**Spring Java**

**Spring Learning Plan (Deep Understanding)**

**Phase 1 — Fundamentals (Core Spring)**

**Goal: Understand what Spring is solving and how it solves it internally.**

* **What is Spring? Why was it created?**
* **Core Concepts:**
  + **Inversion of Control (IoC)**
  + **Dependency Injection (DI)**
  + **Beans and Bean Lifecycle**
  + **ApplicationContext vs BeanFactory**
  + **Scope of beans (singleton, prototype, etc.)**
* **How Spring internally creates and manages objects (with examples)**
* **XML Configuration vs Java-based Configuration**
* **Annotations: @Component, @Service, @Repository, @Controller**
* **Autowiring: @Autowired (byType, byName, Constructor Injection)**
* **Profiles and Environment Management**

**Phase 2 — Spring AOP (Aspect-Oriented Programming)**

**Goal: Understand cross-cutting concerns like logging, security.**

* **What is AOP? Why is it needed?**
* **JoinPoint, Advice, Pointcut, Aspect**
* **Implementing AOP using Spring’s @Aspect**
* **Internal working: How proxies (JDK Dynamic Proxy / CGLIB) work inside Spring**

**Phase 3 — Spring Data Access (JDBC + ORM)**

**Goal: How Spring interacts with databases internally.**

* **Spring JDBC Template**
* **DAO layer creation**
* **ORM with Hibernate (using Spring ORM)**
* **Transactions: @Transactional and internal mechanism (ACID properties)**

**Phase 4 — Spring MVC (Web Development)**

**Goal: Build full-stack Java backend.**

* **DispatcherServlet working internally**
* **Front Controller Design Pattern**
* **Handler Mapping, View Resolver, Controllers**
* **Form Handling, Validation**
* **Exception Handling in Spring MVC**

**Phase 5 — Spring Security (Basic Introduction)**

**Goal: Protect your application**

* **Authentication vs Authorization**
* **Basic Authentication Setup**
* **How Spring Security filters work internally**

**What is Spring? Why was it created?**

**Problem Before Spring:**

**Before Spring (in early J2EE/Java EE days), developers had to:**

* **Manually write complex code to connect components (tight coupling).**
* **Use heavyweight components like EJBs (Enterprise JavaBeans) which were hard to test, configure, and maintain.**
* **Struggle with boilerplate code and dependency management.**

**Why Spring?**

**Spring was created to:**

* **Simplify Java development.**
* **Promote loose coupling between components.**
* **Replace heavy EJBs with lightweight POJOs (Plain Old Java Objects).**
* **Support easy testability, modularity, and scalability.**

**Spring is a lightweight framework that provides infrastructure support for developing Java applications with a focus on Inversion of Control, Dependency Injection, and modular design.**

**Core Concepts**

**1. Inversion of Control (IoC)**

**Definition: IoC means the control of creating and managing objects is inverted from the programmer to the framework.**

**Without IoC:**

**Car car = new Car(); // Developer controls creation**

**With IoC:**

**Car car = context.getBean(Car.class); // Spring creates & manages the object**

**Diagram (IoC Container):**

**+---------------------+**

**| Spring Container |**

**| (IoC Container) |**

**+---------------------+**

**|**

**v**

**+---------------------+**

**| Creates & Injects |**

**| dependencies (Beans)|**

**+---------------------+**

**2. Dependency Injection (DI)**

**Definition: DI is a design pattern in which dependencies are injected by the container, rather than the object creating them itself.**

**Types of DI:**

* **Constructor Injection**
* **Setter Injection**
* **Field Injection (not recommended in real apps)**

**Example:**

**class Engine {}**

**class Car {**

**private Engine engine;**

**// Constructor Injection**

**public Car(Engine engine) {**

**this.engine = engine;**

**}**

**}**

**Spring injects the Engine into Car.**

**3. Beans and Bean Lifecycle**

**Bean: A Java object managed by Spring.**

**Bean Lifecycle Steps:**

1. **Instantiate (via constructor)**
2. **Populate properties**
3. **Call setBeanName(), setBeanFactory(), etc. (aware interfaces)**
4. **@PostConstruct / afterPropertiesSet()**
5. **Bean is ready to use**
6. **@PreDestroy / destroy() when context closes**

