**✅ 1. What design patterns are commonly used in Spring Boot?**

**Answer:**  
Spring Boot uses several design patterns extensively:

* **Singleton** – Beans are singleton by default in Spring.
* **Factory Method** – Used in BeanFactory and ApplicationContext.
* **Proxy** – Used for AOP (Aspect-Oriented Programming).
* **Template Method** – Seen in JDBC Template, RestTemplate, etc.
* **Dependency Injection (DI)** – Core pattern used in Spring.
* **Builder** – Used in creating complex objects like HttpHeaders, UriComponentsBuilder.
* **Observer** – Event handling in Spring (ApplicationEventPublisher).
* **Strategy** – For interchangeable algorithms (e.g., custom validation, sorting logic).
* **Adapter** – For integration with external APIs or frameworks.

**✅ 2. Explain how the Singleton pattern is implemented in Spring Boot.**

**Answer:**

* By default, Spring beans are **singleton-scoped**.
* Only **one instance** of a bean is created per Spring container.
* This is managed internally by the **ApplicationContext** or **BeanFactory**.

java

CopyEdit

@Service

public class MyService {

// Singleton by default

}

Cross-question: *How to make a prototype bean instead of singleton?*  
Use @Scope("prototype").

**✅ 3. Where is the Factory design pattern used in Spring Boot?**

**Answer:**

* Used in **BeanFactory** and **ApplicationContext**.
* Spring uses **factory methods** to instantiate beans.
* Example: @Bean methods can act like factories.

java

CopyEdit

@Configuration

public class AppConfig {

@Bean

public MyService myService() {

return new MyServiceImpl(); // Factory method

}

}

**✅ 4. How does Spring implement the Proxy pattern?**

**Answer:**

* Used in **AOP** and **transaction management**.
* Creates **proxy objects** around beans using:
  + JDK Dynamic Proxies (for interfaces)
  + CGLIB Proxies (for concrete classes)

java

CopyEdit

@Transaction

public void updateAccount() {

// Proxy intercepts method to apply transaction logic

}

**✅ 5. What is the Template Method pattern? Where is it used in Spring?**

**Answer:**

* A base class defines the **skeleton** of an algorithm.
* Subclasses override specific steps.
* Used in:
  + JdbcTemplate
  + RestTemplate
  + TransactionTemplate

java

CopyEdit

jdbcTemplate.query("SELECT \* FROM users", new RowMapper<User>() {

public User mapRow(ResultSet rs, int rowNum) {

return new User(rs.getString("name"));

}

});

**✅ 6. How does Dependency Injection relate to design patterns?**

**Answer:**

* It's based on **Inversion of Control (IoC)**.
* Achieved using:
  + Constructor Injection
  + Setter Injection
  + Field Injection (not recommended)

java

CopyEdit

@Service

public class OrderService {

private final PaymentService paymentService;

@Autowired

public OrderService(PaymentService paymentService) {

this.paymentService = paymentService;

}

}

**✅ 7. Describe how the Strategy pattern is applied in Spring Boot.**

**Answer:**

* You define **interchangeable behaviors** and inject them based on conditions.
* Example: Implementing payment gateways.

java

CopyEdit

public interface PaymentStrategy {

void pay();

}

@Component("creditCard")

public class CreditCardPayment implements PaymentStrategy {

public void pay() { /\* logic \*/ }

}

@Component("paypal")

public class PayPalPayment implements PaymentStrategy {

public void pay() { /\* logic \*/ }

}

@Service

public class PaymentService {

@Autowired

@Qualifier("creditCard")

private PaymentStrategy paymentStrategy;

}

Cross-question: *How can we make this dynamic at runtime?*  
Use a Map<String, PaymentStrategy> and fetch the strategy using a key.

**✅ 8. Can you explain the Observer pattern in the Spring framework?**

**Answer:**

* Spring uses **Observer** in its **event mechanism**:
  + ApplicationEventPublisher publishes events.
  + Listeners receive and react to events.

java

CopyEdit

@Component

public class MyListener {

@EventListener

public void handleCustomEvent(CustomEvent event) {

// React to event

}

}

**✅ 9. What is the role of Builder pattern in Spring Boot?**

**Answer:**

* Used to build **complex objects** step-by-step.
* Examples:
  + ResponseEntity
  + UriComponentsBuilder
  + HttpHeaders

java

CopyEdit

ResponseEntity.ok()

.header("Content-Type", "application/json")

.body("response");

**✅ 10. Where is the Adapter pattern used in Spring?**

**Answer:**

* Used to **integrate third-party APIs**.
* Spring MVC uses HandlerAdapter to adapt various HandlerMapping strategies.

