Spring

- **Spring Basics**
- **Spring Core: IOC & Dependency Injection (DI)**
- **Spring Advanced Features**
- **Spring MVC & Web Development**
- **Spring Data**
- **Spring Integration & Microservices**
- **Spring Testing**
- **Reactive Programming (Optional / Advanced)**
- **Minor Gaps (Optional but Useful)**
- **Projects**

Spring Introduction

Spring is a lightweight, open-source framework for building Java applications. It was created to simplify enterprise Java development, especially compared to older, heavier approaches like EJB (Enterprise Java Beans).

At its core, Spring provides two key things:

- It manages objects for you (beans) and their dependencies.
- It helps you build applications in a loosely coupled, testable, and maintainable way.

To use Spring, you'll need:

- JDK installed
- A build tool like Maven or Gradle
- Spring libraries (managed easily via Maven/Gradle dependencies)

A simple "Hello Spring" example:

```
// Define a simple bean (a class managed by Spring)
public class HelloService {
   public void sayHello() {
      System.out.println("Hello, Spring Framework!");
   }
}
```

To use this bean, you configure Spring with XML or Java-based configuration. Here's a Java-based config:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
@Configuration
public class AppConfig {
   @Bean
   public HelloService helloService() {
      return new HelloService();
   }
}
```

And a small main class:

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
public class MainApp {
   public static void main(String[] args) {
      ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);
      HelloService hello = context.getBean(HelloService.class);
      hello.sayHello();
   }
}
```

When you run this, Spring creates and manages the HelloService object, and you can call it without worrying about manual instantiation.

Spring Modules

Spring is not just one piece of software; it's made up of modules. Each module solves a different problem. The major modules are:

- **Core Container** – Provides IoC and Dependency Injection features.
- **AOP (Aspect-Oriented Programming)** – Handles cross-cutting concerns like logging, security.
- **Data Access / ORM** – Simplifies database access using JDBC or ORM frameworks like Hibernate.
- **Web / Spring MVC** – For web application development.
- **Spring Security** – Adds authentication, authorization, and protection.
- **Spring Batch, Spring Messaging, etc.** – Specialized use cases.

You don't have to learn all modules at once. Start with **Core and IoC/DI**, then expand to MVC, ORM, and Security as you go.

Spring Architecture

Spring follows a layered architecture. The key layers are:

1. **Core Container** – Manages beans and dependency injection.
2. **Context** – Provides access to objects and configurations.
3. **AOP** – Handles aspects (logging, security, transactions).
4. **Data Access Layer** – Deals with JDBC, ORM frameworks.
5. **Web Layer** – For web-related stuff (MVC, REST, etc.).
6. **Test** – Provides support for unit and integration testing.

This layered design means you can use just what you need. For example, if you only want IoC and not MVC, you can do that.

Spring Life Cycle

Beans in Spring follow a lifecycle.

1. Spring instantiates the bean (creates the object).
2. Dependencies are injected.
3. Initialization methods (if any) are called.
4. The bean is ready for use.
5. When the application context closes, destruction methods (if any) are called.

Example:

```
public class DemoBean {
   public void init() {
      System.out.println("Bean is initialized!");
   }

   public void destroy() {
      System.out.println("Bean is destroyed!");
   }
}
```

Configuration:

```
@Configuration
public class BeanConfig {

@Bean(initMethod = "init", destroyMethod = "destroy")
   public DemoBean demoBean() {
      return new DemoBean();
   }
}
```

Running this inside a AnnotationConfigApplicationContext, you'll see initialization message at start and destruction when context closes.

Spring Bean Scope

By default, Spring creates beans as **Singletons** (only one instance for the whole application). But Spring supports multiple scopes:

- **singleton** – One shared instance (default).
- **prototype** – A new instance every time you request the bean.
- **request** – One bean per HTTP request (for web apps).
- **session** – One bean per HTTP session.

Example:

```
@Component
@Scope("prototype")

public class PrototypeBean {
   public PrototypeBean() {
      System.out.println("New Prototype Bean created!");
   }
}
```

Each time you request this bean from the context, you'll see a new instance being created.

