BLOG

Implementing JWT Authentication on Spring Boot APIs

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

Bruno Krebs

August 10, 2017

42 51 29

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 JWTs

JSON Web Tokens, commonly known as JWTs, are tokens that are used to authenticate users on 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.

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-blog/spring-boot-auth.git

cd spring-boot-auth

# run the unsecured RESTful API
gradle bootRun
```

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 existing task
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.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.user`. Let's create this package and add a new entity class called `ApplicationUser` to it:

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

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

@Entity
public class ApplicationUser {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    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 will create an interface called `ApplicationUserRepository`. This interface will be an extension of `JpaRepository` —which gives us access to some common methods like `save` —and will be created in the same package of the `ApplicationUser` class:

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

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

public interface ApplicationUserRepository extends JpaRepository<ApplicationUser, Long> {
    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.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 applicationUserRepository;
    private BCryptPasswordEncoder bCryptPasswordEncoder;

    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 `Application` class:

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

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

@SpringBootApplication
public class Application {

    @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 Authorization on Spring Boot

To support both authentication and authorization in our application, we are going to:

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.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.security;

import com.auth0.samples.authapi.user.ApplicationUser;
import com.fasterxml.jackson.databind.ObjectMapper;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
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.UsernamePasswordAuthenticationFilter;

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 java.util.Date;

import static com.auth0.samples.authapi.security.SecurityConstants.EXPIRATION_TIME;
import static com.auth0.samples.authapi.security.SecurityConstants.HEADER_STRING;
import static com.auth0.samples.authapi.security.SecurityConstants.SECRET;
import static com.auth0.samples.authapi.security.SecurityConstants.TOKEN_PREFIX;
```

```
public class JWTAuthenticationFilter extends UsernamePasswordAuthenticationFilter {
    private AuthenticationManager authenticationManager;

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

    @Override
    public Authentication attemptAuthentication(HttpServletRequest req,
                                                HttpServletResponse res) throws AuthenticationException {
        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 IOException, ServletException {

        String token = Jwts.builder()
                .setSubject(((User) auth.getPrincipal()).getUsername())
                .setExpiration(new Date(System.currentTimeMillis() + EXPIRATION_TIME))
                .signWith(SignatureAlgorithm.HS512, SECRET.getBytes())
                .compact();
        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 this user.

Our IDE will probably complain about the code in this class for two reasons. First, because the code imports four constants from a class that we haven't created yet, `SecurityConstants` . Second, because this class generates JWTS with the help of a class called `Jwts` , 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("io.jsonwebtoken:jjwt:0.7.0")
}
```

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.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 used later.

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.security` package:

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

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

import io.jsonwebtoken.Jwts;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.web.authentication.www.BasicAuthenticationFilter;

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.security.SecurityConstants.HEADER_STRING;
import static com.auth0.samples.authapi.security.SecurityConstants.SECRET;
import static com.auth0.samples.authapi.security.SecurityConstants.TOKEN_PREFIX;

public class JWTAuthorizationFilter extends BasicAuthenticationFilter {

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

    @Override
    protected void doFilterInternal(HttpServletRequest req,
                                    HttpServletResponse res,
                                    FilterChain chain) throws IOException, ServletException {
        String header = req.getHeader(HEADER_STRING);

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

        UsernamePasswordAuthenticationToken authentication = getAuthentication(req);

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

    private UsernamePasswordAuthenticationToken getAuthentication(HttpServletRequest request) {
        String token = request.getHeader(HEADER_STRING);
        if (token != null) {
            // parse the token.
            String user = Jwts.parser()
                    .setSigningKey(SECRET.getBytes())
                    .parseClaimsJws(token.replace(TOKEN_PREFIX, ""))
                    .getBody()
                    .getSubject();
```

```
            if (user != null) {
                return new UsernamePasswordAuthenticationToken(user, null, new ArrayList<>());
            }
            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 `Jwts` 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.security` package:

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

import org.springframework.http.HttpMethod;
import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.core.userdetails.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.annotation.Bean;

import static com.auth0.samples.authapi.security.SecurityConstants.SIGN_UP_URL;

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

    public WebSecurity(UserDetailsService userDetailsService, BCryptPasswordEncoder bCryptPasswordEncoder) {
        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(SessionCreationPolicy.STATELESS);
    }

    @Override
    public void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.userDetailsService(userDetailsService).passwordEncoder(bCryptPasswordEncoder);
    }

  @Bean
  CorsConfigurationSource corsConfigurationSource() {
    final UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
    source.registerCorsConfiguration("/**", new CorsConfiguration().applyPermitDefaultValues());
    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.user` package to provide one:

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

import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
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 applicationUserRepository) {
        this.applicationUserRepository = applicationUserRepository;
    }

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

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 be publicly exposed and that we can support authentication and authorization with JWTS on Spring Boot properly. To check everything, let's run our application (through the IDE or through `gradle bootRun`) and issue the following requests:

```
# 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
```

Aside: Securing Spring APIs with Auth0

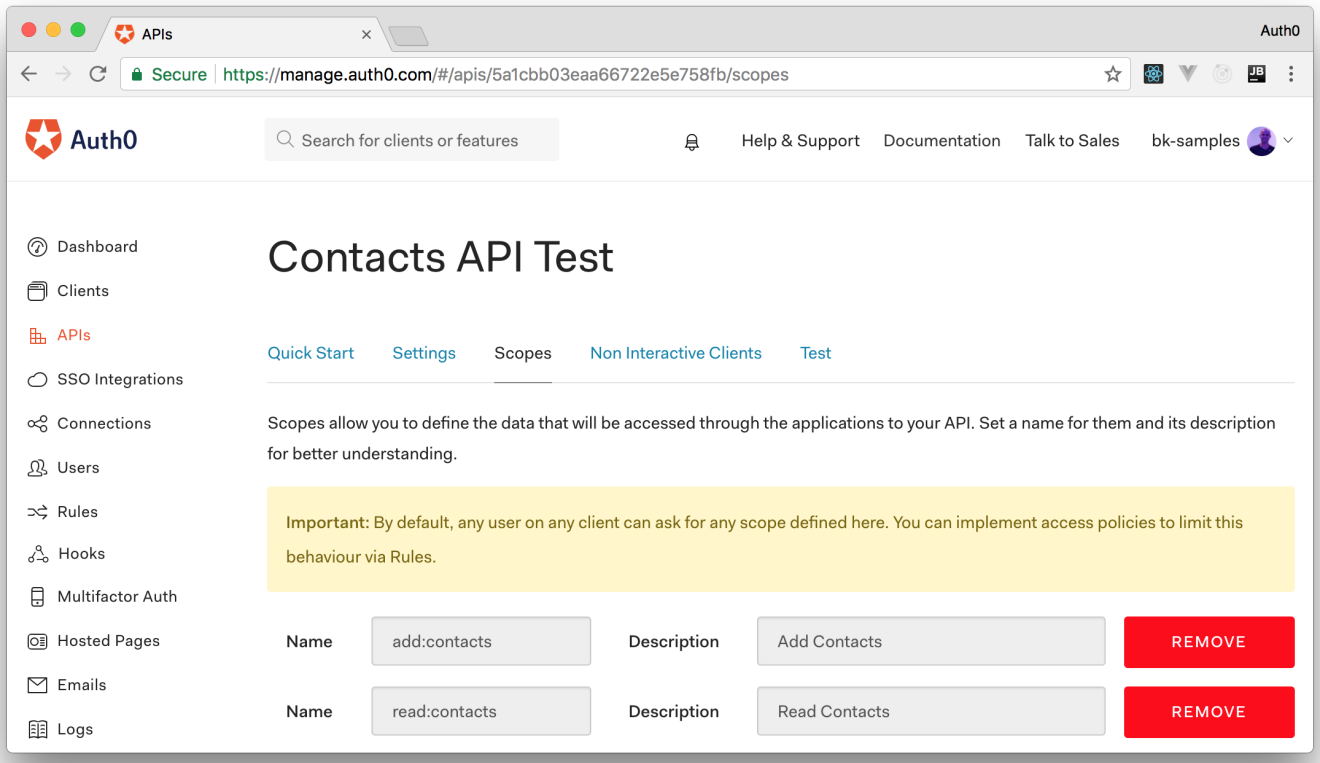
Securing applications with Auth0 is very easy and brings a lot of great features to the table. With Auth0, we only have to write a few lines of code to get solid identity management solution, single sign-on, support for social identity providers (like Facebook, GitHub, Twitter, etc.), and support for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