**Diagram: Bean Lifecycle**

**Constructor --> Set Properties --> Custom Init --> Use --> Custom Destroy**

**4. ApplicationContext vs BeanFactory**

| **Feature** | **BeanFactory** | **ApplicationContext** |
| --- | --- | --- |
| **Lazy Initialization** | **Yes** | **No (eager by default)** |
| **Advanced Features** | **No** | **Yes (AOP, Events, I18n, etc.)** |
| **Usage** | **Lightweight apps** | **Recommended for real-world apps** |

**ApplicationContext is a superset of BeanFactory.**

**5. Scope of Beans**

**Spring allows different scopes for managing bean lifecycles:**

| **Scope** | **Description** |
| --- | --- |
| **singleton** | **Default. One instance per Spring container** |
| **prototype** | **New instance every time it's requested** |
| **request** | **One per HTTP request (Web apps only)** |
| **session** | **One per HTTP session (Web apps only)** |
| **application** | **One per ServletContext** |

**Example:**

**<bean id="myBean" class="com.MyClass" scope="prototype"/>**

**Internal Flow Summary (Diagram)**

**+---------------------------+**

**| ApplicationContext |**

**| (Spring IoC Container) |**

**+------------+--------------+**

**|**

**v**

**Loads bean definitions from XML/@Config**

**|**

**v**

**Creates Beans using DI (Constructor/Setter)**

**|**

**v**

**Initializes Beans (PostConstruct, etc.)**

**v**

**Ready to Use!**

**How Spring Internally Creates and Manages Objects (Beans)**

**Core Principle: IoC + DI**

**Spring uses an IoC Container (like ApplicationContext) to:**

1. **Read configuration (XML/Java-based/annotations)**
2. **Create objects (beans)**
3. **Inject dependencies into those beans**
4. **Manage lifecycle (init, use, destroy)**

**Internal Process:**

**1. Scan Configuration (XML/@ComponentScan)**

**2. Load Bean Definitions**

**3. Instantiate Bean**

**4. Inject Dependencies (Constructor/Setter/Field)**

**5. Post-process (init-method, @PostConstruct)**

**6. Use Bean**

**7. Destroy on context close (if configured)**

**Example:**

**<!-- XML Config -->**

**<bean id="engine" class="com.Engine" />**

**<bean id="car" class="com.Car">**

**<constructor-arg ref="engine" />**

**</bean>**

**public class Car {**

**private Engine engine;**

**public Car(Engine engine) {**

**this.engine = engine;**

**}**

**}**

**Spring handles:**

* **Object creation (new Car(engine))**
* **Wiring (engine is injected into car)**

**XML Configuration vs Java-based Configuration**

| **Feature** | **XML Configuration** | **Java-based Configuration** |
| --- | --- | --- |
| **Style** | **Declarative (XML file)** | **Programmatic (Java class with @Configuration)** |
| **Flexibility** | **Less flexible, static** | **Type-safe, easier to refactor** |
| **Readability** | **Separate from code** | **Integrated with code** |
| **Annotations Required?** | **No** | **Yes (@Configuration, @Bean)** |

**XML Example:**

**<bean id="engine" class="com.Engine" />**

**Java-based Example:**

**@Configuration**

**public class AppConfig {**

**@Bean**

**public Engine engine() {**

**return new Engine();**

**}**

**}**

**Both approaches tell Spring: create and manage Engine bean.**

**Annotations: @Component, @Service, @Repository, @Controller**

**These are stereotype annotations — they tell Spring to auto-detect and register classes as beans.**

**How It Works Internally:**

1. **Spring scans the package (via @ComponentScan)**
2. **Finds classes with annotations like @Component**
3. **Registers them as beans in the container**

**Differences:**

| **Annotation** | **Purpose** | **Layer** |
| --- | --- | --- |
| **@Component** | **Generic bean** | **Any** |
| **@Service** | **Marks business logic class** | **Service Layer** |
| **@Repository** | **Marks DAO/data access class** | **Persistence Layer** |
| **@Controller** | **Marks web controller** | **Presentation Layer** |

**All are detected by @ComponentScan, but used for clarity and tool support (like exception translation in @Repository).**

