Full-Stack Java + Angular + Azure Program

Duration: 15 Days | 120 Hours | 8 Hours/Day
Capstone Project: Banking Portal — Customer management, account operations, transaction history, deployed to Azure

WEEK 1 — Java Foundations + MS SQL + GitHub

Day 1 — Java OOP Fundamentals (8 hrs)

Learning Objectives

- Set up Java development environment (JDK 17+, IntelliJ IDEA)
- Apply core OOP principles: encapsulation, inheritance, polymorphism, abstraction
- Distinguish abstract classes from interfaces

Topics

- Java syntax: variables, data types, operators, control flow (if/for/while/switch)
- Classes, objects, constructors, access modifiers (public/private/protected)
- Inheritance and `super` keyword
- Method overriding vs method overloading
- Abstract classes and interfaces
- Introduction to capstone project domain and architecture overview

Lab: Banking Domain Model

- Create `Account` (abstract class) with `deposit()` and `withdraw()` abstract methods
- Implement `SavingsAccount` and `CurrentAccount` extending `Account`
- Create `Customer` class with name, email, phone fields
- Demonstrate polymorphism: `Account acc = new SavingsAccount(...)`, call `deposit()`
- Add `Printable` interface with `printSummary()` method, implement on both account types

Deliverable: Runnable Java project with domain model classes committed to GitHub

---

Day 2 — Collections, Generics, Exception Handling (8 hrs)

Learning Objectives

- Use Java Collections Framework for data management
- Write type-safe code with Generics
- Handle exceptions using checked, unchecked, and custom exceptions

Topics

- Collections: `List`, `Set`, `Map` — `ArrayList`, `HashMap`, `HashSet`, `LinkedList`
- Iterators and enhanced for-each loop
- `Collections` utility class: sort, reverse, min, max
- Generics: type parameters, bounded types (`<T extends Account>`), wildcards
- Exception hierarchy: `Throwable`, `Error`, `Exception`, `RuntimeException`
- `try-catch-finally`, multi-catch, try-with-resources
- Checked vs unchecked exceptions
- Custom exceptions with constructors

Lab: Transaction Manager

- Create `TransactionManager` class using `ArrayList<Transaction>`
- Store accounts in `HashMap<String, Account>` keyed by account number
- Create custom exceptions: `InsufficientFundsException`, `AccountNotFoundException`, `DuplicateAccountException`
- Implement `transfer(from, to, amount)` with proper exception handling
- Use `Set<String>` to track unique account numbers

Deliverable: Transaction operations with full exception handling

---

Day 3 — Java Modern: Streams, Lambdas, Optional (8 hrs)

Learning Objectives

- Write concise code with lambda expressions and functional interfaces

- Process collections using the Stream API
- Handle null values safely with Optional

Topics

- Lambda expressions: syntax, target typing
- Functional interfaces: `Predicate<T>`, `Function<T,R>`, `Consumer<T>`, `Supplier<T>`
- Method references: static, instance, constructor (`ClassName::method`)
- Stream pipeline: `filter()`, `map()`, `flatMap()`, `sorted()`, `distinct()`, `limit()`
- Terminal operations: `collect()`, `reduce()`, `count()`, `findFirst()`, `anyMatch()`
- `Collectors`: `toList()`, `groupingBy()`, `summingDouble()`, `joining()`
- `Optional<T>`: `of()`, `ofNullable()`, `orElse()`, `map()`, `ifPresent()`
- `LocalDate`, `LocalDateTime`, `DateTimeFormatter`

Lab: Transaction Analytics

- Filter transactions by date range using Stream + lambda
- Group transactions by account using `Collectors.groupingBy()`
- Calculate total deposits, total withdrawals using `reduce()`
- Find top 5 transactions by amount using `sorted()` + `limit()`
- Replace all `null` checks in previous code with `Optional`
- Generate a transaction summary report as formatted string

Deliverable: Analytics utility class using entirely functional-style code

---

Day 4 — MS SQL: Schema Design, Queries, Stored Procedures (8 hrs)

Learning Objectives

- Design normalized relational database schema
- Write complex SQL queries with joins and aggregations
- Create reusable stored procedures
- Understand indexing and query performance

