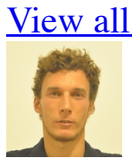
# REST Security with JWT using Java and Spring Security

View all articles



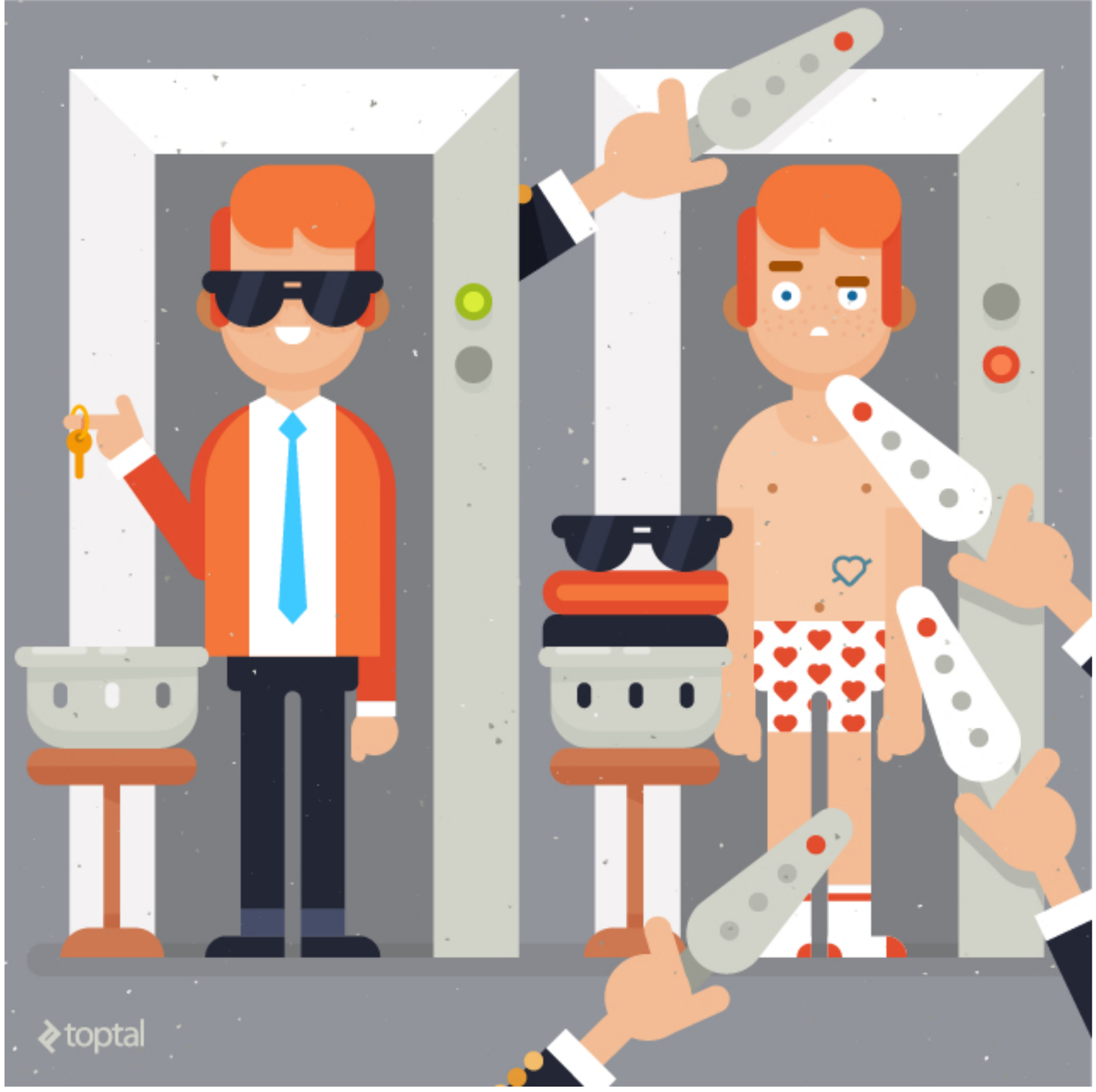
by **Dejan Milosevic** - Freelance Software Engineer @ Toptal

#Java #JWT #SpringFramework #SpringSecurity



## Security

Security is the enemy of convenience, and vice versa. This statement is true for any system, virtual or real, from the physical house entrance to web banking platforms. Engineers are constantly trying to find the right balance for the given use case, leaning to one side or the other. Usually, when a new threat appears, we move towards security and away from convenience. Then, we see if we can recover some lost convenience without reducing the security too much. Moreover, this vicious circle goes on forever.




