(/)

Supercharge Java Authentication with JSON Web Tokens (JWTs)

Last modified: October 1, 2021

by Micah Silverman (Stormpath)
(https://www.baeldung.com/author/micah-stormpath/)

Security (https://www.baeldung.com/category/security-2/)

JWT (https://www.baeldung.com/tag/jwt/)

Getting ready to build, or struggling with, secure authentication in your Java application? Unsure of the benefits of using tokens (and specifically JSON web tokens), or how they should be deployed? I'm excited to answer these questions, and more, for you in this tutorial!

Before we dive into JSON Web Tokens (JWTs (https://en.wikipedia.org/wiki/JSON_Web_Token)), and the JJWT library (https://github.com/jwtk/jjwt) (created by Stormpath's CTO, Les Hazlewood and maintained by a community of contributors (https://github.com/jwtk/jjwt/graphs/contributors)), let's cover some basics.

1. Authentication vs. Token Authentication

The set of protocols an application uses to confirm user identity is authentication. Applications have traditionally persisted identity through session cookies. This paradigm relies on server-side storage of session IDs

which forces developers to create session storage that is either unique and server-specific, or implemented as a completely separate session storage layer.

Token authentication was developed to solve problems server-side session IDs didn't, and couldn't. Just like traditional authentication, users present verifiable credentials, but are now issued a set of tokens instead of a session ID. The initial credentials could be the standard username/password pair, API keys, or even tokens from another service. (Stormpath's API Key Authentication Feature is an example of this.)

1.1. Why Tokens?

Very simply, using tokens in place of session IDs can lower your server load, streamline permission management, and provide better tools for supporting a distributed or cloud-based infrastructure. In the case of JWT, this is primarily accomplished through the stateless nature of these types of tokens (more on that below).

Tokens offer a wide variety of applications, including: Cross Site Request Forgery (CSRF (https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html)) protection schemes, OAuth 2.0 (https://tools.ietf.org/html/rfc6749) interactions, session IDs, and (in cookies) as authentication representations. In most cases, standards do not specify a particular format for tokens. Here's an example of a typical Spring Security CSRF token (https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/web/csrf/CsrfToken.html) in an HTML form:

```
<input name="_csrf" type="hidden"
  value="f3f42ea9-3104-4d13-84c0-7bcb68202f16"/>
```

If you try to post that form without the right CSRF token, you get an error response, and that's the utility of tokens. The above example is a “dumb” token. This means there is no inherent meaning to be gleaned from the token itself. This is also where JWTs make a big difference.

Read more (/rest-api-spring-oauth2-angular) →

OAuth2 for a Spring REST API – Handle the Refresh Token in Angular (/spring-security-oauth2-refresh-token-angular)

Have a look at how to refresh a token using the Spring Security 5 OAuth stack and leveraging a Zuul proxy.

Read more (/spring-security-oauth2-refresh-token-angular) →

2. What's in a JWT?

JWTs (pronounced "jots") are URL-safe, encoded, cryptographically signed (sometimes encrypted) strings that can be used as tokens in a variety of applications. Here's an example of a JWT being used as a CSRF token:

```
<input name="_csrf" type="hidden"

value="eyJhbGciOiJIUzI1NiJ9.eyJqdGkiOiJlNjc4ZjIzMzQ3ZTM0MTBkYjdlNjg3Njc4
MjNiMmQ3MCIsImlhdCI6MTQ2NjYzMzMxNywibmJmIjoxNDY2NjMzMzE3LCJleHAiOjE0NjY2
MzY5MTd9.rgx_o8VQGuDa2AqCHSgVOD5G68Ld_YYM7N7THmvLIKc"/>
```

In this case, you can see that the token is much longer than in our previous example. Just like we saw before, if the form is submitted without the token you get an error response.

So, why JWT?

The above token is cryptographically signed and therefore can be verified, providing proof that it hasn't been tampered with. Also, JWTs are encoded with a variety of additional information.

Let's look at the anatomy of a JWT to better understand how we squeeze all this goodness out of it. You may have noticed that there are three distinct sections separated by periods (`.`):

Header	eyJhbGciOiJIUzI1NiJ9

Payload	eyJqdGkiOiJlNjc4ZjIzMzQ3ZTM0MTBkYjdlNjg3Njc4MjNiMmQ3MCIsImlhdC I6MTQ2NjYzMzMxNywibmJmIjoxNDY2NjMzMzE3LCJleHAiOjE0NjY2MzY5MTd9
Signature	rgx_o8VQGuDa2AqCHSgVOD5G68Ld_YYM7N7THmvLIKc

Each section is base64 (https://en.wikipedia.org/wiki/Base64) URL-encoded. This ensures that it can be used safely in a URL (more on this later). Let's take a closer look at each section individually.

2.1. The Header

If you base64 to decode the header, you will get the following JSON string:

```
{"alg":"HS256"}
```

This shows that the JWT was signed with HMAC (https://en.wikipedia.org/wiki/Hash-based_message_authentication_code) using SHA-256 (https://en.wikipedia.org/wiki/SHA-2).

2.2. The Payload

If you decode the payload, you get the following JSON string (formatted for clarity):

```
{
  "jti": "e678f23347e3410db7e68767823b2d70",
  "iat": 1466633317,
  "nbf": 1466633317,
  "exp": 1466636917
}
```

Within the payload, as you can see, there are a number of keys with values. These keys are called “claims” and the JWT specification (https://tools.ietf.org/html/rfc7519#section-4.1) has seven of these specified as “registered” claims. They are:

iss	Issuer
sub	Subject
aud	Audience
exp	Expiration
nbf	Not Before
iat	Issued At
jti	JWT ID

When building a JWT, you can put in any custom claims you wish. The list above simply represents the claims that are reserved both in the key that is used and the expected type. Our CSRF has a JWT ID, an “Issued At” time, a

"Not Before" time, and an Expiration time. The expiration time is exactly one minute past the issued at time.

2.3. The Signature

Finally, the signature section is created by taking the header and payload together (with the . in between) and passing it through the specified algorithm (HMAC using SHA-256, in this case) along with a known secret. Note that the secret is *always* a byte array, and should be of a length that makes sense for the algorithm used. Below, I use a random base64 encoded string (for readability) that's converted into a byte array.

