Spring Security JWT Authentication & Authorization

Farhan Tanvir · Follow
Published in Code With Farhan · 9 min read · Jun 8, 2023

This tutorial will guide you to secure a Spring Boot application with JWT (JSON Web Token) Authentication & Authorization using Spring Security.

The source code of this tutorial is published in git :
https://github.com/farhantanvirtushar/spring-jwt

Creating A Spring Boot REST API

Create a spring bootapplication from Spring Initializr with Java version 17. Then add following dependency in **pom.xml** :

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

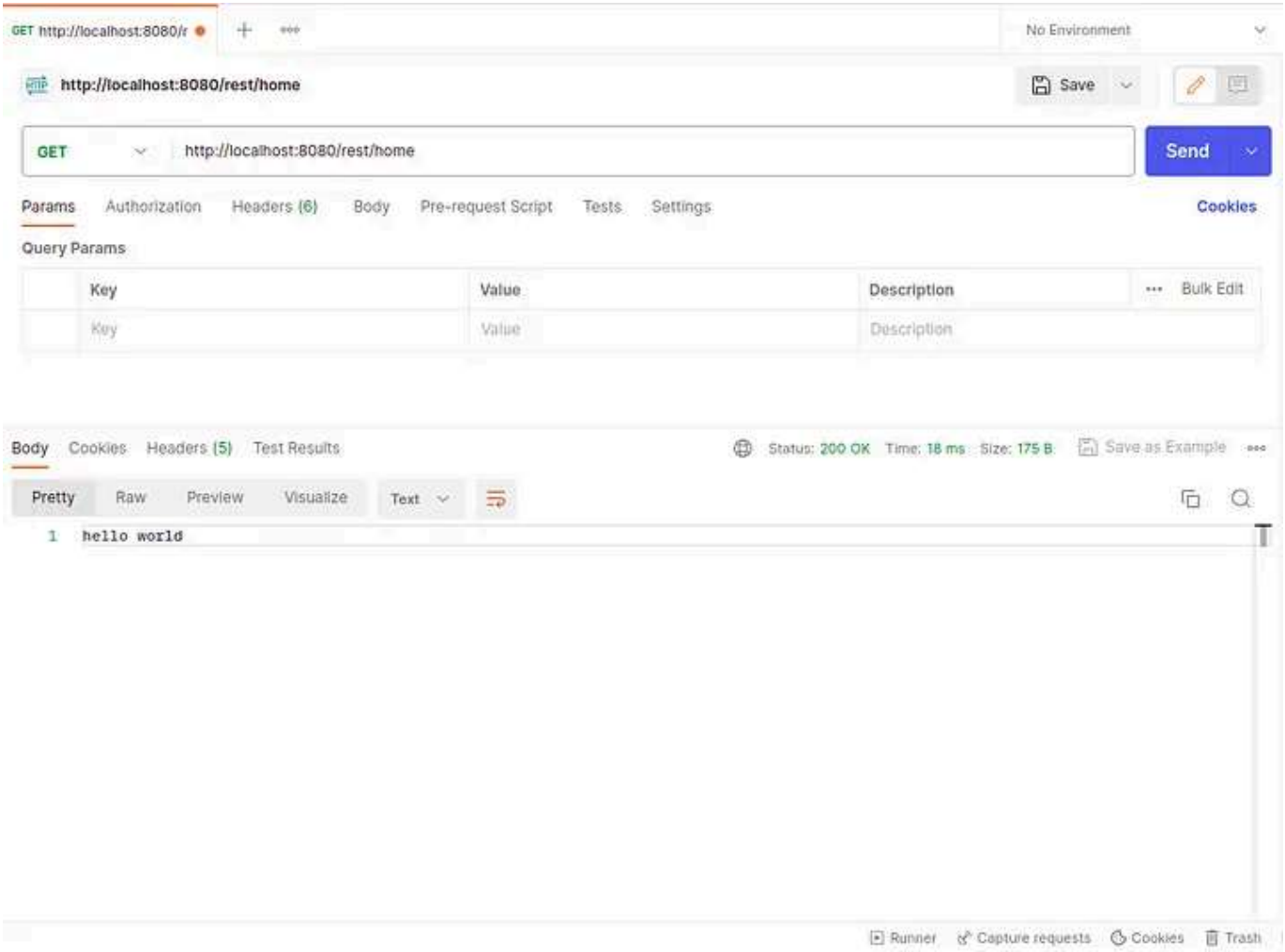
Create a new package named controllers. Then create a new controller named **HomeController.java** inside our newly created controllers package. Write the following code inside HomeController.java :

```
package com.example.springjwt.controllers;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
@RequestMapping("/rest/home")
public class HomeController {
    @ResponseBody
    @RequestMapping(value = "",method = RequestMethod.GET)
    public String hello(){
        return "hello world";
    }

}
```

We have created an api endpoint “**/rest/home**” for GET request. Now run the application through IDE or through command line with the following command :

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

Now open Postman and send a GET request to http://localhost:8080/rest/home



Securing Routes

Now we want to secure our REST api end points jwt authentication & authorization. Add the following dependencies in pom.xml file :

```
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-security</artifactId>
</dependency>

<dependency>
 <groupId>io.jsonwebtoken</groupId>
 <artifactId>jjwt</artifactId>
 <version>0.9.1</version>
</dependency>

<dependency>
```

```
    <groupId>javax.xml.bind</groupId>
    <artifactId>jaxb-api</artifactId>
    <version>2.3.0</version>
  </dependency>
```

We have added two dependencies, one for spring security & other for jwt.