In the following sections, we are going to learn how to use Auth0 to secure Spring APIs. As we will see, the process is simple and fast.

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.mycompany.com` (we will use this value later).

After creating it, we have to go to the "Scopes" tab of the API and define the desired scopes. For this sample, we will define two scopes: `read:contacts` and `add:contacts`. They will represent two different operations (read and add) over the same entity (contacts).



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.EnableGlobalMethodSecurity;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

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

`auth0.issuer` : This is our domain at Auth0, including the HTTP protocol. For example: `https://bk-samples.auth0.com/` .

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

```
auth0.issuer:https://bk-samples.auth0.com/
auth0.apiAudience:https://contacts.mycompany.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").build(),
            Contact.builder().name("John Doe").phone("+5551888884444").build()
    );

    @GetMapping
    @PreAuthorize("hasAuthority('read:contacts')")
    public List<Contact> getContacts() {
        return contacts;
    }

    @PostMapping
    @PreAuthorize("hasAuthority('add:contacts')")
    public void addContact(@RequestBody Contact contact) {
        contacts.add(contact);
    }
}
```

Note that the integration allows us to use the `hasAuthority` Spring EL Expression to restrict access to endpoints based on the `scope` of the `access_token`. Let's see how to get this token now.

Creating an Auth0 Client

As the focus of this section is to secure Spring APIs with Auth0, we are going to use a live Angular app that has a configurable Auth0 client. To use this app we need to create an Auth0 Client that represents it. Let's head to the Clients section of the management dashboard and click on the "Create Client" button to create this client.

On the popup shown, let's set the name of this new client as "Contacts Client" and choose "Single Page Web App" as the client type. After hitting the "Create" button, we have to go to the "Settings" tab of this client and change two properties. First, we have to set `http://auth0.digituz.com.br/` in the "Allowed Web Origins" property. Second, we have to set `http://auth0.digituz.com.br/callback` in the "Allowed Callback URLs" property.

That's it, we can save the client and head to the sample Angular app secured with Auth0. On it, we just need to set the correct values to the four properties:

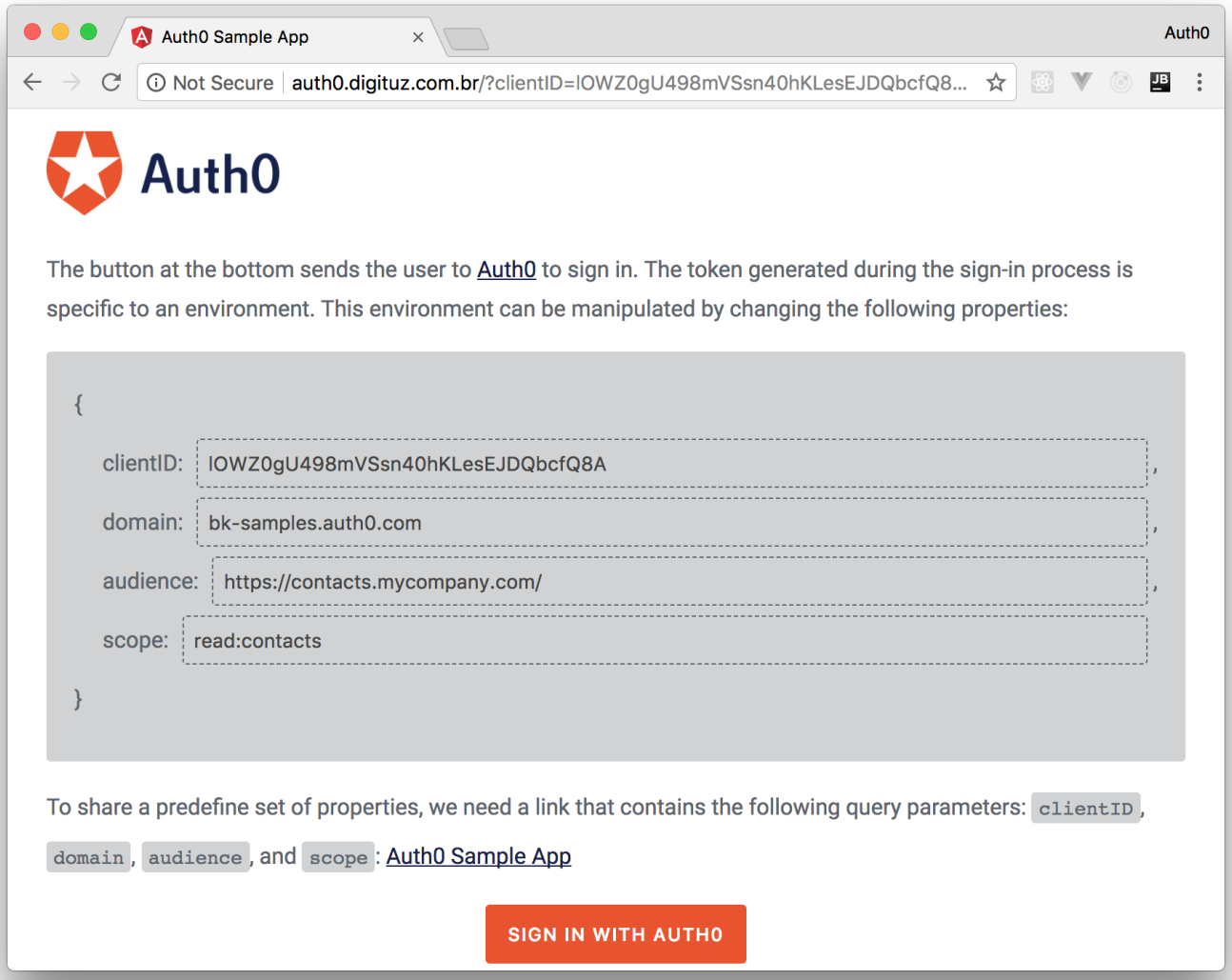
`clientID` : We have to copy this value from the "Client ID" field of the "Settings" tab of "Contacts Client".

`domain` : We can also copy this value from the "Settings" tab of "Contacts Client".

