FIRST MONTH FOUNDATION CURRICULUM

Structure:

- **Week 1:** Programming Fundamentals
- **Week 2:** Object-Oriented Programming (OOP)
- **Week 3:** Databases
- **Week 4:** Practical Application

Here's a more detailed, **teaching-ready expansion** of your Week 1 TypeScript fundamentals course. I'll keep it practical, beginner-friendly, and aligned with how students actually learn.

WEEK 1 — Programming Fundamentals (TypeScript)

Goal

By the end of this week, students will:

- Understand how programs execute step by step
- Write basic TypeScript programs confidently
- Build logical thinking (problem-solving mindset)
- Create a small real-world mini project

Day 1 — Introduction + Variables + Data Types

1. What is Programming?

Programming is giving **instructions to a computer**.

Real-life analogy:

- Recipe = Program
- Chef = Computer
- Ingredients = Data

Example:

console.log("Hello World");

This tells the computer: **print "Hello World"**

---

2. Variables (Storing Data)

Variables are **containers** for storing values.

let name: string = "Ali";

let age: number = 20;

Rules:

- Use meaningful names
- Cannot start with numbers
- Use camelCase (userName)

---

3. Data Types (Core Types)

1. String (Text)

let city: string = "Lahore";

2. Number

let price: number = 100;

3. Boolean (true/false)

let isStudent: boolean = true;

String Interpolation

console.log(`My name is ${name} and I am ${age} years old`);

More Examples

let country: string = "Pakistan";

let isMarried: boolean = false;

```
console.log(`${name} lives in ${country}`);
```

Exercises (Expanded)

1. Print:
    - Name
    - Age
    - City
    - Profession
2. Create variables for:
    - Favorite food
    - Favorite color
    - Print them in one sentence

Day 2 — Operators + Conditions

1. Arithmetic Operators

```
let a = 10;
let b = 5;

console.log(a + b); // 15
console.log(a - b); // 5
console.log(a * b); // 50
console.log(a / b); // 2
console.log(a % b); // 0
```

---

2. Comparison Operators

```
console.log(a > b);  // true
console.log(a < b);  // false
```

```
console.log(a == b); // false
```

---

3. Conditions (if/else)

```
let marks = 70;

if (marks >= 50) {
  console.log("Pass");
} else {
  console.log("Fail");
}
```

Real-Life Logic Example

```
let temperature = 35;

if (temperature > 30) {
  console.log("It's hot");
} else {
  console.log("Weather is normal");
}
```

Exercises (Expanded)

1. Calculator

```
let num1 = 10;
let num2 = 5;

console.log("Addition:", num1 + num2);
```

---

2. Even/Odd Checker

```
let num = 7;


if (num % 2 === 0) {
  console.log("Even");
} else {
  console.log("Odd");
}
```

---

3. Grade System

```
let marks = 85;


if (marks >= 80) {
  console.log("A Grade");
} else if (marks >= 60) {
  console.log("B Grade");
} else {
  console.log("Fail");
}
```

Day 3 — Loops + Functions

1. Loops (Repetition)

for loop

```
for (let i = 1; i <= 5; i++) {
  console.log(i);
}
```

---

while loop

```
let i = 1;
while (i <= 5) {
  console.log(i);
  i++;
}
```

---

2. Functions

Functions are reusable blocks of code.

```
function add(a: number, b: number): number {
  return a + b;
}

console.log(add(2, 3)); // 5
```

Why Functions?

- Avoid repetition
- Make code clean
- Easier debugging

Exercises (Expanded)

1. Print 1–10

```
for (let i = 1; i <= 10; i++) {
  console.log(i);
}
```

---

2. Sum of Numbers

```
let sum = 0;

for (let i = 1; i <= 10; i++) {
  sum += i;
}

console.log(sum);
```

---

3. Calculator Functions

```
function multiply(a: number, b: number): number {
  return a * b;
}
```

Day 4 — Arrays + Mini Project

1. Arrays

Arrays store multiple values.

```
let numbers: number[] = [1, 2, 3, 4];
```

Loop Through Array

```
numbers.forEach(num => console.log(num));
```

Access Elements

```
console.log(numbers[0]); // 1
```

More Examples

```
let names: string[] = ["Ali", "Ahmed", "Sara"];
```

```
names.forEach(name => {
  console.log(name);
});
```