**Autowiring: @Autowired**

**Spring automatically injects dependencies without needing <constructor-arg> or <property>.**

**🔧 Modes of Injection:**

1. **Constructor Injection (Recommended)**

**@Component**

**public class Car {**

**private Engine engine;**

**@Autowired**

**public Car(Engine engine) {**

**this.engine = engine;**

**}**

**}**

1. **Setter Injection**

**@Component**

**public class Car {**

**private Engine engine;**

**@Autowired**

**public void setEngine(Engine engine) {**

**this.engine = engine;**

**}**

**}**

1. **Field Injection ❌ (easy but bad for testing)**

**@Autowired**

**private Engine engine;**

**How It Works Internally:**

* **Spring uses reflection to find @Autowired annotations.**
* **Resolves the dependency by type (default).**
* **Throws error if multiple beans match → use @Qualifier.**

**Profiles and Environment Management**

**Spring can load different configurations for different environments (dev, test, prod).**

**Why Profiles?**

* **Use one bean config in dev, another in prod.**
* **Load different DB settings, services, etc.**

**@Profile Example:**

**@Configuration**

**public class AppConfig {**

**@Bean**

**@Profile("dev")**

**public DataSource devDataSource() {**

**return new H2DataSource();**

**}**

**@Bean**

**@Profile("prod")**

**public DataSource prodDataSource() {**

**return new OracleDataSource();**

**}**

**}**

**Internally:**

* **Spring checks the active profile via environment variable or config file.**
* **Registers only beans matching the active profile.**

**Activate Profile:**

**In application.properties:**

**spring.profiles.active=dev**

**Or in XML:**

**<context:property-placeholder location="classpath:application.properties"/>**

**Summary Diagram**

**+--------------------+**

**| Spring Container |**

**+--------------------+**

**|**

**+----------------------------------------------+**

**| Reads @Configuration, @ComponentScan, XML |**

**| Creates Beans |**

**| Injects Dependencies via @Autowired |**

**| Manages Lifecycle |**

**| Activates @Profile-based Beans |**

**+----------------------------------------------+**

**Phase 2 — Spring AOP (Aspect-Oriented Programming)**

**What is AOP? Why is it needed?**

**AOP Meaning:**

* **Aspect-Oriented Programming is a programming technique where you separate cross-cutting concerns (things that affect multiple parts of the app) into separate modules called Aspects.**
* **Cross-cutting concerns: Logging, Security, Transactions, Caching, etc.**

| **OOP (Object-Oriented) Focus** | **AOP (Aspect-Oriented) Focus** |
| --- | --- |
| **Focus on what the class does** | **Focus on what cuts across classes** |

**Why AOP?**

**Without AOP:**

* **Logging, security, transaction code mixed with business logic.**
* **Code becomes repetitive, messy, hard to maintain.**

**With AOP:**

* **You separate cross-cutting logic into separate reusable Aspects.**
* **Cleaner and modular code.**

**Core AOP Concepts:**

| **Term** | **Meaning** | **Example** |
| --- | --- | --- |
| **JoinPoint** | **A point in the application (like method call) where you can apply extra behavior** | **Method execution** |
| **Advice** | **The action taken at a JoinPoint (code you want to run)** | **Logging, Security check** |
| **Pointcut** | **Condition to select which JoinPoints to apply the Advice** | **All methods of service layer** |
| **Aspect** | **A class that groups multiple Advices** | **LoggingAspect, SecurityAspect** |

**Simple Diagram:**

**Application Code -----> [JoinPoints (Methods)]**

**|**

**(Pointcut matches?)**

**|**

**[Apply Advice (Before/After/Around)]**

**Implementing AOP using Spring’s @Aspect**

1. **Create a normal class.**
2. **Annotate it with @Aspect.**
3. **Write @Before, @After, @Around advices.**

**Example: LoggingAspect**

**@Aspect**

**@Component**

**public class LoggingAspect {**

**// Advice: Run before every method inside package com.example.service**

**@Before("execution(\* com.example.service.\*.\*(..))")**

**public void logBeforeMethod(JoinPoint joinPoint) {**

**System.out.println("Before Method: " + joinPoint.getSignature().getName());**

**}**

**}**

**Explanation:**

* **@Aspect: Tells Spring that this is an Aspect.**
* **@Before: Advice that runs before methods matching the Pointcut.**
* **execution(\* com.example.service.\*.\*(..)): Pointcut expression meaning → any method inside com.example.service package.**

