Spring boot - Part9 (Role based Authorization - Annotations)

Tuesday, 22 April 2025 5:25 PM

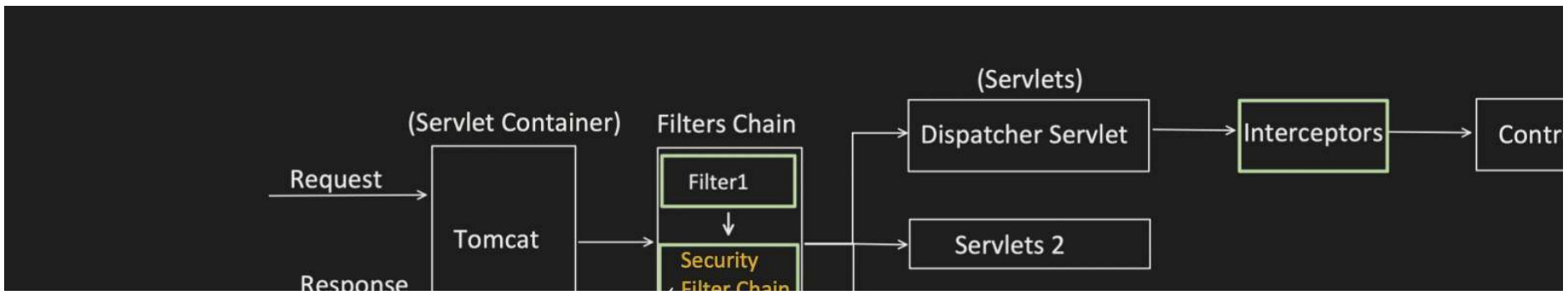
Role based Authorization - (Annotation)

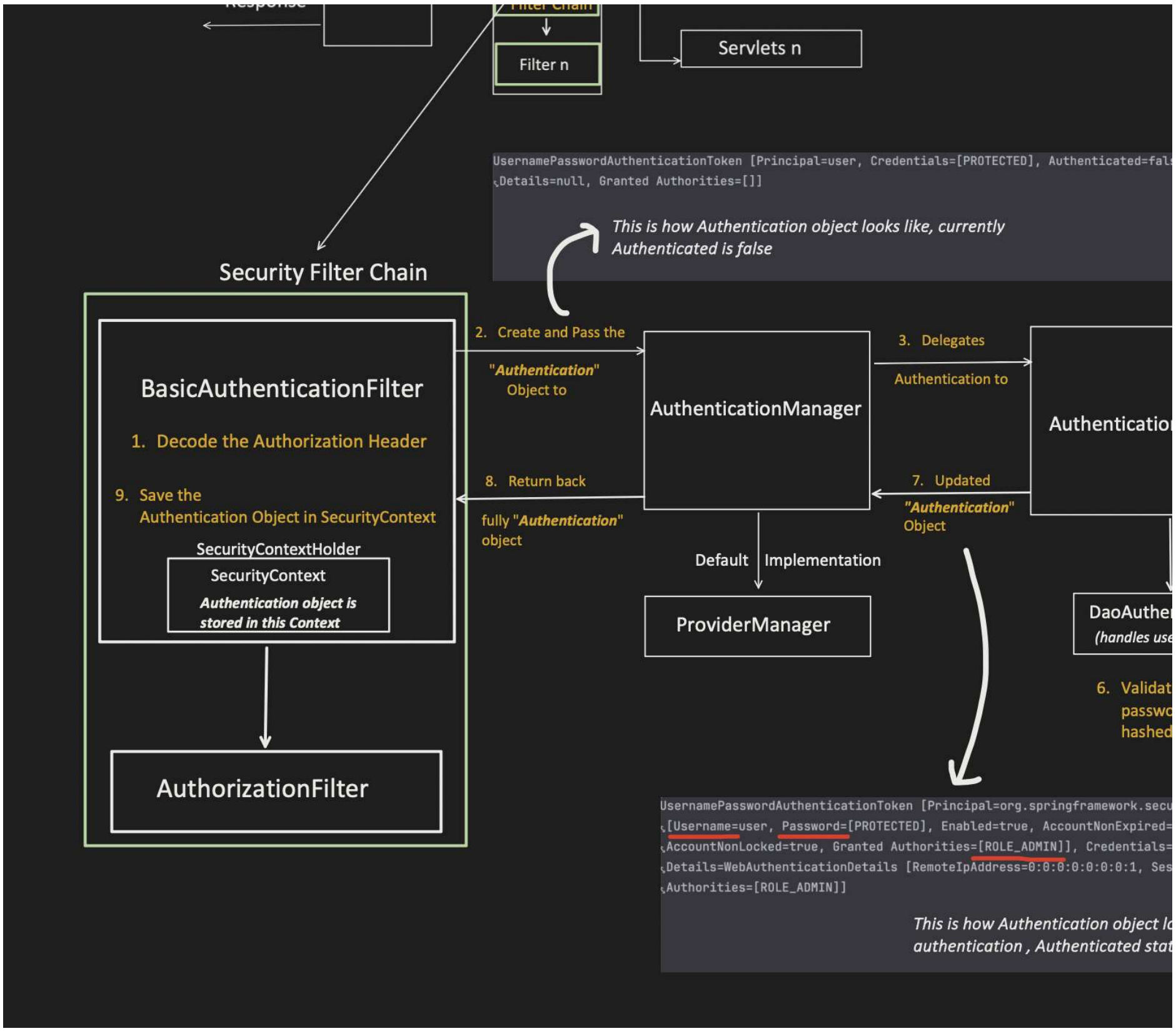
- We have already covered different type of Authentications.
- And with that, we have also covered, how at Security Filter layer itself we can do authori

