**Protect REST APIs with Spring Security and JWT**

When you design REST APIs, you have to consider how to protect REST APIs. In a Spring based application, Spring Security is a great authentication and authorization solution, and it provides several options for securing your REST APIs.

**>>Spring WebFlux users, move to this**[**post.**](https://medium.com/@hantsy/protect-rest-apis-with-spring-security-reactive-and-jwt-7b209a0510f1)

The simplest approach is utilizing HTTP Basic which is activated by default when you are bootstrap a Spring Boot based application. It is good for development purpose, and it is used frequently in development phase, but it is not recommended in a production environment.

Spring Session(with Spring Security) provides a simple strategy to create and validate header based token(session id), it can be used for protecting RESTful APIs, I have demonstrated it in my [microservice sample](https://github.com/hantsy/spring-microservice-sample) and [my RESTful sample](https://github.com/hantsy/angular-spring-reactive-sample).

Beside these, Spring Security OAuth (a subproject under Spring Security) provides a complete solution of OAuth authorization, including the implementations of all roles defined in OAuth2 protocol, such as Authorization Server, Resource Server, OAuth2 Client etc. Spring Cloud adds Single Sign On capability to **OAuth2 Client** via its subproject Spring Cloud Security. In the Spring Security OAuth based solution, the content of access token can be a signed JWT token or an opaque value, and we have to follow the standard OAuth2 authorization flow to obtain access token.

But for those applications owned by the resource owner and there is no plan to expose these APIs to third party applications, a simple JWT token based authorization is more simple and reasonable(we do not need manage the credentials of third party client applications). Spring Security itself does not provide such an option, fortunately it is not difficult to implement it by weaving our custom filter into the Spring Security Filter Chain. In this post, we will create such a custom JWT authentication solution.

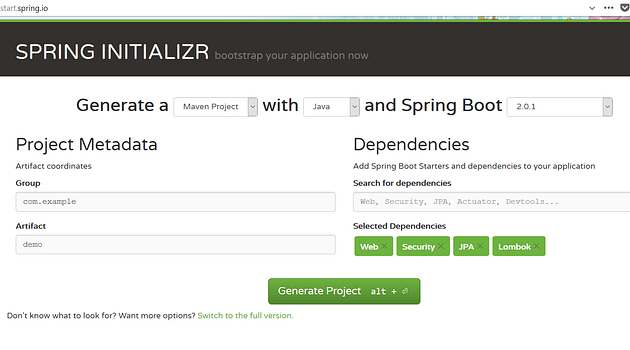
In this sample application, the custom JWT token based authentication flow can be designated as the following steps.

1. Get the JWT based token from the authentication endpoint, eg /auth/signin.
2. Extract token from the authentication result.
3. Set the HTTP header Authorization value as Bearer jwt\_token.
4. Then send a request to access the protected resources.
5. If the requested resource is protected, Spring Security will use our custom Filter to validate the JWT token, and build an Authentication object and set it in Spring Security specific SecurityContextHolder to complete the authentication progress.
6. If the JWT token is valid it will return the requested resource to client.

**Generate the project skeleton**

The quickest way to create a new Spring Boot project is using [Spring Initializr](http://start.spring.io/) to generate the base codes.

Open your browser, and go to [http://start.spring.io](http://start.spring.io/). In the **Dependencies** field, select **Web**, **Security**, **JPA**, **Lombok**, then click **Generate** button or press **ALT+ENTER** keys to generate the project skeleton codes.



Waiting for a while for downloading the generated codes, when it is done, extract the zip file into your local system.

Open your favorite IDE, eg, Intellij IDEA, NetBeans IDE, and import it.

**Create a sample REST APIs**

In this application, we will expose REST APIs for vehicle resources.

