Cross-Origin Resource Sharing (CORS) Policy Enforcement in Spring Boot: Security Implications and Best Practices

Meerim Kakitaeva * and Mekia Shigute Gaso

Posted Date: 16 May 2025

doi: 10.20944/preprints202505.1312.v1

Keywords: CORS; spring boot; security; Web API; authentication; performance optimization; preflight requests

 doi:10.20944/preprints202505.1312.v1

Article

Cross-Origin Resource Sharing (CORS) Policy Enforcement in Spring Boot: Security Implications and Best Practices

Meerim Kakitaeva [1,*] and Mekia Shigute Gaso [2,]

[1] Student, Department of Computer Science, Ala-Too International University
[2] Senior Lecturer, Department of Computer Science, Ala-Too International University
* Correspondence: meerim.kakitaeva@alatoo.edu.kg

Abstract: Cross-Origin Resource Sharing (CORS) is an important function for securing cross-origin requests in web applications between server and client. A cross-origin request is when a web application sends an HTTP request to a different domain, protocol, or port than the one that hosted the original web page. Cross-origin requests typically occur when a client from one domain tries to access resources (such as APIs, images, or other data) across a different domain. Incorrect and broken CORS configurations could influence the security of the application. This work investigates CORS policy enforcement in Spring Boot applications focusing on security considerations and performance concerns. It clarifies common configuration mishaps, such as the embracement of all sources with credentials, and threats associated with them.This research also looks into the preflight OPTIONS request performance effect, especially in authentication-heavy contexts. In addition, it shows how misconfigurations may expose security weaknesses like cross-site request forgery (CSRF) and data exposure, and quantify the performance overhead of CORS checks. The outcome of the work gives advice for securing Spring Boot applications at minimal performance cost and demonstrates that cautious configuration to avoid security and performance bottlenecks is vital.

Keywords: CORS; spring boot; security; Web API; authentication; performance optimization; preflight requests

Classifier: UDC 004.056.55

1. Introduction

In modern web applications, Cross-Origin Resource Sharing (CORS) plays a prominent role in providing safe communication among clients and servers of various domains. With Single Page Applications (SPAs) and microservices architecture becoming widespread, it became increasingly necessary that cross-origin requests be properly dealt with. By default, the behavior of web browsers is to restrict requests for a domain unless explicitly allowed by the server, and that's where CORS policies come into action [1]. Spring Boot, among the leading Java-based microservice frameworks [2], provides numerous ways of configuring CORS policies to secure and control access to APIs. But poorly configured CORS policies have been found to create security vulnerabilities, unleashing sensitive information on malicious origins or allowing Cross-Site Request Forgery (CSRF). CORS misconfigurations [3], on the other hand, also lead to loss of performance resulting from the cost of preflight OPTIONS requests, especially for APIs requiring authentication. This paper explores the security and performance implications of CORS policy enforcement in Spring Boot applications, providing a comparative analysis of different configuration methods. We also provide guidelines for securely implementing CORS in a microservices-based application with minimal system performance overhead.

 doi:10.20944/preprints202505.1312.v1

2. Literature Review

CORS is a web standard which manages communications between web servers and browsers of different origins. MDN Web Docs describes that CORS is designed to render it impossible for a malicious website to access unauthorized information on another domain. In the past, developers had to implement custom headers and server-side configuration to facilitate cross-origin requests for a prolonged period, which was time-consuming as well as error-prone.

Within the Spring Boot environment, Spring Security has out-of-the-box mechanisms through which the CORS policies can be globally or controller-based configured. However, one must remember that misconfiguration continues to be a common root cause of security weaknesses. Vast amount of web applications are vulnerable to security weaknesses due to improperly designed CORS policy configurations [4]. For instance, allowing all origins with credentials enabled can lead to serious vulnerabilities such as unauthorized exposure of protected resources.

This essay hypothesizes that malformed CORS configurations in Spring Boot applications cause a colossal security vulnerability and an extra load on performance. Specifically, we conjecture that allowing broad origins with credentials or mucking up the preflight process in CORS introduces security vulnerabilities with subpar application performance, particularly microservices-based application schemes.

