Contents

1.1 Angular Recap & Project Setup:

- Angular is a **TypeScript-based open-source front-end framework** developed by **Google** for building **single-page applications (SPAs)**.
- It follows a **component-based architecture** and provides powerful tools for routing, state management, and dependency injection.

1.1.1 Brief Recap of Angular 17 Core Concepts:

1. Components:

- A **component** is the **basic building block** of an Angular application.
- Each component controls a **part of the UI** and consists of:
  - HTML template (View)
  - TypeScript class (Logic)
  - CSS styles (Design)
- **Creating a component:**
  - ng g c CompName

    here, 'g' is for generate and 'c' is for component

Component Structure:

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

@Component({
  selector: 'app-login',
  standalone: true,
  imports: [],
  templateUrl: './login.html',
  styleUrls: ['./login.css']
})
export class AppComponent {
  title = 'My Angular App';
}
```

Key parts:

Part	Description
@Component	Decorator that defines metadata
selector	Custom HTML tag used to display component
templateUrl	HTML file linked to the component
styleUrls	CSS files for component styling
class	Contains application logic

Component Life Cycle:

Hook	Purpose
ngOnInit()	Runs when component loads
ngOnDestroy()	Cleanup before component is destroyed
ngOnChanges()	Runs when input data changes

2. Services

- A **service** is used to **share data and business logic** across multiple components.
- Examples: API calls, Authentication, Logging, Data storage
- Why Use Services?
    - Code reusability
    - Separation of concerns
    - Cleaner components
    - Easy testing
- **Creating a service:**
    - ng generate service user

Example:

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

@Injectable({
  providedIn: 'root'
})
export class UserService {
  getUsers() {
    return ['Alex', 'John', 'Vinny'];
  }
}
```

3. Routing

- Routing allows navigation between different views/pages **without reloading** the application.
- Angular uses **client-side routing**.
- Why Routing is Important?
    - Enables SPA behavior
    - Improves user experience
    - Manages application navigation

Basic routing concept:

```
import { Routes } from '@angular/router';

export const routes: Routes = [
  { path: ' ', component: HomeComponent },
  { path: 'login', component: LoginComponent },
  { path: 'dashboard', component: DashboardComponent }
];
```

Router Outlet

- Used to load routed components dynamically

```
<router-outlet></router-outlet>
```

Navigation Using RouterLink

```
<a routerLink="/login">Login</a>
```

OR

```
constructor(private router: Router) {}
this.router.navigate(['/dashboard']);
```

1.1.2 Setting up a scalable Angular project using Angular CLI with Standalone Component

Step-1: Download latest version of node js

[https://nodejs.org/en/download]

Step-2: Install the downloaded .msi file

Step-3: Open node.js command prompt and check the version **[node -v]**

Step-4: Go to your folder location **[Ex: cd DemoProject]**

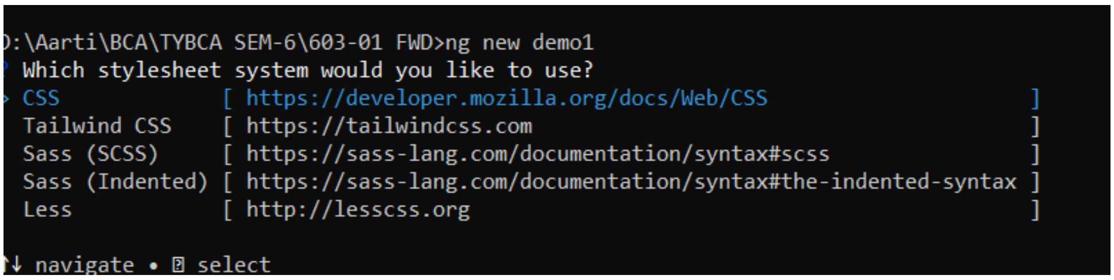
Step-5: Install latest angular CLI **[npm i -g @angular/cli@latest]**

Step-6: After the successful installation, check the version **[ng version]**

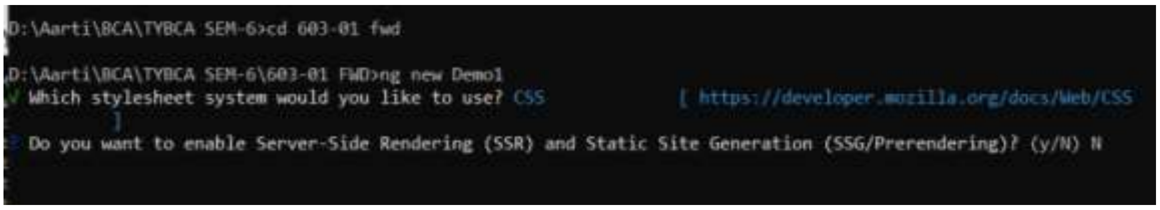
Step-7: Now create a new angular app

[ng new DemoApp1] OR [ng new Demo1 --no-standalone]

Step-8: Select any one from the below given options and press 'N'

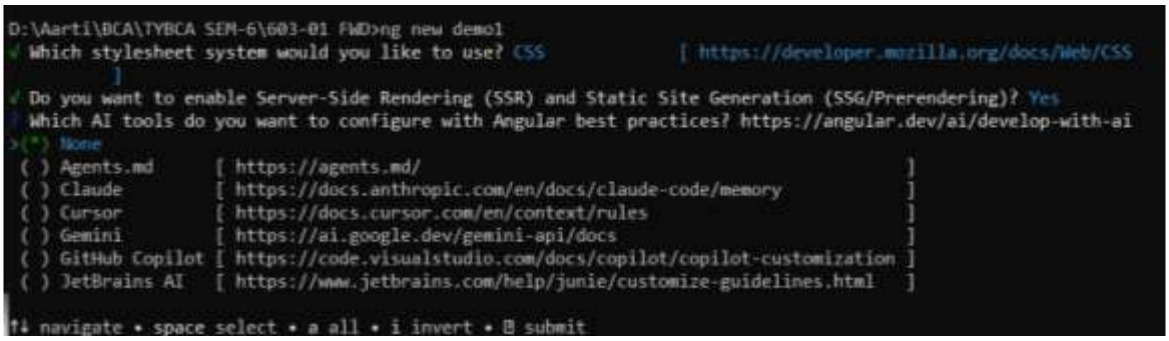


```
D:\Aarti\BCA\TYBCA SEM-6\603-01 FWD>ng new demo1
? Which stylesheet system would you like to use?
> CSS             [ https://developer.mozilla.org/docs/Web/CSS                     ]
  Tailwind CSS    [ https://tailwindcss.com                                        ]
  Sass (SCSS)     [ https://sass-lang.com/documentation/syntax#scss                ]
  Sass (Indented) [ https://sass-lang.com/documentation/syntax#the-indented-syntax ]
  Less            [ http://lesscss.org                                             ]

↑↓ navigate • ⏎ select
```



```
D:\Aarti\BCA\TYBCA SEM-6>cd 603-01 fwd

D:\Aarti\BCA\TYBCA SEM-6\603-01 FWD>ng new Demo1
✓ Which stylesheet system would you like to use? CSS             [ https://developer.mozilla.org/docs/Web/CSS
     ]
? Do you want to enable Server-Side Rendering (SSR) and Static Site Generation (SSG/Prerendering)? (y/N) N
```

Step-9: Select any 1 option from the below **[Ex: 'None' in this case]**



```
D:\Aarti\BCA\TYBCA SEM-6\603-01 FWD>ng new demo1
✓ Which stylesheet system would you like to use? CSS             [ https://developer.mozilla.org/docs/Web/CSS
     ]
✓ Do you want to enable Server-Side Rendering (SSR) and Static Site Generation (SSG/Prerendering)? Yes
? Which AI tools do you want to configure with Angular best practices? https://angular.dev/ai/develop-with-ai
>(*) None
 ( ) Agents.md      [ https://agents.md/                                          ]
 ( ) Claude         [ https://docs.anthropic.com/en/docs/claude-code/memory       ]
 ( ) Cursor         [ https://docs.cursor.com/en/context/rules                    ]
 ( ) Gemini         [ https://ai.google.dev/gemini-api/docs                       ]
 ( ) GitHub Copilot [ https://code.visualstudio.com/docs/copilot/copilot-customization ]
 ( ) JetBrains AI   [ https://www.jetbrains.com/help/junie/customize-guidelines.html  ]

↑↓ navigate • space select • a all • i invert • ⏎ submit
```

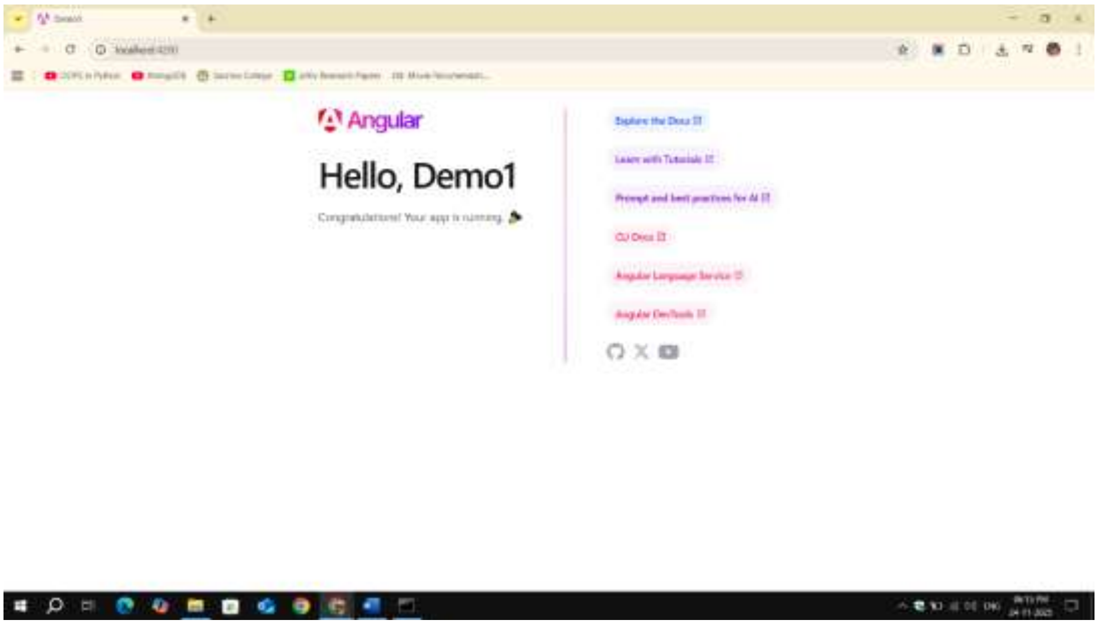
Step-10: Move to the created app's folder **[cd Demo1]**

Step-11: Run the app on the server

[ng serve] - this will the show the localhost port on which your app is running [Ex: http://localhost:4200/] – open this in your browser