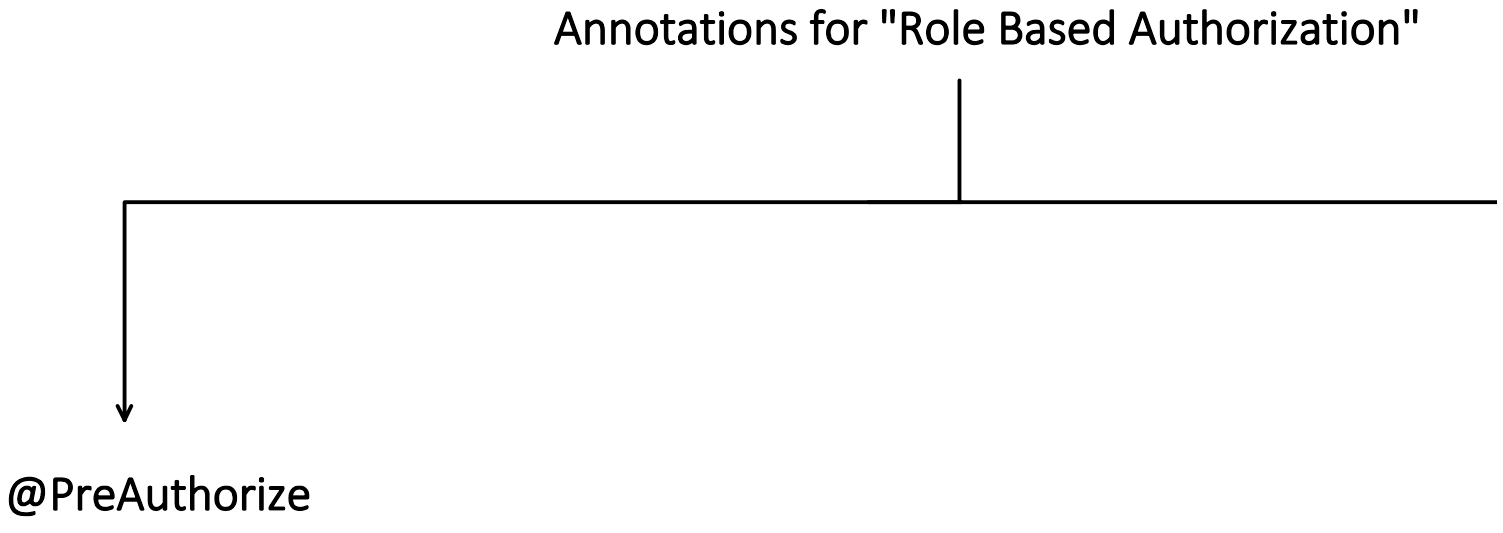
```java
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http)
        http
                .authorizeHttpRequests(auth -> auth
                        .requestMatchers( ...patterns: "/users").hasRole("U
                        .anyRequest().authenticated()
                )
                .formLogin(Customizer.withDefaults());
        return http.build();
    }
}
```

- But in large scale applications, where we might have 100s of APIs, in that scenarios m become difficult and cause scalable issue.

  That's where, Annotation based "Role Based Authorization" comes into the picture. A Controller class.




Filter n
Servlets n
UsernamePasswordAuthenticationToken [Principal=user, Credentials=[PROTECTED], Authenticated=fals
Details=null, Granted Authorities=[]]
This is how Authentication object looks like, currently Authenticated is false
Security Filter Chain
BasicAuthenticationFilter
1. Decode the Authorization Header
9. Save the Authentication Object in SecurityContext
SecurityContextHolder
SecurityContext
Authentication object is stored in this Context
AuthorizationFilter
2. Create and Pass the "Authentication" Object to
AuthenticationManager
3. Delegates Authentication to
Authentication
8. Return back fully "Authentication" object
7. Updated "Authentication" Object
Default Implementation
ProviderManager
DaoAuthe
(handles use
6. Validat
passwo
hashed
UsernamePasswordAuthenticationToken [Principal=org.springframework.secu
[Username=user, Password=[PROTECTED], Enabled=true, AccountNonExpired=
AccountNonLocked=true, Granted Authorities=[ROLE_ADMIN]], Credentials=
Details=WebAuthenticationDetails [RemoteIpAddress=0:0:0:0:0:0:0:1, Ses
Authorities=[ROLE_ADMIN]]
This is how Authentication object lo
authentication , Authenticated stat


Annotations for "Role Based Authorization"
@PreAuthorize

Does Authorization, before execution of the API.

