SPRING BOOT

Implementing JWT Authentication on Spring Boot APIs

Let's learn the correct way to secure Spring Boot RESTful APIs with JWTs.

Bruno Krebs
R&D Content Architect

Last Updated On: July 17, 2018

TL;DR In this blog post, we will learn how to handle authentication and authorization on *RESTful APIs* written with *Spring Boot*. We will clone, from *GitHub*, a simple Spring Boot application that exposes public endpoints, and then we will secure these endpoints with *Spring Security* and *JWTS*.

Securing RESTful APIs with JW

JSON Web Tokens, commonly known as JWTs, ar 's applications. This technology has gained popularity over the past few years because it enables

backends to accept requests simply by validating the contents of these JWTS. That is, applications that use JWTS no longer have to hold cookies or other session data about their users. This characteristic facilitates scalability while keeping applications secure.

During the authentication process, when a user successfully logs in using their credentials, a JSON Web Token is returned and must be saved locally (typically in local storage). Whenever the user wants to access a protected route or resource (an endpoint), the user agent must send the JWT, usually in the `Authorization` header using the *Bearer schema*, along with the request.

When a backend server receives a request with a JWT, the first thing to do is to validate the token. This consists of a series of steps, and if any of these fails then, the request must be rejected. The following list shows the validation steps needed:

- Check that the JWT is well formed
- Check the signature
- Validate the standard claims
- Check the Client permissions (scopes)

We won't get into the nitty-gritty details about JWTS in this article but, if needed, this resource can provide more about information about JWTS and this resource about JWT validation.

For an even more in-depth look at JSON Web Tokens, you can download our free ebook below.

The RESTful Spring Boot API Overview

The RESTful Spring Boot API that we are going to secure is a task list manager. The task list is kept globally, which means that all users will see and interact with the same list. To clone and run this application, let's issue the following commands:

```
# clone the starter project
git clone https://github.com/auth0-blo

cd springboot-auth-updated
```

To ensure compatibility with Java 10, we have to add the following line to the `build.gradle` file:

```
...

dependencies {
    ...
    compile('javax.xml.bind:jaxb-api:2.3.0')
}
```

Then, we run the unsecured RESTful API by either issuing the command `gradle bootRun` from the command line or by building and running the project in our favorite IDE.

If everything works as expected, our RESTful Spring Boot API will be up and running. To test it, we can use a tool like Postman or curl to issue request to the available endpoints:

```
# issue a GET request to see the (empty) list of tasks
curl http://localhost:8080/tasks

# issue a POST request to create a new task
curl -H "Content-Type: application/json" -X POST -d '{
    "description": "Buy some milk(shake)"
}'  http://localhost:8080/tasks

# issue a PUT request to update the recently created task
curl -H "Content-Type: application/json" -X PUT -d '{
    "description": "Buy some milk"
}'  http://localhost:8080/tasks/1

# issue a DELETE request to remove the
curl -X DELETE http://localhost:8080/tasks/1
```

All the endpoints used in the commands above are defined in the `TaskController` class, which belongs to the `com.auth0.samples.authapi.springbootauthupdated.task` package. Besides this class, this package contains two other classes:

- `Task`: the entity model that represents tasks in the application.
- `TaskRepository`: the class responsible for handling the persistence of `Tasks`.

The persistence layer of our application is backed by an in-memory database called *HSQLDB*. We would typically use a production-ready database like *PostgreSQL* or *MySQL* on real applications, but for this tutorial this in-memory database will be enough.

Enabling User Registration on Spring Boot APIs

Now that we took a look at the endpoints that our RESTful Spring Boot API exposes, we are going to start securing it. The first step is to allow new users to register themselves. The classes that we will create in this feature will belong to a new package called `com.auth0.samples.authapi.springbootauthupdated.user`. Let's create this package and add a new entity class called `ApplicationUser` to it:

```
package com.auth0.samples.authapi.springbootauthupdated.user;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class ApplicationUser {
    @Id
    @GeneratedValue(strategy = Generat
    private long id;
    private String username;
```

```
    private String password;

    public long getId() {
        return id;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
}
```

This entity class contains three properties:

- the `id` that works as the primary identifier of a user instance in the application,
- the `username` that will be used by users to identify themselves,
- and the `password` to check the user identity.

To manage the persistence layer of this entity, we
`ApplicationUserRepository`. This interface wil
gives us access to some common methods like `s`
the `ApplicationUser` class:

```
package com.auth0.samples.authapi.springbootauthupdated.user;

import org.springframework.data.jpa.repository.JpaRepository;

public interface ApplicationUserRepository extends JpaRepository<ApplicationUs
    ApplicationUser findByUsername(String username);
}
```

We have also added a method called `findByUsername` to this interface. This method will be used when we implement the authentication feature.

