Cracking the Spring Boot Coding Challenge: Common Interview Tasks Solved

Technical interviews for Java developers increasingly include hands-on Spring Boot coding challenges that test both your theoretical knowledge and practical implementation skills. Whether you're interviewing for a junior position or a senior role, being prepared for these challenges can make the difference between success and failure.

In this comprehensive guide, I'll walk you through the most common Spring Boot coding challenges you'll encounter in interviews, providing step-by-step solutions and explaining the reasoning behind each approach. By the end, you'll have the confidence to tackle even the most demanding Spring Boot interview tasks.

Challenge 1: Building a RESTful API from Scratch

The Challenge: "Create a RESTful API for a simple blog application with posts and comments. Implement CRUD operations for posts and allow adding comments to posts."

This is perhaps the most common Spring Boot challenge in interviews. Let's break it down into manageable steps:

Step 1: Set Up the Project Structure

```
 // Entity classes
@Entity
public class Post {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String title;

    @Column(columnDefinition = "TEXT")
    private String content;

    @JsonFormat(pattern = "yyyy-MM-dd HH:mm:ss")
    private LocalDateTime createdAt;

    @OneToMany(mappedBy = "post", cascade = CascadeType.ALL, fetch =
FetchType.LAZY)
    private List<Comment> comments = new ArrayList<>();

    // Getters, setters, constructors
}




@Entity
public class Comment {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(columnDefinition = "TEXT")
    private String content;

    private String author;

    @JsonFormat(pattern = "yyyy-MM-dd HH:mm:ss")
    private LocalDateTime createdAt;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "post_id")
    @JsonIgnore
    private Post post;

    // Getters, setters, constructors
}
```

Step 2: Create the Repositories

```
 public interface PostRepository extends JpaRepository<Post, Long> {
    List<Post> findAllByOrderByCreatedAtDesc();
}


public interface CommentRepository extends JpaRepository<Comment, Long> {
    List<Comment> findByPostIdOrderByCreatedAtDesc(Long postId);
}
```

Step 3: Implement the Services

```
 @Service
public class PostService {
    private final PostRepository postRepository;

    public PostService(PostRepository postRepository) {
        this.postRepository = postRepository;
    }

    public List<Post> getAllPosts() {
        return postRepository.findAllByOrderByCreatedAtDesc();
    }

    public Post getPostById(Long id) {
        return postRepository.findById(id)
            .orElseThrow(() -> new ResourceNotFoundException("Post not found
with id: " + id));
    }

    public Post createPost(Post post) {
        post.setCreatedAt(LocalDateTime.now());
        return postRepository.save(post);
    }

    public Post updatePost(Long id, Post postDetails) {
        Post post = getPostById(id);
        post.setTitle(postDetails.getTitle());
        post.setContent(postDetails.getContent());
        return postRepository.save(post);
    }

    public void deletePost(Long id) {
        Post post = getPostById(id);
        postRepository.delete(post);
    }
}



@Service
public class CommentService {
    private final CommentRepository commentRepository;
    private final PostRepository postRepository;

    public CommentService(CommentRepository commentRepository,
PostRepository postRepository) {
        this.commentRepository = commentRepository;
        this.postRepository = postRepository;
    }

    public List<Comment> getCommentsByPostId(Long postId) {
        return commentRepository.findByPostIdOrderByCreatedAtDesc(postId);
    }

    public Comment addCommentToPost(Long postId, Comment comment) {
        Post post = postRepository.findById(postId)
            .orElseThrow(() -> new ResourceNotFoundException("Post not found
```

```
with id: " + postId));

        comment.setPost(post);
        comment.setCreatedAt(LocalDateTime.now());
        return commentRepository.save(comment);
    }
}
```