Topics

- SQL Server setup (local) / Azure SQL connection
- DDL: `CREATE TABLE`, `ALTER TABLE`, `DROP TABLE`, constraints (PK, FK, UNIQUE, NOT NULL, DEFAULT)
- DML: `INSERT`, `UPDATE`, `DELETE`, `SELECT`
- Filtering: `WHERE`, `BETWEEN`, `IN`, `LIKE`, `IS NULL`
- Aggregations: `GROUP BY`, `HAVING`, `COUNT`, `SUM`, `AVG`, `MAX`, `MIN`
- Joins: `INNER JOIN`, `LEFT JOIN`, `RIGHT JOIN`, `SELF JOIN`
- Subqueries and Common Table Expressions (CTEs)
- Stored procedures: `CREATE PROCEDURE`, input/output parameters, `EXEC`
- Indexes: clustered vs non-clustered, when to index, `EXPLAIN` / execution plan

Lab: Banking Schema

```
    -- Build and populate these tables:
    -- customers (id, name, email, phone, created_at)
    -- accounts (id, account_number, type, balance, customer_id,
created_at)
    -- transactions (id, account_id, type, amount, description,
transaction_date)
```

- Write query: top 5 customers by total account balance
- Write query: transaction history for an account with running balance
- Write query: monthly transaction summary per account
- Create stored procedure: `sp_GetAccountSummary(@customerId INT)`
- Create stored procedure: `sp_TransferFunds(@fromAccId INT, @toAccId INT, @amount DECIMAL)`
- Add indexes on `transactions.account_id` and `accounts.customer_id`, compare execution plans

Deliverable: Full database schema script + stored procedures script

---

Day 5 — GitHub: Git Flow, Branching, PRs, GitHub Actions (8 hrs)

Learning Objectives

- Use Git for version control in a team workflow
- Apply feature branch strategy with pull requests
- Set up a basic CI pipeline with GitHub Actions

Topics

- Git fundamentals: `init`, `clone`, `add`, `commit`, `push`, `pull`, `fetch`
- Branching: `branch`, `checkout`, `merge`, `rebase` basics
- Merge conflict: causes, resolution strategies
- `.gitignore` patterns for Java/Maven projects
- GitHub: repositories, collaborators, branch protection rules
- Feature branch workflow: `main` → `develop` → `feature/xxx`
- Pull requests: creating, reviewing, requesting changes, approving
- Commit message conventions (Conventional Commits: `feat:`, `fix:`, `chore:`)
- GitHub Actions: YAML workflow structure, triggers (`push`, `pull_request`), jobs, steps
- Basic CI: checkout → set up JDK → `mvn compile`

Lab: Team Repository Setup

- Initialize capstone project as Maven project, push to GitHub
- Each participant creates a `feature/day5-yourname` branch
- Intentionally create a merge conflict in `pom.xml`, resolve it together
- Open a PR: add reviewer, add labels, comment, approve, merge
- Create `.github/workflows/ci.yml`:
  - Trigger: `push` to any branch, `pull_request` to `main`
  - Steps: checkout → JDK 17 → `mvn clean compile test`
  - Observe green/red check on PR

Deliverable: GitHub repo with branch protection + working CI workflow

---

WEEK 2 — Spring Boot Backend

Day 6 — Spring Boot Core: IoC, DI, First REST API (8 hrs)

Learning Objectives

- Understand Spring IoC container and Dependency Injection patterns
- Bootstrap a Spring Boot project from scratch
- Build and test GET and POST REST endpoints

Topics

- Spring Framework overview: IoC, DI, AOP, context
- Spring Boot: auto-configuration, starter dependencies, embedded Tomcat
- `@SpringBootApplication`, component scanning
- Stereotype annotations: `@Component`, `@Service`, `@Repository`, `@Controller`, `@RestController`
- `@Autowired`, constructor injection (preferred), `@Qualifier`
- `@RequestMapping`, `@GetMapping`, `@PostMapping`
- `@PathVariable`, `@RequestParam`, `@RequestBody`
- `ResponseEntity<T>`: status codes, headers, body
- Jackson: JSON serialization/deserialization, `@JsonProperty`, `@JsonIgnore`
- `application.properties`: server port, app name

Lab: Customer REST API (In-Memory)