Adding Model Classes

Next we will create some model class for API requests & responses. Create a package named model.request and add a model class LoginReq :

```
package com.example.springjwt.model.request;


public class LoginReq {
    private String email;
    private String password;

    public LoginReq(String email, String password) {
        this.email = email;
        this.password = password;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getPassword() {
        return password;
    }

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

```
    }
}
```

Create another package named model.response for API response & add two model class LoginRes.java & ErrorRes.java :

```
package com.example.springjwt.model.response;

public class LoginRes {
    private String email;
    private String token;

    public LoginRes(String email, String token) {
        this.email = email;
        this.token = token;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getToken() {
        return token;
    }

    public void setToken(String token) {
        this.token = token;
    }
}
```

```
package com.example.springjwt.model.response;

import org.springframework.http.HttpStatus;

public class ErrorRes {
    HttpStatus httpStatus;
```

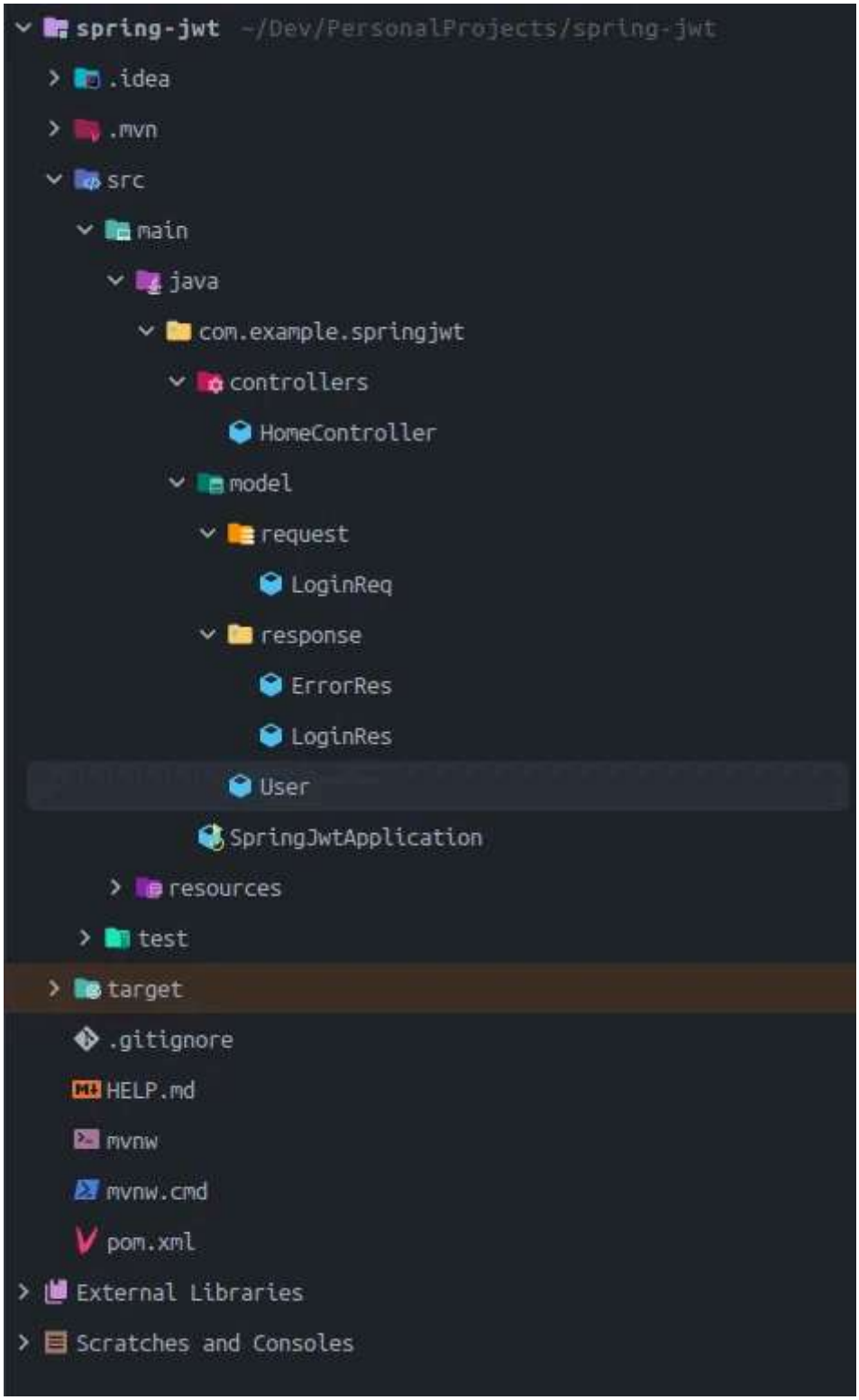
```
    String message;

    public ErrorRes(HttpStatus httpStatus, String message) {
        this.httpStatus = httpStatus;
        this.message = message;
    }

    public HttpStatus getHttpStatus() {
        return httpStatus;
    }

    public void setHttpStatus(HttpStatus httpStatus) {
        this.httpStatus = httpStatus;
    }

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }
}
```

Lastly, add a user model class inside model package named User.java for user login.

