Reflektion Java Web Services

Individuell laboration

När jag började arbeta med min individuella laboration insåg jag att säkerhetsdelen innebar många steg och flera viktiga avvägningar att göra.

Jag funderade över för- och nackdelar med att hantera säkerhet via SecurityFilterChain jämfört med annoteringar som @PreAuthorize. Efter viss efterforskning valde jag att använda @PreAuthorize, eftersom det gör koden mer lättläst och överskådlig – behörighetsreglerna är direkt kopplade till respektive metod, vilket ökar tydligheten.

Jag valde också att aktivera Keycloaks inloggningsfönster i stället för att hantera inloggning manuellt via Postman. Detta minskar risken för att känslig information, som användarnamn och lösenord, skulle exponeras i klartext – något som är extra viktigt i verkliga system.

Jag reflekterade även över hur jag skulle hantera kontrollen av ägarskap samt om användaren är admin. Skulle jag göra kontrollen i BlogPostController eller i BlogPostServiceImpl?

Utifrån ett säkerhetsperspektiv skyddas endpointsen redan av @PreAuthorize, vilket innebär att endast användare med rätt roll får åtkomst.

För att hålla controllerlagret så enkelt som möjligt valde jag att lägga logiken i servicelagret. Jag tar därför in ett Jwt- eller Authentication- objekt i de endpoints där denna kontroll ska göras, och skickar vidare objektet till serviceklassen. Där använder jag en gemensam hjälpmetod för att kontrollera om det är den ägande användaren eller admin som försöker uppdatera eller ta bort en post.

Detta är ett försök till att göra koden enkel att testa, med så få beroenden som möjligt. Genom att endast behöva mocka Jwt eller Authentication i testerna – i stället för att manipulera hela SecurityContextHolder – kan testerna skrivas isolerat, utan global kontext.

För att ytterligare öka testbarheten och separationen av ansvar har jag valt att skapa en util-klass med statiska metoder för att hämta användarinformation. Där skickar jag in ett Jwt- eller Authentication- objekt och får ut sådant som roll, användarnamn och sub.

Detta underlättar både testning och återanvändning av logik, samtidigt som det minskar beroenden i serviceklassen.

För att separera databaskopplade entiteter från det som skickas till och från klienten använde jag DTO:er. På så sätt undviker jag att exponera intern struktur eller känsliga fält, så som id, userID och username, av misstag. Det gör applikationen mer flexibel vid

förändringar och ökar säkerheten genom att endast relevant data skickas in och ut – något som är extra viktigt i verkliga system.

Avslutningsvis använder ResponseStatusException för att explicit returnera tydliga och korrekta HTTP-statuskoder, t.ex. 403 Forbidden när en användare försöker uppdatera en post de inte äger. Detta ger tydlig återkoppling till klienten, förbättrar API:ets användbarhet och förenklar felsökning vid fel.