**Internal Working — How Proxies Work in Spring AOP**

**Spring AOP is proxy-based:**

* **It creates proxy objects of your beans.**
* **When you call a method, it actually calls the proxy.**
* **Proxy adds extra behavior (like logging) before/after calling your original method.**

**Proxy Creation Flow:**

**Client (your code)**

**|**

**v**

**[Proxy Object (created by Spring AOP)]**

**|**

**v**

**[Your Target Business Object (actual bean)]**

**Proxy adds:**

* **Logging**
* **Security checks**
* **Transactions before/after calling real method.**

**Types of Proxy:**

| **Proxy Type** | **When Used** | **How Works** |
| --- | --- | --- |
| **JDK Dynamic Proxy** | **If the bean implements an Interface** | **Proxy object implements the same Interface** |
| **CGLIB Proxy** | **If the bean is a class (no interface)** | **Proxy subclasses the target class** |

**Internal Working:**

1. **At startup:  
   Spring scans for @Aspect classes.**
2. **At bean creation:  
   If a bean matches any Pointcut:**
   * **Spring wraps it in a Proxy instead of returning the real bean.**
3. **When method called:**
   * **Proxy intercepts the call.**
   * **Applies Advices (@Before, @After etc.)**
   * **Then calls real method.**

**Flow Example:**

**You call car.drive()**

**Actually Spring calls proxy.drive()**

**Before Advice: "Logging before drive"**

**Real car.drive() runs**

**After Advice: "Logging after drive"**

**Full AOP Working Diagram:**

**+----------------+**

**| Your Code |**

**| (calls method) |**

**+--------+--------+**

**|**

**v**

**+---------+---------+**

**| Spring Proxy |**

**| (generated bean) |**

**+---------+----------+**

**|**

**+-----------------+------+----------------+**

**| |**

**+-------v--------+ +----------v--------+**

**| Apply Advices | | Call Real Method |**

**| (before/after) | | (Business logic) |**

**+----------------+ +-------------------+**

**🚀 Quick Recap:**

| **Concept** | **Key** |
| --- | --- |
| **AOP** | **Separate cross-cutting concerns** |
| **JoinPoint** | **Where extra code can run (methods)** |
| **Advice** | **What extra code you run** |
| **Pointcut** | **Where you apply the advice (method matcher)** |
| **Aspect** | **Class holding the advices** |
| **Proxy** | **Middle layer to apply extra behavior** |

**Great! You're now entering Phase 3 — Spring Data Access, which is all about how Spring communicates with the database using JDBC, Hibernate (ORM), and manages transactions.  
Let’s break it down theoretically, internally, diagrammatically, and with flow.**

**📘 Phase 3 — Spring Data Access (JDBC + ORM)**

**1. Spring JDBC Template**

**🔹 What is it?**

**Spring provides JdbcTemplate as a helper class to simplify working with JDBC (Java Database Connectivity).  
It hides the boilerplate code like:**

* **Opening a connection**
* **Creating a statement**
* **Handling exceptions**
* **Closing resources**

**Internal Working of JdbcTemplate**

**jdbcTemplate.query("SELECT \* FROM users", rowMapper);**

**🔄 Workflow Diagram**

**Client Code (Your DAO)**

**|**

**v**

**JdbcTemplate.query()**

**|**

**v**

**Opens Connection from DataSource**

**|**

**v**

**Prepares Statement**

**|**

**v**

**Executes Query**

**|**

**v**

**Maps ResultSet using RowMapper**

**|**

**v**

**Returns List/Result**

**Example:**

**public class UserDao {**

**@Autowired**

**private JdbcTemplate jdbcTemplate;**

**public List<User> getAllUsers() {**

**return jdbcTemplate.query("SELECT \* FROM users", new BeanPropertyRowMapper<>(User.class));**

**}**

**}**

**2. DAO Layer (Data Access Object)**

* **DAO Layer contains logic to access the database (queries, inserts, updates).**
* **Keeps business logic and DB logic separated.**

