Full - Stack

ANGULAR

1.1 Architecture

- Modular with NgModules and feature modules
- Component-based (isolated, reusable UI pieces)
- Dependency injection: services injected via constructor
- Change detection strategies: Default, OnPush
- Routing: SPA navigation with guards and resolvers

1.2 Core Angular Concepts

- **Component**: UI + logic, decorated with `@Component`
- **Directive**: structural (e.g., `ngIf`) and attribute (e.g., `ngClass`)
- **Pipe**: transforms data in templates (e.g., date formatting)
- **Service**: reusable business/data logic with DI
- **Injector hierarchy**: controls service lifetime/scope

1.3 Data Binding

- Interpolation: `{{ value }}`
- Property binding: `[src]="imgUrl"`
- Event binding: `(click)="handleClick()"`
- Two-way binding: `[(ngModel)]="value"`
- Template reference variables: `#myInput`

1.4 Routing

- `RouterModule.forRoot()`

- Route parameters: `/product/:id`
- Child routes
- Lazy loading
- Guards: `CanActivate`, `CanLoad`
- Resolvers

1.5 Dependency Injection

- `@Injectable()` to mark services
- Tree-shakable `providedIn: 'root'`
- Injection tokens for custom DI keys
- Hierarchical injectors for scope
- Multi-providers

1.6 Reactive Patterns

- RxJS Observables for async streams
- Async pipes in templates
- Operators: map, mergeMap, switchMap, debounceTime, etc.
- Subjects and BehaviorSubjects
- Signals (Angular 17+)
- Subscription management with `takeUntil`, `unsubscribe`

1.7 Forms

- **Template-driven**: quick, simpler
- **Reactive**: testable, scalable
  - `FormGroup`, `FormControl`, `FormArray`
  - Validators & async validators
  - Dynamic forms

- Custom form controls with `ControlValueAccessor`

1.8 Advanced Angular

- Standalone components (Angular 14+)
- Dynamic component rendering (`ViewContainerRef`)
- Angular Universal (server-side rendering)
- Zone.js and NgZone
- Progressive Web Apps (PWA)
- Signals state management (Angular 17+)
- CDK features (drag-drop, virtual scroll)

1.9 Performance Tips

- Lazy loading modules
- OnPush change detection
- `trackBy` with `ngFor`
- Ahead-of-Time (AOT) compilation
- Tree shaking
- Preloading strategies
- Differential loading
- Consistent folder structure

1.10 Testing

- Jasmine / Karma for unit testing
- TestBed for DI testing
- Cypress / Playwright for E2E
- Mocks with `spyOn` and `HttpTestingController`
- Marble testing for observables

1.11 Angular Pitfalls to Avoid

- Forgetting `trackBy` on *ngFor
- Unmanaged subscriptions → memory leaks
- Business logic inside components instead of services
- Overly large modules
- Tightly coupled services

2️⃣ SPRING BOOT

2.1 Architecture

- Based on Spring Framework
- Auto-configuration with `@SpringBootApplication`
- Embedded servers (Tomcat/Jetty/Undertow)
- Starter dependencies
- Profile support (`dev`, `test`, `prod`)

2.2 Core Concepts

- `@RestController` to build APIs
- `@Service` for business logic
- `@Repository` for persistence
- `@Component` for generic beans
- `@Autowired` for DI
- `@Qualifier` for choosing specific beans

2.3 Spring Data JPA

- `JpaRepository` for CRUD

- Paging & sorting built-in
- Derived query methods
- `@Query` for custom queries
- DTO projections
- Entity lifecycle: `@PrePersist`, `@PreUpdate`

2.4 Spring Security

- Filter chain
- JWT token support
- OAuth2
- Role-based permissions
- Password encoding
- CORS config
- Method security with `@PreAuthorize`

2.5 Exception Handling

- `@ControllerAdvice` for global errors
- `@ExceptionHandler` for custom handlers
- Consistent error response DTOs
- API error status best practices

2.6 API Best Practices

- Versioning (e.g., `/api/v1`)
- DTOs over entities
- Validate with `@Valid`
- Consistent error structures
- Use `ResponseEntity` for status codes

- Swagger/OpenAPI for documentation

2.7 Testing

- JUnit 5 + Mockito
- Integration tests with `@SpringBootTest`
- MockMvc for API testing
- Testcontainers for real database testing
- Slices like `@WebMvcTest` for controller focus

