633974483

console.log(`Perimeter of Circle: ${myCircle.perimeter()}`); // Perimeter of Circle: 31.41592653589793

🧠 Key Concepts:

1. Interface (Shape):
   * Specifies the structure that any shape should have: methods like area() and perimeter().
2. Class (Circle):
   * Implements the Shape interface and provides specific logic for the area and perimeter methods.

The Circle class defines its specific methods for calculating area and perimeter based on the radius.

Would you like additional shapes added (like Rectangle, Square, etc.) or help with extending this design?

You said:

**Q7: Class called Bank account and method deposit and withdrawal and cannot withdraw if balance is less than 100 in typescript**

ChatGPT said:

Here’s how you can implement a BankAccount class in TypeScript with deposit and withdrawal methods. The withdrawal method ensures that the balance cannot go below 100.

✅ TypeScript Code

ts

CopyEdit

class BankAccount {

private balance: number;

constructor(initialBalance: number) {

// Ensures initial balance is set and is at least 100

if (initialBalance < 100) {

throw new Error("Initial balance must be at least 100.");

}

this.balance = initialBalance;

}

// Method to deposit money into the account

deposit(amount: number): void {

if (amount <= 0) {

console.log("Deposit amount must be greater than zero.");

return;

}

this.balance += amount;

console.log(`Deposited: $${amount}. New balance: $${this.balance}`);

}

// Method to withdraw money from the account

withdraw(amount: number): void {

if (amount <= 0) {

console.log("Withdrawal amount must be greater than zero.");

return;

}

if (this.balance - amount < 100) {

console.log("Cannot withdraw. Insufficient balance to maintain a minimum balance of $100.");

return;

}

this.balance -= amount;

console.log(`Withdrew: $${amount}. New balance: $${this.balance}`);

}

// Method to check the balance

getBalance(): number {

return this.balance;

}

}

// Test the BankAccount class

const myAccount = new BankAccount(500); // Initial balance $500

myAccount.deposit(200); // Deposit $200

myAccount.withdraw(450); // Withdraw $450

myAccount.withdraw(200); // Attempt to withdraw $200 (fails due to minimum balance requirement)

console.log(`Current Balance: $${myAccount.getBalance()}`); // Print the current balance

🧠 Key Concepts:

1. Private Property (balance):
   * Ensures the account balance is only modified through methods (encapsulation).
2. Deposit Method:
   * Adds money to the account, ensuring it is a positive amount.
3. Withdraw Method:
   * Ensures that withdrawals don’t reduce the balance below $100.
   * If a withdrawal attempt would bring the balance below $100, an error message is shown.

💡 Example Output:

pgsql

CopyEdit

Deposited: $200. New balance: $700

Withdrew: $450. New balance: $250

Cannot withdraw. Insufficient balance to maintain a minimum balance of $100.

Current Balance: $250

Would you like to add more features to this (e.g., transaction history, interest rates)?

You said:

**Q8: typescript with html-temperature converter**

ChatGPT said:

Here’s an example of a Temperature Converter using TypeScript and HTML. The converter will allow the user to convert temperatures between Celsius and Fahrenheit.

📁 Folder Structure:

pgsql

CopyEdit

temperature-converter/

├── index.html

├── style.css

└── script.ts

🧑‍💻 index.html – Basic Structure

html

CopyEdit

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Temperature Converter</title>

<link rel="stylesheet" href="style.css">

</head>

<body>

<h1>Temperature Converter</h1>

<div class="converter">

<label for="tempValue">Enter Temperature:</label>

<input type="number" id="tempValue" placeholder="Enter temperature">

<select id="unitSelect">

<option value="C">Celsius</option>

<option value="F">Fahrenheit</option>

</select>

<button onclick="convertTemperature()">Convert</button>

<p id="result"></p>

</div>

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

</body>

</html>

🎨 style.css – Basic Styling

css

CopyEdit

body {

font-family: Arial, sans-serif;

text-align: center;

margin-top: 50px;

}

h1 {

color: #4CAF50;

}

.converter {

display: inline-block;

padding: 20px;

border: 1px solid #ddd;

border-radius: 10px;

margin-top: 20px;

}

input, select {

margin: 10px;

padding: 10px;

font-size: 16px;

}

button {

padding: 10px 20px;

font-size: 16px;

cursor: pointer;

}

button:hover {

background-color: #4CAF50;

color: white;

}

💻 script.ts – TypeScript Logic for Conversion

ts

CopyEdit

// Function to convert temperature

function convertTemperature(): void {

const tempValue = <HTMLInputElement>document.getElementById("tempValue");

const unitSelect = <HTMLSelectElement>document.getElementById("unitSelect");

const result = document.getElementById("result");

const temperature = parseFloat(tempValue.value);

const unit = unitSelect.value;

if (isNaN(temperature)) {

result.textContent = "Please enter a valid number.";

return;

}

let convertedTemp: number;

let convertedUnit: string;

if (unit === "C") {

// Convert from Celsius to Fahrenheit

convertedTemp = (temperature \* 9 / 5) + 32;

convertedUnit = "Fahrenheit";

} else {

// Convert from Fahrenheit to Celsius

convertedTemp = (temperature - 32) \* 5 / 9;

convertedUnit = "Celsius";

}

result.textContent = `${temperature}° ${unit} is equal to ${convertedTemp.toFixed(2)}° ${convertedUnit}`;

}

🔧 Steps to Set Up:

1. Install TypeScript if you haven't already:

bash

CopyEdit

npm install -g typescript

1. Compile the TypeScript into JavaScript:

bash

CopyEdit

tsc script.ts

1. This will generate a script.js file.
2. Open index.html in the browser to see the Temperature Converter in action.

🧠 Key Concepts:

1. DOM Manipulation in TypeScript (accessing elements with getElementById).
2. Type Casting: document.getElementById returns a generic HTMLElement, so we cast it to the correct type like HTMLInputElement or HTMLSelectElement for type safety.
3. Conversion Logic: The formula to convert from Celsius to Fahrenheit is (C \* 9/5) + 32, and from Fahrenheit to Celsius is (F - 32) \* 5/9.

