UNIT – I

Overview of Web Development & Full Stack Architecture

1. Introduction to Web Development

Web development is the process of **designing, developing, and maintaining websites and web applications** that run on the internet or intranet. It involves creating user interfaces, handling user requests, processing data, and displaying results dynamically.

In simple words, **web development connects users with software through a web browser**.

Definition

Web Development is the combination of technologies used to build applications that can be accessed using a web browser and work over a network such as the Internet.

2. Evolution of Web Development

2.1 Static Websites

- Developed using **HTML and CSS**
- Content is fixed
- No interaction with users

Example:
College department information website

2.2 Dynamic Websites

- Uses **JavaScript and backend technologies**
- Content changes based on user input

Example:
Online registration forms

2.3 Modern Web Applications

- Highly interactive
- Use **frontend frameworks, backend APIs, databases, and AI models**

Example:
Gmail, Amazon, Netflix

3. What is Full Stack Web Development?

Definition

Full Stack Web Development refers to the development of **both frontend (client-side)** and **backend (server-side)** parts of a web application along with database management.

A **Full Stack Developer** works on:

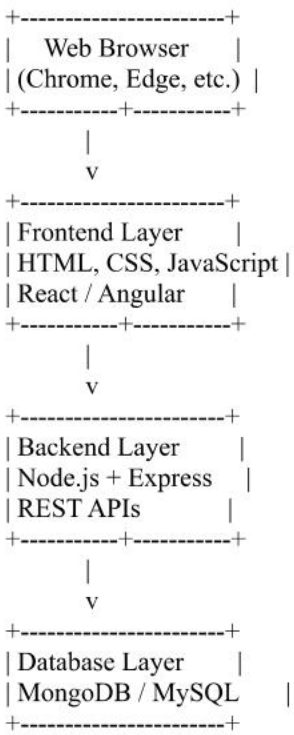
- Frontend
- Backend
- Database
- Deployment
- AI/ML integration (in modern applications)

4. Components of Full Stack Architecture

A full stack web application consists of **three main layers**:

1. Frontend Layer
2. Backend Layer
3. Database Layer

Figure 1: Full Stack Architecture



5. Frontend (Client-Side) Development

Definition

Frontend is the **visible part of the application** that users interact with directly.

Technologies Used

- **HTML5** – Structure
- **CSS3** – Styling and layout
- **JavaScript** – Interactivity
- **Frameworks** – React, Angular

Responsibilities

- User interface design
- Form handling
- Input validation
- Sending requests to backend

Example

Student Registration Form:

- Input fields for name, email, password
- Submit button
- Error messages shown to user

Live Example

Google Forms

- Form creation
- Validation
- Submission handling

6. Backend (Server-Side) Development

Definition

Backend is responsible for **processing requests, executing business logic, and communicating with databases**.

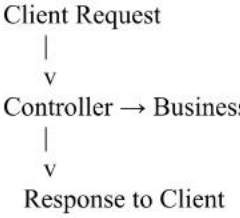
Technologies Used

- **Node.js**
- **Express.js**
- Python (Flask/Django)

Responsibilities

- Handling HTTP requests
- Authentication & authorization
- Data processing
- API creation

Figure 2: Backend Request Flow



Example

Login System:

- Validate username & password
- Generate JWT token
- Send response to frontend

Live Example

Online Banking Applications

7. Database Layer

Definition

Database is used to **store, retrieve, update, and delete data permanently**.

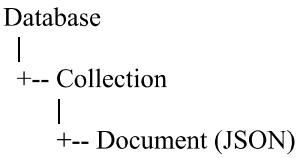
Types of Databases

- **SQL:** MySQL, PostgreSQL
- **NoSQL:** MongoDB

MongoDB Features

- Document-based
- Stores data in JSON format
- Scalable

Figure 3: MongoDB Structure



Example

Student Database:

- Name
- Roll Number
- Email
- Marks

8. Communication Between Layers

Frontend to Backend

- Uses **HTTP requests**
- GET, POST, PUT, DELETE

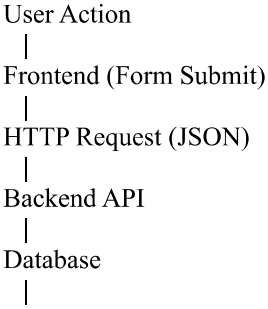
Backend to Database

- Uses database queries

Data Format

- **JSON (JavaScript Object Notation)**

Figure 4: Request–Response Cycle



HTTP Response
|
Frontend Display

9. Role of AI in Full Stack Development

In modern applications, AI is integrated to make systems **intelligent and automated**.

Examples

- Chatbots
- Recommendation systems
- Sentiment analysis
- Disease detection (forms + images)

Live Example

- Amazon product recommendations
- ChatGPT-based assistants

10. Advantages of Full Stack Architecture

1. Easy maintenance
2. Better scalability
3. Faster development
4. Seamless AI integration
5. End-to-end control

11. Real-Time Live Example (End-to-End)

Online Student Registration System

1. User fills form (Frontend)
2. Data sent to Node.js server
3. Server validates data
4. Data stored in MongoDB
5. Confirmation message shown

12. Conclusion

Full stack web development provides a **complete framework** to build modern, scalable, and intelligent web applications. Understanding full stack architecture is essential for integrating **AI and machine learning models**, making applications smarter and more efficient.

Tip (One-Line Answer)

Full stack architecture connects frontend, backend, and database to build complete, scalable web applications.

2. HTML5: Structure, Forms, Tables, Multimedia & Semantic Tags

1. Introduction to HTML5

HTML5 stands for **HyperText Markup Language – Version 5**. It is the **standard language used to create and structure web pages**. HTML5 provides improved support for multimedia, forms, and semantic elements compared to earlier versions.

Definition

HTML5 is a markup language used to define the **structure and content** of modern web applications in a browser.

2. Basic Structure of an HTML5 Document

Every HTML5 document follows a standard structure.

Syntax

```
<!DOCTYPE html>
<html>
<head>
   <title>My Page</title>
</head>
<body>
   <h1>Welcome</h1>
</body>
</html>
```

Explanation

- `<!DOCTYPE html>` → Declares HTML5 version
- `<html>` → Root element
- `<head>` → Metadata (title, links)
- `<body>` → Visible content

Figure 1: HTML Page Structure

```
HTML
 |
 +-- HEAD (Title, Meta)
 |
 +-- BODY (Content)
```

3. HTML5 Text and Formatting Elements

Common tags:

- `<h1>` to `<h6>` – Headings
- `<p>` – Paragraph
- `<b>` – Bold
- `<i>` – Italic
- `<br>` – Line break

Example

```
<h1>Student Portal</h1>
<p>This is a registration page</p>
```

Live Example

News websites like **BBC** use headings and paragraphs extensively.

4. HTML5 Forms

Definition

Forms are used to **collect user input** and send it to a server for processing.

Basic Form Structure

```
<form>
 <label>Name:</label>
 <input type="text">
 <input type="submit">
</form>
```

Common Input Types

- text
- email
- password
- number
- date
- radio
- checkbox

Figure 2: Form Working

User Input → Form → Server → Response

Example

Student Registration Form:

- Name
- Email
- Password
- Submit button

Live Example

Online exam registration forms

5. Form Attributes in HTML5

- `required` – Mandatory field
- `placeholder` – Hint text
- `readonly` – Read-only field
- `maxlength` – Limit input length

Example

```
<input type="email" required placeholder="Enter email">
```

Advantage

Reduces invalid data entry before backend processing.

6. HTML Tables

Definition

Tables are used to **display data in rows and columns**.

Table Structure

```
<table>
 <tr>
   <th>Name</th>
   <th>Marks</th>
 </tr>
 <tr>
   <td>John</td>
   <td>85</td>
 </tr>
</table>
```

Tags Used

- `<table>` – Table

- `<tr>` – Row
- `<th>` – Header
- `<td>` – Data

Figure 3: Table Representation

```
+-------+-------+
| Name  | Marks |
+-------+-------+
| John  |  85   |
+-------+-------+
```

Example

Student marks list

Live Example

University result portals

7. HTML5 Multimedia Elements

HTML5 supports multimedia **without external plugins**.

Audio Tag

```
<audio controls>
 <source src="song.mp3">
</audio>
```

Video Tag

```
<video controls>
 <source src="video.mp4">
</video>
```

Figure 4: Multimedia Support

```
[ Video Player ]
[ Play | Pause ]
```

Advantages

- No Flash required
- Better performance
- Mobile-friendly

Live Example

YouTube video embedding

8. Semantic Tags in HTML5

Definition

Semantic tags clearly describe the **meaning of content** to browsers and developers.