It looks like this in pseudo-code:

```
computeHMACSHA256(
    header + "." + payload,

base64DecodeToByteArray("4pE8z3PBoHjnV1AhvGk+e8h2p+ShZpOnpr8cwHmMh1w=")
)
```

As long as you know the secret, you can generate the signature yourself and compare your result to the signature section of the JWT to verify that it has not been tampered with. Technically, a JWT that's been cryptographically signed is called a JWS (https://tools.ietf.org/html/rfc7515). JWTs can also be encrypted and would then be called a JWE (https://tools.ietf.org/html/rfc7516). (In actual practice, the term JWT is used to describe JWEs and JWSs.)

This brings us back to the benefits of using a JWT as our CSRF token. We can verify the signature and we can use the information encoded in the JWT to confirm its validity. So, not only does the string representation of the JWT need to match what's stored server-side, we can ensure that it's not expired simply by inspecting the *exp* claim. This saves the server from maintaining additional state.

Well, we've covered a lot of ground here. Let's dive into some code!

3. Setup the JJWT Tutorial

JJWT (https://github.com/jwtk/jjwt (https://github.com/jwtk/jjwt)) is a Java library providing end-to-end JSON Web Token creation and verification. Forever free and open-source (Apache License, Version 2.0), it was designed with a builder-focused interface hiding most of its complexity.

The primary operations in using JJWT involve building and parsing JWTs. We'll look at these operations next, then get into some extended features of the JJWT, and finally, we'll see JWTs in action as CSRF tokens in a Spring Security, Spring Boot application.

The code demonstrated in the following sections can be found here (https://github.com/eugenp/tutorials/tree/master/jjwt). Note: The project uses Spring Boot from the beginning as its easy to interact with

the API that it exposes.

One of the great things about Spring Boot is how easy it is to build and fire up an application (/spring-boot-run-maven-vs-executable-jar). To run the JJWT Fun application, simply do the following:

```
mvn clean spring-boot:run
```

There are ten endpoints exposed in this example application (I use httpie to interact with the application. It can be found here (https://github.com/jkbrzt/httpie).)

```
http localhost:8080
```

```
Available commands (assumes httpie - https://github.com/jkbrzt/httpie):

  http http://localhost:8080/
        This usage message

  http http://localhost:8080/static-builder
        build JWT from hardcoded claims

  http POST http://localhost:8080/dynamic-builder-general claim-1=value-
1 ... [claim-n=value-n]
        build JWT from passed in claims (using general claims map)

  http POST http://localhost:8080/dynamic-builder-specific claim-
1=value-1 ... [claim-n=value-n]
        build JWT from passed in claims (using specific claims methods)

  http POST http://localhost:8080/dynamic-builder-compress claim-
1=value-1 ... [claim-n=value-n]
        build DEFLATE compressed JWT from passed in claims

  http http://localhost:8080/parser?jwt=<jwt>
        Parse passed in JWT

  http http://localhost:8080/parser-enforce?jwt=<jwt>
        Parse passed in JWT enforcing the 'iss' registered claim and the
'hasMotorcycle' custom claim

  http http://localhost:8080/get-secrets
        Show the signing keys currently in use.

  http http://localhost:8080/refresh-secrets
        Generate new signing keys and show them.

  http POST http://localhost:8080/set-secrets
    HS256=base64-encoded-value HS384=base64-encoded-value HS512=base64-
encoded-value
        Explicitly set secrets to use in the application.
```

In the sections that follow, we will examine each of these endpoints and the JJWT code contained in the handlers.

4. Building JWTs With JJWT

Because of JJWT's fluent interface (https://en.wikipedia.org/wiki/Fluent_interface), the creation of the JWT is basically a three-step process:

1. The definition of the internal claims of the token, like Issuer, Subject, Expiration, and ID.
2. The cryptographic signing of the JWT (making it a JWS).
3. The compaction of the JWT to a URL-safe string, according to the JWT Compact Serialization (https://tools.ietf.org/html/draft-ietf-jose-json-web-signature-41#section-3.1) rules.

The final JWT will be a three-part base64-encoded string, signed with the specified signature algorithm, and using the provided key. After this point, the token is ready to be shared with the another party.

Here's an example of the JJWT in action:

```
String jws = Jwts.builder()
  .setIssuer("Stormpath")
  .setSubject("msilverman")
  .claim("name", "Micah Silverman")
  .claim("scope", "admins")
  // Fri Jun 24 2016 15:33:42 GMT-0400 (EDT)
  .setIssuedAt(Date.from(Instant.ofEpochSecond(1466796822L)))
  // Sat Jun 24 2116 15:33:42 GMT-0400 (EDT)
  .setExpiration(Date.from(Instant.ofEpochSecond(4622470422L)))
  .signWith(
    SignatureAlgorithm.HS256,

TextCodec.BASE64.decode("Yn2kjibddFAWtnPJ2AFlL8WXmohJMCvigQggaEypa5E=")
  )
  .compact();
```

This is very similar to the code that's in the *StaticJWTController.fixedBuilder* method of the code project.

At this point, it's worth talking about a few anti-patterns related to JWTs and signing. If you've ever seen JWT examples before, you've likely encountered one of these signing anti-pattern scenarios:

1.

```
.signWith(
    SignatureAlgorithm.HS256,
   "secret".getBytes("UTF-8")
)
```

2.

```
.signWith(
    SignatureAlgorithm.HS256,
    "Yn2kjibddFAWtnPJ2AFlL8WXmohJMCvigQggaEypa5E=".getBytes("UTF-
8")
)
```

3.

```
.signWith(
    SignatureAlgorithm.HS512,

TextCodec.BASE64.decode("Yn2kjibddFAWtnPJ2AFlL8WXmohJMCvigQggaEypa5
E=")
)
```

Any of the *HS* type signature algorithms takes a byte array. It's convenient for humans to read to take a string and convert it to a byte array.

Anti-pattern 1 above demonstrates this. This is problematic because the secret is weakened by being so short and it's not a byte array in its native form. So, to keep it readable, we can base64 encode the byte array.

However, anti-pattern 2 above takes the base64 encoded string and converts it directly to a byte array. What should be done is to decode the base64 string back into the original byte array.

Number 3 above demonstrates this. So, why is this one also an anti-pattern? It's a subtle reason in this case. Notice that the signature algorithm is HS512. The byte array is not the maximum length that *HS512* can support, making it a weaker secret than what is possible for that algorithm.

The example code includes a class called *SecretService* that ensures secrets of the proper strength are used for the given algorithm. At application startup time, a new set of secrets is created for each of the HS algorithms. There are endpoints to refresh the secrets as well as to explicitly set the secrets.

