Week	Main Topic	Key Topics Covered
1	Advanced Java Foundations + Kotlin Fundamental + Clean Code + Architecture Basics	Advanced OOPS
		Java Collections & Data Handling
		Concurrency & Multithreading
		Functional Programming with Java Streams
		Kotlin fundamentals
		Difference and use of Java 17 features,
		Solid Principles
		Layered/Hexagonal Architecture understanding.
2	Spring Boot Advanced Backend Development	Spring Boot Internals & Advanced Usage
		Dependency Injection (DI) & Inversion of Control (IoC)
		REST API Design – Beyond CRUD
		Spring Boot 3.x & Modern Spring Features

		JPA / Hibernate – ORM with a Design Mindset
		Spring Data – Power with Responsibility
		Security with Spring Security & JWT
		Integrations & External Communication
		Testing – Building Confidence in Code
		Swagger / OpenAPI – API Contract as a First-Class Citizen
3	Advanced Angular + TypeScript Mastery – Course Structure	TypeScript Deep Dive
		Angular Architecture
		Components & Services
		Forms
		Routing & Navigation
		Angular 15 Features
		State Management (RxJS / NgRx)
		UI Frameworks
		Testing

4	Full-Stack Integration + Real-World Patterns	Connecting Angular & Spring Boot
		End-to-End CRUD Flows
		Authentication Workflow
		File Handling
		Pagination & Filtering
		Role-Based UI & Authorization
		API Performance Considerations
		End-to-End (E2E) Testing
6	Capstone Project (Real Production-like Full-Stack Build)	Full-Stack Application Design
		Backend Development (Java / Spring Boot)
		Frontend Development (Angular)
		Database Design (PostgreSQL)
		Authentication & Authorization (JWT)
		End-to-End Workflows
		Testing Strategy
		Containerization (Docker)
		Deployment & Environment Setup
		Observability & Production Readiness

Learning Goal Summary	Indicative Practicals / Exercise
Revisiting OOP principles with real-world backend use cases to build extensible and maintainable systems	Refactor a tightly coupled service into extensible, testable components
Internal workings of collections to gain design-Level understanding to choose the right collection based on performance, concurrency, and memory considerations	Optimize a data-processing module by choosing correct collections
Common concurrency issues in production systems and how to avoid them	Fix race conditions in a simulated order-processing system
Understanding When to use (and not use) streams in real applications & Writing expressive, readable, and performant stream pipelines	Convert imperative logic to streams; compare readability & performance
Writing idiomatic Kotlin for backend APIs & Interoperability between Java and Kotlin in mixed codebases	Rewrite a Java service class in idiomatic Kotlin
Comparing legacy Java approaches with modern alternatives with stress on when and how to adopt newer features	Modernize legacy Java code using records, switch expressions, etc.
Understanding beyon definitions & Applying principles to improve flexibility, testability, and extensibility	Identify SOLID violations in existing code and refactor
Designing systems that isolate business logic from frameworks	Design a small service using Layered vs Hexagonal architecture
Move beyond "using Spring Boot" to understanding how it works under the hood, enabling developers to reason about performance, startup behavior, configuration, and framework-driven design decisions in real-world systems.	Debug and modify auto-configuration behavior in an existing Spring Boot service Analyze startup logs and optimize unnecessary bean loading
Understand DI as a design principle, not just an annotation-driven feature, to build loosely coupled, testable, and extensible backend components.	Refactor a tightly coupled service to use proper DI and interfaces Identify hidden dependencies and redesign for testability
Design REST APIs that are consumer-friendly, evolvable, and maintainable, suitable for real-world frontend, mobile, and third-party integrations.	Redesign a poorly structured REST API for clarity and consistency Introduce proper error contracts and status code usage
Understand what's new in Spring Boot 3.x and Spring Framework 6, and how these changes impact modern backend development and migration strategies.	Upgrade a Spring Boot 2.x application to 3.x Resolve breaking changes and refactor deprecated usage