Exercise: Find Largest Number

```
let nums = [10, 25, 5, 40];

let max = nums[0];

for (let i = 1; i < nums.length; i++) {
  if (nums[i] > max) {
    max = nums[i];
  }
}

console.log("Largest:", max);
```

Mini Project — Student Result System

Requirements

- Input marks of subjects
- Calculate average
- Show grade/result

Example Implementation

```
let marks: number[] = [80, 70, 90, 60, 85];

let total = 0;
```

```
for (let i = 0; i < marks.length; i++) {
  total += marks[i];
}

let average = total / marks.length;

console.log("Average:", average);

// Grade logic
if (average >= 80) {
  console.log("Grade: A");
} else if (average >= 60) {
  console.log("Grade: B");
} else {
  console.log("Fail");
}
```

Bonus Improvements (For Strong Students)

- Take input dynamically
- Add subjects names
- Show result like:

Math: 80

English: 70

Average: 75

Grade: B

Here's a **fully expanded, teaching-friendly version of Week 2 (OOP in TypeScript)**— structured the same way as Week 1 but deeper, more practical, and focused on real-world thinking.

WEEK 2 — Object-Oriented Programming (OOP)

Goal

By the end of this week, students will:

- Understand how to model real-world systems in code
- Write structured, scalable programs
- Use OOP principles like:
    - Encapsulation
    - Inheritance
    - Polymorphism
- Build real mini projects (Library + Banking system)

Day 1 — Classes + Objects + Methods

1. What is OOP?

OOP = **Organizing code like real-world objects**

Real-world example:

- Student → has name, age
- Car → has color, speed
- Bank Account → has balance, deposit()

---

2. Classes (Blueprint)

A class is a **template** for creating objects.

class Student {

```
  name: string;

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

---

3. Objects (Instances)

```
let s1 = new Student("Ali");
let s2 = new Student("Ahmed");

console.log(s1.name); // Ali
```

---

4. Methods (Functions inside class)

```
class Calculator {
  add(a: number, b: number): number {
    return a + b;
  }

  subtract(a: number, b: number): number {
    return a - b;
  }
}

let calc = new Calculator();
console.log(calc.add(5, 3));
```

Real-World Example

```
class Car {
  brand: string;

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

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

let car1 = new Car("Toyota");
car1.start();
```

Exercises (Expanded)

1. Create a Person class:
    - name, age
    - method: introduce()
2. Create a Rectangle class:
    - width, height
    - method: area()

Day 2 — Encapsulation + Inheritance

---

1. Encapsulation (Data Protection)

Hide internal data using private

```
class Account {
  private balance: number = 0;

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

  getBalance(): number {
    return this.balance;
  }
}
```

Why Encapsulation?

- Prevent misuse
- Secure data
- Control access

Example Usage

```
let acc = new Account();
acc.deposit(1000);

console.log(acc.getBalance());
```

---

2. Inheritance (Reuse Code)

```
class Animal {
  speak() {
```

```
    console.log("Animal makes sound");
  }
}

class Dog extends Animal {
  bark() {
    console.log("Dog barks");
  }
}

let d = new Dog();
d.speak();
d.bark();
```

Real-Life Analogy

- Animal → Parent
- Dog → Child

Exercises

1. Create:
    - Vehicle → method: start()
    - Bike extends Vehicle
2. Create:
    - Employee
    - Manager extends Employee

Day 3 — Polymorphism + Practice

---

1. Polymorphism (Many Forms)

Same method, different behavior

```
class Animal {
  speak() {
    console.log("Animal sound");
  }
}

class Dog extends Animal {
  speak() {
    console.log("Bark");
  }
}

class Cat extends Animal {
  speak() {
    console.log("Meow");
  }
}
```

How it works

Same method speak(), different outputs.

---

2. Shape Example