```
package com.example.springjwt.model;

public class User {
    private String email;
    private String password;
    private String firstName;
    private String lastName;

    public User(String email, String password) {
        this.email = email;
        this.password = password;
    }

    public String getEmail() {
        return email;
    }
```

```
    public void setEmail(String email) {
        this.email = email;
    }

    public String getPassword() {
        return password;
    }

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

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

Now our directory structure looks like this :



Adding Security Config & JWT Filter

Create a new package named “auth” and add a component class named **JwtUtil.java** . This class will be used for creating & resolving jwt tokens.

```
package com.example.springjwt.auth;

import com.example.springjwt.model.User;
import io.jsonwebtoken.*;
import jakarta.servlet.http.HttpServletRequest;
import org.springframework.security.core.AuthenticationException;
import org.springframework.stereotype.Component;

import java.util.Date;
import java.util.List;
import java.util.concurrent.TimeUnit;

@Component
public class JwtUtil {


    private final String secret_key = "mysecretkey";
    private long accessTokenValidity = 60*60*1000;

    private final JwtParser jwtParser;

    private final String TOKEN_HEADER = "Authorization";
    private final String TOKEN_PREFIX = "Bearer ";

    public JwtUtil(){
        this.jwtParser = Jwts.parser().setSigningKey(secret_key);
    }

    public String createToken(User user) {
        Claims claims = Jwts.claims().setSubject(user.getEmail());
        claims.put("firstName",user.getFirstName());
        claims.put("lastName",user.getLastName());
        Date tokenCreateTime = new Date();
        Date tokenValidity = new Date(tokenCreateTime.getTime() + TimeUnit.MINUT
        return Jwts.builder()
                .setClaims(claims)
                .setExpiration(tokenValidity)
                .signWith(SignatureAlgorithm.HS256, secret_key)
                .compact();
    }

    private Claims parseJwtClaims(String token) {
        return jwtParser.parseClaimsJws(token).getBody();
    }

    public Claims resolveClaims(HttpServletRequest req) {
        try {
            String token = resolveToken(req);
```

```
            if (token != null) {
                return parseJwtClaims(token);
            }
            return null;
        } catch (ExpiredJwtException ex) {
            req.setAttribute("expired", ex.getMessage());
            throw ex;
        } catch (Exception ex) {
            req.setAttribute("invalid", ex.getMessage());
            throw ex;
        }
    }

    public String resolveToken(HttpServletRequest request) {

        String bearerToken = request.getHeader(TOKEN_HEADER);
        if (bearerToken != null && bearerToken.startsWith(TOKEN_PREFIX)) {
            return bearerToken.substring(TOKEN_PREFIX.length());
        }
        return null;
    }

    public boolean validateClaims(Claims claims) throws AuthenticationException
        try {
            return claims.getExpiration().after(new Date());
        } catch (Exception e) {
            throw e;
        }
    }

    public String getEmail(Claims claims) {
        return claims.getSubject();
    }

    private List<String> getRoles(Claims claims) {
        return (List<String>) claims.get("roles");
    }


}
```

Let us see what each method does in this class.

The **createToken()** method takes an User object , creates a **Claims** object from user data and builds a jwt token with **Jwts.builder()** . The **Claims** object is used as the jwt body.

The **resolveClaims()** method takes an **HttpServletRequest** object as parameter and resolve **Claims** object from **Bearer** token in the request header. It will through an exception if no such token is present in request header or the token is expired or invalid.

UserDetailsService & UserRepository

Spring security uses an interface called **UserDetailsService** to load user details and match the user with the user input. We have to implement our own custom user details service. We also need to create a repository to load user details from database. First create a repository named **UserRepository.java** in "**repositories**" package :

```
package com.example.springjwt.repositories;

import com.example.springjwt.model.User;
import org.springframework.stereotype.Repository;

@Repository
public class UserRepository {
    public User findUserByEmail(String email){
        User user = new User(email,"123456");
        user.setFirstName("FirstName");
        user.setLastName("LastName");
        return user;
    }
}
```

We created a method inside **UserRepository** class to find users by email addresses. This method will search a user by email from database and return it. In our example we are returning a static **User** object with password : “123456”. We are using a plain text password because we are not using any encoding for passwords. **UserDetailsService** will use this password to verify the user. Let’s create our custom **UserDetailsService.** Create a new class **CustomUserDetailsService** inside a new package “**services**” :

```
package com.example.springjwt.services;

import com.example.springjwt.model.User;
import com.example.springjwt.repositories.UserRepository;
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 java.util.ArrayList;
import java.util.List;

@Service
public class CustomUserDetailsService implements UserDetailsService {

    private final UserRepository userRepository;

    public CustomUserDetailsService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    @Override
    public UserDetails loadUserByUsername(String email) throws UsernameNotFoundE
        User user = userRepository.findUserByEmail(email);
        List<String> roles = new ArrayList<>();
        roles.add("USER");
        UserDetails userDetails =
                org.springframework.security.core.userdetails.User.builder()
                        .username(user.getEmail())
                        .password(user.getPassword())
                        .roles(roles.toArray(new String[0]))
                        .build();
        return userDetails;
```

```
        }
    }
```

We inject an **UserRepository** object using dependency injection and override the **loadUserByUsername()** method, which returns an UserDetails object. In our implementation, **loadUserByUsername()** get user by the email address from **UserRepository** , construct an **UserDetails** object from it and then returns. Spring security will internally call this method with user provided email, and then match the password from **UserDetails** object with user provided password.

