UNIT 3 - FRONTEND TECHNOLOGIES

What is Web Framework? A web framework is a software tool that provides a standard way to build and deploy web applications. It supports the development of web applications including web services, web resources and web API's.

Why Web Framework?

→ Faster development → organized code (MVC Pattern) → Database Integration
→ Reduce development time & cost. → Re-use of code / Handles Repetative tasks. → Secured
→ Easy deployment & maintainance

Web Framework Types

1) Server side framework - set of tools and libraries that run on a web server to handle business logic, database operations and API responses. Eg. Django (Python), Express.js (Node.js), Laravel (PHP), Spring (Java)

2) Client side framework - set of tools and libraries that run in the user's browser to build and manage the user interface of web application. Eg. React, Angular, Vue.js

Examples in detail

i) React (JS - Clientside)
React is a JS library (often treated as framework) for building user interfaces, especially single-page applications.
JSX Syntax - combines HTML and JS in same file for cleaner code.
Works with other libraries like Redux, React Router, etc.
UI is built using reusable components.

ii) Express.js (Node.js - Serverside)
Minimal & flexible web framework for Node.js used to build API and web applications.
Middleware support - Allows customization of request/response handling.
Clean and efficient way handle different HTTP routes.
Perfect for building API used by mobile apps or frontend.
Flexible structure.

iii) Vue.js (JS-Clientside)
Used to build interactive web interfaces & single page applications (SPA)
Easy to learn; Component Based; When your data changes, the UI automatically updates; Two way data binding; Built in Tools (CLI, Devtools, Router, etc); Works with REST API.

Model-View-Controller

MVC pattern is a software architecture pattern that divides an application into 3 interconnected components : the Model, the View & the Controller.

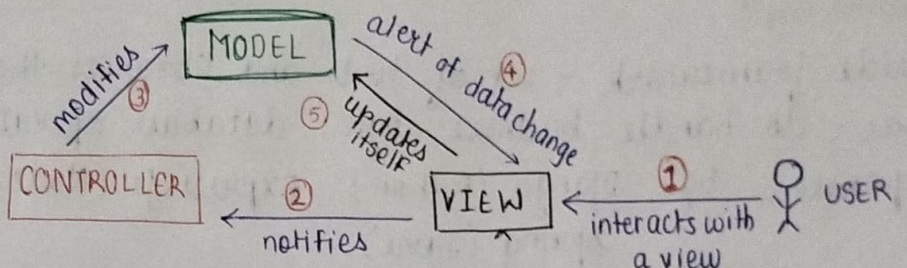
Cornerstone of modern software development (web applications).

Model - Manages the data & business logic of the application.

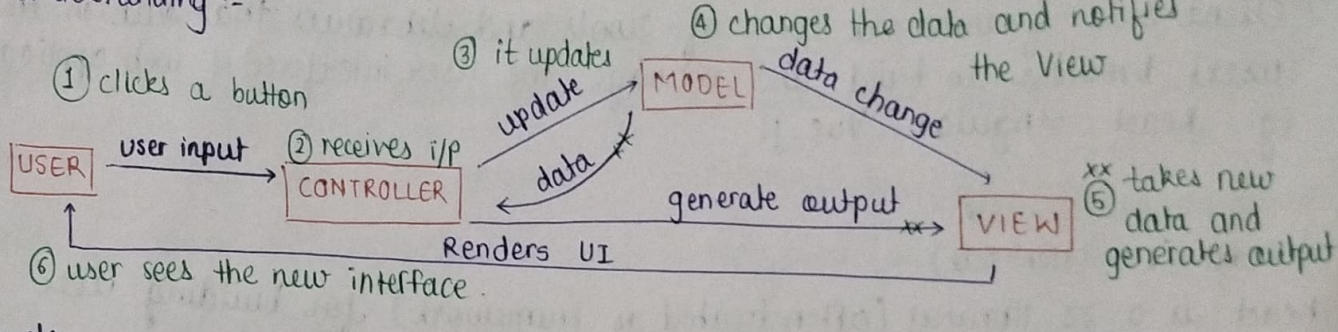
View - Responsible for displaying the data to the user.