Step 4: Create the Controllers

```
 @RestController
@RequestMapping("/api/posts")
public class PostController {
    private final PostService postService;

    public PostController(PostService postService) {
        this.postService = postService;
    }

    @GetMapping
    public List<Post> getAllPosts() {
        return postService.getAllPosts();
    }

    @GetMapping("/{id}")
    public Post getPostById(@PathVariable Long id) {
        return postService.getPostById(id);
    }

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public Post createPost(@RequestBody Post post) {
        return postService.createPost(post);
    }

    @PutMapping("/{id}")
    public Post updatePost(@PathVariable Long id, @RequestBody Post post) {
        return postService.updatePost(id, post);
    }

    @DeleteMapping("/{id}")
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public void deletePost(@PathVariable Long id) {
        postService.deletePost(id);
    }
}



@RestController
@RequestMapping("/api/posts/{postId}/comments")
public class CommentController {
    private final CommentService commentService;

    public CommentController(CommentService commentService) {
        this.commentService = commentService;
    }

    @GetMapping
    public List<Comment> getCommentsByPostId(@PathVariable Long postId) {
        return commentService.getCommentsByPostId(postId);
    }

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public Comment addCommentToPost(
```

```
            @PathVariable Long postId,
            @RequestBody Comment comment) {
        return commentService.addCommentToPost(postId, comment);
    }
}
```

Step 5: Handle Exceptions

```
 @ResponseStatus(HttpStatus.NOT_FOUND)
public class ResourceNotFoundException extends RuntimeException {
    public ResourceNotFoundException(String message) {
        super(message);
    }
}


@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<ErrorResponse> handleResourceNotFoundException(
            ResourceNotFoundException ex, WebRequest request) {
        ErrorResponse errorResponse = new ErrorResponse(
            HttpStatus.NOT_FOUND.value(),
            ex.getMessage(),
            LocalDateTime.now()
        );
        return new ResponseEntity<>(errorResponse, HttpStatus.NOT_FOUND);
    }

    // Other exception handlers
}
@Data
@AllArgsConstructor
public class ErrorResponse {
    private int status;
    private String message;
    private LocalDateTime timestamp;
}
```

Interview Insights: This challenge tests your ability to:

1. Implement proper REST API design
2. Handle entity relationships (One-to-Many)
3. Implement proper exception handling
4. Structure your code in a maintainable way

Challenge 2: Implementing Authentication and Authorization

The Challenge: "Extend the blog API to include user authentication with JWT tokens and role-based access control."

Authentication and authorization are critical for any production application, and interviewers want to know you can implement them securely.

Step 1: Add User Entity and Repository

```
 @Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(unique = true)
    private String username;

    private String password;

    private String email;

    @ElementCollection(fetch = FetchType.EAGER)
    private Set<String> roles = new HashSet<>();

    // Getters, setters, constructors
}



public interface UserRepository extends JpaRepository<User, Long> {
    Optional<User> findByUsername(String username);
    boolean existsByUsername(String username);
    boolean existsByEmail(String email);
}
```

Step 2: Configure Security

```
 @Configuration
@EnableWebSecurity
public class SecurityConfig {

    private final UserDetailsService userDetailsService;
    private final JwtAuthenticationEntryPoint unauthorizedHandler;

    public SecurityConfig(
            UserDetailsService userDetailsService,
            JwtAuthenticationEntryPoint unauthorizedHandler) {
        this.userDetailsService = userDetailsService;
        this.unauthorizedHandler = unauthorizedHandler;
    }

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws
Exception {
        http
            .csrf(csrf -> csrf.disable())
            .exceptionHandling(exception -> exception
                .authenticationEntryPoint(unauthorizedHandler))
            .sessionManagement(session -> session
                .sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/api/auth/**").permitAll()
                .requestMatchers(HttpMethod.GET,
"/api/posts/**").permitAll()
                .requestMatchers("/api/posts/**").hasRole("ADMIN")
                .anyRequest().authenticated());

        http.addFilterBefore(
            authenticationJwtTokenFilter(),
            UsernamePasswordAuthenticationFilter.class);

        return http.build();
    }

    @Bean
    public AuthTokenFilter authenticationJwtTokenFilter() {
        return new AuthTokenFilter();
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Bean
    public AuthenticationManager authenticationManager(
            AuthenticationConfiguration authConfig) throws Exception {
        return authConfig.getAuthenticationManager();
    }
}
```

