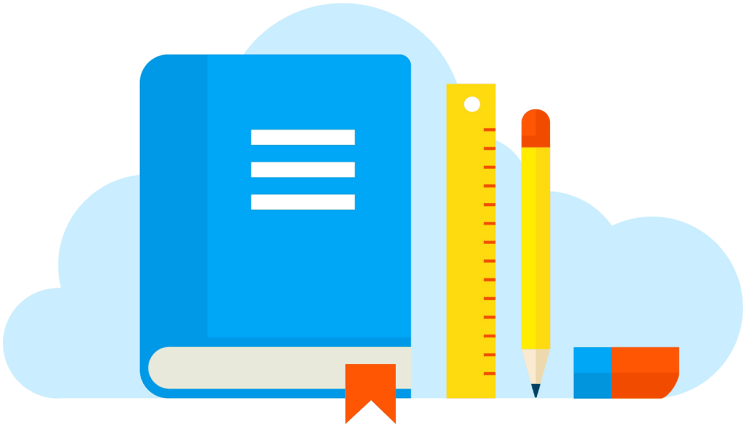
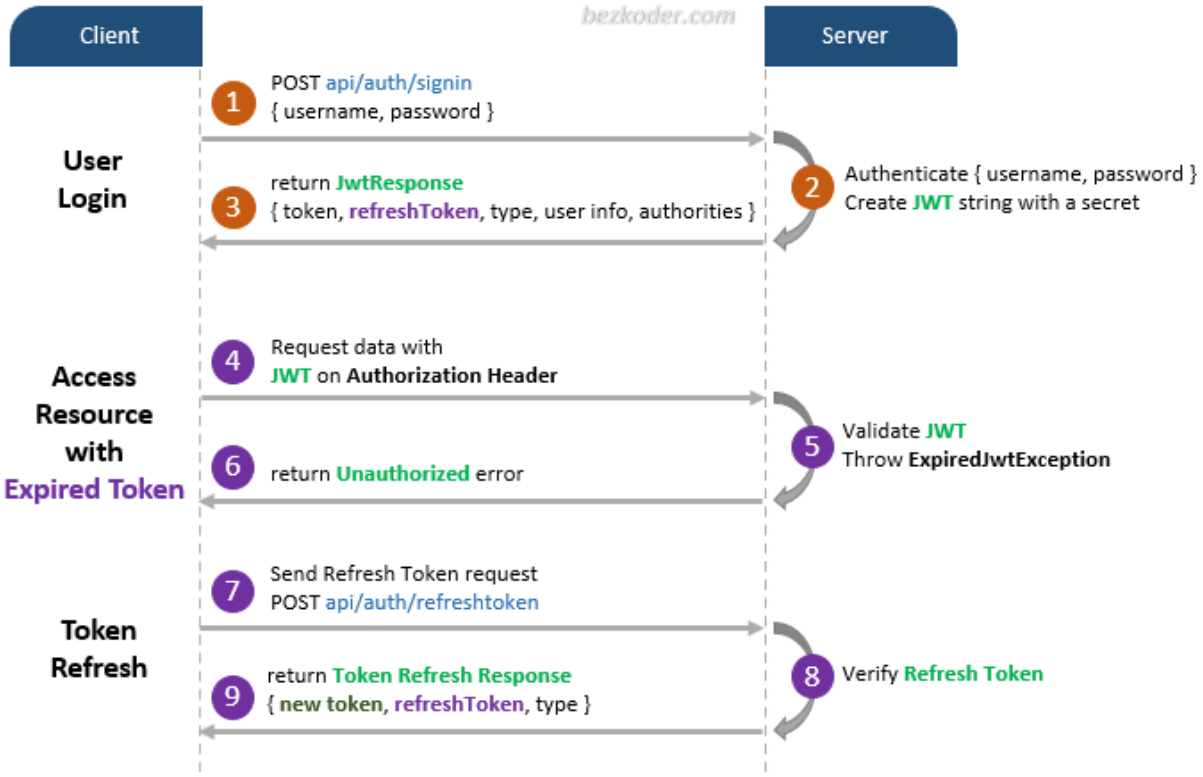