Spring Core: IoC & Dependency Injection (DI)

IoC and Dependency Injection (DI)

Inversion of Control (IoC) means that instead of you creating and managing objects manually, Spring takes care of it. You just tell Spring what you need, and it supplies it.

Dependency Injection (DI) is how IoC is implemented. Instead of a class creating its own dependencies, they are injected by Spring.

Example without Spring:

```
public class Car {
   private Engine engine = new Engine(); // tightly coupled
}
```

With Spring (loose coupling):

```
public class Car {
   private Engine engine;
   public Car(Engine engine) {  // dependency injected
      this.engine = engine;
   }
}
```

Now, Spring provides the `Engine` when it creates the `Car` object.

Constructor Injection (CI)

Spring can inject dependencies via a constructor.

Example:

```
public class Engine {
   public void start() {
      System.out.println("Engine started...");
   }
}

public class Car {
   private Engine engine;
   public Car(Engine engine) {   // constructor injection
      this.engine = engine;
   }
   public void drive() {
      engine.start();
      System.out.println("Car is moving...");
   }
}
```

Config with Java:

```
@Configuration
public class AppConfig {
   @Bean
   public Engine engine() {
      return new Engine();
   }
   @Bean
   public Car car() {
      return new Car(engine());
   }
}
```

CI with Dependent Object

If a bean depends on another bean:

```
public class Student {
   private Address address;
   public Student(Address address) {
      this.address = address;
   }
   public void show() {
      System.out.println("Lives at: " + address.getCity());
   }
}

public class Address {
   private String city;
   public Address(String city) {
      this.city = city;
   }
   public String getCity() { return city; }
}
```

Config:

```
@Bean
public Address address() {
   return new Address("Hyderabad");
}
@Bean
public Student student() {
   return new Student(address());
}
```

CI with String Object

```
public class Person {
   private String name;
   public Person(String name) { this.name = name; }
   public void show() { System.out.println("Name: " + name); }
}
```

Config:

```
@Bean
public Person person() {
   return new Person("Mazid");
}
```

CI with Collection

```
public class Library {
   private List<String> books;
   public Library(List<String> books) {
      this.books = books;
   }
   public void showBooks() {
      books.forEach(System.out::println);
   }
}
```

Config:

```
@Bean
public Library library() {
   return new Library(Arrays.asList("Spring in Action", "Java Concurrency", "Clean Code"));
}
```

Setter Injection (SI)

Instead of constructor, you can inject dependencies using setters.

```
public class Employee {
   private String name;
   public void setName(String name) { this.name = name; }
   public void show() { System.out.println("Employee: " + name); }
}
```

Config:

```
@Bean
public Employee employee() {
   Employee e = new Employee();
   e.setName("John");
   return e;
}
```

SI with Dependent Object

```
public class Company {
   private Address address;
   public void setAddress(Address address) { this.address = address; }
   public void show() { System.out.println("Company in " + address.getCity()); }
}
```

Config:

```
@Bean
public Company company() {
   Company c = new Company();
   c.setAddress(address());
   return c;
}
```

SI with String Object

Same as above, but using a string property.

SI with Collection

```
public class Team {
   private List<String> members;

public void setMembers(List<String> members) { this.members = members; }
   public void show() { members.forEach(System.out::println); }
}
```

Config:

```
@Bean
public Team team() {

 Team t = new Team();
   t.setMembers(Arrays.asList("Alice", "Bob", "Charlie"));
   return t;
}
```

Auto-wiring

Instead of manually wiring beans, Spring can automatically inject them.