Lets create User First (Dynamic one)

We have already seen its implementation in


```java
@Entity
@Table(name = "user_login")
public class UserLoginEntity implements UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(unique = true, nullable = false)
    private String username;

    @Column(nullable = false)
    private String password;

    private String role; // e.g., {"ROLE_USER" or "ROLE_ADMIN"

    @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
    private List<UserPermissionEntity> permissions = new ArrayList<>();

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        Set<GrantedAuthority> authorities = new HashSet<>();

        authorities.add(new SimpleGrantedAuthority(role));

        permissions.forEach(permission ->
```

```
            authorities.add(new SimpleGrantedAuthority(permission.getName())));

        return authorities;
    }

    @Override
    public String getPassword() {
        return password;
    }

    @Override
    public String getUsername() {
        return username;
    }
```

```
@RestController
public class UserLoginController {

    @Autowired
    UserLoginEntityService userLoginEntityService;

    @Autowired
    PasswordEncoder passwordEncoder;

    @PostMapping("/user-login")
    public ResponseEntity<String> login(@RequestBody UserLoginEntity userLoginEntity) {
        // Hash the password before saving
        userLoginEntity.setPassword(passwordEncoder.encode(userLoginEntity.getPassword()

        userLoginEntityService.save(userLoginEntity);

        return ResponseEntity.ok( body: "User registered successfully!");
    }
}
```

Now, we should be able to create user, and in this one User can have 1 role like ROLE_AI
But we can give many permissions like ORDER_READ, SALES_DELETE etc.... (more granul

POST localhost:8080/user-login

Params Authorization Headers (9) Body Scripts S

none form-data x-www-form-urlencoded raw

```
{
  "username": "sj",
  "password": "123",
  "role": "ROLE_USER",
  "permissions": [
    {
        "name" : "ORDER_READ"
    }
  ]
}
```

Now, User Creation part is done, lets update our Security C

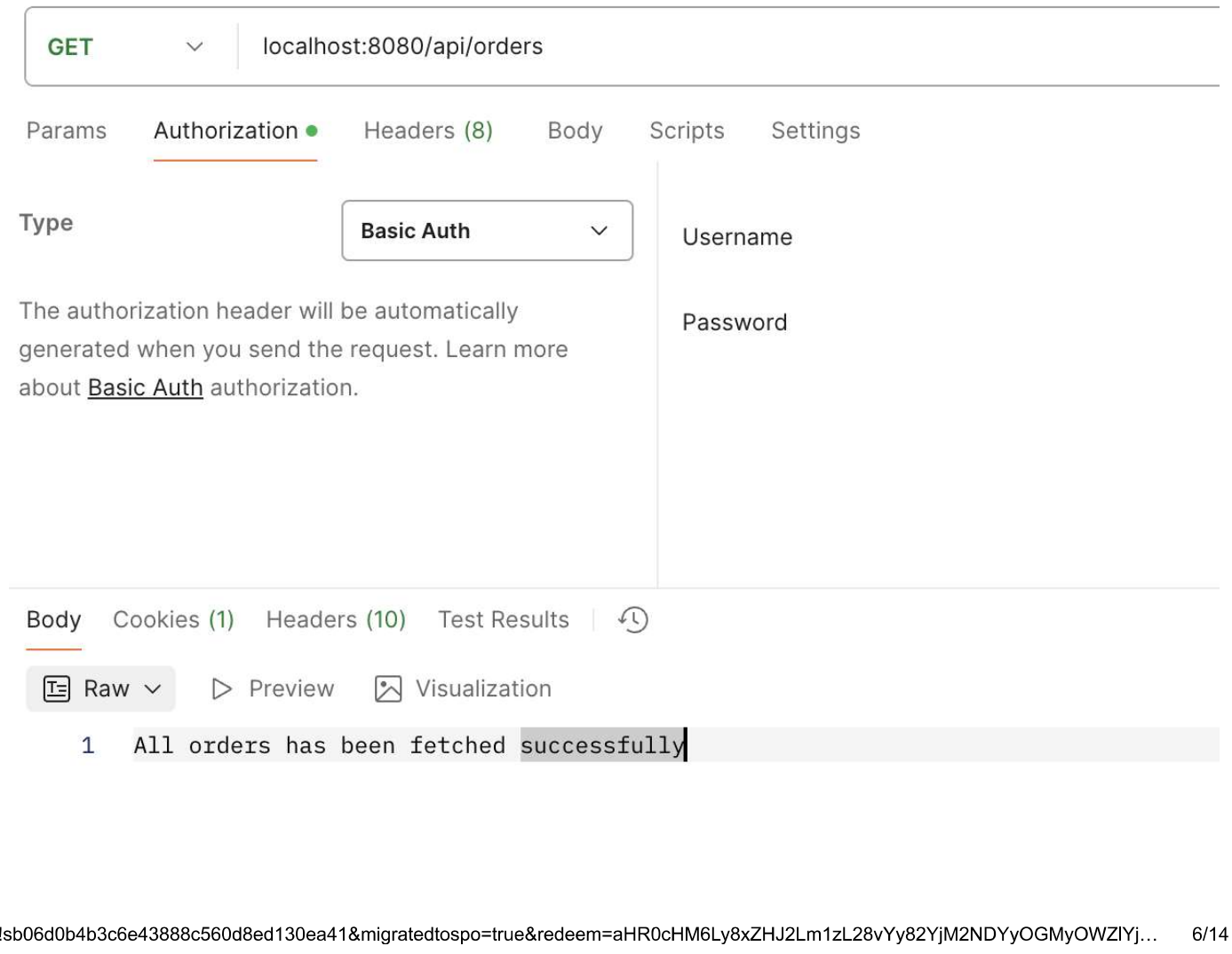
```
@Configuration
@EnableWebSecurity
@EnableMethodSecurity(prePostEnabled = true)
public class SecurityConfig{

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

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecur

        http.authorizeHttpRequests(auth -> auth
                        .requestMatchers( ...patterns: "/user-lo
                        .anyRequest().authenticated())
                .sessionManagement(session -> session
                        .sessionCreationPolicy(SessionCreati
                .csrf(csrf -> csrf.disable())
                .httpBasic(Customizer.withDefaults());
```

```
        return http.build();
    }
}
```