Use JPA/Hibernate effectively by understanding how ORM works internally, avoiding performance pitfalls and modeling domain data correctly.	Identify and fix performance issues caused by poor JPA mappings Refactor entities to align better with domain boundaries
Leverage Spring Data for productivity without sacrificing control, clarity, or performance in complex data-access layers.	Optimize a repository layer suffering from over-abstraction Introduce custom queries for performance-critical paths
Understand security as a cross-cutting architectural concern, not just configuration, enabling developers to build secure, stateless backend APIs.	Implement JWT-based authentication for a REST service Debug and fix authorization issues across endpoints
Design backend services that safely and reliably integrate with external systems, APIs, and services.	Integrate with an external API and handle failure scenarios gracefully Refactor integration logic to isolate external dependencies
Shift mindset from “testing for coverage” to testing for confidence, correctness, and safe refactoring.	Add unit and integration tests to an existing service Refactor code to improve testability
Treat API contracts as design artifacts, enabling better collaboration between frontend, backend, and external consumers.	Design an API contract using Swagger/OpenAPI Improve an existing API’s documentation for clarity and usability
Use TypeScript’s type system as a design tool to model domain concepts, prevent bugs, and improve maintainability in large Angular applications	Refactor loosely typed Angular code to strongly typed, expressive contracts; model complex backend API responses using advanced types
Understand Angular as an application architecture platform to build scalable, modular, and long-lived frontend systems	Redesign a flat Angular project into a feature-based architecture; identify and fix architectural smells
Design components and services with clear responsibilities to improve readability, reusability, and ease of change	Break down large components into smaller focused components and services; refactor business logic out of UI components
Model complex user workflows using Angular forms that scale in validation, UX, and maintainability	Build a multi-step reactive form with custom and cross-field validators; refactor poorly structured form logic
Treat routing as a core architectural concern to enable clear navigation, lazy loading, and secure access	Introduce lazy-loaded routes and guards; redesign route configuration for better UX and performance
Adopt Angular 15 features intentionally to improve performance, developer experience, and architecture	Migrate a module-based app to standalone components; adopt typed forms to reduce runtime errors
Use reactive programming and state management only where it adds real value, avoiding unnecessary complexity	Refactor imperative async logic into RxJS pipelines; introduce NgRx selectively for a complex feature
Integrate UI frameworks without letting them dictate application architecture or business logic	Build a feature using a UI library while maintaining clean component boundaries and theming
Write reliable, intention-revealing frontend tests that support safe refactoring and long-term maintenance	Add meaningful unit tests for components and services; refactor code to improve testability

Understand how frontend and backend collaborate through clear contracts, enabling scalable and maintainable full-stack systems	Connect an Angular application to a Spring Boot backend using well-defined REST contracts; handle errors and loading states consistently
Design complete CRUD workflows that work reliably across UI, API, service, and database layers	Implement a full CRUD feature spanning Angular UI, Spring Boot REST API, service layer, and persistence
Understand authentication as a cross-cutting concern spanning UI, backend, and security layers	Implement a JWT-based login flow with secure token handling and protected routes
Handle file upload and download workflows safely and efficiently in real-world applications	Implement file upload/download with validation, size limits, and error handling
Design pagination and filtering that scales for large datasets and provides good user experience	Implement server-side pagination and filtering and consume it cleanly in Angular
Align frontend UI behavior with backend authorization rules without duplicating logic	Implement role-based access control affecting both UI visibility and backend endpoints
Reason about API performance from both frontend and backend perspectives	Identify and fix slow APIs using pagination, caching, and optimized data transfer
Validate real user workflows across the entire stack with confidence	Write E2E tests using Cypress (or equivalent) covering critical user journeys