OR

[ng serve --open] – this will run the app on localhost and open it as well



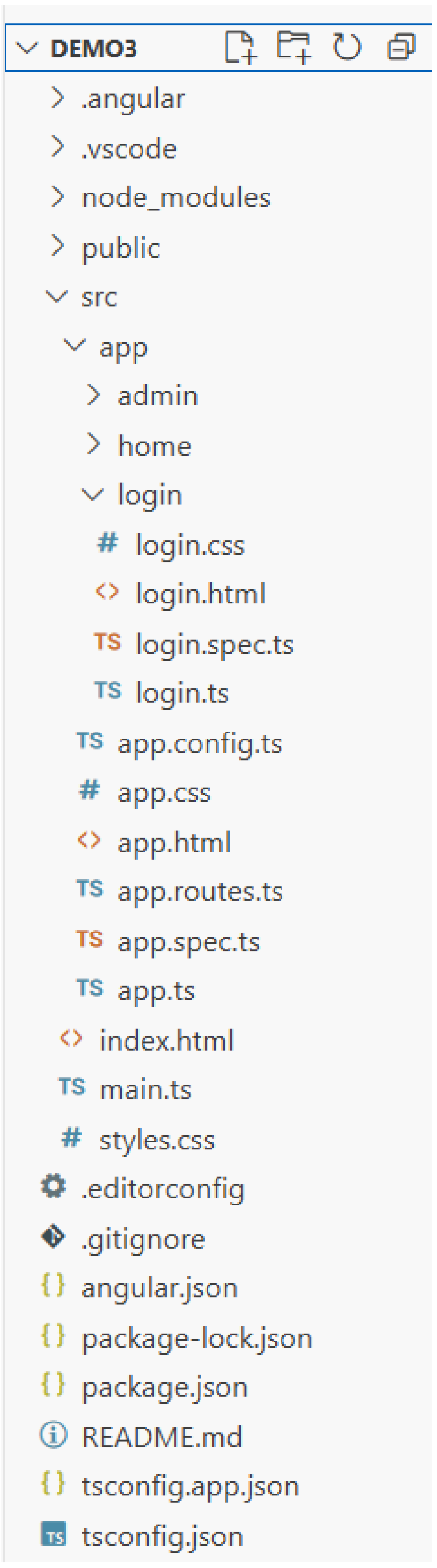
1.1.3 Folder Structure and Module Organization for Large Projects

- As Angular applications grow, maintaining a **well-organized folder structure** becomes critical for:
  - Scalability
  - Maintainability
  - Team collaboration
  - Faster debugging and feature development

Why Folder Structure Matters in Large Projects

- Improves code readability
- Reduces duplication
- Enables parallel team work
- Supports lazy loading
- Makes testing easier

- This is the directory structure you get when you create an angular app.



```
DEMO3
  > .angular
  > .vscode
  > node_modules
  > public
  v src
    v app
      > admin
      > home
      v login
        # login.css
        <> login.html
        TS login.spec.ts
        TS login.ts
      TS app.config.ts
      # app.css
      <> app.html
      TS app.routes.ts
      TS app.spec.ts
      TS app.ts
    <> index.html
    TS main.ts
    # styles.css
  .editorconfig
  .gitignore
  {} angular.json
  {} package-lock.json
  {} package.json
  README.md
  {} tsconfig.app.json
  tsconfig.json
```

1) .angular/

- Internal Angular cache folder
- Used by Angular CLI for faster builds
- ✕ Do not edit manually

2) .vscode/

- VS Code editor configuration
- Contains:
- Recommended extensions
- Debug settings
- Helpful for team consistency

3) node_modules/

- Contains all installed npm packages
- Created using:

```
npm install
```

- Never modify manually
- Not uploaded to GitHub

4) public/

- Stores **static public assets**
- Files here are served **as-is**
- Example use: Favicon, Static images, Public JSON files

5) src/ – Main Application Code

- This folder contains **all application source code**.

6) src/app/ – Application Logic

- This is the **heart of the Angular application**.

7) admin/

- Intended for **Admin-related features**
- Example components:
    - Admin dashboard
    - User management

- Follows **feature-based organization**
- Good practice for large projects

8) login/

- This is a **complete feature folder** for login functionality.
- This structure shows **component-based separation**

```
login/
├── login.css
├── login.html
├── login.spec.ts
└── login.ts
```

- `login.ts` → Login component logic
- `login.html` → UI template
- `login.css` → Component styles
- `login.spec.ts` → Unit testing file

Other Important Files in `src/`

1. index.html

- Main HTML file
- Angular app loads inside: **<app-root></app-root>**

2. main.ts

- Application entry point
- Bootstraps the app
  bootstrapApplication(AppComponent, appConfig);
- Replaces `main.module.ts`

3. styles.css

- Global styles
- Applies to entire application
- Good place for:
  - Theme styles
  - Bootstrap import

Configuration Files (Root Level)

1) angular.json

- Angular CLI configuration

- Controls:
    - Build settings
    - Assets
    - Styles

2) package.json

- Project dependencies
- Scripts like:
    - `ng serve`
    - `ng build`

3) package-lock.json

- Exact dependency versions
- Ensures consistency across systems

4) tsconfig.json & tsconfig.app.json

- TypeScript compiler settings
- Controls strictness and paths

5) .editorconfig

- Maintains coding style
- Indentation, line endings, etc.

6) .gitignore

- Files excluded from Git
- Example:
    - `node_modules`
    - build files

1.2 Advanced Routing & State Handling

Angular provides powerful mechanisms for routing and state management, allowing developers to build scalable, maintainable, and performant applications. Beyond basic routing, advanced routing techniques like lazy loading, route guards, and resolvers, along with reactive state management using RxJS, help optimize applications.

1.2.1 Implementing Lazy Loading with Feature Modules

Lazy Loading is a design pattern in Angular where feature modules are loaded **on demand**, rather than loading everything when the application starts. This reduces the initial load time and improves performance, especially for large applications.

- **Feature Modules**: Angular modules dedicated to a specific feature (e.g., `UserModule`, `AdminModule`).
- **Purpose**: Only load modules when the user navigates to a route that requires them.

How it works:

1. Create a feature module using `ng generate module moduleName --route routeName --module app.module`.
2. Define the routes inside the feature module using `RouterModule.forChild()`.
3. Configure the `AppRoutingModule` to load the module lazily with `loadChildren`.

Benefits:

- Faster initial load
- Better resource utilization
- Scales well for large applications

1.2.2 Route Guards: CanActivate, CanDeactivate for Securing Routes

Route guards allow developers to **control access to routes**. Angular provides multiple types of guards:

1. **CanActivate**: Determines if a route **can be accessed**.
2. **CanDeactivate**: Determines if the user **can leave the current route**.
3. **Other guards**: `CanLoad`, `Resolve`, `CanActivateChild`.

Benefits:

- Enhances security
- Improves user experience
- Prevents unwanted navigation

1.2.3 Route Resolvers for Preloading Data

Route Resolvers allow Angular to fetch data before a route is activated, ensuring components have the necessary data at the moment they are loaded.

- **Use Case**: Fetching a user profile or a list of items before displaying the page.
- **How it works**: Implement the `Resolve` interface, return an observable, and configure it in the route.

Benefits:

- Avoids loading components with empty or incomplete data
- Improves user experience
- Can handle errors before component load

1.2.4 Introduction to Advaned State Handling using RxJS Subjects and BehaviorSubjects

State management in Angular involves keeping track of application data and updating the UI reactively. RxJS provides **Subjects** and **BehaviorSubjects** for reactive state handling.

Subjects

- Special type of Observable
- Acts as both **observable and observer**
- Can **emit new values** to multiple subscribers

BehaviorSubjects

- Like a Subject but stores the latest value
- New subscribers immediately receive the current value

Use Case in Angular:

- Centralized state management
- Sharing data between components
- Reacting to asynchronous data changes (e.g., user login status)

1.3 Reactive Forms in Real Applications

Reactive Forms in Angular are **model-driven forms**, meaning the form structure, validation, and state are defined **in the component class**, not in the template. They are ideal for complex forms in real-world applications because they provide **better control, scalability, and testability** compared to template-driven forms.
Reactive forms rely on **FormControl**, **FormGroup**, and **FormArray**, along with **Observables** to manage form state and reactively respond to user input.

1.3.1 Dynamic Form Generation using FormArray

- FormArray is a special type of form control that holds an array of FormControls, FormGroups, or even other FormArrays.
- It allows dynamic creation of forms where the number of fields is not known at compile time.
- Common real-world use cases:
  - Adding multiple addresses or contacts dynamically.
  - Repeating sections in a form (e.g., order items, tasks, questions in a survey).
- FormArray provides methods to add, remove, or insert controls dynamically.

Key point: It enables forms to be **flexible and data-driven**, adapting to the user's requirements.

1.3.2 Custom Validators and Asynchronous Validation

- **Custom Validators**: Functions that enforce **business-specific rules** beyond built-in validators (like required, minLength, etc.).
  - They allow checking conditions like password strength, date ranges, or specific patterns.
- Asynchronous Validators: Validators that perform asynchronous checks, such as:
  - Checking whether a username or email is already registered by querying an API.

    - Performing server-side validations without blocking the UI.

Key point: Validators (synchronous or asynchronous) help ensure **data integrity and correctness** before form submission.

1.3.3 Centralized Error Handling and Displaying Validation Messages

- In real applications, forms often have many fields with different validation rules.
- Centralized error handling involves defining all validation messages in one place, rather than scattered throughout the template.
- Benefits:
    - Simplifies management of error messages.
    - Ensures consistency across forms.
    - Improves maintainability as validation rules evolve.
- This is often combined with reactive subscriptions to **form status or value changes** to dynamically show/hide messages.

Key point: Centralized error handling improves user experience and developer productivity.

1.3.4 Submitting Forms to APIs and Form State Management

- **Form Submission** in real applications usually involves sending data to a backend API.
- **Reactive forms** provide:
    - **Form state management**: Tracks `valid`, `invalid`, `pristine`, `dirty`, `touched`, and `untouched` states.
    - **Conditional submission**: Forms are submitted only when `valid`.
    - **Reactive updates**: Allows disabling submit buttons or showing loading indicators based on form state.

Form State	Purpose
pristine	Form not modified

dirty	User changed values
touched	Field visited
untouched	Field not visited
pending	Async validation in progress
disabled	Control not editable

- Real applications may also handle:
    - Transforming form data into the required API payload.
    - Resetting or patching forms after submission.
    - Handling API errors and updating form state accordingly.

Key point: Reactive Forms provide **full control over form lifecycle**, enabling seamless integration with backend APIs and sophisticated form state handling.

1.4 Building Reusable UI Components & Design Patterns

Modern Angular applications rely on reusable UI components and clean design patterns to ensure scalability, maintainability, and performance. Angular 17 provides powerful tools to create flexible, reusable, and well-structured components.