The endpoint that enables new users to register will be handled by a new `@Controller` class. We will call this controller `UserController` and add it to the same package as the `ApplicationUser` class:

```
package com.auth0.samples.authapi.springbootauthupdated.user;

import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/users")
public class UserController {

    private ApplicationUserRepository
    private BCryptPasswordEncoder bCry
```

```
    public UserController(ApplicationUserRepository applicationUserRepository,
                          BCryptPasswordEncoder bCryptPasswordEncoder) {
        this.applicationUserRepository = applicationUserRepository;
        this.bCryptPasswordEncoder = bCryptPasswordEncoder;
    }

    @PostMapping("/sign-up")
    public void signUp(@RequestBody ApplicationUser user) {
        user.setPassword(bCryptPasswordEncoder.encode(user.getPassword()));
        applicationUserRepository.save(user);
    }
}
```

The implementation of the endpoint is quite simple. All it does is encrypt the password of the new user (holding it as plain text wouldn't be a good idea) and then save it to the database. The encryption process is handled by an instance of `BCryptPasswordEncoder`, which is a class that belongs to the Spring Security framework.

Right now we have two gaps in our application:

1. We didn't include the Spring Security framework as a dependency to our project.
2. There is no default instance of `BCryptPasswordEncoder` that can be injected in the `UserController` class.

The first problem we solve by adding the Spring Security framework dependency to the `./build.gradle` file:

```
...

dependencies {
    ...
```

```
    compile("org.springframework.boot:spring-boot-starter-security")
}
```

The second problem, the missing `BCryptPasswordEncoder` instance, we solve by implementing a method that generates an instance of `BCryptPasswordEncoder`. This method must be annotated with `@Bean` and we will add it in the `SpringbootAuthUpdatedApplication` class:

```
package com.auth0.samples.authapi.springbootauthupdated;

// ... other imports
import org.springframework.context.annotation.Bean;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

@SpringBootApplication
public class SpringbootAuthUpdatedApplication {

    @Bean
    public BCryptPasswordEncoder bCryptPasswordEncoder() {
        return new BCryptPasswordEncoder();
    }

    // ... main method definition
}
```

This ends the user registration feature, but we still lack support for user authentication and authorization. Let's tackle these features next.

User Authentication and Author

To support both authentication and authorization

- implement an authentication filter to issue JWTS to users sending credentials,
- implement an authorization filter to validate requests containing JWTS,
- create a custom implementation of `UserDetailsService` to help Spring Security loading user-specific data in the framework,
- and extend the `WebSecurityConfigurerAdapter` class to customize the security framework to our needs.

Before proceeding to the development of these filters and classes, let's create a new package called `com.auth0.samples.authapi.springbootauthupdated.security`. This package will hold all the code related to our app's security.

The Authentication Filter

The first element that we are going to create is the class responsible for the authentication process. We are going to call this class `JWTAuthenticationFilter`, and we will implement it with the following code:

```
package com.auth0.samples.authapi.springbootauthupdated.security;

import com.auth0.jwt.JWT;
import com.auth0.samples.authapi.springbootauthupdated.user.ApplicationUser;
import com.fasterxml.jackson.databind.ObjectMapper;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.UsernamePasswordAuthenticat
import org.springframework.security.core.Authentication;
import org.springframework.security.core.AuthenticationException;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.web.authentication.UsernamePasswordAuthent

import javax.servlet.FilterChain;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletR
import javax.servlet.http.HttpServletResponse;
```

```
import java.io.IOException;
import java.util.ArrayList;
import java.util.Date;

import static com.auth0.jwt.algorithms.Algorithm.HMAC512;
import static com.auth0.samples.authapi.springbootauthupdated.security.Securit
import static com.auth0.samples.authapi.springbootauthupdated.security.Securit
import static com.auth0.samples.authapi.springbootauthupdated.security.Securit
import static com.auth0.samples.authapi.springbootauthupdated.security.Securit

public class JWTAuthenticationFilter extends UsernamePasswordAuthenticationFil
    private AuthenticationManager authenticationManager;

    public JWTAuthenticationFilter(AuthenticationManager authenticationManager
        this.authenticationManager = authenticationManager;
    }

    @Override
    public Authentication attemptAuthentication(HttpServletRequest req,
                                                HttpServletResponse res) throw
        try {
            ApplicationUser creds = new ObjectMapper()
                    .readValue(req.getInputStream(), ApplicationUser.class);

            return authenticationManager.authenticate(
                    new UsernamePasswordAuthenticationToken(
                            creds.getUsername(),
                            creds.getPassword(),
                            new ArrayList<>())
            );
        } catch (IOException e) {
            throw new RuntimeException(e);
```

```
        }
    }

    @Override
    protected void successfulAuthentication(HttpServletRequest req,
                                            HttpServletResponse res,
                                            FilterChain chain,
                                            Authentication auth) throws IOExce

        String token = JWT.create()
                .withSubject(((User) auth.getPrincipal()).getUsername())
                .withExpiresAt(new Date(System.currentTimeMillis() + EXPIRATIO
                .sign(HMAC512(SECRET.getBytes()));
        res.addHeader(HEADER_STRING, TOKEN_PREFIX + token);
    }
}
```

Note that the authentication filter that we created extends the `UsernamePasswordAuthenticationFilter` class. When we add a new filter to Spring Security, we can explicitly define where in the *filter chain* we want that filter, or we can let the framework figure it out by itself. By extending the filter provided within the security framework, Spring can automatically identify the best place to put it in the security chain.