2.8 DevOps & Deployment

- Maven or Gradle builds
- Docker containers
- Externalized configuration (YAML/env)
- Profiles for dev/prod separation
- Spring Boot Actuator for monitoring
- Prometheus / Grafana metrics
- Kubernetes support

2.9 Spring Boot Pitfalls to Avoid

- Putting business logic in controllers
- Exposing entities directly (use DTOs)
- Ignoring validation
- Hardcoding secrets
- Mixing concerns between layers

3️⃣ FULL-STACK INTEGRATION

- Enable CORS in Spring Boot

- Angular HttpInterceptor attaches JWT
- Route guards in Angular aligned with Spring Security
- Refresh token logic
- Consistent error messages
- E2E tests with Cypress on full stack
- API versioning
- Separate or monorepo deployment
- Microservices best practices (circuit breakers, fallback)

4️⃣ CHECKLIST

✅ Clean architecture

✅ Clear folder structure

✅ Consistent code style

✅ Unit + integration + E2E tests

✅ CI/CD pipelines

✅ Docker for reproducibility

✅ Proper secrets management

✅ Monitoring & logs

✅ Load testing

✅ Documentation

5️⃣ RESOURCES

- Angular Docs: angular.io
- Spring Boot Docs: spring.io/projects/spring-boot
- Spring Security: spring.io/projects/spring-security
- RxJS: rxjs.dev

- Cypress: cypress.io
- Docker: docker.com

6️⃣ TIPS

- Learn by building real projects
- Document as you go
- Break problems into small components
- Keep your code DRY (don’t repeat yourself)
- Regularly test & refactor
- Follow security best practices
- Share knowledge with team members

NODE JS:

NODE.JS

1.1 What is Node.js?

- Node.js is a **runtime** that runs JavaScript on the server side (outside the browser).
- It uses the **V8 engine** (from Chrome) to execute JS code.
- Its **event-driven, non-blocking I/O** model makes it very efficient for handling concurrent requests, especially in I/O-heavy systems (APIs, web servers).

1.2 Key Node.js Features

- **Single-threaded** with asynchronous callbacks
- Event loop architecture
- No built-in multithreading (but uses worker threads / clustering for concurrency)

- Lightweight and scalable
- Large ecosystem via npm
- JSON-based workflows make it easy to work with front-ends like Angular

---

1.3 When to Choose Node.js?

✅ When you want JavaScript across the entire stack

✅ Real-time apps (chats, streaming)

✅ High concurrency, low-latency systems

✅ Microservices with simple deployment

✅ JSON-heavy APIs

1.4 Pitfalls

❌ CPU-bound tasks (like complex data science) are slower in Node

❌ Too many nested callbacks (use `async/await` or promises)

❌ Over-reliance on large npm dependencies without checking security

---

2️⃣ API DEVELOPMENT WITH NODE.JS (EXPRESS.JS)

2.1 Why Express.js?

- Most popular minimalist framework on Node
- Built on top of HTTP core module
- Clean routing
- Middleware support
- Easily integrates with databases (MongoDB, PostgreSQL, etc.)

2.2 Express Basics

```
js
CopyEdit
```

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

app.get('/api/hello', (req, res) => {
  res.json({ message: 'Hello World' });
});

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

2.3 Middlewares

- Functions that run before your route handler
- Used for authentication, logging, error handling

```
js
CopyEdit
app.use((req, res, next) => {
  console.log(`Request: ${req.method} ${req.url}`);
  next();
});
```

2.4 Routing Patterns

- Organize routes with `express.Router()`
- Supports RESTful patterns
- Easy to create modular routes

2.5 Error Handling

```
js
CopyEdit
```

```
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).send('Something broke!');
});
```

2.6 Testing

✅ Supertest for route tests

✅ Mocha/Chai or Jest for unit testing

3️⃣ FULL STACK INTEGRATION: ANGULAR + NODE.JS

3.1 API and Frontend

- Node (Express) provides **REST API endpoints**
- Angular consumes them over HTTP (using `HttpClient`)
- JWT tokens can be generated in Node and passed to Angular
- Angular route guards protect views, while Express protects routes
- CORS must be enabled in Node for Angular (different ports in dev)

3.2 CORS Example in Express

```
js
CopyEdit
const cors = require('cors');
app.use(cors({ origin: 'http://localhost:4200' }));
```

