```
Spring Framework Interview:
```

🟢 Cấp độ 1 - Cơ bản

➡️ 1. Spring Boot là gì?

Spring Boot là một framework mở rộng từ Spring, giúp phát triển ứng dụng Java nhanh chóng với cấu hình tự động, tích hợp sẵn server (như Tomcat) và hỗ trợ giám sát, logging dễ dàng.

➡️ 2. Khác biệt giữa Spring và Spring Boot

Tiêu chí	Spring Framework	Spring Boot
Cấu hình	Thủ công	Tự động (Auto-configuration)
Triển khai	WAR + server ngoài	JAR + embedded server
Quản lý dependencies	Khai báo từng cái	Dùng Starter POMs có sẵn

---

➡️ 3. Ưu điểm của Spring Boot

- Tự động cấu hình (Auto-configuration)
- Tích hợp server nội bộ (Embedded server)
- Tạo dự án nhanh với Spring Initializr
- Cấu hình qua `application.properties` hoặc `application.yml`
- Hỗ trợ RESTful API dễ dàng

➡️ 4. Starter là gì?

Là module định nghĩa sẵn các dependency cho một mục đích cụ thể, ví dụ:

- `spring-boot-starter-web`: cho ứng dụng web
- `spring-boot-starter-data-jpa`: cho JPA

➡️ 5. Spring Initializr

Công cụ tạo nhanh project Spring Boot: https://start.spring.io

---

➡️ 6. `@SpringBootApplication` gồm những gì?

Gộp 3 annotation:

- `@Configuration`
- `@EnableAutoConfiguration`
- `@ComponentScan`

➡️ 7. Auto-Configuration là gì?

Spring Boot tự cấu hình các thành phần dựa trên thư viện có trong classpath.

➡️ 8. Spring Boot Actuator

Cung cấp các endpoint để giám sát ứng dụng như:

- `/actuator/health`
- `/actuator/metrics`

➡️ 9. Spring Boot DevTools

Cung cấp:

- Tự động restart
- LiveReload
- Cấu hình tối ưu cho phát triển

➡️ 10. `application.properties` vs `application.yml`

Tính năng	`application.properties`	`application.yml`
Cú pháp	Key-Value đơn giản	YAML – cấu trúc rõ ràng
Phù hợp cho	Cấu hình đơn giản	Cấu hình lồng nhau

➡️ 11. Đọc cấu hình

- `@Value("${key}")`: cho giá trị đơn
- `@ConfigurationProperties`: cho nhiều cấu hình có cấu trúc

➡️ 12. `@Value` vs `@ConfigurationProperties`

Tiêu chí	`@Value`	`@ConfigurationProperties`
Đơn giản	✓	✕ (cấu trúc phức tạp hơn)
Phù hợp	Giá trị đơn	Nhóm cấu hình
Hỗ trợ xác thực	✕	✓ (`@Validated`)

➡️ 13. `@RestController` vs `@Controller`

- `@Controller`: Trả về view (MVC)
- `@RestController`: Trả về dữ liệu (JSON/XML)

➡️ 14. `@Component` vs `@Service` vs `@Repository` vs `@Bean`

Annotation	Vai trò
`@Component`	Bean thông thường
`@Service`	Logic nghiệp vụ (Service Layer)
`@Repository`	Truy cập dữ liệu (DAO)
`@Bean`	Khai báo bean thủ công

➡️ 15. Spring Profiles

Cho phép cấu hình theo môi trường (dev, test, prod) bằng:

```
spring.profiles.active=dev
```

🟡 Cấp độ 2 - Trung cấp

➡️ 16. Constructor Injection vs Field Injection

Tiêu chí	Constructor Injection	Field Injection
Immutability	✓ (dùng `final`)	×
Test dễ	✓	× (cần framework hỗ trợ)
Phát hiện lỗi sớm	✓	×
Cú pháp	Dài hơn	Ngắn hơn

Khuyên dùng: Constructor Injection vì rõ ràng, dễ test, bảo trì tốt hơn.