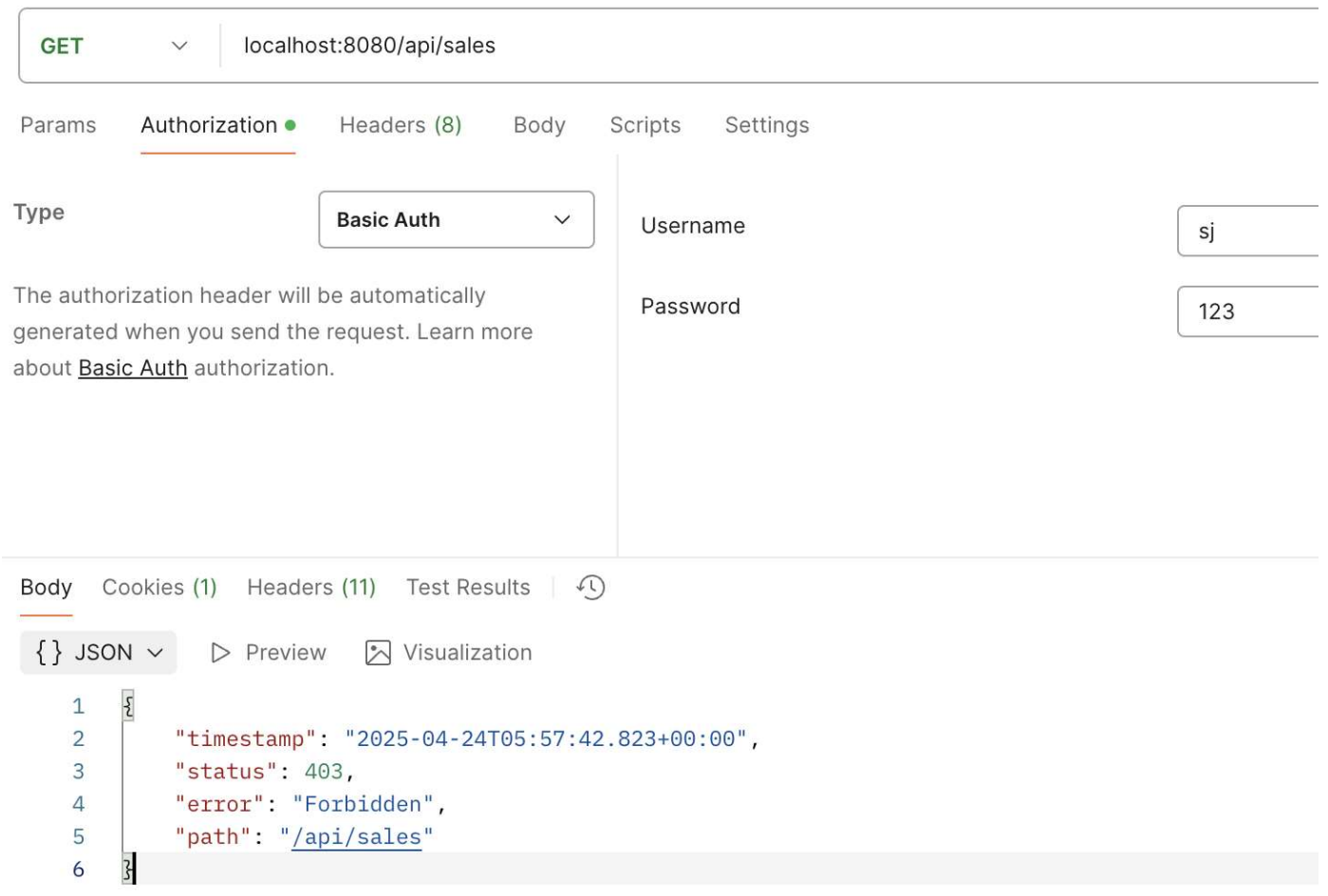
Now, lets create 2 Controller class, one for ORDER and another f

```
@RestController
@RequestMapping("/api")
public class OrderController {

    @GetMapping("/orders")
    @PreAuthorize("hasRole('USER') and hasAuthority('ORDER_READ')")
    public ResponseEntity<String> readOrders() {
        return ResponseEntity.ok( body: "All orders has been fetched successfully");
    }
}
```

Above we have created user, who has both the permissions ROLE_USER ar successfully executed.