Security is the enemy of convenience, and vice versa.

Tweet

Let’s try to see where REST services currently stand regarding security and convenience. REST (which stands for Representational State Transfer) services started off as an extremely simplified approach to Web Services that had huge specifications and cumbersome formats, such as WSDL for describing the service, or SOAP for specifying the message format. In REST, we have none of those. We can describe the REST service in a plain text file and use any message format we want, such as JSON, XML or even plain text again. The simplified approach was applied to the security of REST services as well; no defined standard imposes a particular way to authenticate users.

Although REST services do not have much specified, an important one is the lack of state. It means the server does not keep any client state, with sessions as a good example. Thus, the server replies to each request as if it was the first the client has made. However, even now, many implementations still use cookie based authentication, which is inherited from standard website architectural design. The stateless approach of REST makes session cookies inappropriate from the security standpoint, but nevertheless, they are still widely used. Besides ignoring the required statelessness, simplified approach came as an expected security trade-off. Compared to the WS-Security standard used for Web Services, it is much easier to create and consume REST services, hence convenience went through the roof. The trade-off is pretty slim security; session hijacking and cross-site request forgery (XSRF) are the most common security issues.

In trying to get rid of client sessions from the server, some other methods have been used occasionally, such as Basic or Digest HTTP authentication. Both use an `Authorization` header to transmit user credentials, with some encoding (HTTP Basic) or encryption (HTTP Digest) added. Of course, they carried the same flaws found in websites: HTTP Basic had to be used over HTTPS since username and password are sent in easily reversible base64 encoding, and HTTP Digest forced the use of obsolete MD5 hashing that is proven to be insecure.

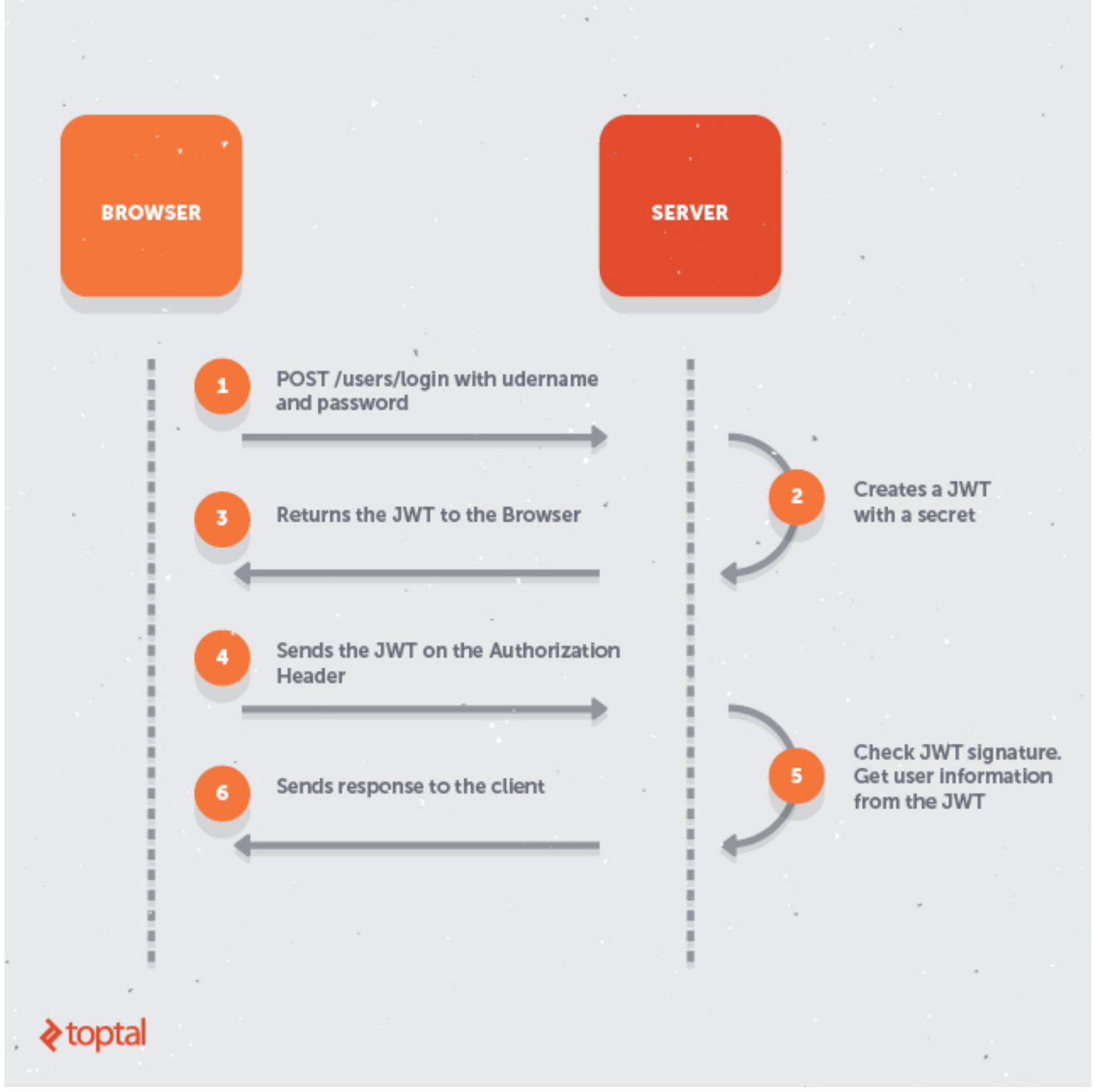
Finally, some implementations used arbitrary tokens to authenticate clients. This option seems to be the best we have, for now. If implemented properly, it fixes all the security problems of HTTP Basic, HTTP Digest or session cookies, it is simple to use, and it follows the stateless pattern.