1.4.1 Creating Reusable Card, Modal, and Alert Components

- Reusable components are UI elements that can be used across multiple pages with different data.
- Key Characteristics
    - Accept data using `@Input()`
    - Emit events using `@Output()`
    - Avoid business logic
    - Focus only on UI rendering

Example: Reusable Card Component

```
@Component({
  selector: 'app-card',
  standalone: true,
  template: `
    <div class="card">
      <h3>{{ title }}</h3>
      <p>{{ description }}</p>
    </div>  `
})
export class CardComponent {
  @Input() title!: string;
  @Input() description!: string;
}
```

Usage:

```
<app-card
  title="Angular 17"
  description="Reusable UI components">
</app-card>
```

Modal and Alert Components

- Modal: Confirmation dialogs, forms
- Alert: Success, error, warning messages
- Controlled via inputs and events

1.4.2 Component Interaction with RxJS and Shared Services

- When components are not directly related, **shared services with RxJS** are used for communication.
- Why Shared Services
    - Avoid deeply nested component communication
    - Maintain centralized state
    - Improve scalability

Example: Shared Service Using BehaviorSubject

```
@Injectable({ providedIn: 'root' })
export class MessageService {
  private messageSource = new BehaviorSubject<string>('Default');
  message$ = this.messageSource.asObservable();

  updateMessage(msg: string) {
    this.messageSource.next(msg);
  }
}
```

Component Interaction

- Component A updates data
- Component B subscribes to data

This pattern is commonly used for:

- Notifications
- User authentication state
- Theme and UI preferences

1.4.3 Use of ng-template, ng-container, ng-content

Angular provides structural tools for flexible component layouts.

ng-container

- Groups elements without adding extra DOM nodes
- Useful with structural directives

```
<ng-container *ngIf="isLoggedIn">
  <p>Welcome User</p>
</ng-container>
```

ng-template

- Defines reusable HTML blocks
- Rendered conditionally or dynamically
- Used with: *ngIf, ngTemplateOutlet

```
<ng-template #loading>
  <p>Loading...</p>
</ng-template>
```

1. ng-template with ngIf

`ng-template` is commonly used to define an **else block** for `*ngIf`.

```
<div *ngIf="isLoggedIn; else loginTemplate">
  <p>Welcome back!</p>
</div>

<ng-template #loginTemplate>
  <p>Please log in</p>
</ng-template>
```

- Explanation
  - If `isLoggedIn` is `true`, the first block is shown
  - If `false`, the `ng-template` named `loginTemplate` is rendered
  - `ng-template` itself does not render unless referenced

2. ng-template with ngTemplateOutlet

`ngTemplateOutlet` is used to **dynamically render templates**.

```
//in the component's html file

<ng-container
  *ngTemplateOutlet="selectedTemplate">
</ng-container>

<ng-template #templateOne>
  <p>This is Template One</p>
</ng-template>

<ng-template #templateTwo>
  <p>This is Template Two</p>
</ng-template>
```

```
//in the component's ts file

selectedTemplate = this.templateOne;
```

- Explanation
  - `ngTemplateOutlet` renders the template referenced by `selectedTemplate`
  - Useful for dynamic layouts, tabs, or role-based UI
  - `ng-container` avoids extra DOM elements

1.4.4 Smart vs Dumb Components (Best Practices)

Angular applications follow a **separation of concerns** pattern.

- Smart Components (Container Components)
  - Handle data fetching and business logic
  - Communicate with services
  - Pass data to child components
  - Example: Pages, Dashboards, Feature components
- Dumb Components (Presentational Components)
  - Display UI only
  - Receive data via `@Input()`
  - Emit events via `@Output()`
  - No service injection
  - Example: Buttons, Cards, Lists, Forms UI

Feature	Smart Component	Dumb Component
Business logic	Yes	No
Service usage	Yes	No
Reusability	Low	High
UI focused	No	Yes

Best Practices

- Keep components small and focused

- Prefer dumb components for UI
- Let smart components control data flow
- Use shared services for cross-component communication

Real-World Use Cases

- Cards for product listing
- Modals for delete confirmation
- Alerts for form submission status
- Shared services for login state
- Content projection for layout components

1.5 Application Deployment & Performance Optimization

After building a functional Angular application, the final steps involve **building, deploying, and optimizing** the application for performance, scalability, and production readiness.

1.5.1 Angular Build Process, Environments, and Optimization Flags

Angular Build Process

Angular applications are built using **Angular CLI**, which:

- Compiles TypeScript to JavaScript
- Bundles modules
- Optimizes assets
- Minifies code for production
- Command: **ng build**
- Production build: **ng build --configuration=production**

Environment Configuration

- Angular supports multiple environments such as:
    - Development
    - Testing
    - Production
- Each environment has its own configuration file.

Example:

```
export const environment = {
  production: true,
  apiUrl: 'https://api.myapp.com'
};
```

Usage:

```
this.http.get(environment.apiUrl + '/users');
```

Optimization Flags (Production Mode)

- When building for production, Angular enables:
  - Ahead-of-Time compilation (AOT)
  - Minification and uglification
  - Tree shaking (removal of unused code)
  - Build optimization
- These reduce bundle size and improve load time.

1.5.2 Deploying Angular Applications Using Firebase Hosting

Firebase Hosting is a popular and simple option for deploying Angular SPAs.

- Deployment Steps
  - Build the Angular project
  - Install Firebase CLI
  - Initialize Firebase
  - Deploy the application

Build command:

```
ng build --configuration=production
```

Deploy command:

```
firebase deploy
```

Why Firebase Hosting

- Fast global CDN
- HTTPS by default
- Easy SPA routing support
- Suitable for Angular applications

1.5.3 Performance Tuning Techniques

Performance optimization ensures smooth UI rendering and faster application response.

a. trackBy in ngFor

- Angular recreates DOM elements when data changes. `trackBy` helps Angular identify items uniquely and avoid unnecessary re-rendering.

Example:

```
<!—in html file -->
<li *ngFor="let user of users; trackBy: trackById">
  {{ user.name }}
</li>
```

```
//in ts file
trackById(index: number, user: any) {
  return user.id;
}
```

b. OnPush Change Detection

- Default change detection checks all components.
- `OnPush` improves performance by checking components **only when inputs change**.
- Use cases:
    - Lists
    - Dashboards
    - Reusable UI components

Example:

```
@Component({
  selector: 'app-user-card',
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class UserCardComponent {
  @Input() user!: User;
}
```

c. Lazy Loading Routes and Components

- Lazy loading loads modules or components **only when required**, reducing initial bundle size.

Example:

```
//in ts file
{
  path: 'admin',
  loadChildren: () =>
    import('./admin/admin.routes').then(m => m.ADMIN_ROUTES)
}
```

- Benefits:
  - Faster initial load
  - Better user experience
  - Optimized resource usage

Real-World Performance Scenarios

- trackBy → Large data tables
- OnPush → Reusable UI components
- Lazy loading → Admin panels, dashboards
- Production build → Faster app loading

Unit 2: Introduction to Express.js and Server-Side Basics with Node.js

2.1 Introduction to Node.js and Express.js

Node.js revolutionizes server-side development with its event-driven, non-blocking I/O model powered by the V8 engine and libuv library. Express.js builds atop Node.js, offering structured routing, middleware, and request handling absent in raw HTTP servers. Section 2.1 covers foundational setup for scalable backends.

2.1.1 Installing Node.js (v20+) and Setting up Express Server

Installation Steps:

```
# 1. Download Node.js v20+ from nodejs.org (LTS recommended)
# 2. Verify installation
node --version  # v20.10.0 or higher
npm --version   # 10.x.x or higher

# 3. Create project
mkdir my-express-api
cd my-express-api
npm init -y

# 4. Install Express
npm install express

# 5. Install nodemon (development)
npm install --save-dev nodemon
```

First Express Server:

```
// server.js - Your first Express application
const express = require('express');
const app = express();

// Basic route
app.get('/', (req, res) => {
  res.json({
    message: 'Welcome to Express.js API',
    version: 'Node ' + process.version
  });
});

// Start server
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(`Server running on http://localhost:${PORT}`);
});
```

package.json Scripts:

```
{
  "scripts": {
    "start": "node server.js",
    "dev": "nodemon server.js"
  }
}
```

Run: `npm run dev`

2.1.2 Creating a RESTful Backend using Express.js

REST API Structure:

```
GET    /api/users       → List all users
GET    /api/users/:id  → Get single user
POST   /api/users       → Create new user
PUT    /api/users/:id  → Update user
DELETE /api/users/:id  → Delete user
```

Complete REST Server Example:

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

// Middleware (essential)
app.use(express.json());  // Parse JSON bodies

// Sample data (replace with database later)
let users = [
  { id: 1, name: 'John Doe', email: 'john@example.com' },
  { id: 2, name: 'Jane Smith', email: 'jane@example.com' }
];

// RESTful routes
app.get('/api/users', (req, res) => {
  res.json({ success: true, data: users });
});

app.get('/api/users/:id', (req, res) => {
  const user = users.find(u => u.id === parseInt(req.params.id));
  if (!user) return res.status(404).json({ error: 'User not found' });
  res.json({ success: true, data: user });
});
```

```
app.listen(3000, () => console.log('API ready on port 3000'));
```

2.1.3 Introduction to Nodemon and Project Structuring

Project Structure:

```
my-express-api/
├── controllers/
|   └── userController.js
├── routes/
|   └── userRoutes.js
├── middleware/
|   └── logger.js
├── public/          # Static files (images, CSS)
├── .env           # Environment variables
├── server.js
└── package.json
```

Nodemon Configuration:

```
// package.json
{
  "scripts": {
    "start": "node server.js",
    "dev": "nodemon server.js",
    "test": "echo \"Error: no test specified\" && exit 1"
  }
}
```

nodemon.json (Optional Customization):

```
{
  "watch": ["server.js", "routes/", "controllers/"],
  "ext": "js,json",
  "ignore": ["node_modules/", "public/"],
  "env": {
```

```
  "NODE_ENV": "development"
 }
}
```

2.2 Handling Routes and HTTP Methods

2.2.1 Defining Routes using GET, POST, PUT, DELETE

HTTP Method Reference:

Method	Purpose	Example URL	Status Code
GET	Retrieve data	/api/users	200
POST	Create data	/api/users	201
PUT	Update data	/api/users/1	200
DELETE	Delete data	/api/users/1	200

Complete Route Examples:

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

app.use(express.json());

// GET - List all items
app.get('/api/products', (req, res) => {
  res.status(200).json({ products: [] });
});

// POST - Create new item
app.post('/api/products', (req, res) => {
  const newProduct = req.body;
  res.status(201).json({ message: 'Created', product: newProduct });
});

// PUT - Update existing item
app.put('/api/products/:id', (req, res) => {
  const id = req.params.id;
  res.json({ message: `Updated product ${id}` });
});

// DELETE - Remove item
app.delete('/api/products/:id', (req, res) => {
  res.json({ message: 'Deleted' });
});
```

2.2.2 Sending Responses and Working with Route/Query Parameters

Parameter Types:

```
app.get('/api/users/:id/:action', (req, res) => {
  console.log('Route params:', req.params);     // { id: '1', action: 'view' }
  console.log('Query params:', req.query);      // { sort: 'name', limit: '10' }
  console.log('Body data:', req.body);          // { name: 'John' } (POST/PUT)

  res.json({
    routeParams: req.params,
    queryParams: req.query,
    message: `User ${req.params.id} - ${req.params.action}`
  });
});
```