Common Semantic Tags

- `<header>`
- `<nav>`
- `<section>`
- `<article>`
- `<aside>`
- `<footer>`

Figure 5: Semantic Layout

```
<header>
<nav>
<section>
 <article>
<footer>
```

Importance

- Improves SEO
- Better accessibility
- Easy code readability

Example

```
<header>
  <h1>College Website</h1>
</header>
```

Live Example

Blog and news websites

9. Difference Between Semantic and Non-Semantic Tags

Semantic	Non-Semantic
header	div
article	span
footer	div

Semantic tags provide **meaning**, non-semantic do not.

10. Advantages of HTML5

1. Simple syntax
2. Multimedia support
3. Built-in form validation
4. Semantic structure
5. Mobile-friendly

11. Real-Time Example (Complete Use Case)

College Admission Portal

- HTML form collects details
- Tables show merit list
- Video shows campus tour
- Semantic tags structure page

12. Conclusion

HTML5 is the **foundation of web development**. It provides a structured, semantic, and multimedia-rich way to build modern web applications and works seamlessly with CSS, JavaScript, and backend technologies.

Tip (One Line)

HTML5 is used to structure web content with semantic meaning and multimedia support.

3.CSS3: Selectors, Layouts, Flexbox, Grid, Responsive Design & Tailwind CSS

1. Introduction to CSS3

CSS stands for **Cascading Style Sheets**. CSS3 is the latest version used to **style and design web pages** created using HTML. It controls layout, colors, fonts, spacing, and responsiveness.

Definition

CSS3 is a stylesheet language used to describe the **presentation and layout** of HTML elements on different devices.

2. Types of CSS

2.1 Inline CSS

Applied directly inside HTML elements.

```
<p style="color:red;">Hello</p>
```

2.2 Internal CSS

Defined inside `<style>` tag.

```
<style>
 p { color: blue; }
</style>
```

2.3 External CSS

Defined in a separate `.css` file.

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

3. CSS Selectors

Definition

Selectors are used to **select HTML elements** to apply styles.

Types of Selectors

Element Selector

p { color: red; }

Class Selector

.box { background: yellow; }

ID Selector

#main { width: 100%; }

Group Selector

h1, h2 { color: green; }

Figure 1: Selector Targeting

HTML Element → CSS Selector → Style Applied

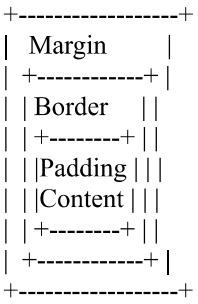
4. CSS Box Model

The CSS box model defines the **space occupied by elements**.

Components

- Content
- Padding
- Border
- Margin

Figure 2: Box Model



Example

```
div {
 margin: 10px;
 padding: 15px;
}
```

5. CSS Layout Techniques

5.1 Float Layout (Old)

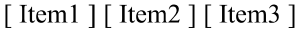
Used earlier but difficult to manage.

5.2 Flexbox Layout

Flexbox is used for **one-dimensional layouts** (row or column).

```
.container {
 display: flex;
 justify-content: space-between;
}
```

Figure 3: Flexbox Layout



Advantages

- Easy alignment
- Responsive

Live Example

Navigation bars

6. CSS Grid Layout

Grid is used for **two-dimensional layouts** (rows and columns).

```
.container {
 display: grid;
 grid-template-columns: 1fr 1fr;
}
```

Figure 4: Grid Layout

```
+------+------+
| Box1 | Box2 |
+------+------+
| Box3 | Box4 |
+------+------+
```

Use Case

Dashboard layouts

7. Responsive Web Design

Definition

Responsive design ensures websites **adapt to different screen sizes**.

Techniques

- Media Queries
- Flexible images
- Relative units

```
@media(max-width:600px) {
 body { background: lightblue; }
}
```

Figure 5: Responsive Design

Desktop → Tablet → Mobile

Live Example

E-commerce websites

8. CSS3 Advanced Features

- Transitions
- Animations
- Gradients
- Shadows

Example

```
button:hover {
 background: green;
}
```

9. Tailwind CSS

Definition

Tailwind CSS is a **utility-first CSS framework**.

Example

```
<button class="bg-blue-500 text-white p-2">
 Submit
</button>
```

Advantages

- Faster development
- No custom CSS needed
- Responsive utilities

Live Example

Modern startup websites

10. Advantages of CSS3

1. Improves UI design
2. Responsive layouts
3. Cross-browser support
4. Reusable styles
5. Clean separation of content & design

11. Real-Time Example

Online Shopping Website

- Flexbox for navbar

- Grid for product display
- Media queries for mobile view
- Tailwind for quick styling

12. Conclusion

CSS3 plays a vital role in creating **attractive, responsive, and user-friendly interfaces**. Combined with HTML5 and JavaScript, it forms the core of frontend development.

Tip

4. CSS3 controls layout, responsiveness, and visual appearance of web pages.

1. Introduction to JavaScript

JavaScript is a **client-side scripting language** used to create **dynamic and interactive web pages**. It allows web pages to respond to user actions without reloading the page.

Definition

JavaScript is a lightweight, interpreted programming language used to add **interactivity and logic** to web applications.

2. JavaScript Execution Environment

JavaScript runs in:

- Web browsers (Chrome, Edge)
- Server-side (Node.js)

Figure 1: JavaScript Working

HTML → JavaScript → Browser Engine → Output

3. JavaScript Variables

Variables store data values.

Types of Variables

var

- Function scoped
- Older method

let

- Block scoped

const

- Constant values

Example

```
let age = 20;
const pi = 3.14;
```

Live Example

Storing user age in a registration form.

4. JavaScript Data Types

- Number
- String
- Boolean
- Object
- Array
- Undefined
- Null

Example

```
let name = "John";
let marks = 85;
```

5. JavaScript Functions

Functions are reusable blocks of code.

Types of Functions

Normal Function

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

Arrow Function (ES6)

```
const add = (a,b) => a+b;
```

Figure 2: Function Execution

Function Call → Execution → Return Value

Live Example

Calculating total marks.

6. JavaScript Arrays

Arrays store multiple values in a single variable.

Example

```
let marks = [80, 85, 90];
```

Array Methods

- push()
- pop()
- length
- map()

Live Example

Storing student marks list.

7. JavaScript Events

Events are actions performed by users.

Common Events

- onclick
- onsubmit
- onkeyup
- onload

Example

```
<button onclick="alert('Hello')">Click</button>
```

Figure 3: Event Handling

User Action → Event → JavaScript Function

Live Example

Button click validation.

8. Document Object Model (DOM)

Definition

DOM represents the **HTML document as a tree structure**.

Figure 4: DOM Tree

```
Document
 |
 +-- html
    |
    +-- body
       |
       +-- h1
       +-- p
```

DOM Manipulation

```
document.getElementById("msg").innerHTML = "Welcome";
```

Live Example

Displaying error messages dynamically.

9. Form Handling using JavaScript

JavaScript validates input before sending to server.

Example

```
if(password.length < 6){
 alert("Weak password");
}
```

Advantage

Reduces server load.

10. ES6+ Features

Important ES6 Features

- Arrow functions
- let & const
- Template literals
- Spread operator
- Destructuring

Example

```
let msg = `Welcome ${name}`;
```

11. Advantages of JavaScript

1. Client-side validation
2. Faster execution
3. Interactive UI
4. Cross-platform
5. Easy integration

12. Real-Time Example

Online Feedback Form

- JavaScript validates inputs
- Displays confirmation message
- Updates content dynamically

13. Conclusion

JavaScript is the **backbone of interactive web applications**. It works seamlessly with HTML and CSS and is essential for full stack development.

Viva Tip

JavaScript adds interactivity and dynamic behavior to web pages

5. Form Elements & Validation, Debugging Tools

1. Introduction to Forms in Web Applications

Forms are an essential part of web applications. They are used to **collect user input**, such as login details, registration information, feedback, and search queries. Forms act as a **bridge between users and servers**.

Definition

A **web form** is a section of a web page that allows users to enter data and submit it for processing.

2. HTML Form Elements

HTML provides various input elements to collect different types of data.

Common Form Elements

- `<form>`
- `<input>`
- `<label>`
- `<textarea>`
- `<select>`
- `<option>`
- `<button>`

Example

```
<form>
 <label>Name:</label>
 <input type="text">
 <button>Submit</button>
</form>
```

3. Input Types in HTML5

HTML5 introduced several advanced input types to improve user experience.

Important Input Types

- text
- email
- password
- number
- date
- radio
- checkbox
- file

Example

```
<input type="email" required>
<input type="password">
```

Figure 1: Input Types

Text | Email | Password | Date | Checkbox

4. Client-Side Form Validation

Definition

Client-side validation checks user input **before sending it to the server**.

Types of Validation

1. Built-in HTML validation
2. JavaScript-based validation

5. HTML5 Built-in Validation

Validation Attributes

- `required`
- `minlength`
- `maxlength`
- `pattern`
- `readonly`
- `disabled`

Example

```
<input type="text" required minlength="3">
```

Advantage

- Reduces server load
- Immediate feedback to users

6. JavaScript Form Validation

JavaScript allows **custom validation logic**.