However, with such arbitrary tokens, there’s little standard involved. Every service provider had his or her idea of what to put in the token, and how to encode or encrypt it. Consuming services from different providers required additional setup time, just to adapt to the specific token format used. The other methods, on the other hand (session cookie, HTTP Basic and HTTP Digest) are well known to developers, and almost all browsers on all devices work with them out of the box. Frameworks and languages are ready for these methods, having built-in functions to deal with each seamlessly.

# JWT

JWT (shortened from JSON Web Token) is the missing standardization for using tokens to authenticate on the web in general, not only for REST services. Currently, it is in draft status as RFC 7519. It is robust and can carry a lot of information, but is still simple to use even though its size is relatively small. Like any other token, JWT can be used to pass the identity of authenticated users between an identity provider and a service provider (which are not necessarily the same systems). It can also carry all the user’s claim, such as authorization data, so the service provider does not need to go into the database or external systems to verify user roles and permissions for each request; that data is extracted from the token.

Here is how JWT is designed to work:




- Clients logs in by sending their credentials to the identity provider.
- The identity provider verifies the credentials; if all is OK, it retrieves the user data, generates a JWT containing user details and permissions that will be used to access the services, and it also sets the expiration on the JWT (which might be unlimited).
- Identity provider signs, and if needed, encrypts the JWT and sends it to the client as a response to the initial request with credentials.
- Client stores the JWT for a limited or unlimited amount of time, depending on the expiration set by the identity provider.
- Client sends the stored JWT in an Authorization header for every request to the service provider.
- For each request, the service provider takes the JWT from the `Authorization` header and decrypts it, if needed, validates the signature, and if everything is OK, extracts the user data and permissions. Based on this data solely, and again without looking up further details in the database or contacting the identity provider, it can accept or deny the client request. The only requirement is that the identity and service providers have an agreement on encryption so that service can verify the signature or even decrypt which identity was encrypted.

This flow allows for great flexibility while still keeping things secure and easy to develop. By using this approach, it is easy to add new server nodes to the service provider cluster, initializing them with only the ability to verify the signature and decrypt the tokens by providing them a shared secret key. No session replication, database synchronization or inter-node communication is required. REST in its full glory.

The main difference between JWT and other arbitrary tokens is the standardization of the token's content. Another recommended approach is to send the JWT token in the `Authorization` header using the Bearer scheme. The content of the header should look like this:

```
Authorization: Bearer <token>
```

# Implementation

For REST services to work as expected, we need a slightly different authorization approach compared to classic, multi-page websites.

Instead of triggering the authentication process by redirecting to a login page when a client requests a secured resource, the REST server authenticates all requests using the data available in the request itself, the JWT token in this case. If such an authentication fails, redirection makes no sense. The REST API simply sends an HTTP code 401 (Unauthorized) response and clients should know what to do; for example, a browser will show a dynamic div to allow the user to supply the username and password.