URI request response description /vehicles GET 200, [{id: 1, name:’title’}, {id:’2', name:’title 2'}] Get all vehicles /vehicles POST {name:’title’} 201, no content in body, the value of HTTP response header **Location** is the uri of the new created vehicle Create a new vehicle /vehicles/{id} GET 200, {id:’1', name:’title’} Get a vehicle by id /vehicles/{id} PUT {name:’title’} 204, no content in body Update a certain vehicle by id /vehicles/{id} DELETE 204, no content Delete a vehicle by id

Create a JPA entity Vehicle.

@Entity  
@Table(name="vehicles")  
@Data  
@Builder  
@AllArgsConstructor  
@NoArgsConstructor  
public class Vehicle implements Serializable { @Id  
 @GeneratedValue(strategy = GenerationType.AUTO)  
 private Long id ; @Column  
 private String name;  
}

Create a Repository for Vehicle.

public interface VehicleRepository extends JpaRepository<Vehicle, Long> {  
}

Create a Spring MVC basec Controller to expose REST APIs.

@RestController  
@RequestMapping("/v1/vehicles")  
public class VehicleController { private VehicleRepository vehicles; public VehicleController(VehicleRepository vehicles) {  
 this.vehicles = vehicles;  
 }  
 @GetMapping("")  
 public ResponseEntity all() {  
 return ok(this.vehicles.findAll());  
 } @PostMapping("")  
 public ResponseEntity save(@RequestBody VehicleForm form, HttpServletRequest request) {  
 Vehicle saved = this.vehicles.save(Vehicle.builder().name(form.getName()).build());  
 return created(  
 ServletUriComponentsBuilder  
 .fromContextPath(request)  
 .path("/v1/vehicles/{id}")  
 .buildAndExpand(saved.getId())  
 .toUri())  
 .build();  
 } @GetMapping("/{id}")  
 public ResponseEntity get(@PathVariable("id") Long id) {  
 return ok(this.vehicles.findById(id).orElseThrow(() -> new VehicleNotFoundException()));  
 }  
 @PutMapping("/{id}")  
 public ResponseEntity update(@PathVariable("id") Long id, @RequestBody VehicleForm form) {  
 Vehicle existed = this.vehicles.findById(id).orElseThrow(() -> new VehicleNotFoundException());  
 existed.setName(form.getName()); this.vehicles.save(existed);  
 return noContent().build();  
 } @DeleteMapping("/{id}")  
 public ResponseEntity delete(@PathVariable("id") Long id) {  
 Vehicle existed = this.vehicles.findById(id).orElseThrow(() -> new VehicleNotFoundException());  
 this.vehicles.delete(existed);  
 return noContent().build();  
 }  
}

It is simple and stupid. We defined a VehicleNotFoundException which will be thrown if the vehicle is not found by id.

Create a simple exception handler to handle our custom exceptions.

@RestControllerAdvice  
@Slf4j  
public class RestExceptionHandler { @ExceptionHandler(value = {VehicleNotFoundException.class})  
 public ResponseEntity vehicleNotFound(VehicleNotFoundException ex, WebRequest request) {  
 log.debug("handling VehicleNotFoundException...");  
 return notFound().build();  
 }  
}

Create a CommandLineRunner bean to initialize some vehicles data at the application startup stage.

@Component  
@Slf4j  
public class DataInitializer implements CommandLineRunner { @Autowired  
 VehicleRepository vehicles;  
 @Override  
 public void run(String... args) throws Exception {  
 log.debug("initializing vehicles data...");  
 Arrays.asList("moto", "car").forEach(v -> this.vehicles.saveAndFlush(Vehicle.builder().name(v).build())); log.debug("printing all vehicles...");  
 this.vehicles.findAll().forEach(v -> log.debug(" Vehicle :" + v.toString()));  
 }  
}

Run the application via executing command line mvn spring-boot:run in your terminal or running Application class in IDE directly .

Open your terminal, use curl to test the APIs.