Example

```
function validate(){
 if(password.length < 6){
  alert("Password too short");
 }
}
```

Figure 2: Validation Flow

User Input → Validation → Submit / Error

Live Example

Checking password strength in signup forms.

7. Error Messages and User Feedback

Proper error messages improve usability.

Techniques

- Alert boxes
- Inline error messages
- Color indicators

Example

```
<span style="color:red;">Invalid Email</span>
```

8. Form Handling Process

Steps

1. User enters data
2. Client-side validation
3. Data sent to backend
4. Server processing
5. Response displayed

Figure 3: Form Submission Cycle

User → Form → JS Validation → Server → Response

9. Debugging in Web Development

Definition

Debugging is the process of **identifying and fixing errors** in a program.

10. Console Debugging Tools

Browser Console

- `console.log()`
- `console.error()`
- `console.warn()`

Example

```
console.log("Form submitted");
```

Live Example

Debugging JavaScript errors in Chrome DevTools.

11. Browser Developer Tools

Features

- Elements tab
- Console tab
- Network tab
- Sources tab

Figure 4: Developer Tools

Elements | Console | Network | Sources

Use Case

Checking form submission errors.

12. Common Form Errors

- Missing required fields
- Invalid email format
- Weak passwords

Solution

- Validation
- Clear error messages

13. Advantages of Form Validation & Debugging

1. Improved data accuracy
2. Better user experience
3. Reduced backend errors
4. Faster development

14. Real-Time Example

Online Job Application Form

- Required fields validation
- Password rules
- Console used to debug submission errors

15. Conclusion

Form elements and validation ensure **secure and accurate data collection**. Debugging tools help developers detect and fix errors efficiently.

Git & GitHub: Version Control, Branching, Merging & Project Hosting

1. Introduction to Version Control Systems

In software development, multiple developers work on the same project. A **Version Control System (VCS)** helps manage changes made to source code over time and avoids conflicts.

Definition

A **Version Control System** is a tool that tracks changes in files, allows multiple developers to collaborate, and maintains previous versions of code.

2. Types of Version Control Systems

2.1 Local Version Control

- Stores versions locally
- No collaboration support

2.2 Centralized Version Control

- Single central repository
- Example: SVN

2.3 Distributed Version Control

- Each user has a full copy
- Example: Git

3. Introduction to Git

Definition

Git is a **distributed version control system** used to track code changes and collaborate with teams efficiently.

Features of Git

- Fast

- Secure
- Distributed
- Open source

4. Git Architecture

Figure 1: Git Workflow

Working Directory → Staging Area → Repository

Explanation

- Working Directory: Files being edited
- Staging Area: Files ready to commit
- Repository: Stored history

5. Basic Git Commands

Initialize Repository

```
git init
```

Add Files

```
git add .
```

Commit Changes

```
git commit -m "Initial commit"
```

Check Status

```
git status
```

6. Branching in Git

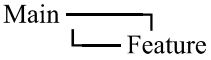
Definition

A **branch** allows developers to work on new features **without affecting the main code**.

Example

```
git branch feature
git checkout feature
```

Figure 2: Branching Model



Advantage

Parallel development

7. Merging in Git

Definition

Merging combines changes from different branches into one.

Example

```
git checkout main
git merge feature
```

Merge Conflicts

Occurs when two branches modify the same file.

Solution

Manual conflict resolution

8. Introduction to GitHub

Definition

GitHub is a cloud-based platform used to **host Git repositories** and collaborate online.

Features

- Code hosting
- Collaboration
- Issue tracking
- Pull requests

9. GitHub Workflow

Figure 3: GitHub Collaboration

Local Repo → Push → GitHub Repo → Pull Request

Example

Team project collaboration

10. Pull Requests

Definition

A pull request allows developers to **request merging code changes** into the main branch.

Advantage

- Code review
- Quality control

11. Project Hosting using GitHub Pages

Definition

GitHub Pages allows hosting **static websites for free**.

Steps

1. Create repository
2. Push project
3. Enable GitHub Pages

Live Example

Personal portfolio website

12. Security & Best Practices

- Commit frequently
- Use meaningful messages
- Protect main branch

13. Real-Time Example

College Website Hosting

- Code managed using Git
- Hosted on GitHub Pages
- Team collaboration using pull requests

14. Advantages of Git & GitHub

1. Code backup
2. Team collaboration

3. Easy rollback
4. Free hosting
5. Industry standard

UNIT – II

Introduction to Server-Side Programming using Node.js

1. Introduction to Server-Side Programming

In web applications, some processing happens on the **client-side (browser)** and some on the **server-side**. Server-side programming refers to executing code on the **server** to process user requests, interact with databases, and send responses back to the client.

Definition

Server-side programming is the process of writing programs that run on a server to handle requests, apply business logic, and manage data.

2. Client-Side vs Server-Side Programming

Client-Side	Server-Side
Runs in browser	Runs on server
HTML, CSS, JS	Node.js, Python
UI interaction	Data processing
Less secure	More secure

Example

- Client-side: Form validation
- Server-side: Storing data in database

3. Introduction to Node.js

Definition

Node.js is an **open-source, server-side JavaScript runtime environment** that allows JavaScript to run outside the browser.

Node.js uses the **V8 JavaScript engine**, which is also used by Google Chrome

4. Why Node.js for Backend Development?

Earlier, JavaScript was only used in browsers. Node.js made it possible to use **JavaScript for backend development**, enabling full stack development using a single language.

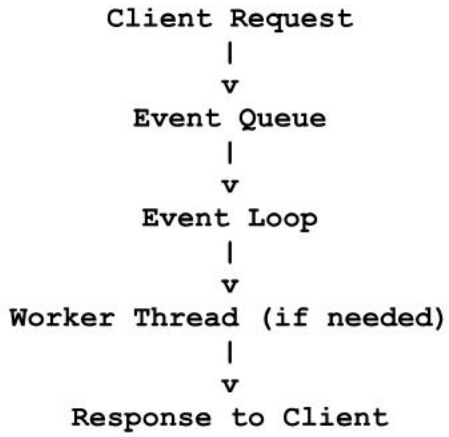
Advantages

- Same language for frontend & backend
- High performance
- Scalable
- Large ecosystem (NPM)

5. Node.js Architecture

Node.js follows an **event-driven, non-blocking I/O model**.

Figure 1: Node.js Architecture



Explanation

- Handles multiple requests simultaneously
- Does not block execution

6. Event Loop in Node.js

Definition

The **Event Loop** is the core mechanism that handles asynchronous operations in Node.js.

Working

1. Receives request
2. Registers callback
3. Executes task
4. Sends response

Example

Handling multiple users accessing a website at the same time.

7. Blocking vs Non-Blocking I/O

Blocking I/O

- Waits for task completion
- Slower

Non-Blocking I/O

- Executes tasks asynchronously
- Faster and scalable

Figure 2: Blocking vs Non-Blocking

```
Blocking:   Task1 → Task2 → Task3
Non-Block:  Task1
            Task2
            Task3
```

8. Installing Node.js

Steps

1. Download from official website
2. Install Node.js
3. Verify installation

```
node -v
npm -v
```

9. Node.js Modules

Definition

Modules are reusable blocks of code.

Types of Modules

- Built-in modules (http, fs)
- User-defined modules
- Third-party modules

Example

```
const http = require('http');
```

10. Creating a Simple Node.js Server

Example Code

```
const http = require('http');

http.createServer((req, res) => {
 res.write("Hello World");
 res.end();
}).listen(3000);
```

Explanation

- Creates a server
- Listens on port 3000
- Sends response to browser

Live Example

Opening `http://localhost:3000`

11. NPM (Node Package Manager)

Definition

NPM is used to **install, manage, and share Node.js packages**.

Commands

```
npm init
npm install express
```

Advantage

Easy dependency management

12. Node.js Use Cases

- REST APIs
- Chat applications
- Real-time systems
- Streaming services

Live Examples

- Netflix
- PayPal
- LinkedIn

13. Advantages of Node.js

1. High performance
2. Non-blocking architecture
3. Large community
4. Easy scalability
5. Full stack development

14. Real-Time Example

Online Feedback System

1. User submits feedback
2. Node.js server receives request
3. Data processed
4. Stored in database
5. Response sent back

15. Conclusion

Node.js plays a crucial role in backend development by enabling **efficient, scalable, and fast server-side applications** using JavaScript.

> **Tip**

Node.js allows JavaScript to run on the server using an event-driven, non-blocking model.

Express.js Framework: Routing, Middleware, Body Parsing & CORS

1. Introduction to Express.js

Node.js alone provides low-level APIs for creating servers. To simplify backend development, frameworks are used. **Express.js** is the most popular web framework built on top of Node.js.

Definition

Express.js is a lightweight and flexible **Node.js framework** used to build web applications and RESTful APIs easily.