On the other hand, after a successful authentication in classic, multi-page websites, the user is redirected by using HTTP code 301 (Moved permanently), usually to a home page or, even better, to the page the user initially requested that triggered the authentication process. With REST, again this makes no sense. Instead we would simply continue with the execution of the request as if the resource was not secured at all, return HTTP code 200 (OK) and expected response body.

# Spring Security


Now, let's see how can we implement the JWT token based REST API using Java and Spring, while trying to reuse the Spring security default behavior where we can. As expected, Spring Security framework comes with many ready to plug-in classes that deal with "old" authorization mechanisms: session cookies, HTTP Basic, and HTTP Digest. However, it lacks the native support for JWT, and we need to get our hands dirty to make it work.

First, we start with the usual Spring Security filter definition in `web.xml`:

```
<filter>
        <filter-name>springSecurityFilterChain</filter-name>
        <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>
<filter-mapping>
        <filter-name>springSecurityFilterChain</filter-name>
        <url-pattern>/*</url-pattern>
</filter-mapping>
```

Note that the name of the filter must be exactly `springSecurityFilterChain` for the rest of the Spring config to work out of the box.

Next comes the XML declaration of the Spring beans related to security. In order to simplify the XML, we will set the default namespace to `security` by adding `xmlns="http://www.springframework.org/schema/security"` to the root XML element. The rest of the XML looks like this:

```
    <global-method-security pre-post-annotations="enabled" />  (1)

    <http pattern="/api/login" security="none"/>   (2)
    <http pattern="/api/signup" security="none"/>

    <http pattern="/api/**" entry-point-ref="restAuthenticationEntryPoint" create-session="stateless"> (3)
        <csrf disabled="true"/>  (4)
        <custom-filter before="FORM_LOGIN_FILTER" ref="jwtAuthenticationFilter"/>  (5)
    </http>

    <beans:bean id="jwtAuthenticationFilter" class="com.toptal.travelplanner.security.JwtAuthenticationFilter">  (6)
        <beans:property name="authenticationManager" ref="authenticationManager" />
        <beans:property name="authenticationSuccessHandler" ref="jwtAuthenticationSuccessHandler" />  (7)
    </beans:bean>

    <authentication-manager alias="authenticationManager">
        <authentication-provider ref="jwtAuthenticationProvider" />  (8)
    </authentication-manager>
```

- (1) In this line, we activate `@PreFilter`, `@PreAuthorize`, `@PostFilter`, `@PostAuthorize` annotations on any spring beans in the context.
- (2) We define the login and signup endpoints to skip security; even "anonymous" should be able to do these two operations.
- (3) Next, we define the filter chain applied to all requests while adding two important configs: Entry point reference and setting the session creation to `stateless` (we do not want the session created for security purposes as we are using tokens for each request).
- (4) We do not need `csrf` protection because our tokens are immune to it.
- (5) Next, we plug in our special authentication filter within the Spring's predefined filter chain, just before the form login filter.
- (6) This bean is the declaration of our authentification filter; since it is extending Spring's `AbstractAuthenticationProcessingFilter`, we need to declare it in XML to wire its properties (auto wire does not work here). We will explain later what the filter does.
- (7) The default success handler of `AbstractAuthenticationProcessingFilter` is not good enough for REST purposes because it redirects the user to a success page; that is why we set our own here.
- (8) The declaration of the provider created by the `authenticationManager` is used by our filter to authenticate users.

Now let's see how we implement the specific classes declared in the XML above. Note that Spring will wire them for us. We start with the simplest ones.

## RestAuthenticationEntryPoint.java

```
public class RestAuthenticationEntryPoint implements AuthenticationEntryPoint {

    @Override
    public void commence(HttpServletRequest request, HttpServletResponse response, AuthenticationException authException) throws IOException {
        // This is invoked when user tries to access a secured REST resource without supplying any credentials
        // We should just send a 401 Unauthorized response because there is no 'login page' to redirect to
        response.sendError(HttpServletResponse.SC_UNAUTHORIZED, "Unauthorized");
    }
}
```

As explained above, this class just returns HTTP code 401 (Unauthorized) when authentication fails, overriding default Spring's redirecting.



Like what you're reading?
Get the latest updates first.
Enter your email address...
Get Exclusive Updates
No spam. Just great engineering and design posts.
Like what you're reading?
Get the latest updates first.
Thank you for subscribing!
You can edit your subscription preferences here.