By Name

If bean name matches property name, Spring injects it.

```
@Component
public class ServiceA {

 public void serve() { System.out.println("Serving..."); }
}

@Component
public class Client {
   @Autowired
   private ServiceA serviceA; // matched by name
}
```

By Type

Spring matches by type.

```
@Component
public class ServiceB { }
@Component
public class Consumer {
   @Autowired
   private ServiceB serviceB; // matched by type
}
```

Profiles & Environment Configuration

Spring Profiles allow you to define beans for different environments (dev, test, prod).

```
@Configuration
@Profile("dev")
public class DevConfig {
   @Bean
   public String datasource() {
      return "Development DB";
   }
}
@Configuration
@Profile("prod")
public class ProdConfig {
   @Bean
   public String datasource() {
      return "Production DB";
   }
}
```

Run with:

```
System.setProperty("spring.profiles.active", "dev");
```

Event Handling (ApplicationEventPublisher)

Spring allows beans to publish and listen to events.

```
public class CustomEvent extends ApplicationEvent {
   public CustomEvent(Object source) { super(source); }
}
```

Publisher:

```
@Component
public class EventPublisher {
   @Autowired
   private ApplicationEventPublisher publisher;
   public void publish() {
      publisher.publishEvent(new CustomEvent(this));
   }
}
```

Listener:

```
@Component
public class EventListenerBean {
   @EventListener
   public void handleEvent(CustomEvent event) {
      System.out.println("Event received: " + event);
   }
}
```

Spring Advanced Features

Aspect-Oriented Programming (AOP)

AOP is about separating **cross-cutting concerns** (like logging, security, transaction handling) from your business logic. Instead of writing logging code in every method, you write it once as an aspect.

Example:

```
@Aspect
@Component
public class LoggingAspect {

 @Before("execution(* com.example.service.*.*(..))")
   public void logBefore() {
      System.out.println("Method execution started...");
   }
   @After("execution(* com.example.service.*.*(..))")
   public void logAfter() {
      System.out.println("Method execution finished...");
   }
}
```

Usage:

```
@Service
public class OrderService {

 public void placeOrder() {
      System.out.println("Placing order...");
   }
}
```

When placeOrder() is called, logs will be printed before and after automatically.

Transaction Management

Transactions ensure that either all operations succeed or none do (atomicity). Spring manages transactions declaratively using @Transactional.

Example:

```
@Service
public class BankService {

@Autowired
   private AccountRepository repo;

@Transactional
   public void transfer(Long fromId, Long toId, double amount) {
      Account from = repo.findById(fromId).get();
      Account to = repo.findById(toId).get();

      from.setBalance(from.getBalance() - amount);
      to.setBalance(to.getBalance() + amount);

      repo.save(from);
      repo.save(to);
   }
}
```

If something fails in between, Spring automatically rolls back.

Caching (Spring Cache abstraction)

Spring provides an easy caching mechanism using annotations.

Setup: Add cache support in config:

```
@EnableCaching
@Configuration

public class CacheConfig { }
```

Usage:

```
@Service
public class ProductService {

 @Cacheable("products")
   public String getProductById(int id) {
      System.out.println("Fetching from DB...");
      return "Product-" + id;
   }
}
```

First call fetches from DB, subsequent calls fetch from cache.

ORM (Hibernate/JPA integration)

Spring integrates easily with ORM frameworks like Hibernate and JPA for database handling.

Entity class:

```
@Entity
public class Student {
   @Id
   @GeneratedValue
   private Long id;
   private String name;
   // getters & setters
}
```

Repository:

```
public interface StudentRepository extends JpaRepository<Student, Long> { }
```

Usage:

```
@Service
public class StudentService {
   @Autowired
   private StudentRepository repo;
   public void saveStudent() {
      Student s = new Student();
      s.setName("John");
      repo.save(s);
   }
}
```

Spring Boot simplifies setup with spring-boot-starter-data-jpa, but you can also configure manually with Spring.

Batch – Handling large-scale batch processing

Spring Batch is used when you need to process large amounts of data in chunks (e.g., reading CSV, processing records, writing to DB).

Basic Job Example:

```
@Configuration
@EnableBatchProcessing
public class BatchConfig {
   @Bean
   public Job job(JobBuilderFactory jobBuilders, StepBuilderFactory stepBuilders) {
      Step step = stepBuilders.get("step1")
            .<String, String>chunk(2)
            .reader(() -> "Hello Batch")
            .processor(item -> item.toUpperCase())
            .writer(items -> items.forEach(System.out::println))
            .build();
      return jobBuilders.get("job1").start(step).build();
```

```
    }
}
```

This simple job reads items, processes them, and writes them out in chunks.