2. Why Express.js?

Express.js reduces complexity and provides ready-made features.

Advantages

- Simplifies routing
- Supports middleware
- Fast and minimal
- Easy API creation

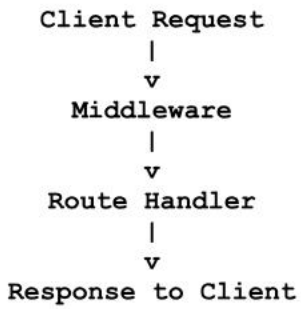
Live Example

Most backend services of **MEAN / MERN stack** use Express.js.

3. Express.js Architecture

Express works on a **request–response cycle**.

Figure 1: Express.js Flow



4. Creating an Express Application

Steps

1. Initialize Node project
2. Install Express
3. Create server

Example Code

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

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

Explanation

- `express()` creates app
- `listen()` starts server

5. Routing in Express.js

Definition

Routing refers to **handling different URLs (paths) with specific logic**.

Types of Routes

- GET
- POST
- PUT
- DELETE

Example

```
app.get('/home', (req, res) => {
 res.send("Welcome Home");
});
```

Figure 2: Routing Mechanism

```
URL → Route → Function → Response
```

Live Example

- `/login`
- `/register`

- `/dashboard`

6. RESTful APIs using Express.js

REST Principles

- Stateless
- Uses HTTP methods
- Resource-based URLs

Example API

```
app.post('/students', (req, res) => {
 res.send("Student Added");
});
```

Live Example

Student Registration API

7. Middleware in Express.js

Definition

Middleware is a function that executes **between request and response**.

Types of Middleware

1. Application-level
2. Router-level
3. Built-in
4. Third-party

Example

```
app.use((req, res, next) => {
 console.log("Middleware executed");
 next();
});
```

Figure 3: Middleware Chain

```
Request → Middleware1 → Middleware2 → Route → Response
```

8. Built-in Middleware (Body Parsing)

Express uses middleware to read request data.

JSON Body Parser

```
app.use(express.json());
```

URL Encoded Parser

```
app.use(express.urlencoded({ extended: true }));
```

Purpose

- Read form data
- Parse JSON requests

9. Body Parsing Example

Example

```
app.post('/login', (req, res) => {
 console.log(req.body);
 res.send("Login Success");
});
```

Live Example

Angular form submitting data to Express server.

10. CORS (Cross-Origin Resource Sharing)

Definition

CORS allows or restricts **requests from different origins**.

Problem Without CORS

Frontend and backend run on different ports.

Solution

Use CORS middleware.

Example

```
const cors = require('cors');
app.use(cors());
```

Figure 4: CORS Working

```
Frontend (4200) → Backend (3000)
          |
        CORS Enabled
```

11. Error Handling in Express.js

Default Error Handling

Express handles basic errors automatically.

Custom Error Handling

```
app.use((err, req, res, next) => {
 res.status(500).send("Error occurred");
});
```

Importance

- Improves reliability
- User-friendly messages

12. Security Best Practices in Express.js

- Validate inputs
- Use HTTPS
- Use authentication
- Handle errors properly

13. Real-Time Example

Contact Form Application

1. User submits form
2. Express route handles request
3. Middleware parses data
4. Response sent to frontend

14. Advantages of Express.js

1. Fast development
2. Minimal code
3. Middleware support
4. REST API friendly
5. Scalable

15. Conclusion

Express.js simplifies server-side development by providing powerful routing, middleware, and request-handling features, making it ideal for modern web applications.

✅ Tip

Express.js is a Node.js framework used to build web servers and RESTful APIs easily.

MongoDB & Mongoose: Database Setup and CRUD Operations

1. Introduction to Databases

A database is used to **store, manage, and retrieve data efficiently**. In web applications, databases play a vital role in maintaining user records, transactions, and application data.

Definition

A **database** is an organized collection of data that can be easily accessed, managed, and updated.

2. Introduction to MongoDB

Definition

MongoDB is a **NoSQL, document-oriented database** that stores data in **JSON-like documents** called BSON.

Features of MongoDB

- Schema-less
- High scalability
- Fast performance
- Document-based storage

3. MongoDB vs Relational Databases

MongoDB	**Relational DB**
NoSQL	SQL
Document-based	Table-based
Flexible schema	Fixed schema
High scalability	Limited scalability

4. MongoDB Data Structure

MongoDB organizes data in a hierarchical structure.

Figure 1: MongoDB Structure

```
Database
  |
  +-- Collection
        |
        +-- Document (JSON)
```

Example Document

```
{
 name: "Alice",
 age: 20,
 course: "CSE"
}
```

5. Installing and Setting Up MongoDB

Steps

1. Install MongoDB server
2. Start MongoDB service
3. Use MongoDB shell or Compass

Verify Installation

```
mongod
mongo
```

6. Introduction to Mongoose

Definition

Mongoose is an **Object Data Modeling (ODM) library** for MongoDB and Node.js.

Purpose of Mongoose

- Defines schema
- Validates data
- Simplifies database operations

7. Mongoose Schema and Model

Schema Definition

```
const studentSchema = new mongoose.Schema({
 name: String,
 age: Number,
 email: String
```

```
});
```

Model Creation

```
const Student = mongoose.model('Student', studentSchema);
```

Figure 2: Mongoose Workflow

```
Schema → Model → Database
```

8. Connecting Node.js with MongoDB

Example

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

Explanation

- Connects to MongoDB database
- Handles connection errors

9. CRUD Operations

CRUD stands for:

- Create
- Read
- Update
- Delete

10. Create Operation

Example

```
const student = new Student({
 name: "John",
 age: 21,
 email: "john@gmail.com"
});
student.save();
```

Live Example

Adding student details from a registration form.

11. Read Operation

Example

```
Student.find()
.then(data => console.log(data));
```

Live Example

Displaying student list in dashboard.

12. Update Operation

Example

```
Student.updateOne({name:"John"}, {age:22});
```

Live Example

Updating student profile details.

13. Delete Operation

Example

```
Student.deleteOne({name:"John"});
```

Live Example

Deleting inactive user accounts.

14. Error Handling in MongoDB

Common Errors

- Connection errors
- Validation errors

Solution

- Try-catch
- Promise handling

15. Advantages of MongoDB & Mongoose

1. Flexible data structure
2. Easy integration with Node.js
3. Faster development
4. Scalable
5. Efficient CRUD operations

16. Real-Time Example

Student Management System

1. User submits data
2. Express API processes request
3. MongoDB stores data
4. Data displayed dynamically

17. Conclusion

MongoDB combined with Mongoose provides a **powerful, scalable, and flexible database solution** for full stack web applications.

✅ Tip

MongoDB stores data as documents, and Mongoose provides schema-based interaction with MongoDB.

AJAX & JSON: Asynchronous Data Exchange

1. Introduction to AJAX

Modern web applications update content dynamically without refreshing the entire page. This is achieved using **AJAX**.

Definition

AJAX (Asynchronous JavaScript and XML) is a technique used to **send and receive data from a server asynchronously** without reloading the web page.

2. Need for AJAX

Before AJAX, every user request required a full page reload. AJAX improves **performance and user experience**.

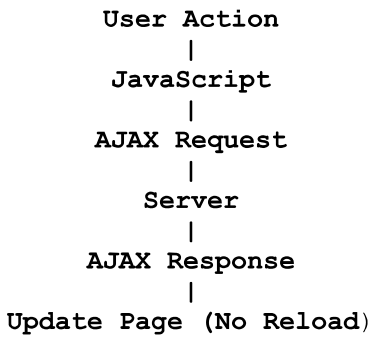
Example

- Search suggestions
- Live form validation
- Dynamic content loading

3. Working of AJAX

AJAX works using **JavaScript and HTTP requests**.

Figure 1: AJAX Working Flow



4. Synchronous vs Asynchronous Requests

Synchronous

- Waits for response
- Page freezes

Asynchronous

- Non-blocking
- Page remains responsive

Figure 2: Request Comparison

```
Synchronous: Task → Wait → Response
Asynchronous: Task → Continue → Response
```

5. AJAX Technologies

AJAX uses:

- JavaScript
- XMLHttpRequest / Fetch API
- Server-side scripts
- Data format (JSON/XML)

6. XMLHttpRequest Object

Example

```
let xhr = new XMLHttpRequest();
xhr.open("GET", "/data", true);
xhr.send();
```

Explanation

- `open()` defines request
- `send()` sends request

7. Fetch API (Modern AJAX)

Fetch API is a modern alternative to XMLHttpRequest.

Example

```
fetch('/data')
.then(res => res.json())
.then(data => console.log(data));
```

Advantages

- Cleaner syntax
- Promise-based

8. Introduction to JSON

Definition

JSON (JavaScript Object Notation) is a lightweight **data interchange format** used for data exchange between client and server.