If you have the project running as described above, execute the following so that the JWT examples below match the responses from your project.

```
http POST localhost:8080/set-secrets \
  HS256="Yn2kjibddFAWtnPJ2AFlL8WXmohJMCvigQggaEypa5E=" \

HS384="VW96zL+tYlrJLNCQ0j6QPTp+d1q75n/Wa8LVvpWyG8pPZOP6AA5X7XOIlI90sDwx"
\

HS512="cd+Pr1js+w2qfT2BoCD+tPcYp9LbjpmhSMEJqUob1mcxZ7+Wmik4AYdjX+DlDjmE4
yporzQ9tm7v3z/j+QbdYg=="
```

Now, you can hit the */static-builder* endpoint:

```
http http://localhost:8080/static-builder
```

This produces a JWT that looks like this:

```
eyJhbGciOiJIUzI1NiJ9.
eyJpc3MiOiJTdG9ybXBhdGgiLCJzdWIiOiJtc2lsdmVybWFuIiwibmFtZSI6Ik1pY2FoIFNp
bHZlcm1hbiIsInNjb3BlIjoiYWRtaW5zIiwiaWF0IjoxNDY2Nzk2ODIyLCJleHAiOjQ2MjI0
NzA0MjJ9.
kP0i_RvTAmI8mgpIkDFhRX3XthSdP-eqqFKGcU92ZIQ
```

Now, hit:

```
http http://localhost:8080/parser?
jwt=eyJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJTdG9ybXBhdGgiLCJzdWIiOiJtc2lsdmVybWF
uIiwibmFtZSI6Ik1pY2FoIFNpbHZlcm1hbiIsInNjb3BlIjoiYWRtaW5zIiwiaWF0IjoxNDY
2Nzk2ODIyLCJleHAiOjQ2MjI0NzA0MjJ9.kP0i_RvTAmI8mgpIkDFhRX3XthSdP-
eqqFKGcU92ZIQ
```

The response has all the claims that we included when we created the JWT.

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
...
{
    "jws": {
        "body": {
            "exp": 4622470422,
            "iat": 1466796822,
            "iss": "Stormpath",
            "name": "Micah Silverman",
            "scope": "admins",
            "sub": "msilverman"
        },
        "header": {
            "alg": "HS256"
        },
        "signature": "kP0i_RvTAmI8mgpIkDFhRX3XthSdP-eqqFKGcU92ZIQ"
    },
    "status": "SUCCESS"
}
```

This is the parsing operation, which we'll get into in the next section.

Now, let's hit an endpoint that takes claims as parameters and will build a custom JWT for us.

```
http -v POST localhost:8080/dynamic-builder-general iss=Stormpath
sub=msilverman hasMotorcycle:=true
```

Note: There's a subtle difference between the *hasMotorcycle* claim and the other claims. httpie assumes that JSON parameters are strings by default. To submit raw JSON using using httpie, you use the `:=` form rather than `=`. Without that, it would submit *"hasMotorcycle": "true"*, which is not what we want.

Here's the output:

```
POST /dynamic-builder-general HTTP/1.1
Accept: application/json
...
{
    "hasMotorcycle": true,
    "iss": "Stormpath",
    "sub": "msilverman"
}

HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
...
{
    "jwt":

"eyJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJTdG9ybXBhdGgiLCJzdWIiOiJtc2lsdmVybWFuIi
wiaGFzTW90b3JjeWNsZSI6dHJ1ZX0.OnyDs-zoL3-rw1GaSl_KzZzHK9GoiNocu-
YwZ_nQNZU",
    "status": "SUCCESS"
}
```

Let's take a look at the code that backs this endpoint:

```
@RequestMapping(value = "/dynamic-builder-general", method = POST)
public JwtResponse dynamicBuilderGeneric(@RequestBody Map<String,
Object> claims)
  throws UnsupportedEncodingException {
    String jws =  Jwts.builder()
        .setClaims(claims)
        .signWith(
            SignatureAlgorithm.HS256,
            secretService.getHS256SecretBytes()
        )
        .compact();
    return new JwtResponse(jws);
}
```

Line 2 ensures that the incoming JSON is automatically converted to a Java Map<String, Object>, which is super handy for JJWT as the method on line 5 simply takes that Map and sets all the claims at once.

As terse as this code is, we need something more specific to ensure that the claims that are passed are valid. Using the *.setClaims(Map<String, Object> claims)* method is handy when you already know that the claims represented in the map are valid. This is where the type-safety of Java comes into the JJWT library.

For each of the Registered Claims defined in the JWT specification, there's a corresponding Java method in the JJWT that takes the spec-correct type.

Let's hit another endpoint in our example and see what happens:

```
http -v POST localhost:8080/dynamic-builder-specific iss=Stormpath
sub:=5 hasMotorcycle:=true
```

Note that we've passed in an integer, 5, for the “sub” claim. Here's the output:

```
POST /dynamic-builder-specific HTTP/1.1
Accept: application/json
...
{
    "hasMotorcycle": true,
    "iss": "Stormpath",
    "sub": 5
}

HTTP/1.1 400 Bad Request
Connection: close
Content-Type: application/json;charset=UTF-8
...
{
    "exceptionType": "java.lang.ClassCastException",
    "message": "java.lang.Integer cannot be cast to java.lang.String",
    "status": "ERROR"
}
```

Now, we're getting an error response because the code is enforcing the type of the Registered Claims. In this case, *sub* must be a string. Here's the code that backs this endpoint:

```
@RequestMapping(value = "/dynamic-builder-specific", method = POST)
public JwtResponse dynamicBuilderSpecific(@RequestBody Map<String,
Object> claims)
  throws UnsupportedEncodingException {
    JwtBuilder builder = Jwts.builder();

    claims.forEach((key, value) -> {
        switch (key) {
            case "iss":
                builder.setIssuer((String) value);
                break;
            case "sub":
                builder.setSubject((String) value);
                break;
            case "aud":
                builder.setAudience((String) value);
                break;
            case "exp":
                builder.setExpiration(Date.from(

Instant.ofEpochSecond(Long.parseLong(value.toString()))
                ));
                break;
            case "nbf":
                builder.setNotBefore(Date.from(

Instant.ofEpochSecond(Long.parseLong(value.toString()))
                ));
                break;
            case "iat":
                builder.setIssuedAt(Date.from(

Instant.ofEpochSecond(Long.parseLong(value.toString()))
                ));
                break;
            case "jti":
                builder.setId((String) value);
                break;
            default:
                builder.claim(key, value);
        }
    });

    builder.signWith(SignatureAlgorithm.HS256,
secretService.getHS256SecretBytes());

    return new JwtResponse(builder.compact());
}
```

Just like before, the method accepts a *Map<String, Object>* of claims as its parameter. However, this time, we are calling the specific method for each of the Registered Claims which enforces type.

One refinement to this is to make the error message more specific. Right now, we only know that one of our claims is not the correct type. We don't know which claim was in error or what it should be. Here's a method that will give us a more specific error message. It also deals with a bug in the current code.

