Stateless Spring Security - Stateless Authentication

When talking about Authentication, its all about having the client identify itself to the server in a verifiable manner. Typically this start with the server providing the client with a challenge, like a request to fill in a username / password. Today I want to focus on what happens after passing such initial (manual) challenge and how to deal with automatic re-authentication of futher HTTP requests.

## **Common approaches**

### **Session Cookie based**

The most common approach we probably all know is to use a server generated secret token (Session key) in the form of a JSESSIONID cookie. Initial setup for this is near nothing these days perhaps making you forget you have a choice to make here in the first place. Even without further using this “Session key” to store any other state “in the session”, the key itself is in fact state as well.  I.e. without a shared and persistent storage of these keys, no successful authentication will survive a server reboot or requests being load balanced to another server.

### **OAuth2 / API keys**

Whenever talking about REST APIs and Security; OAuth2 and other types of API keys are mentioned. Basically they involve sending custom tokens/keys within the HTTP Authorization header. When used properly both relieve clients from dealing with Cookies using the header instead. This solves CSRF vulnerabilities and other Cookie related issues. One thing they do not solve however is the need for the server to check the presented authentication keys, pretty much demanding some persistent and maintainable shared storage for linking the keys to users/authorizations.

## **Stateless approaches**

### **1. HTTP Basis Auth**

The oldest and most crude way of dealing with authentication. Simply have the user send its username/password with every request. This probably sounds horrible, but considering any of the approaches mentioned above also send secret keys over the wire, this isn’t really all that less secure at all. Its mainly the user experience and flexibility that makes the other approaches a better choice.

### **2. Server signed tokens**

A neat little trick to dealing with state across requests in a stateless way is to have the server “sign” it. It can then be transported back and forth between the client/server each request with the guarantee that it is not tempered with. This way any user identification data can be shared in plain-text, adding a special signing hash to it. Considering it is signed, the server can simply validate if the signing hash still matches the received content, without needing to hold any server-side state.

The common standard that can be used for this is [JSON Web Tokens](http://self-issued.info/docs/draft-ietf-oauth-json-web-token.html) (JWT) which is still in draft. For this blog post I’d like to get down and dirty though, skipping full compliance and the scream for using a library that comes with it. Picking just what we actually need from it. (Leaving out the header/variable hash algoritms and url-safe base64 encoding)

#### **Implementation**

As mentioned we’re going to roll our own implementation, using  Spring Security and Spring Boot to plug it all together. Without any library or fancy API obfuscating what’s really happening on the token level. The token is going to look like this in pseudo-code:

content = toJSON(user\_details)

token = BASE64(content) + "." + BASE64(HMAC(content))

The dot in the token serves as a separator, so each part can be identified and decoded separately as the dot character is not part of any base64 encoded string. The [HMAC](http://en.wikipedia.org/wiki/Hash-based_message_authentication_code) stands for a Hash-based Message Authentication Code, which is basically a hash made from any data using a predefined secret key.

In actual Java the generation of the token looks a lot like the pseudo-code:

create token

public String createTokenForUser(User user) {

byte[] userBytes = toJSON(user);

byte[] hash = createHmac(userBytes);

final StringBuilder sb = new StringBuilder(170);

sb.append(toBase64(userBytes));

sb.append(SEPARATOR);

sb.append(toBase64(hash));

return sb.toString();

}

The relevant User properties used in the JSON are **id, username, expires and roles**, but could be anything you want really. I marked the “password” property of the User object to be ignored during jackson JSON serialization so it does not become part of the token:

Ignore password

@JsonIgnore

public String getPassword() {

return password;

}

For real worlds scenarios you probably just want to use a dedicated object for this.