➡️ 17. Dependency Injection (DI) là gì?

Spring DI là cơ chế Spring cung cấp dependency cho các object. DI giúp:

- Tách biệt logic nghiệp vụ và khởi tạo đối tượng
- Quản lý lifecycle của bean

3 kiểu DI:

- **Constructor Injection** (ưu tiên dùng)
- **Setter Injection**
- **Field Injection** (ít dùng)

Ví dụ:

```
@Service
public class OrderService {
    private final PaymentService paymentService;

    public OrderService(PaymentService paymentService) {
        this.paymentService = paymentService;
    }
}
```

Mẹo:

- Dùng `@Qualifier` khi có nhiều bean cùng kiểu
- Dùng Lombok với `@RequiredArgsConstructor` để gọn code

➡️ 18. Spring Boot hỗ trợ các cơ sở dữ liệu như thế nào?

Mở rộng câu trả lời:

- Spring Boot hỗ trợ kết nối và làm việc với nhiều loại cơ sở dữ liệu (SQL và NoSQL) thông qua các **starter** và cấu hình đơn giản. Một số cơ chế chính:
    - **JPA/Hibernate**: Sử dụng `spring-boot-starter-data-jpa` để làm việc với các cơ sở dữ liệu quan hệ như MySQL, PostgreSQL, Oracle.
    - **JDBC**: Sử dụng `spring-boot-starter-jdbc` để thực hiện các truy vấn SQL trực tiếp.
    - **NoSQL**: Hỗ trợ MongoDB (`spring-boot-starter-data-mongodb`), Redis (`spring-boot-starter-data-redis`), và Cassandra (`spring-boot-starter-data-cassandra`).
    - **Cấu hình**:

      ```
      # MySQL configuration
      spring.datasource.url=jdbc:mysql://localhost:3306/mydb
      spring.datasource.username=root
      spring.datasource.password=password
      spring.jpa.hibernate.ddl-auto=update
      ```

      hoặc YAML:

      ```
      ```

```yaml
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/mydb
    username: root
    password: password
  jpa:
    hibernate:
      ddl-auto: update
```

- **Ví dụ thực tế**:
  - **Ứng dụng quản lý nhân sự**:

```java
@Entity
public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String department;
    // Getters and setters
}

@Repository
public interface EmployeeRepository extends
JpaRepository<Employee, Long> {
    List<Employee> findByDepartment(String department);
}
```

    Trong ví dụ này, Spring Boot tự động cấu hình kết nối tới MySQL và cung cấp các phương thức CRUD thông qua `JpaRepository`.

- **Ứng dụng thực tế**:
  - **Trong thương mại điện tử**: Sử dụng JPA để lưu trữ thông tin sản phẩm, đơn hàng, và khách hàng trong MySQL hoặc PostgreSQL. Redis có thể được dùng để cache danh sách sản phẩm hot nhằm giảm tải cho cơ sở dữ liệu chính.
  - **Trong ứng dụng phân tích dữ liệu**: MongoDB được sử dụng để lưu trữ dữ liệu phi cấu trúc như log hoặc dữ liệu người dùng, trong khi JPA xử lý các bảng cấu trúc như báo cáo tài chính.
  - **Hệ thống thời gian thực**: Redis được sử dụng để lưu trữ trạng thái phiên người dùng (session) hoặc dữ liệu tạm thời để tăng tốc độ truy xuất.

- **Mẹo**:
  - Sử dụng **HikariCP** (mặc định trong Spring Boot) để quản lý connection pool, tối ưu hiệu suất kết nối cơ sở dữ liệu.

- Đối với các ứng dụng lớn, hãy cấu hình **Flyway** hoặc **Liquibase** để quản lý schema migration.
- Khi làm việc với NoSQL, hãy cân nhắc sử dụng các thư viện như **Spring Data MongoDB** để tận dụng các tính năng như query method.