3.3 Deployment

✅ Option 1: Serve Angular as static assets from Node

✅ Option 2: Deploy Node and Angular separately behind a reverse proxy (Nginx, Apache)

✅ Option 3: Dockerize both and orchestrate via Docker Compose or Kubernetes

---

4️⃣ COMPARING NODE.JS vs SPRING BOOT FOR BACKEND

Feature	Node.js (Express)	Spring Boot
Language	JavaScript	Java
Concurrency	Non-blocking event loop	Multi-threaded
Ecosystem	npm (huge, rapid)	Maven/Gradle (mature, stable)
Performance	Great for I/O	Great for CPU + enterprise
Deployment	Fast, minimal config	More heavyweight, but robust
Security	Middleware-based	Strong security features built-in
Use cases	Real-time, small services	Enterprise-grade, robust apps

---

Rule of thumb:

- **Node.js** → best for fast prototypes, real-time data, JSON-heavy services
- **Spring Boot** → best for complex business rules, large teams, strict security

---

5️⃣ FULL STACK API ARCHITECTURE

✅ Consistent error handling

✅ Validate all inputs (avoid trusting frontend)

✅ JWT authentication for Angular

✅ Role-based access in the backend

✅ Protect routes (Angular guards + backend auth)

✅ Use environment variables for secrets

✅ Swagger/OpenAPI for API documentation

✅ Separate DTO from domain models

✅ Monitor logs and metrics

6️⃣ TIPS

✅ Prefer `async/await` for clean async code

✅ Modularize routes in Node

✅ Use Mongoose/Sequelize or Prisma for DB modeling

✅ Containerize with Docker

✅ Write unit + integration tests

✅ Keep code DRY and SOLID

✅ Write documentation

✅ Build sample projects to test your knowledge

7️⃣ RESOURCES

- Node.js: nodejs.org
- Express.js: expressjs.com
- JWT: jwt.io
- Angular: angular.io
- Spring Boot: spring.io

REACT:

WHAT IS REACT?

- A **JavaScript library** for building user interfaces
- Created and maintained by Meta (Facebook)
- Component-based, declarative

- Uses **Virtual DOM** to improve performance
- Can be combined with other libraries for routing, state management, etc.
- Supports both **client-side** and **server-side rendering (Next.js)**

2️⃣ REACT CORE CONCEPTS

2.1 Components

✅ Functional components (recommended)

✅ Class components (older style, still relevant in legacy)

✅ Props for data input

✅ State for internal data

✅ Component lifecycle (with class)

✅ Hooks for functional (e.g., `useState`, `useEffect`)

2.2 JSX

- JavaScript + XML syntax
- Looks like HTML but compiles to JS
- Must return one root element
- Can embed JS with `{}`

```
jsx
CopyEdit
function Welcome() {
  return <h1>Hello React</h1>;
}
```

2.3 Props

- Read-only data passed from parent to child

- Promotes reusability
- Example:

```
jsx
CopyEdit
<MyButton label="Click me" />
```

2.4 State

- Internal data storage
- Triggers re-render on change
- Use `useState`:

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

2.5 Lifecycle / Hooks

- Hooks replace lifecycle methods
- Common hooks:
  - `useState` (state)
  - `useEffect` (side effects, lifecycle)
  - `useRef` (access DOM)
  - `useMemo` / `useCallback` (performance optimization)