9. JSON Structure

Example

```
{
 "name": "Alice",
 "age": 20,
 "course": "CSE"
}
```

Figure 3: JSON Data Flow

```
Database → JSON → Server → Client
```

10. JSON vs XML

JSON	XML
Lightweight	Heavy

Easy to read	Complex
Faster	Slower

11. Parsing JSON Data

Convert JSON to Object

```
JSON.parse(jsonData);
```

Convert Object to JSON

```
JSON.stringify(obj);
```

12. AJAX with Express & MongoDB

AJAX fetches data from Express APIs which retrieve data from MongoDB.

Example Flow

1. Frontend sends AJAX request
2. Express handles API
3. MongoDB returns data
4. JSON response sent
5. UI updated dynamically

13. Error Handling in AJAX

Example

```
fetch('/data')
.catch(error => console.log(error));
```

Importance

- Prevents crashes
- Improves reliability

14. Real-Time Live Examples

- Google search suggestions
- Gmail inbox loading
- Online shopping filters

15. Advantages of AJAX & JSON

1. No page reload
2. Faster response
3. Better user experience
4. Reduced server load
5. Efficient data exchange

16. Real-Time Example (Use Case)

Student Dashboard

- AJAX fetches student list
- JSON data displayed
- Page updates dynamically

17. Conclusion

AJAX and JSON form the backbone of **modern interactive web applications**, enabling seamless communication between frontend and backend systems.

✅ Tip

AJAX allows asynchronous data exchange without reloading the web page using JSON.

Angular: Components, Forms, Two-Way Data Binding & API Integration

1. Introduction to Angular

Angular is a **frontend framework** developed by Google for building **dynamic, single-page web applications (SPAs)**. It uses **TypeScript** and follows a **component-based architecture**.

Definition

Angular is a platform and framework used to develop **client-side applications** using HTML, CSS, and TypeScript.

2. Features of Angular

- Component-based architecture
- Two-way data binding
- Dependency injection
- Built-in routing
- Strong support for REST APIs

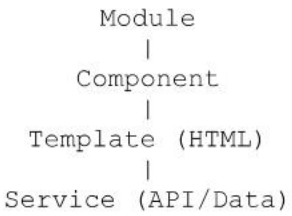
Live Example

Web dashboards, admin panels, enterprise applications

3. Angular Architecture

Angular applications are made up of **modules, components, templates, and services**.

Figure 1: Angular Architecture



4. Angular Components

Definition

A **component** controls a part of the user interface and contains:

- HTML template
- TypeScript logic
- CSS styles

Component Structure

```
@Component({
 selector: 'app-root',
 templateUrl: './app.component.html'
})
export class AppComponent { }
```

Explanation

- Selector defines HTML tag
- Template controls view
- Class controls logic

Live Example

Login component, registration component

5. Angular Templates

Templates define the **view layer** of Angular.

Features

- Data binding
- Directives
- Event handling

Example

```
<h1>{{ title }}</h1>
<button (click)="submit()">Submit</button>
```

6. Angular Forms

Angular supports two types of forms:

6.1 Template-Driven Forms

- Simple
- Uses HTML directives

Example

```
<input [(ngModel)]="name">
```

6.2 Reactive Forms

- Model-driven
- More control and validation

Example

```
this.form = new FormGroup({
 name: new FormControl('')
});
```

Figure 2: Angular Form Handling

```
User Input → Angular Form → Validation → Submit
```

7. Two-Way Data Binding

Definition

Two-way data binding keeps **model and view synchronized**.

Syntax

```
[(ngModel)]="username"
```

Figure 3: Two-Way Binding

```
Component ↔ View
```

Live Example

Updating text dynamically as user types.

8. Angular Directives

Types of Directives

- Structural: `*ngIf`, `*ngFor`
- Attribute: `ngClass`, `ngStyle`

Example

```
<div *ngIf="isLoggedIn">Welcome</div>
```

9. Services in Angular

Definition

Services are used to **share data and logic** across components.

Example

```
@Injectable()
export class StudentService { }
```

Use Case

API calls, data sharing

10. API Integration in Angular

Angular uses **HttpClient** for API communication.

Example

```
this.http.get('/api/students')
.subscribe(data => this.students = data);
```

Figure 4: API Integration Flow

```
Angular → Express API → MongoDB → Response
```

11. Handling HTTP Methods

- GET – Fetch data
- POST – Send data
- PUT – Update data
- DELETE – Remove data

Live Example

CRUD operations in dashboard

12. Error Handling in Angular

Example

```
this.http.get('/api')
.subscribe({
 error: err => console.log(err)
});
```

Importance

- Improves stability
- Better user experience

13. Advantages of Angular

1. Modular architecture
2. Efficient data binding
3. Scalable applications
4. Built-in validation
5. Strong community support

14. Real-Time Example

Student Registration System

1. Angular form collects data
2. Two-way binding updates model
3. API sends data to Express
4. MongoDB stores records
5. Confirmation shown to user

15. Conclusion

Angular simplifies frontend development by providing **structured components, powerful forms, and seamless backend integration**, making it ideal for full stack applications.

✅ Tip

Angular uses component-based architecture with two-way data binding for dynamic web applications.

Authentication (JWT), Error Handling & Security Best Practices

1. Introduction to Authentication

In web applications, authentication is used to **verify the identity of users** before allowing access to protected resources.

Definition

Authentication is the process of confirming that a user is who they claim to be.

2. Authentication vs Authorization

Authentication	**Authorization**
Verifies identity	Verifies access rights
Login process	Permission control
Username & password	Role-based access

Example

- Authentication: Login
- Authorization: Access admin page

3. Need for Authentication in Web Applications

Without authentication:

- Unauthorized access occurs
- Data theft risk
- Security breaches

Live Example

Online banking and e-commerce websites.

4. Introduction to JWT (JSON Web Token)

Definition

JWT is a compact and secure method for **token-based authentication** used in modern web applications.

JWT is commonly used in **RESTful APIs**.

5. Structure of JWT

A JWT consists of **three parts**:

Figure 1: JWT Structure

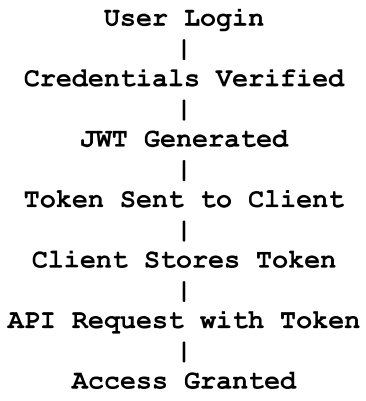
```
Header.Payload.Signature
```

Explanation

- Header: Algorithm & token type
- Payload: User data
- Signature: Verification

6. JWT Authentication Flow

Figure 2: JWT Authentication Process



7. Implementing JWT in Node.js

Token Generation

```
jwt.sign({id:userId}, "secretKey");
```

Token Verification

```
jwt.verify(token, "secretKey");
```

Live Example

Login system in MERN stack applications.

8. Advantages of JWT

1. Stateless authentication
2. Secure
3. Scalable
4. Fast performance
5. Works well with APIs

9. Error Handling in Web Applications

Definition

Error handling is the process of **detecting, managing, and responding to errors** gracefully.

10. Types of Errors

- Client-side errors
- Server-side errors
- Database errors
- Network errors

11. Error Handling in Express.js

Example

```
app.use((err, req, res, next) => {
 res.status(500).send("Server Error");
});
```

Importance

- Prevents app crash
- Improves reliability

12. Common HTTP Error Codes

Code	Meaning

	Success
	Bad Request
	Unauthorized
	Not Found
	Server Error

13. Security Best Practices

Input Validation

- Prevent SQL injection
- Prevent XSS attacks

Password Security

- Hash passwords
- Never store plain text passwords

HTTPS

- Encrypts data transmission

14. Protecting APIs

Techniques

- JWT authentication
- Role-based access
- Rate limiting

Figure 3: Secure API Flow

```
Client → Token → API → Authorized Access
```

15. Cross-Site Attacks

Common Attacks

- XSS (Cross-Site Scripting)
- CSRF (Cross-Site Request Forgery)

Prevention

- Sanitizing inputs
- Using tokens

16. Real-Time Example

Secure Student Portal

1. User logs in
2. JWT generated
3. Token sent to frontend
4. Protected routes accessed
5. Secure logout

17. Advantages of Security Implementation

1. Protects user data
2. Prevents unauthorized access
3. Builds user trust
4. Ensures compliance

18. Conclusion

Authentication, error handling, and security practices are **critical components of full stack development**. JWT provides a secure and scalable authentication mechanism for modern web applications.

✓ Viva Tip

JWT is used for secure, stateless authentication in RESTful web applications.

1. React Fundamentals

(Components, Props, State, Hooks, Lifecycle)

Introduction to React

React is a **JavaScript library used to build user interfaces (UI)**, especially **single page applications (SPA)**.

It was developed by **Facebook** to create fast and interactive web applications.