>curl http://localhost:8080/v1/vehicles  
[ {  
 "id" : 1,  
 "name" : "moto"  
}, {  
 "id" : 2,  
 "name" : "car"  
} ]

Spring Data Rest provides capability of exposing APIs via Repository interface directly.

Add a @RepositoryRestResource annotation on the existed VehicleRepository interface.

@RepositoryRestResource(path = "vehicles", collectionResourceRel = "vehicles", itemResourceRel = "vehicle")  
public interface VehicleRepository extends JpaRepository<Vehicle, Long> {  
}

Restart the application and try to access <http://localhost:8080/vehicles> .

curl -X GET http://localhost:8080/vehicles   
{  
 "\_embedded" : {  
 "vehicles" : [ {  
 "name" : "moto",  
 "\_links" : {  
 "self" : {  
 "href" : "http://localhost:8080/vehicles/1"  
 },  
 "vehicle" : {  
 "href" : "http://localhost:8080/vehicles/1"  
 }  
 }  
 }, {  
 "name" : "car",  
 "\_links" : {  
 "self" : {  
 "href" : "http://localhost:8080/vehicles/2"  
 },  
 "vehicle" : {  
 "href" : "http://localhost:8080/vehicles/2"  
 }  
 }  
 } ]  
 },  
 "\_links" : {  
 "self" : {  
 "href" : "http://localhost:8080/vehicles{?page,size,sort}",  
 "templated" : true  
 },  
 "profile" : {  
 "href" : "http://localhost:8080/profile/vehicles"  
 }  
 },  
 "page" : {  
 "size" : 20,  
 "totalElements" : 2,  
 "totalPages" : 1,  
 "number" : 0  
 }  
}

It utilizes Spring HATEOAS project to expose richer REST APIs which archives Richardson Mature Model Level 3(self documentation).

**Secures the REST APIs**

Now we will create a custom JWT token based authentication filter to validate the JWT token.

Create a Filter name JwtTokenFilter for the JWT token validation.

public class JwtTokenFilter extends GenericFilterBean { private JwtTokenProvider jwtTokenProvider; public JwtTokenFilter(JwtTokenProvider jwtTokenProvider) {  
 this.jwtTokenProvider = jwtTokenProvider;  
 } @Override  
 public void doFilter(ServletRequest req, ServletResponse res, FilterChain filterChain)  
 throws IOException, ServletException { String token = jwtTokenProvider.resolveToken((HttpServletRequest) req);  
 if (token != null && jwtTokenProvider.validateToken(token)) {  
 Authentication auth = token != null ? jwtTokenProvider.getAuthentication(token) : null;  
 SecurityContextHolder.getContext().setAuthentication(auth);  
 }  
 filterChain.doFilter(req, res);  
 }}

It uses JwtTokenProvider to treat with JWT, such as generating JWT token, parsing JWT claims.

@Component  
public class JwtTokenProvider { @Value("${security.jwt.token.secret-key:secret}")  
 private String secretKey = "secret"; @Value("${security.jwt.token.expire-length:3600000}")  
 private long validityInMilliseconds = 3600000; // 1h @Autowired  
 private UserDetailsService userDetailsService; @PostConstruct  
 protected void init() {  
 secretKey = Base64.getEncoder().encodeToString(secretKey.getBytes());  
 } public String createToken(String username, List<String> roles) { Claims claims = Jwts.claims().setSubject(username);  
 claims.put("roles", roles); Date now = new Date();  
 Date validity = new Date(now.getTime() + validityInMilliseconds); return Jwts.builder()//  
 .setClaims(claims)//  
 .setIssuedAt(now)//  
 .setExpiration(validity)//  
 .signWith(SignatureAlgorithm.HS256, secretKey)//  
 .compact();  
 } public Authentication getAuthentication(String token) {  
 UserDetails userDetails = this.userDetailsService.loadUserByUsername(getUsername(token));  
 return new UsernamePasswordAuthenticationToken(userDetails, "", userDetails.getAuthorities());  
 } public String getUsername(String token) {  
 return Jwts.parser().setSigningKey(secretKey).parseClaimsJws(token).getBody().getSubject();  
 } public String resolveToken(HttpServletRequest req) {  
 String bearerToken = req.getHeader("Authorization");  
 if (bearerToken != null && bearerToken.startsWith("Bearer ")) {  
 return bearerToken.substring(7, bearerToken.length());  
 }  
 return null;  
 } public boolean validateToken(String token) {  
 try {  
 Jws<Claims> claims = Jwts.parser().setSigningKey(secretKey).parseClaimsJws(token); if (claims.getBody().getExpiration().before(new Date())) {  
 return false;  
 } return true;  
 } catch (JwtException | IllegalArgumentException e) {  
 throw new InvalidJwtAuthenticationException("Expired or invalid JWT token");  
 }  
 }}