```
class Shape {
  area(): number {
```

```
    return 0;
  }
}
```

Extend Shape

```
class Rectangle extends Shape {
  width: number;
  height: number;

  constructor(w: number, h: number) {
    super();
    this.width = w;
    this.height = h;
  }

  area(): number {
    return this.width * this.height;
  }
}
```

```
class Circle extends Shape {
  radius: number;

  constructor(r: number) {
    super();
    this.radius = r;
  }
```

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

Exercises

1. Create:
    - Triangle class (area)
2. Override area() in:
    - Square
    - Circle

Day 4 — Projects

Mini Project — Library System

Features

- Add books
- Issue books
- Show available books

Example Implementation

```
class Book {
  title: string;
  isIssued: boolean = false;

  constructor(title: string) {
```

```
    this.title = title;
  }
}

class Library {
  books: Book[] = [];

  addBook(title: string) {
    this.books.push(new Book(title));
  }

  issueBook(title: string) {
    let book = this.books.find(b => b.title === title);

    if (book && !book.isIssued) {
      book.isIssued = true;
      console.log(`${title} issued`);
    } else {
      console.log("Not available");
    }
  }

  showBooks() {
    this.books.forEach(b => {
      console.log(`${b.title} - ${b.isIssued ? "Issued" : "Available"}`);
    });
  }
```

}

Final Project — Banking System

Features

- Create account
- Deposit
- Withdraw
- Check balance

Implementation

```
class BankAccount {
  private balance: number = 0;

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

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

  getBalance() {
    return this.balance;
```

```
  }
}
```

Advanced Version (Optional)

```
class SavingAccount extends BankAccount {
  addInterest() {
    let interest = this.getBalance() * 0.05;
    this.deposit(interest);
  }
}
```

Teaching Tips (Very Important)

- Use **real-life analogies** (bank, library, animals)
- Draw diagrams:
  - Class → Object → Methods
- Encourage students to:
  - Modify code
  - Break code and fix it

End of Week Outcome

Students will be able to:

- Build structured applications
- Understand real backend logic
- Prepare for:
  - APIs
  - Databases
  - Full-stack development

WEEK 3 — Databases (MongoDB + Concepts)

Goal

By the end of this week, students will:

- Understand how data is stored and managed
- Perform full **CRUD operations** (Create, Read, Update, Delete)
- Connect MongoDB with a Node.js app using **Mongoose**
- Build a simple **User Management System**

Day 1 — Database Basics + Setup

1. What is a Database?

A database is a system to **store, manage, and retrieve data**.

Real-life examples:

- WhatsApp → messages stored in DB
- Instagram → users, posts, likes
- Banking → transactions, accounts

---

2. Types of Databases

SQL (Relational)

- Tables (rows + columns)
- Example: MySQL

NoSQL (MongoDB)

- Collections (like folders)
- Documents (JSON objects)

Tables vs Collections

SQL (MySQL)	MongoDB
Table	Collection
Row	Document
Column	Field

MongoDB Document Example

```
{
  "name": "Ali",
  "age": 20,
  "city": "Lahore"
}
```

---

3. Setup Tasks

Install MongoDB

- Download MongoDB Community Server
- Install locally

Install MongoDB Compass

- GUI tool to view data

Practice

- Create database: test
- Create collection: users
- Insert manually using Compass

Day 2 — MongoDB CRUD (Shell / Compass)

---

1. CREATE

```
db.users.insertOne({ name: "Ali", age: 20 });

db.users.insertMany([
  { name: "Ahmed", age: 25 },
  { name: "Sara", age: 22 }
]);
```

---

2. READ

```
db.users.find();

db.users.find({ age: 20 });
```

---

3. UPDATE

```
db.users.updateOne(
  { name: "Ali" },
  { $set: { age: 21 } }
);
```

---

4. DELETE

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

Important Operators

- $set → update value
- $gt → greater than
- $lt → less than