SecurityConfig

Create a class inside auth package named **SecurityConfig.java:**

```
ckage com.example.springjwt.auth;

port com.example.springjwt.services.CustomUserDetailsService;
port org.springframework.context.annotation.Bean;
port org.springframework.context.annotation.Configuration;
port org.springframework.security.authentication.AuthenticationManager;
port org.springframework.security.config.annotation.authentication.builders.Authenti
port org.springframework.security.config.annotation.web.builders.HttpSecurity;
port org.springframework.security.config.annotation.web.configuration.EnableWebSecur
port org.springframework.security.config.http.SessionCreationPolicy;
port org.springframework.security.crypto.password.NoOpPasswordEncoder;
port org.springframework.security.web.SecurityFilterChain;

onfiguration
nableWebSecurity
blic class SecurityConfig  {

  private final CustomUserDetailsService userDetailsService;

  public SecurityConfig(CustomUserDetailsService customUserDetailsService) {
      this.userDetailsService = customUserDetailsService;
```

```
}
@Bean
public AuthenticationManager authenticationManager(HttpSecurity http, NoOpPassword
        throws Exception {
    AuthenticationManagerBuilder authenticationManagerBuilder = http.getSharedObje
    authenticationManagerBuilder.userDetailsService(userDetailsService).passwordEn
    return authenticationManagerBuilder.build();
}


@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception

    http.csrf().disable()
            .authorizeRequests()
            .requestMatchers("/rest/auth/**").permitAll()
            .anyRequest().authenticated()
            .and().sessionManagement().sessionCreationPolicy(SessionCreationPolicy

    return http.build();
}


@SuppressWarnings("deprecation")
@Bean
public NoOpPasswordEncoder passwordEncoder() {
    return (NoOpPasswordEncoder) NoOpPasswordEncoder.getInstance();
}
```

We used two annotations : “**@Configuration**” and “**@EnableWebSecurity**”. These annotations tells spring security to use our custom security configuration instead of default one. We created a Bean of **SecurityFilterChain** which implements our custom filter logic.

```
  http.csrf().disable()
                .authorizeRequests()
                .requestMatchers("/rest/auth/**").permitAll()
```

```
                .anyRequest().authenticated()
                .and().sessionManagement().sessionCreationPolicy(SessionCreation
```

This line tells spring security to permit all request without any authentication that starts with **"/rest/auth/"**. We will use this url prefix for login & registration api

We also created another Bean of **NoOpPasswordEncoder**. We are using this bean because we don't want any encoding (e.g. Bcrypt) to our passwords. That means, we are storing plain text into database as passwords. In practice, we don't store plain text passwords into database.

We created a Bean of **AuthenticationManager** which will be used to authenticate the user. We passed our **CustomUserDetailsService** object & password encoding object to **AuthenticationManager.**

Adding API Route For Login

We need to add an api for user login. Create a controller class named **AuthController.java** in controllers package.

```
package com.example.springjwt.controllers;

import com.example.springjwt.auth.JwtUtil;
import com.example.springjwt.model.User;
import com.example.springjwt.model.request.LoginReq;
import com.example.springjwt.model.response.ErrorRes;
import com.example.springjwt.model.response.LoginRes;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.BadCredentialsException;
import org.springframework.security.authentication.UsernamePasswordAuthenticatio
```

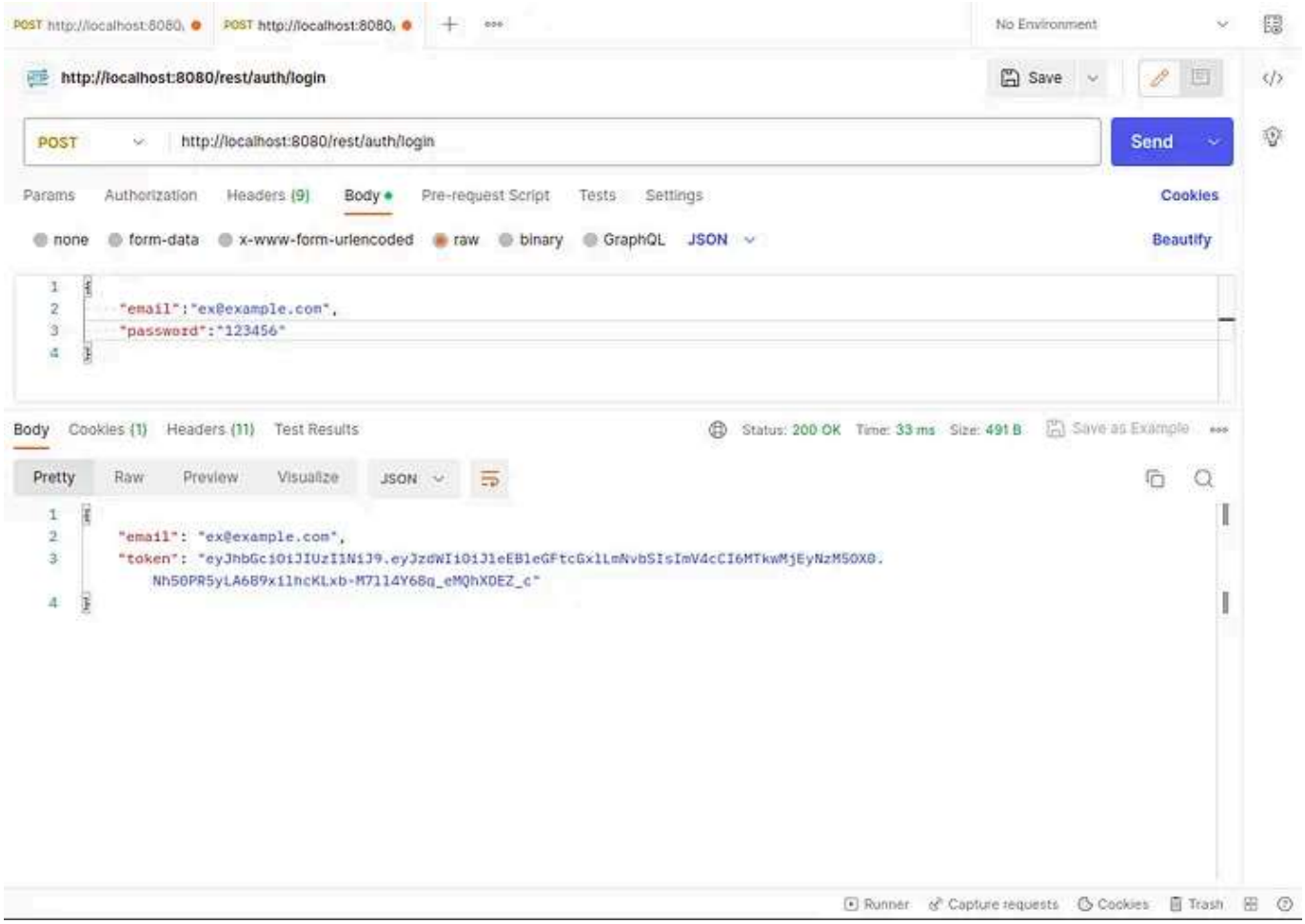
```
import org.springframework.security.core.Authentication;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.*;

@Controller
@RequestMapping("/rest/auth")
public class AuthController {

    private final AuthenticationManager authenticationManager;


    private JwtUtil jwtUtil;
    public AuthController(AuthenticationManager authenticationManager, JwtUtil j
        this.authenticationManager = authenticationManager;
        this.jwtUtil = jwtUtil;

    }

    @ResponseBody
    @RequestMapping(value = "/login",method = RequestMethod.POST)
    public ResponseEntity login(@RequestBody LoginReq loginReq)  {

        try {
            Authentication authentication =
                    authenticationManager.authenticate(new UsernamePasswordAuthe
            String email = authentication.getName();
            User user = new User(email,"");
            String token = jwtUtil.createToken(user);
            LoginRes loginRes = new LoginRes(email,token);

            return ResponseEntity.ok(loginRes);

        }catch (BadCredentialsException e){
            ErrorRes errorResponse = new ErrorRes(HttpStatus.BAD_REQUEST,"Invali
            return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(errorRespo
        }catch (Exception e){
            ErrorRes errorResponse = new ErrorRes(HttpStatus.BAD_REQUEST, e.getM
            return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(errorRespo
        }
    }
}
```