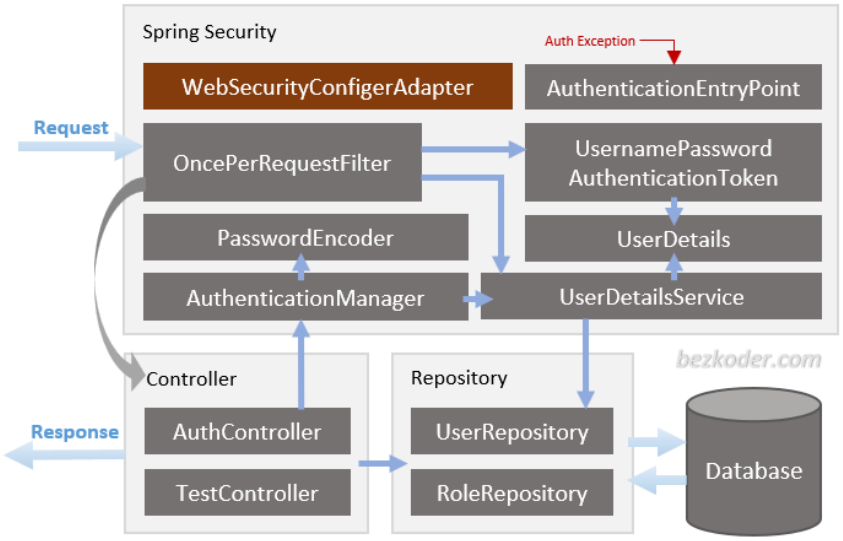
Springboot Security

- Spring Boot Signup & Login with JWT Authentication Flow
- Spring Boot Server Architecture with Spring Security Technology
- Project Structure
- Setup new Spring Boot project
- Configure Spring Datasource, JPA, App properties
- Create the models
- Implement Repositories
- Configure Spring Security
- Implement UserDetails & UserDetailsService
- Filter the Requests
- Create JWT Utility class
- Handle Authentication Exception
- Define payloads for Spring RestController
- Create Spring RestAPIs Controllers
- Run & Test
- Conclusion

1.Spring Boot Signup & Login with JWT Authentication Flow



2. Spring Boot Server Architecture with Spring Security



Spring Security

– `WebSecurityConfigurerAdapter` is the crux of our security implementation. It provides `HttpSecurity` configurations to configure cors, csrf, session management, rules for protected resources. We can also extend and customize the default configuration that contains the elements below.

– `UserDetailsService` interface has a method to load User by *username* and returns a `UserDetails` object that Spring Security can use for authentication and validation.

– `UserDetails` contains necessary information (such as: username, password, authorities) to build an Authentication object.

– `UsernamePasswordAuthenticationToken` gets {username, password} from login Request, `AuthenticationManager` will use it to authenticate a login account.

– `AuthenticationManager` has a `DaoAuthenticationProvider` (with help of `UserDetailsService` & `PasswordEncoder`) to validate `UsernamePasswordAuthenticationToken` object. If successful, `AuthenticationManager` returns a fully populated Authentication object (including granted authorities).

– `OncePerRequestFilter` makes a single execution for each request to our API. It provides a `doFilterInternal()` method that we will implement parsing & validating JWT, loading User details (using `UserDetailsService`), checking Authorizaion (using `UsernamePasswordAuthenticationToken`).

– `AuthenticationEntryPoint` will catch authentication error.

Repository contains `UserRepository` & `RoleRepository` to work with Database, will be imported into **Controller**.

Controller receives and handles request after it was filtered by `OncePerRequestFilter`.

– `AuthController` handles signup/login requests

– `TestController` has accessing protected resource methods with role based validations.

	Authentication	Authorization
1.	In authentication process, the identity of users are checked for providing the access to the system.	While in authorization process, person's or user's authorities are checked for accessing the resources.
2.	In authentication process, users or persons are verified.	While in this process, users or persons are validated.
3.	It is done before the authorization process.	While this process is done after the authentication process.
4.	It needs usually user's login details.	While it needs user's privilege or security levels.
5.	Authentication determines whether the person is user or not.	While it determines What permission do user have?

3. Technology

- Java 8
- Spring Boot 2.6.1 (with Spring Security, Spring Web, Spring Data JPA)
- jjwt 0.9.1
- PostgreSQL/MySQL
- Maven 3.6.1

4.Project Structure

security: we configure Spring Security & implement Security Objects here.