Step 3: Implement JWT Utilities

```
 @Component
public class JwtUtils {
    @Value("${app.jwt.secret}")
    private String jwtSecret;

    @Value("${app.jwt.expiration}")
    private int jwtExpirationMs;

    public String generateJwtToken(Authentication authentication) {
        UserDetailsImpl userPrincipal = (UserDetailsImpl)
authentication.getPrincipal();

        return Jwts.builder()
            .setSubject(userPrincipal.getUsername())
            .setIssuedAt(new Date())
            .setExpiration(new Date(System.currentTimeMillis() +
jwtExpirationMs))
            .signWith(SignatureAlgorithm.HS512, jwtSecret)
            .compact();
    }

    public String getUsernameFromJwtToken(String token) {
        return Jwts.parser()
            .setSigningKey(jwtSecret)
            .parseClaimsJws(token)
            .getBody()
            .getSubject();
    }

    public boolean validateJwtToken(String authToken) {
        try {

Jwts.parser().setSigningKey(jwtSecret).parseClaimsJws(authToken);
            return true;
        } catch (Exception e) {
            // Log exception
            return false;
        }
    }
}
```

Step 4: Implement Authentication Filter

```
 public class AuthTokenFilter extends OncePerRequestFilter {
    @Autowired
    private JwtUtils jwtUtils;

    @Autowired
    private UserDetailsServiceImpl userDetailsService;

    @Override
    protected void doFilterInternal(
            HttpServletRequest request,
            HttpServletResponse response,
            FilterChain filterChain) throws ServletException, IOException {
        try {
            String jwt = parseJwt(request);
            if (jwt != null && jwtUtils.validateJwtToken(jwt)) {
                String username = jwtUtils.getUsernameFromJwtToken(jwt);

                UserDetails userDetails =
userDetailsService.loadUserByUsername(username);
                UsernamePasswordAuthenticationToken authentication =
                    new UsernamePasswordAuthenticationToken(
                        userDetails, null, userDetails.getAuthorities());

                authentication.setDetails(
                    new
WebAuthenticationDetailsSource().buildDetails(request));


SecurityContextHolder.getContext().setAuthentication(authentication);
            }
        } catch (Exception e) {
            // Log exception
        }

        filterChain.doFilter(request, response);
    }

    private String parseJwt(HttpServletRequest request) {
        String headerAuth = request.getHeader("Authorization");

        if (StringUtils.hasText(headerAuth) && headerAuth.startsWith("Bearer
")) {
            return headerAuth.substring(7);
        }

        return null;
    }
}
```

Step 5: Implement Authentication Controller

```
 @RestController
@RequestMapping("/api/auth")
public class AuthController {
    private final AuthenticationManager authenticationManager;
    private final UserRepository userRepository;
    private final PasswordEncoder encoder;
    private final JwtUtils jwtUtils;

    public AuthController(
            AuthenticationManager authenticationManager,
            UserRepository userRepository,
            PasswordEncoder encoder,
            JwtUtils jwtUtils) {
        this.authenticationManager = authenticationManager;
        this.userRepository = userRepository;
        this.encoder = encoder;
        this.jwtUtils = jwtUtils;
    }

    @PostMapping("/login")
    public ResponseEntity<?> authenticateUser(@RequestBody LoginRequest
loginRequest) {
        Authentication authentication = authenticationManager.authenticate(
            new UsernamePasswordAuthenticationToken(
                loginRequest.getUsername(),
                loginRequest.getPassword()));


SecurityContextHolder.getContext().setAuthentication(authentication);
        String jwt = jwtUtils.generateJwtToken(authentication);

        UserDetailsImpl userDetails = (UserDetailsImpl)
authentication.getPrincipal();
        List<String> roles = userDetails.getAuthorities().stream()
            .map(GrantedAuthority::getAuthority)
            .collect(Collectors.toList());

        return ResponseEntity.ok(new JwtResponse(
            jwt,
            userDetails.getId(),
            userDetails.getUsername(),
            userDetails.getEmail(),
            roles));
    }

    @PostMapping("/register")
    public ResponseEntity<?> registerUser(@RequestBody SignupRequest
signUpRequest) {
        if (userRepository.existsByUsername(signUpRequest.getUsername())) {
            return ResponseEntity.badRequest()
                .body(new MessageResponse("Error: Username is already
taken!"));
        }

        if (userRepository.existsByEmail(signUpRequest.getEmail())) {
            return ResponseEntity.badRequest()
```

```
                .body(new MessageResponse("Error: Email is already in
use!"));
        }

        // Create new user's account
        User user = new User(
            signUpRequest.getUsername(),
            encoder.encode(signUpRequest.getPassword()),
            signUpRequest.getEmail());

        Set<String> strRoles = signUpRequest.getRoles();
        Set<String> roles = new HashSet<>();

        if (strRoles == null) {
            roles.add("ROLE_USER");
        } else {
            strRoles.forEach(role -> {
                switch (role) {
                    case "admin":
                        roles.add("ROLE_ADMIN");
                        break;
                    default:
                        roles.add("ROLE_USER");
                }
            });
        }

        user.setRoles(roles);
        userRepository.save(user);

        return ResponseEntity.ok(new MessageResponse("User registered
successfully!"));
    }
}
```

