1. Spring Boot Overview

Spring Boot simplifies Spring application development by providing production-ready defaults for application setup. It features:

- Embedded servers (Tomcat, Jetty)
- Auto-configuration
- Starter dependencies (like `spring-boot-starter-web`, `spring-boot-starter-data-jpa`)
- Spring Boot Actuator for monitoring
- Simplified project structure

Spring Boot promotes layered architecture and reduces boilerplate using powerful annotations and conventions.

---

2. Spring Bean Lifecycle

Spring manages beans in several lifecycle phases:

1. **Instantiation**
2. **Dependency Injection** (`@Autowired`, `@Value`)
3. **Aware Interfaces**: `BeanNameAware`, `ApplicationContextAware`
4. **BeanPostProcessor** – before init
5. **Initialization**:
    - `@PostConstruct`
    - `InitializingBean.afterPropertiesSet()`
    - `init-method`
6. **BeanPostProcessor** – after init
7. **Bean ready for use**
8. **Destruction**:
    - `@PreDestroy`
    - `DisposableBean.destroy()`

Best practice: Use `@PostConstruct` and `@PreDestroy` for initialization and cleanup.

---

3. Core Spring Boot Annotations

Annotation	Purpose
`@SpringBootApplication`	Combines `@Configuration`, `@ComponentScan`, `@EnableAutoConfiguration`
`@Component`, `@Service`, `@Repository`, `@Controller`, `@RestController`	Stereotype annotations
`@Autowired`	Injects bean dependencies
`@Value`	Injects values from `application.properties`

Annotation	Purpose
`@Qualifier`	Disambiguates autowired beans
`@PostConstruct`, `@PreDestroy`	Lifecycle hooks
`@Configuration`, `@Bean`	Manual bean registration
`@Scope("prototype")`	Bean scope definition

4. REST API Example

```
@RestController
@RequestMapping("/api")
public class MyController {

    @GetMapping("/hello")
    public String hello() {
        return "Hello World!";
    }
}
```

---

5. Spring Boot Layered Architecture

Modular design promotes separation of concerns:

- **Controller Layer**: Handles HTTP requests. Uses `@RestController`
- **Service Layer**: Business logic. Annotated with `@Service`
- **Repository Layer**: Database access using Spring Data. Uses `@Repository`
- **Entity Layer**: Represents DB tables using JPA `@Entity`
- **Security Layer**: Auth and authorization using Spring Security

---

6. Spring Data JPA & Hibernate

Spring Boot simplifies ORM with Spring Data JPA (built on Hibernate).

Entity Example:

```
@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
}
```

Repository Example:

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {}
```

---

7. Transaction Management

Use `@Transactional` for defining transactional methods.

```
@Service
public class UserService {
    @Autowired
    private UserRepository userRepository;

    @Transactional
    public void createUser(User user) {
        userRepository.save(user);
    }
}
```

Properties of `@Transactional`:

- `rollbackFor`, `readOnly`, `isolation`, `propagation`

---

8. Spring Security Overview

Spring Security secures REST APIs with filters and configuration.

Key Annotations

- `@EnableWebSecurity` — Enables Spring Security config
- `@Configuration` — Declares configuration class
- `@Bean` — Registers `SecurityFilterChain`
- `@PreAuthorize` — Method-level security

Example:

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http.csrf().disable()
            .authorizeHttpRequests()
```

```
            .requestMatchers("/api/public").permitAll()
            .anyRequest().authenticated()
            .and().httpBasic();
        return http.build();
    }
}
```

---

9. JWT Authentication in Spring Boot

Flow:

1. User logs in with credentials
2. Server verifies and creates JWT
3. JWT is returned to client
4. Client sends JWT in `Authorization` header
5. Server verifies JWT in a filter

Token Generation:

```
public String generateToken(UserDetails userDetails) {
    return Jwts.builder()
        .setSubject(userDetails.getUsername())
        .setIssuedAt(new Date(System.currentTimeMillis()))
        .setExpiration(new Date(System.currentTimeMillis() + 1000 * 60 * 60 * 10))
        .signWith(SignatureAlgorithm.HS256, SECRET_KEY)
        .compact();
}
```

JWT Filter:

```
public class JwtFilter extends OncePerRequestFilter {

    @Autowired
    private JwtService jwtService;

    @Override
    protected void doFilterInternal(HttpServletRequest request,
                                    HttpServletResponse response,
                                    FilterChain filterChain) throws
ServletException, IOException {
        final String authHeader = request.getHeader("Authorization");

        if (authHeader != null && authHeader.startsWith("Bearer ")) {
            String jwt = authHeader.substring(7);
            String username = jwtService.extractUsername(jwt);
            // Validate token and authenticate
        }
```

```
        filterChain.doFilter(request, response);
    }
}
```

---

10. Real-World Project Structure

```
src/main/java/com/example/app/
│
├── config/            → Configs (security, CORS, Swagger)
├── controller/        → REST Controllers
├── service/           → Interfaces & implementations
├── repository/        → Spring Data Repositories
├── model/             → Entity + DTOs
├── security/          → JWT filters, config, details service
├── exception/         → Global exception handling
├── util/              → Utility classes
└── AppApplication.java → Main Spring Boot class

src/main/resources/
├── application.yml / .properties
├── static/
└── templates/         → (Thymeleaf if needed)

test/java/com/example/app/
├── unit/              → JUnit unit tests
└── integration/       → End-to-end test classes
```

Best Practices

- Separate DTOs from Entities
- Use `@ControllerAdvice` for global error handling
- Use `ModelMapper` or `MapStruct` for conversions
- Write interface-first service layers
- Use multiple config classes: `SecurityConfig`, `SwaggerConfig`, etc.
- Split features into modules for large apps