Now, when we try to access /sales API, which above user do not have permi



@PreAuthorize("hasRole('USER') and hasAuthority(

Spring Boot, treat both "hasRole" and "hasAuthority" in a similar way, only diffe

For "hasRole" : "ROLE_" is appended

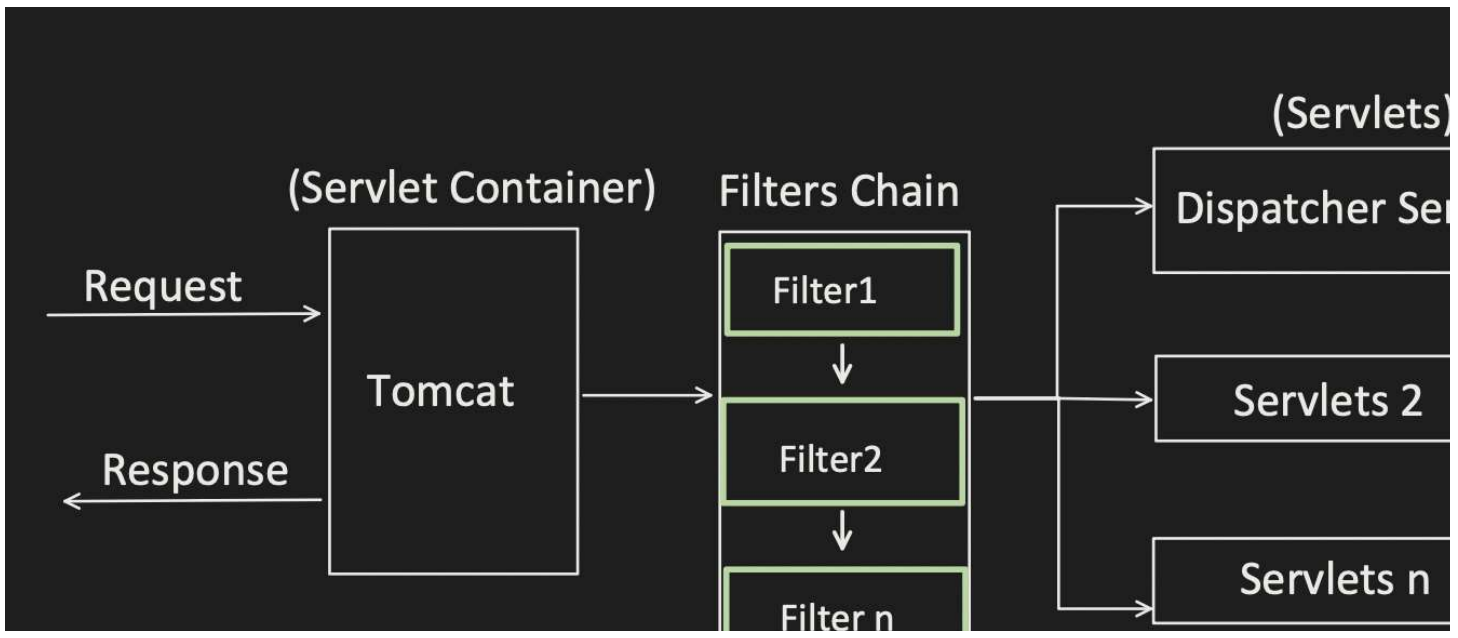
SecurityExpressionRoot.java

```
private String defaultRolePrefix = "ROLE_";

@Override
public final boolean hasAuthority(String authority) {
```

```
    return hasAnyAuthority(authority);
}

@Override
public final boolean hasAnyAuthority(String... authorities)
    return hasAnyAuthorityName( prefix: null, authorities);
}

@Override
public final boolean hasRole(String role) {
    return hasAnyRole(role);
}

@Override
public final boolean hasAnyRole(String... roles) {
    return hasAnyAuthorityName(this.defaultRolePrefix, role
}

private boolean hasAnyAuthorityName(String prefix, String..
    Set<String> roleSet = getAuthoritySet();
    for (String role : roles) {
        String defaultedRole = getRoleWithDefaultPrefix(pre
        if (roleSet.contains(defaultedRole)) {
            return true;
        }
    }
    return false;
}
```