- Generate project: Spring Initializr (Web, Lombok, DevTools)
- Create `Customer` model with Lombok (`@Data`, `@Builder`, `@NoArgsConstructor`)
- Create `CustomerService` with in-memory `List<Customer>`
- Create `CustomerController`:
  - `GET /api/customers` — return all customers
  - `GET /api/customers/{id}` — return one customer
  - `POST /api/customers` — create customer, return 201 Created
- Test all endpoints with Postman, inspect request/response JSON

Deliverable: Running Spring Boot app with testable Customer API

Day 7 — REST API: PUT/DELETE, Validation, Error Handling, DTOs (8 hrs)

Learning Objectives

- Complete CRUD REST operations
- Validate incoming request data
- Return consistent, structured error responses
- Use DTOs to decouple API contract from domain model

Topics

- `@PutMapping`, `@DeleteMapping`
- Bean Validation (Jakarta): `@Valid`, `@NotNull`, `@NotBlank`, `@Size`, `@Min`, `@Max`, `@Email`, `@Pattern`
- `@ControllerAdvice`, `@ExceptionHandler`
- Custom exception classes: `ResourceNotFoundException`, `ValidationException`
- Structured error response: `ApiError` (timestamp, status, message, errors list)
- DTO pattern: `CustomerRequest` (input), `CustomerResponse` (output)
- Manual mapping vs MapStruct
- HTTP status code semantics: 200, 201, 204, 400, 404, 409, 500

Lab: Full CRUD + Error Handling

- Add `PUT /api/customers/{id}` and `DELETE /api/customers/{id}` to CustomerController
- Create `CustomerRequest` DTO with validation annotations
- Create `CustomerResponse` DTO (exclude internal fields)
- Create `GlobalExceptionHandler` returning `ApiError` JSON
- Test: submit invalid email → get structured 400 error
- Create `AccountController` with full CRUD for accounts
- Test all error cases in Postman

Deliverable: Validated CRUD APIs with consistent error response structure

---

Day 8 — Spring Data JPA: Entities, Repositories, Relationships (8 hrs)

Learning Objectives

- Map Java entities to MS SQL tables using JPA annotations
- Use Spring Data JPA repositories for database operations
- Model one-to-many and many-to-one relationships

Topics

- JPA / Hibernate overview: ORM concept, persistence context
- `@Entity`, `@Table`, `@Column`, `@Id`, `@GeneratedValue`
- Spring Data JPA: `JpaRepository<T, ID>`, `CrudRepository`
- Derived query methods: `findByEmail()`, `findByCustomerIdAndType()`
- `@Query` with JPQL and native SQL
- Relationships: `@OneToMany`, `@ManyToOne`, `@JoinColumn`
- `CascadeType`, `FetchType.LAZY` vs `EAGER`
- MS SQL datasource configuration (mssql-jdbc driver)
- `@Transactional`: service layer transactions
- `application.properties`: datasource, JPA/Hibernate DDL, show-sql

Lab: Connect to MS SQL

```
spring.datasource.url=jdbc:sqlserver://localhost:1433;databaseName=bank
ingdb
    spring.datasource.driver-class-
name=com.microsoft.sqlserver.jdbc.SQLServerDriver
    spring.jpa.hibernate.ddl-auto=validate
```

- Create `Customer`, `Account`, `Transaction` entities matching Day 4 schema
- Create repositories: `CustomerRepository`, `AccountRepository`, `TransactionRepository`
- Implement `AccountService.getAccountsByCustomer(Long customerId)`
- Custom query: find accounts with balance greater than a threshold
- Implement `TransactionService.transfer()` using `@Transactional`
- Verify data persists in MS SQL after API calls

Deliverable: Backend fully connected to MS SQL with relational data

---

Day 9 — Spring Security: JWT Authentication + Role-Based Access (8 hrs)

Learning Objectives

- Secure REST APIs with stateless JWT authentication
- Implement register and login endpoints
- Restrict API access by user role

Topics

- Spring Security architecture: filter chain, `AuthenticationManager`, `UserDetailsService`
- `SecurityFilterChain` configuration (lambda DSL)
- `BCryptPasswordEncoder` for password hashing
- JWT structure: header, payload, signature
- `jjwt` library: generate token, validate token, extract claims
- `JwtAuthenticationFilter` extending `OncePerRequestFilter`
- `User` entity + `UserRepository` + `CustomUserDetailsService`
- `/auth/register` and `/auth/login` endpoints (publicly accessible)
- `@PreAuthorize("hasRole('ADMIN')")`, `@Secured`
- CORS configuration in `SecurityFilterChain`
- Role enum: `ROLE_ADMIN`, `ROLE_USER`