```
db.users.find({ age: { $gt: 20 } });
```

Exercises

1. Insert 5 users
2. Find users with age > 22
3. Update one user
4. Delete one user

Day 3 — Mongoose + Schema

---

1. What is Mongoose?

Mongoose is a **library to connect Node.js with MongoDB**.

It provides:

- Structure (Schema)
- Validation
- Easy queries

---

2. Install

npm install mongoose

---

3. Connect Database

```
import mongoose from "mongoose";


mongoose.connect("mongodb://localhost:27017/test")
  .then(() => console.log("Connected"))
  .catch(err => console.log(err));
```

---

4. Schema (Structure)

```
const UserSchema = new mongoose.Schema({
  name: String,
```

```
  age: Number
});
```

---

5. Model (Collection)

```
const User = mongoose.model("User", UserSchema);
```

Flow

Schema → Model → Database

Example

```
const user = new User({
  name: "Ali",
  age: 20
});

user.save();
```

Exercises

1. Add fields:
    - email
    - city
2. Add validation:

```
age: { type: Number, required: true }
```

Day 4 — CRUD via Code + Project

CREATE

```
await User.create({ name: "Ali", age: 20 });
```

READ

```
const users = await User.find();

console.log(users);
```

UPDATE

```
await User.updateOne(
  { name: "Ali" },
  { age: 25 }
);
```

DELETE

```
await User.deleteOne({ name: "Ali" });
```

Mini Project — User Management System

Features

- Add user
- View users
- Update user
- Delete user

Basic Implementation

```
import mongoose from "mongoose";


mongoose.connect("mongodb://localhost:27017/test");
```

```
const UserSchema = new mongoose.Schema({
  name: String,
  age: Number
});

const User = mongoose.model("User", UserSchema);

// CREATE
await User.create({ name: "Ali", age: 20 });

// READ
const users = await User.find();
console.log(users);
```

Improved Version (Menu-Based)

```
async function run() {
  await User.create({ name: "Ahmed", age: 25 });

  const users = await User.find();
  console.log("All Users:", users);
}

run();
```

Bonus Concepts (Important for Growth)

Query Filtering

```
User.find({ age: { $gt: 20 } });
```

Sorting

User.find().sort({ age: -1 });

Limit

User.find().limit(2);

Teaching Tips

- Show **Compass + Code side-by-side**
- Let students:
    - Insert data manually
    - Fetch via code
- Explain JSON clearly (very important)

End of Week Outcome

Students will:

- Understand databases deeply
- Perform CRUD confidently
- Connect DB with backend
- Be ready for:
    - APIs (Week 4)
    - Full-stack apps

Here's a **fully expanded, practical, real-world version of Week 4 — Practical Application (OOP + Database)**. This week ties everything together and turns students into backend developers.

WEEK 4 — Practical Application (OOP + Database)

Goal (Expanded)

By the end of this week, students will:

- Build a **real backend system**
- Combine:
    - TypeScript
    - OOP concepts
    - MongoDB (via Mongoose)
- Create and test **REST APIs**
- Complete a **production-style project (Task Manager Backend)**

Day 1 — Project Setup + Architecture

1. Backend Architecture (Very Important)

Explain structure clearly:

src/

├── models/ → Database schemas

├── services/ → Business logic (OOP)

├── controllers/ → API logic

├── routes/ → API endpoints

├── config/ → DB connection

└── app.ts → Entry point

---

2. Setup Node + TypeScript

npm init -y

npm install express mongoose

npm install -D typescript ts-node @types/node @types/express

npx tsc --init

---

3. Basic Express Server

```
import express from "express";

const app = express();
app.use(express.json());

app.get("/", (req, res) => {
  res.send("Server Running");
});

app.listen(3000, () => {
  console.log("Server started");
});
```

---

4. Connect MongoDB

```
import mongoose from "mongoose";

mongoose.connect("mongodb://localhost:27017/taskdb")
  .then(() => console.log("DB Connected"))
  .catch(err => console.log(err));
```

Tasks

- Run server
- Connect DB
- Test / route in browser

Day 2 — OOP + Database Integration

---

1. Create Task Schema (Model)

```
import mongoose from "mongoose";

const TaskSchema = new mongoose.Schema({
  title: String,
  completed: Boolean
});

export const Task = mongoose.model("Task", TaskSchema);
```

---

2. Service Layer (OOP Logic)

This is where Week 2 (OOP) is applied.