➡️ 19. Giải thích các annotation `@Entity`, `@Id`, `@GeneratedValue` trong JPA?

Mở rộng câu trả lời:

- `@Entity`: Đánh dấu một lớp là một thực thể JPA, ánh xạ tới một bảng trong cơ sở dữ liệu. Mỗi instance của lớp đại diện cho một hàng trong bảng.
  - **Ví dụ**:

```
@Entity
public class Product {
    @Id
    private Long id;
    private String name;
    private Double price;
    // Getters and setters
}
```

    Lớp `Product` sẽ ánh xạ tới bảng `product` trong cơ sở dữ liệu.

- `@Id`: Đánh dấu trường là khóa chính của thực thể. Mỗi thực thể phải có ít nhất một trường được đánh dấu `@Id`.
  - **Lưu ý**: Có thể sử dụng các kiểu dữ liệu như `Long`, `String`, hoặc `UUID` tùy thuộc vào yêu cầu.
- `@GeneratedValue`: Chỉ định chiến lược sinh giá trị tự động cho khóa chính. Các chiến lược phổ biến:
  - `GenerationType.AUTO`: Để JPA tự chọn chiến lược phù hợp (thường là `IDENTITY` hoặc `SEQUENCE`).
  - `GenerationType.IDENTITY`: Sử dụng cột tự tăng của cơ sở dữ liệu (phổ biến với MySQL).
  - `GenerationType.SEQUENCE`: Sử dụng sequence của cơ sở dữ liệu (phổ biến với PostgreSQL, Oracle).
  - `GenerationType.TABLE`: Sử dụng bảng riêng để lưu trữ giá trị khóa chính (ít được dùng).
  - **Ví dụ**:

```
@Entity
public class Order {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
    private Long id;
    private String orderNumber;
    // Getters and setters
}
```

- **Ứng dụng thực tế**:
  - **Trong hệ thống quản lý kho**: Sử dụng `@Entity` để ánh xạ các lớp như `InventoryItem`, `@Id` để đánh dấu mã sản phẩm duy nhất, và `@GeneratedValue` để tự động tạo ID cho mỗi mục hàng mới.
  - **Trong ứng dụng đặt vé**: Một thực thể `Ticket` có thể sử dụng `@Id` với `GenerationType.SEQUENCE` để tạo mã vé duy nhất theo thứ tự tăng dần, đảm bảo không trùng lặp.
  - **Trong hệ thống tài chính**: Sử dụng `@Entity` để ánh xạ các giao dịch (`Transaction`), với `@Id` và `@GeneratedValue` để tạo mã giao dịch tự động, giúp theo dõi dễ dàng.
- **Mẹo**:
  - Luôn sử dụng `@GeneratedValue` với chiến lược phù hợp với cơ sở dữ liệu (ví dụ: `IDENTITY` cho MySQL, `SEQUENCE` cho PostgreSQL).
  - Nếu cần khóa chính phức tạp (composite key), sử dụng `@EmbeddedId` hoặc `@IdClass`.
  - Kiểm tra schema cơ sở dữ liệu để đảm bảo rằng cột khóa chính được cấu hình đúng với chiến lược đã chọn.

➡️ 20. `CrudRepository` và `JpaRepository` khác nhau thế nào?

Mở rộng câu trả lời:

- `CrudRepository` cung cấp các phương thức cơ bản để thực hiện các thao tác CRUD (Create, Read, Update, Delete).
  - **Phương thức chính**:
    - `save(T entity)`: Lưu hoặc cập nhật thực thể.
    - `findById(ID id)`: Tìm thực thể theo ID.
    - `findAll()`: Lấy tất cả thực thể.
    - `deleteById(ID id)`: Xóa thực thể theo ID.
  - **Ví dụ**:

```
public interface UserRepository extends CrudRepository<User,
Long> {
    // Không cần định nghĩa phương thức CRUD, đã có sẵn
}
```

- `JpaRepository` mở rộng `CrudRepository` và thêm các tính năng như phân trang (pagination), sắp xếp (sorting), và các phương thức tiện ích khác.