Interview Insights: This challenge tests your ability to:

1. Implement JWT-based authentication
2. Configure Spring Security properly
3. Handle user registration and login
4. Implement role-based access control

Challenge 3: Implementing Validation and Error Handling

The Challenge: "Enhance the blog API with proper validation for all inputs and comprehensive error handling."

Proper validation and error handling are crucial for robust applications, and employers want to ensure you understand their importance.

Step 1: Add Validation Dependencies

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

Step 2: Add Validation to Entities and DTOs

```
 public class PostRequest {
    @NotBlank(message = "Title cannot be blank")
    @Size(min = 5, max = 100, message = "Title must be between 5 and 100
characters")
    private String title;

    @NotBlank(message = "Content cannot be blank")
    @Size(min = 10, message = "Content must be at least 10 characters")
    private String content;

    // Getters, setters
}
public class CommentRequest {
    @NotBlank(message = "Content cannot be blank")
    @Size(min = 5, max = 500, message = "Comment must be between 5 and 500
characters")
    private String content;

    @NotBlank(message = "Author cannot be blank")
    @Size(min = 2, max = 50, message = "Author name must be between 2 and 50
characters")
    private String author;

    // Getters, setters
}
```

Step 3: Update Controllers to Use Validation

```
 @RestController
@RequestMapping("/api/posts")
public class PostController {
    // ... other methods

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public Post createPost(@Valid @RequestBody PostRequest postRequest) {
        Post post = new Post();
        post.setTitle(postRequest.getTitle());
        post.setContent(postRequest.getContent());
        return postService.createPost(post);
    }

    @PutMapping("/{id}")
    public Post updatePost(
            @PathVariable Long id,
            @Valid @RequestBody PostRequest postRequest) {
        Post post = new Post();
        post.setTitle(postRequest.getTitle());
        post.setContent(postRequest.getContent());
        return postService.updatePost(id, post);
    }
}
```

Step 4: Implement Comprehensive Error Handling

```
 @ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<ErrorResponse> handleResourceNotFoundException(
            ResourceNotFoundException ex, WebRequest request) {
        ErrorResponse errorResponse = new ErrorResponse(
            HttpStatus.NOT_FOUND.value(),
            ex.getMessage(),
            LocalDateTime.now()
        );
        return new ResponseEntity<>(errorResponse, HttpStatus.NOT_FOUND);
    }

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<ValidationErrorResponse>
handleValidationExceptions(
            MethodArgumentNotValidException ex) {
        ValidationErrorResponse errorResponse = new ValidationErrorResponse(
            HttpStatus.BAD_REQUEST.value(),
            "Validation error",
            LocalDateTime.now()
        );

        ex.getBindingResult().getAllErrors().forEach(error -> {
            String fieldName = ((FieldError) error).getField();
            String errorMessage = error.getDefaultMessage();
            errorResponse.getErrors().put(fieldName, errorMessage);
        });

        return new ResponseEntity<>(errorResponse, HttpStatus.BAD_REQUEST);
    }

    @ExceptionHandler(DataIntegrityViolationException.class)
    public ResponseEntity<ErrorResponse>
handleDataIntegrityViolationException(
            DataIntegrityViolationException ex) {
        ErrorResponse errorResponse = new ErrorResponse(
            HttpStatus.CONFLICT.value(),
            "Database error: " + ex.getMostSpecificCause().getMessage(),
            LocalDateTime.now()
        );
        return new ResponseEntity<>(errorResponse, HttpStatus.CONFLICT);
    }

    @ExceptionHandler(Exception.class)
    public ResponseEntity<ErrorResponse> handleGlobalException(
            Exception ex, WebRequest request) {
        ErrorResponse errorResponse = new ErrorResponse(
            HttpStatus.INTERNAL_SERVER_ERROR.value(),
            "An unexpected error occurred: " + ex.getMessage(),
            LocalDateTime.now()
        );
        return new ResponseEntity<>(errorResponse,
HttpStatus.INTERNAL_SERVER_ERROR);
    }
```

```
}
@Data
public class ValidationErrorResponse extends ErrorResponse {
    private Map<String, String> errors = new HashMap<>();

    public ValidationErrorResponse(int status, String message, LocalDateTime
timestamp) {
        super(status, message, timestamp);
    }
}
```