Lab: Secure the Banking API

- Add dependencies: `spring-boot-starter-security`, `jjwt-api`, `jjwt-impl`
- Create `User` entity with `email`, `password`, `role` fields
- Implement `/auth/register` (hash password, save user)
- Implement `/auth/login` (validate credentials, return JWT)
- Create `JwtAuthenticationFilter`, register in security chain
- Protect endpoints: authenticated users can read, only `ADMIN` can delete
- Test with Postman: register → login → copy token → call protected endpoint

Deliverable: Secured API — all endpoints require valid JWT except `/auth/**`

Day 10 — Testing + Azure App Service Deployment (8 hrs)

Learning Objectives

- Write unit tests with JUnit 5 and Mockito
- Write integration tests with MockMvc
- Deploy Spring Boot application to Azure App Service
- Connect deployed app to Azure SQL

Topics (Testing — 4 hrs)

- JUnit 5: `@Test`, `@BeforeEach`, `@AfterEach`, `@ParameterizedTest`, Assertions
- Mockito: `@Mock`, `@InjectMocks`, `@ExtendWith(MockitoExtension.class)`, `when/thenReturn`, `verify()`
- `@SpringBootTest`, `@WebMvcTest`, `MockMvc`, `@MockBean`
- `@DataJpaTest` with H2 in-memory DB for repository tests
- Test naming convention: `methodName_scenario_expectedBehavior`

Topics (Azure — 4 hrs)

- Azure portal overview: resource groups, subscriptions
- Azure App Service: plans (Free/B1/S1), create web app for Java
- Azure SQL Database: create server + database, firewall rules
- Deploy JAR: GitHub Actions `azure/webapps-deploy` action
- `application-prod.properties` with environment variable placeholders
- App Service → Configuration → Application Settings (environment variables)
- Smoke test deployed API with Postman

Lab Part A: Testing

- Unit test `AccountService.transfer()` — mock `AccountRepository`, verify calls
- Unit test `CustomerService` — test all CRUD methods with Mockito
- Integration test `CustomerController` using `MockMvc`:
  - Test `GET /api/customers` returns 200
  - Test `POST /api/customers` with invalid body returns 400

Lab Part B: Azure Deployment

- Create Azure Resource Group: `rg-banking-dev`
- Create Azure SQL Database, update schema
- Create Azure App Service (Java 17)
- Add Azure deploy step to GitHub Actions workflow
- Store connection string in App Service Application Settings
- Verify: hit live API URL, get customer list from Azure SQL

Deliverable: Live backend running on Azure, accessible via public URL

WEEK 3 — Angular Frontend + Cloud Integration

Day 11 — Angular Core: Components, Binding, Directives, Pipes (8 hrs)

Learning Objectives

- Set up Angular project and understand architecture
- Build components with templates and data binding
- Use built-in directives for conditional/list rendering
- Format data with pipes

Topics

- Node.js, npm, Angular CLI: `ng new`, `ng serve`, `ng generate component`
- Angular architecture: modules (or standalone components), components, templates, metadata
- Component lifecycle: `ngOnInit`, `ngOnChanges`, `ngOnDestroy`
- Data binding:
  - Interpolation: `{{ expression }}`
  - Property binding: `[property]="value"`
  - Event binding: `(click)="handler()"`
  - Two-way binding: `[(ngModel)]="field"`
- Built-in directives: `*ngIf`, `*ngFor`, `ngClass`, `ngStyle`

- Built-in pipes: `date`, `currency`, `uppercase`, `lowercase`, `number`, `async`
- Angular Material or Bootstrap setup for styling

Lab: Account Dashboard UI

```
ng new banking-portal --routing --style=scss
ng generate component features/accounts/account-list
ng generate component features/accounts/account-card
ng generate component shared/header
```

- Create mock `accounts` array in component with 5 sample accounts
- Use `*ngFor` to render `account-card` for each account
- Use `*ngIf` to show/hide balance based on `showBalance` toggle
- Apply `currency:'USD'` pipe to balance display
- Apply `date:'mediumDate'` pipe to account creation date
- Use `ngClass` to add CSS class based on account type (savings/current)
- Click on a card to select it and show details below the list