- **Phương thức bổ sung**:
  - `findAll(Pageable pageable)`: Lấy danh sách thực thể với phân trang.
  - `findAll(Sort sort)`: Lấy danh sách thực thể với sắp xếp.
  - `deleteAllInBatch()`: Xóa tất cả thực thể trong một lần thực thi.
- **Ví dụ**:

```
public interface UserRepository extends JpaRepository<User,
Long> {
    Page<User> findByRole(String role, Pageable pageable);
}
```

- **So sánh chi tiết**:

Tiêu chí	`CrudRepository`	`JpaRepository`
Phạm vi tính năng	Chỉ hỗ trợ CRUD cơ bản	Hỗ trợ CRUD + phân trang, sắp xếp
Hiệu suất	Nhẹ hơn, ít phương thức hơn	Nặng hơn do có thêm nhiều tính năng
Tính linh hoạt	Ít linh hoạt hơn	Linh hoạt hơn với các ứng dụng lớn

- **Ứng dụng thực tế**:
  - **Trong ứng dụng quản lý blog**: Sử dụng `CrudRepository` cho các thao tác đơn giản như thêm, sửa, xóa bài viết. Nếu cần hiển thị danh sách bài viết phân trang trên giao diện người dùng, hãy sử dụng `JpaRepository` với `findAll(Pageable pageable)`.
  - **Trong hệ thống thương mại điện tử**: `JpaRepository` được sử dụng để lấy danh sách sản phẩm với phân trang và sắp xếp theo giá hoặc lượt xem, giúp tối ưu trải nghiệm người dùng.
  - **Trong ứng dụng báo cáo**: Sử dụng `JpaRepository` để lấy dữ liệu thống kê với các tiêu chí sắp xếp và phân trang, ví dụ: danh sách giao dịch theo ngày hoặc trạng thái.

- **Mẹo**:
  - Sử dụng `CrudRepository` trong các dự án nhỏ hoặc khi chỉ cần các thao tác cơ bản để giảm chi phí tài nguyên.
  - Với `JpaRepository`, hãy cẩn thận khi sử dụng các phương thức như `findAll()` mà không có phân trang vì có thể gây tải nặng nếu bảng dữ liệu lớn.
  - Kết hợp với **Spring Data JPA Query Methods** để định nghĩa các truy vấn tùy chỉnh mà không cần viết SQL.

🔴 Cấp độ 3 - Nâng cao (Mở rộng)

➡️ 21. Các cách quản lý transaction trong Spring Boot?

Mở rộng câu trả lời:

- **Transaction** trong Spring Boot đảm bảo tính toàn vẹn dữ liệu khi thực hiện các thao tác với cơ sở dữ liệu. Spring cung cấp hai cách chính để quản lý transaction:

- **Annotation-based (@Transactional)**:
  - Sử dụng annotation `@Transactional` trên phương thức hoặc lớp để khai báo rằng các thao tác trong phạm vi đó sẽ được thực thi trong một transaction.
  - Các thuộc tính quan trọng:
    - `propagation`: Quy định cách transaction được lan truyền (ví dụ: `REQUIRED`, `REQUIRES_NEW`).
    - `isolation`: Quy định mức độ cách ly của transaction (ví dụ: `READ_COMMITTED`, `SERIALIZABLE`).
    - `rollbackOn`: Chỉ định các ngoại lệ gây rollback (mặc định là `RuntimeException`).
  - **Ví dụ**:

```
@Service
public class OrderService {
    @Transactional
    public void processOrder(Order order) {
        orderRepository.save(order);
        paymentService.processPayment(order.getAmount());
    }
}
```

    Trong ví dụ này, nếu `processPayment` thất bại, toàn bộ transaction sẽ rollback, đảm bảo `order` không được lưu.