Messaging / WebSockets – Real-time messaging

Spring supports messaging with JMS, RabbitMQ, Kafka, and real-time WebSockets.

Example using WebSocket:

Config:

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {
    @Override
    public void configureMessageBroker(MessageBrokerRegistry config) {
        config.enableSimpleBroker("/topic");
        config.setApplicationDestinationPrefixes("/app");
    }

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        registry.addEndpoint("/chat").withSockJS();
    }
}
```

Controller:

```
@Controller
public class ChatController {
    @MessageMapping("/sendMessage")
    @SendTo("/topic/messages")
    public String sendMessage(String message) {
        return "Received: " + message;
    }
}
```

Now clients can connect via `/chat` WebSocket endpoint and exchange real-time messages.

Spring MVC & Web Development

Spring MVC

Spring MVC follows the **Model-View-Controller** pattern.

- **Model**: Data (Java objects, DTOs, entities).
- **View**: UI (JSP, Thymeleaf, JSON for APIs).
- **Controller**: Handles requests, prepares model, chooses view.

Flow:

1. Request comes to DispatcherServlet.
2. It finds the right Controller.
3. Controller processes data and returns Model + View.
4. Response is rendered back.

Basic Controller:

```
@Controller
public class HomeController {
   @GetMapping("/home")
   public String home(Model model) {
      model.addAttribute("msg", "Welcome to Spring MVC");
      return "home"; // refers to home.jsp or home.html
   }
}
```

Model and Controller

The **Controller** uses the **Model** to pass data to the view.

```
@Controller
public class StudentController {
   @GetMapping("/student")
   public String getStudent(Model model) {
      model.addAttribute("name", "Mazid");
      model.addAttribute("course", "Spring Framework");
      return "student"; // student.jsp or student.html
   }
}
```

The view can display this data dynamically.

REST API Development (GET, POST, PUT, DELETE)

Spring MVC makes building REST APIs simple with @RestController.

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

   private List<String> students = new ArrayList<>(List.of("John", "Alice"));

   @GetMapping
   public List<String> getAll() {
      return students;
   }

   @PostMapping
   public String add(@RequestBody String name) {
```

```
        students.add(name);
        return "Student added: " + name;
    }

    @PutMapping("/{id}")
    public String update(@PathVariable int id, @RequestBody String name) {
        students.set(id, name);
        return "Updated student at index " + id;
    }
    @DeleteMapping("/{id}")
    public String delete(@PathVariable int id) {
        students.remove(id);
        return "Deleted student at index " + id;
    }
}
```

Request Mapping, Path Variables, Request Parameters

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

    @GetMapping("/hello")
    public String hello() {
        return "Hello World";
    }
    @GetMapping("/user/{id}") // Path Variable
    public String getUser(@PathVariable int id) {
        return "User ID: " + id;
    }
    @GetMapping("/search") // Request Parameter
    public String search(@RequestParam String keyword) {
        return "Searching for: " + keyword;
    }
}
```

Exception Handling

Instead of showing ugly errors, use `@ControllerAdvice`.

```
@ControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler(Exception.class)
    public ResponseEntity<String> handleException(Exception e) {
        return new ResponseEntity<>("Error: " + e.getMessage(), HttpStatus.BAD_REQUEST);
    }
}
```

Now, all exceptions return a neat response.

Validation & Data Binding

Spring supports validation with `@Valid` and JSR-303 annotations.

```
public class User {
   @NotEmpty(message = "Name is required")
   private String name;
   @Email(message = "Invalid email")
   private String email;
   // getters/setters
}
```

Controller:

```
@RestController
@RequestMapping("/api/users")
public class UserController {
   @PostMapping
   public ResponseEntity<String> createUser(@Valid @RequestBody User user, BindingResult result) {
      if (result.hasErrors()) {
         return ResponseEntity.badRequest().body(result.getAllErrors().toString());
      }
      return ResponseEntity.ok("User created: " + user.getName());
   }
}
```

Spring Security (Web Applications)

Authentication & Authorization

Authentication = who you are.
Authorization = what you can do.

Basic config:

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
   @Override
   protected void configure(HttpSecurity http) throws Exception {
      http.authorizeRequests()
         .antMatchers("/admin/**").hasRole("ADMIN")
         .antMatchers("/user/**").hasAnyRole("USER","ADMIN")
         .anyRequest().authenticated()
         .and()
         .formLogin()
         .and()
         .httpBasic();
   }
}
```

Role-based Security

Users can have roles, and access is restricted based on roles.

