

Spring Security + JWT Authentication — Beginner Friendly Note (Using TODO API Example)

This note explains **Spring Security**, **JWT tokens**, authentication flow, and integrates everything into the **TODO REST API**. Written for beginners with crystal-clear examples.

★ 1. What is Spring Security?

Spring Security is a powerful framework that secures:

- REST APIs
- MVC applications
- WebSockets
- Method-level authorization

It provides:

✓ Authentication (who are you?) ✓ Authorization (what can you access?) ✓ Password hashing ✓
Session & token management ✓ Filters ✓ CSRF protection

★ 2. Why JWT for REST APIs?

Traditional login creates a **server session**, but REST APIs must be:

- **Stateless**
- **Scalable**
- **Distributed**

So instead of storing user data in server memory, we use a **JWT token**.

What is JWT?

A JSON Web Token is:

- A string
- Containing encrypted user info
- Signed (not encrypted)
- Sent with every request

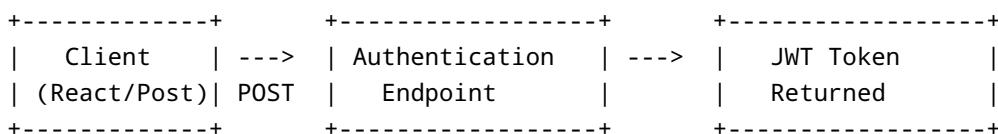
Example JWT:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
```

JWT contains:

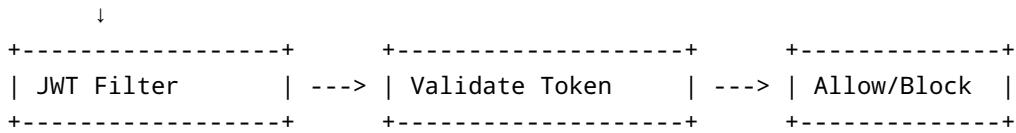
- Header (algorithm)
- Payload (user info)
- Signature (security)

★ 3. Spring Security + JWT Architecture (Simple Diagram)



Then for every request:

Client: Authorization: Bearer <token>



★ 4. Project Structure

```
com.example.security
├── config
│   └── SecurityConfig.java
├── controller
│   ├── AuthController.java
│   └── TodoController.java
├── dto
│   ├── LoginRequest.java
│   └── JwtResponse.java
├── entity
│   ├── UserEntity.java
│   └── TodoEntity.java
├── filter
│   └── JwtAuthFilter.java
└── repository
```

```
|   └── UserRepository.java  
|       └── TodoRepository.java  
└── service  
    └── JwtService.java  
        └── UserService.java  
└── SecurityJwtApplication.java
```

★ 5. Step 1 — Add Required Dependencies

In `pom.xml`:

```
<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-api</artifactId>  
    <version>0.11.5</version>  
</dependency>  
<dependency>  
    <groupId>io.jsonwebtoken</groupId>  
    <artifactId>jjwt-impl</artifactId>  
    <version>0.11.5</version>  
    <scope>runtime</scope>  
</dependency>  
<dependency>  
    <groupId>io.jsonwebtoken</groupId>  
    <artifactId>jjwt-jackson</artifactId>  
    <version>0.11.5</version>  
    <scope>runtime</scope>  
</dependency>
```

★ 6. Step 2 — Create User Entity

```
@Entity  
public class UserEntity {  
    @Id
```

```

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

    private String username;
    private String password;
}

```

★ 7. Step 3 — UserRepository

```

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

```

★ 8. Step 4 — JWT Service

Utility class to create and validate JWT.

```

@Service
public class JwtService {

    private String secret = "mysecretkey12345";

    public String generateToken(String username) {
        return Jwts.builder()
            .setSubject(username)
            .setIssuedAt(new Date())
            .setExpiration(new Date(System.currentTimeMillis() + 3600 *
1000)) // 1 hour
            .signWith(SignatureAlgorithm.HS256, secret)
            .compact();
    }

    public String extractUsername(String token) {
        return Jwts.parser()
            .setSigningKey(secret)
            .parseClaimsJws(token)
            .getBody()
            .getSubject();
    }

    public boolean validateToken(String token) {
        try {
            Jwts.parser().setSigningKey(secret).parseClaimsJws(token);
            return true;
        }
    }
}

```

```
        } catch (Exception e) {
            return false;
        }
    }
}
```

★9. Step 5 — Create Authentication Request + Response DTOs

LoginRequest

```
public class LoginRequest {
    public String username;
    public String password;
}
```

JwtResponse

```
public class JwtResponse {
    public String token;

    public JwtResponse(String token) {
        this.token = token;
    }
}
```

★10. Step 6 — Authentication Controller

This controller logs in users and returns a JWT.