💡 Example Usage:

* Input: 25 Celsius
* Output: 25° Celsius is equal to 77.00° Fahrenheit

Would you like to add more features, such as conversion history or error handling for invalid input?

You said:

**Q9: Student registration form**

ChatGPT said:

Here's an example of a Student Registration Form using HTML and TypeScript. The form will include fields for Name, Email, and Password, and the Password will be validated to ensure it is exactly 6 characters long using ng-pattern for AngularJS (but for simplicity, we'll use TypeScript validation).

📁 Folder Structure:

pgsql

CopyEdit

student-registration/

├── index.html

├── style.css

└── script.ts

🧑‍💻 index.html – Student Registration Form

html

CopyEdit

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Student Registration</title>

<link rel="stylesheet" href="style.css">

</head>

<body>

<h1>Student Registration Form</h1>

<div class="form-container">

<form id="registrationForm">

<div>

<label for="name">Name:</label>

<input type="text" id="name" placeholder="Enter your name" required>

</div>

<div>

<label for="email">Email:</label>

<input type="email" id="email" placeholder="Enter your email" required>

</div>

<div>

<label for="password">Password (6 characters):</label>

<input type="password" id="password" placeholder="Enter password" required>

</div>

<button type="button" onclick="validateForm()">Submit</button>

</form>

<p id="errorMessage" class="error"></p>

</div>

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

</body>

</html>

🎨 style.css – Basic Styling

css

CopyEdit

body {

font-family: Arial, sans-serif;

text-align: center;

margin-top: 50px;

}

h1 {

color: #4CAF50;

}

.form-container {

display: inline-block;

padding: 20px;

border: 1px solid #ddd;

border-radius: 10px;

margin-top: 20px;

}

input {

margin: 10px;

padding: 10px;

font-size: 16px;

}

button {

padding: 10px 20px;

font-size: 16px;

cursor: pointer;

}

button:hover {

background-color: #4CAF50;

color: white;

}

.error {

color: red;

font-weight: bold;

margin-top: 20px;

}

💻 script.ts – TypeScript Logic for Validation

ts

CopyEdit

// Function to validate the form inputs

function validateForm(): void {

const name = <HTMLInputElement>document.getElementById("name");

const email = <HTMLInputElement>document.getElementById("email");

const password = <HTMLInputElement>document.getElementById("password");

const errorMessage = document.getElementById("errorMessage");

// Clear previous error messages

errorMessage!.textContent = "";

// Validate name (ensure it's not empty)

if (name.value.trim() === "") {

errorMessage!.textContent = "Name is required.";

return;

}

// Validate email (basic email validation)

if (email.value.trim() === "") {

errorMessage!.textContent = "Email is required.";

return;

}

// Validate password (exactly 6 characters)

if (password.value.length !== 6) {

errorMessage!.textContent = "Password must be exactly 6 characters.";

return;

}

// If everything is valid, show success message

errorMessage!.textContent = "Registration successful!";

errorMessage!.style.color = "green";

console.log(`Name: ${name.value}, Email: ${email.value}, Password: ${password.value}`);

}

🔧 Steps to Set Up:

1. Install TypeScript if you haven't already:

bash

CopyEdit

npm install -g typescript

1. Compile the TypeScript into JavaScript:

bash

CopyEdit

tsc script.ts

1. This will generate a script.js file.
2. Open index.html in the browser to see the Student Registration Form in action.

🧠 Key Concepts:

1. Form Validation:
   * Ensure that Name and Email are not empty.
   * Ensure that Password is exactly 6 characters.
2. Error Display:
   * Errors are displayed in a <p> element with a class of error.
3. Type Casting:
   * We use TypeScript's HTMLInputElement to safely access form fields and their values.

💡 Example Output:

* Input:
  + Name: John Doe
  + Email: johndoe@example.com
  + Password: abc123
* Output: Registration successful!

Would you like to add more fields to the form or additional features like password strength validation?

You said:

**Q10: calculator in ts without css**

ChatGPT said:

Here's an implementation of a simple Calculator using TypeScript and HTML, without CSS. The calculator will perform basic operations such as addition, subtraction, multiplication, and division.

📁 Folder Structure:

pgsql

CopyEdit

calculator/

├── index.html

└── script.ts

🧑‍💻 index.html – Calculator Structure

html

CopyEdit

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Simple Calculator</title>

</head>

<body>

<h1>Simple Calculator</h1>

<div>

<input type="number" id="num1" placeholder="Enter first number">

<input type="number" id="num2" placeholder="Enter second number">

</div>

<div>

<button onclick="add()">Add</button>

<button onclick="subtract()">Subtract</button>

<button onclick="multiply()">Multiply</button>

<button onclick="divide()">Divide</button>

</div>

<div>

<h2>Result: <span id="result">0</span></h2>

</div>

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

</body>

</html>

💻 script.ts – TypeScript Logic for Calculator

ts

CopyEdit

// Function to add numbers

function add(): void {

const num1 = <HTMLInputElement>document.getElementById("num1");

const num2 = <HTMLInputElement>document.getElementById("num2");

const result = document.getElementById("result");

const number1 = parseFloat(num1.value);

const number2 = parseFloat(num2.value);

if (isNaN(number1) || isNaN(number2)) {

result!.textContent = "Please enter valid numbers.";

return;

}

const sum = number1 + number2;

result!.textContent = sum.toString();

}

// Function to subtract numbers

function subtract(): void {

const num1 = <HTMLInputElement>document.getElementById("num1");

const num2 = <HTMLInputElement>document.getElementById("num2");

const result = document.getElementById("result");

const number1 = parseFloat(num1.value);

const number2 = parseFloat(num2.value);

if (isNaN(number1) || isNaN(number2)) {

result!.textContent = "Please enter valid numbers.";

return;

}

const difference = number1 - number2;

result!.textContent = difference.toString();