We created an api “**/rest/auth/login**” for login requests.

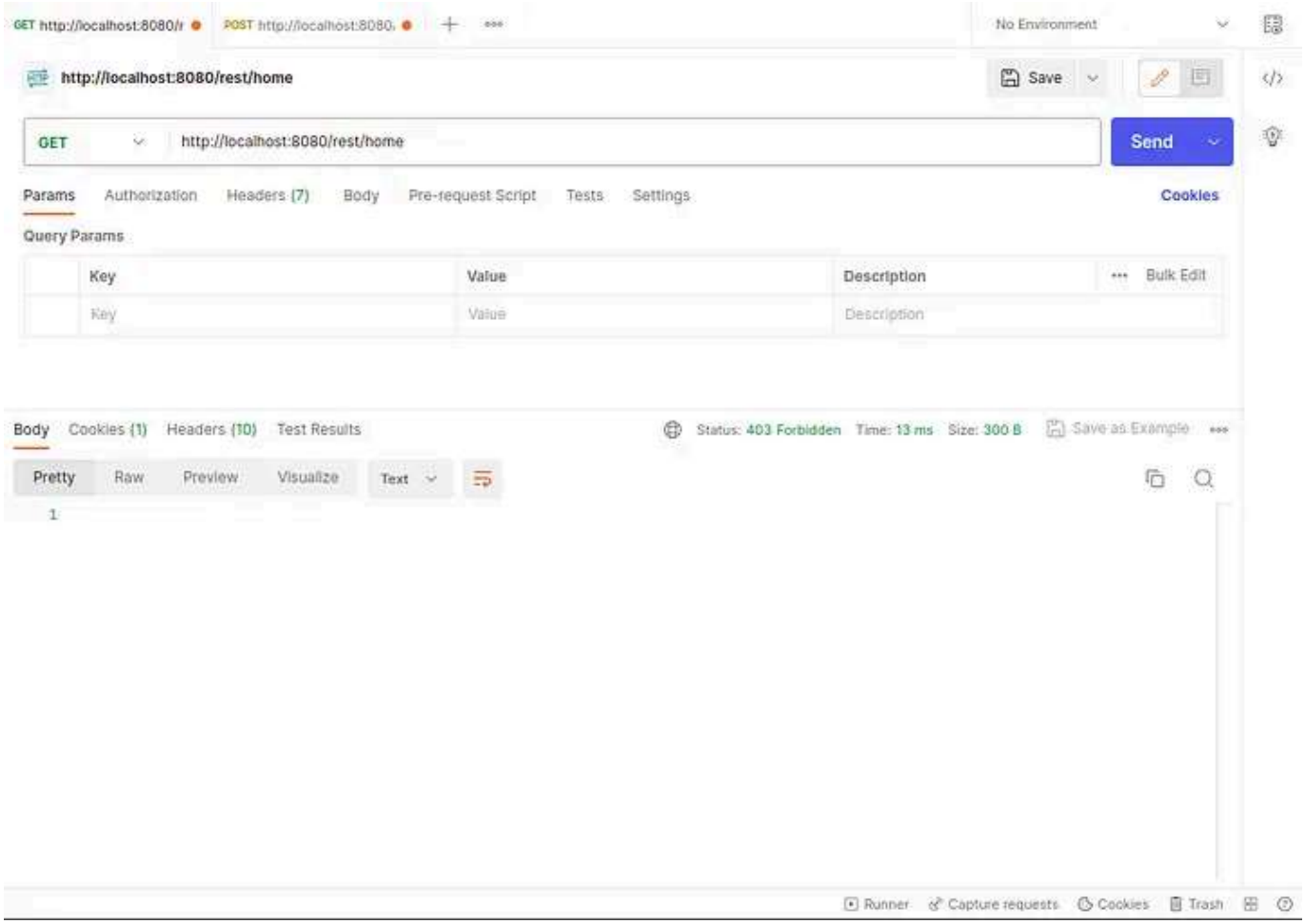
```
Authentication authentication =
                    authenticationManager.authenticate(new UsernamePasswordAuthe
            String email = authentication.getName();
```