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
```

```
    auth.inMemoryAuthentication()
        .withUser("mazid").password("{noop}password").roles("USER")
        .and()
        .withUser("admin").password("{noop}admin").roles("ADMIN");
}
```

JWT / Token-based Security (Optional)

For stateless REST APIs, we use **JWT tokens** instead of sessions.

Basic flow:

1. User logs in → Server issues JWT.
2. Client stores JWT (usually in headers).
3. For each request, JWT is sent → Server validates.

Spring Security + JWT typically needs:

- A `JwtUtil` class for token creation/validation.
- A `JwtFilter` to intercept requests.
- Configuration to add the filter into Spring Security chain.

Spring Data

Introduction to Spring Data JPA

Spring Data JPA simplifies working with databases by eliminating most boilerplate code for data access. It builds on top of JPA (Java Persistence API) and Hibernate. Instead of writing queries manually, you can use repositories and conventions to handle database operations.

Install / Setup

- Add dependency in `pom.xml`:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

- Configure `application.properties`:

```
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
```

```
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

Mini Example

```
@Entity
public class Student {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;
   private String name;
   private int age;
   // getters and setters
}
public interface StudentRepository extends JpaRepository<Student, Long> {}
```

CRUD Operations & Repositories

CRUD stands for Create, Read, Update, Delete. Spring Data JPA provides methods out-of-the-box through `JpaRepository`.

Mini Example

```
@Autowired
private StudentRepository studentRepository;

public void demo() {

 // Create
   Student s = new Student();
   s.setName("Alice");
   s.setAge(22);
   studentRepository.save(s);

 // Read
   List<Student> students = studentRepository.findAll();

// Update
   Student st = studentRepository.findById(1L).get();
   st.setAge(25);
   studentRepository.save(st);

 // Delete
   studentRepository.deleteById(1L);
}
```

Derived Query Methods

Spring Data JPA allows you to define query methods by just declaring method names following naming conventions.

Mini Example

```
public interface StudentRepository extends JpaRepository<Student, Long> {
   List<Student> findByName(String name);
   List<Student> findByAgeGreaterThan(int age);
   List<Student> findByNameContaining(String keyword);
}
```

Usage:

```
List<Student> result = studentRepository.findByAgeGreaterThan(20);
```

JPQL / Native Queries

Sometimes, you need custom queries. Spring Data allows both JPQL and native SQL.

Mini Example

```
public interface StudentRepository extends JpaRepository<Student, Long> {
   @Query("SELECT s FROM Student s WHERE s.age > :age")
   List<Student> getStudentsOlderThan(@Param("age") int age);

   @Query(value = "SELECT * FROM student WHERE name LIKE %:keyword%", nativeQuery = true)
   List<Student> searchByName(@Param("keyword") String keyword);
}
```

Pagination & Sorting

For large datasets, Spring Data supports pagination and sorting using `Pageable`.

Mini Example

```
public void paginationDemo() {
   Pageable pageable = PageRequest.of(0, 5, Sort.by("name").ascending());
   Page<Student> page = studentRepository.findAll(pageable);

   page.getContent().forEach(s -> System.out.println(s.getName()));
}
```

Spring Integration & Microservices

Spring Integration Basics

Spring Integration provides a lightweight framework for building message-driven and event-driven applications. It enables systems to communicate asynchronously using channels and adapters.