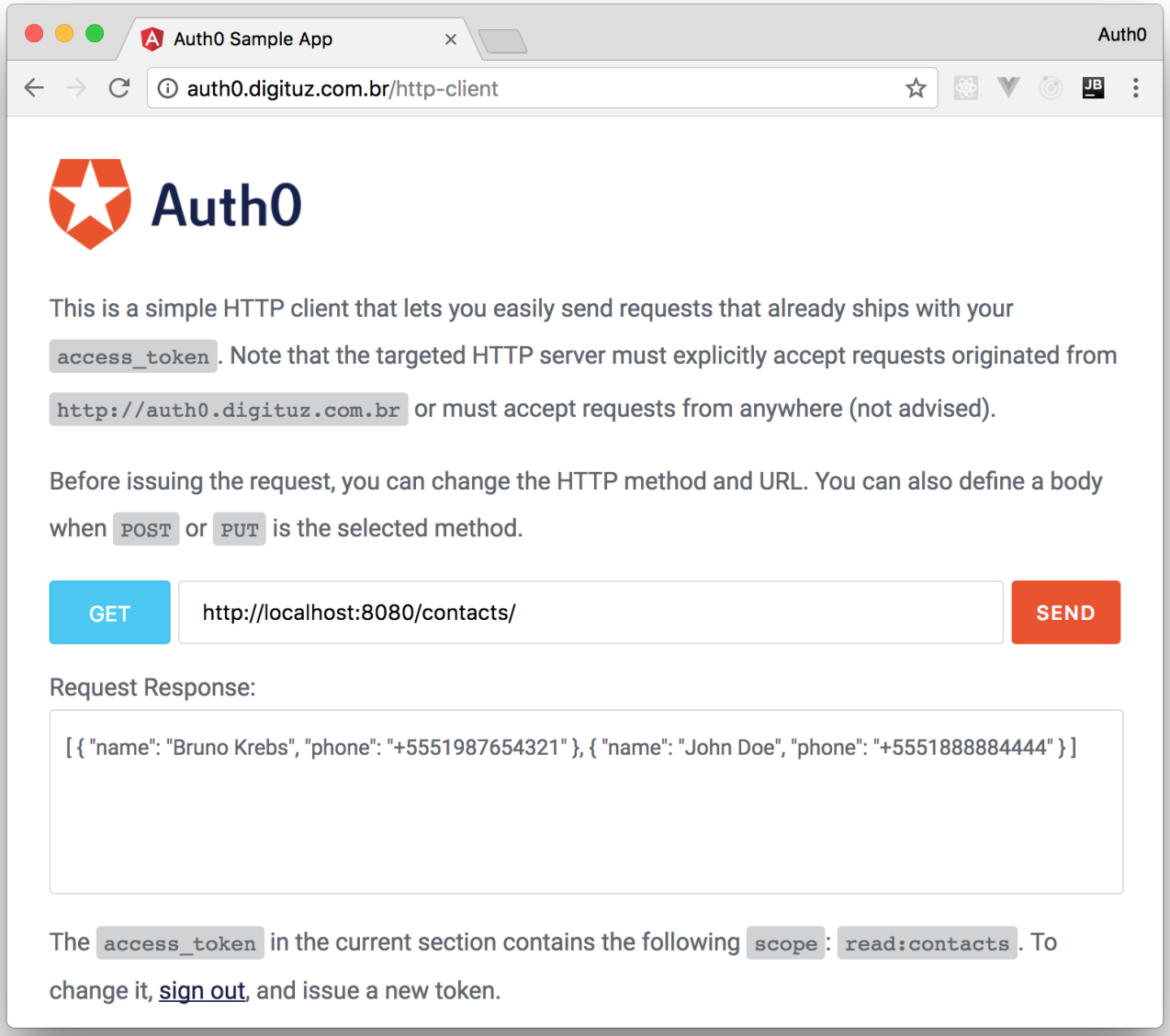
`audience` : We have to set this property to meet the identifier of the "Contacts API" that we created earlier.

`scope` : This property will define the `authority` that the `access_token` will get access to in the backend API. For example: `read:contacts` or both `read:contacts add:contacts` .

Then we can hit the "Sign In with Auth0" button.



After signing in, we can use the application to submit requests to our secured Spring API. For example, if we issue a GET request to `http://localhost:8080/contacts/`, the Angular app will include the `access_token` in the `Authorization` header and our API will respond with a list of contacts.



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.

135 Comments Auth0 Blog Login

Recommend 9 Share Sort by Best

Join the discussion…

LOG IN WITH OR SIGN UP WITH DISQUS

Name

Ivan Eggel • 3 months ago

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 requests even pass if there is no authorization header present in the request header.

Maybe you would consider extending the method 'configure(HttpSecurity http)' in the WebSecurity class with the following code:
http.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);

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

3 • Reply • Share ›

Bruno Krebs ➚ Ivan Eggel • 3 months ago

Thanks for the contribution, I will take a look.

• Reply • Share ›

Naresh Thota ➚ Bruno Krebs • 2 months ago

any work around on above issue ?