This line authenticate the user with email & password. The **authenticationManager.authenticate()** method will internally call **loadUserByUsername()** method from our **CustomUserDetailsService** class. Then it will match the password from **userDetailsService** with the password found from **LoginReq**. This method will throw exception if the authentication is not successful.

Now build & run the application. Send a POST request from Postman to **"/rest/auth/login"** api. Add email & password "123456" in the request body.



We have successfully created a jwt token and sent back to user. Now if we want to access any other route, we will get an error:



We got a **403 Forbidden** error because this request is not authenticated. We need to add a jwt authorization filter for each request. This filter will block all requests that don't have jwt token in the request header.

Adding JwtAuthorizationFilter

Create a new class named **JwtAuthorizationFilter.java** inside auth package.

```
package com.example.springjwt.auth;

import com.fasterxml.jackson.databind.ObjectMapper;
import io.jsonwebtoken.Claims;
import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
```

```
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.security.authentication.AuthenticationServiceExceptio
import org.springframework.security.authentication.UsernamePasswordAuthenticatio
import org.springframework.security.core.Authentication;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;

import java.io.IOException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;

@Component
public class JwtAuthorizationFilter extends OncePerRequestFilter {

    private final JwtUtil jwtUtil;
    private final ObjectMapper mapper;

    public JwtAuthorizationFilter(JwtUtil jwtUtil, ObjectMapper mapper) {
        this.jwtUtil = jwtUtil;
        this.mapper = mapper;
    }
    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletRespo
        Map<String, Object> errorDetails = new HashMap<>();

        try {
            String accessToken = jwtUtil.resolveToken(request);
            if (accessToken == null ) {
                filterChain.doFilter(request, response);
                return;
            }
            System.out.println("token : "+accessToken);
            Claims claims = jwtUtil.resolveClaims(request);

            if(claims != null & jwtUtil.validateClaims(claims)){
                String email = claims.getSubject();
                System.out.println("email : "+email);
                Authentication authentication =
                        new UsernamePasswordAuthenticationToken(email,"",new Arr
                SecurityContextHolder.getContext().setAuthentication(authenticat
            }

        }catch (Exception e){
            errorDetails.put("message", "Authentication Error");
            errorDetails.put("details",e.getMessage());
            response.setStatus(HttpStatus.FORBIDDEN.value());
            response.setContentType(MediaType.APPLICATION_JSON_VALUE);
```

```
                mapper.writeValue(response.getWriter(), errorDetails);

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

We override the **doFilterInternal()** method. This method will be called for every request to out application. This method reads Bearer token from request headers and resolves claims. First, it checks if any access token is present in the request header. If the **accessToken** is null. It will pass the request to next filter chain. Any login request will not have jwt token in their header, therefore they will be passed to next filter chain. If any acessToken is present, then it will validate the token and then authenticate the request in **SecurityContext**.