Create a standalone Configurer class to setup JwtTokenFilter.

public class JwtConfigurer extends SecurityConfigurerAdapter<DefaultSecurityFilterChain, HttpSecurity> { private JwtTokenProvider jwtTokenProvider; public JwtConfigurer(JwtTokenProvider jwtTokenProvider) {  
 this.jwtTokenProvider = jwtTokenProvider;  
 } @Override  
 public void configure(HttpSecurity http) throws Exception {  
 JwtTokenFilter customFilter = new JwtTokenFilter(jwtTokenProvider);  
 http.addFilterBefore(customFilter, UsernamePasswordAuthenticationFilter.class);  
 }  
}

Apply this configurer in our application scoped SecurityConfig.

@Configuration  
public class SecurityConfig extends WebSecurityConfigurerAdapter {  
 @Autowired  
 JwtTokenProvider jwtTokenProvider; @Bean  
 @Override  
 public AuthenticationManager authenticationManagerBean() throws Exception {  
 return super.authenticationManagerBean();  
 } @Override  
 protected void configure(HttpSecurity http) throws Exception {  
 //@formatter:off  
 http  
 .httpBasic().disable()  
 .csrf().disable()  
 .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS)  
 .and()  
 .authorizeRequests()  
 .antMatchers("/auth/signin").permitAll()  
 .antMatchers(HttpMethod.GET, "/vehicles/\*\*").permitAll()  
 .antMatchers(HttpMethod.DELETE, "/vehicles/\*\*").hasRole("ADMIN")  
 .antMatchers(HttpMethod.GET, "/v1/vehicles/\*\*").permitAll()  
 .anyRequest().authenticated()  
 .and()  
 .apply(new JwtConfigurer(jwtTokenProvider));  
 //@formatter:on  
 }  
}

To enable Spring Security, we have to provide a custom UserDetailsService bean at runtime.

@Component  
public class CustomUserDetailsService implements UserDetailsService { private UserRepository users; public CustomUserDetailsService(UserRepository users) {  
 this.users = users;  
 } @Override  
 public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {  
 return this.users.findByUsername(username)  
 .orElseThrow(() -> new UsernameNotFoundException("Username: " + username + " not found"));  
 }  
}

The CustomUserDetailsService is trying to fetch user data by username from database.

The User is a standard JPA entity, and to simplify the work, it also implements the Spring Security specific UserDetails interface.

@Entity  
@Table(name="users")  
@Data  
@Builder  
@NoArgsConstructor  
@AllArgsConstructor  
public class User implements UserDetails {  
 @Id  
 @GeneratedValue(strategy = GenerationType.AUTO)  
 Long id; @NotEmpty  
 private String username; @NotEmpty  
 private String password; @ElementCollection(fetch = FetchType.EAGER)  
 @Builder.Default  
 private List<String> roles = new ArrayList<>(); @Override  
 public Collection<? extends GrantedAuthority> getAuthorities() {  
 return this.roles.stream().map(SimpleGrantedAuthority::new).collect(toList());  
 } @Override  
 public String getPassword() {  
 return this.password;  
 } @Override  
 public String getUsername() {  
 return this.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;  
 }  
}

Create a Repository interface for User entity.

public interface UserRepository extends JpaRepository<User, Long> { Optional<User> findByUsername(String username);}

Create a controller to authenticate user.