- **Programmatic Transaction Management**:
  - Sử dụng `TransactionTemplate` hoặc `PlatformTransactionManager` để kiểm soát transaction một cách thủ công.
  - **Ví dụ**:

```
  @Service
  public class OrderService {
      private final TransactionTemplate
transactionTemplate;

      public OrderService(PlatformTransactionManager
transactionManager) {
          this.transactionTemplate = new
TransactionTemplate(transactionManager);
      }

      public void processOrder(Order order) {
          transactionTemplate.execute(new
TransactionCallbackWithoutResult() {
              @Override
              protected void
doInTransactionWithoutResult(TransactionStatus status) {
```

```
                    orderRepository.save(order);

            paymentService.processPayment(order.getAmount());
                }
            });
        }
    }
```

- **Ứng dụng thực tế**:
  - **Trong hệ thống ngân hàng**: Sử dụng `@Transactional` để đảm bảo rằng khi chuyển tiền, cả hai thao tác trừ tiền từ tài khoản nguồn và cộng tiền vào tài khoản đích đều thành công, hoặc không thao tác nào được thực hiện nếu có lỗi.
  - **Trong thương mại điện tử**: Khi xử lý đơn hàng, `@Transactional` đảm bảo rằng việc lưu đơn hàng, trừ hàng tồn kho, và ghi log giao dịch được thực hiện đồng bộ. Nếu bất kỳ bước nào thất bại (ví dụ: hết hàng), toàn bộ transaction sẽ rollback.
  - **Trong hệ thống đặt lịch**: Sử dụng transaction để đảm bảo rằng khi đặt một lịch hẹn, cả trạng thái lịch và thông tin người dùng đều được cập nhật đồng thời.
- **Mẹo**:
  - Chỉ sử dụng `@Transactional` trên các phương thức public, vì Spring sử dụng proxy để quản lý transaction.
  - Cẩn thận với các transaction dài (long-running transactions) vì chúng có thể làm giảm hiệu suất cơ sở dữ liệu.
  - Sử dụng `REQUIRES_NEW` khi cần tách biệt transaction con để tránh ảnh hưởng đến transaction chính.

➡️ 22. Spring Boot hỗ trợ Asynchronous như thế nào?

Mở rộng câu trả lời:

- Spring Boot hỗ trợ xử lý bất đồng bộ (asynchronous) thông qua annotation `@Async`, cho phép các phương thức chạy trong một thread riêng biệt, không chặn thread chính của ứng dụng.
  - **Cấu hình**:
    - Kích hoạt hỗ trợ async bằng annotation `@EnableAsync` trên lớp cấu hình hoặc lớp chính của ứng dụng.
    - Đánh dấu phương thức với `@Async` để chạy bất đồng bộ.
  - **Ví dụ**:

```
@SpringBootApplication
@EnableAsync
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
```

```
}

@Service
public class NotificationService {
    @Async
    public CompletableFuture<String> sendEmail(String
recipient, String message) {
        // Giả lập gửi email
        Thread.sleep(2000);
        return CompletableFuture.completedFuture("Email sent
to " + recipient);
    }
}
```

- **Thread Pool Configuration**:
  - Theo mặc định, Spring sử dụng `SimpleAsyncTaskExecutor`, nhưng bạn có thể tùy chỉnh thread pool:

```
@Configuration
public class AsyncConfig implements AsyncConfigurer {
    @Override
    public Executor getAsyncExecutor() {
        ThreadPoolTaskExecutor executor = new
ThreadPoolTaskExecutor();
        executor.setCorePoolSize(5);
        executor.setMaxPoolSize(10);
        executor.setQueueCapacity(25);
        executor.initialize();
        return executor;
    }
}
```

- **Ứng dụng thực tế**:
  - **Trong hệ thống thương mại điện tử**: Sử dụng `@Async` để gửi email xác nhận đơn hàng hoặc thông báo khuyến mãi mà không làm chậm quá trình đặt hàng của người dùng.
  - **Trong ứng dụng phân tích dữ liệu**: Xử lý các tác vụ nặng như tính toán thống kê hoặc xử lý log trong background để không ảnh hưởng đến giao diện người dùng.
  - **Trong hệ thống chat**: Gửi thông báo đẩy (push notification) bất đồng bộ để đảm bảo phản hồi nhanh chóng cho người dùng.

