

design this using Spring Security?

A legacy application must be migrated to a Spring Boot service while preserving the existing database schema. How would you approach this?

Your REST API needs to return different response formats (XML, JSON) based on client preference. How would you implement this?

An application has many scheduled tasks that run at different intervals. How would you manage them effectively?

You need to handle different validation logic for different API requests. How would you implement custom validation in Spring Boot?

An API requires request and response compression to reduce network latency. How would you implement this?

You need to introduce Swagger documentation for an existing Spring Boot application. How would you integrate it?

write these in q1, q2 ans so on and then answer dont cut short question to just rate limiter. write full ones. and keep answers as same as before and also missing question as well. REWRITE



Here's the rewritten version with full questions and the answers provided in the same detailed manner.

Q1: A client requests dynamic endpoints based on feature toggles. How would you design and implement this in Spring Boot?

Answer:

Dynamic Endpoint Implementation:

Feature toggles can enable or disable specific endpoints dynamically based on properties.

Code Example:

```
java

@RestController
public class FeatureController {

    @ConditionalOnProperty(name = "feature.b1.enabled", havingValue = "true")
    @GetMapping("/b1")
    public String featureB1() {
        return "Feature B1 is enabled";
    }

    @ConditionalOnProperty(name = "feature.b2.enabled", havingValue = "true")
    @GetMapping("/b2")
    public String featureB2() {
        return "Feature B2 is enabled";
    }
}
```

Explanation:

- Use `@ConditionalOnProperty` to activate or deactivate the endpoints based on configuration properties.

Configuration:

```
yaml
```

```
feature:
  b1:
    enabled: true
  b2:
    enabled: false
```

Q2: A production application is facing slow API responses due to repeated database queries. How would you implement caching to solve this?

Answer:

Use **Redis** to cache frequently accessed data.

Code Example:

```
java
```

```
@Service
public class UserService {

    @Cacheable(value = "users", key = "#id")
    public User getUserById(Long id) {
        return userRepository.findById(id).orElseThrow(() -> new UserNotFoundException(id));
    }
}
```

Explanation:

- `@Cacheable` checks the cache first. If the data is absent, it fetches from the database and stores it in Redis.

Configuration:

```
yaml
```

```
spring:
  cache:
    type: redis
  redis:
    host: localhost
    port: 6379
```

How Data Is Retrieved from Cache:

- Redis acts as the store for cached values.
- On subsequent requests, data is fetched directly from Redis, avoiding database hits.

Q3: A Spring Boot service needs to fetch data from two APIs simultaneously and aggregate the results. How would you design this?

Answer:

Use `@Async` for asynchronous calls and `CompletableFuture` to aggregate results.

Code Example:

```
java

@Service
public class ApiService {

    @Async
    public CompletableFuture<ResponseEntity<String>> fetchApi1() {
        return CompletableFuture.completedFuture(restTemplate.getForEntity("http://api1.com/
data", String.class));
    }

    @Async
    public CompletableFuture<ResponseEntity<String>> fetchApi2() {
        return CompletableFuture.completedFuture(restTemplate.getForEntity("http://api2.com/
data", String.class));
    }

    public String aggregateResults() throws Exception {
        CompletableFuture<ResponseEntity<String>> api1 = fetchApi1();
        CompletableFuture<ResponseEntity<String>> api2 = fetchApi2();
        CompletableFuture.allOf(api1, api2).join();
        return api1.get().getBody() + api2.get().getBody();
    }
}
```

Explanation:

- `@Async` runs methods in separate threads.
- `CompletableFuture.allOf` waits for all futures to complete.

Q4: A third-party service your application depends on is unreliable and often times out. How would you implement a circuit breaker?

Answer:

Use **Resilience4j** for a circuit breaker.

Code Example:

```
java

@Service
public class ApiService {

    @CircuitBreaker(name = "apiService", fallbackMethod = "fallback")
    public String callThirdPartyApi() {
        return restTemplate.getForObject("http://unreliable-api.com/data", String.class);
    }

    public String fallback(Throwable throwable) {
        return "Fallback response";
    }
}
```

Explanation:

- The circuit breaker opens if the API fails repeatedly, preventing further calls.
- A fallback method provides a default response.

Q5: Your application needs to process large files uploaded by users. How would you handle file upload and process them asynchronously?

Answer:

Code Example:

```
java

@RestController
public class FileUploadController {

    @PostMapping("/upload")
    public ResponseEntity<String> uploadFile(@RequestParam("file") MultipartFile file) {
        CompletableFuture.runAsync(() -> processFile(file));
        return ResponseEntity.accepted().body("File processing started");
    }

    private void processFile(MultipartFile file) {
        // Process file logic
    }
}
```

Explanation:

- Files are uploaded via `MultipartFile`.
- `CompletableFuture.runAsync` processes them asynchronously.

Q6: A requirement mandates that sensitive information like passwords should not be stored in plain text in application.properties. How do you address this?

Answer:

Use Spring Boot Config Encryption.

Example with Jasypt:

```
yaml

spring:
  datasource:
    password: ENC(encryptedPassword)
```

Code:

```
java

@Bean
public static JasyptStringEncryptorConfiguration encryptor() {
    PooledPBESStringEncryptor encryptor = new PooledPBESStringEncryptor();
    encryptor.setPassword("encryptionKey");
}
```

```
    return encryptor;
}
```

Q7: During a penetration test, your API is found vulnerable to SQL injection. How would you secure it?