Deliverable: Running Angular app displaying mock account data with formatting

---

Day 12 — Angular Services: HTTP Client, RxJS, Routing, Guards (8 hrs)

Learning Objectives

- Create injectable services for data access
- Call REST APIs using HttpClient
- Navigate between views with Angular Router
- Protect routes with Auth Guards

Topics

- `@Injectable({ providedIn: 'root' })` services
- `HttpClientModule`, `HttpClient`: `get<T>()`, `post<T>()`, `put<T>()`, `delete<T>()`
- RxJS `Observable`: subscribe, unsubscribe, `async` pipe

- RxJS operators: `map()`, `catchError()`, `tap()`, `switchMap()`, `forkJoin()`, `of()`
- `BehaviorSubject` for state sharing between components
- Angular Router: `RouterModule.forRoot(routes)`, `<router-outlet>`, `routerLink`
- Route parameters: `ActivatedRoute.snapshot.params`
- Route Guards: `CanActivate` interface
- `environment.ts`: `apiBaseUrl: 'http://localhost:8080'`

Lab: Connected Services + Routing

```
ng generate service core/services/account
ng generate service core/services/customer
ng generate service core/services/auth
ng generate guard core/guards/auth
```

- `AccountService`: `getAll()`, `getById(id)`, `create(account)`, `update(id, account)`, `delete(id)`
- `CustomerService`: similar CRUD methods
- Set up routes: `/login`, `/dashboard`, `/customers`, `/accounts`, `/accounts/:id`
- `AuthGuard`: check `localStorage.getItem('token')`, redirect to `/login` if missing
- `AccountDetailComponent`: use `ActivatedRoute` to get ID, call `getById()`, display details
- Handle loading state and error state in template

Deliverable: Multi-page Angular app with routing and real HTTP calls (to mock or live API)

Day 13 — Angular + Spring Boot Integration: CORS, Forms, Auth Flow (8 hrs)

Learning Objectives

- Connect Angular frontend to Spring Boot backend end-to-end
- Build reactive forms with client-side validation
- Implement full JWT login/logout flow with HTTP interceptor

Topics

- CORS: configure `@CrossOrigin` or global CORS in Spring `SecurityFilterChain`
- Reactive Forms: `FormGroup`, `FormControl`, `FormBuilder`, `Validators`
- Built-in validators: `required`, `email`, `minLength`, `maxLength`, `pattern`
- Custom validators
- Form submission, form state: `valid`, `invalid`, `dirty`, `touched`
- Displaying validation errors in template
- HTTP Interceptors: `HttpInterceptor`, attaching `Authorization: Bearer <token>` header
- `AuthService`: `login()`, `logout()`, `getToken()`, `isAuthenticated()`
- Token storage and retrieval from `localStorage`
- Redirect after login, redirect after logout

Lab: End-to-End Integration

Spring Boot side:

```
// In SecurityFilterChain:
.cors(cors -> cors.configurationSource(corsConfigurationSource()))
// Allow: http://localhost:4200
```

Angular side:

```
ng generate interceptor core/interceptors/auth
ng generate component features/auth/login
ng generate component features/accounts/create-account
```

- Build `LoginComponent` with reactive form: email + password, validation messages
- On submit: call `AuthService.login()` → store token → navigate to `/dashboard`
- Create `AuthInterceptor`: attach JWT to every outgoing request
- Build `CreateAccountComponent` with reactive form: account type, initial balance
- On submit: call `AccountService.create()` → navigate to account list
- Test full flow: register user → login → create account → view in list → logout

Deliverable: Fully connected full-stack application running locally

---

Day 14 — Azure: Static Web Apps, CI/CD Pipeline, End-to-End Deploy (8 hrs)

Learning Objectives

- Host Angular frontend on Azure Static Web Apps
- Build a complete GitHub Actions CI/CD pipeline for both frontend and backend
- Manage configuration secrets securely