• Reply • Share ›

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

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

• Reply • Share ›

Nguyễn Thanh Bình ➚ Bruno Krebs • 22 days ago

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

• Reply • Share ›

Bruno S. Krebs Mod ➚ Nguyễn Thanh Bình • 21 days ago

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

• Reply • Share ›

Nguyễn Thanh Bình ➚ Bruno S. Krebs • 21 days ago

Could you please improve this solution by adding refresh token?

• Reply • Share ›

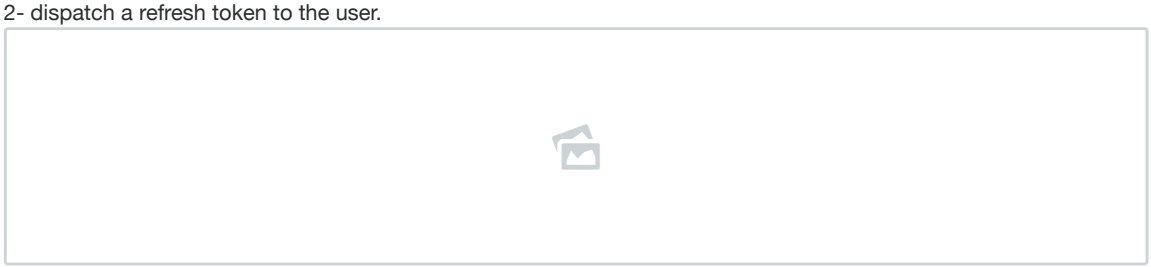
Photon Point ➚ Nguyễn Thanh Bình • 11 days ago

1- catch expired token to return information to the user.

Reply • Share ›

Photon Point ➚ Photon Point • 11 days ago

2- dispatch a refresh token to the user.



Reply • Share ›

Photon Point ➚ Photon Point • 11 days ago

3- Make two WebSecurityConfigurerAdapter instance. and add to customfilters

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

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

@Configuration
@Order(0) // first filter
public static class RestApi2WebSecurityConfigurerAdapter extends WebSecurityConfigurerAdapter { // add other custom filters}
```

Reply • Share ›

Avatar This comment was deleted.

Photon Point ➚ Guest • 11 days ago

Organize second filter in compliance with your purpose.

SecurityContextHolder.getContext().setAuthentication(authentication); // 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 ›

Mojtaba • 4 months ago

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

2 Reply • Share ›

Angel Casapía ➚ Mojtaba • a month ago

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

public class ExpiredJwtExceptionextends 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 months ago

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.ExpiredJwtException",
    "message": "JWT expired at 2017-09-06T22:17:45Z. Current time: 2017-09-06T22:18:03Z, a difference of 18222 millisec
    "path": "/tasks"
}
```

Reply • Share ›

Tautvydas T. • a month ago

Hi. When I try to access API without token I get 403. Shouldn't I get 401? How to configure that?

1 Reply • Share ›

Tautvydas T. ➚ Tautvydas T. • a month ago

Fixed it by adding Http401AuthenticationEntryPoint to security config. Btw, great tutorial!

1 Reply • Share ›

Asiel Leal Celdeiro ➚ Tautvydas T. • 14 days ago

Please, could you share with us how did you do it?

Reply • Share ›

Asiel Leal Celdeiro ➚ Asiel Leal Celdeiro • 14 days ago

In case any one find it useful.. it can be done this way:

....
.and()
.exceptionHandling()
.authenticationEntryPoint(new Http401AuthenticationEntryPoint("headerName"));
.....

Reply • Share ›

Bruno S. Krebs Mod ➚ Tautvydas T. • a month ago

Or you could change the JWTAuthorizationFilter as shown here.