}

// Function to multiply numbers

function multiply(): void {

const num1 = <HTMLInputElement>document.getElementById("num1");

const num2 = <HTMLInputElement>document.getElementById("num2");

const result = document.getElementById("result");

const number1 = parseFloat(num1.value);

const number2 = parseFloat(num2.value);

if (isNaN(number1) || isNaN(number2)) {

result!.textContent = "Please enter valid numbers.";

return;

}

const product = number1 \* number2;

result!.textContent = product.toString();

}

// Function to divide numbers

function divide(): void {

const num1 = <HTMLInputElement>document.getElementById("num1");

const num2 = <HTMLInputElement>document.getElementById("num2");

const result = document.getElementById("result");

const number1 = parseFloat(num1.value);

const number2 = parseFloat(num2.value);

if (isNaN(number1) || isNaN(number2)) {

result!.textContent = "Please enter valid numbers.";

return;

}

if (number2 === 0) {

result!.textContent = "Cannot divide by zero.";

return;

}

const quotient = number1 / number2;

result!.textContent = quotient.toString();

}

🔧 Steps to Set Up:

1. Install TypeScript if you haven't already:

bash

CopyEdit

npm install -g typescript

1. Compile TypeScript:

bash

CopyEdit

tsc script.ts

1. This will generate a script.js file.
2. Open index.html in the browser to use the calculator.

🧠 Key Concepts:

1. HTML Input Elements:
   * Two <input> fields are used for entering numbers.
2. TypeScript Logic:
   * Four functions (add(), subtract(), multiply(), divide()) handle the basic arithmetic operations.
   * Validation: It checks whether the input values are valid numbers and ensures division by zero is prevented.
3. DOM Manipulation:
   * document.getElementById is used to access and modify the elements on the webpage.

💡 Example Usage:

1. Input:
   * Number 1: 10
   * Number 2: 5
2. Output:
   * Add: 15
   * Subtract: 5
   * Multiply: 50
   * Divide: 2

Would you like to add more features, such as additional mathematical operations or a reset button?

You said:

**Q11. Any 4 angular js services**

ChatGPT said:

In AngularJS, services are objects or functions that are used to share data and logic across different parts of an application. Here are four common AngularJS services:

1. $http Service

* Purpose: This service is used to make HTTP requests to the server to retrieve or send data.
* Example:

js

CopyEdit

angular.module('myApp', [])

.controller('myController', function($http) {

$http.get('https://api.example.com/data')

.then(function(response) {

console.log(response.data);

}, function(error) {

console.error(error);

});

});

2. $scope Service

* Purpose: The $scope service is used to bind data between the controller and the view. It allows you to define variables and functions within the controller and access them in the HTML template.
* Example:

js

CopyEdit

angular.module('myApp', [])

.controller('myController', function($scope) {

$scope.message = "Hello, AngularJS!";

});

In the HTML template:

html

CopyEdit

<div ng-controller="myController">

<p>{{ message }}</p>

</div>

3. $route Service

* Purpose: The $route service is part of AngularJS's routing module. It is used for configuring routes and handling navigation between views in a single-page application (SPA).
* Example:

js

CopyEdit

angular.module('myApp', ['ngRoute'])

.config(function($routeProvider) {

$routeProvider

.when('/home', {

templateUrl: 'home.html',

controller: 'HomeController'

})

.when('/about', {

templateUrl: 'about.html',

controller: 'AboutController'

})

.otherwise({

redirectTo: '/home'

});

});

4. $timeout Service

* Purpose: The $timeout service is used to delay the execution of a function or expression. It works like JavaScript’s setTimeout() method.
* Example:

js

CopyEdit

angular.module('myApp', [])

.controller('myController', function($scope, $timeout) {

$scope.message = "This message will change in 3 seconds.";

$timeout(function() {

$scope.message = "The message has been updated!";

}, 3000);

});

In this example, after 3 seconds, the message on the page will change.

These services help in different aspects of AngularJS development, such as making HTTP requests, managing routing, delaying actions, and interacting with the scope for data binding.

You said:

**Q12; angular js filters- Search box implement**

ChatGPT said:

In AngularJS, filters are used to format the data before displaying it on the view. To implement a search box using AngularJS filters, we can use the built-in filter filter. This allows us to filter the displayed data based on user input.

Below is an example of implementing a search box with AngularJS, where users can search for items in a list using the filter functionality.

📁 Folder Structure:

pgsql

CopyEdit

angular-search-box/

├── index.html

└── script.js

🧑‍💻 index.html – Search Box Implementation

html

CopyEdit

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>AngularJS Search Box</title>

</head>

<body>

<h1>Search Items</h1>

<div ng-app="myApp" ng-controller="myController">

<!-- Search Input Box -->

<input type="text" ng-model="searchText" placeholder="Search for an item..." />

<!-- Displaying List with Filter -->

<ul>

<li ng-repeat="item in items | filter:searchText">

{{ item }}

</li>

</ul>

</div>

<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js"></script>

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

</body>

</html>

💻 script.js – AngularJS Controller for Search

js

CopyEdit

angular.module('myApp', [])

.controller('myController', function($scope) {

// Sample list of items

$scope.items = [

'Apple',

'Banana',

'Orange',

'Grapes',

'Pineapple',

'Mango',

'Strawberry'

];

// ng-model binds to the search text

$scope.searchText = '';

});

🔧 Steps to Set Up:

1. Install AngularJS: You can link to AngularJS via the CDN in the <script> tag in the index.html, as shown in the example. No installation is required if you use the CDN.
2. Run the App:
   * Open index.html in your browser, and you should see a search box that filters the list of fruits.

🧠 Key Concepts:

* ng-model: The ng-model directive binds the value of the input field to a property in the scope (in this case, searchText).
* ng-repeat: The ng-repeat directive is used to loop over the items array and display them.
* filter filter: The built-in filter filter is used to filter the displayed list based on the searchText entered by the user. It automatically updates as the user types.

💡 Example Usage:

1. Search for: "Man"
   * Output:
     + Mango
2. Search for: "Ap"
   * Output:
     + Apple

The search box automatically filters the list of fruits as the user types.

This simple search functionality can be expanded by using more complex filtering logic or searching within objects (e.g., objects with multiple properties). Would you like to add any other features, like case-insensitive filtering or search highlighting?