Our custom authentication filter overwrites two methods of the base class:

- `attemptAuthentication`: where we parse the user's credentials and issue them to the `AuthenticationManager`.
- `successfulAuthentication`: which is the method called when a user successfully logs in. We use this method to generate a JWT for thi

Our IDE will probably complain about the code in se
code imports four constants from a class that we

Second, because this class generates JWTS with the help of a class called `JWT`, which belongs to a library that we haven't added as dependency to our project.

Let's solve the missing dependency first. In the `./build.gradle` file, let's add the following line of code:

```
...

dependencies {
    ...
    compile("com.auth0:java-jwt:3.4.0")
}
```

This will add the Java JWT: JSON Web Token for Java and Android library to our project, and will solve the issue of the missing classes. Now we have to create the `SecurityConstants` class:

```
package com.auth0.samples.authapi.springbootauthupdated.security;

public class SecurityConstants {
    public static final String SECRET = "SecretKeyToGenJWTs";
    public static final long EXPIRATION_TIME = 864_000_000; // 10 days
    public static final String TOKEN_PREFIX = "Bearer ";
    public static final String HEADER_STRING = "Authorization";
    public static final String SIGN_UP_URL = "/users/sign-up";
}
```

This class contains all four constants referenced by the `JWTAuthenticationFilter` class, alongside a `SIGN_UP_URL` constant that will be u

The Authorization Filter

As we have implemented the filter responsible for authenticating users, we now need to implement the filter responsible for user authorization. We create this filter as a new class, called `JWTAuthorizationFilter`, in the `com.auth0.samples.authapi.springbootauthupdated.security` package:

```
package com.auth0.samples.authapi.springbootauthupdated.security;

import com.auth0.jwt.JWT;
import com.auth0.jwt.algorithms.Algorithm;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.UsernamePasswordAuthenticat
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.web.authentication.www.BasicAuthentication

import javax.servlet.FilterChain;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.ArrayList;

import static com.auth0.samples.authapi.springbootauthupdated.security.Securit
import static com.auth0.samples.authapi.springbootauthupdated.security.Securit
import static com.auth0.samples.authapi.springbootauthupdated.security.Securit

public class JWTAuthorizationFilter extends BasicAuthenticationFilter {

    public JWTAuthorizationFilter(AuthenticationManager authManager) {
        super(authManager);
    }

    @Override
```

```
protected void doFilterInternal(HttpServletRequest req,
                                HttpServletResponse res,
                                FilterChain chain) throws IOException, Ser
    String header = req.getHeader(HEADER_STRING);

    if (header == null || !header.startsWith(TOKEN_PREFIX)) {
        chain.doFilter(req, res);
        return;
    }

    UsernamePasswordAuthenticationToken authentication = getAuthentication

    SecurityContextHolder.getContext().setAuthentication(authentication);
    chain.doFilter(req, res);
}

private UsernamePasswordAuthenticationToken getAuthentication(HttpServletR
    String token = request.getHeader(HEADER_STRING);
    if (token != null) {
        // parse the token.
        String user = JWT.require(Algorithm.HMAC512(SECRET.getBytes()))
                .build()
                .verify(token.replace(TOKEN_PREFIX, ""))
                .getSubject();

        if (user != null) {
            return new UsernamePasswordAuthenticationToken(user, null, new
        }
        return null;
    }
    return null;
```

```
    }
}
```

We have extended the `BasicAuthenticationFilter` to make Spring replace it in the *filter chain* with our custom implementation. The most important part of the filter that we've implemented is the private `getAuthentication` method. This method reads the JWT from the `Authorization` header, and then uses `JWT` to validate the token. If everything is in place, we set the user in the `SecurityContext` and allow the request to move on.