@RestController  
@RequestMapping("/auth")  
public class AuthController { @Autowired  
 AuthenticationManager authenticationManager; @Autowired  
 JwtTokenProvider jwtTokenProvider; @Autowired  
 UserRepository users; @PostMapping("/signin")  
 public ResponseEntity signin(@RequestBody AuthenticationRequest data) { try {  
 String username = data.getUsername();  
 authenticationManager.authenticate(new UsernamePasswordAuthenticationToken(username, data.getPassword()));  
 String token = jwtTokenProvider.createToken(username, this.users.findByUsername(username).orElseThrow(() -> new UsernameNotFoundException("Username " + username + "not found")).getRoles()); Map<Object, Object> model = new HashMap<>();  
 model.put("username", username);  
 model.put("token", token);  
 return ok(model);  
 } catch (AuthenticationException e) {  
 throw new BadCredentialsException("Invalid username/password supplied");  
 }  
 }  
}

Create an endpoint to fetch current user info.

@RestController()  
public class UserinfoController { @GetMapping("/me")  
 public ResponseEntity currentUser(@AuthenticationPrincipal UserDetails userDetails){  
 Map<Object, Object> model = new HashMap<>();  
 model.put("username", userDetails.getUsername());  
 model.put("roles", userDetails.getAuthorities()  
 .stream()  
 .map(a -> ((GrantedAuthority) a).getAuthority())  
 .collect(toList())  
 );  
 return ok(model);  
 }  
}

When the current user is authenticated, @AuthenticationPrincipal will bind to the current principal.

Add two users for test purpose in our initializing class.

@Component  
@Slf4j  
public class DataInitializer implements CommandLineRunner { //... @Autowired  
 UserRepository users; @Autowired  
 PasswordEncoder passwordEncoder; @Override  
 public void run(String... args) throws Exception {  
 //... this.users.save(User.builder()  
 .username("user")  
 .password(this.passwordEncoder.encode("password"))  
 .roles(Arrays.asList( "ROLE\_USER"))  
 .build()  
 ); this.users.save(User.builder()  
 .username("admin")  
 .password(this.passwordEncoder.encode("password"))  
 .roles(Arrays.asList("ROLE\_USER", "ROLE\_ADMIN"))  
 .build()  
 ); log.debug("printing all users...");  
 this.users.findAll().forEach(v -> log.debug(" User :" + v.toString()));  
 }  
}

Now use curl to try this authentication progress.

Sign in via user/password pair.

curl -X POST http://localhost:8080/auth/signin -H "Content-Type:application/json" -d "{\"username\":\"user\", \"password\":\"password\"}"  
{  
 "username" : "user",  
 "token" : "eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJ1c2VyIiwicm9sZXMiOlsiUk9MRV9VU0VSIl0sImlhdCI6MTUyNDY0OTI4OSwiZXhwIjoxNTI0NjUyODg5fQ.Lj1w6vPJNdJbcY6cAhO3DbkgCAqpG7lzztzUeKMyNyE"  
}

Put the token value to HTTP header Authorization, set its value as Bearer token, then access the current user info.

curl -X GET http://localhost:8080/me -H "Authorization: Bearer eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJ1c2VyIiwicm9sZXMiOlsiUk9MRV9VU0VSIl0sImlhdCI6MTUyNDY0OTI4OSwiZXhwIjoxNTI0NjUyODg5fQ.Lj1w6vPJNdJbcY6cAhO3DbkgCAqpG7lzztzUeKMyNyE"  
{  
 "roles" : [ "ROLE\_USER" ],  
 "username" : "user"  
}