Key Features

- Component-based architecture
- Virtual DOM for fast updates
- Reusable UI components
- Easy state management

1.1 Components in React

Definition

A **component** is a **reusable piece of UI** that returns HTML using JavaScript.

Example: Navbar, Login Form, Footer etc.

Types of Components

1. Functional Components
2. Class Components

Example: Functional Component

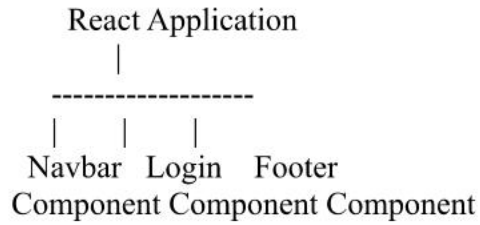
```
function Welcome() {
  return <h1>Hello Student</h1>;
}

export default Welcome;
```

Example: Class Component

```
import React, { Component } from "react";

class Welcome extends Component {
  render() {
    return <h1>Hello Student</h1>;
```

```
  }
}

export default Welcome;
```

Figure: React Component Structure



Each component works **independently and can be reused**.

1.2 Props in React

Definition

Props (properties) are **used to pass data from parent component to child component**.

Props are **read-only**.

Example

Parent Component

```
function App() {
  return <Student name="Lakshmi" branch="CSE" />;
}
```

Child Component

```
function Student(props) {
  return (
    <h2>
      Name: {props.name} Branch: {props.branch}
    </h2>
  );
}
```

Output

Name: Lakshmi Branch: CSE

Props Data Flow Diagram

Parent Component
|
| Props
v
Child Component

1.3 State in React

Definition

State is a **built-in object that stores dynamic data of a component**.

When state changes, **the component re-renders automatically**.

Example

```
import React, { useState } from "react";

function Counter() {

 const [count, setCount] = useState(0);

 return (
   <div>
     <h2>{count}</h2>
     <button onClick={() => setCount(count + 1)}>
       Increase
     </button>
   </div>
 );
}

export default Counter;
```

State Flow

User Clicks Button
|
v
State Updates
|
v
Component Re-renders

1.4 React Hooks

Definition

Hooks allow **functional components to use state and lifecycle features**.

Hooks were introduced in **React 16.8**.

Common Hooks

Hook	Purpose
useState	Manage state
useEffect	Side effects
useContext	Access global state
useRef	Access DOM elements
Hook	**Purpose**

Example: useState

```
const [name, setName] = useState("Lakshmi");
```

Example: useEffect

```
useEffect(() => {
 console.log("Component Loaded");
}, []);
```

Figure: Hook Working

```
Component
   |
   |---- useState()
   |---- useEffect()
   |---- useContext()
```

1.5 React Lifecycle

Lifecycle represents **different stages of a component**.

Lifecycle Phases

1. Mounting
2. Updating
3. Unmounting

1. Mounting

Component is **created and inserted into DOM**.

Methods

constructor()
render()
componentDidMount()

2. Updating

Occurs when **state or props change**.

Methods

shouldComponentUpdate()
render()
componentDidUpdate()

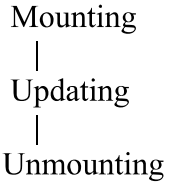
3. Unmounting

Component is **removed from DOM**.

Method

componentWillUnmount()

Lifecycle Diagram



2. Forms in React

Forms are used to **collect user input**.

Example:

- Login form
- Registration form
- Feedback form

2.1 Controlled Components

Definition

Form elements whose **values are controlled by React state**.

Example

```
import { useState } from "react";

function Form() {

 const [name, setName] = useState("");

 return (
   <form>
     <input
       type="text"
       value={name}
       onChange={(e)=>setName(e.target.value)}
     />
   </form>
 );
}
```

Diagram

```
Input Field
    |
    v
React State
    |
    v
Component UI
```

2.2 Uncontrolled Components

In uncontrolled components **DOM itself manages form data**.

React uses **refs** to access input values.

Example

```
import { useRef } from "react";

function Form(){
```

```
const inputRef = useRef();

const handleSubmit = () => {
  alert(inputRef.current.value);
};

return (
  <>
    <input type="text" ref={inputRef}/>
    <button onClick={handleSubmit}>Submit</button>
  </>
);
}
```

Controlled vs Uncontrolled

Feature	Controlled	Uncontrolled
Data handling	React State	DOM
Validation	Easy	Difficult
Use case	Complex forms	Simple forms

3. Form Validation

React Hook Form / Formik

Form validation ensures **correct user input**.

Example:

- Email format
- Password length

React Hook Form Example

```
import { useForm } from "react-hook-form";

function Login(){

 const { register, handleSubmit } = useForm();

 const onSubmit = data => console.log(data);

 return (
  <form onSubmit={handleSubmit(onSubmit)}>
    <input {...register("email")} />
```

```
    <button type="submit">Submit</button>
  </form>
 );
}
```

Formik Example

```
import { Formik, Form, Field } from "formik";

<Formik
 initialValues={{email:""}}
 onSubmit={values => console.log(values)}
>
<Form>
 <Field name="email" type="email"/>
 <button type="submit">Submit</button>
</Form>
</Formik>
```

4. API Communication

Fetch / Axios

React communicates with backend using **HTTP requests**.

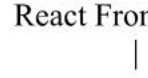
Fetch Example

```
useEffect(() => {

 fetch("https://api.example.com/users")
 .then(res => res.json())
 .then(data => console.log(data));

}, []);
```

Axios Example

```
import axios from "axios";

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

API Communication Diagram



| HTTP Request
v
Node.js Server
|
v
MongoDB Database

5. Connecting React → Express → MongoDB

This is a **full stack architecture**.

Architecture Diagram

User
|
v
React Frontend
|
| API Request
v
Express Server
|
| Query
v
MongoDB Database

Workflow

1 User submits form in React
2 React sends request using Axios
3 Express receives request
4 Server interacts with MongoDB
5 Data sent back to React

6. Routing using React Router

Routing allows **navigation between pages without page reload**.

Example pages

- Home
- About
- Contact

Installation

npm install react-router-dom

Example

```
import { BrowserRouter, Route, Routes } from "react-router-dom";

<BrowserRouter>
 <Routes>
   <Route path="/" element={<Home />} />
   <Route path="/about" element={<About />} />
 </Routes>
</BrowserRouter>
```

Routing Diagram

```
Browser URL
    |
    v
React Router
    |
    |---- /home
    |---- /about
    |---- /contact
```

7. State Management

Redux / Context API

Large applications need **global state management**.

Context API

Used to share data across components.

Example

Theme
User Login
Language

Example

```
const UserContext = createContext();
```

```
<UserContext.Provider value="Lakshmi">
  <Child />
</UserContext.Provider>
```

Redux

Redux manages **application state in a central store**.

Redux Flow

```
Component
   |
Dispatch Action
   |
Reducer
   |
Store Updated
   |
Component Re-render
```

8. Error Handling & Form Submission Feedback

Applications must handle errors properly.

Examples

- Invalid login
- Network error
- Server error

Example

```
try {

 const res = await axios.post("/login",data);

 alert("Login Successful");

}
catch(error){

 alert("Login Failed");