- `WebSecurityConfig` extends `WebSecurityConfigurerAdapter`
- `UserDetailsServiceImpl` implements `UserDetailsService`
- `UserDetailsImpl` implements `UserDetails`
- `AuthEntryPointJwt` implements `AuthenticationEntryPoint`
- `AuthTokenFilter` extends `OncePerRequestFilter`
- `JwtUtils` provides methods for generating, parsing, validating JWT

controllers handle signup/login requests & authorized requests.

- `AuthController`: @PostMapping('/signin'), @PostMapping('/signup')
- `TestController`: @GetMapping('/api/test/all'), @GetMapping('/api/test/[role]')

repository has intefaces that extend Spring Data JPA `JpaRepository` to interact with Database.

- `UserRepository` extends `JpaRepository<User, Long>`
- `RoleRepository` extends `JpaRepository<Role, Long>`

models defines two main models for Authentication (`User`) & Authorization (`Role`). They have many-to-many relationship.

- `User`: id, username, email, password, roles
- `Role`: id, name

payload defines classes for Request and Response objects

We also have **application.properties** for configuring Spring Datasource, Spring Data JPA and App properties (such as JWT Secret string or Token expiration time).

5. Setup

Use Spring web tool or your development tool (Spring Tool Suite, Eclipse, Intellij) to create a Spring Boot project.

Then open **pom.xml** and add these dependencies:

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

```
<dependency>
	<groupId>org.springframework.boot</groupId>
	<artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
	<groupId>org.springframework.boot</groupId>
	<artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
	<groupId>io.jsonwebtoken</groupId>
	<artifactId>jjwt</artifactId>
	<version>0.9.1</version>
</dependency>
```

We also need to add one more dependency.

– If you want to use **PostgreSQL**:

```
<dependency>
	<groupId>org.postgresql</groupId>
	<artifactId>postgresql</artifactId>
	<scope>runtime</scope>
</dependency>
```

– or **MySQL** is your choice:

```
<dependency>
	<groupId>mysql</groupId>
	<artifactId>mysql-connector-java</artifactId>
	<scope>runtime</scope>
</dependency>
```

6. Configure Spring Datasource, JPA, App properties

Under *src/main/resources* folder, open *application.properties*, add some new lines.

For PostgreSQL

```
spring.datasource.url= jdbc:postgresql://localhost:5432/testdb

spring.datasource.username= postgres

spring.datasource.password= 123

spring.jpa.properties.hibernate.jdbc.lob.non_contextual_creation= true

spring.jpa.properties.hibernate.dialect=
org.hibernate.dialect.PostgreSQLDialect

# Hibernate ddl auto (create, create-drop, validate, update)

spring.jpa.hibernate.ddl-auto= update

# App Properties

mkk.app.jwtSecret= secretKey

mkk.app.jwtExpirationMs= 86400000
```

For MySQL

```
spring.datasource.url= jdbc:mysql://localhost:3306/testdb?useSSL=false

spring.datasource.username= root

spring.datasource.password= 123456

spring.jpa.properties.hibernate.dialect=
org.hibernate.dialect.MySQL5InnoDBDialect

spring.jpa.hibernate.ddl-auto= update

# App Properties

mkk.app.jwtSecret= secretKey

mkk.app.jwtExpirationMs= 86400000
```

7. Create the models

We're gonna have 3 tables in database: **users**, **roles** and **user_roles** for many-to-many relationship.

Let's define these models.

In *models* package, create 3 files:

`ERole` enum in *ERole.java*.

In this example, we have 3 roles corresponding to 3 enum.

```
package com.mkk.springjwt.models;

public enum ERole {
      ROLE_USER,
    ROLE_MODERATOR,
    ROLE_ADMIN
}
```

`Role` model in *Role.java*

```
package com.mkk.springjwt.models;
import javax.persistence.*;
@Entity
@Table(name = "roles")
public class Role {
      @Id
      @GeneratedValue(strategy = GenerationType.IDENTITY)
      private Integer id;
      @Enumerated(EnumType.STRING)
      @Column(length = 20)
```

```
	private ERole name;

	public Role() {

	}

	public Role(ERole name) {
		this.name = name;
	}

	public Integer getId() {
		return id;
	}

	public void setId(Integer id) {
		this.id = id;
	}

	public ERole getName() {
		return name;
	}

	public void setName(ERole name) {
		this.name = name;
	}
}
```

`User` model in *User.java*.