Integrating the Security Filters on Spring Boot

Now that we have both security filters properly created, we have to configure them on the Spring Security filter chain. To do that, we are going to create a new class called `WebSecurity` in the `com.auth0.samples.authapi.springbootauthupdated.security` package:

```
package com.auth0.samples.authapi.springbootauthupdated.security;

import org.springframework.http.HttpMethod;
import org.springframework.security.config.annotation.authentication.builders.
import org.springframework.security.config.annotation.web.builders.HttpSecurit
import org.springframework.security.config.annotation.web.configuration.Enable
import org.springframework.security.config.annotation.web.configuration.WebSec
import org.springframework.security.config.http.SessionCreationPolicy;
import com.auth0.samples.authapi.springbootauthupdated.user.UserDetailsService
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.web.cors.CorsConfiguration;
import org.springframework.web.cors.CorsConfigurationSource;
import org.springframework.web.cors.UrlBasedCorsConfigurationSource;
import org.springframework.context.ann

import static com.auth0.samples.authapi.springbootauthupdated.security.Securit
```

```
@EnableWebSecurity
public class WebSecurity extends WebSecurityConfigurerAdapter {
    private UserDetailsServiceImpl userDetailsService;
    private BCryptPasswordEncoder bCryptPasswordEncoder;

    public WebSecurity(UserDetailsServiceImpl userDetailsService, BCryptPasswo
        this.userDetailsService = userDetailsService;
        this.bCryptPasswordEncoder = bCryptPasswordEncoder;
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.cors().and().csrf().disable().authorizeRequests()
                .antMatchers(HttpMethod.POST, SIGN_UP_URL).permitAll()
                .anyRequest().authenticated()
                .and()
                .addFilter(new JWTAuthenticationFilter(authenticationManager()
                .addFilter(new JWTAuthorizationFilter(authenticationManager())
                // this disables session creation on Spring Security
                .sessionManagement().sessionCreationPolicy(SessionCreationPoli
    }

    @Override
    public void configure(AuthenticationManagerBuilder auth) throws Exception
        auth.userDetailsService(userDetailsService).passwordEncoder(bCryptPass
    }

  @Bean
  CorsConfigurationSource corsConfigur
    final UrlBasedCorsConfigurationSou
    source.registerCorsConfiguration("/**", new CorsConfiguration().applyPermi
```

```
    return source;
  }
}
```

We have annotated this class with `@EnableWebSecurity` and made it extend `WebSecurityConfigurerAdapter` to take advantage of the default web security configuration provided by Spring Security. This allows us to fine-tune the framework to our needs by defining three methods:

- `configure(HttpSecurity http)`: a method where we can define which resources are public and which are secured. In our case, we set the `SIGN_UP_URL` endpoint as being public and everything else as being secured. We also configure CORS (Cross-Origin Resource Sharing) support through `http.cors()` and we add a custom security filter in the Spring Security filter chain.
- `configure(AuthenticationManagerBuilder auth)`: a method where we defined a custom implementation of `UserDetailsService` to load user-specific data in the security framework. We have also used this method to set the encrypt method used by our application (`BCryptPasswordEncoder`).
- `corsConfigurationSource()`: a method where we can allow/restrict our CORS support. In our case we left it wide open by permitting requests from any source (`/**`).

Spring Security doesn't come with a concrete implementation of `UserDetailsService` that we could use out of the box with our in-memory database. Therefore, we create a new class called `UserDetailsServiceImpl` in the `com.auth0.samples.authapi.springbootauthupdated.user` package to provide one:

```
package com.auth0.samples.authapi.springbootauthupdated.user;

import org.springframework.security.co
import org.springframework.security.co
import org.springframework.security.co
import org.springframework.security.core.userdetails.UsernameNotFoundException
```

```
import org.springframework.stereotype.Service;

import static java.util.Collections.emptyList;

@Service
public class UserDetailsServiceImpl implements UserDetailsService {
    private ApplicationUserRepository applicationUserRepository;

    public UserDetailsServiceImpl(ApplicationUserRepository applicationUserRep
        this.applicationUserRepository = applicationUserRepository;
    }

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotF
        ApplicationUser applicationUser = applicationUserRepository.findByUser
        if (applicationUser == null) {
            throw new UsernameNotFoundException(username);
        }
        return new User(applicationUser.getUsername(), applicationUser.getPass
    }
}
```

The only method that we had to implement is `loadUserByUsername`. When a user tries to authenticate, this method receives the username, searches the database for a record containing it, and (if found) returns an instance of `User`. The properties of this instance (`username` and `password`) are then checked against the credentials passed by the user in the login request. This last process is executed outside this class, by the Spring Security framework.