- 1.1Kshares
-
-
-

## JwtAuthenticationSuccessHandler.java

```
public class JwtAuthenticationSuccessHandler implements AuthenticationSuccessHandler {

    @Override
    public void onAuthenticationSuccess(HttpServletRequest request, HttpServletResponse response, Authentication authentication) {
        // We do not need to do anything extra on REST authentication success, because there is no page to redirect to
    }

}
```

This simple override removes the default behavior of a successful authentication (redirecting to home or any other page the user requested). If you are wondering why we do not need to override the `AuthenticationFailureHandler`, it is because default implementation will not redirect anywhere if its redirect URL is not set, so we just avoid setting the URL, which is good enough.

## JwtAuthenticationFilter.java

```
public class JwtAuthenticationFilter extends AbstractAuthenticationProcessingFilter {

    public JwtAuthenticationFilter() {
        super("/**");
    }

    @Override
    protected boolean requiresAuthentication(HttpServletRequest request, HttpServletResponse response) {
        return true;
    }

    @Override
    public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response) throws AuthenticationException {

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

        if (header == null || !header.startsWith("Bearer ")) {
            throw new JwtTokenMissingException("No JWT token found in request headers");
        }

        String authToken = header.substring(7);
```

```
        JwtAuthenticationToken authRequest = new JwtAuthenticationToken(authToken);

        return getAuthenticationManager().authenticate(authRequest);
    }

    @Override
    protected void successfulAuthentication(HttpServletRequest request, HttpServletResponse response, FilterChain chain, Authentication authRe
            throws IOException, ServletException {
        super.successfulAuthentication(request, response, chain, authResult);

        // As this authentication is in HTTP header, after success we need to continue the request normally
        // and return the response as if the resource was not secured at all
        chain.doFilter(request, response);
    }
}
```

This class is the entry point of our JWT authentication process; the filter extracts the JWT token from the request headers and delegates authentication to the injected `AuthenticationManager`. If the token is not found, an exception is thrown that stops the request from processing. We also need an override for successful authentication because the default Spring flow would stop the filter chain and proceed with a redirect. Keep in mind we need the chain to execute fully, including generating the response, as explained above.

## JwtAuthenticationProvider.java

```
public class JwtAuthenticationProvider extends AbstractUserDetailsAuthenticationProvider {

    @Autowired
    private JwtUtil jwtUtil;

    @Override
    public boolean supports(Class<?> authentication) {
        return (JwtAuthenticationToken.class.isAssignableFrom(authentication));
    }

    @Override
    protected void additionalAuthenticationChecks(UserDetails userDetails, UsernamePasswordAuthenticationToken authentication) throws Authenti
    }

    @Override
    protected UserDetails retrieveUser(String username, UsernamePasswordAuthenticationToken authentication) throws AuthenticationException {
        JwtAuthenticationToken jwtAuthenticationToken = (JwtAuthenticationToken) authentication;
        String token = jwtAuthenticationToken.getToken();

        User parsedUser = jwtUtil.parseToken(token);

        if (parsedUser == null) {
            throw new JwtTokenMalformedException("JWT token is not valid");
        }

        List<GrantedAuthority> authorityList = AuthorityUtils.commaSeparatedStringToAuthorityList(parsedUser.getRole());

        return new AuthenticatedUser(parsedUser.getId(), parsedUser.getUsername(), token, authorityList);
    }

}
```

In this class, we are using Spring's default `AuthenticationManager`, but we inject it with our own `AuthenticationProvider` that does the actual authentication process. To implement this, we extend the `AbstractUserDetailsAuthenticationProvider`, which requires us only to return `UserDetails` based on the authentication request, in our case, the JWT token wrapped in the `JwtAuthenticationToken` class. If the token is not valid, we throw an exception. However, if it is valid and decryption by `JwtUtil` is successful, we extract the user details (we will see exactly how in the `JwtUtil` class), without accessing the database at all. All the information about the user, including his or her roles, is contained in the token itself.