```
private void ensureType(String registeredClaim, Object value, Class
expectedType) {
    boolean isCorrectType =
        expectedType.isInstance(value) ||
        expectedType == Long.class && value instanceof Integer;

    if (!isCorrectType) {
        String msg = "Expected type: " + expectedType.getCanonicalName()
+
                     " for registered claim: '" + registeredClaim + "',
but got value: " +
                         value + " of type: " +
value.getClass().getCanonicalName();
        throw new JwtException(msg);
    }
}
```

Line 3 checks that the passed in value is of the expected type. If not, a *JwtException* is thrown with the specific error. Let's take a look at this in action by making the same call we did earlier:

```
http -v POST localhost:8080/dynamic-builder-specific iss=Stormpath
sub:=5 hasMotorcycle:=true
```

```
POST /dynamic-builder-specific HTTP/1.1
Accept: application/json
...
User-Agent: HTTPie/0.9.3

{
    "hasMotorcycle": true,
    "iss": "Stormpath",
    "sub": 5
}

HTTP/1.1 400 Bad Request
Connection: close
Content-Type: application/json;charset=UTF-8
...
{
    "exceptionType": "io.jsonwebtoken.JwtException",
    "message":
      "Expected type: java.lang.String for registered claim: 'sub', but
got value: 5 of type: java.lang.Integer",
    "status": "ERROR"
}
```

Now, we have a very specific error message telling us that the *sub* claim is the one in error.

Let's circle back to that bug in our code. The issue has nothing to do with the JJWT library. The issue is that the JSON to Java Object mapper built into Spring Boot is too smart for our own good.

If there's a method that accepts a Java Object, the JSON mapper will automatically convert a passed in number that is less than or equal to 2,147,483,647 into a Java *Integer*. Likewise, it will automatically convert a passed in number that is greater than 2,147,483,647 into a Java *Long*. For the

iat, *nbf*, and *exp* claims of a JWT, we want our ensureType test to pass whether the mapped Object is an Integer or a Long. That's why we have the additional clause in determining if the passed in value is the correct type:

```
boolean isCorrectType =
    expectedType.isInstance(value) ||
    expectedType == Long.class && value instanceof Integer;
```

If we're expecting a Long, but the value is an instance of Integer, we still say it's the correct type. With an understanding of what's happening with this validation, we can now integrate it into our *dynamicBuilderSpecific* method:

```
@RequestMapping(value = "/dynamic-builder-specific", method = POST)
public JwtResponse dynamicBuilderSpecific(@RequestBody Map<String,
Object> claims)
  throws UnsupportedEncodingException {
    JwtBuilder builder = Jwts.builder();

    claims.forEach((key, value) -> {
        switch (key) {
            case "iss":
                ensureType(key, value, String.class);
                builder.setIssuer((String) value);
                break;
            case "sub":
                ensureType(key, value, String.class);
                builder.setSubject((String) value);
                break;
            case "aud":
                ensureType(key, value, String.class);
                builder.setAudience((String) value);
                break;
            case "exp":
                ensureType(key, value, Long.class);
                builder.setExpiration(Date.from(

Instant.ofEpochSecond(Long.parseLong(value.toString()))
                                ));
                break;
            case "nbf":
                ensureType(key, value, Long.class);
                builder.setNotBefore(Date.from(

Instant.ofEpochSecond(Long.parseLong(value.toString()))
                                ));
                break;
            case "iat":
                ensureType(key, value, Long.class);
                builder.setIssuedAt(Date.from(

Instant.ofEpochSecond(Long.parseLong(value.toString()))
                                ));
                break;
            case "jti":
                ensureType(key, value, String.class);
                builder.setId((String) value);
                break;
            default:
                builder.claim(key, value);
        }
    });

    builder.signWith(SignatureAlgorithm.HS256,
secretService.getHS256SecretBytes());

    return new JwtResponse(builder.compact());
}
```

Note: In all the example code in this section, JWTs are signed with the HMAC using SHA-256 algorithm. This is to keep the examples simple. The JJWT library supports 12 different signature algorithms that you can take advantage of in your own code.

5. Parsing JWTs With JJWT

We saw earlier that our code example has an endpoint for parsing a JWT. Hitting this endpoint:

```
http http://localhost:8080/parser?
jwt=eyJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJTdG9ybXBhdGgiLCJzdWIiOiJtc2lsdmVybWF
uIiwibmFtZSI6Ik1pY2FoIFNpbHZlcm1hbiIsInNjb3BlIjoiYWRtaW5zIiwiaWF0IjoxNDY
2Nzk2ODIyLCJleHAiOjQ2MjI0NzA0MjJ9.kP0i_RvTAmI8mgpIkDFhRX3XthSdP-
eqqFKGcU92ZIQ
```

produces this response:

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
...
{
    "claims": {
        "body": {
            "exp": 4622470422,
            "iat": 1466796822,
            "iss": "Stormpath",
            "name": "Micah Silverman",
            "scope": "admins",
            "sub": "msilverman"
        },
        "header": {
            "alg": "HS256"
        },
        "signature": "kP0i_RvTAmI8mgpIkDFhRX3XthSdP-eqqFKGcU92ZIQ"
    },
    "status": "SUCCESS"
}
```

The *parser* method of the *StaticJWTController* class looks like this:

```
@RequestMapping(value = "/parser", method = GET)
public JwtResponse parser(@RequestParam String jwt) throws
UnsupportedEncodingException {
    Jws<Claims> jws = Jwts.parser()
        .setSigningKeyResolver(secretService.getSigningKeyResolver())
        .parseClaimsJws(jwt);

    return new JwtResponse(jws);
}
```

Line 4 indicates that we expect the incoming string to be a signed JWT (a JWS). And, we are using the same secret that was used to sign the JWT in parsing it. Line 5 parses the claims from the JWT. Internally, it is verifying the signature and it will throw an exception if the signature is invalid.