**🎯 Structure:**

**Controller ➝ Service ➝ DAO ➝ JdbcTemplate / Hibernate**

**✍ Example DAO Layer:**

**@Repository**

**public class UserDao {**

**@Autowired**

**private JdbcTemplate jdbcTemplate;**

**public int addUser(User user) {**

**String sql = "INSERT INTO users(name, email) VALUES (?, ?)";**

**return jdbcTemplate.update(sql, user.getName(), user.getEmail());**

**}**

**}**

**3. ORM with Hibernate (using Spring ORM)**

**🔹 What is ORM?**

**ORM (Object-Relational Mapping) maps Java objects to database tables.**

**Spring integrates with Hibernate using:**

* **LocalSessionFactoryBean**
* **HibernateTransactionManager**

**📈 Hibernate Internal Workflow**

**Client (DAO)**

**|**

**v**

**Hibernate SessionFactory (Spring-managed)**

**|**

**v**

**Hibernate Session**

**|**

**v**

**Translates Objects ↔ SQL**

**|**

**v**

**Database**

**💡 Spring + Hibernate Example**

**🔧 Hibernate Configuration (XML or Java config)**

**<bean id="sessionFactory" class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">**

**<property name="dataSource" ref="dataSource"/>**

**<property name="packagesToScan" value="com.example.model"/>**

**<property name="hibernateProperties">**

**<props>**

**<prop key="hibernate.dialect">org.hibernate.dialect.PostgreSQLDialect</prop>**

**<prop key="hibernate.hbm2ddl.auto">update</prop>**

**</props>**

**</property>**

**</bean>**

**🧾 DAO with Hibernate**

**@Repository**

**public class UserDao {**

**@Autowired**

**private SessionFactory sessionFactory;**

**public void saveUser(User user) {**

**Session session = sessionFactory.getCurrentSession();**

**session.save(user);**

**}**

**}**

**4. Transactions: @Transactional and Internal Mechanism**

**🔹 What is a Transaction?**

**A transaction is a sequence of operations performed as a single unit of work, maintaining ACID properties.**

| **Property** | **Meaning** |
| --- | --- |
| **A** | **Atomicity: all or nothing** |
| **C** | **Consistency: DB remains valid** |
| **I** | **Isolation: parallel safety** |
| **D** | **Durability: permanent changes** |

**🔹 @Transactional Annotation**

**Spring provides @Transactional to manage transactions declaratively.**

**@Service**

**@Transactional**

**public class UserService {**

**@Autowired**

**private UserDao userDao;**

**public void registerUser(User user) {**

**userDao.save(user);**

**// if exception → rollback happens automatically**

**}**

**}**

**🧠 Internal Working of @Transactional**

1. **Spring creates a proxy for the class.**
2. **When method is called:**
   * **Transaction Manager starts a transaction.**
   * **Method executes.**
   * **If no error, it commits.**
   * **If exception, it rolls back.**

**🔁 Transaction Flow**

**Service method called ➝ Is @Transactional?**

**|**

**|-- Yes**

**v**

**TransactionManager starts TX**

**|**

**v**

**DAO call ➝ DB interaction**

**|**

**v**

**If success → Commit**

**If exception → Rollback**

**⚙️ Diagram of Spring ORM + Transaction Flow:**

**[Controller]**

**↓**

**[Service Layer]**

**@Transactional**

**↓**

**[Transaction Proxy]**

**↓**

**[Hibernate SessionFactory]**

**↓**

**[Hibernate Session]**

**↓**

**[Database]**

**Summary Table:**

| **Concept** | **Summary** |
| --- | --- |
| **JdbcTemplate** | **Simplifies JDBC operations** |
| **DAO Layer** | **Manages DB logic and queries** |
| **ORM (Hibernate)** | **Maps objects to database tables** |
| **@Transactional** | **Automatically manages transactions** |
| **Internals** | **Spring uses proxies and Hibernate SessionFactory internally** |

**Phase 4 — Spring MVC (Web Development)**

**What is Spring MVC?**

**Spring MVC is a part of the Spring Framework that helps in creating web applications using the Model-View-Controller design pattern.  
It separates the application logic into 3 interconnected components:**

* **Model — Business logic / data**
* **View — User interface (JSP, Thymeleaf)**
* **Controller — Handles HTTP requests and responses**

**Internal Working: DispatcherServlet**

**🔹 What is DispatcherServlet?**

* **It's the front controller in Spring MVC.**
* **All incoming web requests go through DispatcherServlet.**
* **It delegates work to Controllers, ViewResolvers, and HandlerMappings.**