The decoding of the token is a bit more complex with some input validation to prevent/catch parsing errors due to tempering with the token:

decode the token

public User parseUserFromToken(String token) {

final String[] parts = token.split(SEPARATOR\_SPLITTER);

if (parts.length == 2 && parts[0].length() > 0 && parts[1].length() > 0) {

try {

final byte[] userBytes = fromBase64(parts[0]);

final byte[] hash = fromBase64(parts[1]);

boolean validHash = Arrays.equals(createHmac(userBytes), hash);

if (validHash) {

final User user = fromJSON(userBytes);

if (new Date().getTime() < user.getExpires()) {

return user;

}

}

} catch (IllegalArgumentException e) {

//log tampering attempt here

}

}

return null;

}

It essentially validates if the provided hash is the same as a fresh computed hash of the content. **Because the createHmac method uses an undisclosed secret key internally to compute the hash, no client will be able to temper with the content and provide a hash that is the same as the one the server will produce.** Only after passing this test the provided data will be interpreted as JSON representing a User object.

Zooming in on the Hmac part, lets see the exact Java involved. First it must be initialized with a secret key, which I do as part of TokenHandler’s constructor:

HMAC initialization

...

private static final String HMAC\_ALGO = "HmacSHA256";

private final Mac hmac;

public TokenHandler(byte[] secretKey) {

try {

hmac = Mac.getInstance(HMAC\_ALGO);

hmac.init(new SecretKeySpec(secretKey, HMAC\_ALGO));

} catch (NoSuchAlgorithmException | InvalidKeyException e) {

throw new IllegalStateException(

"failed to initialize HMAC: " + e.getMessage(), e);

}

}

After initialization it can be (re-)used, using a single method call! (doFinal’s JavaDoc reads “Processes the given array of bytes and finishes the MAC operation. A call to this method resets this Mac object to the state it was in when previously initialized via a call to init(Key) or init(Key, AlgorithmParameterSpec)…”)

createHmac

// synchronized to guard internal hmac object

private synchronized byte[] createHmac(byte[] content) {

return hmac.doFinal(content);

}

I used some crude synchronization here, to prevent conflicts when used within a Spring Singleton Service. The actual method is very fast (~0.01ms) so it shouldn’t cause a problem unless your going for 10k+ requests per seconds per server.

Speaking of the Service, lets work our way up to a fully working token-based authentication service:

TokenAuthenticationService

@Service

public class TokenAuthenticationService {

private static final String AUTH\_HEADER\_NAME = "X-AUTH-TOKEN";

private static final long TEN\_DAYS = 1000 \* 60 \* 60 \* 24 \* 10;

private final TokenHandler tokenHandler;

@Autowired

public TokenAuthenticationService(@Value("${token.secret}") String secret) {

tokenHandler = new TokenHandler(DatatypeConverter.parseBase64Binary(secret));

}

public void addAuthentication(HttpServletResponse response, UserAuthentication authentication) {

final User user = authentication.getDetails();

user.setExpires(System.currentTimeMillis() + TEN\_DAYS);

response.addHeader(AUTH\_HEADER\_NAME, tokenHandler.createTokenForUser(user));

}

public Authentication getAuthentication(HttpServletRequest request) {

final String token = request.getHeader(AUTH\_HEADER\_NAME);

if (token != null) {

final User user = tokenHandler.parseUserFromToken(token);

if (user != null) {

return new UserAuthentication(user);

}

}

return null;

}

}

Pretty straight-forward, initializing a private TokenHandler to do the heavy lifting. It provides methods for adding and reading the custom HTTP token header. As you can see it does not use any (database driven) UserDetailsService to lookup the user details. All details required to let Spring Security handle further authorization checks are provided by means of the token.  
Finally we can now plug-in all of this into Spring Security adding two custom filters in the Security configuration:

Security configuration inside StatelessAuthenticationSecurityConfig

...

@Override

protected void configure(HttpSecurity http) throws Exception {

http

...

// custom JSON based authentication by POST of

// {"username":"<name>","password":"<password>"}

// which sets the token header upon authentication

.addFilterBefore(new StatelessLoginFilter("/api/login", ...),

UsernamePasswordAuthenticationFilter.class)

// custom Token based authentication based on

// the header previously given to the client

.addFilterBefore(new StatelessAuthenticationFilter(...),

UsernamePasswordAuthenticationFilter.class);

}

...

The StatelessLoginFilter adds the token upon successful authentication:

StatelessLoginFilter

...