It has 5 fields: id, username, email, password, roles.

```
package com.mkk.springjwt.models;

import java.util.HashSet;

import java.util.Set;

import javax.persistence.*;

import javax.validation.constraints.Email;
```

```
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.Size;
@Entity
@Table(	name = "users",
		uniqueConstraints = {
			@UniqueConstraint(columnNames = "username"),
			@UniqueConstraint(columnNames = "email")
		})
public class User {
	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	private Long id;
	@NotBlank
	@Size(max = 20)
	private String username;
	@NotBlank
	@Size(max = 50)
	@Email
	private String email;
	@NotBlank
	@Size(max = 120)
	private String password;
	@ManyToMany(fetch = FetchType.LAZY)
	@JoinTable( name = "user_roles",
				joinColumns = @JoinColumn(name = "user_id"),
				inverseJoinColumns = @JoinColumn(name = "role_id"))
	private Set<Role> roles = new HashSet<>();
```

```
public User() {
}
public User(String username, String email, String password) {
	this.username = username;
	this.email = email;
	this.password = password;
}
public Long getId() {
	return id;
}
public void setId(Long id) {
	this.id = id;
}
public String getUsername() {
	return username;
}
public void setUsername(String username) {
	this.username = username;
}
public String getEmail() {
	return email;
}
public void setEmail(String email) {
	this.email = email;
}
public String getPassword() {
	return password;
```

```
	}
	public void setPassword(String password) {
		this.password = password;
	}
	public Set<Role> getRoles() {
		return roles;
	}
	public void setRoles(Set<Role> roles) {
		this.roles = roles;
	}
}
```

8. Implement Repositories

Now, each model above needs a repository for persisting and accessing data. In *repository* package, let's create 2 repositories.

UserRepository

There are 3 necessary methods that `JpaRepository` supports.

```
package com.mkk.springjwt.repository;
import java.util.Optional;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import com.mkk.springjwt.models.User;
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
	Optional<User> findByUsername(String username);
	Boolean existsByUsername(String username);
	Boolean existsByEmail(String email);
```

```
}
```

RoleRepository

This repository also extends `JpaRepository` and provides a finder method.

```
package com.mkk.springjwt.repository;

import java.util.Optional;

import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.stereotype.Repository;

import com.mkk.springjwt.models.ERole;

import com.mkk.springjwt.models.Role;

@Repository

public interface RoleRepository extends JpaRepository<Role, Long> {

      Optional<Role> findByName(ERole name);

}
```

9. Configure Spring Security

In *security* package, create `WebSecurityConfig` class that extends `WebSecurityConfigurerAdapter`.

WebSecurityConfig.java

```
package com.mkk.springjwt.security;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.authentication.AuthenticationManager;

import
org.springframework.security.config.annotation.authentication.builders.Aut
henticationManagerBuilder;
```

```
import
org.springframework.security.config.annotation.method.configuration.Enable
GlobalMethodSecurity;
import
org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.configuration.EnableWeb
Security;
import
org.springframework.security.config.annotation.web.configuration.WebSecuri
tyConfigurerAdapter;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import
org.springframework.security.web.authentication.UsernamePasswordAuthentica
tionFilter;
import com.mkk.springjwt.security.jwt.AuthEntryPointJwt;
import com.mkk.springjwt.security.jwt.AuthTokenFilter;
import com.mkk.springjwt.security.services.UserDetailsServiceImpl;
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(
            // securedEnabled = true,
            // jsr250Enabled = true,
            prePostEnabled = true)
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
      @Autowired
      UserDetailsServiceImpl userDetailsService;
```

```
	@Autowired
	private AuthEntryPointJwt unauthorizedHandler;
	@Bean
	public AuthTokenFilter authenticationJwtTokenFilter() {
		return new AuthTokenFilter();
	}
	@Override
	public void configure(AuthenticationManagerBuilder
authenticationManagerBuilder) throws Exception {

authenticationManagerBuilder.userDetailsService(userDetailsService).passwo
rdEncoder(passwordEncoder());
	}
	@Bean
	@Override
	public AuthenticationManager authenticationManagerBean() throws
Exception {
		return super.authenticationManagerBean();
	}
	@Bean
	public PasswordEncoder passwordEncoder() {
		return new BCryptPasswordEncoder();
	}
	@Override
	protected void configure(HttpSecurity http) throws Exception {
		http.cors().and().csrf().disable()

.exceptionHandling().authenticationEntryPoint(unauthorizedHandler).and()
```

```
.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS
).and()

.authorizeRequests().antMatchers("/api/auth/**").permitAll()
                    .antMatchers("/api/test/**").permitAll()
                    .anyRequest().authenticated();
            http.addFilterBefore(authenticationJwtTokenFilter(),
UsernamePasswordAuthenticationFilter.class);
      }
}
```

Let me explain the code above.