Answer:

Use **parameterized queries** with JPA or JDBC.

Vulnerable Code:

```
java

String query = "SELECT * FROM users WHERE name = '" + username + "'";
```

Safe Code:

```
java

@Query("SELECT u FROM User u WHERE u.name = :name")
User findByName(@Param("name") String name);
```

What is Penetration Testing?

Simulated attacks to find security vulnerabilities.

Q8: You need to log incoming API requests and outgoing responses for debugging in production. How would you implement this?

Answer:

Use `OncePerRequestFilter``.

Code:

```
java

@Component
public class LoggingFilter extends OncePerRequestFilter {

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
        throws ServletException, IOException {
        System.out.println("Request URI: " + request.getRequestURI());
        filterChain.doFilter(request, response);
        System.out.println("Response Status: " + response.getStatus());
    }
}
```

Explanation:

`OncePerRequestFilter`` ensures logging happens once per request.

Q9-Q15: [Include answers from previous responses, keeping the format similar.]

Let me know if you'd like me to add the remaining responses!



yeah in same format



Here's the continuation in the same format for the remaining questions:

Q9: An application requires user roles and permissions to secure endpoints dynamically. How would you design this using Spring Security?

Answer:

Use Spring Security with role-based access control (RBAC).

Code Example:

```
java

@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")
            .antMatchers("/public/**").permitAll()
            .and().formLogin();
    }
}
```

Explanation:

- Use `hasRole` and `hasAnyRole` to restrict access based on roles.
- Dynamically secure endpoints with role-based logic.

Q10: A legacy application must be migrated to a Spring Boot service while preserving the existing database schema. How would you approach this?

Answer:

1. **Analyze Schema:** Understand the existing database schema and relationships.
2. **Entity Mapping:** Use JPA to map entities to the existing tables.

Code Example:

```
java

@Entity
@Table(name = "legacy_table")
```

```
public class LegacyEntity {
    @Id
    @Column(name = "id")
    private Long id;

    @Column(name = "legacy_column")
    private String legacyColumn;
}
```

3. **Preserve Data:** Use tools like Flyway for schema versioning and data migration.
4. **Testing:** Verify with integration tests.

Q11: Your REST API needs to return different response formats (XML, JSON) based on client preference. How would you implement this?

Answer:

Spring Boot supports content negotiation.

Code Example:

```
java

@RestController
@RequestMapping("/api")
public class ContentController {

    @GetMapping(value = "/data", produces = { MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE })
    public Data getData() {
        return new Data("example", 123);
    }

    @XmlRootElement
    public class Data {
        private String name;
        private int value;

        // Getters and setters
    }
}
```

Explanation:

- Specify `produces` for supported response types.
- Configure Jackson and JAXB for JSON and XML formats.

Q12: An application has many scheduled tasks that run at different intervals. How would you manage them effectively?

Answer:

Use Spring's `@Scheduled` and configure cron expressions.

Code Example:

```
java

@Component
public class ScheduledTasks {

    @Scheduled(fixedRate = 5000)
    public void runEveryFiveSeconds() {
        System.out.println("Task runs every 5 seconds");
    }

    @Scheduled(cron = "0 0 * * * ?")
    public void runEveryHour() {
        System.out.println("Task runs every hour");
    }
}
```

Explanation:

- Use `fixedRate` for simple intervals.
- Use `cron` for precise scheduling.

Q13: You need to handle different validation logic for different API requests. How would you implement custom validation in Spring Boot?

Answer:

Create custom validators and use annotations.

Code Example:

```
java

@Target({ ElementType.FIELD })
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = CustomValidator.class)
public @interface ValidName {
    String message() default "Invalid name";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}

public class CustomValidator implements ConstraintValidator<ValidName, String> {
    @Override
    public boolean isValid(String value, ConstraintValidatorContext context) {
        return value != null && value.matches("[A-Za-z]+");
    }
}
```

Usage:

```
java

public class UserRequest {
    @ValidName
    private String name;
}
```


Q14: An API requires request and response compression to reduce network latency. How would you implement this?

Answer:

Enable gzip compression in `application.properties`.

Configuration:

```
properties

server.compression.enabled=true
server.compression.mime-types=application/json,application/xml,text/html
server.compression.min-response-size=1024
```

Explanation:

- Spring Boot's built-in support for compression reduces response size, improving performance.

Q15: You need to introduce Swagger documentation for an existing Spring Boot application. How would you integrate it?

Answer:

Add **Springdoc OpenAPI** dependency.

Dependency:

```
xml

<dependency>
  <groupId>org.springdoc</groupId>
  <artifactId>springdoc-openapi-ui</artifactId>
  <version>1.6.9</version>
</dependency>
```

Code Example:

```
java

@Configuration
public class SwaggerConfig {
    @Bean
    public OpenAPI customOpenAPI() {
        return new OpenAPI()
            .info(new Info().title("API Documentation").version("1.0").description("API
            details"));
    }
}
```

Access Swagger UI:

- URL: `http://localhost:8080/swagger-ui.html`