Interview Insights: This challenge tests your ability to:

1. Implement proper input validation
2. Handle different types of exceptions appropriately
3. Provide meaningful error messages
4. Secure your application against invalid inputs

Challenge 4: Implementing Pagination and Filtering

The Challenge: "Update the blog API to support pagination, sorting, and filtering of posts."

As data grows, efficient data retrieval becomes crucial. This challenge tests your ability to implement these features.

Step 1: Update Repository Methods

```
 public interface PostRepository extends JpaRepository<Post, Long> {
    Page<Post> findAll(Pageable pageable);

    Page<Post> findByTitleContainingIgnoreCase(String title, Pageable
pageable);

    @Query("SELECT p FROM Post p WHERE " +
           "LOWER(p.title) LIKE LOWER(CONCAT('%', :keyword, '%')) OR " +
           "LOWER(p.content) LIKE LOWER(CONCAT('%', :keyword, '%'))")
    Page<Post> searchPosts(String keyword, Pageable pageable);
}
```

Step 2: Update Service Methods

```
 @Service
public class PostService {
    private final PostRepository postRepository;

    public PostService(PostRepository postRepository) {
        this.postRepository = postRepository;
    }

    public Page<Post> getAllPosts(int page, int size, String sortBy, String
direction) {
        Sort sort = Sort.by(
            direction.equalsIgnoreCase("desc") ? Sort.Direction.DESC :
Sort.Direction.ASC,
            sortBy
        );
        Pageable pageable = PageRequest.of(page, size, sort);
        return postRepository.findAll(pageable);
    }

    public Page<Post> searchPosts(String keyword, int page, int size, String
sortBy, String direction) {
        Sort sort = Sort.by(
            direction.equalsIgnoreCase("desc") ? Sort.Direction.DESC :
Sort.Direction.ASC,
            sortBy
        );
        Pageable pageable = PageRequest.of(page, size, sort);

        if (keyword == null || keyword.isEmpty()) {
            return postRepository.findAll(pageable);
        }

        return postRepository.searchPosts(keyword, pageable);
    }

    // Other methods
}
```

Step 3: Update Controller Methods

```
 @RestController
@RequestMapping("/api/posts")
public class PostController {
    private final PostService postService;

    public PostController(PostService postService) {
        this.postService = postService;
    }

    @GetMapping
    public Page<Post> getAllPosts(
            @RequestParam(defaultValue = "0") int page,
            @RequestParam(defaultValue = "10") int size,
            @RequestParam(defaultValue = "createdAt") String sortBy,
            @RequestParam(defaultValue = "desc") String direction) {
        return postService.getAllPosts(page, size, sortBy, direction);
    }

    @GetMapping("/search")
    public Page<Post> searchPosts(
            @RequestParam(required = false) String keyword,
            @RequestParam(defaultValue = "0") int page,
            @RequestParam(defaultValue = "10") int size,
            @RequestParam(defaultValue = "createdAt") String sortBy,
            @RequestParam(defaultValue = "desc") String direction) {
        return postService.searchPosts(keyword, page, size, sortBy,
direction);
    }

    // Other methods
}
```

Interview Insights: This challenge tests your ability to:

1. Use Spring Data's pagination and sorting capabilities
2. Implement custom queries for filtering
3. Design flexible API endpoints that support various query parameters
4. Optimize database queries for performance

Challenge 5: Implementing Caching

The Challenge: "Implement caching for frequently accessed blog posts and optimize the API for performance."

Caching is a critical performance optimization technique, and interviewers want to ensure you understand how to implement it properly.