@Override

protected void successfulAuthentication(HttpServletRequest request, HttpServletResponse response,

FilterChain chain, Authentication authentication) throws IOException, ServletException {

// Lookup the complete User object from the database and create an Authentication for it

final User authenticatedUser = userDetailsService.loadUserByUsername(authentication.getName());

final UserAuthentication userAuthentication = new UserAuthentication(authenticatedUser);

// Add the custom token as HTTP header to the response

tokenAuthenticationService.addAuthentication(response, userAuthentication);

// Add the authentication to the Security context

SecurityContextHolder.getContext().setAuthentication(userAuthentication);

}

the StatelessAuthenticationFilter simply sets the authentication based upon the header:

StatelessAuthenticationFilter

...

@Override

public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain)

throws IOException, ServletException {

SecurityContextHolder.getContext().setAuthentication(

tokenAuthenticationService.getAuthentication((HttpServletRequest) req));

chain.doFilter(req, res); // always continue

}

...

Note that unlike most Spring Security related filters, I choose to continue down the filter chain regardless of successful authentication. I wanted to support triggering Spring’s  AnonymousAuthenticationFilter to support anonymous authentication. The big difference here being that the filter is not configured to map to any url specifically meant for authentication, so not providing the header isn’t really a fault.

### **Client-side Implementation**

Client-side implementation is again pretty straight-forward. Again I’m keeping it minimalistic to prevent the authentication bit being lost in AngularJS details. If you’re looking for an AngularJS JWT example more thoroughly integrated with routes you should take a look [here](http://angular-tips.com/blog/2014/05/json-web-tokens-examples/). I borrowed some of the interceptor logic from it.  
Logging in, is simply a matter of storing the token (in [localStorage](http://diveintohtml5.info/storage.html)):

login

$scope.login = function () {

var credentials = { username: $scope.username, password: $scope.password };

$http.post('/api/login', credentials).success(function (result, status, headers) {

$scope.authenticated = true;

TokenStorage.store(headers('X-AUTH-TOKEN'));

});

};

Logging out is even simpler (no call to the server necessary):

Logout

$scope.logout = function () {

// Just clear the local storage

TokenStorage.clear();

$scope.authenticated = false;

};

To check if a user is “already logged in” ng-init=”init()” works nicely:

init

$scope.init = function () {

$http.get('/api/users/current').success(function (user) {

if(user.username !== 'anonymousUser'){

$scope.authenticated = true;

$scope.username = user.username;

}

});

};

I choose to use an anonymously reachable endpoint to prevent triggering 401/403′s. You could also decode the token itself and check the expiration time, trusting the local client time to be accurate enough.

Finally in order to automate the process of adding the header a simple interceptor much like in last blog entry does nicely:

TokenAuthInterceptor

factory('TokenAuthInterceptor', function($q, TokenStorage) {

return {

request: function(config) {

var authToken = TokenStorage.retrieve();

if (authToken) {

config.headers['X-AUTH-TOKEN'] = authToken;

}

return config;

},

responseError: function(error) {

if (error.status === 401 || error.status === 403) {

TokenStorage.clear();

}

return $q.reject(error);

}

};

}).config(function($httpProvider) {

$httpProvider.interceptors.push('TokenAuthInterceptor');

});

It also takes care of automatically clearing the token after receiving an HTTP 401 or 403, assuming the client isn’t going to allow calls to areas that need higher privileges.

### **TokenStorage**

The TokenStorage is just a wrapper service over localStorage which I’ll not bother you with. Putting the token in the localStorage protects it from being read by script outside the origin of the script that saved it, just like cookies. However because the token is not an actual Cookie, no browser can be instructed add it to requests automatically. This is essential as it completely prevents any form of CSRF attacks. Thus saving you from having to implement any (Stateless) CSRF protection mentioned in my previous blog.

* You can find a complete working example with some nice extras at [github](https://github.com/Robbert1/boot-stateless-auth).

<https://github.com/Robbert1/boot-stateless-auth>

Make sure you have gradle 2.0 installed and simply run it using “gradle build” followed by a “gradle run”. If you want to play with it in your IDE like Eclipse, go with “gradle eclipse” and just import and run it from within your IDE (no server needed).