Translate requirements into a well-structured, end-to-end full-stack system	Define domain model, API contracts, UI flows, and architecture for the capstone application
Build a clean, scalable backend aligned with product-grade engineering practices	Implement REST APIs with validation, security, persistence, and business logic
Develop a maintainable, responsive UI that consumes backend APIs correctly	Build feature-based Angular modules with forms, routing, and state management
Design relational schemas that support performance, integrity, and future evolution	Create tables, relationships, indexes, and migrations
Implement secure, stateless authentication workflows across the stack	Build login, role-based access control, and protected APIs
Ensure features work seamlessly across UI, API, service, and database layers	Implement full CRUD flows with validation and error handling
Build confidence through automated tests at multiple layers	Write unit, integration, and E2E tests covering critical flows
Package the application for consistent local and production environments	Dockerize frontend and backend with multi-stage builds
Experience real deployment workflows similar to production	Deploy the application using Docker (and optionally cloud infrastructure)
Reason about system behavior in production-like conditions	Add logging, basic monitoring, and error tracking

Key Focus Area
Object-Oriented Programming in Kotlin Data Classes & Special Classes Lambdas & Functional Programming Scope Functions (Core Kotlin Feature) Extension Functions & Properties Error Handling Generics Kotlin + Java Interoperability Basic Coroutines
Auto-configuration: when it helps, when it hurts Application context lifecycle & bean creation Profiles, configuration hierarchy, and environment-driven design Trade-offs of Spring Boot conventions vs explicit configuration
Constructor vs field vs setter injection (and why it matters) Bean scopes and lifecycle implications Designing components around interfaces and contracts Avoiding common DI anti-patterns in large codebases
Resource modeling vs endpoint-driven APIs HTTP semantics: verbs, status codes, idempotency Versioning strategies and backward compatibility Error handling and validation strategies
Jakarta EE namespace changes and implications Native image readiness and performance considerations Observability improvements Aligning Spring Boot 3 with Java 17 features

Entity lifecycle and persistence context Lazy vs eager loading trade-offs N+1 query problem and fetch strategies Mapping domain models vs database schemas
Derived queries vs explicit queries Pagination, sorting, and projections Custom repository implementations When not to use Spring Data abstractions
Authentication vs authorization concepts JWT-based stateless security Security filter chain and request lifecycle Common security misconfigurations in REST APIs
REST clients (WebClient / RestTemplate – trade-offs) Error handling, retries, and timeouts Designing integration boundaries Avoiding tight coupling with external systems
Unit vs integration vs slice testing Writing meaningful tests using JUnit & Mockito Mocking responsibly Testing Spring components without slow test suites
Designing APIs contract-first vs code-first OpenAPI annotations and structure API documentation as a communication tool Keeping documentation in sync with implementation
Generics, union/intersection types, mapped & conditional types, type inference vs explicit typing, avoiding any
Feature-based structure, standalone components, separation of concerns, testability
Responsibility-driven design, DI usage, lifecycle awareness, avoiding god components
Reactive forms, dynamic forms, custom validators, form state management
Lazy loading, guards, resolvers, URL design, deep linking
Standalone components, typed forms, performance optimizations, migration strategies
Observables, async flows, local vs global state, NgRx store/effects/selectors
Angular Material, theming, accessibility, responsive design, UI abstraction
Component testing, service testing, async & RxJS testing, test readability

API contracts, DTOs, HTTP semantics, error handling, frontend–backend responsibility boundaries
Data flow consistency, validation at multiple layers, DTO ↔ entity mapping
Auth flow design, token storage, route guards, backend authorization
Multipart handling, streaming, security considerations, frontend progress indicators
API pagination design, sorting strategies, frontend state synchronization
RBAC concepts, UI guards, backend enforcement, avoiding security leaks
Over-fetching vs under-fetching, payload size, backend query optimization
Test stability, environment setup, avoiding flaky tests, realistic test scenarios

Requirement analysis, domain modeling, system boundaries
Clean architecture, SOLID principles, layered / hexagonal design
Component design, state handling, UX consistency
Normalization, indexing, schema evolution
Security boundaries, token lifecycle, RBAC
Data consistency, error propagation, UX feedback
Test pyramid, testability, regression prevention
Environment parity, image optimization
Configuration management, environment separation
Operability, diagnostics, production mindset