Test URLs:

```
GET  /api/users/123/view?sort=name&limit=10
GET  /api/users/Jane?active=true
POST /api/users (body: {name: "New User"})
```

Response Patterns:

```
// Success responses
res.status(200).json({ success: true, data: users });
res.status(201).json({ success: true, data: newUser });

// Error responses
res.status(400).json({ success: false, error: 'Validation failed' });
res.status(404).json({ success: false, error: 'Not found' });
res.status(500).json({ success: false, error: 'Server error' });
```

2.2.3 Connecting Routes to Controller Logic

Separation of Concerns:

```
// controllers/userController.js
exports.getAllUsers = (req, res) => {
  res.json({ users: [] });
};

exports.getUserById = (req, res) => {
  res.json({ user: { id: req.params.id } });
};

exports.createUser = (req, res) => {
  res.status(201).json({ user: req.body });
};


// routes/userRoutes.js
const express = require('express');
const router = express.Router();
const userController = require('../controllers/userController');

// Connect controller methods to routes
router.get('/', userController.getAllUsers);
router.get('/:id', userController.getUserById);
router.post('/', userController.createUser);

module.exports = router;


// server.js - Register routes
const userRoutes = require('./routes/userRoutes');
app.use('/api/users', userRoutes);
```

2.3 Middleware and API Basics

2.3.1 Understanding Middleware in Express

Middleware Flow:

```
Request → Middleware 1 → Middleware 2 → Route Handler → Response
```

Middleware Types:

```
// 1. Built-in middleware
app.use(express.json());          // Parse JSON
app.use(express.urlencoded());    // Parse forms

// 2. Application-level middleware
app.use((req, res, next) => {
  console.log(`${req.method} ${req.url}`);
  next();  // Continue to next middleware
});

// 3. Router-level middleware
const router = express.Router();
router.use((req, res, next) => {
  req.requestTime = new Date();
  next();
});

// 4. Error-handling middleware (4 parameters)
app.use((err, req, res, next) => {
  res.status(500).json({ error: err.message });
});
```

2.3.2 Using Built-in and Custom Middleware

Complete Middleware Stack:

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

// 1. Security headers
app.use((req, res, next) => {
  res.setHeader('X-Powered-By', undefined);
  next();
});

// 2. Body parsers (MUST be before routes)
app.use(express.json({ limit: '10kb' }));
app.use(express.urlencoded({ extended: true }));

// 3. Custom logging middleware
app.use((req, res, next) => {
  console.log(`[${new Date().toISOString()}] ${req.method} ${req.url}`);
  next();
});

// 4. Static files
app.use(express.static(path.join(__dirname, 'public')));

// 5. Routes (AFTER middleware)
app.use('/api', require('./routes/api'));

// 6. 404 handler
app.use('*', (req, res) => {
  res.status(404).json({ error: 'Route not found' });
});
```

2.3.3 Introduction to CORS and Environment Variables

CORS Configuration:

```
npm install cors

// Enable CORS for specific origins
const cors = require('cors');
app.use(cors({
  origin: 'http://localhost:4200',  // Angular dev server
  credentials: true
}));

// OR allow all origins (development only)
app.use(cors());
```

Environment Variables (.env file):

```
npm install dotenv
# .env
PORT=3000
NODE_ENV=development
MONGODB_URI=mongodb://localhost:27017/myapp
API_KEY=your-secret-key


// server.js
require('dotenv').config();

const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(`Server running on port ${PORT} (${process.env.NODE_ENV})`);
});
```

Add to .gitignore:

```
node_modules/
.env
.DS_Store
```

Unit 3: Building Web App Components and Backend Integration

3.1 Working with Forms and APIs

3.1.1 Handling Form Submissions in Express

Express processes form data through middleware that parses incoming requests:

```
// server.js - Essential middleware setup
const express = require('express');
const app = express();

// Parse JSON bodies (for API calls)
app.use(express.json());

// Parse URL-encoded bodies (for HTML forms)
app.use(express.urlencoded({ extended: true }));

app.listen(3000, () => console.log('Server running on port 3000'));
```

Form Processing Flow:

```
HTML Form/Angular → POST /api/users → express.urlencoded() → req.body → Controller
```

Example: User Registration Endpoint

```
app.post('/api/users', (req, res) => {
  const { name, email, password } = req.body;

  // Validation
  if (!name || !email || !password) {
    return res.status(400).json({ error: 'All fields required' });
  }
  // Process data
  const newUser = { name, email, password };
  res.status(201).json({ message: 'User created', user: newUser });
});
```

3.1.2 Sending JSON Responses and Extracting Request Data

Standard JSON Response Patterns:

Status	Method	Example
200 OK	res.json(data)	{ users: [{id:1, name:"John"}] }
201 Created	res.status(201).json(newItem)	{ message: "User created", id: "123" }
400 Bad Request	res.status(400).json({error})	{ error: "Email required" }
404 Not Found	res.status(404).json({error})	{ error: "User not found" }

```
// Complete API endpoint example
app.get('/api/users/:id', async (req, res) => {
  try {
    const userId = req.params.id;
    const user = await User.findById(userId);

    if (!user) {
      return res.status(404).json({ error: 'User not found' });
    }

    res.json({
      success: true,
      data: user
    });
  } catch (error) {
    res.status(500).json({ error: 'Server error' });
  }
});
```

3.1.3 Connecting Angular Forms to Express APIs

Angular Reactive Form Setup:

```
// user-form.component.ts
import { FormBuilder, FormGroup, Validators } from '@angular/forms';
import { HttpClient } from '@angular/common/http';

export class UserFormComponent {
  userForm: FormGroup;

  constructor(private fb: FormBuilder, private http: HttpClient) {
    this.userForm = this.fb.group({
      name: ['', Validators.required],
      email: ['', [Validators.required, Validators.email]],
      password: ['', Validators.required]
    });
  }

  onSubmit() {
    if (this.userForm.valid) {
      this.http.post('http://localhost:3000/api/users',
        this.userForm.value)
        .subscribe({
          next: (response) => console.log('User created!', response),
          error: (error) => console.error('Error:', error)
        });
    }
  }
}
```

Proxy Configuration (Development):

```
// proxy.conf.json - Eliminates CORS issues
{
  "/api/*": {
    "target": "http://localhost:3000",
    "secure": false,
    "changeOrigin": true
  }
}
```

Run: `ng serve --proxy-config proxy.conf.json`

3.2 Organizing Code with Models and Controllers

3.2.1 Structuring Backend with Models, Services, and Controllers

```
backend/
├── models/
│   └── User.js
├── controllers/
│   └── userController.js
├── routes/
│   └── userRoutes.js
├── middleware/
│   └── auth.js
└── server.js
```

MVC Responsibilities:

Component	Purpose	Example
Model	Database schema & validation	Mongoose User schema
Controller	Business logic & HTTP responses	createUser(), getUsers()
Route	URL mapping	router.post('/users', createUser)

3.2.2 Creating Data Models for Users/Products using MongoDB (Mongoose)

```
// models/User.js
const mongoose = require('mongoose');
const bcrypt = require('bcryptjs');

const userSchema = new mongoose.Schema({
  name: { type: String, required: true, trim: true },
  email: { type: String, required: true, unique: true },
  password: { type: String, required: true, minlength: 6 },
  createdAt: { type: Date, default: Date.now }
});

// Hash password before saving
userSchema.pre('save', async function(next) {
  if (!this.isModified('password')) return next();
  this.password = await bcrypt.hash(this.password, 12);
  next();
});

module.exports = mongoose.model('User', userSchema);
```

Product Model Example:

```
// models/Product.js
const productSchema = new mongoose.Schema({
  name: { type: String, required: true },
  price: { type: Number, required: true, min: 0 },
  description: String,
  category: String,
  stock: { type: Number, default: 0 }
});

module.exports = mongoose.model('Product', productSchema);
```

3.2.3 Basic CRUD Operations using Express and MongoDB

Complete Controller Implementation:

```
// controllers/userController.js
const User = require('../models/User');

exports.getUsers = async (req, res) => {
  try {
    const users = await User.find().select('-password');
    res.json({ success: true, count: users.length, data: users });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
};

exports.createUser = async (req, res) => {
  try {
    const user = new User(req.body);
    await user.save();
    res.status(201).json({
      success: true,
      message: 'User created successfully',
      data: { id: user._id, name: user.name, email: user.email }
    });
  } catch (error) {
    res.status(400).json({ error: error.message });
  }
};

exports.updateUser = async (req, res) => {
  try {
    const user = await User.findByIdAndUpdate(
      req.params.id,
      req.body,
      { new: true, runValidators: true }
    ).select('-password');

    if (!user) {
      return res.status(404).json({ error: 'User not found' });
    }

    res.json({ success: true, data: user });
  } catch (error) {
    res.status(400).json({ error: error.message });
  }
};

exports.deleteUser = async (req, res) => {
  try {
    const user = await User.findByIdAndDelete(req.params.id);
    if (!user) {
      return res.status(404).json({ error: 'User not found' });
    }
    res.json({ success: true, message: 'User deleted' });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
};
```

Route Registration:

```
// routes/userRoutes.js
const express = require('express');
const router = express.Router();
const userController = require('../controllers/userController');

router.route('/')
  .get(userController.getUsers)
  .post(userController.createUser);

router.route('/:id')
  .put(userController.updateUser)
  .delete(userController.deleteUser);

module.exports = router;
```

Main Server Setup:

```
// server.js
const express = require('express');
const mongoose = require('mongoose');
const userRoutes = require('./routes/userRoutes');

mongoose.connect('mongodb://localhost:27017/fullstack-app')
  .then(() => console.log('MongoDB connected'))
  .catch(err => console.error(err));

const app = express();
app.use(express.json());
app.use('/api/users', userRoutes);
app.listen(3000, () => console.log('Server running on port 3000'));
```

3.3 Authentication and Security Basics

3.3.1 Introduction to Firebase Authentication (Email & Password)

Firebase Console Setup:

1. Enable "Email/Password" provider in Authentication
2. Get Firebase config from Project Settings
3. Install: `npm install firebase @angular/fire`

3.3.2 Adding User Registration and Login to Angular Frontend

Auth Service:

```
// services/auth.service.ts
import { Injectable } from '@angular/core';
import { Auth, createUserWithEmailAndPassword, signInWithEmailAndPassword, signOut } from
'@angular/fire/auth';

@Injectable({ providedIn: 'root' })
export class AuthService {
  constructor(private auth: Auth) {}

  register(email: string, password: string) {
    return createUserWithEmailAndPassword(this.auth, email, password);
  }

  login(email: string, password: string) {
    return signInWithEmailAndPassword(this.auth, email, password);
  }
```

```
 logout() {
   return signOut(this.auth);
 }
}
```

Login Component:

```
// login.component.ts
export class LoginComponent {
 loginForm = this.fb.group({
   email: ['', [Validators.required, Validators.email]],
   password: ['', Validators.required]
 });

 constructor(
   private fb: FormBuilder,
   private authService: AuthService,
   private router: Router
 ) {}

 onLogin() {
   if (this.loginForm.valid) {
     this.authService.login(
       this.loginForm.value.email!,
       this.loginForm.value.password!
     ).then(() => this.router.navigate(['/dashboard']))
       .catch(error => console.error('Login failed:', error));
   }
 }
}
```

3.3.3 Securing Backend Routes with Firebase Admin SDK and JWT

Install Admin SDK:

```
npm install firebase-admin
```

Service Account Setup:

1. Download service account JSON from Firebase Console → Project Settings → Service Accounts
2. Save as `serviceAccount.json` in project root

Admin SDK Initialization:

```
// config/firebaseAdmin.js
const admin = require('firebase-admin');
const serviceAccount = require('./serviceAccount.json');

admin.initializeApp({
  credential: admin.credential.cert(serviceAccount)
});

module.exports = admin;
```

Authentication Middleware:

```
// middleware/auth.js
const admin = require('../config/firebaseAdmin');

module.exports = async (req, res, next) => {
  try {
    const token = req.headers.authorization?.split('Bearer ')[1];
    if (!token) {
      return res.status(401).json({ error: 'No token provided' });
    }
    const decodedToken = await admin.auth().verifyIdToken(token);
    req.user = decodedToken;
    next();
  } catch (error) {
    res.status(401).json({ error: 'Invalid token' });
  }
};
```

Protected Route Example:

```
// routes/protectedRoutes.js
const express = require('express');
const router = express.Router();
const auth = require('../middleware/auth');

router.get('/profile', auth, async (req, res) => {
  res.json({
    message: 'Protected data',
    userId: req.user.uid,
    email: req.user.email
  });
});

module.exports = router;
```

Unit 4: Firebase and React Integration

4.1 Firebase Firestore and Realtime Database

4.1.1 Setting up Firebase project and Firestore database

4.1.2 Performing basic CRUD operations on Firestore

(add, read, update, delete)

4.1.3 Structuring collections and subcollections

4.2 Connecting Firebase with Angular and Express

4.2.1 Integrating Firebase SDK in Angular to fetch/store data

4.2.2 Connecting Firebase Admin SDK in Express for backend access

4.2.3 Basic deployment using Firebase Hosting (for frontend)

4.3 Overview of React Frontend for Full Stack Developers

4.3.1 Setting up a basic React app using Vite or Create React App

4.3.2 Understanding JSX, functional components, and hooks

(useState, useEffect)

4.3.3 Integrating Firebase in a React app for data access and authentication

4.1 Firebase Firestore and Realtime Database

4.1.1 Setting Up Firebase Project and Firestore Database

What is Firebase?

Firebase is Google's platform for building web and mobile applications with **real-time databases, authentication, hosting, and cloud functions**. For this course, we focus on:

- **Firestore**: A document-oriented, NoSQL database with real-time synchronization
- **Realtime Database**: Alternative with JSON structure (older, still widely used)
- **Firebase Hosting**: CDN-based static site hosting
- **Firebase Authentication**: Built-in user management

Why Firestore over Realtime Database?

Feature	Firestore	Realtime Database
Data Model	Document-Oriented (JSON-like)	JSON structure
Querying	Rich queries (filtering, sorting, pagination)	Limited query support
Pricing	Pay per read/write	Pay per GB stored
Scaling	Better for large datasets	Slower with massive data
Use Case	Structured data apps	Real-time messaging, simple CRUD

Step 1: Create a Firebase Project

1. Go to firebase.google.com
2. Click "Go to Console" → Sign in with your Google account
3. Click "Create a Project"
4. Enter project name: `full-stack-edu` (example)
5. Accept terms, click "Create Project"
6. Wait for provisioning (1–2 minutes)
7. Click "Continue"

Step 2: Enable Firestore Database

1. In Firebase Console, go to **Build → Firestore Database**
2. Click **"Create Database"**
3. Select location closest to your users (e.g., asia-south1 for India)
4. Choose **"Start in Test Mode"** (for development)
    - Test Mode: Open read/write (DO NOT use in production)
    - Production Mode: Requires security rules
5. Click **"Create"**

⚠ **Security Warning**: Test Mode allows anyone with your config to read/write. For production, set security rules (Section 4.1.2).

Step 3: Get Firebase Configuration

1. Click **Project Settings** (gear icon)
2. Go to **"Your apps"** section
3. Click **Web app** (</> icon)
4. Copy the config object:

```
const firebaseConfig = {
apiKey: "YOUR_API_KEY",
authDomain: "YOUR_PROJECT.firebaseapp.com",
projectId: "YOUR_PROJECT_ID",
storageBucket: "YOUR_PROJECT.appspot.com",
messagingSenderId: "YOUR_SENDER_ID",
appId: "YOUR_APP_ID"
};
```

Store this safely (use .env file in production).

Step 4: Install Firebase SDK

```
npm install firebase
```

Step 5: Initialize Firebase in Your App

For React/Angular (Web Client):

```
// src/firebase.js
import { initializeApp } from "firebase/app";
import { getFirestore } from "firebase/firestore";
import { getAuth } from "firebase/auth";
```

```
const firebaseConfig = {
apiKey: process.env.REACT_APP_API_KEY,
authDomain: process.env.REACT_APP_AUTH_DOMAIN,
projectId: process.env.REACT_APP_PROJECT_ID,
storageBucket: process.env.REACT_APP_STORAGE_BUCKET,
messagingSenderId: process.env.REACT_APP_MESSAGING_SENDER_ID,
appId: process.env.REACT_APP_APP_ID
};

const app = initializeApp(firebaseConfig);
export const db = getFirestore(app);
export const auth = getAuth(app);
```

For Angular 20+ (Standalone Components):

```
// src/app/app.config.ts
import { ApplicationConfig } from '@angular/core';
import { provideFirebaseApp, initializeApp } from '@angular/fire/app';
import { provideFirestore, getFirestore } from '@angular/fire/firestore';
import { provideAuth, getAuth } from '@angular/fire/auth';

export const appConfig: ApplicationConfig = {
providers: [
provideFirebaseApp(() => initializeApp({
apiKey: "YOUR_API_KEY",
projectId: "YOUR_PROJECT_ID",
// ... full config
})),
provideFirestore(() => getFirestore()),
provideAuth(() => getAuth()),
]
};
```

4.1.2 Performing Basic CRUD Operations on Firestore

Understanding Firestore Data Model

Hierarchy:

```
Firestore Database
├── Collection: users
│ ├── Document: user123
│ │ ├── Field: name = "Rahul"
│ │ ├── Field: email = "rahul@example.com"
```

```
| | └── Field: age = 22
| └── Document: user456
| ├── Field: name = "Priya"
| └── Field: email = "priya@example.com"
└── Collection: courses
├── Document: course001
| ├── Field: title = "React Basics"
| └── Field: instructor = "user123"
└── Document: course002
```

CRUD Operations: Complete Code

1. CREATE (Add Documents)

```
// Import necessary functions
import { collection, addDoc, setDoc, doc } from "firebase/firestore";
import { db } from "./firebase"; // Your initialized Firebase

// Method 1: Auto-generated ID (Recommended)
async function addStudent() {
try {
const docRef = await addDoc(collection(db, "students"), {
name: "Rahul Sharma",
email: "rahul@example.com",
rollNo: "BCA2024001",
course: "React",
createdAt: new Date()
});
console.log("Document added with ID:", docRef.id);
} catch (error) {
console.error("Error adding document:", error);
}
}

// Method 2: Custom ID
async function addStudentWithCustomId() {
try {
await setDoc(doc(db, "students", "student_001"), {
name: "Priya Gupta",
email: "priya@example.com",
rollNo: "BCA2024002",
course: "Angular",
createdAt: new Date()
});
console.log("Document created with custom ID: student_001");
} catch (error) {
console.error("Error:", error);
}
}

// Batch Write (Multiple documents)
import { writeBatch } from "firebase/firestore";

async function addMultipleStudents() {
const batch = writeBatch(db);

const studentsData = [
{ name: "Amit", email: "amit@example.com", course: "React" },
{ name: "Neha", email: "neha@example.com", course: "Angular" }
];

studentsData.forEach((data) => {
const newDocRef = doc(collection(db, "students"));
batch.set(newDocRef, { ...data, createdAt: new Date() });
});

await batch.commit();
console.log("Batch write completed");
}
```

2. READ (Fetch Documents)

```
import { collection, getDocs, getDoc, doc, query, where } from "firebase/firestore";

// Read All Documents (Once)
async function getAllStudents() {
try {
const querySnapshot = await getDocs(collection(db, "students"));
const students = [];

querySnapshot.forEach((doc) => {
  students.push({
    id: doc.id,
    ...doc.data()
  });
});

console.log("All students:", students);
return students;

} catch (error) {
console.error("Error reading documents:", error);
}
}
```

```
// Read Single Document by ID
async function getStudentById(studentId) {
try {
const docSnap = await getDoc(doc(db, "students", studentId));

if (docSnap.exists()) {
  console.log("Student data:", docSnap.data());
  return { id: docSnap.id, ...docSnap.data() };
} else {
  console.log("Student not found");
  return null;
}

} catch (error) {
console.error("Error:", error);
}
}

// Read with Filters/Queries
async function getStudentsByCourse(courseName) {
try {
const q = query(
collection(db, "students"),
where("course", "==", courseName)
);

const querySnapshot = await getDocs(q);
const students = querySnapshot.docs.map(doc => ({
  id: doc.id,
  ...doc.data()
}));

console.log(`Students in ${courseName}:`, students);
return students;

} catch (error) {
console.error("Error querying:", error);
}
}

// Real-Time Listener (Live Updates)
import { onSnapshot } from "firebase/firestore";

function watchStudents() {
const unsubscribe = onSnapshot(
collection(db, "students"),
(querySnapshot) => {
const students = [];
querySnapshot.forEach((doc) => {
students.push({ id: doc.id, ...doc.data() });
});
console.log("Real-time students update:", students);
},
(error) => {
console.error("Error listening to updates:", error);
}
);

// Call unsubscribe() later to stop listening
return unsubscribe;
}
```

3. UPDATE (Modify Documents)

```
import { updateDoc, doc, increment } from "firebase/firestore";

// Update Specific Fields
async function updateStudent(studentId) {
try {
await updateDoc(doc(db, "students", studentId), {
email: "newemail@example.com",
course: "Full-Stack",
updatedAt: new Date()
});
console.log("Student updated successfully");
} catch (error) {
console.error("Error updating:", error);
}
}

// Increment Numeric Field
async function incrementStudentScore(studentId, points) {
try {
await updateDoc(doc(db, "students", studentId), {
score: increment(points) // Atomic operation
});
console.log(Score incremented by ${points});
} catch (error) {
console.error("Error:", error);
}
}

// Array Operations
import { arrayUnion, arrayRemove } from "firebase/firestore";

async function addSkill(studentId, skill) {
try {
```

```
await updateDoc(doc(db, "students", studentId), {
skills: arrayUnion(skill) // Add to array if not exists
});
} catch (error) {
console.error("Error:", error);
}
}

async function removeSkill(studentId, skill) {
try {
await updateDoc(doc(db, "students", studentId), {
skills: arrayRemove(skill) // Remove from array
});
} catch (error) {
console.error("Error:", error);
}
}
```

4. DELETE (Remove Documents)