}
```

Feedback Types

- Success Message
- Error Message
- Loading Indicator

9. Deployment using Netlify / Render

After development, React apps are deployed to the internet.

Deploy using Netlify

Steps

1 Build React app

npm run build

2 Upload build folder to Netlify

3 Get live URL

Deploy using Render

Steps

1 Push project to GitHub
2 Connect GitHub with Render
3 Deploy automatically

Deployment Architecture

Developer
  |
  v
GitHub Repository
  |
  v
Netlify / Render
  |
  v
Live Website

Conclusion

React provides a **powerful framework for building modern web applications**.

Key advantages

- Component-based architecture
- Efficient state management
- Easy API integration
- Seamless frontend-backend connectivity

Using tools like **React Router, Redux, Axios, and Netlify**, developers can build **scalable and production-ready full-stack applications**.

Unit-4

Artificial Intelligence & Advanced Algorithms

1. Fundamentals of Artificial Intelligence

What is Artificial Intelligence?

Artificial Intelligence (AI) is a branch of computer science that focuses on creating machines that can **perform tasks that normally require human intelligence**.

These tasks include:

- Learning from data
- Recognizing patterns
- Solving problems
- Understanding language
- Making decisions

Example AI applications:

- Voice assistants
- Self-driving cars
- Chatbots
- Medical diagnosis systems
- Recommendation systems (Netflix, Amazon)

Goals of AI

The main goal of AI is to develop **intelligent systems** that can:

1. **Think like humans**
2. **Act like humans**
3. **Think rationally**
4. **Act rationally**

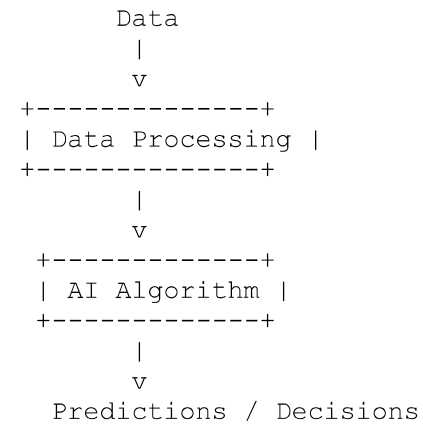
Components of AI Systems

An AI system usually consists of the following components:

- **Data**
- **Algorithms**

- **Learning model**
- **Decision system**

AI System Architecture



Types of Artificial Intelligence

1 Narrow AI

Designed for a **specific task**.

Examples:

- Siri
- Google Maps
- ChatGPT

2 General AI

A system that can perform **any intellectual task like humans**.

Currently **not fully developed**.

3 Super AI

Machines that are **more intelligent than humans**.

This is **theoretical**.

2. Machine Learning and Data-Driven Systems

What is Machine Learning?

Machine Learning (ML) is a subset of AI that allows computers to **learn from data without explicit programming**.

Instead of writing rules manually, the system **learns patterns from data**.

Example:

Email spam detection.

Types of Machine Learning

1 Supervised Learning

Training data contains **input and output labels**.

Example:

Predict house prices.

Algorithms:

- Linear Regression
- KNN

2 Unsupervised Learning

Data has **no labels**.

Goal: find patterns or groups.

Example:

Customer segmentation.

Algorithm:

- K-Means Clustering

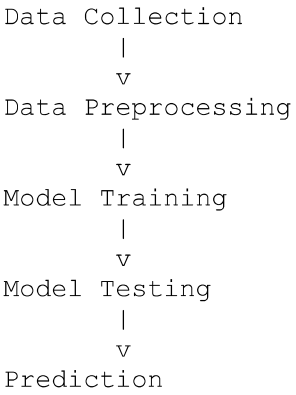
3 Reinforcement Learning

The system learns using **rewards and penalties**.

Example:

Game-playing AI.

Machine Learning Workflow



3. Classical Search Algorithms

Search algorithms help AI systems **find solutions by exploring possible states**.

Example:

Finding a path in a maze.

Breadth First Search (BFS)

Definition

Breadth First Search explores **nodes level by level**.

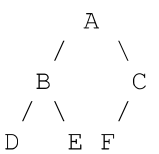
It visits all neighbors first before going deeper.

Data Structure Used

Queue (FIFO)

Example

Graph:



BFS traversal starting from A:

```
A → B → C → D → E → F
```

BFS Process

1. Start with root node
2. Add neighbors to queue
3. Visit nodes level by level

Depth First Search (DFS)

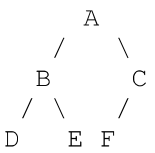
Definition

DFS explores **as deep as possible before backtracking**.

Data Structure Used

Stack (LIFO)

Example



DFS traversal:

```
A → B → D → E → C → F
```

A* Search Algorithm

Definition

A* is a **heuristic search algorithm** used to find the **shortest path**.

It uses:

```
f(n) = g(n) + h(n)
```

Where

- g(n) = cost from start to node
- h(n) = estimated cost to goal

Example

Path finding in maps (Google Maps).

A* Diagram

```
Start --- Node --- Node --- Goal
   g(n)      g(n)      g(n)

Heuristic estimate = h(n)
Total cost = f(n)
```

Hill Climbing Algorithm

Definition

Hill Climbing is a **local search algorithm** that moves toward the **best neighboring solution**.

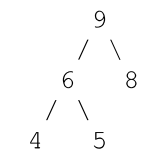
Idea

Always move to a **better state**.

Example:

Climbing a hill step by step until reaching the peak.

Example Graph



Algorithm moves:

```
4 → 6 → 9
```

Problem

It may get stuck in:

- Local maxima
- Plateaus

Best First Search

Definition

Best First Search chooses the **most promising node first using a heuristic function**.

Working

Priority Queue is used.

Nodes with **best heuristic value** are explored first.

Diagram

```
Start
  |
  v
Nodes evaluated using heuristic
  |
  v
Best node selected
```

4. Constraint Satisfaction Problems (CSP)

A Constraint Satisfaction Problem is a problem where we must **assign values to variables while satisfying constraints**.

CSP Components

1 Variables
2 Domains
3 Constraints

8 Puzzle Problem

The puzzle contains:

- 8 tiles
- 1 empty space

Goal: arrange tiles in correct order.

Example

Initial state

```
1 2 3
```

```
4 0 6
7 5 8
```

Goal state

```
1 2 3
4 5 6
7 8 0
```

AI searches possible moves to reach the goal.

N-Queens Problem

Goal:

Place **N queens on a chessboard** such that no queen attacks another.

Example (4-Queens)

Board representation:

```
Q . . .
. . Q .
. Q . .
. . . Q
```

Constraints:

- No same row
- No same column
- No diagonal attack

Water Jug Problem

Two jugs with fixed capacities.

Goal: measure a specific quantity of water.

Example

Jug A = 4L
Jug B = 3L

Goal: measure **2 liters**

Steps:

1 Fill 4L jug
2 Pour into 3L jug
3 Remaining = 1L
4 Repeat until 2L measured

5. Machine Learning Algorithms

Data Preprocessing

Data preprocessing prepares raw data for ML models.

Steps include:

1 Handling missing values
2 Removing duplicates
3 Normalization
4 Feature scaling

Preprocessing Workflow

```
Raw Data
   |
   v
Cleaning
   |
   v
Transformation
   |
   v
Prepared Dataset
```

Linear Regression

Linear Regression predicts a **continuous value**.

Example:

Predict house price.

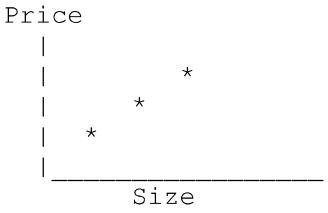
Equation:

```
y = mx + c
```

Where

- x = input variable
- y = predicted value

Example Graph



The line represents the **best fit line**.

K-Means Clustering

K-Means groups data into **K clusters**.

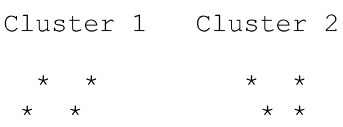
Example:

Customer segmentation.

K-Means Process

1 Choose number of clusters (K)
2 Assign data points to nearest cluster
3 Update cluster centers
4 Repeat until stable

Diagram



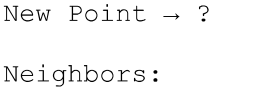
K-Nearest Neighbors (KNN)

KNN is used for **classification and regression**.

Idea:

A data point is classified based on its **nearest neighbors**.

Example



```
Class A
Class A
Class B

Result → Class A
```

6. AI Libraries

NumPy

NumPy is a Python library used for **numerical computations**.

Features:

- Arrays
- Matrix operations
- Mathematical functions

Example:

```
import numpy as np
a = np.array([1,2,3])
```

Pandas

Pandas is used for **data analysis and manipulation**.

Main structures:

- DataFrame
- Series

Example:

```
import pandas as pd
data = pd.read_csv("data.csv")
```

Scikit-Learn

Scikit-Learn is a machine learning library for Python.

Provides algorithms like:

- Linear Regression
- KNN
- K-Means

- Decision Trees

Example:

```
from sklearn.linear_model import LinearRegression
```

7. LangChain and OpenAI API

Prompt Engineering

Prompt engineering is the process of **designing effective prompts to guide AI models**.

Example prompt:

```
Explain BFS in simple words with example.
```

Good prompts produce **better responses**.

RAG (Retrieval Augmented Generation)

RAG combines:

- Retrieval of documents
- AI generation`

RAG Architecture

```
User Question
      |
      v
Document Search
      |
      v
Relevant Data
      |
      v
AI Model
      |
      v
Generated Answer
```

Example:

Chatbot using company documents.

8. Integrating AI with Full Stack

AI models can be integrated with **web applications**.

Architecture

```
Frontend (React)
      |
      v
Backend API (Flask)
      |
      v
AI Model (Python)
      |
      v
Response to User
```

Example Use Case

AI chatbot website.

Steps:

1 User enters question in React UI
2 Request sent to Flask API
3 AI model processes request
4 Response sent back to UI

Example Architecture Diagram

```
User
 |
 v
React Frontend
 |
 v
Flask Backend API
 |
 v
AI Model
 |
 v
Prediction / Response
```

Conclusion

Artificial Intelligence and advanced algorithms play a critical role in modern technology. Classical search algorithms help solve complex problems, while machine learning enables systems to learn from data. AI libraries simplify development, and modern frameworks like LangChain allow integration of AI into full-stack applications.

Unit-5

Python Implementation: Graphs, Data Analysis & Visualization

1. Python Review: OOP, File Handling and Modules

Introduction

Before implementing algorithms or performing data analysis in Python, it is important to understand some basic concepts such as:

- Object-Oriented Programming (OOP)
- File Handling
- Modules and Libraries

These concepts help in writing **organized, reusable, and scalable programs**.

Object-Oriented Programming (OOP)

What is OOP?

Object-Oriented Programming is a programming paradigm that organizes code using **objects and classes**.

A class is a blueprint, and objects are instances of that class.

Example:

- Class → Car
- Objects → Toyota Car, Honda Car

Main OOP Concepts

1. Class

A class defines attributes and functions.

Example:

```
class Student:
    def __init__(self,name,marks):
```

```
        self.name=name
        self.marks=marks
```

2. Object

An object is an instance of a class.