Controller - Handles input from users, converting it to commands for the model or view. (ie updates accordingly)

From Book :-



Better understanding :-



Example :

1) Model : Stores student data & communicates with db.

```
Student = {
    name : "Alice",
    marks : 88
}
```

2) View : Shows data from model

```
<h1> Student : Alice </h1>
<p> Marks : 88 </p>
```

3) Controller

User clicks "Add Student"

Controller receives it, adds student to Model & tells View to update.

Advantages

- Easy code maintainance.
- Developers can work with 3 layers simultaneously (Model, View, Controller).
- Easy to Debug.
- Helps to avoid complexity by dividing an application into 3 units.
- search engine optimization friendly.
- MVC components can be tested separately from the user.

Disadvantages

- Complexity for small Projects
- Tight coupling between components (sometimes).
- Coordination is Required.

MVC in practical => Car driving mechanism

View = User interface (Gear, panel, steering, etc)

Controller = Mechanism (Engine)

Model = Storage (Petrol / Diesel tank)

MVC in framework => Django (Python), Laravel (PHP), Ruby on Rails (Ruby), Angular (JS), Express (JS)

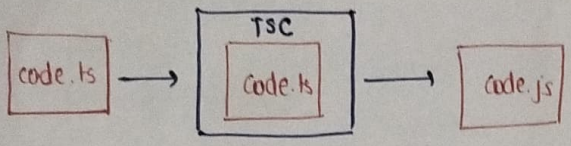
TypeScript - It is an open source, object oriented language developed and maintained by Microsoft.
Typescript is superset of JS. It takes the JS language and adds new features and advantages to it.

Typescript Code → Typescript compiler → Output JS code → JS runs in Browser

Browsers can't execute TS soooo.... TSC converts TS to JS code

Making JS easier to use by adding 'types'

finally runs!



Variables and Constants

[Static typing means that variable types are known and checked at compile time. TS enforces this by requiring you to declare type of a variable, func parameter or return valu

Variables in TS are statistically typed because of static typing.
Variables can be declared using var, let and const

1) Keyword	Scope	Reassignment	Usage Recommendation
let	Block-scope	Allowed	Preffered for variables
Const	Block-scope	Not Allowed	Preffered for constants
var	Function-scope	Allowed	Avoid using (Outdated)

eg:
```
let age : number = 25;
age = 30;
let isLoggedIn : boolean = true;
let username : string = "John";
```

```
const PI : number = 3.14;
const num : number = 100;
//num = 102; ✗
```

```
function testScope() {
if (true) {
var msg = "Im inside";
}
console.log(msg);
}
testScope();
```

'let' is preffered in modern Typescript because it offers block scoping, prevents redeclaration bugs, and promotes cleaner, more predictable code. 'var' is function-scoped, hoisted unsafely and is only used for legacy support.

Modules in Typescript

Modules allows to organize code into separate files and reuse functionality across different parts of an application.
A module can export variables, functions, classes or interfaces using the 'export' keyword, and import them using the 'Import' keyword.

Exporting from a Module (math.ts)

```
export function add(x: number, y: number): number {
  return x+y;
}
export const PI = 3.14;
```

Importing in another file (main.ts).

```
import { add, PI } from './math';
console.log(add(2,3));
Console.log(PI);
```

Angular Version 10+

- Angular is open source web application framework developed by Google.
- Angular 10+ is JS framework which is used to create single page application
- Angular applications are created with the Help of HTML and Typescript.
- It follows MVC architecture.

Angular CLI (Command Line Interface)

A powerful tool to create, build, serve and maintain Angular applications.

ng new project-name	ng serve	ng generate component my-component
(create new app)	(Run development server)	(Create component)

Angular Architecture

Building Blocks of Angular Architecture

1) Modules (NgModules) - a module is a container that groups components, services, directives and pipes. Every angular app has at least 1 root module. Declared using @NgModule decorator.
2) Components - basic building blocks of UI. Defined using @Component.
3) Templates - HTML views that define what to render in UI.
4) Services & Dependency Injection (DI) - Services hold business logic or reusable data (like API calls). Injected into components using Angular's built in DI system
5) Directives - Special makers in templates to manipulate DOM.
6) Metadata - tells how to process a class. Used to decorate the class.
7) Data binding - Communication betn parent & child components.