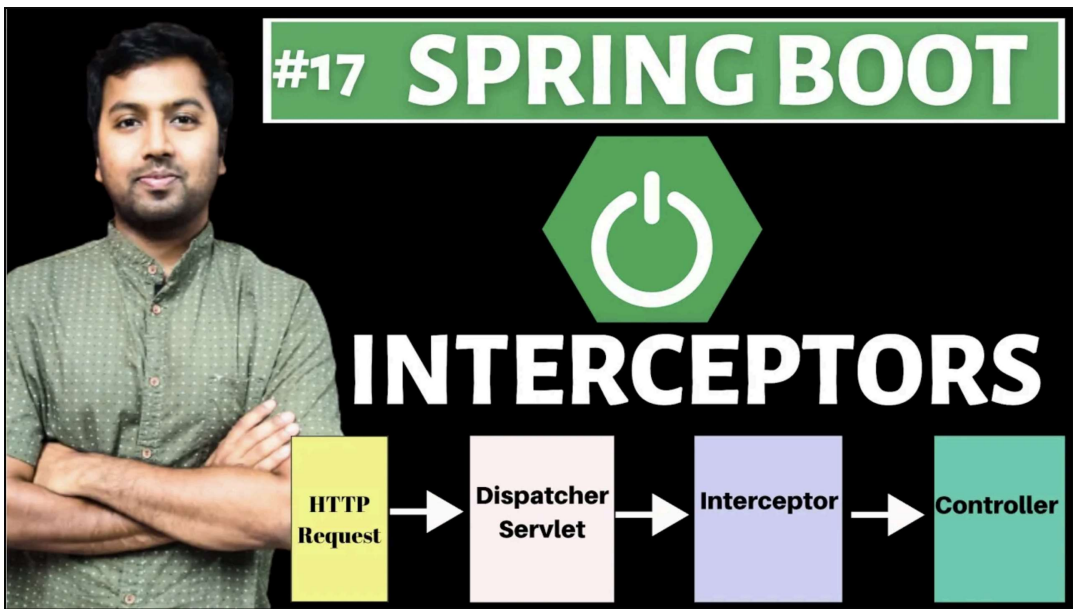
Now, one question comes to mind is, how this Authorization methods invo
Controller method.

Its because of INTERCEPTORS

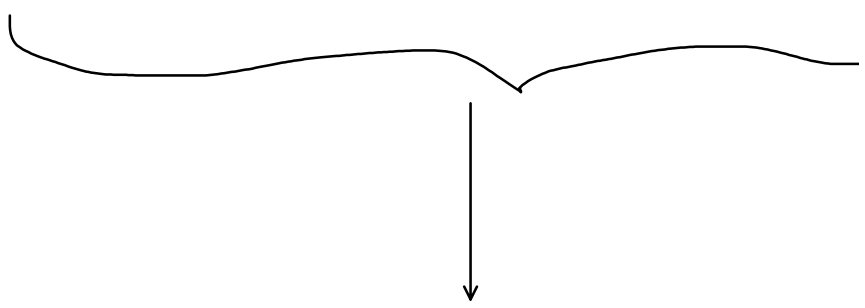


@PreAuthorize Annotation is intercepted by ***AuthorizationManagerBeforeMeth***

What and how Interceptors works and how they are different from filters?
We have already covered both the topics in depth.



@PreAuthorize("hasRole('USER') and hasAuthority('ORDER_



This expression uses SpEL (i.e. Spring Expres

Spring framework has *SpelExpressionParser* class, which help i

Below is the high level flo

AuthorizationManagerBeforeMethod Interceptor

1. Reads String from @PreAuthroize annotation → @PreA

2. Parse it using SpelExpressionParser → **Conve Synta**

3. Resolves the above Expression → • **Ref Wh**

```
@Override
public TypedValue getValueInternal(Expre
    if (!getBooleanValue(state, getLeftO
        // no need to evaluate right ope
        return BooleanTypedValue.FALSE;
    }
    return BooleanTypedValue.forValue(ge
}
```

4. → Validat or not

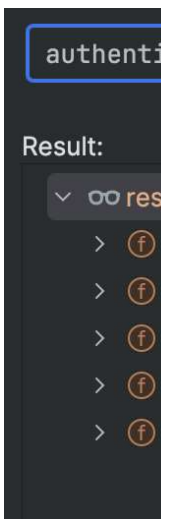
More Logical Operator

Operator	Description	
and	Logical AND	hasRole('AD
or	Logical OR	hasRole(
not	Logical NOT	r
!	Also logical NOT	!h

Relational Operators

Operator	Description	
==	Equal	
!=	Not equal	
<	Less than	
>	Greater than	
<=	Less than or equal	
>=	Greater than or equal	

```
@PreAuthorize("#id == authentication.principal.id")
@GetMapping("/users/{id}")
public User fetchUserDetails(@PathVariable Long id) {
   return userServiceObject.fetchUserDetails(id);
}
```