Step 1: Add Caching Dependencies

```
 <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-cache</artifactId>
</dependency>
<dependency>
    <groupId>com.github.ben-manes.caffeine</groupId>
    <artifactId>caffeine</artifactId>
</dependency>
```

Step 2: Configure Caching

```
 @Configuration
@EnableCaching
public class CacheConfig {

    @Bean
    public Caffeine<Object, Object> caffeineConfig() {
        return Caffeine.newBuilder()
            .expireAfterWrite(60, TimeUnit.MINUTES)
            .initialCapacity(100)
            .maximumSize(500);
    }

    @Bean
    public CacheManager cacheManager(Caffeine<Object, Object> caffeine) {
        CaffeineCacheManager cacheManager = new CaffeineCacheManager();
        cacheManager.setCaffeine(caffeine);
        return cacheManager;
    }
}
```

Step 3: Implement Caching in Services

```
 @Service
public class PostService {
    private final PostRepository postRepository;

    public PostService(PostRepository postRepository) {
        this.postRepository = postRepository;
    }

    @Cacheable(value = "posts", key = "#id")
    public Post getPostById(Long id) {
        return postRepository.findById(id)
            .orElseThrow(() -> new ResourceNotFoundException("Post not found
with id: " + id));
    }

    @CacheEvict(value = "posts", key = "#id")
    public Post updatePost(Long id, Post postDetails) {
        Post post = getPostById(id);
        post.setTitle(postDetails.getTitle());
        post.setContent(postDetails.getContent());
        return postRepository.save(post);
    }

    @CacheEvict(value = "posts", key = "#id")
    public void deletePost(Long id) {
        Post post = getPostById(id);
        postRepository.delete(post);
    }

    @CachePut(value = "posts", key = "#result.id")
    public Post createPost(Post post) {
        post.setCreatedAt(LocalDateTime.now());
        return postRepository.save(post);
    }

    @Cacheable(value = "postsList", key = "T(String).format('%d:%d:%s:%s',
#page, #size, #sortBy, #direction)")
    public Page<Post> getAllPosts(int page, int size, String sortBy, String
direction) {
        Sort sort = Sort.by(
            direction.equalsIgnoreCase("desc") ? Sort.Direction.DESC :
Sort.Direction.ASC,
            sortBy
        );
        Pageable pageable = PageRequest.of(page, size, sort);
        return postRepository.findAll(pageable);
    }
}
```

Step 4: Add Cache Monitoring

```
 @RestController
@RequestMapping("/api/management/cache")
public class CacheController {

    private final CacheManager cacheManager;

    public CacheController(CacheManager cacheManager) {
        this.cacheManager = cacheManager;
    }

    @GetMapping
    public Map<String, Object> getCacheStats() {
        Map<String, Object> stats = new HashMap<>();

        cacheManager.getCacheNames().forEach(cacheName -> {
            Cache cache = cacheManager.getCache(cacheName);
            if (cache instanceof CaffeineCache) {
                CaffeineCache caffeineCache = (CaffeineCache) cache;
                com.github.benmanes.caffeine.cache.Cache<Object, Object>
nativeCache =
                    caffeineCache.getNativeCache();

                stats.put(cacheName, Map.of(
                    "estimatedSize", nativeCache.estimatedSize(),
                    "stats", nativeCache.stats()
                ));
            }
        });

        return stats;
    }

    @DeleteMapping("/{cacheName}")
    public ResponseEntity<Void> clearCache(@PathVariable String cacheName) {
        Cache cache = cacheManager.getCache(cacheName);
        if (cache != null) {
            cache.clear();
            return ResponseEntity.ok().build();
        } else {
            return ResponseEntity.notFound().build();
        }
    }

    @DeleteMapping
    public ResponseEntity<Void> clearAllCaches() {
        cacheManager.getCacheNames().forEach(cacheName -> {
            Cache cache = cacheManager.getCache(cacheName);
            if (cache != null) {
                cache.clear();
            }
        });
        return ResponseEntity.ok().build();
    }
}
```

Interview Insights: This challenge tests your ability to:

1. Configure and use Spring's caching abstraction
2. Understand different caching strategies (@Cacheable, @CachePut, @CacheEvict)
3. Apply caching selectively to improve performance
4. Monitor and manage cache usage.