Install / Setup
Add dependency in `pom.xml`:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-integration</artifactId>
</dependency>
```

Mini Example

```
@Configuration
public class IntegrationConfig {
  @Bean
  public MessageChannel inputChannel() {
    return new DirectChannel();
  }

  @Bean
  public MessageChannel outputChannel() {
    return new DirectChannel();
  }

  @Bean
  @ServiceActivator(inputChannel = "inputChannel", outputChannel = "outputChannel")
  public GenericHandler<String> handler() {
    return (payload, headers) -> {
      System.out.println("Received: " + payload);
      return "Processed: " + payload;
    };
  }
}
```

Usage:

```
@Autowired
MessageChannel inputChannel;

public void sendMessage() {
  inputChannel.send(MessageBuilder.withPayload("Hello Integration").build());
}
```

Messaging & Channels

Messaging is the core of Spring Integration, where communication happens via **channels**.

- **DirectChannel** – synchronous communication.
- **QueueChannel** – asynchronous communication.

Mini Example

```
@Bean
public MessageChannel queueChannel() {
  return new QueueChannel();
}
```

Sending message:

```
queueChannel.send(new GenericMessage<>("Async Message"));
Message<?> received = queueChannel.receive();
System.out.println("Received: " + received.getPayload());
```

REST / SOAP Web Services Integration

Spring Integration supports integration with REST APIs and SOAP web services.

REST Example (Consumer)

```
@RestController
@RequestMapping("/students")
public class StudentController {
   @GetMapping("/{id}")
   public String getStudent(@PathVariable int id) {
      return "Student with ID: " + id;
   }
}
```

Calling REST API

```
@Autowired
private RestTemplate restTemplate;

public void callRestApi() {
   String response = restTemplate.getForObject("http://localhost:8080/students/1", String.class);
   System.out.println(response);
}
```

SOAP Integration uses spring-ws starter and generates client stubs.

Spring Cloud Config / Eureka / Ribbon / Gateway

In a **microservices-heavy architecture**, these tools handle service discovery, load balancing, and centralized configuration.

Dependencies (pom.xml)

```
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework.cloud</groupId>
```

```
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
```

Eureka Server Example

```
@SpringBootApplication
@EnableEurekaServer

public class EurekaServerApp {

 public static void main(String[] args) {
       SpringApplication.run(EurekaServerApp.class, args);
    }
}
```

Eureka Client Example

```
@SpringBootApplication
@EnableEurekaClient
public class StudentServiceApp {

public static void main(String[] args) {
       SpringApplication.run(StudentServiceApp.class, args);
    }
}
```

Gateway Example (application.yml)

```
spring:
  cloud:
    gateway:
      routes:
        - id: student-service
          uri: http://localhost:8081
          predicates:
            - Path=/students/**
```

This way, the gateway forwards requests to the student service automatically.

Spring Testing

Unit Testing with JUnit & Mockito

Unit testing ensures individual components (like services) work correctly in isolation. Spring uses **JUnit** for testing and **Mockito** for mocking dependencies.

Dependencies (pom.xml)

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>
```

Mini Example: Service Test

```
@Service
public class CalculatorService {
  public int add(int a, int b) {
    return a + b;
  }
}
@SpringBootTest
class CalculatorServiceTest {
  @Autowired
  private CalculatorService calculatorService;

  @Test
  void testAddition() {
    assertEquals(5, calculatorService.add(2, 3));
  }
}
```

Mockito Example (Mocking a Repository)

```
@Repository
interface StudentRepository extends JpaRepository<Student, Integer> {}
@Service
class StudentService {
  @Autowired
  private StudentRepository repo;
  public String getStudentName(int id) {
    return repo.findById(id).map(Student::getName).orElse("Not Found");
  }
}
@ExtendWith(MockitoExtension.class)
class StudentServiceTest {
  @Mock
  private StudentRepository repo;
  @InjectMocks
  private StudentService service;
  @Test
  void testGetStudentName() {
    Student s = new Student(1, "John");
    when(repo.findById(1)).thenReturn(Optional.of(s));
    assertEquals("John", service.getStudentName(1));
  }
}
```

Integration Testing with Spring Test Framework

Integration tests verify how multiple components work together in a Spring context.