## JwtUtil.java

```
public class JwtUtil {

    @Value("${jwt.secret}")
    private String secret;

    /**
     * Tries to parse specified String as a JWT token. If successful, returns User object with username, id and role prefilled (extracted from
     * If unsuccessful (token is invalid or not containing all required user properties), simply returns null.
     *
     * @param token the JWT token to parse
     * @return the User object extracted from specified token or null if a token is invalid.
     */
    public User parseToken(String token) {
        try {
            Claims body = Jwts.parser()
                    .setSigningKey(secret)
                    .parseClaimsJws(token)
                    .getBody();

            User u = new User();
            u.setUsername(body.getSubject());
            u.setId(Long.parseLong((String) body.get("userId")));
            u.setRole((String) body.get("role"));

            return u;

        } catch (JwtException | ClassCastException e) {
            return null;
        }
    }

    /**
     * Generates a JWT token containing username as subject, and userId and role as additional claims. These properties are taken from the spe
     * User object. Tokens validity is infinite.
     *
     * @param u the user for which the token will be generated
```

```
     * @return the JWT token
     */
    public String generateToken(User u) {
        Claims claims = Jwts.claims().setSubject(u.getUsername());
        claims.put("userId", u.getId() + "");
        claims.put("role", u.getRole());

        return Jwts.builder()
                .setClaims(claims)
                .signWith(SignatureAlgorithm.HS512, secret)
                .compact();
    }
}
```

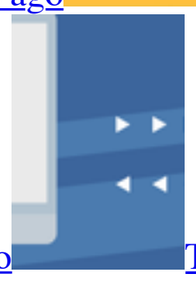
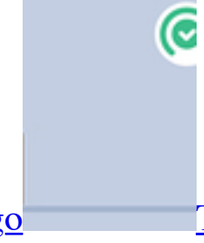
Finally, `JwtUtil` class is in charge of parsing the token into `User` object and generating the token from the `User` object. It is straightforward since it uses the [`jjwt` library](#) to do all the JWT work. In our example, we simply store the username, user ID and user roles in the token. We could also store more arbitrary stuff and add more security features, such as the token's expiration. Parsing of the token is used in the `AuthenticationProvider` as shown above. The `generateToken()` method is called from login and signup REST services, which are unsecured and will not trigger any security checks or require a token to be present in the request. In the end, it generates the token that will be returned to the clients, based on the user.

# Conclusion

Although the old, standardized security approaches (session cookie, HTTP Basic, and HTTP Digest) will work with REST services as well, they all have problems that would be nice to avoid by using a better standard. JWT arrives just in time to save the day, and most importantly it is very close to becoming an IETF standard.

JWT's main strength is handling user authentication in a stateless, and therefore scalable, way, while keeping everything secure with up-to-date cryptography standards. Storing claims (user roles and permissions) in the token itself creates huge benefits in distributed system architectures where the server that issues the request has no access to the authentication data source.

# About the author


The Definitive Guide to DateTime Manipulationabout 13 hours ago 
How Hibernate Almost Ruined My Career15 days ago 
How Sequel and Sinatra Solve Ruby’s API Problem21 days ago 
The 10 Most Common Mistakes That WordPress Developers Make26 days ago 
The Six Commandments of Good Code: Write Code that Stands the Test of Time28 days ago 
Meet RxJava: The Missing Reactive Programming Library for Androidabout 1 month ago 
Celebrating 25 Years of Linux Kernel Developmentabout 1 month ago 
Ten Kotlin Features To Boost Android Developmentabout 1 month ago

Relevant technologies

- Java
- Spring

About the author

Dejan Milosevic

JavaScript Developer

Dejan has many years of experience working with top Java and JavaScript frameworks. He is proficient in Spring/JEE, HTML, CSS, and several popular JavaScript libraries, not to mention a complimentary fluency in the DB layer. Dejan is a powerful addition to any team due to his competency with version control systems (SVN/Git) and internal development tools (Ant/Maven), as well as his results-driven and flexible mindset.

- Node.js Developers
- Objective-C Developers
- OpenGL Developers
- PHP Developers
- Python Developers
- React.js Developers
- Ruby Developers
- Ruby on Rails Developers
- Salesforce Developers
- Software Developers
- Unity or Unity3D Developers
- WordPress Developers

See more freelance developers

# Join the Toptal community.

Hire a developer
or
Apply as a Developer

## Highest In-Demand Talent

- iOS Developer
- Java Developer
- .NET Developer
- Front-End Developer
- UX Designer
- UI Designer

## About

- Top 3%
- Clients
- Freelance developers
- Freelance designers
- About Us

## Contact

- Contact Us
- Press Center
- Careers
- FAQ

## Social

- Facebook
- Twitter
- Google+
- LinkedIn

Toptal

Hire the top 3% of freelance talent



- Privacy Policy
- Website Terms