You said:

**Q13: create single page application of online shopping cart using angular js where the cart value should update when user clicks on the product**

ChatGPT said:

To create a Single Page Application (SPA) for an online shopping cart using AngularJS, where the cart value updates when the user clicks on a product, you can follow this approach:

🛒 Steps:

1. Display a list of products.
2. Allow the user to add items to the cart.
3. Update the total cart value whenever a product is added.

📁 Folder Structure:

pgsql

CopyEdit

shopping-cart/

├── index.html

└── script.js

🧑‍💻 index.html – Basic HTML Structure for the Shopping Cart

html

CopyEdit

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Online Shopping Cart</title>

</head>

<body>

<h1>Online Shopping Cart</h1>

<div ng-app="myApp" ng-controller="shoppingCartController">

<!-- List of Products -->

<div>

<h3>Products</h3>

<ul>

<li ng-repeat="product in products">

<p>{{ product.name }} - ${{ product.price }}</p>

<button ng-click="addToCart(product)">Add to Cart</button>

</li>

</ul>

</div>

<!-- Cart Section -->

<div>

<h3>Your Cart</h3>

<ul>

<li ng-repeat="item in cart">

<p>{{ item.name }} - ${{ item.price }} x {{ item.quantity }}</p>

</li>

</ul>

<h3>Total: ${{ total }}</h3>

</div>

</div>

<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js"></script>

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

</body>

</html>

💻 script.js – AngularJS Logic for Shopping Cart

js

CopyEdit

angular.module('myApp', [])

.controller('shoppingCartController', function($scope) {

// List of products available in the store

$scope.products = [

{ name: 'Product 1', price: 20 },

{ name: 'Product 2', price: 50 },

{ name: 'Product 3', price: 100 },

{ name: 'Product 4', price: 30 }

];

// The cart array that stores the products added to the cart

$scope.cart = [];

// Variable to store the total value of the cart

$scope.total = 0;

// Function to add products to the cart

$scope.addToCart = function(product) {

// Check if product is already in the cart

let found = false;

for (let i = 0; i < $scope.cart.length; i++) {

if ($scope.cart[i].name === product.name) {

$scope.cart[i].quantity += 1; // Increase quantity

found = true;

break;

}

}

// If product is not in the cart, add it with quantity 1

if (!found) {

$scope.cart.push({ name: product.name, price: product.price, quantity: 1 });

}

// Recalculate the total value of the cart

$scope.calculateTotal();

};

// Function to calculate the total value of the cart

$scope.calculateTotal = function() {

$scope.total = 0;

for (let i = 0; i < $scope.cart.length; i++) {

$scope.total += $scope.cart[i].price \* $scope.cart[i].quantity;

}

};

});

🔧 Steps to Run the Application:

1. Install AngularJS: You don't need to install anything locally if you use the CDN link for AngularJS, as shown in the <script> tag of the index.html.
2. Run the Application: Simply open index.html in a browser, and the shopping cart should work.

🧠 Key Concepts:

1. ng-repeat: This directive is used to loop through the list of products and cart items.
2. ng-click: This directive is used to bind the addToCart function to the "Add to Cart" button.
3. Model Binding: AngularJS handles the binding of the cart and total price automatically when you modify them.
4. Cart Updates: Every time a product is added to the cart, the addToCart function checks if the product is already in the cart. If it is, it updates the quantity; otherwise, it adds a new item to the cart.

💡 Example Usage:

1. Products List:
   * Product 1 - $20
   * Product 2 - $50
   * Product 3 - $100
   * Product 4 - $30
2. Add Products:
   * Click on "Add to Cart" for Product 1 (Price: $20).
   * The cart will show: Product 1 - $20 x 1 and the total will be $20.
   * Click on "Add to Cart" for Product 2 (Price: $50).
   * The cart will show: Product 1 - $20 x 1 and Product 2 - $50 x 1 with the total of $70.

💡 Extra Features (optional):

* Remove Item from Cart: You could add a "Remove" button next to each item in the cart to allow users to remove items from the cart.
* Cart Empty Check: You could display a message when the cart is empty (e.g., "Your cart is empty").

Would you like to add any other features to this application?

You said:

**Q14: create a task based SPA(task manager ) by using angular directives and services**

ChatGPT said:

To create a Task-based SPA (Single Page Application) for a Task Manager using AngularJS directives and services, the application will allow users to add tasks, mark them as completed, and delete them. It will use AngularJS features such as directives, services, and two-way data binding.

📁 Folder Structure:

pgsql

CopyEdit

task-manager/

├── index.html

└── script.js

🧑‍💻 index.html – Task Manager HTML Layout

html

CopyEdit

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Task Manager</title>

</head>

<body>

<div ng-app="taskApp" ng-controller="taskController">

<h1>Task Manager</h1>

<!-- Task Input Section -->

<div>

<input type="text" ng-model="taskText" placeholder="Enter a task" />

<button ng-click="addTask()">Add Task</button>

</div>

<!-- Task List Section -->

<div>

<h3>Tasks</h3>

<ul>

<li ng-repeat="task in tasks">

<span ng-class="{'completed': task.completed}">{{ task.text }}</span>

<button ng-click="toggleCompletion(task)">Mark as {{ task.completed ? 'Incomplete' : 'Completed' }}</button>

<button ng-click="removeTask(task)">Delete</button>

</li>

</ul>

</div>

</div>

<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js"></script>

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

</body>

</html>

💻 script.js – Task Manager Logic using AngularJS Directives and Services

js

CopyEdit

angular.module('taskApp', [])

.controller('taskController', function($scope, taskService) {

// Initializing the task array by fetching data from taskService

$scope.tasks = taskService.getTasks();

// Function to add a new task

$scope.addTask = function() {

if ($scope.taskText) {

const newTask = { text: $scope.taskText, completed: false };

taskService.addTask(newTask); // Add the task to service

$scope.taskText = ''; // Clear the input field

$scope.tasks = taskService.getTasks(); // Update task list

}

};

// Function to toggle task completion status

$scope.toggleCompletion = function(task) {

taskService.toggleTaskCompletion(task);