```
import { Task } from "../models/task.model";

export class TaskService {

  async createTask(title: string) {
    return await Task.create({ title, completed: false });
  }

  async getTasks() {
    return await Task.find();
  }

}
```

Why Service Layer?

- Clean code
- Reusable logic
- Easy to scale

Tasks

- Create Task model
- Implement createTask
- Save data in DB

Day 3 — API Development + CRUD

---

1. What is API?

API = **Communication between frontend & backend**

---

2. Routes (Endpoints)

```
import express from "express";
import { TaskService } from "./services/task.service";

const router = express.Router();
const service = new TaskService();

router.post("/task", async (req, res) => {
  const task = await service.createTask(req.body.title);
  res.json(task);
});
```

Full CRUD APIs

CREATE

```
router.post("/task", async (req, res) => {
  const task = await service.createTask(req.body.title);
  res.json(task);
});
```

READ

```
router.get("/task", async (req, res) => {
  const tasks = await service.getTasks();
  res.json(tasks);
});
```

UPDATE

```
router.put("/task/:id", async (req, res) => {
  const updated = await Task.findByIdAndUpdate(
    req.params.id,
    req.body,
    { new: true }
  );
  res.json(updated);
});
```

DELETE

```
router.delete("/task/:id", async (req, res) => {
```

```
  await Task.findByIdAndDelete(req.params.id);
  res.json({ message: "Deleted" });
});
```

Tasks

- Test all endpoints
- Handle errors (basic try/catch)

Day 4 — Testing + Final Project

---

1. API Testing (Postman)

Tool:

- Postman

Test:

- POST → create task
- GET → list tasks
- PUT → update
- DELETE → remove

Example Request

```
POST /task
{
  "title": "Learn TypeScript"
}
```

Final Project — Task Manager Backend

Features

- Add task
- View all tasks
- Update task
- Delete task
- Store in MongoDB

Complete Flow

Request → Route → Controller → Service → Model → Database

Suggested Folder Structure

src/

├── models/task.model.ts

├── services/task.service.ts

├── routes/task.routes.ts

├── config/db.ts

└── app.ts

Bonus Features (For Strong Students)

- Add:
    - dueDate
    - priority
- Filter:

Task.find({ completed: false });

- Add validation
- Add status toggle

Teaching Tips

- Show **flow diagram repeatedly**
- Let students:
  - Break APIs
  - Debug errors
- Emphasize:
  - Separation of concerns
  - Clean architecture

FINAL OUTCOME

By end of Week 4, students will:

- Understand programming logic
- Apply OOP in real systems
- Work with MongoDB professionally
- Build and test REST APIs
- Create real backend applications

SECOND MONTH — MERN STACK DEVELOPMENT

Goal:

By the end of this month, students will:

- Build full-stack apps
- Understand frontend + backend connection
- Work with real APIs
- Create a complete MERN application

WEEK 1 — Frontend (React → Next.js)

Day 1 — Lecture (React Basics + Components)

Concepts:

- What is SPA
- Components
- Introduction to React

Example:

```
const App = () => {
  return <h1>Hello World</h1>;
};
```

Exercise:

- Create page with heading + button

Day 2 — Lab (Props + State + Events)

Concepts:

- Props
- State
- Event handling

Example:

```
type Props = {
  name: string;
};

const Welcome: React.FC<Props> = ({ name }) => {
  return <h1>Welcome {name}</h1>;
};

const [count, setCount] = useState<number>(0);
```

Exercises:

- Card component
- Counter app

Day 3 — Lab (Forms + Routing + Next.js)

Concepts:

- Forms & input handling
- Routing (React Router)
- Introduction to Next.js
- File-based routing
- SSR (basic idea)

Example:

```
const [name, setName] = useState<string>("");