Mini Example: Testing Service + Repository

```
@SpringBootTest
@AutoConfigureTestDatabase
class StudentIntegrationTest {

   @Autowired
   private StudentRepository repo;

   @Test
   void testSaveStudent() {
      Student s = new Student(1, "Alice");
      repo.save(s);
      assertTrue(repo.findById(1).isPresent());
   }
}
```

Here, Spring spins up an in-memory database (H2 by default for tests).

Testing REST APIs

Spring Boot provides **MockMvc** to test REST controllers without deploying them.

Controller Example

```
@RestController
@RequestMapping("/students")
class StudentController {
   @GetMapping("/{id}")
   public String getStudent(@PathVariable int id) {
      return "Student ID: " + id;
   }
}
```

Test with MockMvc

```
@SpringBootTest
@AutoConfigureMockMvc
class StudentControllerTest {

   @Autowired
   private MockMvc mockMvc;

   @Test
   void testGetStudent() throws Exception {
      mockMvc.perform(get("/students/1"))
            .andExpect(status().isOk())
            .andExpect(content().string("Student ID: 1"));
```

```
    }
}
```

This simulates an HTTP call to the controller without running a real server.

Reactive Programming (Optional / Advanced)

Spring WebFlux Basics

Spring WebFlux is the reactive, non-blocking counterpart of Spring MVC.

- Built on **Project Reactor** (Mono, Flux) instead of traditional Servlet API.
- Best for applications needing scalability with thousands of concurrent requests (e.g., chat apps, streaming apps).
- Works with **Netty**, **Undertow**, or **Tomcat (reactive mode)**.

Install / Setup

Add WebFlux dependency in pom.xml:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```

Reactive Streams & Mono / Flux

Reactive streams follow the **Publisher-Subscriber** model.

- **Mono** → Emits 0 or 1 element (like Optional).
- **Flux** → Emits 0…N elements (like List/Stream).

Mini Example

```
@RestController
@RequestMapping("/reactive")
public class ReactiveController {

  @GetMapping("/mono")
  public Mono<String> getMono() {
    return Mono.just("Hello from Mono");
  }
  @GetMapping("/flux")
  public Flux<String> getFlux() {
    return Flux.just("Spring", "WebFlux", "Reactive");
  }
}
```

- /mono → returns single response.
- /flux → returns a stream of responses.

Building Reactive REST APIs

Reactive APIs look similar to MVC controllers but return **Mono/Flux** instead of normal objects.

Entity

```
@Data
@AllArgsConstructor
@NoArgsConstructor
@Document(collection = "students")
public class Student {
   @Id
   private String id;
   private String name;
   private int age;
}
```

Reactive Repository

```
public interface StudentRepository extends ReactiveCrudRepository<Student, String> {
   Flux<Student> findByAgeGreaterThan(int age);
}
```

Reactive Controller

```
@RestController
@RequestMapping("/students")
public class StudentReactiveController {

   @Autowired
   private StudentRepository repository;

   @GetMapping
   public Flux<Student> getAllStudents() {
      return repository.findAll();
   }

   @GetMapping("/{id}")
   public Mono<Student> getStudent(@PathVariable String id) {
      return repository.findById(id);
   }

   @PostMapping
   public Mono<Student> saveStudent(@RequestBody Student student) {
      return repository.save(student);
   }
}
```

This API:

- Returns all students (Flux)
- Returns single student by ID (Mono)
- Saves student (Mono)

Spring Internationalization (i18n) – Multilingual Apps

Internationalization (i18n) is about designing apps that support multiple languages. Spring provides **MessageSource** and property files for managing translations.

Setup

Create property files:

```
# messages.properties (default)
welcome.message=Welcome to Spring App

# messages_fr.properties (French)
welcome.message=Bienvenue dans l'application Spring
```

Configuration

```
@Configuration
public class AppConfig {
   @Bean
   public MessageSource messageSource() {
      ResourceBundleMessageSource source = new ResourceBundleMessageSource();
      source.setBasename("messages");
      source.setDefaultEncoding("UTF-8");
      return source;
   }
}
```

Usage

```
@RestController
public class WelcomeController {
   @Autowired
   private MessageSource messageSource;

   @GetMapping("/welcome")
   public String getMessage(@RequestHeader(name="Accept-Language", required=false) Locale locale)
{
      return messageSource.getMessage("welcome.message", null, locale);
   }
}
```

Now `/welcome` returns English or French depending on the `Accept-Language` header.