```
@RestController
@RequestMapping("/auth")
public class AuthController {

    @Autowired
    private UserRepository userRepo;

    @Autowired
    private JwtService jwtService;

    @PostMapping("/login")
    public ResponseEntity<?> login(@RequestBody LoginRequest request) {
```

```

        UserEntity user = userRepo.findByUsername(request.username)
            .orElseThrow(() -> new RuntimeException("User not found"));

        if (!user.getPassword().equals(request.password)) {
            return ResponseEntity.status(401).body("Invalid credentials");
        }

        String token = jwtService.generateToken(user.getUsername());

        return ResponseEntity.ok(new JwtResponse(token));
    }
}

```

★11. Step 7 — JWT Authentication Filter

This filter checks JWT token for every request.

```

@Component
public class JwtAuthFilter extends OncePerRequestFilter {

    @Autowired
    private JwtService jwtService;

    @Autowired
    private UserRepository userRepo;

    @Override
    protected void doFilterInternal(HttpServletRequest request,
                                    HttpServletResponse response,
                                    FilterChain filterChain)
        throws ServletException, IOException {

        String authHeader = request.getHeader("Authorization");

        if (authHeader != null && authHeader.startsWith("Bearer ")) {
            String token = authHeader.substring(7);

            if (jwtService.validateToken(token)) {
                String username = jwtService.extractUsername(token);

                UserEntity user =
userRepo.findByUsername(username).orElse(null);

                if (user != null) {
                    UsernamePasswordAuthenticationToken authToken =
                        new

```

```

        UsernamePasswordAuthenticationToken(username, null, List.of()));

        SecurityContextHolder.getContext().setAuthentication(authToken);
    }
}

filterChain.doFilter(request, response);
}
}

```

★12. Step 8 — Security Configuration

```

@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Autowired
    private JwtAuthFilter jwtFilter;

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws
Exception {

        http.csrf().disable()
            .authorizeHttpRequests()
            .requestMatchers("/auth/**").permitAll()
            .anyRequest().authenticated()
            .and()
            .addFilterBefore(jwtFilter,
UsernamePasswordAuthenticationFilter.class);

        return http.build();
    }
}

```

★13. Step 9 — Secured TODO API

```

@RestController
@RequestMapping("/api/todos")
public class TodoController {

    private List<String> todos = new ArrayList<>(List.of("Learn Java",

```

```
"Build REST API"));

    @GetMapping
    public List<String> getTodos() {
        return todos;
    }

    @PostMapping
    public String addTodo(@RequestBody String todo) {
        todos.add(todo);
        return "Added";
    }
}
```

Now, this TODO API is secured by JWT.

★14. Testing the Flow (Using Postman)

Step 1 — Login

```
POST http://localhost:8080/auth/login Body:
```

```
{
  "username": "john",
  "password": "1234"
}
```

Response:

```
{
  "token": "eyJhbGciOiJIUzI1NiIs..."
```

Step 2 — Access TODO API

```
GET http://localhost:8080/api/todos Headers:
```

```
Authorization: Bearer <your_jwt_token>
```

Success !!

★15. Summary Table

Concept	Explanation
Spring Security	Handles authentication & authorization
JWT	Token containing user info
AuthController	Generates JWT
JwtService	Creates/validates JWT
JwtFilter	Secures every request
SecurityConfig	Defines public vs private URLs
TODO API	Secured example endpoint

Final Notes

This setup gives you:

- Stateless Authentication
- JWT protection
- Secure TODO API
- Customizable roles & permissions

If you want, I can also add:

- Refresh Tokens
- Logout handling
- Role-based authorization (ADMIN/USER)
- Storing users in database with encrypted passwords
- Swagger Documentation

Just tell me!