Event Binding - bind events to your app & respond to user input.

Property Binding - used to pass data from component class & facilates the interpolate values.

Angular Project Structure.

```
src/
├─ app/
│  ├─ app.module.ts              // Root module
│  ├─ app.component.ts           // Root component logic
│  ├─ app.component.html         // Component template
│  ├─ services/                  // Reusable services
│  ├─ components/                // All other components
├─ assets/                       // Static files like images
├─ environments/                 // Environment specific settings
```

Angular Lifecycle Hooks:

constructor
ngOnChanges — (1)
ngOnInit — (2)
ngDoCheck — (3)
(4) — ngAfterContentInit
(5) — ngAfterContentChecked
(6) — ngAfterViewInit
(7) — ngAfterViewChecked
ngOnDestroy — (8)

Lifecycle Hooks are special methods that are executed at specific points during the lifecycle of a component or directive.

(1) Called whenever an input property of component or directive changes. It provides the previous & current value of input as arguments.

(2) called immediately after a component or directive is initialized.

(3) Called during every change in detection cycle (ie every time when the input properties of components are checked). It allows developers to perform their own change detection & make updates to the component or directive as needed.

(4) called after a component's / directive's content has been initialized. This method executes only for the first time when all the bindings of the component need to be checked for the first time.

(5) called after content of a component / directive has been checked.

(6) called after a component's / directive's view has been initialized.

(7) called after the view of component has been checked.

(8) called just before a component is destroyed. It is good place to perform any cleanup or tear-down tasks before component is removed from the DOM.

Angular Modules

- Modules are containers that group related code together - including components, directives, pipes and services - into cohesive units.
- They help in organizing an application and enable lazy loading, dependency injection and feature separation.

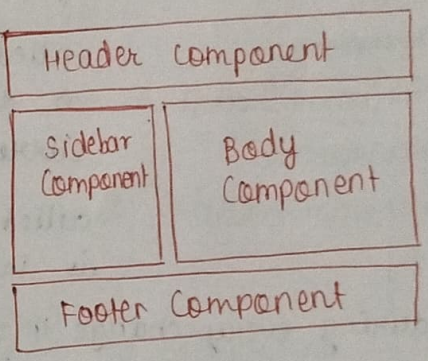
Features: Grouping Related code, Lazy Loading (load on demand), Reusability.

Types of Angular Modules:

1) Root Module - Every Angular application has atleast one root module, typically named as AppModule, which bootstraps the application.
2) Feature Module - These encapsulate specific features or sections of the application, such as authentication or user management.
3) Shared Module - Used to share common components, directives & pipes across other modules.
4) Core Module - Contains singleton services & application-wide providers.

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

@NgModule({
  imports:[...],
  declarations: [...],
  bootstrap:[...]
  })
  export class AppModule{}
```



Angular Components

Components are the building blocks of the UI. Each component controls a part of screen, known as view and contains logic, template & styling

Angular Component = HTML Template + Component Class + Component Metadata

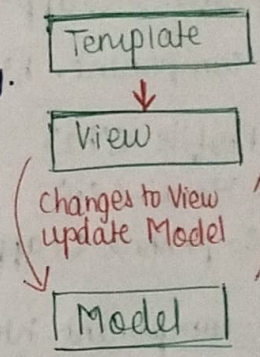
1) Template - Defines HTML view presented to the user.
2) Class - Written in Typescript, this contains the logic & data of the component.
3) Metadata - Provided via @Component decorator, it tells how to process the component, including selector, template & style files.

Example with multiple components:

```
AppComponent // Root of appln
├ HeaderComponent // Displays the website title
└ UserProfileComponent // show user details
    └ ProfilePhotoComponent. // Display user's photo
```

Data Binding

Refers to the mechanism that connects the component's logic (Typescript) with the view (HTML).



One way Data Binding : data flows in 1 direction only.

1) Interpolation - binds data from the component to the view.

Syntax : {{expression}}

eg : `<h1> {{ title }} </h1>`

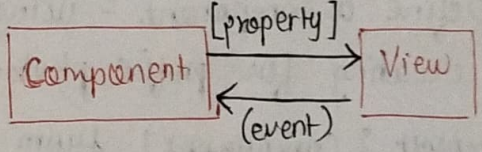
Usecase : displaying dynamic values in the template.

2) Property binding - one way from component to view. It updates the value of element's property on the page. (ie bind DOM property to component property)

Syntax : [property] = "expression"

eg :