$scope.tasks = taskService.getTasks(); // Update task list after toggling completion

};

// Function to remove a task

$scope.removeTask = function(task) {

taskService.removeTask(task);

$scope.tasks = taskService.getTasks(); // Update task list after removal

};

})

.service('taskService', function() {

// Initial tasks (In a real-world app, this could be fetched from a database)

let tasks = [];

// Function to get all tasks

this.getTasks = function() {

return tasks;

};

// Function to add a new task

this.addTask = function(task) {

tasks.push(task);

};

// Function to toggle task completion status

this.toggleTaskCompletion = function(task) {

task.completed = !task.completed;

};

// Function to remove a task

this.removeTask = function(task) {

const index = tasks.indexOf(task);

if (index !== -1) {

tasks.splice(index, 1);

}

};

});

🔧 Steps to Run the Application:

1. Install AngularJS: You can link to AngularJS via the CDN, which is already included in the <script> tag in the index.html file.
2. Run the Application: Open index.html in any browser, and you should see a simple task manager with functionalities for adding tasks, marking them as completed/incomplete, and deleting them.

🧠 Key Concepts:

1. ng-repeat: Loops through the tasks array and displays each task.
2. ng-model: Binds the task input field to the taskText property in the scope.
3. ng-click: Triggers AngularJS expressions such as adding, removing, or toggling task completion when the respective button is clicked.
4. ng-class: Dynamically applies a CSS class (completed) to a task when it's marked as completed.

🖥️ Features:

* Add Task: You can add a task by typing into the input field and clicking the "Add Task" button.
* Toggle Completion: You can toggle whether a task is completed or not by clicking "Mark as Completed" or "Mark as Incomplete."
* Remove Task: You can remove a task by clicking the "Delete" button next to it.
* Two-Way Data Binding: The input field for adding tasks is bound to the taskText property of the scope, and changes are reflected immediately.

💡 Extra Features (optional):

* Persistent Data: In a real-world application, you could persist the task list to a backend (using AJAX or a RESTful API) or local storage so the tasks remain even after a page refresh.
* Completion Filter: Add a filter to display only completed tasks or incomplete tasks.
* Edit Task: Implement an "Edit" functionality to modify existing tasks.
* Task Prioritization: Allow tasks to have priorities (High, Medium, Low).

Would you like to add more features, or need further customization to the task manager?

You said:

**Q15: Create a task manager using angular-UI logic**

ChatGPT said:

To create a Task Manager using AngularJS and Angular UI (for better UI elements), we can leverage Angular UI Bootstrap components like modal dialogs, alerts, and other UI elements. The goal is to build a task manager where users can add, remove, edit, and complete tasks with a clean UI.

📁 Folder Structure:

pgsql

CopyEdit

task-manager/

├── index.html

└── script.js

🧑‍💻 index.html – HTML Layout for the Task Manager using Angular UI

html

CopyEdit

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Task Manager</title>

<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">

<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/angular-ui-bootstrap/2.5.6/ui-bootstrap-csp.min.css">

</head>

<body>

<div ng-app="taskApp" ng-controller="taskController" class="container">

<h1>Task Manager</h1>

<!-- Task Input Section -->

<div class="form-group">

<input type="text" ng-model="taskText" class="form-control" placeholder="Enter a new task">

<button ng-click="addTask()" class="btn btn-primary btn-block">Add Task</button>

</div>

<!-- Task List Section -->

<h3>Tasks</h3>

<ul class="list-group">

<li ng-repeat="task in tasks" class="list-group-item">

<span ng-class="{'completed': task.completed}">{{ task.text }}</span>

<button ng-click="toggleCompletion(task)" class="btn btn-warning btn-sm pull-right">Toggle</button>

<button ng-click="removeTask(task)" class="btn btn-danger btn-sm pull-right">Delete</button>

</li>

</ul>

<!-- Modal for Editing Task -->

<script type="text/ng-template" id="editTaskModal.html">

<div class="modal-header">

<h3 class="modal-title">Edit Task</h3>

</div>

<div class="modal-body">

<input type="text" ng-model="taskToEdit.text" class="form-control" placeholder="Edit task">

</div>

<div class="modal-footer">

<button class="btn btn-default" ng-click="cancel()">Cancel</button>

<button class="btn btn-primary" ng-click="save()">Save</button>

</div>

</script>

</div>

<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js"></script>

<script src="https://cdnjs.cloudflare.com/ajax/libs/angular-ui-bootstrap/2.5.6/ui-bootstrap-tpls.min.js"></script>

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

</body>

</html>

💻 script.js – AngularJS Logic using Angular UI

js

CopyEdit

angular.module('taskApp', ['ui.bootstrap'])

.controller('taskController', function($scope, $uibModal) {

// Initial task list

$scope.tasks = [];

// Function to add a new task

$scope.addTask = function() {

if ($scope.taskText) {

const newTask = { text: $scope.taskText, completed: false };

$scope.tasks.push(newTask);

$scope.taskText = ''; // Clear the input field

}

};

// Function to toggle task completion

$scope.toggleCompletion = function(task) {

task.completed = !task.completed;

};

// Function to remove a task

$scope.removeTask = function(task) {

const index = $scope.tasks.indexOf(task);

if (index !== -1) {

$scope.tasks.splice(index, 1);

}

};

// Function to open modal for editing a task

$scope.editTask = function(task) {

const modalInstance = $uibModal.open({

templateUrl: 'editTaskModal.html',

controller: 'editTaskModalController',

resolve: {

task: function() {

return task;

}

}

});

modalInstance.result.then(function(updatedTask) {

// Update task with the new value from modal

task.text = updatedTask.text;

});

};

})

.controller('editTaskModalController', function($scope, $uibModalInstance, task) {

// Initial task to edit

$scope.taskToEdit = angular.copy(task); // Copy task to avoid direct binding

// Function to save changes to the task

$scope.save = function() {

$uibModalInstance.close($scope.taskToEdit);

};

// Function to cancel editing

$scope.cancel = function() {

$uibModalInstance.dismiss('cancel');

};

});

🔧 Steps to Run the Application:

1. Install AngularJS and Angular UI Bootstrap: Include the necessary CDN links for AngularJS and Angular UI Bootstrap.
   * AngularJS (v1.8.2): https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js
   * UI Bootstrap (v2.5.6): https://cdnjs.cloudflare.com/ajax/libs/angular-ui-bootstrap/2.5.6/ui-bootstrap-tpls.min.js
2. Run the Application: Open index.html in a browser to see the task manager in action.