```
    Authentication authentication =
                            new UsernamePasswordAuthenticationToken(email,"",new Arr
                    SecurityContextHolder.getContext().setAuthentication(authenticat
                }
```

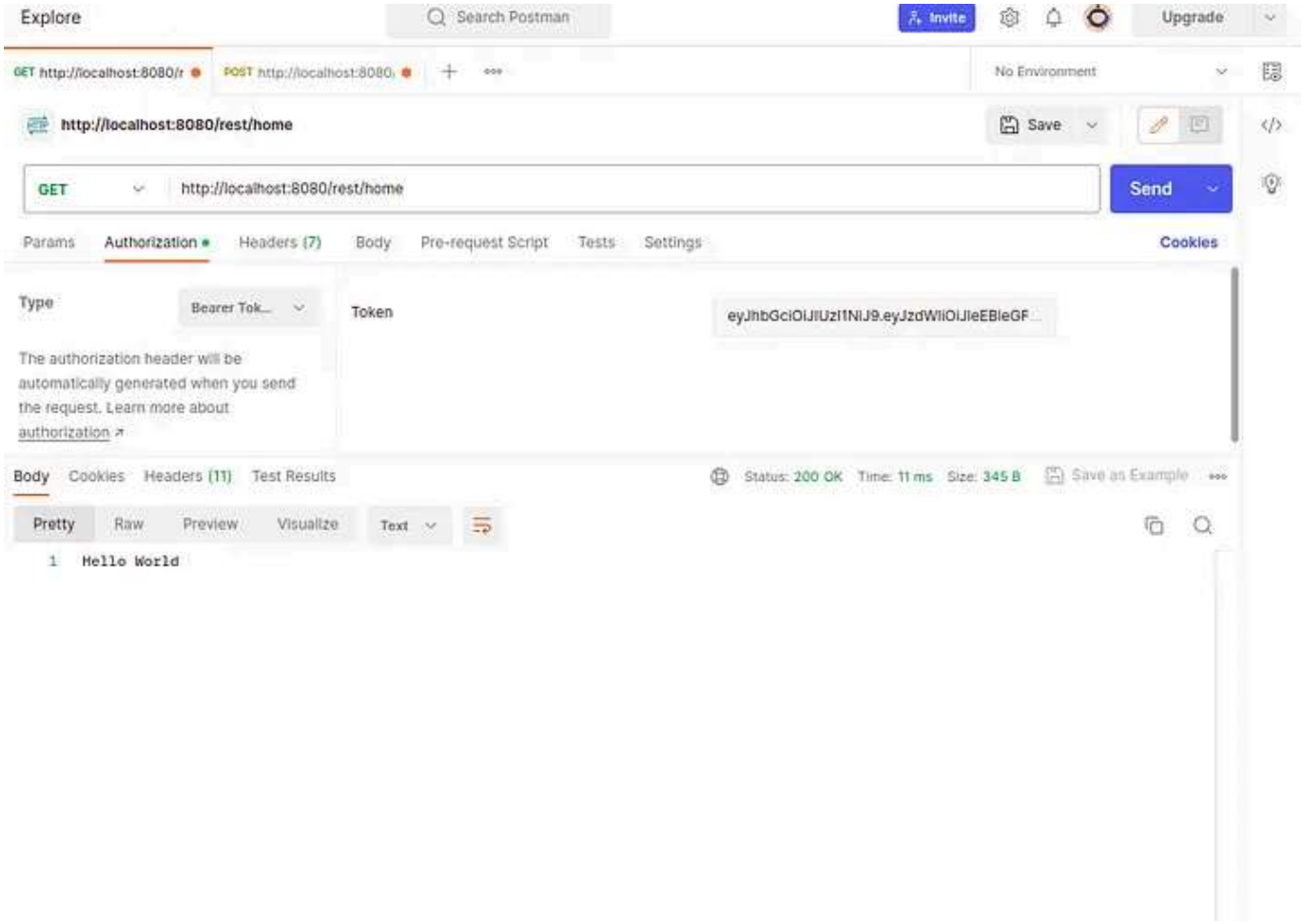
This line will authenticate the request to the **SecurityContext**. So, any request having a jwt token in their header will be authenticated & permited by spring security.

Finally we need to add this filter to our **SecurityConfig.java** class.

```
public class SecurityConfig  {

    private final CustomUserDetailsService userDetailsService;
    private final JwtAuthorizationFilter jwtAuthorizationFilter;

    public SecurityConfig(CustomUserDetailsService customUserDetailsService, Jwt
        this.userDetailsService = customUserDetailsService;
        this.jwtAuthorizationFilter = jwtAuthorizationFilter;

    }

    /.../

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exc

        http.csrf().disable()
                .authorizeRequests()
                .requestMatchers("/rest/auth/**").permitAll()
                .anyRequest().authenticated()
                .and().sessionManagement().sessionCreationPolicy(SessionCreation
                .and().addFilterBefore(jwtAuthorizationFilter,UsernamePasswordAu