3. Methods

3.1. Experimental Setup

To see the impact of CORS configurations on security and performance, we set a Spring Boot-based microservices ecosystem with varying services communicating via HTTP/HTTPS. Two configurations were tested:

1. Basic CORS configuration with `allowedOrigins("*")` and `allowCredentials(true)`.
2. Controlled CORS configuration with specified allowed origins and restricted methods.

Both services were designed to serve both public and private endpoints, with private endpoints that require authentication with JWT tokens.

3.2. Preflight Requests and Performance Testing

We compared the latency of preflight-checked API requests and non-preflight-checked requests. The average latency increase caused by preflight checks was 30ms per request. The performance effect of such preflight requests was tested under various loads using Apache JMeter [5].

3.3. Security Testing

Penetration testing was conducted using OWASP ZAP to assess vulnerabilities such as CSRF and data leakage [6,7]. We also examined the impact of the incorrect configurations, such as allowing all origins with the credentials.

4. Technical Implementation

4.1. Spring Boot CORS Configuration

The implementation of CORS policies in Spring Boot applications is achieved by configuring the server to specify which origins and methods, and headers are allowed for cross-origin requests. Following is an overview of how we configured the CORS policies in our Spring Boot-based microservices.

4.1.1. Global CORS Configuration

Spring Boot provides ways to configure CORS both globally and per controller in multiple ways. In the existing implementation, we used global configuration so that all endpoints have the same CORS policy [8]. The configuration was executed in class `WebConfig` with the following code:

 doi:10.20944/preprints202505.1312.v1


```java
@Configuration
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/**")
                .allowedOrigins("https://trusted-domain.com", "https://another-trusted.com")
                .allowedMethods("GET", "POST", "PUT", "DELETE")
                .allowCredentials(true)
                .allowedHeaders("Authorization", "Content-Type")
                .maxAge(3600); // Cache the CORS preflight response for 1 hour
    }
}
```

4.1.2. Controller-Level CORS Configuration

For some endpoints that require different CORS policies, we can override the global setting at the controller level using the `@CrossOrigin` annotation. Below is an example of how to configure CORS for a particular controller:

```java
@RestController
public class SampleController {

    @CrossOrigin(origins = "https://trusted-domain.com", allowedHeaders = "Authorization")
    @GetMapping("/public-data")
    public ResponseEntity<String> getPublicData() {
        return ResponseEntity.ok("This␣is␣public␣data.");
    }

    @CrossOrigin(origins = "https://another-trusted.com", allowCredentials = true)
    @PostMapping("/private-data")
    public ResponseEntity<String> postPrivateData(@RequestBody Data data) {
        // process data
        return ResponseEntity.status(HttpStatus.CREATED).body("Private␣data␣stored␣successfully.");
    }
}
```

4.2. Preflight Requests Handling

CORS preflight requests are automatically processed by the browser in certain circumstances, i.e., for non-GET and non-POST methods or custom headers. In Spring Boot, these kinds of requests are automatically processed, but they can be overridden by using a filter. Below is an example of how to create a custom filter for logging preflight requests:

```java
@Component
public class PreflightFilter implements Filter {

    @Override
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
            throws IOException, ServletException {
        HttpServletRequest httpRequest = (HttpServletRequest) request;
        HttpServletResponse httpResponse = (HttpServletResponse) response;

        if ("OPTIONS".equalsIgnoreCase(httpRequest.getMethod())) {
            httpResponse.setStatus(HttpServletResponse.SC_OK);
            httpResponse.setHeader("Access-Control-Allow-Origin", "https://trusted-domain.com");
            httpResponse.setHeader("Access-Control-Allow-Methods", "GET,␣POST,␣PUT,␣DELETE");
            httpResponse.setHeader("Access-Control-Allow-Headers", "Authorization,␣Content-Type");
            httpResponse.setHeader("Access-Control-Allow-Credentials", "true");
            httpResponse.setHeader("Access-Control-Max-Age", "3600"); // Cache preflight response for 1
                ↪ hour
        } else {
```

 doi:10.20944/preprints202505.1312.v1


```
            chain.doFilter(request, response); // Proceed with the chain if not a preflight request
        }
    }

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {}

    @Override
    public void destroy() {}
}
```