– `@EnableWebSecurity` allows Spring to find and automatically apply the class to the global Web Security.

– `@EnableGlobalMethodSecurity` provides AOP security on methods. It enables `@PreAuthorize`, `@PostAuthorize`, it also supports JSR-250. You can find more parameters in configuration in Method Security Expressions.

– We override the `configure(HttpSecurity http)` method from `WebSecurityConfigurerAdapter` interface. It tells Spring Security how we configure CORS and CSRF, when we want to require all users to be authenticated or not, which filter (`AuthTokenFilter`) and when we want it to work (filter before `UsernamePasswordAuthenticationFilter`), which Exception Handler is chosen (`AuthEntryPointJwt`).

– Spring Security will load User details to perform authentication & authorization. So it has `UserDetailsService` interface that we need to implement.

– The implementation of `UserDetailsService` will be used for configuring `DaoAuthenticationProvider` by `AuthenticationManagerBuilder.userDetailsService()` method.

– We also need a `PasswordEncoder` for the `DaoAuthenticationProvider`. If we don't specify, it will use plain text.

10. Implement UserDetails & UserDetailsService

If the authentication process is successful, we can get User's information such as username, password, authorities from an `Authentication` object.

```
Authentication authentication =
        authenticationManager.authenticate(
            new UsernamePasswordAuthenticationToken(username, password)
        );
UserDetails userDetails = (UserDetails) authentication.getPrincipal();
// userDetails.getUsername()
// userDetails.getPassword()
// userDetails.getAuthorities()
```

If we want to get more data (id, email…), we can create an implementation of this `UserDetails` interface.

security/services/UserDetailsImpl.java

```
package com.mkk.springjwt.security.services;
import java.util.Collection;
import java.util.List;
import java.util.Objects;
import java.util.stream.Collectors;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;
import com.mkk.springjwt.models.User;
import com.fasterxml.jackson.annotation.JsonIgnore;
```

```
public class UserDetailsImpl implements UserDetails {
	private static final long serialVersionUID = 1L;
	private Long id;
	private String username;
	private String email;
	@JsonIgnore
	private String password;
	private Collection<? extends GrantedAuthority> authorities;
	public UserDetailsImpl(Long id, String username, String email,
String password,
			Collection<? extends GrantedAuthority> authorities) {
		this.id = id;
		this.username = username;
		this.email = email;
		this.password = password;
		this.authorities = authorities;
	}
	public static UserDetailsImpl build(User user) {
		List<GrantedAuthority> authorities = user.getRoles().stream()
				.map(role -> new
SimpleGrantedAuthority(role.getName().name()))
				.collect(Collectors.toList());
		return new UserDetailsImpl(
				user.getId(),
				user.getUsername(),
				user.getEmail(),
				user.getPassword(),
				authorities);
```

```
}
@Override
public Collection<? extends GrantedAuthority> getAuthorities() {
	return authorities;
}
public Long getId() {
	return id;
}
public String getEmail() {
	return email;
}
@Override
public String getPassword() {
	return password;
}
@Override
public String getUsername() {
	return username;
}
@Override
public boolean isAccountNonExpired() {
	return true;
}
@Override
public boolean isAccountNonLocked() {
	return true;
}
```

```
	@Override
	public boolean isCredentialsNonExpired() {
		return true;
	}
	@Override
	public boolean isEnabled() {
		return true;
	}
	@Override
	public boolean equals(Object o) {
		if (this == o)
			return true;
		if (o == null || getClass() != o.getClass())
			return false;
		UserDetailsImpl user = (UserDetailsImpl) o;
		return Objects.equals(id, user.id);
	}
}
```

Look at the code above, you can notice that we convert `Set<Role>` into `List<GrantedAuthority>`. It is important to work with Spring Security and `Authentication` object later.