<input onChange={(e) => setName(e.target.value)} />


<Route path="/dashboard" element={<Dashboard />} />


export default function Home() {
  return <h1>Home Page</h1>;
}
```

Exercises:

- Login form
- Multi-page app

Day 4 — Lab (UI + Project)

Project:

Frontend Dashboard UI

- Navbar
- Sidebar
- Cards

WEEK 2 — Backend (Node + NestJS + MongoDB)

Day 1 — Lecture (Node + Express + APIs)

Concepts:

- Node.js basics
- Express.js
- REST APIs (GET, POST, PUT, DELETE)

Example:

```
app.get("/api/test", (req, res) => {
  res.send("API working");
});


app.post("/users", (req, res) => {
  res.send("User created");
});
```

Day 2 — Lab (NestJS + Controllers + Services)

Concepts:

- NestJS structure
- Controllers
- Services

Example:

```
@Controller('users')
```

```
export class UserController {
  @Get()
  findAll() {
    return [];
  }
}
```

Day 3 — Lab (MongoDB + Schema Design)

Concepts:

- MongoDB integration
- Schema design

Example:

```
await User.create({ name: "Ali" });


const UserSchema = new Schema({
  name: String,
  email: String
});
```

Day 4 — Lab (Backend Project + Testing)

Project:

User Management API

- Create user
- Get users

Tools:

- Postman

WEEK 3 — API Integration (Frontend + Backend)

Day 1 — Lecture (Client-Server Communication)

Concepts:

- How frontend talks to backend
- Request/response cycle

Day 2 — Lab (Fetch + Axios)

Example:

```
fetch("/api/users")
  .then(res => res.json())
  .then(data => console.log(data));


axios.get("/api/users");
```

Day 3 — Lab (UI + Forms + Error Handling)

Concepts:

- Display data in UI
- Forms with API
- Error handling

Example:

```
try {
  await axios.post("/api/users");
} catch (error) {
  console.log(error);
}
```

Exercises:

- Show users in table
- Submit form → save user

Day 4 — Lab (Authentication + Mini Project)

Concepts:

- Login system
- Token basics

Project:

Connected App

- Frontend + Backend integration

WEEK 4 — Full MERN Stack Application

Day 1 — Lecture (Project Architecture)

Concepts:

- Full-stack structure
- Data flow

Day 2 — Lab (Backend + Database)

Tasks:

- Setup project
- Task schema
- CRUD APIs

Day 3 — Lab (Frontend + Integration)

Tasks:

- Dashboard UI

- Task list
- Connect frontend + backend

Day 4 — Lab (Features + Final Project)

Features:

- Add task
- Delete task
- Update task

Final Project:

Task Manager App

FINAL OUTCOME

Students will:

- Build frontend apps
- Create backend APIs
- Connect systems
- Build full MERN applications

THIRD MONTH — AI-DRIVEN CURRICULUM

WEEK 1 — AI DEVELOPMENT + MERN FOUNDATION

Day 1 — Lecture (Setup + AI Workflow + First Backend)

Step 1: Environment Setup

Students install:

- Node.js
- MongoDB

- VS Code

Verify:

node -v

npm -v

Step 2: First Prompt (Project Scaffold)

Prompt:

Create a MERN stack project folder structure.

- frontend (React)

- backend (Node + Express)

- proper separation of concerns

- include package.json for both

Step 3: Backend Generation

Prompt:

Create a basic Express server with one route /api/test returning "Server running"

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

app.get('/api/test', (req, res) => {
res.send("Server running");
});

app.listen(5000, () =>console.log("Server started"));
```

Key Insight:

Students experience:
Prompt → Code → Running system

Day 2 — Lab (System Design + Prompt Engineering)

Step 1: App Design Prompt

Design a MERN stack task management app.

Include:

- Features

- User flow

- Database schema

- API endpoints

Step 2: Prompt Improvement

Bad:

Create app

Good:

Create a scalable MERN app with authentication and task CRUD.

Use clean architecture and best practices.