        return http.build();
    }

    /.../

}
```

Here we have added our jwt filter before the **UsernamePasswordAuthenticationFilter.** Because we want every request to be authenticated before going through spring security filter.

Now add the Bearer token from postman & you will be able to acess any route.



Hire Me For Your Project

Java Spring Boot Spring Jwt Security

Published in Code With Farhan

11 Followers · Last published Jan 8, 2024

Follow

Hi, Im Farhan Tanvir. I help people turn their ideas into web apps. I provide solutions for business, companies, startups launch and grow their products with professional and cost-effective services.

Written by Farhan Tanvir

201 Followers · 6 Following

Follow

Software Engineer

Responses (11)

What are your thoughts?

Respond

이초다

about 1 year ago

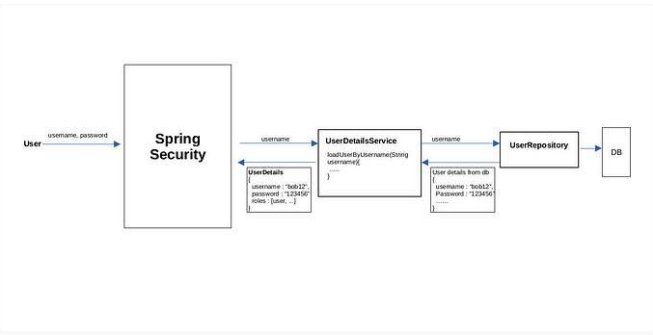
Thank you, you are my Savior

4

Reply

See all responses

More from Farhan Tanvir and Code With Farhan



In Code With Farhan by Farhan Tanvir

Spring Boot Database Connection : JDBC vs. JPA & Hibernate

This is a complete tutorial of how to use database in Spring Boot . I will discuss two...

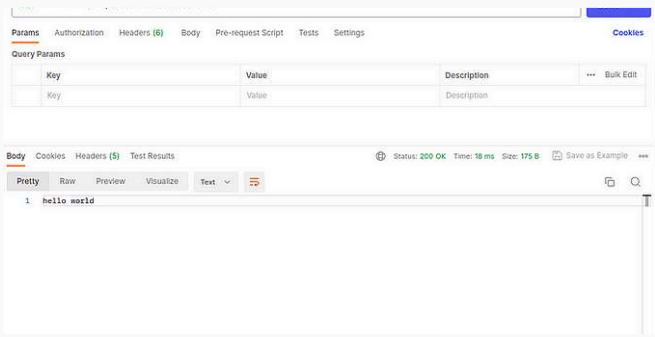
Jan 3 104 2

In Code With Farhan by Farhan Tanvir

Spring Security 2023

This tutorial will guide you to secure a Spring MVC application using Spring Security. You...

Feb 7, 2023 126 1



In Code With Farhan by Farhan Tanvir

Spring Boot REST API Full Tutorial

This is a complete tutorial of building a REST api with Spring Boot with. You will learn how...

Jul 12, 2023 96 2

In Open Physics Class by Farhan Tanvir

Faraday Cage : Why Are Cars & Planes Safe From Lighting

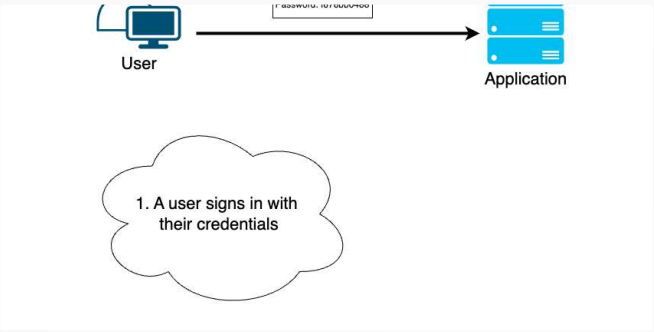
Have you ever wondered if you could survive if you were struck by lightning while you wer...

Nov 25, 2020 44 1

See all from Farhan Tanvir

See all from Code With Farhan

Recommended from Medium

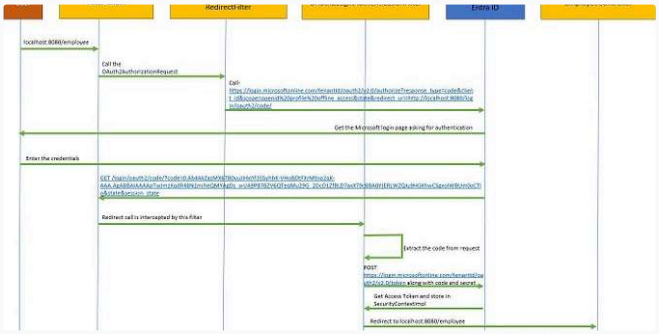


In Stackademic by Yileng Yao

JWT Authentication in Spring Security

From parent article

Aug 28 22



JavaInUse

Spring Boot Azure AD (Entra ID) OAuth 2.0 Authentication Example

In one of the previous OAuth 2 tutorial we had seen the different types of OAuth 2.0 flows. I...

Jul 26 12

Lists

General Coding Knowledge

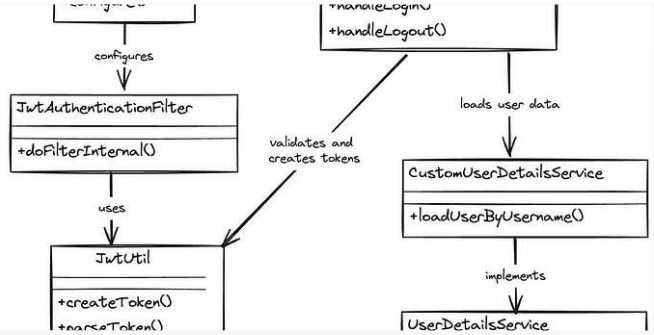
20 stories · 1803 saves

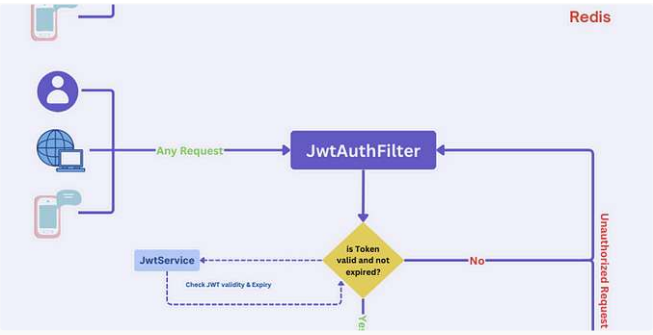
data science and AI

40 stories · 296 saves

Staff picks

780 stories · 1493 saves





Rima

Authentication Across Microservices

The most common and recommended approach in microservices architectures is...

Jul 16

In Level Up Coding by Zeeshan Adil

Invalidate/Blacklist the JWT using Redis: Logout Mechanism in Sprin...

I've put together a step-by-step guide on how to implement JWT in your application using...

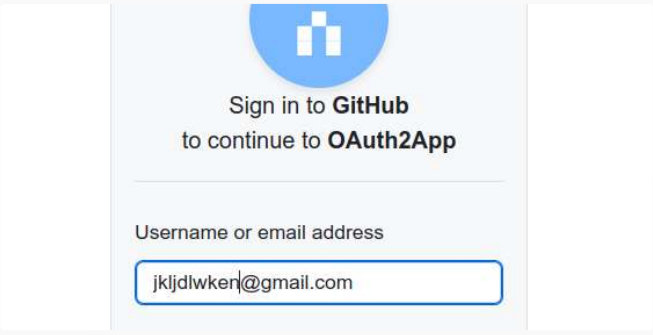
Aug 18 140

Nagarjun (Arjun) Nagesh

Spring Security with AWS Cognito using JWT Token

Introduction

Jun 16 7 1



Mohit Sehgal

Implementing OAuth2 Login via Github Authorization Server

Websites give us multiple options to sign-in. It may ask us to provide a username/password...

Jul 26 4

See more recommendations