Notice that in this case we are passing in a *SigningKeyResolver* rather than a key itself. This is one of the most powerful aspects of JJWT. The header of JWT indicates the algorithm used to sign it. However, we need to verify the JWT before we trust it. It would seem to be a catch 22. Let's look at the *SecretService.getSigningKeyResolver* method:

```
private SigningKeyResolver signingKeyResolver = new
SigningKeyResolverAdapter() {
    @Override
    public byte[] resolveSigningKeyBytes(JwsHeader header, Claims
claims) {
        return
TextCodec.BASE64.decode(secrets.get(header.getAlgorithm()));
    }
};
```

Using the access to the *JwsHeader*, I can inspect the algorithm and return the proper byte array for the secret that was used to sign the JWT. Now, JJWT will verify that the JWT has not been tampered with using this byte array as the key.

If I remove the last character of the passed in JWT (which is part of the signature), this is the response:

```
HTTP/1.1 400 Bad Request
Connection: close
Content-Type: application/json;charset=UTF-8
Date: Mon, 27 Jun 2016 13:19:08 GMT
Server: Apache-Coyote/1.1
Transfer-Encoding: chunked

{
    "exceptionType": "io.jsonwebtoken.SignatureException",
    "message":
      "JWT signature does not match locally computed signature. JWT
validity cannot be asserted and should not be trusted.",
    "status": "ERROR"
}
```

6. JWTs in Practice: Spring Security CSRF Tokens

While the focus of this post is not Spring Security, we are going to delve into it a bit here to showcase some real-world usage of the JJWT library.

Cross Site Request Forgery (https://en.wikipedia.org/wiki/Cross-site_request_forgery) is a security vulnerability whereby a malicious website tricks you into submitting requests to a website that you have established trust with. One of the common remedies for this is to implement a synchronizer token pattern (https://docs.spring.io/spring-security/site/docs/5.0.x/reference/html/csrf.html). This approach inserts a token into the web form and the application server checks the incoming token against its repository to confirm that it is correct. If the token is missing or invalid, the server will respond with an error.

Spring Security has the synchronizer token pattern built in. Even better, if you are using the Spring Boot and Thymeleaf templates (https://spring.io/guides/gs/serving-web-content/), the synchronizer token is automatically inserted for you.

By default, the token that Spring Security uses is a “dumb” token. It's just a series of letters and numbers. This approach is just fine and it works. In this section, we enhance the basic functionality by using JWTs as the token. In addition to verifying that the submitted token is the one expected, we validate the JWT to further prove that the token has not been tampered with and to ensure that it is not expired.

To get started, we are going to configure Spring Security using Java configuration. By default, all paths require authentication and all POST endpoints require CSRF tokens. We are going to relax that a bit so that what we've built so far still works.

```
@Configuration
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

    private String[] ignoreCsrfAntMatchers = {
        "/dynamic-builder-compress",
        "/dynamic-builder-general",
        "/dynamic-builder-specific",
        "/set-secrets"
    };

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .csrf()
                .ignoringAntMatchers(ignoreCsrfAntMatchers)
            .and().authorizeRequests()
                .antMatchers("/**")
                .permitAll();
    }
}
```

We are doing two things here. First, we are saying the CSRF tokens are *not* required when posting to our REST API endpoints (line 15). Second, we are saying that unauthenticated access should be allowed for all paths (lines 17 – 18).

Let's confirm that Spring Security is working the way we expect. Fire up the app and hit this url in your browser:

```
http://localhost:8080/jwt-csrf-form
```

Here's the Thymeleaf template for this view:

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
    <head>
        <!--/*/ <th:block th:include="fragments/head :: head"/> /*/-->
    </head>
    <body>
        <div class="container-fluid">
            <div class="row">
                <div class="box col-md-6 col-md-offset-3">
                    <p/>
                    <form method="post" th:action="@{/jwt-csrf-form}">
                        <input type="submit" class="btn btn-primary"
value="Click Me!"/>
                    </form>
                </div>
            </div>
        </div>
    </body>
</html>
```

This is a very basic form that will POST to the same endpoint when submitted. Notice that there is no explicit reference to CSRF tokens in the form. If you view the source, you will see something like:

```
<input type="hidden" name="_csrf" value="5f375db2-4f40-4e72-9907-
a290507cb25e" />
```

This is all the confirmation you need to know that Spring Security is functioning and that the Thymeleaf templates are automatically inserting the CSRF token.

To make the value a JWT, we will enable a custom *CsrfTokenRepository*. Here's how our Spring Security configuration changes:

```
@Configuration
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    CsrfTokenRepository jwtCsrfTokenRepository;

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .csrf()
                .csrfTokenRepository(jwtCsrfTokenRepository)
                .ignoringAntMatchers(ignoreCsrfAntMatchers)
            .and().authorizeRequests()
                .antMatchers("/**")
                .permitAll();
    }
}
```

To connect this, we need a configuration that exposes a bean that returns the custom token repository. Here's the configuration:

```
@Configuration
public class CSRFConfig {

    @Autowired
    SecretService secretService;

    @Bean
    @ConditionalOnMissingBean
    public CsrfTokenRepository jwtCsrfTokenRepository() {
        return new
JWTCsrfTokenRepository(secretService.getHS256SecretBytes());
    }
}
```

And, here's our custom repository (the important bits):

```
public class JWTCsrfTokenRepository implements CsrfTokenRepository {

    private static final Logger log =
LoggerFactory.getLogger(JWTCsrfTokenRepository.class);
    private byte[] secret;

    public JWTCsrfTokenRepository(byte[] secret) {
        this.secret = secret;
    }

    @Override
    public CsrfToken generateToken(HttpServletRequest request) {
        String id = UUID.randomUUID().toString().replace("-", "");

        Date now = new Date();
        Date exp = new Date(System.currentTimeMillis() + (1000*30)); //
30 seconds

        String token;
        try {
            token = Jwts.builder()
                .setId(id)
                .setIssuedAt(now)
                .setNotBefore(now)
                .setExpiration(exp)
                .signWith(SignatureAlgorithm.HS256, secret)
                .compact();
        } catch (UnsupportedEncodingException e) {
            log.error("Unable to create CSRf JWT: {}", e.getMessage(),
e);
            token = id;
        }

        return new DefaultCsrfToken("X-CSRF-TOKEN", "_csrf", token);
    }

    @Override
    public void saveToken(CsrfToken token, HttpServletRequest request,
HttpServletResponse response) {
        ...
    }