4.3. Security Configuration with Spring Security

To secure API endpoints in Spring Boot, we utilize Spring Security to enforce authentication and authorization [9]. Here is how you can merge CORS with Spring Security:

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.cors()
            .and()
            .authorizeRequests()
            .antMatchers("/public-data").permitAll()
            .antMatchers("/private-data").authenticated()
            .and()
            .httpBasic(); // Basic authentication for simplicity
    }

    @Bean
    public CorsConfigurationSource corsConfigurationSource() {
        CorsConfiguration configuration = new CorsConfiguration();
        configuration.setAllowedOrigins(Arrays.asList("https://trusted-domain.com"));
        configuration.setAllowedMethods(Arrays.asList("GET", "POST", "PUT", "DELETE"));
        configuration.setAllowCredentials(true);
        configuration.setAllowedHeaders(Arrays.asList("Authorization", "Content-Type"));

        UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
        source.registerCorsConfiguration("/**", configuration);
        return source;
    }
}
```

5. Results

5.1. Security Analysis

Our results show that while CORS needs to be used to protect cross-origin requests, misconfigurations can also yield security flaws and performance degradation. Misconfiguration (i.e., using `allowedOrigins("*")` with `allowCredentials(true)`) can bring about serious security issues, like unauthorized access and CSRF attacks. Preflight requests impose additional performance overhead, with perceptible performance degradation in latency, especially in high-traffic systems.

5.2. Performance Analysis

We benchmarked the latency of API calls with and without preflight checks [10]. The average latency increase due to preflight checks was 30ms per request. This may be inconsequential in low-traffic APIs but became larger in high-traffic systems, particularly those that are authenticated.

 doi:10.20944/preprints202505.1312.v1

6. Conclusions

In conclusion, CORS policy enforcement in Spring Boot applications is crucial to providing security in modern web models. However, improper configuration can result in exposing applications to security vulnerabilities and performance issues. Proper configuration and best practices implementation of CORS policies can mitigate these risks. Additional optimization techniques and best practices for fine-tuning CORS behavior in microservices-based systems are what future studies should focus on.

References

1. Mozilla. (n.d.). *Cross-origin resource sharing (CORS)*. MDN Web Docs. https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS
2. Spring Team. (n.d.). *Spring Boot reference documentation*. Spring. https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/
3. Djaber, Y. (2023). *Understanding and preventing CORS misconfiguration*. Vaadata Blog. https://www.vaadata.com/blog/understanding-and-preventing-cors-misconfiguration/
4. OWASP. (n.d.). *CORS origin header scrutiny*. OWASP. https://owasp.org/www-community/attacks/CORS_OriginHeaderScrutiny
5. Apache. (n.d.). *Apache JMeter user's manual*. https://jmeter.apache.org/usermanual/index.html
6. PortSwigger. (n.d.). *Cross-site request forgery (CSRF)*. PortSwigger Web Security Academy. https://portswigger.net/web-security/csrf
7. Pixel QA. (2023). *How to perform pen testing with OWASP ZAP*. https://www.pixelqa.com/blog/post/how-to-perform-pen-testing-with-owasp-zap
8. Stack Overflow. (2016, May 1). *How to configure CORS in a Spring Boot Spring Security application*. https://stackoverflow.com/questions/36968963/how-to-configure-cors-in-a-spring-boot-spring-security-application
9. Spring. (n.d.). *Spring Security*. https://spring.io/projects/spring-security
10. Mozilla. (n.d.). *Preflight request*. MDN Web Docs. https://developer.mozilla.org/en-US/docs/Glossary/Preflight_request