We can now rest assured that our endpoints won't [illegible] upport authentication and authorization with JWTS on Sp[illegible]g, our application (through the IDE or through `grad`[illegible]q

```
# issues a GET request to retrieve tasks with no JWT
# HTTP 403 Forbidden status is expected
curl http://localhost:8080/tasks

# registers a new user
curl -H "Content-Type: application/json" -X POST -d '{
    "username": "admin",
    "password": "password"
}' http://localhost:8080/users/sign-up

# logs into the application (JWT is generated)
curl -i -H "Content-Type: application/json" -X POST -d '{
    "username": "admin",
    "password": "password"
}' http://localhost:8080/login

# issue a POST request, passing the JWT, to create a task
# remember to replace xxx.yyy.zzz with the JWT retrieved above
curl -H "Content-Type: application/json" \
-H "Authorization: Bearer xxx.yyy.zzz" \
-X POST -d '{
    "description": "Buy watermelon"
}'  http://localhost:8080/tasks

# issue a new GET request, passing the JWT
# remember to replace xxx.yyy.zzz with the JWT retrieved above
curl -H "Authorization: Bearer xxx.yyy.zzz" http://localhost:8080/tasks
```

> **Note: You might be wondering what** … **issued to the** `/login` **endpoint. The a** … **pl**

> `JWTAuthenticationFilter` class that you created previously extends `UsernamePasswordAuthenticationFilter`. This filter, which is provided by Spring Security, registers itself as the responsible for this endpoint. As such, whenever your backend API gets a request to `/login`, your specialization of this filter (i.e., `JWTAuthenticationFilter`) goes into action and handles the authentication attempt (through the `attemptAuthentication` method).

Aside: Securing Spring APIs with Auth0

Securing Spring Boot APIs with Auth0 is easy and brings a lot of great features to the table. With Auth0, we only have to write a few lines of code to get solid identity management solution, single sign-on, support for social identity providers (like Facebook, GitHub, Twitter, etc.), and support for enterprise identity providers (like Active Directory, LDAP, SAML, custom, etc.).

In the following sections, we are going to learn how to use Auth0 to secure APIs written with Spring Boot.

Creating the API

First, we need to create an API on our free Auth0 account. To do that, we have to go to the APIs section of the management dashboard and click on "Create API". On the dialog that appears, we can name our API as "Contacts API" (the name isn't really important) and identify it as `https://contacts.blog-samples.com` (we will use this value later).

Registering the Auth0 Dependency

The second step is to import a dependency called `auth0-spring-security-api`. This can be done on a Maven project by including the following configuration to `pom.xml` (it's not harder to do this on Gradle, Ivy, and so on):


```
<project ...>

    <!-- everything else ... -->
```

```
    <dependencies>
        <!-- other dependencies ... -->
        <dependency>
            <groupId>com.auth0</groupId>
            <artifactId>auth0-spring-security-api</artifactId>
            <version>1.0.0-rc.3</version>
        </dependency>
    </dependencies>
</project>
```

Integrating Auth0 with Spring Security

The third step consists of extending the WebSecurityConfigurerAdapter class. In this extension, we use `JwtWebSecurityConfigurer` to integrate Auth0 and Spring Security:

```
package com.auth0.samples.secure;

import com.auth0.spring.security.api.JwtWebSecurityConfigurer;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.method.configuration.Ena
```

```
import org.springframework.security.config.annotation.web.configuration.WebSec

@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class SecurityConfig extends We
    @Value(value = "${auth0.apiAudienc
    private String apiAudience;
    @Value(value = "${auth0.issuer}")
```

```
    private String issuer;

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        JwtWebSecurityConfigurer
                .forRS256(apiAudience, issuer)
                .configure(http)
                .cors().and().csrf().disable().authorizeRequests()
                .anyRequest().permitAll();
    }
}
```

As we don't want to hard code credentials in the code, we make `SecurityConfig` depend on two environment properties:

- `auth0.apiAudience` : This is the value that we set as the identifier of the API that we created at Auth0 (`https://contacts.blog-samples.com`).
- `auth0.issuer` : This is our domain at Auth0, including the HTTP protocol. For example: `https://blog-samples.auth0.com/` .

Let's set them in a properties file on our Spring application (e.g. `application.properties`):

```
auth0.issuer:https://blog-samples.auth0.com/
auth0.apiAudience:https://contacts.blog-samples.com/
```