```
<img [src] = "imgPath" alt = "Dynamic Image" />
export class AppComponent {
  imgPath = " path.jpg";
}
```



It dynamically updates DOM values.
More powerful & secure than plain HTML attribute interpolation.
Allows boolean values (True / False).

3) Event binding - one way from template (view) to component. It is used to listen to DOM events (like click, keyup, submit) and call methods in the component when the event occurs.

Syntax : (event) = " handler ($event)" or (event) = " methodname ()"

eg :

```
<button (click) = " onButtonClick ()" > Click Me </button>
export class AppComponent {
  onButtonClick () {
    alert ('Button clicked!');
  }
}
```

Usecase : Handling user interactions like click, submit, etc.

Two way Data Binding : component & view in sync with both direction

Syntax : [(ngModel)] = "property"

eg : `<input [(ngModel)] = " username" >`

app.component.html

```
<app-header></app-header>
<app-user-profile></app-user-profile>
```

user-profile-component.html

```
<h2> User Profile </h2>
<p> Name: Hani </p>
<app-profile-photo></app-profile-photo>
```

profile-photo-component.html

```
<img src = "address" alt = "User Photo">
```

How to create & use components in Angular?

1) Create component using Angular CLI

ng generate component component-name eg. ng g c header

2) Define a component - using decorator @Component

eg. creating for profile-photo-c.ts

```
import { Component } from '@angular/core';
@Component ({
  selector : 'app-profile-photo',
  template : '<img src="address" alt="User photo">',
  styles : ['img{border-radius : 50%; width : 100px;}']
})
export class ProfilePhotoComponent { }
```

3) Register component in module

```
@NgModule ({
  declarations : [
  AppComponent,
  HeaderComponent,
  UserProfileComponent,
  ProfilePhotoComponent
  ],
  imports : [BrowserModule],
  bootstrap : [AppComponent]
})
export class AppModule { }
```

4) Use component in Template

```
<app-profile-photo>
</app-profile-photo>
```

Directives and Pipes

Directives - These are classes that adds behaviour to elements in your applications, allowing you to manipulate the DOM and enhance user interactions.

There types of directives:

1) **Component Directives** - Angular Components are specialized directives.
- Directives are templated. It directs how the component should be initialized & processed at runtime and how it should be used.
- The component Directive is used to specify HTML ~~elemen~~ templates. It has the structure design & working order of how component should be.
- Defined using @Component decorator.

- eg: html:

```
<app-user-profile></app-user-profile>
```

ts:

```
@Component({
  selector: 'app-user-profile',
  template: <h2> User Info </h2>
})
export class UserProfileComponent{
}
```

2) **Structural Directives** :-
- Used to change the structure of the DOM, such as adding or removing elements.
- Recognized by asterisk (*) prefix in templates.
- Common builtin structural directives:
  - *ngIf - conditionally includes or excludes an element.
  - *ngFor - Repeats element for each item in a list.
  - *ngSwitch - Switches among multiple possible elements based on condition.

- eg:

```
<div *ngIf="isVisible"> Visible Content </div>
<li *ngFor="let the item of items"> {{ item }} </li>
```

- Structural directives are essential for dynamic layouts & conditional rendering.

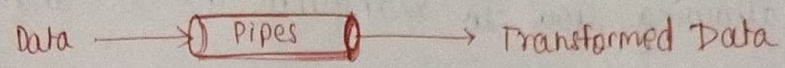
3) **Attribute Directives** - Change the appearance / behaviour of DOM elements, components, etc.
- Applied as attribute to elements.
- Common builtin attribute directives:
  - ngClass - Adds / removes CSS classes.
  - ngStyle - Adds / removes inline styles.
  - ngModel - Enables two-way data binding for form elements.

eg.

```
<div [ngClass]="{'active':isActive}">
</div>
<input [ngModel]="userName">
```

- Useful for dynamic styling, class toggling and form control.

Pipes - Simple functions that transform data before it is displayed in the view. They are used directly in template expressions, allowing you to format, filter or transform values without changing underlying data.



Syntax : Pipes are used with pipe (|) symbol in templates.