- **Mẹo**:
  - Luôn trả về `CompletableFuture` hoặc `Future` trong các phương thức `@Async` để xử lý kết quả bất đồng bộ.
  - Tránh gọi phương thức `@Async` trực tiếp trong cùng một lớp (vì Spring sử dụng proxy), hãy tách thành các service riêng.

- Theo dõi hiệu suất của thread pool để tránh tình trạng quá tải khi có quá nhiều tác vụ bất đồng bộ.

➡️ 23. Giải thích về Bean Scopes trong Spring Boot?

Mở rộng câu trả lời:

- **Bean Scopes** xác định vòng đời và phạm vi của một bean trong Spring Container. Spring Boot hỗ trợ các scope sau:
  - **Singleton** (mặc định): Chỉ một instance duy nhất của bean được tạo cho toàn bộ ứng dụng.
    - **Ví dụ**:

```
@Service
public class SingletonService {
    // Chỉ một instance được tạo
}
```

  - **Prototype**: Mỗi lần request bean, Spring tạo một instance mới.
    - **Ví dụ**:

```
@Service
@Scope("prototype")
public class PrototypeService {
    // Mỗi lần inject, một instance mới được tạo
}
```

  - **Request**: Một instance được tạo cho mỗi HTTP request (chỉ áp dụng trong ứng dụng web).
  - **Session**: Một instance được tạo cho mỗi HTTP session.
  - **Application**: Một instance được tạo cho toàn bộ vòng đời của ứng dụng web.
  - **Websocket**: Một instance được tạo cho mỗi kết nối WebSocket.
- **Ứng dụng thực tế**:
  - **Singleton**: Phù hợp với các service hoặc repository được sử dụng chung, như `UserService` hoặc `OrderRepository` trong một hệ thống thương mại điện tử.
  - **Prototype**: Sử dụng khi cần các instance riêng biệt cho mỗi yêu cầu, ví dụ: một lớp xử lý dữ liệu tạm thời trong quá trình nhập liệu.
  - **Request/Session**: Trong ứng dụng web, sử dụng để lưu trữ thông tin người dùng trong một phiên (session) hoặc trạng thái của một yêu cầu cụ thể (request), ví dụ: giỏ hàng của người dùng.
  - **Application**: Sử dụng cho các cấu hình hoặc tài nguyên toàn cục, như cấu hình bảo mật hoặc thông tin ứng dụng.
- **Mẹo**:

- Hạn chế sử dụng **Prototype** trong các ứng dụng lớn vì có thể gây rò rỉ bộ nhớ nếu không quản lý đúng vòng đời của bean.
- Khi sử dụng **Request** hoặc **Session** scope, hãy đảm bảo ứng dụng chạy trong môi trường web (Spring Web hoặc Spring Boot Web).
- Theo dõi số lượng bean trong container để tối ưu hiệu suất.

➡️ 24. Spring Boot hỗ trợ Security như thế nào?

Mở rộng câu trả lời:

- Spring Boot tích hợp bảo mật thông qua **Spring Security** với starter `spring-boot-starter-security`. Nó cung cấp các tính năng như xác thực (authentication), phân quyền (authorization), bảo vệ CSRF, và hỗ trợ OAuth2/JWT.
    - **Cấu hình cơ bản**:

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain
securityFilterChain(HttpSecurity http) throws Exception {
        http
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/public/**").permitAll()
                .anyRequest().authenticated()
            )
            .formLogin(form -> form
                .loginPage("/login")
                .permitAll()
            )
            .logout(logout -> logout.permitAll());
        return http.build();
    }