As I have said before, we need `UserDetailsService` for getting `UserDetails` object. You can look at `UserDetailsService` interface that has only one method:

```
public interface UserDetailsService {
    UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException;
}
```

So we implement it and override `loadUserByUsername()` method.

security/services/UserDetailsServiceImpl.java

```
package com.mkk.springjwt.security.services;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.security.core.userdetails.UserDetails;

import org.springframework.security.core.userdetails.UserDetailsService;

import
org.springframework.security.core.userdetails.UsernameNotFoundException;

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

import com.mkk.springjwt.models.User;

import com.mkk.springjwt.repository.UserRepository;

@Service

public class UserDetailsServiceImpl implements UserDetailsService {

      @Autowired

      UserRepository userRepository;

      @Override

      @Transactional

      public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {

            User user = userRepository.findByUsername(username)

                        .orElseThrow(() -> new
UsernameNotFoundException("User Not Found with username: " + username));

            return UserDetailsImpl.build(user);

      }

}
```

In the code above, we get full custom User object using `UserRepository`, then we build a `UserDetails` object using static `build()` method.

11. Filter the Requests

Let's define a filter that executes once per request. So we create `AuthTokenFilter` class that extends `OncePerRequestFilter` and override `doFilterInternal()` method.

security/jwt/AuthTokenFilter.java

```
package com.mkk.springjwt.security.jwt;

import java.io.IOException;

import javax.servlet.FilterChain;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.beans.factory.annotation.Autowired;

import
org.springframework.security.authentication.UsernamePasswordAuthentication
Token;

import org.springframework.security.core.context.SecurityContextHolder;

import org.springframework.security.core.userdetails.UserDetails;

import
org.springframework.security.web.authentication.WebAuthenticationDetailsSo
urce;

import org.springframework.util.StringUtils;

import org.springframework.web.filter.OncePerRequestFilter;

import com.mkk.springjwt.security.services.UserDetailsServiceImpl;
```

```
public class AuthTokenFilter extends OncePerRequestFilter {

	@Autowired

	private JwtUtils jwtUtils;

	@Autowired

	private UserDetailsServiceImpl userDetailsService;

	private static final Logger logger =
LoggerFactory.getLogger(AuthTokenFilter.class);

	@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) {
                  logger.error("Cannot set user authentication: {}", e);
            }
            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, headerAuth.length());
            }
            return null;
      }
}
```

What we do inside `doFilterInternal()`:

– get `JWT` from the Authorization header (by removing `Bearer` prefix)

– if the request has `JWT`, validate it, parse `username` from it

– from `username`, get `UserDetails` to create an `Authentication` object

– set the current `UserDetails` in SecurityContext using `setAuthentication(authentication)` method.

After this, everytime you want to get `UserDetails`, just use `SecurityContext` like this:

```
UserDetails userDetails =
      (UserDetails)
SecurityContextHolder.getContext().getAuthentication().getPrincipal();
```

```
// userDetails.getUsername()

// userDetails.getPassword()

// userDetails.getAuthorities()
```

12.Create JWT Utility class

This class has 3 funtions:

- generate a `JWT` from username, date, expiration, secret
- get username from `JWT`
- validate a `JWT`

security/jwt/JwtUtils.java

```
package com.mkk.springjwt.security.jwt;

import java.util.Date;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.beans.factory.annotation.Value;

import org.springframework.security.core.Authentication;

import org.springframework.stereotype.Component;

import com.mkk.springjwt.security.services.UserDetailsImpl;

import io.jsonwebtoken.*;

@Component

public class JwtUtils {

      private static final Logger logger =
LoggerFactory.getLogger(JwtUtils.class);

      @Value("${mkk.app.jwtSecret}")

      private String jwtSecret;
```

```
	@Value("${mkk.app.jwtExpirationMs}")
	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((new Date()).getTime() +
jwtExpirationMs))
				.signWith(SignatureAlgorithm.HS512, jwtSecret)
				.compact();
	}
	public String getUserNameFromJwtToken(String token) {
		return
Jwts.parser().setSigningKey(jwtSecret).parseClaimsJws(token).getBody().get
Subject();
	}
	public boolean validateJwtToken(String authToken) {
		try {

Jwts.parser().setSigningKey(jwtSecret).parseClaimsJws(authToken);
			return true;
		} catch (SignatureException e) {
			logger.error("Invalid JWT signature: {}",
e.getMessage());
```

```
            } catch (MalformedJwtException e) {
                logger.error("Invalid JWT token: {}", e.getMessage());
            } catch (ExpiredJwtException e) {
                logger.error("JWT token is expired: {}", e.getMessage());
            } catch (UnsupportedJwtException e) {
                logger.error("JWT token is unsupported: {}",
e.getMessage());
            } catch (IllegalArgumentException e) {
                logger.error("JWT claims string is empty: {}",
e.getMessage());
            }
            return false;
      }
}
```

Remember that we've added `mkk.app.jwtSecret` and `mkk.app.jwtExpirationMs` properties in `application.properties` file.