```
{{ value | pipeName : arg1 : arg2 }}
```

Purpose : They change how data appears to the user, such as formatting dates, numbers, or text, but do not alter actual data.

Chaining : Multiple pipes can be chained together

```
{{ birthday | date : 'longDate' | uppercase }}
```

Arguments : Pipes can accept arguments, separated by colons.

```
{{ amount | currency : 'INR' : 'symbol' }}
```

Types of Pipes :

1) Built-in Pipes - Provided by Angular.
data, uppercase / lowercase, currency, percent, json, etc.

eg. ts

```
export class AppComponent {
name: string = 'homi';
today: Date = new Date();
price : number = 2500;
}
```

html

```
<p> Uppercase : {{ name | uppercase }} </p>
<p> Lowercase : {{ name | lowercase }} </p>
<p> TitleCase : {{ name | titlecase }} </p>
<p> Today's Date : {{ today | date : 'fullDate' }} </p>
<p> Price : {{ price | currency : 'USD' }} </p>
```

Output :

Uppercase : HOMI
Lowercase : homi
TitleCase : Homi
Today's Date : Saturday, May 10, 2025
Price : $2,500.0

2) Custom Pipes - Created by developers for specific transformation needs.

- Create a Typescript class.
- Add @Pipe decorator with a name.
- Implement the PipeTransform interface with transform() method.
- Register the pipe in your module's declarations.

eg. Step 1 : Create the Pipe (eg KebabCase)

```
ng generate pipe pipename
```

Step 2 : Write the Pipe class.

```
import { Pipe, PipeTransform } from '@angular/core';
@Pipe ({
   name : 'kebabCase',
   })
export class KebabCasePipe implements PipeTransform {
 transform (value: string) : string {
  return value.toLowerCase().replace (//g, '-');
 }
}
```

Step 3 : Register the Pipe

```
import { KebabCasePipe } from './kebab-case.pipe';
@NgModule ({
   declarations: [ KebabCase Pipe
                  //other pipes / components
                 ],
   // imports or providers or bootstrap
   })
   export class AppModule { }
```

Step 4 : Use Pipe in Template

```
<p> {{ "Hello Angular Pipes" | KebabCase }} </p>
```

Output : hello-angular-pipes

Angular Services

- Special classes used to organize & share code, data or functionality across multiple components in your Angular application.
- Usually, a service is a typescript class, decorated with @Injectable()
- We use Dependency injection to provide service to components that need it.

eg : Creating a Service

```
import { Injectable } from '@angular/core';
@Injectable ({
 providedIn : 'root'
  })
export class DataService {
  getData () {
  return ['Apple', 'Cherry', 'Mango'];
  }
}
```

	Component	Service
→	Display & manage UI	Share logic / data across app
→	@Component()	@Injectable()
→	Multiple instances are needed	Singleton (one instance)
→	Eg use : Rendering views, handling events	Eg use : API calls, sharing data, business logic.

Dependency Injections

It is a design pattern and core feature in Angular that allows components, services and other classes to receive the resources and services they need from an external source, rather than creating them directly. This makes your code more modular, flexible and easier to test.

Working:

Key Concept - Dependency: An object that another relies on.

Injection - Providing that dependency from outside rather than letting the class create it.

Why we need it; Improves testability, promotes reusability, makes code more flexible and maintainable.

Working:

1: Create a service

```
import { Injectable } from '@angular/core';
@Injectable({
    providedIn: 'root'})
export class UserService {
  getUser() {
   return{name : 'Alice', age: 25};
  }
}
```

2: Inject ~~into~~ service into a component

```
import {Component } from '@angular/core';
import {UserService} from './user.service';
@component ({ })
export class UserComponent {
   user : any;
// Angular injects the UserService here
constructor(private UserService : UserService) {
    this.user = this.userService.getUser();
}
}
```

Visual Summary:

1) You write `constructor(private myService: MyService) { }`

2) Angular does: ↓
   - → finds / creates an instance of MyService
   - → passes it into your component automatically

Angular JS and its Features.

It is an open source, JS based frontend framework developed by Google and Released in 2010.

Features:

1) MVC architecture
2) Two-way Data Binding
3) DI
4) Templates
5) Testing Support
6) Plain JS
7) Component Reusability (Build once, use multiple times)

React JS

Introduction to React.js

- React.js is a popular JS library developed by facebook for building user interfaces, especially single-page applications.
- It allows developers to create large web applications that can update and render efficiently in response to data changes.

React Components

React component is reusable, independent piece of UI. Components can be of 2 types : i) Functional Components : Simple JS functions that return JSX
ii) Class Components : ES6 classes that extend React.component and have a render method.