```
s1 = Student("Ravi",90)
```

3. Inheritance

Inheritance allows one class to **inherit properties from another class**.

Example:

```
class Animal:
    def speak(self):
        print("Animal speaks")

class Dog(Animal):
    pass
```

4. Encapsulation

Encapsulation means **hiding internal data and protecting it**.

Example:

```
class Bank:
    def __init__(self):
        self.__balance = 1000
```

OOP Structure

```
        Class
          |
          v
      Objects
   (Data + Methods)
```

File Handling in Python

What is File Handling?

File handling allows Python programs to **read and write data to files**.

Files help store information permanently.

Types of File Operations

1. Read
2. Write
3. Append

Opening a File

```
file = open("data.txt","r")
```

Modes:

Mode	Meaning
r	Read
w	Write
a	Append

Reading File

```
file.read()
```

Writing File

```
file.write("Hello Python")
```

File Handling Diagram

```
Program
   |
   v
Open File
   |
   v
Read / Write
   |
   v
Close File
```

Python Modules

What is a Module?

A module is a **Python file containing functions and variables**.

It helps organize code into reusable components.

Example:

```
math
numpy
pandas
```

Importing Modules

```
import math
print(math.sqrt(16))
```

Advantages

- Code reuse
- Easy maintenance
- Modular programming

2. Graph Algorithms in Python

Graphs are widely used in AI, networks, and route planning.

A graph contains:

- Nodes (vertices)
- Edges (connections)

Graph Representation

```
A ---- B
|      |
C ---- D
```

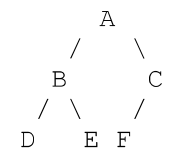
Breadth First Search (BFS)

Definition

BFS explores graph **level by level**.

It uses a **Queue data structure**.

Example Graph



BFS Traversal

```
A → B → C → D → E → F
```

Python Implementation

```
from collections import deque

graph = {
'A':['B','C'],
'B':['D','E'],
'C':['F'],
'D':[],
'E':[],
'F':[]
}

def bfs(start):
    visited=set()
    queue=deque([start])

    while queue:
        node=queue.popleft()
        if node not in visited:
            print(node)
            visited.add(node)
            queue.extend(graph[node])

bfs('A')
```

Depth First Search (DFS)

Definition

DFS explores nodes **deep first before backtracking**.

Uses **Stack or Recursion**.

DFS Traversal

```
A → B → D → E → C → F
```

Python Code

```
def dfs(node,visited):
    if node not in visited:
        print(node)
        visited.add(node)
```

```
        for neighbor in graph[node]:
            dfs(neighbor,visited)

visited=set()
dfs('A',visited)
```

Dijkstra Algorithm

Purpose

Dijkstra algorithm finds the **shortest path in weighted graphs**.

Used in:

- Google Maps
- Network routing

Example Graph

```
A --1-- B
|       |
4       2
|       |
C --3-- D
```

Algorithm Idea

1 Initialize distances
2 Pick smallest distance node
3 Update neighbors
4 Repeat

Python Example

```
import heapq

graph={
'A':{'B':1,'C':4},
'B':{'D':2},
'C':{'D':3},
'D':{}
}
```

Kruskal Algorithm

Purpose

Find **Minimum Spanning Tree (MST)**.

MST connects all nodes with **minimum total edge weight**.

Example

Edges

```
A-B (1)
B-C (2)
A-C (3)
```

MST

```
A ---- B ---- C
```

Total cost = 3

Prim Algorithm

Prim also finds **Minimum Spanning Tree**.

But it grows the tree **node by node**.

Prim Example

```
Start Node → Add smallest edge → Expand tree
```

Diagram

```
Start

 A
 |
 B
 |
 C
```

3. Data Extraction using PyMongo

Introduction

MongoDB is a **NoSQL database** used to store JSON-like documents.

Python can connect to MongoDB using **PyMongo library**.

Architecture

```
Python Program
```

```
      |
      v
   PyMongo
      |
      v
   MongoDB Database
```

Installing PyMongo

```
pip install pymongo
```

Connecting to Database

```
from pymongo import MongoClient

client=MongoClient("mongodb://localhost:27017")

db=client["studentdb"]
collection=db["students"]
```

Fetching Data

```
data=collection.find()

for i in data:
    print(i)
```

Example Document

```
{
"name":"Ravi",
"age":21,
"course":"CSE"
}
```

4. Pandas and NumPy for Data Analysis

NumPy

NumPy is used for **numerical computing and arrays**.

NumPy Array

```
import numpy as np

arr = np.array([1,2,3,4])
```

NumPy Structure

```
List → NumPy Array → Matrix Operations
```

Pandas

Pandas is used for **data analysis and manipulation**.

Main structures:

- Series
- DataFrame

DataFrame Example

```
Name    Age   Marks
Ravi     20     85
Priya    21     90
```

Loading Data

```
import pandas as pd

df=pd.read_csv("data.csv")
```

Data Cleaning

Cleaning removes incorrect data.

Steps:

- Remove null values
- Remove duplicates
- Format data

Example:

```
df.dropna()
```

Data Aggregation

Aggregation summarizes data.

Example:

```
Average Marks
Total Sales
Count of Users
```

Python example:

```
df.groupby("city").sum()
```

Statistical Analysis

Statistical functions:

```
mean()
median()
std()
max()
min()
```

Example:

```
df.mean()
```

5. Data Visualization: Matplotlib and Seaborn

Visualization helps understand data **visually**.

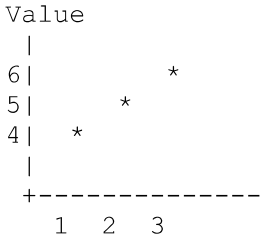
Matplotlib

Matplotlib is used to create **graphs and charts**.

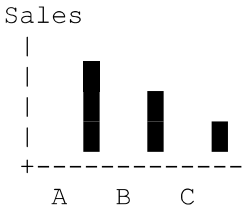
Line Graph Example

```
import matplotlib.pyplot as plt

x=[1,2,3]
y=[4,5,6]

plt.plot(x,y)
plt.show()
```

Line Graph



Bar Chart



Seaborn

Seaborn provides **advanced statistical visualization**.

Example

```
import seaborn as sns

sns.histplot(data=df["marks"])
```

Visualization of User Form Data

Example: Online form responses.

Fields:

- Age
- Course
- Marks

Charts can show:

- Average marks
- Course distribution
- Age distribution

6. Streamlit / Flask Dashboards

What is Streamlit?

Streamlit helps build **interactive data applications quickly**.

Streamlit Architecture

```
User Interface
     |
```

```
     v
Streamlit App
     |
     v
Python Data Analysis
```

Example Code

```
import streamlit as st

st.title("Student Data Dashboard")
st.write(df)
```

Flask for Data Applications

Flask is a **Python web framework** used to build APIs and web apps.

Flask Architecture

```
User Browser
      |
      v
Flask Server
      |
      v
Database / Data Analysis
```

Example

```
from flask import Flask

app=Flask(__name__)

@app.route("/")
def home():
    return "Welcome"

app.run()
```

Interactive Dashboard Example

```
User Form Data
      |
      v
MongoDB Database
      |
      v
Python Analysis
      |
      v
Dashboard (Streamlit)
```

7. Exploratory Data Analysis (EDA)

What is EDA?

EDA is the process of **analyzing datasets to understand patterns and insights**.

Before building machine learning models, we perform EDA.

Steps in EDA

1 Data collection
2 Data cleaning
3 Data visualization
4 Statistical analysis
5 Pattern identification

EDA Workflow

```
Dataset
  |
  v
Cleaning
  |
  v
Visualization
  |
  v
Statistical Analysis
  |
  v
Insights
```

Example EDA Questions

- What is the average age?
- Which course has most students?
- What is the highest mark?

Sample EDA Visualization

```
Students by Course

CSE
ECE
MECH
```

Conclusion

Python provides powerful tools for implementing graph algorithms, extracting data from databases, analyzing datasets, and creating visualizations. Libraries like NumPy, Pandas, Matplotlib, and Seaborn simplify data analysis, while frameworks like Streamlit and Flask enable development of interactive dashboards. Exploratory Data Analysis helps understand data patterns before applying machine learning techniques.