Task:

Students:

- Add features (priority, deadlines)

Day 3 — Lab (Frontend Generation + UI Prompting)

React App Prompt

Create a React app with:

- Navbar

- Login page

- Dashboard

- Routing

<BrowserRouter>

<Routes>

<Route path="/" element={<Login />} />

<Route path="/dashboard" element={<Dashboard />} />

</Routes>

</BrowserRouter>

UI Prompt (Tailwind)

Create a modern dashboard UI with sidebar, navbar, and task cards

---

Teaching Focus:

- AI follows clarity
- Structure matters

Day 4 — Lab (Backend + Database + Integration)

Backend Architecture Prompt

Create Express backend with MVC and task CRUD APIs

MongoDB Schema Prompt

Create MongoDB schema for Task with title, description, status

```
const taskSchema = new mongoose.Schema({
  title: String,
  description: String,
  status: { type: String, default: "pending" }
});
```

Integration Prompt

Write React code to fetch tasks from backend API and display them

```
useEffect(() => {
  fetch("http://localhost:5000/api/tasks")
.then(res =>res.json())
.then(data =>setTasks(data));
}, []);
```

Outcome:

Students now have:

- UI
- Backend
- Database
- Data flow

WEEK 2 — ADVANCED AI DEVELOPMENT

Day 1 — Lecture (Authentication + Debugging Mindset)

Auth Prompt

Create JWT authentication with login and signup

Concepts:

- Token flow
- Protected routes

Debugging Example:

Error:

Cannot read property 'map' of undefined

Prompt:

Fix this error and explain

Day 2 — Lab (CRUD + Refactoring)

Prompt:

Create update and delete APIs and connect to frontend buttons

---

Refactor Prompt:

Improve this code with best practices

Day 3 — Lab (Feature Expansion)

Prompt:

Add search and filter to tasks

Outcome:

Students extend real system

Day 4 — Lab (Deployment + Final Build)

Deployment Prompt

Deploy MERN app on Vercel and MongoDB Atlas step by step

Final Task:

Students:

- Build independently
- Use prompts
- Present

WEEK 3 — TYPESCRIPT VIDEO CALL APP

Day 1 — Lecture (TypeScript Setup + Backend)

Prompt:

Create MERN stack project using TypeScript with proper setup

Server Example:

```
import express, { Request, Response } from "express";


app.get("/api/test", (req: Request, res: Response) => {
res.send("Server running with TypeScript");
});
```

Concepts:

- Types
- Request/Response

Day 2 — Lab (Socket.IO + React TS)

Prompt:

Create Socket.IO server with typed events

```
interface SignalData {
  to: string;
  from: string;
  signal: any;
}
```

React TS Prompt:

Create React app with typed props and routing

Day 3 — Lab (WebRTC + Integration)

Prompt:

Create WebRTC video call using RTCPeerConnection with TypeScript

const peerRef = useRef<RTCPeerConnection | null>(null);

Integration Prompt:

Integrate WebRTC with Socket.IO signaling

Day 4 — Lab (UI + Room System)

Prompt:

Create video call UI with mute, camera toggle, and routing

const { roomId } = useParams<{ roomId: string }>();

Outcome:

1-to-1 video call app

WEEK 4 — ADVANCED FEATURES

Day 1 — Lecture (Scaling + Architecture)

Concepts:

- Multi-user systems
- Real-time architecture

Day 2 — Lab (Screen Sharing + Multi-user)

Prompt:

Add screen sharing using getDisplayMedia

```
const screenStream: MediaStream = await navigator.mediaDevices.getDisplayMedia({
  video: true
});
```

Multi-user:

```
const peers: Record<string, RTCPeerConnection> = {};
```

Day 3 — Lab (Chat + Cleanup + Database)

Chat:

```
interface Message {
userId: string;
  text: string;
}
```

Cleanup:

```
socket.on("user-left", (id: string) => {
  peers[id]?.close();
  delete peers[id];
});
```

MongoDB TS:

```
interface IMeeting extends Document {
roomId: string;
}
```

Day 4 — Lab (Deployment + Final Demo)

Prompt:

Deploy TypeScript MERN app with build steps

Final Demo:

Students present:

- Video call
- Screen sharing
- Chat
- Deployment