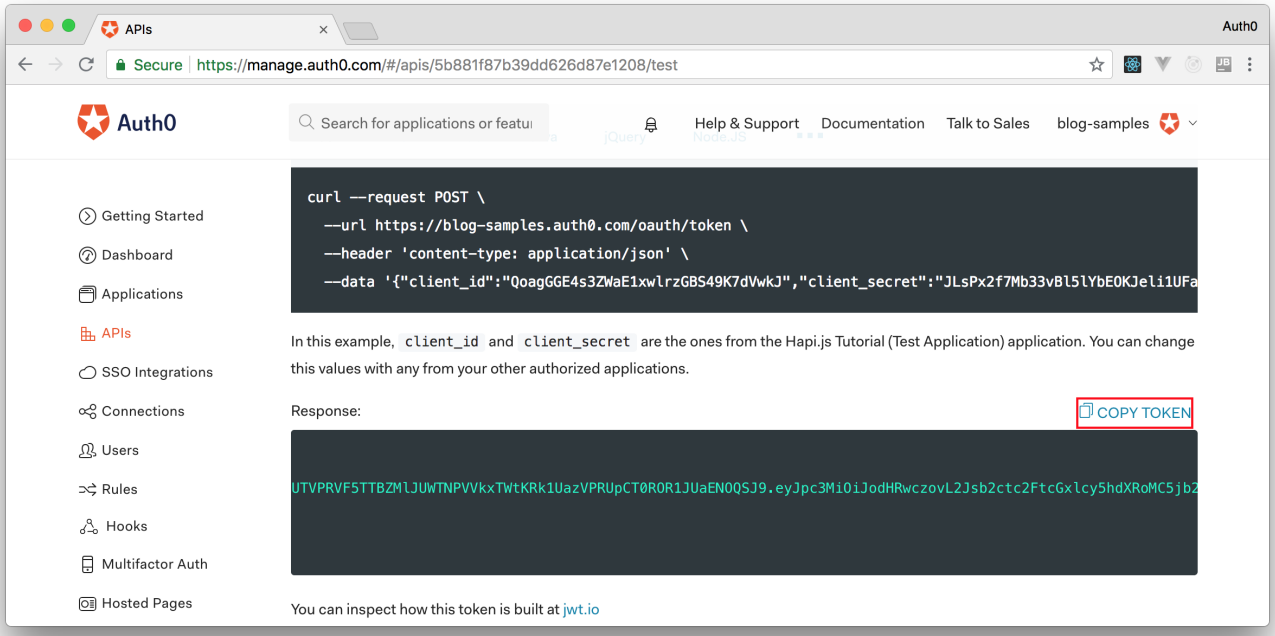
Securing Endpoints with Auth0

After integrating Auth0 and Spring Security, we can easily secure our endpoints with Spring Security annotations:

```
package com.auth0.samples.secure;

import com.google.common.collect.Lists;
import org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.List;

@RestController
@RequestMapping(value = "/contacts/")
public class ContactController {
    private static final List<Contact> contacts = Lists.newArrayList(
            Contact.builder().name("Bruno Krebs").phone("+5551987654321").buil
            Contact.builder().name("John Doe").phone("+5551888884444").build()
    );

    @GetMapping
    public List<Contact> getContacts() {
        return contacts;
    }

    @PostMapping
    public void addContact(@RequestBody Contact contact) {
        contacts.add(contact);
    }
}
```

Now, to be able to interact with our endpoints, we will have to obtain an access token from Auth0. There are multiple ways to do this and the strategy that we will use depends on the type of the client application we are developing. For example, if we are developing a Single Page Application (SPA), we will use what is called the *Implicit Grant*. If we are developing a mobile application, we will use the *Authorization Code Grant Flow with PKCE*. There are other flows available at Auth0. However, for a simple test like this one, we can use our Auth0 dashboard to get one.

Therefore, we can head back to the *APIs* section in our Auth0 dashboard, click on the API we created before, and then click on the *Test* section of this API. There, we will find a button called *Copy Token*. Let's click on this button to copy an access token to our clipboard.



After copying this token, we can open a terminal and issue the following commands:

```
# create a variable with our token
ACCESS_TOKEN=<OUR_ACCESS_TOKEN>

# use this variable to fetch contacts
curl -H 'Authorization: Bearer '$ACCES
```

> **Note: We will have to replace `<OUR_ACCESS_TOKEN>` with the token we copied from our dashboard.**

As we are now using our access token on the requests we are sending to our API, we will manage to get the list of contacts again.

That's how we secure our Node.js backend API. Easy, right?

Conclusion

Securing RESTful Spring Boot API with JWTS is not a hard task. This article showed that by creating a couple of classes and extending a few others provided by Spring Security, we can protect our endpoints from unknown users, enable users to register themselves, and authenticate existing users based on JWTS.

Of course that for a production-ready application we would need a few more features, like password retrieval, but the article demystified the most sensible parts of dealing with JWTS to authorize requests on Spring Boot applications.

Bruno Krebs
R&D CONTENT ARCHITECT

I am passionate about developing highly scalable, resilient applications. I love everything from the database, to microservices (Kubernetes, Docker, etc), to the frontend. I find amazing to think about how all pieces work together to provide a fast and pleasurable experience to end users, mainly because they have no clue how complex that "simple" app is.

VIEW PROFILE ▸

More like this

SPRING BOOT

Developing JSF applications with Spring Boot

FLYWAY

Database Versioning with Flyw

JAVA

Java Platform and Java Community Process Overview

Follow the conversation

Comments Community Privacy Policy 1 Login

Recommend 30 Tweet Share Sort by Best

Join the discussion…

LOG IN WITH OR SIGN UP WITH DISQUS ?

