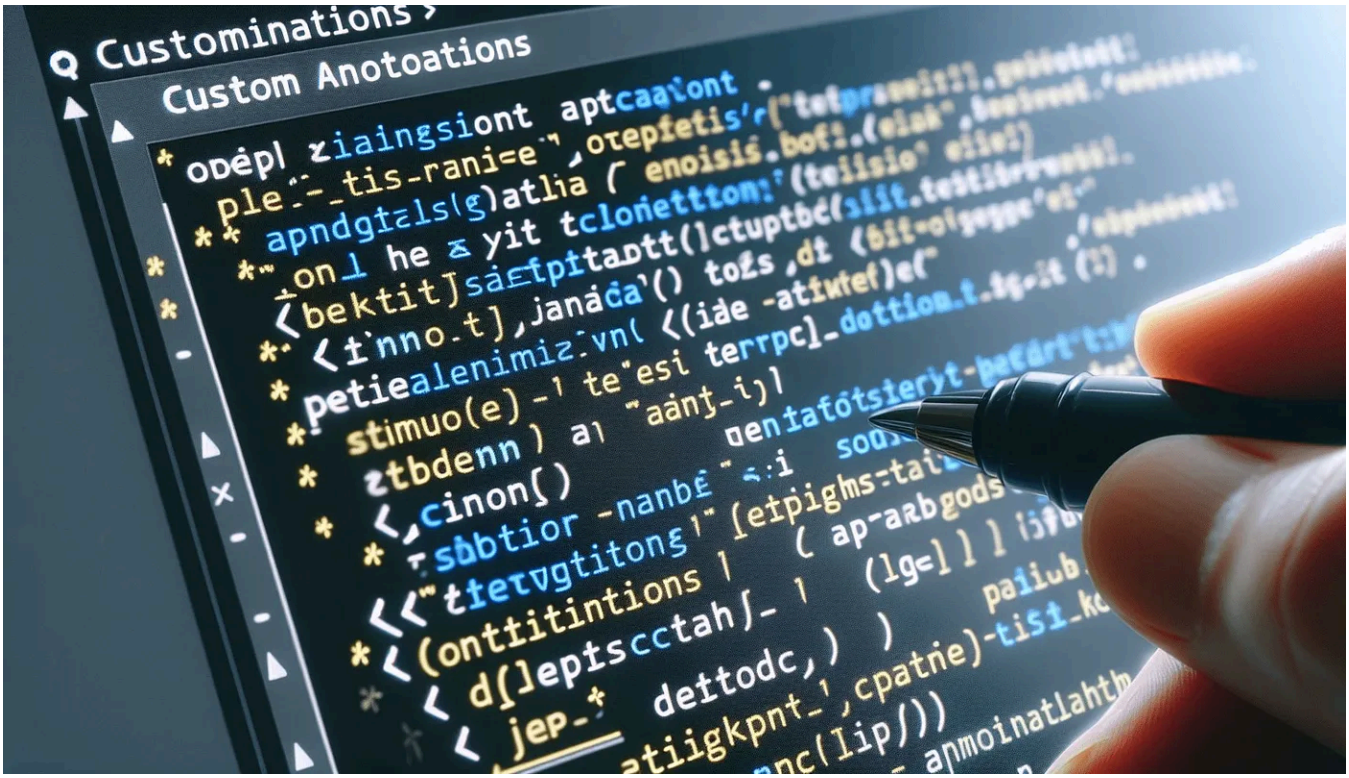
Beyond Basics: Creating Advanced Custom Annotations in Spring



Annotations in Java serve as powerful metadata markers, enriching your code with additional information. In our previous article, "Mastering Annotations: From Basics to Advanced Techniques" we explored built-in and custom annotations. Now, let's dive deeper into creating custom annotations within the Spring framework.

In this article, we'll create a custom annotation called `@RoleCheck` that allows us to control access to different controller endpoints based on user roles. Whether you're a seasoned Spring developer or just starting your journey, understanding custom annotations is essential for building robust and secure applications.

Prerequisites

Before we proceed, make sure you have a basic understanding of Spring Boot, Spring MVC, and annotations.

Implementation Steps

Let's create our `@RoleCheck` annotation step by step:

0. Create a new spring boot project

If you want to follow along, create a new Spring boot project and let's get to coding 😊 .

1. Define the Annotation

The `@RoleCheck` annotation will help us enforce role-based access control (RBAC) for our Spring endpoints. By applying this annotation to specific controller methods, we can ensure that only authorized users with the correct roles can access those endpoints.

```
@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface RoleCheck {
    String role() default "USER";
}
```

2. Create RoleCheckInterceptor

HandlerInterceptor is an interface in Spring MVC that allows pre-processing and post-processing of requests. We will create RoleCheckInterceptor for intercepting the requests and validate if the request is being invoked with the required privileges.

```
public class RoleCheckInterceptor implements HandlerInterceptor {

    @Override
    public boolean preHandle(HttpServletRequest request,
                             HttpServletResponse response, Object handler)
throws Exception {
        RoleCheck roleCheck = null;
        String userRole = "USER";//Fetch user role based on your application con
        if (handler instanceof HandlerMethod) roleCheck =
                ((HandlerMethod) handler)
                .getMethodAnnotation(RoleCheck.class);
        if (roleCheck != null) {
            String role = roleCheck.role();
```

```
            if (userRole.equals("ADMIN")) {
                // Admin should be able to access all the resources
                return true;
            } else {
                boolean isAuthorized = role.equalsIgnoreCase(userRole);
                if (!isAuthorized) {
                    response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
                    response.getWriter().write("You are not authorized to
access this resource");
                }
                return true;
            }
        }
        return true;
    }
}
```

3. Add the interceptor to WebMvcConfigurer

We will override the default configurations provided by Spring MVC and will add the interceptor that we have created above.

```
@Component
public class MvcConfiguration implements WebMvcConfigurer {
    private List<HandlerInterceptor> interceptors;
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        if (CollectionUtils.isEmpty(interceptors)) interceptors =
new ArrayList<>();
        interceptors.add(new RoleCheckHandler());
        for (HandlerInterceptor interceptor : interceptors) {
            registry.addInterceptor(interceptor);
        }
    }
}
```

4. Adding the annotation in controller

We will add the annotation in controller. For simplicity I've created the below controller and added @RoleCheck annotation with the expected role to couple of endpoints.

```
@RestController
@RequestMapping("/account")
public class AccountManager {
    @RoleCheck(role = "ADMIN")
    @GetMapping("/add")
    public void addAccount() {
        // Logic to add account

        ResponseEntity.ok("Account added successfully");
    }
    @RoleCheck(role = "USER")
    @GetMapping("/view")
    public void viewAccount() {
        // Logic to view account
        ResponseEntity.ok("Account viewed successfully");
    }
}
```

In the above example, only ADMINs can add the account and a normal USER can only view the account. Also if USER tries to add the account, they will get an 401 (Unauthorized) response.

Conclusion

Custom annotations like `@RoleCheck` allow us to express domain-specific rules declaratively. By integrating them we enhance our application's security and maintainability.
Also the above example just shows a simple way to create and use custom annotations. It's no where near a production ready code, so please do your due diligence before adding it to your code directly.

Remember, annotations are more than mere comments — they shape your code's behavior and contribute to a well-structured, maintainable codebase.

Extend your knowledge from our previous article and start using custom annotations effectively in your Spring applications. Happy coding!

Thank you for joining us on this exploration of custom annotations in Spring! 🌱 .

If you found this article helpful or insightful, consider following us for more exciting Java and Spring-related content.

Let's continue learning and building together! 😊🚀