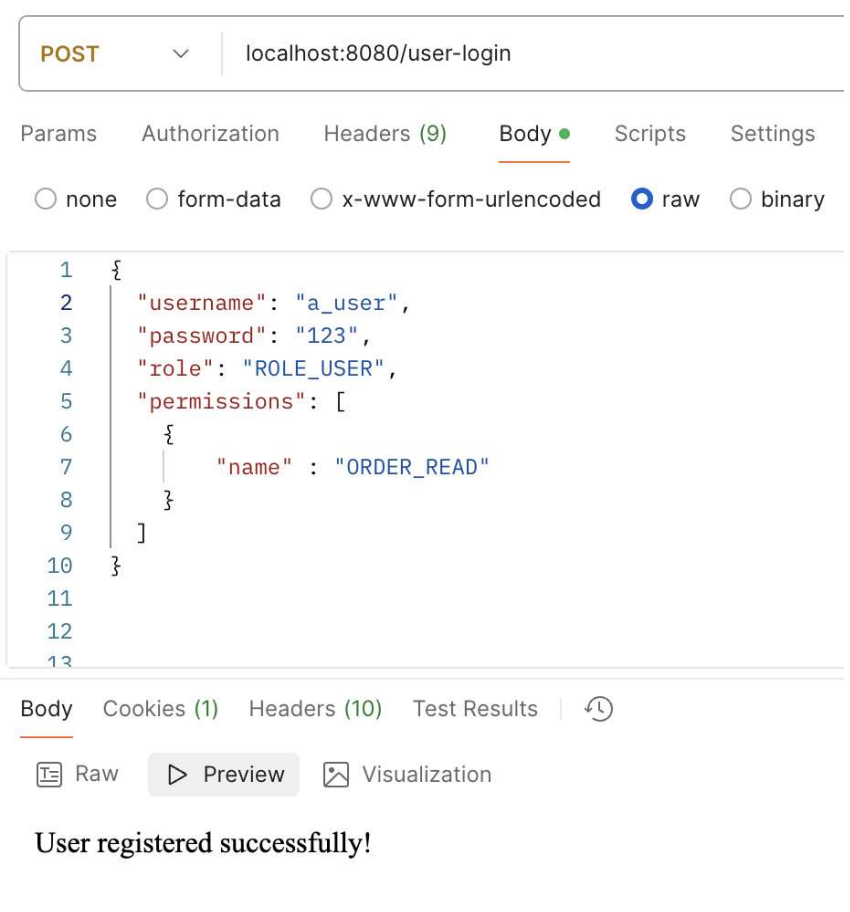
@PostAuthorize

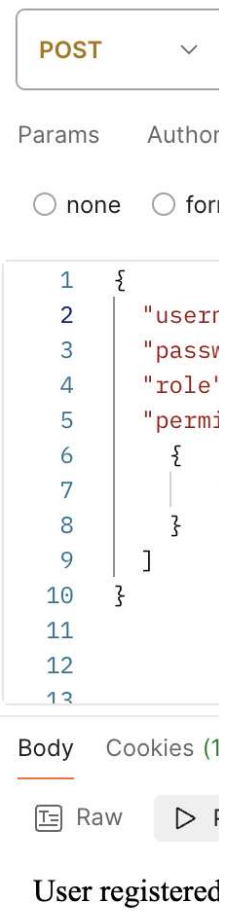
- Does Authorization after execution of the API but before send

- AuthorizationManagerAfterMethodInterceptor, is the one w annotation.