Name

Dexter • 2 years ago • edited

everytime my loadUserByUsername defined in UserDetailsServiceImpl is called, the username is saying "NONE_PROVIDED". But during the call of authenticationManager.authenticate(new UsernamePasswordAuthenticationToken(creds.getUsername(), creds.getPassword(), new ArrayList<>()) inside JWTAUTHENTICATIONFILTER

the Username and password are neither empty neither "NONE_PROVIDED". Any idea what might be wrong? Note that I don't have an username field ir My user table where I am making the sign email which is to be used as username an to be used as password

8 | • Reply • Share ›

Hey, you're back! Things must be getting serious 😉 Want to get in touch with a rep?

1

Bruno S. Krebs Mod → Dexter • 2 years ago

Hmmm, is there some code that I could take a look? Perhaps on GitHub??

2 | • Reply • Share ›

Ryan Hack → Dexter • 2 years ago

I'm also getting this issue

| • Reply • Share ›

Bruno S. Krebs Mod → Ryan Hack • 2 years ago

Hey there, could you share a GitHub repo with details so I can verify?

| • Reply • Share ›

Adelin Ghanayem • a year ago

There is something that I don't really understand when you authorize a toke in your `JWTAuthorizationFilter` you are creating a `UsernamePasswordAuthenticationToken` without adding a password which fails with Bad Credentials when trying to execute a protected method. The exception is throw from `DaoAuthenticationProvider` `additionalAuthenticationChecks` method which check the presence of password ?
I Had to created my own token called `JWTAuthentication` and a custom `JWTAuthenticationProvider`

5 | • Reply • Share ›

Bruno S. Krebs Mod → Adelin Ghanayem • a year ago

Strange, I just followed, step by step, the instructions here and I had no problems executing any of the protected methods... I wonder what is different in my environment and in yours.

| • Reply • Share ›

Sazn Lamsal • 2 years ago

Hi Bruno, thanks for this tutorial.

I am trying to implement multiple user login from different tables. (admin and user).

How can I enable the check if the user is a normal user ?

2 | • Reply • Share ›

Bruno S. Krebs Mod → Sazn Lamsal • 2 years ago

Why are you using two different tables? A better approach is to have only one table with a flag (a boolean column) indicating who the admins are.

Reply • Share ›

andremorua ➚ Bruno S. Krebs
• 2 years ago • edited

Because many DBAs keep records separately: external users and internal users or something else.

Reply • Share ›

Bruno S. Krebs Mod ➚
andremorua • 2 years ago

Interesting. Thanks for letting me know (although I don't really understand the reason).

1 • Reply • Share ›

andremorua ➚ Bruno S. Krebs
• 2 years ago • edited

requirements for security.
As well as no: long id, only string what you can get java.util.UUID - 36 chars... and so on... does JPA work?

Reply • Share ›

Bruno S. Krebs Mod ➚
andremorua • 2 years ago

Sorry, didn't understand the message properly. If you are asking if you can use UUID and JPA together, I do not know for sure but, apparently, you can:

https://medium.com/@swhp/wo...

Reply • Share ›

Mojtaba • 3 years ago

It does not check token expiration time.
How do I implement that?

2 • Reply • Share ›

Angel Casapía ➚ Mojtaba • 2 years ago

http://javadox.com/io.jsonw...

public class ExpiredJwtExceptionextends

Hey, you're back! Things must be getting serious 😉 Want to get in touch with a rep?

1

JwtException
Exception indicating that a JWT was accepted after it expired and must be rejected.
Since:
0.3

<dependency>
<groupid>io.jsonwebtoken</groupid>
<artifactid>jjwt</artifactid>
<version>0.9.0</version>
</dependency>

Reply • Share ›

Rowan ➚ Mojtaba • 3 years ago • edited

I tested it with a 1 minute expiration and the token was rejected during parsing:

```
{
    "timestamp": 1504703883233,
    "status": 500,
    "error": "Internal Server Error",
    "exception": "io.jsonwebtoken.ExpiredJwtExcep
    "message": "JWT expired at 2017-09-06T22:17:4
    "path": "/tasks"
}
```

Reply • Share ›

Joshi ➚ Rowan • 2 years ago

i resolved this .

Reply • Share ›

Ivan Eggel • 3 years ago • edited

Hi Bruno,

First of all: nice tutorial, great work.
However I think there is a security hole in the code. It seems that sessions are still enabled on the server side. That means if a user gets successfully authorized and gets a JSESSIONID Cookie subsequent reques[illegible] there is no authorization header present in[illegible] header.

Maybe you would consider extending the [illegible] 'configure(HttpSecurity http)' in the WebSecurity class with the following code:

the following code:
http.sessionManagement().sessionCreationPolicy(SessionC

This prevents spring from creating and accepting cookies. If you have a better solution regarding this security risk, please let me now.

3 ^ | v 2 • Reply • Share ›

Bruno Krebs ➚ Ivan Eggel • 3 years ago

Thanks for the contribution, I will take a look.

^ | v • Reply • Share ›

Naresh Thota ➚ Bruno Krebs • 2 years ago

any work around on above issue ?