```
function Welcome (props) {
    return <h1> Hello , {props.name} </h1>;
}
```

Inter Component Communication

Components in React communicate mainly through props (properties):
Parent to Child : Data is passed from parent to child using props.
Child to Parent : Child components can communicate with parents by calling callback functions passed as props.
Sibling to Sibling : Usually done via their common parent.

Component Styling

Inline Styling - using style attribute with JS objects
CSS Styling - importing .css file
CSS Modules - Scoped css to specific components.
Styled components - using third party libraries for component-level styling

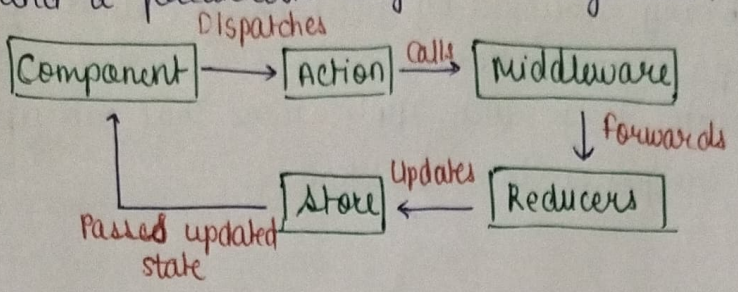
Routing

Routing in React is managed using libraries like React Router. It allows navigation between different components / pages without reloading the page

eg. `import { BrowserRouter, Route, Link } from 'react-router-dom';`

Redux Architecture

- Redux is a state management library used with React. It provides centralized store and a predictable way to manage data.



1) Store - holds the application state.
   created using createStore(reducer).
2) Action - Plain JS objects describing what happened.
   Must have a type property. eg. {type: 'INCREMENT'}
3) Reducers - Pure functions that take the previous state & an action & returns the next state.

eg.
```
function counter(state=0, action){
    switch(action.type){
        case 'INCREMENT':
            return state+1;
        default:
            return state;
    }
}
```

4) Dispatch - method used to send actions to the store.
5) Selectors - functions to extract specific pieces of state from the store.

User Action → Dispatch(Action) → Reducer → Update Store → React Component

Hooks

Hooks are special function that let you use state and other react features in functional components, without writing class components.

1) useState() Hook:
   Purpose - Adds state to functional components.
   How - Returns a state variable and a function to update it.

eg.
```
import React, {useState} from 'react';
function Counter(){
    const [count, setCount] = useState(0); // count is state, setCount updates it.
    return(
    <div>
        <p> count: {count} </p>
```

```
        <button onClick={() => setCount(count+1)}>Increment</button>
      </div>
    );
  }
```

2) useEffect() Hook:

Purpose: Performs side effects in components (eg. data fetching, subscriptions, manually changing the DOM)

How: Takes a function (effect) and an optional dependency way.

eg.
```
import React, { useState, useEffect } from 'react';
function Example(){
    const [count, setCount] = useState(0);

    useEffect(() => {
        document.title = "You clicked ${count} times";
    }, [count]);

    return (
     <div>
        <p> You clicked {count} times </p>
        <button onClick = {() => setCount(count+1)}> Click me </button>
     </div>
    );
}
```

3) useContext() Hook:

Purpose: lets you use values from a React Context without passing props through every level.

How: You create a context, provide a value, and useContext to access it in any child component.

eg.
```
import React, { createContext, useContext } from 'react';
const MyContext = createContext();
function Parent(){
  return (
      <MyContext.Provider value={{message: 'Hello, world!'}}>
      <Child />
      </MyContext.Provider>
  );
}
function Child(){
   const {message} = useContext(MyContext);
   return <p> {message} </p>;
}
```