Reply • Share ›

Jason Kurniawan • 2 months ago

Hi great tutorial, I wonder how we can get current authenticated user's (i.e the id) from the JWToken ?
Thanks

1 Reply • Share ›

Khomotso Glen Mfubha ➚ Jason Kurniawan • a month ago

Then from this you can use findByUsername() on the repository to find related information about the user

Reply • Share ›

Alex • 4 months ago

This is fantastic and it works.
Do you think there is any way to make the /login post return the user object? how can I get the user info/ profile from the ui (made in js)? or how can I know which user I Am from the token?
1 • Reply • Share ›

Luiz Eduardo ➚ Alex • 4 months ago
you can always create an endpoint that return this information :)
• Reply • Share ›

Alex ➚ Luiz Eduardo • 4 months ago
Right, but how can I know which user am I from my token?
• Reply • Share ›

adobot Mod ➚ Alex • 4 months ago
It depends on what information is stored in the token. If the token contains user info, then you can just decode the token and it should tell you, otherwise, if you just have a user identifier in the token then you would need to request user information from your database and then return in. Does that make sense?
• Reply • Share ›

Alex ➚ adobot • 4 months ago
Yes, thank you! I'll try that.
Now there is another thing I'd appreciate if you can explain briefly. Maybe i'm not fully understanding this.

The /login entrypoint is allowed so the client can sign-in and get a token. But I've saw that the client sends two requests. 1st OPTIONS and 2nd POST (this is the right one).

And the issue is that the JWTLoginFilter is running also through the OPTIONS request which has no username/ login info, which makes the server throw a nullpointer exception.

What's the right way to deal with that? know what I mean?

Thanks in advance and thank you very much, your post is really useful.
• Reply • Share ›

Arpan Sharma ➚ Alex • 12 days ago
Hi Alex
Can you please explain why only on /login the user is checked for authentication and not on other url example - /login2
Is there a way i can fine tune.....
• Reply • Share ›

Bruno S. Krebs Mod ➚ Arpan Sharma • 7 days ago
Not sure I understood, perhaps this is what you want?
1 • Reply • Share ›

Arpan Sharma ➚ Bruno S. Krebs • 6 days ago
Same thing-
JWTAuthFilter authFilter=new JWTAuthFilter(authenticationManager());
authFilter.setRequiresAuthenticationRequestMatcher(
new AntPathRequestMatcher("/login2","POST"));

Sir great work :-)
Please do come up with these great posts regularly!!
• Reply • Share ›

Bruno Krebs ➚ Alex • 4 months ago
Hi Alex, take a look at this commit. It solves this issue that you are facing.

BTW, I'm going to update the post, thanks for pointing out the problem.
• Reply • Share ›

Arpan Sharma ➚ Bruno Krebs • 12 days ago
Hi Bruno-If any workaround is there Please suggest

```
this.service.getToken(this.post).subscribe(
(res:Response) => {
});
```

I am getting the Authorization Header in Network tab but in Angular 5 i am getting - "Http failure during parsing for http://localhost:8080/login"
I have set the content-type for response as application/json in filter also.

Reply • Share ›

Bruno S. Krebs Mod ➚ Arpan Sharma • 7 days ago

hard to pin point without seeing the big picture, if you share a sample on a GitHub repo I might be able to help

Reply • Share ›

sarbe • 5 days ago

getting null pointer exception. I think CompressionCodecec is missing while building the token on successful login. can anybody help

Reply • Share ›

Bruno S. Krebs Mod ➚ sarbe • 4 days ago

I really don't think that this is the problem. Please, provide more information about your error, or share a github repo with your code.

Reply • Share ›

sarbe ➚ sarbe • 4 days ago

i think i am running the application on windows and by default compressioncodec not available on windows machine. can someone tell how to install it

Reply • Share ›

Witold Kupś • 7 days ago

Hi, very good article... However, i cloned your repo, switched to secure and tried to run... and then i failed because when i tried to create a user, i got "Access Denied". There is a comment on your commit, and i admit that even when there is "permitAll", request goes through filters. Any idea how to skip this?

Reply • Share ›

Bruno S. Krebs Mod ➚ Witold Kupś • 7 days ago

Looks like I forgot to keep it up to date. Sorry, now it's looking good.