^ | v • Reply • Share ›

Bruno S. Krebs Mod ➚ Naresh Thota • 2 years ago

Hi Naresh, Ivan's solution is just fine. I've updated the article, as you can see here.

^ | v • Reply • Share ›

Nguyễn Thanh Bình ➚ Bruno Krebs • 2 years ago

When creating WebSecurity class, you should import org.springframework.security.config.http.Sess after applying Ivan's solution.

^ | v 1 • Reply • Share ›

Bruno S. Krebs Mod ➚ Nguyễn Thanh Bình • 2 years ago

Thanks for catching this :) Fixing in 3, 2, 1...

^ | v • Reply • Share ›

Nguyễn Thanh Bình ➚ Bruno S. Krebs • 2 years ago

Could you please improve this solution by adding refresh token?

^ | v • Reply • Share ›

Photon Point ➚ Nguyễ Bình • 2 years ago

1- catch expired tok information to the user.

• Reply • Share ›

Photon Point ➚ Photon Point
• 2 years ago • edited

2- dispatch a refresh token to the user.

• Reply • Share ›

Photon Point ➚ Photon Point
• 2 years ago

3- Make two WebSecurityConfigurerAdapter instance. and add to customfilters

```
@EnableWebSecurity
@EnableGlobalMethodSecurity(preP
= true)
public class SecurityConfiguration {

@Configuration
@Order(1) // second
public static class
RestApiWebSecurity
extends
WebSecurityConfigurerAdapter {
//add custom filters}
```

```
@Configuration
@Order(0) // first filter
public static class
RestApi2WebSecurityConfigurerAdap
extends
WebSecurityConfigurerAdapter { //
add other custom filters}
```

1 • Reply • Share ›

This comment was deleted.

Photon Point Guest
• 2 years ago • edited

Organize second filter in compliance with your purpose.

```
SecurityContextHolder.getContext().se
// important!! this calls next filter
```

I don't think this is the best way. But it is enough for me at the moment.
Good luck!

• Reply • Share ›

Joe • 4 months ago

Great tutorial. I am trying to adapt this to an app which has both a web front-end and a REST API, with two WebSecurityConfigurerAdapters. Is there any way to map the JWTAuthenticationFilter to an endpoint other than "/login", for example "/api/login"? My app's web front-end uses form login and that uses the "/login" endpoint.

1 • Reply • Share ›

Bruno S. Krebs Mod Joe • 3 months ago

I guess it would be something like this: https://gist.github.com/bru...

• Reply • Share ›

Nifty Sherlock • 10 months ago

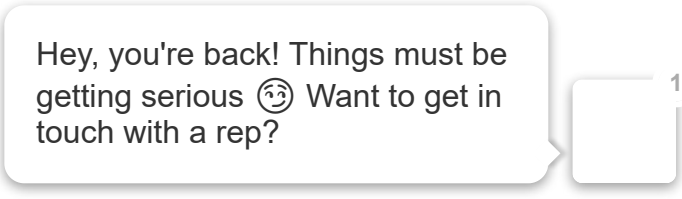
valuable content, thank you...

1 • Reply • Share ›

Bruno S. Krebs Mod Nifty Sherlock
• 9 months ago

Glad we could help :]

1 • Reply • Share ›



Henrik Fuiks • a year ago

Is there a reason why you don't use @Autowired at all?

1 | • Reply • Share ›

Bruno S. Krebs Mod → Henrik Fuiks • a year ago

I usually have only one constructor on a bean. As such, I can omit adding @Autowired. Perhaps, for clarity, I could add @Autowired to them though.

| • Reply • Share ›

Kaz Mu • 2 years ago

Thanks for the great tutorial! You saved me a lot of time!

1 | • Reply • Share ›

md7 zn4 • 2 years ago

Thanks. I learned a lot from this post. But what do you think about logout mechanism?

1 | • Reply • Share ›

petivagyok16 → md7 zn4 • 2 years ago

I'm curious as well. Basically, there's no such thing "logout" in a stateless authentication, but is it a good practice to put those jwt tokens into a blacklist database after user pushed the logout button on the client, or is it totally okay to leave the unused jwt tokens in the "ether" and let them just expire?

1 | • Reply • Share ›

Bruno Krebs → petivagyok16 • 2 years ago

You don't have to worry about existing tokens, as long as you use short-lived ones. If you think some token has been exposed to the public, then you can use blacklists.

That is, llogging out is a matter of deleting the token from your clients.

1 | • Reply • Share ›

petivagyok16 → Bruno Krebs • 2 years ago

Alright, thanks!

1 | • Reply • S

RIDVAN TANIK • 2 years ago

Hi

Hi,
Thanks for the great tutorial. But it seems i could not make it to invoke JWTAuthenticationFilter. Although JWTAuthorizationFilter is invoked and JWTAuthenticationFilter should be invoked before