**Flow Diagram: Spring MVC Request Life Cycle**

**Browser (Client)**

**↓ HTTP Request**

**[ DispatcherServlet ] <----> (Web.xml or Java Config)**

**↓**

**[ HandlerMapping ] → Finds correct Controller**

**↓**

**[ Controller ] → Executes logic**

**↓**

**[ Model and View ]**

**↓**

**[ ViewResolver ] → Resolves actual JSP/HTML/Thymeleaf view**

**↓**

**[ DispatcherServlet ]**

**↓**

**HTTP Response → Sent back to browser**

**1. Front Controller Design Pattern**

**Spring uses the Front Controller Design Pattern, where a single servlet handles all incoming requests.**

**<!-- web.xml -->**

**<servlet>**

**<servlet-name>spring</servlet-name>**

**<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>**

**<load-on-startup>1</load-on-startup>**

**</servlet>**

**<servlet-mapping>**

**<servlet-name>spring</servlet-name>**

**<url-pattern>/</url-pattern>**

**</servlet-mapping>**

**2. HandlerMapping, Controllers, ViewResolver**

**HandlerMapping**

* **Responsible for mapping HTTP requests to the correct Controller methods.**
* **Example: Maps /users to UserController.getUsers().**

**Controller**

**@Controller**

**public class UserController {**

**@GetMapping("/users")**

**public String getUsers(Model model) {**

**model.addAttribute("userList", userService.getAllUsers());**

**return "user-view"; // Logical View Name**

**}**

**}**

**ViewResolver**

* **Converts logical view names ("user-view") into actual views (user-view.jsp or HTML).**
* **Common ones:**
  + **InternalResourceViewResolver (for JSPs)**
  + **ThymeleafViewResolver (for Thymeleaf)**

**<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">**

**<property name="prefix" value="/WEB-INF/views/" />**

**<property name="suffix" value=".jsp" />**

**</bean>**

**🧾 3. Form Handling + Validation**

**Form Binding**

**<form action="/register" method="post">**

**<input name="name" />**

**<input name="email" />**

**<input type="submit" />**

**</form>**

**Controller to Handle Form**

**@PostMapping("/register")**

**public String registerUser(@ModelAttribute("user") User user) {**

**userService.save(user);**

**return "success";**

**}**

**Validation Using @Valid**

**@PostMapping("/register")**

**public String register(@Valid @ModelAttribute("user") User user, BindingResult result) {**

**if (result.hasErrors()) {**

**return "register-form";**

**}**

**userService.save(user);**

**return "success";**

**}**

**Use annotations like:**

**@NotNull, @Email, @Size(min = 2)**

**4. Exception Handling in Spring MVC**

**Global Exception Handler**

**Use @ControllerAdvice to handle errors globally:**

**@ControllerAdvice**

**public class GlobalExceptionHandler {**

**@ExceptionHandler(Exception.class)**

**public String handleError(Exception e, Model model) {**

**model.addAttribute("errorMessage", e.getMessage());**

**return "error-page";**

**}**

**}**

**Summary Diagram: Spring MVC Flow**

**Client Browser**

**↓**

**[ DispatcherServlet ] ← Front Controller**

**↓**

**[ HandlerMapping ] → Maps to...**

**↓**

**[ Controller ] → Returns ModelAndView**

**↓**

**[ ViewResolver ] → Maps to .jsp/.html**

**↓**

**[ Actual View (JSP/Thymeleaf) ]**

**↓**

**HTTP Response → Browser**

**Summary Table:**

| **Component** | **Role** |
| --- | --- |
| **DispatcherServlet** | **Front Controller** |
| **HandlerMapping** | **Maps URL to Controller** |
| **Controller** | **Handles logic** |
| **ViewResolver** | **Resolves logical view to JSP/HTML** |
| **Model** | **Holds data for the view** |
| **View** | **Displays output (e.g., JSP)** |
| **@ModelAttribute** | **Binds form data to POJOs** |
| **@Valid** | **Performs validation** |
| **@ExceptionHandler** | **Handles exceptions gracefully** |

**Phase 5 — Spring Security (Basic Introduction)**

**Goal:**

**To secure your Spring application by implementing authentication and authorization, and understanding the internal working of Spring Security filters.**

**Authentication vs Authorization**

| **Term** | **Meaning** |
| --- | --- |
| **Authentication** | **Verifying who the user is (username/password check)** |
| **Authorization** | **Determining what the user is allowed to access (roles, permissions)** |

**Example:**

* **Authentication: Logging in with username and password.**
* **Authorization: Checking if the logged-in user can access /admin/dashboard.**