```
import { deleteDoc, doc } from "firebase/firestore";

// Delete Single Document
async function deleteStudent(studentId) {
try {
await deleteDoc(doc(db, "students", studentId));
console.log("Student deleted successfully");
} catch (error) {
console.error("Error deleting:", error);
}
}

// Delete a Field from Document
import { deleteField } from "firebase/firestore";

async function removeStudentField(studentId, fieldName) {
try {
await updateDoc(doc(db, "students", studentId), {
[fieldName]: deleteField()
});
console.log(Field ${fieldName} removed);
} catch (error) {
console.error("Error:", error);
}
}

// Delete All Documents in Collection (Dangerous!)
async function deleteAllStudents() {
try {
```

```
const querySnapshot = await getDocs(collection(db, "students"));
const batch = writeBatch(db);

querySnapshot.docs.forEach((doc) => {
  batch.delete(doc.ref);
});

await batch.commit();
console.log("All students deleted");

} catch (error) {
console.error("Error:", error);
}
}
```

CRUD Operations Summary Table

Operation	Function	Returns	Use Case
Create	addDoc()	Document Reference with ID	New data, auto-ID
	setDoc()	None	Custom IDs, overwrites
Read	getDoc()	Single document	Get by ID
	getDocs()	All documents in collection	Fetch all
	query() + getDocs()	Filtered documents	Search/filter
	onSnapshot()	Real-time listener	Live updates
Update	updateDoc()	None	Partial update
	increment()	None	Atomic counter
	arrayUnion()	None	Add to array
Delete	deleteDoc()	None	Remove document
	deleteField()	None	Remove field

4.1.3 Structuring Collections and Subcollections

Collection Design Patterns

Pattern 1: Flat Collections (Simple Structure)

Best for: Simple data, frequently accessed together

```
Firestore
├── users/
│ ├── user1 { name, email, role }
│ └── user2 { name, email, role }
├── posts/
│ ├── post1 { title, content, authorId: user1 }
│ └── post2 { title, content, authorId: user2 }
└── comments/
├── comment1 { text, postId: post1, authorId: user1 }
└── comment2 { text, postId: post1, authorId: user2 }
```

Pros: Simple queries, easy to maintain
Cons: Duplicate data, many read operations

Pattern 2: Subcollections (Hierarchical Structure)

Best for: Related data, ownership hierarchies

```
Firestore
├── users/
│ ├── user1/
│ │ ├── name, email, role
│ │ └── subcollection: posts/
│ │ ├── post1 { title, content, likes: 45 }
│ │ ├── post2 { title, content, likes: 23 }
│ │ └── subcollection under post1: comments/
│ │ ├── comment1 { text, authorId }
│ │ └── comment2 { text, authorId }
│ └── user2/
│ └── posts/
│ └── post3 { title, content }
```

Pros: Data isolation, clear ownership, unlimited depth
Cons: More complex queries, hierarchical reads

Pattern 3: Root Collection with Maps (Denormalization)

Best for: Frequently accessed nested data

```
Firestore
├── courses/
│ ├── course1
│ │ ├── title: "React Basics"
│ │ ├── instructor:
│ │ │ ├── id: "instructor1"
│ │ │ ├── name: "Dr. Sharma"
│ │ │ └── email: "sharma@edu.com"
```

```
| | └── modules:
| | ├── [0]: { name: "JSX", order: 1, duration: 2 }
| | └── [1]: { name: "Hooks", order: 2, duration: 3 }
```

Pros: Fewer reads, faster access
Cons: Denormalized, needs sync logic

Practical Example: Educational App Structure

```
// Collection: students
// ├── Document: STU001
// | ├── name: "Rahul Sharma"
// | ├── email: "rahul@college.edu"
// | ├── enrolledCourses: ["REACT101", "ANGULAR101"]
// | └── subcollection: assignments/
// | ├── Document: ASG001
// | | ├── title: "Build Todo App"
// | | ├── dueDate: 2026-02-28
// | | ├── status: "submitted"
// | | └── score: 45/50
// | └── Document: ASG002
// | └── ...
// └── Document: STU002
// └── ...

// Create hierarchical data
async function createStudentWithAssignments(studentData) {
try {
// Add student
const studentRef = await addDoc(collection(db, "students"), {
name: studentData.name,
email: studentData.email,
enrolledCourses: [],
createdAt: new Date()
});

// Add assignments subcollection
const assignmentRef = collection(studentRef, "assignments");

for (const assignment of studentData.assignments) {
  await addDoc(assignmentRef, {
    ...assignment,
    submittedAt: new Date(),
    status: "pending"
  });
}

console.log("Student and assignments created");
return studentRef.id;

} catch (error) {
console.error("Error:", error);
}
}

// Query student with assignments
async function getStudentAssignments(studentId) {
try {
const assignments = await getDocs(
collection(db, "students", studentId, "assignments")
);

const result = assignments.docs.map(doc => ({
  id: doc.id,
  ...doc.data()
}));

return result;

} catch (error) {
console.error("Error:", error);
}
}

// Update nested assignment
async function updateAssignmentScore(studentId, assignmentId, score) {
try {
const assignmentRef = doc(
db,
"students",
studentId,
"assignments",
assignmentId
);

await updateDoc(assignmentRef, {
  score: score,
  gradedAt: new Date()
});

} catch (error) {
console.error("Error:", error);
}
}
```

When to Use Subcollections vs Root Collections

Scenario	Recommendation	Reason
User has 100 posts	Use subcollection users/{userId}/posts	Isolation, cleaner queries
Need to search all posts globally	Use root collection posts	Root collection queries are simpler
Comments on posts	Use posts/{postId}/comments	Clear hierarchy, private to post
User settings (1–2 documents)	Use field in user doc	No need for collection
Product reviews (many per product)	Use products/{productId}/reviews	Organize by parent, scale well

Firestore Limitations & Best Practices

Limitation	Impact	Solution
Max 100 levels of subcollections	Deep nesting impossible	Keep max 3–4 levels
Field depth max 20	Can't deeply nest maps	Flatten structure
Max 1 write per second per doc	Rate limiting on hot docs	Distribute writes
Cost grows with reads/writes	High traffic = high cost	Use batch writes, pagination
No multi-collection queries	Can't JOIN data	Denormalize, fetch separately

---

4.2 Connecting Firebase with Angular and Express

4.2.1 Integrating Firebase SDK in Angular to Fetch/Store Data

Setup Angular 20 with Firebase

Step 1: Install Dependencies

```
ng new full-stack-edu
cd full-stack-edu
npm install firebase @angular/fire   //check the appropriate version of node js
```

Step 2: Configure in app.config.ts

```
// src/app/app.config.ts
import { ApplicationConfig, importProvidersFrom } from '@angular/core';
import { provideRouter } from '@angular/router';
import { provideFirebaseApp, initializeApp } from '@angular/fire/app';
import { provideFirestore, getFirestore } from '@angular/fire/firestore';
import { provideAuth, getAuth } from '@angular/fire/auth';

const firebaseConfig = {
apiKey: "YOUR_API_KEY",
authDomain: "YOUR_PROJECT.firebaseapp.com",
projectId: "YOUR_PROJECT_ID",
storageBucket: "YOUR_PROJECT.appspot.com",
messagingSenderId: "YOUR_SENDER_ID",
appId: "YOUR_APP_ID"
};

export const appConfig: ApplicationConfig = {
providers: [
provideRouter([]),
provideFirebaseApp(() => initializeApp(firebaseConfig)),
provideFirestore(() => getFirestore()),
provideAuth(() => getAuth()),
]
};
```

Step 3: Create a Service for Firebase Operations

```
// src/app/services/student.service.ts
import { Injectable } from '@angular/core';
import { Firestore, collection, addDoc, getDocs, updateDoc, deleteDoc, doc, query, where,
onSnapshot } from '@angular/fire/firestore';
import { Observable } from 'rxjs';

export interface Student {
id?: string;
name: string;
email: string;
rollNo: string;
course: string;
```

```
createdAt?: Date;
}

@Injectable({
providedIn: 'root'
})
export class StudentService {
constructor(private firestore: Firestore) {}

// Create
async addStudent(student: Student): Promise<string> {
try {
const docRef = await addDoc(collection(this.firestore, 'students'), {
...student,
createdAt: new Date()
});
return docRef.id;
} catch (error) {
console.error('Error adding student:', error);
throw error;
}
}

// Read All
async getAllStudents(): Promise<Student[]> {
try {
const querySnapshot = await getDocs(collection(this.firestore, 'students'));
return querySnapshot.docs.map(doc => ({
id: doc.id,
...doc.data() as any
}));
} catch (error) {
console.error('Error fetching students:', error);
throw error;
}
}

// Read Single
async getStudentById(id: string): Promise<Student | null> {
try {
const docSnap = await getDocs(query(
collection(this.firestore, 'students'),
where('id', '==', id)
));
```

```
if (!docSnap.empty) {
  return { id: docSnap.docs[0].id, ...docSnap.docs[0].data() as any };
}
return null;
```

```
} catch (error) {
  console.error('Error fetching student:', error);
  throw error;
}

}

// Update
async updateStudent(id: string, data: Partial<Student>): Promise<void> {
try {
await updateDoc(doc(this.firestore, 'students', id), data);
} catch (error) {
console.error('Error updating student:', error);
throw error;
}
}

// Delete
async deleteStudent(id: string): Promise<void> {
try {
await deleteDoc(doc(this.firestore, 'students', id));
} catch (error) {
console.error('Error deleting student:', error);
throw error;
}
}

// Real-time subscription
subscribeToStudents(callback: (students: Student[]) => void): () => void {
const unsubscribe = onSnapshot(
collection(this.firestore, 'students'),
(snapshot) => {
const students = snapshot.docs.map(doc => ({
id: doc.id,
...doc.data() as any
}));
callback(students);
},
(error) => console.error('Error subscribing:', error)
);
return unsubscribe;
}
}
```

Step 4: Create a Component Using the Service

```
// src/app/components/student-form/student-form.component.ts
import { Component, OnInit } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';
import { StudentService, Student } from '../../services/student.service';