🧠 Key Concepts:

1. Angular UI Bootstrap: Used to provide modal functionality to allow editing of tasks in a clean and organized way.
   * ng-class: Dynamically applies the completed class to mark completed tasks.
   * ui.bootstrap provides modal dialogs for editing tasks.
2. Two-Way Data Binding: ng-model is used to bind the input field and update the taskText in the controller.
3. Modular Design: The editTaskModalController is responsible for handling the modal's logic, separating concerns between different parts of the app.

💡 Features:

1. Add Tasks: Users can input a task in a text box and click Add Task to add it to the list.
2. Complete/Incomplete Tasks: Each task can be toggled between completed or incomplete status.
3. Delete Tasks: Users can delete tasks from the list.
4. Edit Task: Tasks can be edited using a modal dialog (with the help of Angular UI Bootstrap).

💡 Extra Features (optional):

1. Persistent Data: You can connect this to a backend to persist the tasks, or use local storage to save tasks even after page reloads.
2. Task Filtering: Add functionality to filter tasks by status (e.g., show only "completed" tasks).
3. Task Priority: Add a priority level (High, Medium, Low) to each task and allow users to sort tasks by priority.
4. Task Due Date: Add due dates for tasks and allow users to sort tasks by date.

📋 To Do:

* Modify the tasks array to include more information (e.g., due date, priority).
* Implement additional features such as sorting, due dates, or even using AngularJS services to handle tasks more efficiently.

Would you like to explore more on any of the above features?

You said:

**Q16. Develop a web application that uses AJAX to perform dynamic search functionality from a product catalog**

ChatGPT said:

Here’s a simple example of how to develop a web application using AJAX to perform dynamic search functionality from a product catalog using HTML, JavaScript (AJAX), and a mock JSON catalog.

🔧 Folder Structure

pgsql

CopyEdit

ajax-product-search/

├── index.html

├── script.js

└── products.json

📁 products.json – Sample Product Catalog

json

CopyEdit

[

{ "id": 1, "name": "Apple iPhone 15" },

{ "id": 2, "name": "Samsung Galaxy S24" },

{ "id": 3, "name": "Google Pixel 8" },

{ "id": 4, "name": "OnePlus 12" },

{ "id": 5, "name": "Sony Xperia 5" },

{ "id": 6, "name": "Motorola Edge 40" }

]

🧑‍💻 index.html – Frontend UI

html

CopyEdit

<!DOCTYPE html>

<html>

<head>

<title>Product Catalog Search</title>

<style>

body { font-family: Arial, sans-serif; padding: 20px; }

#results { margin-top: 10px; }

.product { margin: 5px 0; }

</style>

</head>

<body>

<h2>Search Product Catalog</h2>

<input type="text" id="searchBox" placeholder="Type to search..." onkeyup="searchProduct()" />

<div id="results"></div>

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

</body>

</html>

📜 script.js – AJAX Logic

javascript

CopyEdit

function searchProduct() {

const query = document.getElementById("searchBox").value.toLowerCase();

const xhr = new XMLHttpRequest();

xhr.open("GET", "products.json", true);

xhr.onload = function () {

if (this.status === 200) {

const products = JSON.parse(this.responseText);

const filtered = products.filter(p =>

p.name.toLowerCase().includes(query)

);

let output = "";

if (filtered.length > 0) {

filtered.forEach(p => {

output += `<div class="product">${p.name}</div>`;

});

} else {

output = "<div>No products found.</div>";

}

document.getElementById("results").innerHTML = output;

}

};

xhr.send();

}

✅ How It Works

1. User types in the input field.
2. The onkeyup event triggers the searchProduct() function.
3. The function sends an AJAX GET request to products.json.
4. Products are filtered client-side based on user input.
5. Matching results are dynamically shown without page reload.

🧪 To Run:

1. Serve the files via a local server (due to AJAX CORS/security policies).
   * Example: use Live Server in VS Code or run npx http-server in the folder.
2. Open index.html in your browser.
3. Start typing to see real-time filtered results from the product catalog.

Would you like a backend version using Flask or Node.js with a real-time DB (like MongoDB)?

You said:

**Q17: angularjs registration form and 6 angular services**

ChatGPT said:

Here's a complete example of an AngularJS Registration Form with 6 commonly used AngularJS services:

✅ Features:

* Form with ng-model and ng-submit
* Password validation using ng-pattern
* Use of 6 AngularJS services:
  1. $http – for mock submission
  2. $timeout – for delayed success message
  3. $log – for logging form data
  4. $location – for potential redirection
  5. $window – for alerts
  6. $filter – for name capitalization

📁 index.html

html

CopyEdit

<!DOCTYPE html>

<html ng-app="regApp">

<head>

<meta charset="utf-8">

<title>AngularJS Registration Form</title>

<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js"></script>

<style>

body { font-family: Arial; margin: 20px; }

input, button { padding: 8px; margin: 5px; }

.error { color: red; }

</style>

</head>

<body ng-controller="RegController">

<h2>Student Registration Form</h2>

<form name="regForm" ng-submit="submitForm()" novalidate>

<label>Name:</label><br>

<input type="text" ng-model="user.name" required /><br>

<label>Email:</label><br>

<input type="email" ng-model="user.email" required /><br>

<label>Password (6 digits):</label><br>

<input type="password" ng-model="user.password" ng-pattern="/^\d{6}$/" required />

<span class="error" ng-show="regForm.password.$error.pattern">Must be exactly 6 digits</span><br>