    @Bean
    public UserDetailsService userDetailsService() {
        UserDetails user = User.withDefaultPasswordEncoder()
            .username("user")
            .password("password")
            .roles("USER")
            .build();
        return new InMemoryUserDetailsManager(user);
    }
}
```

- **Các tính năng chính**:

- **Xác thực**: Hỗ trợ nhiều cơ chế như form login, OAuth2, JWT, LDAP.
- **Phân quyền**: Quy định quyền truy cập dựa trên vai trò (role) hoặc quyền (authority).
- **Bảo mật API**: Sử dụng JWT hoặc OAuth2 để bảo vệ các RESTful API.
- **Bảo vệ CSRF/XSS**: Spring Security tự động kích hoạt CSRF protection cho các form.

- **Ứng dụng thực tế**:
  - **Trong ứng dụng quản lý nhân sự**: Sử dụng Spring Security để phân quyền, đảm bảo chỉ admin mới có thể chỉnh sửa thông tin nhân viên, trong khi nhân viên chỉ có thể xem thông tin cá nhân.
  - **Trong hệ thống thương mại điện tử**: Kích hoạt OAuth2 để cho phép người dùng đăng nhập bằng Google hoặc Facebook, đồng thời sử dụng JWT để bảo vệ các API như `/api/orders`.
  - **Trong ứng dụng tài chính**: Sử dụng Spring Security để mã hóa mật khẩu, kiểm tra quyền truy cập vào các giao dịch nhạy cảm, và ghi log các hành động của người dùng.

- **Mẹo**:
  - Sử dụng **BCryptPasswordEncoder** để mã hóa mật khẩu thay vì các phương pháp lỗi thời như MD5 hoặc SHA.
  - Khi tích hợp OAuth2, hãy cấu hình **refresh token** để tăng cường bảo mật.
  - Kiểm tra các endpoint `/actuator` để đảm bảo chúng được bảo vệ đúng cách trong môi trường sản xuất.

➡️ 25. Giải thích về CORS trong Spring Boot?

Mở rộng câu trả lời:

- **CORS (Cross-Origin Resource Sharing)** là cơ chế cho phép hoặc hạn chế các yêu cầu HTTP từ các domain khác nhau. Spring Boot hỗ trợ cấu hình CORS để đảm bảo ứng dụng web an toàn và linh hoạt khi giao tiếp với các client từ domain khác.
  - **Cấu hình với `@CrossOrigin`**:

```
@RestController
@CrossOrigin(origins = "http://frontend.com")
public class ProductController {
    @GetMapping("/products")
    public List<Product> getProducts() {
        return productService.findAll();
    }
}
```

  - **Cấu hình toàn cục**:

```java
@Configuration
public class WebConfig implements WebMvcConfigurer {
    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/api/**")
            .allowedOrigins("http://frontend.com")
            .allowedMethods("GET", "POST")
            .allowedHeaders("*")
            .allowCredentials(true);
    }
}
```

- **Ứng dụng thực tế**:
  - **Trong ứng dụng web**: Khi frontend chạy trên `http://localhost:3000` (React/Vue) và backend chạy trên `http://localhost:8080`, CORS được cấu hình để cho phép frontend gửi yêu cầu tới backend.
  - **Trong hệ thống microservices**: Các dịch vụ chạy trên các domain khác nhau (ví dụ: `auth-service.com` và `product-service.com`) cần CORS để giao tiếp.
  - **Trong ứng dụng đa nền tảng**: Một ứng dụng mobile gọi API từ backend Spring Boot cần cấu hình CORS để tránh lỗi "Access-Control-Allow-Origin".
- **Mẹo**:
  - Chỉ định rõ `allowedOrigins` thay vì sử dụng `*` để tăng cường bảo mật.
  - Nếu sử dụng Spring Security, hãy đảm bảo cấu hình CORS tương thích với các quy tắc bảo mật.
  - Theo dõi các lỗi CORS trong console trình duyệt để điều chỉnh cấu hình phù hợp.