**Spring Security Basic Flow**

**Client (Browser/Postman)**

**↓**

**[ Security Filter Chain ] ← intercepts every request**

**↓**

**[ AuthenticationManager ] ← validates user credentials**

**↓**

**[ UserDetailsService ] ← loads user details from DB/memory**

**↓**

**[ GrantedAuthorities ] ← sets user roles**

**↓**

**Access granted or denied**

**Components of Spring Security**

**1. Filter Chain (Security Filter Chain)**

* **A series of filters that inspect and process incoming requests.**
* **Most important filters:**
  + **UsernamePasswordAuthenticationFilter**
  + **BasicAuthenticationFilter**
  + **ExceptionTranslationFilter**
  + **FilterSecurityInterceptor**

**http**

**.authorizeRequests()**

**.anyRequest().authenticated()**

**.and()**

**.httpBasic(); // Enables Basic Auth**

**2. AuthenticationManager**

* **Core interface to authenticate users.**
* **Delegates to AuthenticationProvider.**

**3. UserDetailsService**

* **Used to load user-specific data.**

**@Service**

**public class MyUserDetailsService implements UserDetailsService {**

**public UserDetails loadUserByUsername(String username) {**

**return new User("admin", "{noop}password", List.of(new SimpleGrantedAuthority("ROLE\_USER")));**

**}**

**}**

**4. GrantedAuthority & Roles**

* **Roles or permissions assigned to users.**
* **Spring Security checks these during authorization.**

**Basic Authentication Setup (Memory-based)**

**1. Add Dependency**

**<dependency>**

**<groupId>org.springframework.boot</groupId>**

**<artifactId>spring-boot-starter-security</artifactId>**

**</dependency>**

**2. Create Security Config**

**@Configuration**

**@EnableWebSecurity**

**public class SecurityConfig {**

**@Bean**

**public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {**

**http**

**.authorizeHttpRequests()**

**.requestMatchers("/admin/\*\*").hasRole("ADMIN")**

**.anyRequest().authenticated()**

**.and()**

**.httpBasic(); // Enables Basic Authentication**

**return http.build();**

**}**

**@Bean**

**public InMemoryUserDetailsManager userDetailsService() {**

**UserDetails user = User.withUsername("admin")**

**.password("{noop}admin123")**

**.roles("ADMIN")**

**.build();**

**return new InMemoryUserDetailsManager(user);**

**}**

**}**

**{noop} = no password encoder (for demo only)**

**Internal Working of Spring Security Filters**

**Filter Flow**

**Client Request**

**↓**

**[SecurityFilterChain]**

**↓**

**[UsernamePasswordAuthenticationFilter] → Extracts username/password**

**↓**

**[AuthenticationManager] → Calls...**

**↓**

**[UserDetailsService] → Loads UserDetails**

**↓**

**[SecurityContext] ← Stores Authentication object**

**↓**

**Access either granted or denied**

* **Successful login: Spring stores user info in SecurityContextHolder**
* **Each request: Filters check if user is authenticated and authorized**

**Example Response with Basic Auth**

**Request:**

**GET /admin/dashboard**

**Authorization: Basic YWRtaW46YWRtaW4xMjM=**

**Response:**

* **200 OK (if credentials and role match)**
* **401 Unauthorized (if not authenticated)**
* **403 Forbidden (if authenticated but not authorized)**

**Flowchart Summary**

**[Client Sends Request]**

**↓**

**[SecurityFilterChain]**

**↓**

**[UsernamePasswordAuthenticationFilter]**

**↓**

**[AuthenticationManager]**

**↓**

**[UserDetailsService loads User]**

**↓**

**[SecurityContextHolder stores auth info]**

**↓**

**[Authorization Decision (Role-based)]**

**↓**

**Access GRANTED or DENIED**

**Summary Table**

| **Component** | **Description** |
| --- | --- |
| **SecurityFilterChain** | **Chain of filters that intercept all requests** |
| **AuthenticationManager** | **Authenticates the user** |
| **UserDetailsService** | **Loads user info from DB or memory** |
| **GrantedAuthority** | **Stores user roles** |
| **SecurityContextHolder** | **Holds authenticated user info** |
| **@EnableWebSecurity** | **Enables Spring Security configuration** |
| **.httpBasic()** | **Enables Basic Authentication** |