Spring Scheduling – @Scheduled Tasks

Spring supports scheduled tasks using annotations. Useful for jobs like sending reports, cleaning up logs, or updating data periodically.

Enable Scheduling

```
@SpringBootApplication
@EnableScheduling
```

```
public class App {}
```

Mini Example

```
@Component
public class SchedulerTask {
   @Scheduled(fixedRate = 5000)   // runs every 5 seconds
   public void fixedRateTask() {
      System.out.println("Task executed at: " + new Date());
   }

   @Scheduled(cron = "0 0 9 * * ?")   // every day at 9 AM
   public void cronTask() {
      System.out.println("Daily report task triggered!");
   }
}
```

Spring JDBC – Basics

Even though ORM (like Hibernate/JPA) is preferred, many interviews still ask **Spring JDBC basics**. It's lightweight and works directly with JDBC.

Dependency

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
```

Configuration (H2 database)

```
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
```

Using JdbcTemplate

```
@Repository
public class StudentDao {
   @Autowired
   private JdbcTemplate jdbcTemplate;

   public void saveStudent(String name, int age) {
      jdbcTemplate.update("INSERT INTO student(name, age) VALUES (?, ?)", name, age);
   }

   public List<Student> getAllStudents() {
      return jdbcTemplate.query("SELECT * FROM student",
         (rs, rowNum) -> new Student(rs.getInt("id"), rs.getString("name"), rs.getInt("age")));
   }
}
```

Projects:

Beginner Projects (Core Spring Concepts)

1. **Simple Inventory Management System**
    - **Features:** Manage products, add/update/delete items
    - **Concepts Covered:** Spring IOC & DI, Constructor & Setter Injection, Bean Scopes, Basic JDBC / DAO layer
    - **Database:** H2 or MySQL
2. **Student Grade Calculator**
    - **Features:** Input student grades, calculate GPA, generate reports
    - **Concepts Covered:** IOC & DI, Collection Injection, Event Handling, Simple AOP (logging)
3. **Library Management (Console App)**
    - **Features:** Add/Remove books, search books, track borrowed books
    - **Concepts Covered:** Spring Core (IOC & DI), Setter/Constructor Injection, Simple JDBC, Bean Lifecycle

Intermediate Projects (Web & Security)

1. **Online Course Registration System**
    - **Features:** Students register for courses, view schedule, admin manages courses
    - **Concepts Covered:** Spring MVC, Controllers & Models, Request Parameters, JSP/Thymeleaf, Form Validation, Exception Handling
    - **Optional:** Spring Security for admin/student roles
2. **Employee Leave Management System (Web)**
    - **Features:** Employees apply for leave, managers approve/reject
    - **Concepts Covered:** Spring MVC, Spring Security (Role-based), Transaction Management, JDBC/ORM integration
3. **Simple E-commerce Product Catalog**
    - **Features:** Browse products, filter by category, view details
    - **Concepts Covered:** Spring MVC, DAO Layer, ORM (Hibernate/JPA), Pagination & Sorting, Exception Handling

Advanced Projects (Integration, AOP, Messaging)

1. **Real-Time Chat Application**
    - **Features:** Chat between multiple users
    - **Concepts Covered:** Spring WebSockets / Messaging, Spring Security for authentication, Event Handling
2. **Banking System with Transactions**
    - **Features:** Account creation, fund transfer, transaction history

    - **Concepts Covered:** Spring ORM (Hibernate/JPA), Transaction Management (Declarative & Programmatic), AOP (Logging/Transactions)
3. **Order Management System with Spring Batch**
    - **Features:** Batch processing of orders (import/export), generate daily reports
    - **Concepts Covered:** Spring Batch, Transactions, Scheduling, Spring JDBC
4. **Integration with External REST/SOAP Services**
    - **Features:** Fetch data from external APIs and process it
    - **Concepts Covered:** Spring Integration, REST & SOAP clients, Message Channels, Transformation & Routing