13. Handle Authentication Exception

Now we create `AuthEntryPointJwt` class that implements `AuthenticationEntryPoint` interface. Then we override the `commence()` method. This method will be triggerd anytime unauthenticated User requests a secured HTTP resource and an `AuthenticationException` is thrown.

security/jwt/AuthEntryPointJwt.java

```
package com.mkk.springjwt.security.jwt;

import java.io.IOException;

import javax.servlet.ServletException;
```

```
import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.security.core.AuthenticationException;

import org.springframework.security.web.AuthenticationEntryPoint;

import org.springframework.stereotype.Component;

@Component

public class AuthEntryPointJwt implements AuthenticationEntryPoint {

      private static final Logger logger =
LoggerFactory.getLogger(AuthEntryPointJwt.class);

      @Override

      public void commence(HttpServletRequest request, HttpServletResponse
response,

                  AuthenticationException authException) throws
IOException, ServletException {

            logger.error("Unauthorized error: {}",
authException.getMessage());

            response.sendError(HttpServletResponse.SC_UNAUTHORIZED,
"Error: Unauthorized");

      }

}
```

`HttpServletResponse.SC_UNAUTHORIZED` is the **401** Status code. It indicates that the request requires HTTP authentication.

We've already built all things for Spring Security. The next sections of this tutorial will show you how to implement Controllers for our RestAPIs.

14. Define payloads for Spring RestController

Let me summarize the payloads for our RestAPIs:

– Requests:

- LoginRequest: { username, password }
- SignupRequest: { username, email, password }

– Responses:

- JwtResponse: { token, type, id, username, email, roles }
- MessageResponse: { message }

To keep the tutorial not so long, I don't show these POJOs here.

You can find details for payload classes in source code of the project on Github.

15. Create Spring RestAPIs Controllers

Controller for Authentication

This controller provides APIs for register and login actions.

– `/api/auth/signup`

- check existing `username`/`email`
- create new `User` (with `ROLE_USER` if not specifying role)
- save `User` to database using `UserRepository`

– `/api/auth/signin`

- authenticate { username, pasword }
- update `SecurityContext` using `Authentication` object
- generate `JWT`
- get `UserDetails` from `Authentication` object
- response contains `JWT` and `UserDetails` data