Topics

- Azure Static Web Apps: create resource, link to GitHub repo, workflow file
- Static Web Apps routing: `staticwebapp.config.json` for SPA fallback
- Angular production build: `ng build --configuration production`
- GitHub Actions — complete pipeline:
  - Trigger: `push` to `main`
  - Job 1 (Backend): checkout → JDK 17 → `mvn test` → `mvn package` → deploy to App Service
  - Job 2 (Frontend): checkout → Node 18 → `npm ci` → `ng build` → deploy to Static Web Apps
- GitHub Secrets: `AZURE_WEBAPP_PUBLISH_PROFILE`, `AZURE_STATIC_WEB_APPS_API_TOKEN`
- Environment variable substitution: `environment.prod.ts` with `apiBaseUrl`
- Azure App Service: HTTPS, custom domain basics
- Azure Application Insights: add to Spring Boot (starter), monitor live requests

Lab: Full CI/CD Pipeline

```
# .github/workflows/deploy.yml
name: Deploy Banking Portal

on:
  push:
    branches: [main]

jobs:
  deploy-backend:
    runs-on: ubuntu-latest
```

```
        steps:
          - uses: actions/checkout@v4
          - uses: actions/setup-java@v4
            with: { java-version: '17', distribution: 'temurin' }
          - run: mvn clean package -DskipTests
          - uses: azure/webapps-deploy@v3
            with:
              app-name: ${{ secrets.AZURE_APP_NAME }}
              publish-profile: ${{ secrets.AZURE_WEBAPP_PUBLISH_PROFILE
}}
              package: target/*.jar

      deploy-frontend:
        runs-on: ubuntu-latest
        steps:
          - uses: actions/checkout@v4
          - uses: actions/setup-node@v4
            with: { node-version: '18' }
          - run: npm ci && ng build --configuration production
            working-directory: banking-portal
          - uses: Azure/static-web-apps-deploy@v1
            with:
              azure_static_web_apps_api_token: ${{
secrets.AZURE_STATIC_WEB_APPS_API_TOKEN }}
              action: upload
              app_location: banking-portal
              output_location: dist/banking-portal
```

- Add all secrets to GitHub repository settings
- Push to `main`, watch both jobs run in GitHub Actions
- Update `environment.prod.ts` to point to live Azure App Service URL
- Verify end-to-end: open Static Web App URL → login → create/view accounts

Deliverable: Fully deployed, publicly accessible banking portal on Azure

---

Day 15 — Capstone: Integration, Code Review, Presentations (8 hrs)

Learning Objectives

- Demonstrate a working full-stack application deployed to Azure
- Conduct structured code review
- Reflect on architecture decisions and lessons learned

Morning (4 hrs) — Integration Sprint

- Bug fixing, UI polish, missing features
- Final PR: merge all feature branches to `main`
- Verify CI/CD pipeline runs clean
- Final smoke test on live Azure URLs

Afternoon (4 hrs) — Presentations + Review

Each team/individual presents (15 min):

1. Live demo of the deployed application
2. Architecture walkthrough: components, data flow, security
3. One technical challenge encountered and how it was solved
4. What they would improve with more time

Code Review Session (group)

- Review one participant's `AccountService` + `AccountController` as a group
- Discuss: naming, exception handling, transaction boundaries, test coverage
- Trainer provides written feedback per participant

Retrospective

- What concepts need reinforcement
- Recommended next steps per track (backend, frontend, cloud)

Deliverable: Deployed capstone project + architecture diagram + presentation

---

Summary: Technology Coverage

Technology	Days	Key Outcomes
Java	Days 1–3	OOP, Collections, Streams, Lambdas
MS SQL	Day 4	Schema design, complex queries, stored procs
GitHub	Day 5 + every day	Git flow, PRs, CI with GitHub Actions

Technology	Days	Key Outcomes
Spring Boot	Days 6–10	Full REST API, JPA, Security, Testing
Angular	Days 11–13	Components, Services, Routing, Forms, Auth
Azure	Days 10 + 14	App Service, Azure SQL, Static Web Apps, CI/CD

Tools Required

Tool	Purpose
IntelliJ IDEA Community / VS Code	Java + Spring Boot development
VS Code	Angular development
SQL Server Management Studio (SSMS)	MS SQL
Postman	API testing
Node.js 18+ + Angular CLI	Frontend
JDK 17 (Temurin/Eclipse)	Java runtime
Maven 3.9+	Build tool
Azure Account (free tier)	Cloud deployment
GitHub Account	Version control + CI/CD