**✅ 1. What design patterns have you actively used in large-scale Spring Boot applications?**

**Answer:**  
In my 12+ years of experience, I have implemented the following patterns with Spring Boot:

| **Design Pattern** | **Use Case in Spring Boot Application** |
| --- | --- |
| Singleton | Spring beans by default — service, repository, utility classes. |
| Factory | Bean creation using @Bean, dynamic service providers. |
| Proxy | AOP logging, security, retry logic via declarative proxies. |
| Strategy | Dynamic business rules (e.g., promo engine, payment gateway). |
| Template Method | Abstract reusable flows (e.g., CSV/XML parser template). |
| Builder | REST API response builders, domain object creation. |
| Observer | Event-based architecture using Spring’s ApplicationEvent. |
| Chain of Responsibility | Spring Security filter chains, middleware pipelines. |
| Adapter | Third-party service integrations like legacy SOAP to REST. |
| Command | Decoupling tasks in microservice orchestrations. |

**✅ 2. Can you describe a real use of the Strategy Pattern in a Spring Boot microservice?**

**Answer:**  
I implemented a dynamic **pricing strategy** service in a travel application. Depending on booking source (web/mobile/agent), different discount logic applied:

java

CopyEdit

public interface PricingStrategy {

BigDecimal calculateFare(Booking booking);

}

@Component("agentStrategy")

public class AgentPricingStrategy implements PricingStrategy { ... }

@Component("webStrategy")

public class WebPricingStrategy implements PricingStrategy { ... }

@Service

public class PricingService {

@Autowired private Map<String, PricingStrategy> strategies;

public BigDecimal calculate(String channel, Booking booking) {

return strategies.get(channel + "Strategy").calculateFare(booking);

}

}

🔁 **Cross-question:**  
*How would you manage hundreds of such strategies?*  
→ Group by feature, apply Factory pattern with config-based bean resolution.

**✅ 3. Explain how Proxy and AOP enable cross-cutting concerns in Spring Boot.**

**Answer:**

* **Spring AOP** uses **proxy pattern** (JDK Dynamic Proxy or CGLIB) to wrap beans.
* It enables adding functionality like logging, transaction, security, without modifying actual business logic.

java

CopyEdit

@Aspect

@Component

public class LoggingAspect {

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

public void logBefore(JoinPoint joinPoint) {

log.info("Calling: " + joinPoint.getSignature());

}

}

🔁 **Cross-question:**  
*How would AOP behave with self-invocation?*  
→ Proxy won’t intercept self-calls; need to refactor into separate beans or use AspectJ.

**✅ 4. How have you used the Builder pattern to improve code readability and testability?**

**Answer:**

* Used **Builder pattern** for constructing immutable domain models (e.g., BookingRequest, FlightDetails).
* Helps in clean unit tests and reduces constructor clutter.

java

CopyEdit

BookingRequest request = BookingRequest.builder()

.flightId("AI101")

.userId("user1")

.seatClass("Economy")

.build();

🔁 **Cross-question:**  
*How does Lombok’s @Builder help here?*  
→ Reduces boilerplate, supports method chaining, works well with immutability.

**✅ 5. Describe a use case where you used the Observer pattern in a decoupled event-driven design.**

**Answer:**  
In an e-commerce app, on successful payment, I needed to:

* Send confirmation email
* Update inventory
* Notify third-party analytics

Instead of tightly coupling, I used Spring Events:

java

CopyEdit

@Component

public class PaymentService {

@Autowired private ApplicationEventPublisher publisher;

public void processPayment(Order order) {

// logic

publisher.publishEvent(new PaymentSuccessEvent(order));

}

}

java

CopyEdit

@EventListener

public void sendEmail(PaymentSuccessEvent event) {

// handle separately

}

🔁 **Cross-question:**  
*What are pitfalls?*

* Events are async (unless annotated), error handling is tricky.
* Use Transactional Events or Spring Cloud Stream for better guarantees.

**✅ 6. How have you implemented the Chain of Responsibility pattern using Spring Security filters?**

**Answer:**  
Spring Security uses this pattern internally:

* Authentication → Authorization → CSRF → Headers → Logout

Custom implementation example:

java

CopyEdit

@Component

public class IpBlacklistFilter extends OncePerRequestFilter {

protected void doFilterInternal(...) {

// check if IP is blacklisted

filterChain.doFilter(request, response);

}

}

Then add this filter to the security chain:

java

CopyEdit

http.addFilterBefore(new IpBlacklistFilter(), UsernamePasswordAuthenticationFilter.class);

**✅ 7. Can you explain the difference between Factory and Abstract Factory in Spring context?**

**Answer:**

* **Factory Pattern**: Simple creation logic; used via @Bean, FactoryBean, etc.
* **Abstract Factory Pattern**: Families of related objects without specifying concrete classes.

Use case:  
Abstract factory for generating platform-specific beans (e.g., AWS, Azure, GCP services) based on environment.

**✅ 8. How would you use Template Method pattern for service layer abstraction?**

**Answer:**  
Created an **abstract template service** to handle audit, logging, and retry logic:

java

CopyEdit

public abstract class AbstractServiceTemplate<T> {

public final Response execute(T request) {

preProcess(request);

Response result = process(request);

postProcess(result);

return result;

}

protected abstract Response process(T request);

}

Each microservice then extends and overrides only business logic.

**✅ 9. How do you handle design patterns in a distributed microservices architecture?**

**Answer:**  
Used combinations:

| **Pattern** | **Use Case in Microservices** |
| --- | --- |
| Circuit Breaker | Resilience4j or Hystrix for failing dependencies. |
| Adapter | Integrate with legacy services or third-party APIs. |
| Command | Represent service calls with retry/fallback (e.g., Command Pattern in orchestration). |
| Aggregator | Gateway pattern to collect data from multiple services. |

**✅ 10. What are anti-patterns to avoid when using design patterns in Spring Boot?**

**Answer:**

* Overusing Singleton for stateful beans.
* Abusing Proxy logic (e.g., too many aspects → performance hit).
* Using Factory without externalizing config (tight-coupling).
* Writing too many Abstract classes – prefer composition over inheritance.
* Not handling event-driven failures (in Observer/Event patterns).