controllers/AuthController.java

```
package com.mkk.springjwt.controllers;

import java.util.HashSet;

import java.util.List;

import java.util.Set;

import java.util.stream.Collectors;

import javax.validation.Valid;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.http.ResponseEntity;

import org.springframework.security.authentication.AuthenticationManager;

import
org.springframework.security.authentication.UsernamePasswordAuthentication
Token;

import org.springframework.security.core.Authentication;

import org.springframework.security.core.context.SecurityContextHolder;

import org.springframework.security.crypto.password.PasswordEncoder;

import org.springframework.web.bind.annotation.CrossOrigin;

import org.springframework.web.bind.annotation.PostMapping;

import org.springframework.web.bind.annotation.RequestBody;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

import com.mkk.springjwt.models.ERole;

import com.mkk.springjwt.models.Role;

import com.mkk.springjwt.models.User;

import com.mkk.springjwt.payload.request.LoginRequest;

import com.mkk.springjwt.payload.request.SignupRequest;
```

```
import com.mkk.springjwt.payload.response.JwtResponse;

import com.mkk.springjwt.payload.response.MessageResponse;

import com.mkk.springjwt.repository.RoleRepository;

import com.mkk.springjwt.repository.UserRepository;

import com.mkk.springjwt.security.jwt.JwtUtils;

import com.mkk.springjwt.security.services.UserDetailsImpl;

@CrossOrigin(origins = "*", maxAge = 3600)

@RestController

@RequestMapping("/api/auth")

public class AuthController {

      @Autowired

      AuthenticationManager authenticationManager;

      @Autowired

      UserRepository userRepository;

      @Autowired

      RoleRepository roleRepository;

      @Autowired

      PasswordEncoder encoder;

      @Autowired

      JwtUtils jwtUtils;

      @PostMapping("/signin")

      public ResponseEntity<?> authenticateUser(@Valid @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(item -> item.getAuthority())
                        .collect(Collectors.toList());
            return ResponseEntity.ok(new JwtResponse(jwt,

userDetails.getId(),

userDetails.getUsername(),

userDetails.getEmail(),

roles));
      }
      @PostMapping("/signup")
      public ResponseEntity<?> registerUser(@Valid @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(),
                                        signUpRequest.getEmail(),

encoder.encode(signUpRequest.getPassword()));
            Set<String> strRoles = signUpRequest.getRole();
            Set<Role> roles = new HashSet<>();
            if (strRoles == null) {
                Role userRole =
roleRepository.findByName(ERole.ROLE_USER)
                            .orElseThrow(() -> new
RuntimeException("Error: Role is not found."));
                roles.add(userRole);
            } else {
                strRoles.forEach(role -> {
```

```
                        switch (role) {
                        case "admin":
                              Role adminRole =
roleRepository.findByName(ERole.ROLE_ADMIN)
                                          .orElseThrow(() -> new
RuntimeException("Error: Role is not found."));
                              roles.add(adminRole);
                              break;
                        case "mod":
                              Role modRole =
roleRepository.findByName(ERole.ROLE_MODERATOR)
                                          .orElseThrow(() -> new
RuntimeException("Error: Role is not found."));
                              roles.add(modRole);
                              break;
                        default:
                              Role userRole =
roleRepository.findByName(ERole.ROLE_USER)
                                          .orElseThrow(() -> new
RuntimeException("Error: Role is not found."));
                              roles.add(userRole);
                        }
                  });
            }
            user.setRoles(roles);
            userRepository.save(user);
```

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

      }

}
```

Controller for testing Authorization

There are 4 APIs:

- `/api/test/all` for public access
- `/api/test/user` for users has `ROLE_USER` or `ROLE_MODERATOR` or `ROLE_ADMIN`
- `/api/test/mod` for users has `ROLE_MODERATOR`
- `/api/test/admin` for users has `ROLE_ADMIN`

Do you remember that we used `@EnableGlobalMethodSecurity(prePostEnabled = true)` for `WebSecurityConfig` class?

```
@Configuration

@EnableWebSecurity

@EnableGlobalMethodSecurity(prePostEnabled = true)

public class WebSecurityConfig extends WebSecurityConfigurerAdapter { ...
}
```

Now we can secure methods in our Apis with `@PreAuthorize` annotation easily.

controllers/TestController.java

```
package com.mkk.springjwt.controllers;

import org.springframework.security.access.prepost.PreAuthorize;

import org.springframework.web.bind.annotation.CrossOrigin;

import org.springframework.web.bind.annotation.GetMapping;
```

```
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
@CrossOrigin(origins = "*", maxAge = 3600)
@RestController
@RequestMapping("/api/test")
public class TestController {
	@GetMapping("/all")
	public String allAccess() {
		return "Public Content.";
	}

	@GetMapping("/user")
	@PreAuthorize("hasRole('USER') or hasRole('MODERATOR') or
hasRole('ADMIN')")
	public String userAccess() {
		return "User Content.";
	}
	@GetMapping("/mod")
	@PreAuthorize("hasRole('MODERATOR')")
	public String moderatorAccess() {
		return "Moderator Board.";
	}
	@GetMapping("/admin")
	@PreAuthorize("hasRole('ADMIN')")
```

```
	public String adminAccess() {
		return "Admin Board.";
	}
}
```

16. User Logs with AOP

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

```
@Aspect
@Component
@Slf4j
@ConditionalOnExpression("${aspect.enabled:true}")
public class ExecutionTimeAdvice {

    @Around("@annotation(TrackExecutionTime)")
    public Object executionTime(ProceedingJoinPoint point) throws
Throwable {
		long startTime = System.currentTimeMillis();
		Object object = point.proceed();
		long endtime = System.currentTimeMillis();
```

```
        log.info("Class Name: "+
point.getSignature().getDeclaringTypeName() +". Method Name: "+
point.getSignature().getName() + ". Time taken for Execution is : " +
(endtime-startTime) +"ms");

        return object;

    }

}

@Target(ElementType.METHOD)

@Retention(RetentionPolicy.RUNTIME)

public @interface TrackExecutionTime {

}
```