<button type="submit" ng-disabled="regForm.$invalid">Register</button>

</form>

<div ng-if="submitted">

<h4>Registration Successful!</h4>

<p><strong>Name:</strong> {{ user.name | uppercase }}</p>

<p><strong>Email:</strong> {{ user.email }}</p>

</div>

<script>

const app = angular.module("regApp", []);

app.controller("RegController", function($scope, $http, $timeout, $log, $location, $window, $filter) {

$scope.user = {};

$scope.submitted = false;

$scope.submitForm = function() {

const filteredName = $filter('uppercase')($scope.user.name);

$log.info("Submitting registration for:", filteredName);

// Mock HTTP request

$http.post("/register", $scope.user)

.then(function(response) {

$timeout(function() {

$scope.submitted = true;

$window.alert("Registration submitted!");

// $location.path('/success'); // Uncomment to simulate route change

}, 1000);

}, function(error) {

$log.error("Error submitting form:", error);

});

};

});

</script>

</body>

</html>

🔍 AngularJS Services Used:

| Service | Use Case |
| --- | --- |
| $http | Simulates sending form data to a server (mocked here) |
| $timeout | Delays success message by 1 second |
| $log | Logs the form data and errors to console |
| $location | Can be used to change the route after submission |
| $window | Shows a browser alert upon successful registration |
| $filter | Formats name to uppercase (you can use other filters like currency, etc) |

Would you like me to convert this into a modular AngularJS app or integrate with a backend (Node/Mongo)?

You said:

**Q18: Create a front end in (React.js/any JS of your choice including HTML CSS) Create a server page in Node.js to store the data in MongoDB back end**

ChatGPT said:

Here's a complete working setup for your full-stack web app with:

* Frontend: HTML + JavaScript
* Backend: Node.js + Express
* Database: MongoDB

🎯 Use Case:

Let's assume we are storing student result data (Class, PRN number, Name, Email, Subject-wise Marks).

🖼️ Frontend (HTML + JS)

index.html

html

CopyEdit

<!DOCTYPE html>

<html>

<head>

<title>Student Result Submission</title>

<style>

body { font-family: Arial; margin: 20px; }

input, textarea { display: block; margin: 10px 0; padding: 8px; width: 300px; }

button { padding: 10px 20px; }

</style>

</head>

<body>

<h2>Submit Student Result</h2>

<form id="resultForm">

<input type="text" id="class" placeholder="Class" required />

<input type="text" id="prn" placeholder="PRN Number" required />

<input type="text" id="name" placeholder="Name" required />

<input type="email" id="email" placeholder="Email" required />

<textarea id="marks" placeholder='Marks (e.g. {"Math":90,"CS":85})' required></textarea>

<button type="submit">Submit</button>

</form>

<script>

document.getElementById("resultForm").addEventListener("submit", async function(e) {

e.preventDefault();

const data = {

class: document.getElementById("class").value,

prn: document.getElementById("prn").value,

name: document.getElementById("name").value,

email: document.getElementById("email").value,

marks: JSON.parse(document.getElementById("marks").value)

};

const response = await fetch("http://localhost:3000/api/results", {

method: "POST",

headers: { "Content-Type": "application/json" },

body: JSON.stringify(data)

});

const result = await response.json();

alert(result.message || "Submitted!");

});

</script>

</body>

</html>

🖥️ Backend (Node.js + Express + MongoDB)

1. Install dependencies

bash

CopyEdit

npm init -y

npm install express mongoose cors

2. server.js

js

CopyEdit

const express = require("express");

const mongoose = require("mongoose");

const cors = require("cors");

const app = express();

const PORT = 3000;

// Connect to MongoDB

mongoose.connect("mongodb://localhost:27017/studentResults", {

useNewUrlParser: true,

useUnifiedTopology: true

})

.then(() => console.log("MongoDB connected"))

.catch(err => console.error("MongoDB error:", err));

// Middleware

app.use(cors());

app.use(express.json());

// Mongoose schema

const studentSchema = new mongoose.Schema({

class: String,

prn: String,

name: String,

email: String,

marks: Object

});

const Student = mongoose.model("Student", studentSchema);

// API endpoint

app.post("/api/results", async (req, res) => {

try {

const student = new Student(req.body);

await student.save();

res.json({ message: "Result saved successfully" });

} catch (error) {

res.status(500).json({ error: "Failed to save result" });

}

});

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

✅ How to Run

1. Start MongoDB (mongod)
2. Run Node.js server:

bash

CopyEdit

node server.js

1. Open index.html in your browser (via Live Server or localhost)
2. Submit the form → data gets stored in MongoDB studentResults database.

Would you like a React.js frontend version of this instead of plain HTML?

You said:

**Q19: Create MongoDB Database including result data (Class, PRN number,Name,Email, subject wise marks) Use CRUD operation to manipulate data**

ChatGPT said:

Here is a complete implementation of a Node.js + Express + MongoDB app that includes:

* A MongoDB database for student result data: Class, PRN number, Name, Email, and subject-wise marks.
* Full CRUD operations:
  + Create (POST)
  + Read (GET)
  + Update (PUT)
  + Delete (DELETE)

📁 Step 1: Setup Project

1. Install Dependencies:

bash

CopyEdit

npm init -y

npm install express mongoose cors

📄 Step 2: Backend Code (server.js)

js

CopyEdit

const express = require("express");

const mongoose = require("mongoose");

const cors = require("cors");

const app = express();

const PORT = 3000;

// Middleware

app.use(cors());

app.use(express.json());

// MongoDB connection

mongoose.connect("mongodb://localhost:27017/studentResults", {

useNewUrlParser: true,

useUnifiedTopology: true

})

.then(() => console.log("MongoDB connected"))

.catch(err => console.error("MongoDB error:", err));

// Schema

const studentSchema = new mongoose.Schema({

class: String,

prn: String,

name: String,

email: String,

marks: Object

});

const Student = mongoose.model("Student", studentSchema);

// Create (POST)

app.post("/students", async (req, res) => {

try {

const student = new Student(req.body);

await student.save();

res.status(201).json({ message: "Student added", student });

} catch (error) {

res.status(500).json({ error: "Error adding student" });

}

});