@Component({
selector: 'app-student-form',
standalone: true,
imports: [CommonModule, FormsModule],
templateUrl: './student-form.component.html',
styleUrls: ['./student-form.component.css']
})
export class StudentFormComponent implements OnInit {
students: Student[] = [];

form = {
name: '',
email: '',
rollNo: '',
course: ''
};

selectedStudent: Student | null = null;
unsubscribe: (() => void) | null = null;

constructor(private studentService: StudentService) {}

ngOnInit() {
// Real-time subscription
this.unsubscribe = this.studentService.subscribeToStudents((students) => {
this.students = students;
});
}

async addStudent() {
if (this.form.name && this.form.email && this.form.rollNo && this.form.course) {
try {
await this.studentService.addStudent(this.form as Student);
this.resetForm();
} catch (error) {
console.error('Error:', error);
}
}
}

selectStudent(student: Student) {
this.selectedStudent = student;
this.form = { ...student };
}
async updateStudent() {
if (this.selectedStudent?.id) {
try {
await this.studentService.updateStudent(this.selectedStudent.id, this.form);
this.resetForm();
this.selectedStudent = null;
} catch (error) {
console.error('Error:', error);
}
}
}

async deleteStudent(id: string | undefined) {
if (id && confirm('Are you sure?')) {
try {
await this.studentService.deleteStudent(id);
} catch (error) {
console.error('Error:', error);
}
}
}

resetForm() {
this.form = { name: '', email: '', rollNo: '', course: '' };
}

ngOnDestroy() {
if (this.unsubscribe) {
this.unsubscribe();
}
}
}
```

Student Management

{{ selectedStudent ? 'Edit' : 'Add' }} Student

{{ selectedStudent ? 'Update' : 'Add' }} Cancel

Students List

Name	Email	Roll No	Course	Actions

{{ student.name }}	{{ student.email }}	{{ student.rollNo }}	{{ student.course }}	Edit Delete

```
/* student-form.component.css */
.container {
max-width: 1000px;
margin: 0 auto;
padding: 20px;
font-family: Arial, sans-serif;
}

.form-section {
background: #f5f5f5;
padding: 20px;
margin-bottom: 30px;
border-radius: 8px;
}

.input-field {
display: block;
width: 100%;
padding: 10px;
margin-bottom: 10px;
border: 1px solid #ddd;
border-radius: 4px;
font-size: 14px;
}

.btn-primary, .btn-secondary, .btn-edit, .btn-delete {
padding: 10px 15px;
margin-right: 10px;
border: none;
border-radius: 4px;
cursor: pointer;
font-size: 14px;
}

.btn-primary {
background: #007bff;
color: white;
}

.btn-secondary {
background: #6c757d;
color: white;
}
```

```
.btn-edit {
background: #28a745;
color: white;
}

.btn-delete {
background: #dc3545;
color: white;
}

.table {
width: 100%;
border-collapse: collapse;
background: white;
}

.table th, .table td {
padding: 12px;
text-align: left;
border-bottom: 1px solid #ddd;
}

.table th {
background: #333;
color: white;
}

.table tr:hover {
background: #f9f9f9;
}
```

4.2.2 Connecting Firebase Admin SDK in Express for Backend Access

Why Firebase Admin SDK?

- **Server-to-Server:** Secure backend operations
- **Bypass Security Rules:** Admin has full database access
- **Custom Logic:** Implement business logic on server
- **Batch Operations:** Process large data efficiently

Setup Express with Firebase Admin

Step 1: Create Firebase Service Account

1. Go to Firebase Console → **Project Settings**
2. Click **"Service Accounts"** tab
3. Click **"Generate New Private Key"**
4. Save JSON file securely (contains credentials)

Step 2: Install Firebase Admin SDK

```
npm install firebase-admin
```

Step 3: Initialize Firebase Admin in Express

```
// config/firebaseAdmin.js
import admin from 'firebase-admin';
import fs from 'fs';
import path from 'path';
import { fileURLToPath } from 'url';

const __filename = fileURLToPath(import.meta.url);
const __dirname = path.dirname(__filename);

const serviceAccount = JSON.parse(
fs.readFileSync(path.join(__dirname, '../serviceAccountKey.json'), 'utf8')
);

admin.initializeApp({
credential: admin.credential.cert(serviceAccount),
projectId: serviceAccount.project_id
});

export const db = admin.firestore();
export const auth = admin.auth();
```

Step 4: Create Express Routes for CRUD

```
// routes/studentRoutes.js
import express from 'express';
import { db } from '../config/firebaseAdmin.js';

const router = express.Router();

// CREATE Student
router.post('/students', async (req, res) => {
try {
const { name, email, rollNo, course } = req.body;

const docRef = await db.collection('students').add({
 name,
 email,
 rollNo,
 course,
 createdAt: admin.firestore.FieldValue.serverTimestamp()
});

res.status(201).json({
 id: docRef.id,
 message: 'Student added successfully'
});

} catch (error) {
res.status(500).json({ error: error.message });
}
});

// READ All Students
router.get('/students', async (req, res) => {
try {
const snapshot = await db.collection('students').get();
const students = [];

snapshot.forEach(doc => {
 students.push({
  id: doc.id,
  ...doc.data()
 });
});

res.json(students);

} catch (error) {
res.status(500).json({ error: error.message });
}
});

// READ Single Student
router.get('/students/:id', async (req, res) => {
try {
const doc = await db.collection('students').doc(req.params.id).get();

if (!doc.exists) {
 return res.status(404).json({ message: 'Student not found' });
}

res.json({ id: doc.id, ...doc.data() });
```

```
} catch (error) {
res.status(500).json({ error: error.message });
}
});

// UPDATE Student
router.put('/students/:id', async (req, res) => {
try {
const { name, email, course } = req.body;

await db.collection('students').doc(req.params.id).update({
  name,
  email,
  course,
  updatedAt: admin.firestore.FieldValue.serverTimestamp()
});

res.json({ message: 'Student updated successfully' });

} catch (error) {
res.status(500).json({ error: error.message });
}
});

// DELETE Student
router.delete('/students/:id', async (req, res) => {
try {
await db.collection('students').doc(req.params.id).delete();
res.json({ message: 'Student deleted successfully' });
} catch (error) {
res.status(500).json({ error: error.message });
}
});

// BATCH Operations
router.post('/students/batch/add', async (req, res) => {
try {
const batch = db.batch();
const students = req.body.students; // Array of student objects

students.forEach(studentData => {
  const docRef = db.collection('students').doc();
  batch.set(docRef, {
    ...studentData,
    createdAt: admin.firestore.FieldValue.serverTimestamp()
  });
});

await batch.commit();
```

```
res.status(201).json({ message: `${students.length} students added` });

} catch (error) {
res.status(500).json({ error: error.message });
}
});

export default router;
```

Step 5: Complete Express Server Setup

```
// server.js
import express from 'express';
import cors from 'cors';
import dotenv from 'dotenv';
import studentRoutes from './routes/studentRoutes.js';

dotenv.config();

const app = express();
const PORT = process.env.PORT || 5000;

// Middleware
app.use(cors());
app.use(express.json());

// Routes
app.use('/api', studentRoutes);

// Health check
app.get('/health', (req, res) => {
res.json({ status: 'Server is running' });
});

// Error handler
app.use((err, req, res, next) => {
console.error(err.stack);
res.status(500).json({ error: 'Internal server error' });
});

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

Security: Protecting Service Account Key

.gitignore

```
serviceAccountKey.json
.env
```

.env file (local development only)

```
FIREBASE_PROJECT_ID=your_project_id
PORT=5000
```

For production: Use environment variables in cloud deployment (Firebase Functions, Google Cloud Run, Heroku, etc.)

4.2.3 Basic Deployment using Firebase Hosting (for Frontend)

Deploy Angular/React Frontend to Firebase Hosting

Step 1: Install Firebase CLI

```
npm install -g firebase-tools
```

Step 2: Login and Initialize Firebase

```
firebase login
firebase init hosting
```

During setup:

- Select your Firebase project
- Set public directory: `dist` (Angular) or `build` (React)
- Configure as single-page app: **Yes**
- Automatic builds with GitHub: Optional

Step 3: Build and Deploy Angular

Build for production

```
ng build --configuration production
```

Deploy to Firebase Hosting

```
firebase deploy --only hosting
```

Step 4: Deploy React (Vite or Create React App)

Build for production

```
npm run build
```

Deploy to Firebase Hosting

```
firebase deploy --only hosting
```

Step 5: Verify Deployment

```
firebase open hosting:site
```

Your app is now live at: https://YOUR_PROJECT.firebaseapp.com

Environment Configuration for Deployment

```
// src/config/firebase.js (for React)
const firebaseConfig = process.env.NODE_ENV === 'production'
? {
apiKey: process.env.REACT_APP_FIREBASE_API_KEY,
projectId: process.env.REACT_APP_FIREBASE_PROJECT_ID,
// ...
}
: {
// Development config
};
```

4.3 Overview of React Frontend for Full Stack Developers

4.3.1 Setting Up a Basic React App Using Vite or Create React App

React: Key Concepts for Full-Stack Developers

React is a **JavaScript library for building user interfaces** using reusable components. For full-stack developers, React complements backend APIs seamlessly.

Method 1: Create React App (Traditional, Easier)

```
npx create-react-app my-react-app
cd my-react-app
npm start
```

Pros: Zero config, helpful error messages
Cons: Slower builds, larger bundle

Method 2: Vite (Modern, Faster)

```
npm create vite@latest my-react-app -- --template react
cd my-react-app
npm install
npm run dev
```

Pros: Lightning fast, ES modules, smaller bundle
Cons: Requires more setup knowledge

Project Structure (Vite)

```
my-react-app/
├── src/
│ ├── components/
│ │ ├── StudentForm.jsx
│ │ └── StudentList.jsx
│ ├── hooks/
│ │ └── useStudents.js
│ ├── App.jsx
│ ├── App.css
│ └── main.jsx
├── public/
├── index.html
├── vite.config.js
├── package.json
└── .env
```

Install Firebase in React Project

```
npm install firebase
```

Create src/firebase.js:

```
import { initializeApp } from 'firebase/app';
import { getFirestore } from 'firebase/firestore';
import { getAuth } from 'firebase/auth';

const firebaseConfig = {
apiKey: process.env.REACT_APP_API_KEY,
authDomain: process.env.REACT_APP_AUTH_DOMAIN,
projectId: process.env.REACT_APP_PROJECT_ID,
storageBucket: process.env.REACT_APP_STORAGE_BUCKET,
messagingSenderId: process.env.REACT_APP_MESSAGING_SENDER_ID,
appId: process.env.REACT_APP_APP_ID
};

const app = initializeApp(firebaseConfig);
export const db = getFirestore(app);
export const auth = getAuth(app);
```

4.3.2 Understanding JSX, Functional Components, and Hooks (useState, useEffect)

What is JSX?

JSX is **JavaScript syntax that looks like HTML**. It compiles to JavaScript function calls:

```
// JSX
const element =
```

Hello, {name}!

```
;

// Compiles to
const element = React.createElement('h1', { className: 'title' }, Hello, ${name}!);
```

Functional Components

```
// Simple Component
function Welcome() {
return
```

Welcome to React!

```
;
}
```

```
// With Props
function StudentCard({ name, email, course }) {
return (
```

{name}

```
Email: {email}

Course: {course}

);
}

// Component composition
function StudentList() {
const students = [
{ id: 1, name: 'Rahul', email: 'rahul@example.com', course: 'React' },
{ id: 2, name: 'Priya', email: 'priya@example.com', course: 'Angular' }
];

return (
```

Students

```
{students.map(student => (
<StudentCard key={student.id} {...student} />
))}

);
}
```

Hook 1: useState (State Management)

useState allows functional components to have local state:

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

function Counter() {
// Declare state: [variable, setter] = useState(initialValue)
const [count, setCount] = useState(0);
const [name, setName] = useState('Rahul');

return (
<div>
```

Count: {count}

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

```
 <input
   value={name}
   onChange={(e) => setName(e.target.value)}
   placeholder="Enter name"
 />
 <p>Hello, {name}</p>
</div>
```

);
}

Hook 2: useEffect (Side Effects)

useEffect runs code after component renders. Used for API calls, subscriptions:

```
import { useState, useEffect } from 'react';

function StudentFetcher() {
const [students, setStudents] = useState([]);
const [loading, setLoading] = useState(true);
const [error, setError] = useState(null);

// Effect runs once after mount (empty dependency array)
useEffect(() => {
fetchStudents();
}, []);

const fetchStudents = async () => {
try {
const response = await fetch('http://localhost:5000/api/students');
const data = await response.json();
setStudents(data);
setLoading(false);
} catch (err) {
setError(err.message);
setLoading(false);
}
};

if (loading) return

Loading...