    @Override
    public CsrfToken loadToken(HttpServletRequest request) {
        ...
    }
}
```

The *generateToken* method creates a JWT that expires 30 seconds after it's created. With this plumbing in place, we can fire up the application again and look at the source of */jwt-csrf-form*.

Now, the hidden field looks like this:

```
<input type="hidden" name="_csrf"

value="eyJhbGciOiJIUzI1NiJ9.eyJqdGkiOiIxZjIyMDdiNTVjOWM0MjU0YjZlMjY4MjQw
YjIwNzZkMSIsImlhdCI6MTQ2NzA3MDQwMCwibmJmIjoxNDY3MDcwNDAwLCJleHAiOjE0Njcw
NzA0MzB9.2kYLO0iMWUheAncXAzm0UdQC1xUC5I6RI_ShJ_74e5o" />
```

Huzzah! Now our CSRF token is a JWT. That wasn't too hard.

However, this is only half the puzzle. By default, Spring Security simply saves the CSRF token and confirms that the token submitted in a web form matches the one that's saved. We want to extend the functionality to validate the JWT and make sure it hasn't expired. To do that, we'll add in a filter. Here's what our Spring Security configuration looks like now:

```
@Configuration
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

    ...

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .addFilterAfter(new JwtCsrfValidatorFilter(),
CsrfFilter.class)
            .csrf()
                .csrfTokenRepository(jwtCsrfTokenRepository)
                .ignoringAntMatchers(ignoreCsrfAntMatchers)
            .and().authorizeRequests()
                .antMatchers("/**")
                .permitAll();
    }

    ...
}
```

On line 9, we've added in a filter and we are placing it in the filter chain after the default *CsrfFilter*. So, by the time our filter is hit, the JWT token (as a whole) will have already been confirmed to be the correct value saved by Spring Security.

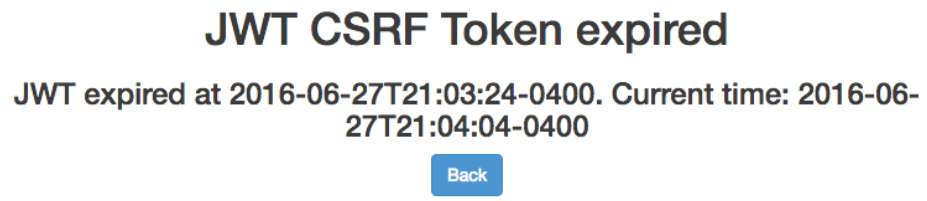
Here's the *JwtCsrfValidatorFilter* (it's private as it's an inner class of our Spring Security configuration):

```
private class JwtCsrfValidatorFilter extends OncePerRequestFilter {

    @Override
    protected void doFilterInternal(
      HttpServletRequest request,
      HttpServletResponse response,
      FilterChain filterChain) throws ServletException, IOException {
        // NOTE: A real implementation should have a nonce cache so the
token cannot be reused
        CsrfToken token = (CsrfToken) request.getAttribute("_csrf");

        if (
            // only care if it's a POST
            "POST".equals(request.getMethod()) &&
            // ignore if the request path is in our list
            Arrays.binarySearch(ignoreCsrfAntMatchers,
request.getServletPath()) < 0 &&
            // make sure we have a token
            token != null
        ) {
            // CsrfFilter already made sure the token matched.
            // Here, we'll make sure it's not expired
            try {
                Jwts.parser()
                    .setSigningKey(secret.getBytes("UTF-8"))
                    .parseClaimsJws(token.getToken());
            } catch (JwtException e) {
                // most likely an ExpiredJwtException, but this will
handle any
                request.setAttribute("exception", e);
                response.setStatus(HttpServletResponse.SC_BAD_REQUEST);
                RequestDispatcher dispatcher =
request.getRequestDispatcher("expired-jwt");
                dispatcher.forward(request, response);
            }
        }

        filterChain.doFilter(request, response);
    }
}
```

Take a look at line 23 on. We are parsing the JWT as before. In this case, if an Exception is thrown, the request is forwarded to the *expired-jwt* template. If the JWT validates, then processing continues as normal.

This closes the loop on overriding the default Spring Security CSRF token behavior with a JWT token repository and validator.

If you fire up the app, browse to */jwt-csrf-form*, wait a little more than 30 seconds and click the button, you will see something like this:



(/wp-content/uploads/2016/08/jwt_expired.png)

7. JJWT Extended Features

We'll close out our JJWT journey with a word on some of the features that extend beyond the specification.

7.1. Enforce Claims

As part of the parsing process, JJWT allows you to specify required claims and values those claims should have. This is very handy if there is certain information in your JWTs that must be present in order for you to consider them valid. It avoids a lot of branching logic to manually validate claims. Here's the method that serves the */parser-enforce* endpoint of our sample project.

```
@RequestMapping(value = "/parser-enforce", method = GET)
public JwtResponse parserEnforce(@RequestParam String jwt)
  throws UnsupportedEncodingException {
    Jws<Claims> jws = Jwts.parser()
        .requireIssuer("Stormpath")
        .require("hasMotorcycle", true)
        .setSigningKeyResolver(secretService.getSigningKeyResolver())
        .parseClaimsJws(jwt);

    return new JwtResponse(jws);
}
```

Lines 5 and 6 show you the syntax for registered claims as well as custom claims. In this example, the JWT will be considered invalid if the iss claim is not present or does not have the value: Stormpath. It will also be invalid if the custom hasMotorcycle claim is not present or does not have the value: true.

Let's first create a JWT that follows the happy path:

```
http -v POST localhost:8080/dynamic-builder-specific \
  iss=Stormpath hasMotorcycle:=true sub=msilverman
```

```
POST /dynamic-builder-specific HTTP/1.1
Accept: application/json
...
{
    "hasMotorcycle": true,
    "iss": "Stormpath",
    "sub": "msilverman"
}

HTTP/1.1 200 OK
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Content-Type: application/json;charset=UTF-8
...
{
    "jwt":

"eyJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJTdG9ybXBhdGgiLCJoYXNNb3RvcmN5Y2xlIjp0cn
VlLCJzdWIiOiJtc2lsdmVybWFuIn0.qrH-
U6TLSVlHkZdYuqPRDtgKNr1RilFYQJtJbcgwhR0",
    "status": "SUCCESS"
}
```

Now, let's validate that JWT:

```
http -v localhost:8080/parser-enforce?
jwt=eyJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJTdG9ybXBhdGgiLCJoYXNNb3RvcmN5Y2xlIjp
0cnVlLCJzdWIiOiJtc2lsdmVybWFuIn0.qrH-
U6TLSVlHkZdYuqPRDtgKNr1RilFYQJtJbcgwhR0
```

```
GET /parser-enforce?jwt=http
  -v localhost:8080/parser-enforce?
jwt=eyJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJTdG9ybXBhdGgiLCJoYXNNb3RvcmN5Y2xlIjp
0cnVlLCJzdWIiOiJtc2lsdmVybWFuIn0.qrH-
U6TLSVlHkZdYuqPRDtgKNr1RilFYQJtJbcgwhR0 HTTP/1.1
Accept: */*
...
HTTP/1.1 200 OK
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Content-Type: application/json;charset=UTF-8
...
{
    "jws": {
        "body": {
            "hasMotorcycle": true,
            "iss": "Stormpath",
            "sub": "msilverman"
        },
        "header": {
            "alg": "HS256"
        },
        "signature": "qrH-U6TLSVlHkZdYuqPRDtgKNr1RilFYQJtJbcgwhR0"
    },
    "status": "SUCCESS"
}
```

So far, so good. Now, this time, let's leave the hasMotorcycle out:

```
http -v POST localhost:8080/dynamic-builder-specific iss=Stormpath
sub=msilverman
```

This time, if we try to validate the JWT:

```
http -v localhost:8080/parser-enforce?
jwt=eyJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJTdG9ybXBhdGgiLCJzdWIiOiJtc2lsdmVybWF
uIn0.YMONlFM1tNgttUYukDRsi9gKIocxdGAOLaJBymaQAWc
```

we get:

```
GET /parser-enforce?jwt=http -v localhost:8080/parser-enforce?
jwt=eyJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJTdG9ybXBhdGgiLCJzdWIiOiJtc2lsdmVybWF
uIn0.YMONlFM1tNgttUYukDRsi9gKIocxdGAOLaJBymaQAWc HTTP/1.1
Accept: */*
...
HTTP/1.1 400 Bad Request
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Connection: close
Content-Type: application/json;charset=UTF-8
...
{
    "exceptionType": "io.jsonwebtoken.MissingClaimException",
    "message":
      "Expected hasMotorcycle claim to be: true, but was not present in
the JWT claims.",
    "status": "ERROR"
}
```

This indicates that our hasMotorcycle claim was expected, but was missing.

Let's do one more example:

```
http -v POST localhost:8080/dynamic-builder-specific iss=Stormpath
hasMotorcycle:=false sub=msilverman
```

This time, the required claim is present, but it has the wrong value. Let's see the output of:

```
http -v localhost:8080/parser-enforce?
jwt=eyJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJTdG9ybXBhdGgiLCJoYXNNb3RvcmN5Y2xlIjp
mYWxzZSwic3ViIjoibXNpbHZlcm1hbiJ9.8LBq2f0eINB34AzhVEgsln_KDo-IyeM8kc-
dTzSCr0c
```

```
GET /parser-enforce?jwt=http
  -v localhost:8080/parser-enforce?
jwt=eyJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJTdG9ybXBhdGgiLCJoYXNNb3RvcmN5Y2xlIjp
mYWxzZSwic3ViIjoibXNpbHZlcm1hbiJ9.8LBq2f0eINB34AzhVEgsln_KDo-IyeM8kc-
dTzSCr0c HTTP/1.1
Accept: */*
...
HTTP/1.1 400 Bad Request
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Connection: close
Content-Type: application/json;charset=UTF-8
...
{
    "exceptionType": "io.jsonwebtoken.IncorrectClaimException",
    "message": "Expected hasMotorcycle claim to be: true, but was:
false.",
    "status": "ERROR"
}
```

This indicates that our hasMotorcycle claim was present, but had a value that was not expected.

MissingClaimException and *IncorrectClaimException* are your friends when enforcing claims in your JWTs and a feature that only the JJWT library has.

7.2. JWT Compression

If you have a lot of claims on a JWT, it can get big – so big, that it might not fit in a GET url in some browsers.

Let's a make a big JWT:

```
http -v POST localhost:8080/dynamic-builder-specific \
  iss=Stormpath hasMotorcycle:=true sub=msilverman the=quick brown=fox
jumped=over lazy=dog \
  somewhere=over rainbow=way up=high and=the dreams=you dreamed=of
```

Here's the JWT that produces:

```
eyJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJTdG9ybXBhdGgiLCJoYXNNb3RvcmN5Y2xlIjp0cnV
lLCJzdWIiOiJtc2lsdmVybWFuIiwidGhlIjoicXVpY2siLCJicm93biI6ImZveCIsImp1bXB
lZCI6Im92ZXIiLCJsYXp5IjoiZG9nIiwic29tZXdoZXJlIjoib3ZlciIsInJhaW5ib3ciOiJ
3YXkiLCJ1cCI6ImhpZ2giLCJhbmQiOiJ0aGUiLCJkcmVhbXMiOiJ5b3UiLCJkcmVhbWVkIjo
ib2YifQ.AHNJxSTiDw_bWNXcuh-LtPLvSjJqwDvOOUcmkk7CyZA
```

That sucker's big! Now, let's hit a slightly different endpoint with the same claims:

```
http -v POST localhost:8080/dynamic-builder-compress \
  iss=Stormpath hasMotorcycle:=true sub=msilverman the=quick brown=fox
jumped=over lazy=dog \
  somewhere=over rainbow=way up=high and=the dreams=you dreamed=of
```

This time, we get:

```
eyJhbGciOiJIUzI1NiIsImNhbGciOiJERUYifQ.eNpEzkESwjAIBdC7sO4JegdXnoC2tIk2o
ZLEGB3v7s84jjse_AFe5FOikc5ZLRycHQ3kOJ0Untu8C43ZigyUyoRYSH6_iwWOyGWHKd2Kn
6_QZFojvOoDupRwyAIq4vDOzwYtugFJg1QnJv-5sY-TVjQqN7gcKJ3f-j8c-6J-
baDFhEN_uGn58XtnpfcHAAD__w.3_wc-2skFBbInk0YAQ96yGWwr8r1xVdbHn-uGPTFuFE
```

62 characters shorter! Here's the code for the method used to generate the JWT:

```
@RequestMapping(value = "/dynamic-builder-compress", method = POST)
public JwtResponse dynamicBuildercompress(@RequestBody Map<String,
Object> claims)
  throws UnsupportedEncodingException {
    String jws =  Jwts.builder()
        .setClaims(claims)
        .compressWith(CompressionCodecs.DEFLATE)
        .signWith(
            SignatureAlgorithm.HS256,
            secretService.getHS256SecretBytes()
        )
        .compact();
    return new JwtResponse(jws);
}
```

Notice on line 6 we are specifying a compression algorithm to use. That's all there is to it.

What about parsing compressed JWTs? The JJWT library automatically detects the compression and uses the same algorithm to decompress:

```
GET /parser?
jwt=eyJhbGciOiJIUzI1NiIsImNhbGciOiJERUYifQ.eNpEzkESwjAIBdC7sO4JegdXnoC2t
Ik2oZLEGB3v7s84jjse_AFe5FOikc5ZLRycHQ3kOJ0Untu8C43ZigyUyoRYSH6_iwWOyGWHK
d2Kn6_QZFojvOoDupRwyAIq4vDOzwYtugFJg1QnJv-5sY-TVjQqN7gcKJ3f-j8c-6J-
baDFhEN_uGn58XtnpfcHAAD__w.3_wc-2skFBbInk0YAQ96yGWwr8r1xVdbHn-uGPTFuFE
HTTP/1.1
Accept: */*
...
HTTP/1.1 200 OK
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Content-Type: application/json;charset=UTF-8
...
{
    "claims": {
        "body": {
            "and": "the",
            "brown": "fox",
            "dreamed": "of",
            "dreams": "you",
            "hasMotorcycle": true,
            "iss": "Stormpath",
            "jumped": "over",
            "lazy": "dog",
            "rainbow": "way",
            "somewhere": "over",
            "sub": "msilverman",
            "the": "quick",
            "up": "high"
        },
        "header": {
            "alg": "HS256",
            "calg": "DEF"
        },
        "signature": "3_wc-2skFBbInk0YAQ96yGWwr8r1xVdbHn-uGPTFuFE"
    },
    "status": "SUCCESS"
}
```

Notice the *calg* claim in the header. This was automatically encoded into the JWT and it provides the hint to the parser about what algorithm to use for decompression.

NOTE: The JWE specification does support compression. In an upcoming release of the JJWT library, we will support JWE and compressed JWEs. We will continue to support compression in other types of JWTs, even though it is not specified.

8. Token Tools for Java Devs

While the core focus of this article was not Spring Boot or Spring Security, using those two technologies made it easy to demonstrate all the features discussed in this article. You should be able to build in fire up the server and start playing with the various endpoints we've discussed. Just hit:

```
http http://localhost:8080
```

Stormpath (https://stormpath.com) is also excited to bring a number of open source developer tools to the Java community. These include:

8.1. JJWT (What We've Been Talking About)

JJWT (https://github.com/jwtk/jjwt) is an easy to use tool for developers to create and verify JWTs in Java (https://stormpath.com/blog/jjwt-how-it-works-why). Like many libraries Stormpath supports, JJWT is completely free and open source (Apache License, Version 2.0), so everyone can see what it does and how it does it. Do not hesitate to report any issues, suggest improvements, and even submit some code!

8.2. jsonwebtoken.io and java.jsonwebtoken.io

jsonwebtoken.io (http://jsonwebtoken.io) is a developer tool we created to make it easy to decode JWTs. Simply paste an existing JWT into the appropriate field to decode its header, payload, and signature. jsonwebtoken.io is powered by nJWT (https://github.com/jwtk/njwt), the cleanest free and open source (Apache License, Version 2.0) JWT library for Node.js developers. You can also see code generated for a variety of languages at this website. The website itself is open-source and can be found here (https://github.com/stormpath/jsonwebtoken.io).

java.jsonwebtoken.io (http://java.jsonwebtoken.io) is specifically for the JJWT library. You can alter the headers and payload in the upper right box, see the JWT generated by JJWT in the upper left box, and see a sample of the builder and parser Java code in the lower boxes. The website itself is open source and can be found here (https://github.com/stormpath/JJWTsite).

8.3. JWT Inspector

The new kid on the block, JWT Inspector (https://www.jwtinspector.io/) is an open source Chrome extension that allows developers to inspect and debug JWTs directly in-browser. The JWT Inspector will discover JWTs on your site (in cookies, local/session storage, and headers) and make them easily accessible through your navigation bar and DevTools panel.

9. JWT This Down!

JWTs add some intelligence to ordinary tokens. The ability to cryptographically sign and verify, build in expiration times and encode other information into JWTs sets the stage for truly stateless session management. This has a big impact on the ability to scale applications.

At Stormpath, we use JWTs for OAuth2 tokens, CSRF tokens and assertions between microservices, among other usages.

Once you start using JWTs, you may never go back to the dumb tokens of the past. Have any questions? Hit me up at @afitnerd (https://twitter.com/afitnerd) on twitter.

Get access to the video lesson (/security-video-guide)

12 COMMENTS

Oldest

View Comments

Comments are closed on this article!

COURSES

ALL COURSES (/ALL-COURSES)
ALL BULK COURSES (/ALL-BULK-COURSES)
THE COURSES PLATFORM (HTTPS://COURSES.BAELDUNG.COM)

SERIES

JAVA "BACK TO BASICS" TUTORIAL (/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (/JACKSON)
HTTPCLIENT 4 TUTORIAL (/HTTPCLIENT-GUIDE)
REST WITH SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)
SPRING PERSISTENCE TUTORIAL (/PERSISTENCE-WITH-SPRING-SERIES)
SECURITY WITH SPRING (/SECURITY-SPRING)
SPRING REACTIVE TUTORIALS (/SPRING-REACTIVE-GUIDE)

ABOUT

ABOUT BAELDUNG (/ABOUT)
THE FULL ARCHIVE (/FULL_ARCHIVE)
WRITE FOR BAELDUNG (/CONTRIBUTION-GUIDELINES)
EDITORS (/EDITORS)
JOBS (/TAG/ACTIVE-JOB/)
OUR PARTNERS (/PARTNERS)
ADVERTISE ON BAELDUNG (/ADVERTISE)