// Read all (GET)

app.get("/students", async (req, res) => {

try {

const students = await Student.find();

res.json(students);

} catch (error) {

res.status(500).json({ error: "Error fetching students" });

}

});

// Read one by PRN (GET)

app.get("/students/:prn", async (req, res) => {

try {

const student = await Student.findOne({ prn: req.params.prn });

if (student) res.json(student);

else res.status(404).json({ error: "Student not found" });

} catch (error) {

res.status(500).json({ error: "Error fetching student" });

}

});

// Update (PUT)

app.put("/students/:prn", async (req, res) => {

try {

const updated = await Student.findOneAndUpdate({ prn: req.params.prn }, req.body, { new: true });

if (updated) res.json({ message: "Student updated", updated });

else res.status(404).json({ error: "Student not found" });

} catch (error) {

res.status(500).json({ error: "Error updating student" });

}

});

// Delete (DELETE)

app.delete("/students/:prn", async (req, res) => {

try {

const deleted = await Student.findOneAndDelete({ prn: req.params.prn });

if (deleted) res.json({ message: "Student deleted" });

else res.status(404).json({ error: "Student not found" });

} catch (error) {

res.status(500).json({ error: "Error deleting student" });

}

});

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

📋 Sample POST Request (via Postman or JS Fetch):

json

CopyEdit

{

"class": "TE Computer",

"prn": "1234567890",

"name": "Rahul Sharma",

"email": "rahul@example.com",

"marks": {

"Maths": 85,

"DBMS": 78,

"AI": 90

}

}

✅ CRUD Endpoints Summary:

| Operation | Method | Endpoint | Description |
| --- | --- | --- | --- |
| Create | POST | /students | Add new student result |
| Read all | GET | /students | Fetch all students |
| Read one | GET | /students/:prn | Fetch student by PRN |
| Update | PUT | /students/:prn | Update student details |
| Delete | DELETE | /students/:prn | Remove student record |

Would you like a frontend interface (React/HTML) for interacting with this backend too?

You said:

**Q20: Create a RESTful API using MongoDB for a web-based appointment booking system. Implement the following features: 1. A view button to display the list of doctors. 2. Functionality to book and cancel appointments**

ChatGPT said:

Here's a full working example of a RESTful API using Node.js, Express, and MongoDB for a web-based appointment booking system with:

✅ Features:

1. View all doctors (GET /doctors)
2. Book appointment (POST /appointments)
3. Cancel appointment (DELETE /appointments/:id)

📦 Step 1: Install Dependencies

bash

CopyEdit

npm init -y

npm install express mongoose cors

🧠 Step 2: Project Structure

pgsql

CopyEdit

appointment-system/

├── server.js

└── models/

├── Doctor.js

└── Appointment.js

📄 models/Doctor.js

js

CopyEdit

const mongoose = require("mongoose");

const doctorSchema = new mongoose.Schema({

name: String,

specialization: String,

availableSlots: [String], // e.g. ["10:00AM", "11:00AM"]

});

module.exports = mongoose.model("Doctor", doctorSchema);

📄 models/Appointment.js

js

CopyEdit

const mongoose = require("mongoose");

const appointmentSchema = new mongoose.Schema({

doctorId: { type: mongoose.Schema.Types.ObjectId, ref: "Doctor" },

patientName: String,

timeSlot: String,

createdAt: { type: Date, default: Date.now },

});

module.exports = mongoose.model("Appointment", appointmentSchema);

🚀 server.js

js

CopyEdit

const express = require("express");

const mongoose = require("mongoose");

const cors = require("cors");

const Doctor = require("./models/Doctor");

const Appointment = require("./models/Appointment");

const app = express();

app.use(cors());

app.use(express.json());

const PORT = 3000;

// MongoDB Connection

mongoose.connect("mongodb://localhost:27017/appointments", {

useNewUrlParser: true,

useUnifiedTopology: true

})

.then(() => console.log("MongoDB connected"))

.catch(err => console.error("MongoDB error:", err));

// Seed a few doctors (only once)

app.get("/seed", async (req, res) => {

const doctors = [

{ name: "Dr. Sharma", specialization: "Cardiologist", availableSlots: ["10:00AM", "11:00AM"] },

{ name: "Dr. Mehta", specialization: "Dentist", availableSlots: ["12:00PM", "1:00PM"] }

];

await Doctor.insertMany(doctors);

res.send("Doctors seeded");

});

// 1. View list of doctors

app.get("/doctors", async (req, res) => {

const doctors = await Doctor.find();

res.json(doctors);

});

// 2. Book an appointment

app.post("/appointments", async (req, res) => {

const { doctorId, patientName, timeSlot } = req.body;

// Check if the slot is available

const doctor = await Doctor.findById(doctorId);

if (!doctor.availableSlots.includes(timeSlot)) {

return res.status(400).json({ error: "Slot not available" });

}

// Book the appointment

const appointment = new Appointment({ doctorId, patientName, timeSlot });

await appointment.save();

// Remove slot from availability

doctor.availableSlots = doctor.availableSlots.filter(slot => slot !== timeSlot);

await doctor.save();

res.json({ message: "Appointment booked", appointment });

});

// 3. Cancel an appointment

app.delete("/appointments/:id", async (req, res) => {

const appointment = await Appointment.findByIdAndDelete(req.params.id);

if (!appointment) {

return res.status(404).json({ error: "Appointment not found" });

}

// Re-add the slot to the doctor

const doctor = await Doctor.findById(appointment.doctorId);

doctor.availableSlots.push(appointment.timeSlot);

await doctor.save();

res.json({ message: "Appointment cancelled" });

});

// Start server

app.listen(PORT, () => {

console.log(`Server running at http://localhost:${PORT}`);

});

🔍 Example API Usage

1. View Doctors

GET http://localhost:3000/doctors

2. Book Appointment

json

CopyEdit

POST http://localhost:3000/appointments

{

"doctorId": "<ObjectId>",

"patientName": "Alice",

"timeSlot": "10:00AM"

}

3. Cancel Appointment

DELETE http://localhost:3000/appointments/<appointment\_id>