;
if (error) return
Error: {error}

;

return (

);
}
```

Dependency Array Rules

```
// Runs once on mount
useEffect(() => { }, []);

// Runs when dependency changes
useEffect(() => { }, [count]);

// Runs after every render
useEffect(() => { });

// Cleanup function (runs on unmount)
useEffect(() => {
const unsubscribe = onSnapshot(collection(db, 'students'), snapshot => {
// Update state
});

return () => unsubscribe(); // Cleanup
}, []);
```

Advanced: Custom Hook for Firebase

```
// src/hooks/useFirebaseData.js
import { useState, useEffect } from 'react';
import { collection, onSnapshot } from 'firebase/firestore';
import { db } from '../firebase';

export function useFirebaseData(collectionName) {
const [data, setData] = useState([]);
const [loading, setLoading] = useState(true);
const [error, setError] = useState(null);

useEffect(() => {
const unsubscribe = onSnapshot(
collection(db, collectionName),
(snapshot) => {
const newData = snapshot.docs.map(doc => ({
id: doc.id,
...doc.data()
}));
setData(newData);
setLoading(false);
},
(err) => {
setError(err.message);
setLoading(false);
}
);

return () => unsubscribe();

}, [collectionName]);

return { data, loading, error };
}

// Usage
function Students() {
const { data: students, loading, error } = useFirebaseData('students');

if (loading) return

Loading...

;
if (error) return

Error: {error}

;

return (



{students.map(student => (


{student.name}


))}



);
}
```

4.3.3 Integrating Firebase in a React App for Data Access and Authentication

Complete React + Firebase CRUD App

Component 1: StudentForm.jsx

```
// src/components/StudentForm.jsx
import { useState } from 'react';
import { collection, addDoc, updateDoc, doc } from 'firebase/firestore';
import { db } from '../firebase';

export function StudentForm({ onStudentAdded, editingStudent, onEditComplete }) {
const [formData, setFormData] = useState({
name: editingStudent?.name || '',
email: editingStudent?.email || '',
rollNo: editingStudent?.rollNo || '',
course: editingStudent?.course || ''
});

const handleChange = (e) => {
const { name, value } = e.target;
setFormData(prev => ({ ...prev, [name]: value }));
};

const handleSubmit = async (e) => {
e.preventDefault();

try {
  if (editingStudent?.id) {
    // Update
    await updateDoc(doc(db, 'students', editingStudent.id), formData);
    onEditComplete();
  } else {
    // Add new
    await addDoc(collection(db, 'students'), {
      ...formData,
      createdAt: new Date()
    });
  }

  setFormData({ name: '', email: '', rollNo: '', course: '' });
  onStudentAdded?.();
} catch (error) {
  console.error('Error:', error);
  alert('Error saving student');
}

};
```

return (
<form onSubmit={handleSubmit} className="form">

{editingStudent ? 'Edit' : 'Add'} Student

```
<input
  type="text"
  name="name"
  placeholder="Student Name"
  value={formData.name}
  onChange={handleChange}
  required
/>

<input
  type="email"
  name="email"
  placeholder="Email"
  value={formData.email}
  onChange={handleChange}
  required
/>

<input
  type="text"
  name="rollNo"
  placeholder="Roll Number"
  value={formData.rollNo}
  onChange={handleChange}
  required
/>

<input
  type="text"
  name="course"
  placeholder="Course"
  value={formData.course}
  onChange={handleChange}
  required
/>

<button type="submit" className="btn-primary">
  {editingStudent ? 'Update' : 'Add Student'}
</button>

{editingStudent && (
  <button type="button" onClick={onEditComplete} className="btn-secondary">
    Cancel
```

```
   </button>
  )}
</form>

);
}
```

Component 2: StudentList.jsx

```
// src/components/StudentList.jsx
import { useState, useEffect } from 'react';
import { collection, onSnapshot, deleteDoc, doc } from 'firebase/firestore';
import { db } from '../firebase';

export function StudentList({ onEdit, refreshKey }) {
const [students, setStudents] = useState([]);
const [loading, setLoading] = useState(true);

useEffect(() => {
const unsubscribe = onSnapshot(
collection(db, 'students'),
(snapshot) => {
const newStudents = snapshot.docs.map(docSnap => ({
id: docSnap.id,
...docSnap.data()
}));
setStudents(newStudents);
setLoading(false);
},
(error) => {
console.error('Error fetching students:', error);
setLoading(false);
}
);

return () => unsubscribe();

}, [refreshKey]);

const handleDelete = async (id) => {
if (window.confirm('Are you sure you want to delete this student?')) {
try {
await deleteDoc(doc(db, 'students', id));
} catch (error) {
console.error('Error deleting:', error);
alert('Error deleting student');
}
}
};
```

if (loading) return

Loading students...

;

```
return (
<div className="student-list">
```

Students List ({students.length})

```
{students.length === 0 ? (
  <p>No students yet. Add one to get started!</p>
) : (
  <table className="table">
    <thead>
      <tr>
        <th>Name</th>
        <th>Email</th>
        <th>Roll No</th>
        <th>Course</th>
        <th>Actions</th>
      </tr>
    </thead>
    <tbody>
      {students.map(student => (
        <tr key={student.id}>
          <td>{student.name}</td>
          <td>{student.email}</td>
          <td>{student.rollNo}</td>
          <td>{student.course}</td>
          <td>
            <button
              onClick={() => onEdit(student)}
              className="btn-edit"
            >
              Edit
            </button>
            <button
              onClick={() => handleDelete(student.id)}
              className="btn-delete"
            >
              Delete
            </button>
          </td>
        </tr>
      ))}
    </tbody>
  </table>
 )}
</div>
```

);
}

Main App Component

// src/App.jsx
import { useState } from 'react';
import { StudentForm } from './components/StudentForm';
import { StudentList } from './components/StudentList';
import './App.css';

function App() {
const [editingStudent, setEditingStudent] = useState(null);
const [refreshKey, setRefreshKey] = useState(0);

const handleEdit = (student) => {
setEditingStudent(student);
};

const handleEditComplete = () => {
setEditingStudent(null);
setRefreshKey(prev => prev + 1);
};

const handleStudentAdded = () => {
setRefreshKey(prev => prev + 1);
};

return (
<div className="app">

Student Management System

Built with React + Firebase

```
 <div className="container">
   <div className="form-section">
     <StudentForm
       onStudentAdded={handleStudentAdded}
       editingStudent={editingStudent}
       onEditComplete={handleEditComplete}
     />
   </div>
```

```
  <div className="list-section">
   <StudentList
    onEdit={handleEdit}
    refreshKey={refreshKey}
   />
  </div>
 </div>
</div>
```

);
}

export default App;

Styling

/* src/App.css */

- {
  margin: 0;
  padding: 0;
  box-sizing: border-box;
  }

body {
font-family: -apple-system, BlinkMacSystemFont, 'Segoe UI', 'Roboto', 'Oxygen',
'Ubuntu', 'Cantarell', 'Fira Sans', 'Droid Sans', 'Helvetica Neue',
sans-serif;
background: #f5f5f5;
}

.app {
min-height: 100vh;
background: linear-gradient(135deg, #667eea 0%, #764ba2 100%);
}

.header {
background: rgba(0, 0, 0, 0.8);
color: white;
padding: 30px;
text-align: center;
}

.header h1 {
font-size: 2.5em;
margin-bottom: 10px;
}

```
.container {
max-width: 1200px;
margin: 30px auto;
display: grid;
grid-template-columns: 1fr 1fr;
gap: 30px;
padding: 0 20px;
}

.form-section, .list-section {
background: white;
padding: 30px;
border-radius: 8px;
box-shadow: 0 4px 6px rgba(0, 0, 0, 0.1);
}

.form h2, .student-list h2 {
margin-bottom: 20px;
color: #333;
}

.form input {
width: 100%;
padding: 12px;
margin-bottom: 15px;
border: 1px solid #ddd;
border-radius: 4px;
font-size: 14px;
transition: border-color 0.3s;
}

.form input:focus {
outline: none;
border-color: #667eea;
}

button {
padding: 12px 20px;
border: none;
border-radius: 4px;
cursor: pointer;
font-size: 14px;
font-weight: bold;
transition: all 0.3s;
}

.btn-primary {
width: 100%;
background: #667eea;
color: white;
margin-bottom: 10px;
}

.btn-primary:hover {
background: #5568d3;
transform: translateY(-2px);
}

.btn-secondary {
width: 100%;
background: #6c757d;
color: white;
}

.btn-edit {
background: #28a745;
color: white;
padding: 8px 12px;
margin-right: 5px;
}

.btn-delete {
background: #dc3545;
color: white;
padding: 8px 12px;
}

.table {
width: 100%;
border-collapse: collapse;
}

.table th {
background: #333;
color: white;
padding: 12px;
text-align: left;
}

.table td {
padding: 12px;
border-bottom: 1px solid #ddd;
}

.table tr:hover {
background: #f9f9f9;
}
```

```
@media (max-width: 768px) {
.container {
grid-template-columns: 1fr;
}

.header h1 {
font-size: 1.8em;
}
}
```

Firebase Authentication in React

```
// src/hooks/useAuth.js
import { useState, useEffect } from 'react';
import {
createUserWithEmailAndPassword,
signInWithEmailAndPassword,
signOut,
onAuthStateChanged
} from 'firebase/auth';
import { auth } from '../firebase';

export function useAuth() {
const [user, setUser] = useState(null);
const [loading, setLoading] = useState(true);

useEffect(() => {
const unsubscribe = onAuthStateChanged(auth, (currentUser) => {
setUser(currentUser);
setLoading(false);
});

return () => unsubscribe();

}, []);

const signup = async (email, password) => {
return createUserWithEmailAndPassword(auth, email, password);
};

const login = async (email, password) => {
return signInWithEmailAndPassword(auth, email, password);
};

const logout = async () => {
return signOut(auth);
};
return { user, loading, signup, login, logout };
}

// Usage in component
function AuthComponent() {
const { user, loading, signup, login, logout } = useAuth();

if (loading) return

Loading...

;

return (


{user ? (

Welcome, {user.email}

Logout

) : (

Please login

)}


);
}
```