Reply • Share ›

Witold Kupś ➚ Bruno S. Krebs • 6 days ago

Thank you very much, looks good :)

Reply • Share ›

Asiel Leal Celdeiro • 10 days ago

Hi, bruno, thanks for the post!

Could you, please assist me with this:

I'm receiving this error:
` Key bytes may only be specified for HMAC signatures. If using RSA or Elliptic Curve, use the signWith(SignatureAlgorithm, Key) method instead.`

...as far as I read (https://stackoverflow.com/q... and https://stackoverflow.com/q...) it is related to this code:

`.signWith(SignatureAlgorithm.HS512, SECRET.getBytes())`

I'm using version 0.9.0 of io.jsonwebtoken:jjwt

The best,
Asiel

Reply • Share ›

Bruno S. Krebs Mod ➚ Asiel Leal Celdeiro • 7 days ago

Do you have a sample repo (GitHub) that I can take a look at?

Reply • Share ›

Asiel Leal Celdeiro ➚ Bruno S. Krebs • 4 days ago

Found the problem! (I haven't had time till today to look closely).
Here the problem:
The post says:
`.signWith(SignatureAlgorithm.HS512, SECRET.getBytes())`

and I was doing:
`.signWith(SignatureAlgorithm.ES512, SECRET.getBytes())`

Notice the difference on the SignatureAlgorithm from `HS512` to `ES512`. That was the problem.

Again. Thanks for your valuable time ;)

Reply • Share ›

Asiel Leal Celdeiro ➚ Bruno S. Krebs • 7 days ago

This is a "real" project where the error persist (https://github.com/lealceld...) and this is a demo I made from the code of the "real" project which (for my surprise) actually works (https://github.com/lealceld...). In both cases there is a `JWTAuthenticationFilter.java` class in the `config.security` package.

In the demo the `loadUserByUsername` method returns a fake user with credentials (username: admin, password: admin). In the first project a default user is created in the DB with the same credentials and there is a lot much more code but basically it should work as the demo, but for some reason it gives that error I mentioned before, despite the `signWith` method receives as well `SignatureAlgorithm.ES512` as first argument and a `string.getBytes()` as second argument.

I'll really appreciate some insight on this.

Thanks in advance!

Reply • Share ›

Pham Lan • 16 days ago

Great post. Thank you very much,Bruno

Reply • Share ›

张威 • 22 days ago

I have a question when user has logined how can I judge which user has logined?

Reply • Share ›

Pratik Singhal • 24 days ago

Thanks a lot! This works perfectly

Reply • Share ›

张威 • a month ago

@Bruno Krebs I had a new question.When user register I want to login and send token response not in '/login' is in '/user/signup' and response the token

Reply • Share ›

Bruno S. Krebs Mod ➚ 张威 • a month ago

Sorry, not sure if I understand your question. Perhaps this is what you want?

Reply • Share ›

张威 ➚ Bruno S. Krebs • 25 days ago

You know your article teatch me got token in "/login" but In fact our project want to got token in "/user/signup" so the "/user/signup" and the "/login" can be return token.

Reply • Share ›

Bruno S. Krebs Mod ➚ 张威 • 24 days ago

Ok, the link that I provided you shows that.

Reply • Share ›

Load more comments

Subscribe Add Disqus to your siteAdd DisqusAdd Privacy

Related Posts

Developing JSF applications with Spring Boot

Bruno Krebs

Java Platform and Java Community Process Overview

Bruno Krebs

Learn More ▸

PRODUCT

Pricing

Why Auth0

How It Works

COMPANY

About Us

Blog

Jobs

Press

SECURITY

Availability & Trust

Security

White Hat

LEARN

Help & Support

Documentation

Open Source

EXTEND

Lock

WordPress

API Explorer

CONTACT

10900 NE 8th St.
Suite 700
Bellevue, WA 98004

+1 (888) 235-2699
+1 (425) 312-6521
Support Center