Let's see with an example

Created 2 Users





SELECT * FROM USER_LOGIN

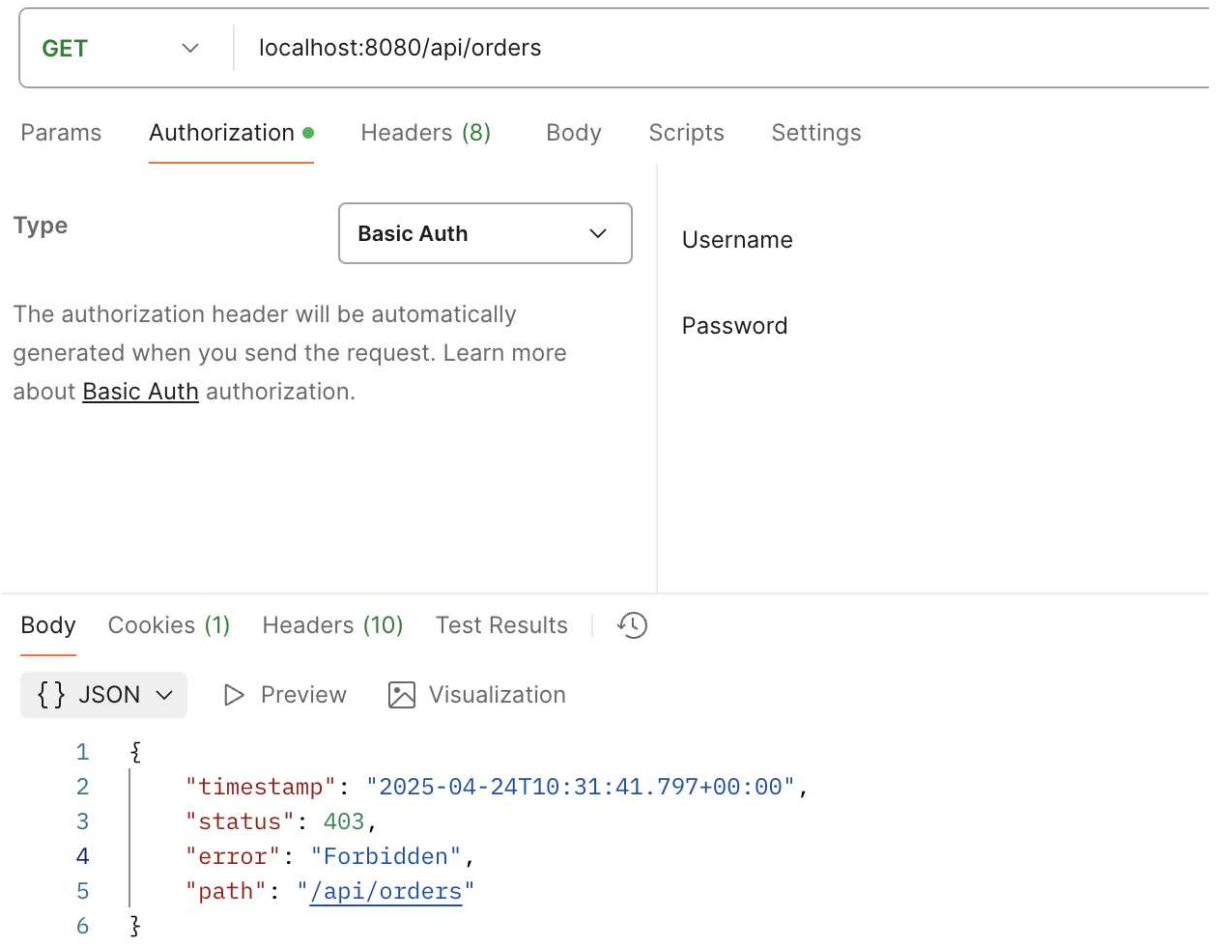
SELECT * FROM USER_LOGIN;

ID	PASSWORD	R
1	$2a$10$oLOsKKtsbEsC63.MmGuNbO2CYqN8kvcCZW8/fwMvBB0p6mtKnQZ5u	R
2	$2a$10$Eb/CtlSeC0uUpTd09iyTUOcXDjdKTNiX.9C6CjZj2zZ9L2Ol0YW.m	R

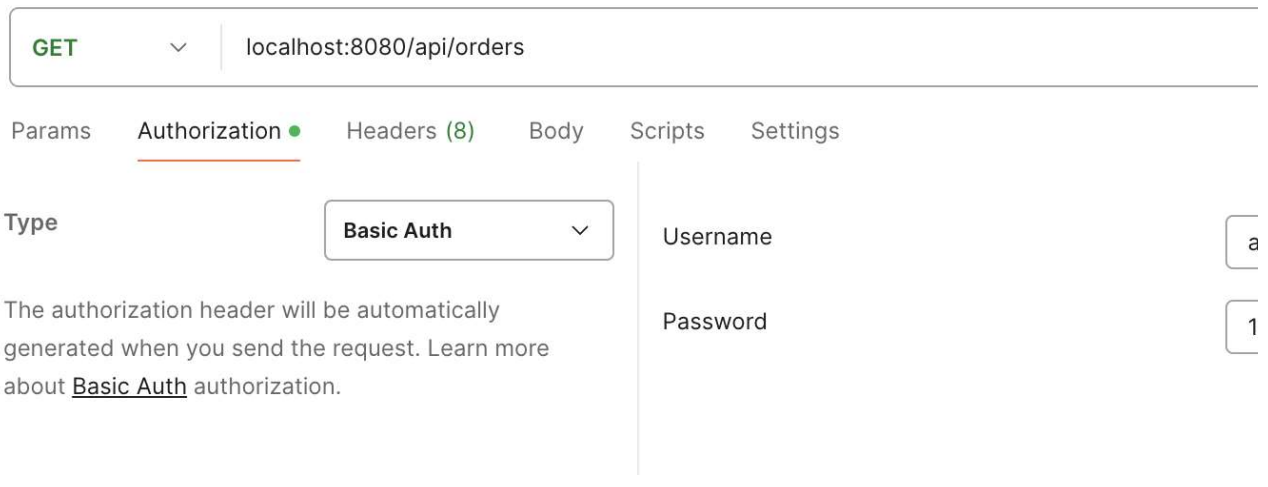
(2 rows, 6 ms)

```
    public OrderDTO readOrders() {
        OrderDTO orderDTO = new OrderDTO();
        orderDTO.userID = 1l; //hardcoding for no
        orderDTO.orderID = 10000l;
        return orderDTO;
    }
}
```

Now, try to invoke this api with "b_users" credentia



Now, try to invoke this api with "a_users" credentials



Body Cookies (1) Headers (10) Test Results

{} JSON Preview Visualization

```
1  {
2      "userID": 1,
3      "orderID": 10000
4  }
```