3️⃣ ROUTING WITH REACT

✅ Use **React Router**

✅ Supports dynamic routes, nested routes, route guards

✅ Works with browser history

✅ Example:

```
jsx
CopyEdit
import { BrowserRouter, Routes, Route } from 'react-router-dom';

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

4️⃣ STATE MANAGEMENT

✅ Local state: `useState`

✅ Global state: Context API

✅ Advanced: Redux / Zustand / MobX

✅ Patterns:

- Lift state up
- Use context for shared state
- Use reducer patterns for complex flows

5️⃣ PERFORMANCE

✅ Virtual DOM updates only diff

✅ Code splitting with React.lazy

✅ Memoization with `React.memo`

✅ Avoid unnecessary renders

✅ Avoid anonymous functions in render

✅ Server-side rendering with Next.js

6️⃣ TESTING

✅ Unit tests with Jest

✅ React Testing Library for component tests

✅ Cypress / Playwright for E2E

✅ Mocks with MSW (Mock Service Worker)

7️⃣ REACT PITFALLS TO AVOID

❌ Directly modifying state

❌ Forgetting dependency arrays in `useEffect`

❌ Uncontrolled components when using forms

❌ Large prop drilling — better to lift state or use Context

❌ Ignoring performance bottlenecks in large lists

8️⃣ REACT + FULL STACK

✅ React consumes APIs via `fetch` or `axios`

✅ JWT auth — store tokens in `HttpOnly` cookies or secure storage

✅ React route guards (conditional rendering)

✅ CORS must be enabled in the backend

✅ SSR with Next.js if SEO is important

✅ Progressive Web App support (PWA with CRA or Next.js)

9️⃣ RESOURCES

- React: react.dev

- React Router: reactrouter.com
- Redux: redux.js.org
- Next.js: nextjs.org
- Testing: testing-library.com

TIPS

✅ Keep components pure

✅ Prefer functional components

✅ Split into small, reusable components

✅ Use prop-types or TypeScript

✅ Consistent naming and file structure

✅ Document public components

✅ Add tests for critical features

🌟 FULL STACK INTEGRATION (ANGULAR / REACT + SPRING BOOT / NODE)

- **Front-end (Angular/React)** → communicates with
- **Backend (Spring Boot/Node/Express)** → RESTful APIs
- Security with JWT
- Store tokens securely
- Angular uses `HttpInterceptor`, React uses Axios interceptors
- Backends must enable CORS
- Consistent error handling
- Use Swagger/OpenAPI for API documentation
- Use Docker to containerize both
- Use Nginx/Apache as a reverse proxy if needed

- Microservice splits possible with API Gateway (e.g., Kong, Spring Cloud Gateway)

ADVANTAGES AND DIS-ADVANTAGES:

ANGULAR

✅ Advantages

- Structured framework with batteries included
- Strong tooling (Angular CLI, Schematics)
- Powerful dependency injection
- Great support for large-scale enterprise apps
- Two-way data binding
- Strong TypeScript integration
- Backed by Google with long-term support
- Excellent testability (Jasmine, Karma)

❌ Disadvantages

- Steep learning curve (many concepts to learn)
- Verbose syntax (boilerplate-heavy)
- Less flexible compared to React for small projects
- Breaking changes between major versions
- Sometimes slower initial bundle size

2️⃣ REACT

✅ Advantages

- Very flexible, can integrate with any backend
- Large community and ecosystem

- Functional programming style with hooks
- Component reusability and composition
- React Native for mobile cross-platform apps
- Faster learning curve compared to Angular
- Supported by Meta (Facebook)

❌ **Disadvantages**

- Only handles view layer, so you must assemble your own stack
- Needs routing, state management, testing separately
- Frequent updates may break libraries
- Prop drilling can get complex
- JSX may be strange for new developers

3️⃣ NODE.JS (EXPRESS)

✅ **Advantages**

- JavaScript across full stack
- Extremely fast for I/O heavy apps
- Huge npm ecosystem
- Lightweight and simple to get started
- Easy JSON handling
- Great for real-time apps (sockets, chats)
- Non-blocking async programming

❌ **Disadvantages**

- Single-threaded (not good for CPU-heavy tasks)
- Callback hell (though mostly solved by async/await now)
- Still needs discipline to organize large-scale architecture
- Less mature security tools compared to Java Spring

- Requires extra care for error handling and validations

4️⃣ SPRING BOOT

✅ Advantages

- Java ecosystem, enterprise-grade
- Robust security with Spring Security
- Strict layered architecture (clean code)
- Lots of plugins and integrations
- Great support for databases, transactions
- Good for complex business logic
- Supported by a massive community
- Profiles for easy environment management
- Highly scalable

❌ Disadvantages

- Higher memory footprint than Node
- Slower cold start (JVM startup time)
- More configuration for advanced setups
- Steeper learning curve if new to Java
- Deployment usually heavier than Node.js

5️⃣ FULL STACK INTEGRATION (ALL TOGETHER)

✅ Advantages

- Angular or React provide dynamic, modern front-ends
- Spring Boot or Node.js provide solid back-end APIs
- JWT for secure user sessions
- Microservices patterns possible

- Docker/Kubernetes for easy deployment
- End-to-end testability
- Developer flexibility (choose the right tool for the job)

❌ **Disadvantages**

- Complex to manage multiple frameworks
- More moving parts (CI/CD, version upgrades)
- Security must be handled on both sides
- Learning curve is higher if you want to master *everything*
- Consistency in team standards can be challenging