

Spring Boot + Spring Security + JWT + MySQL + React Full Stack Polling app - Part 2



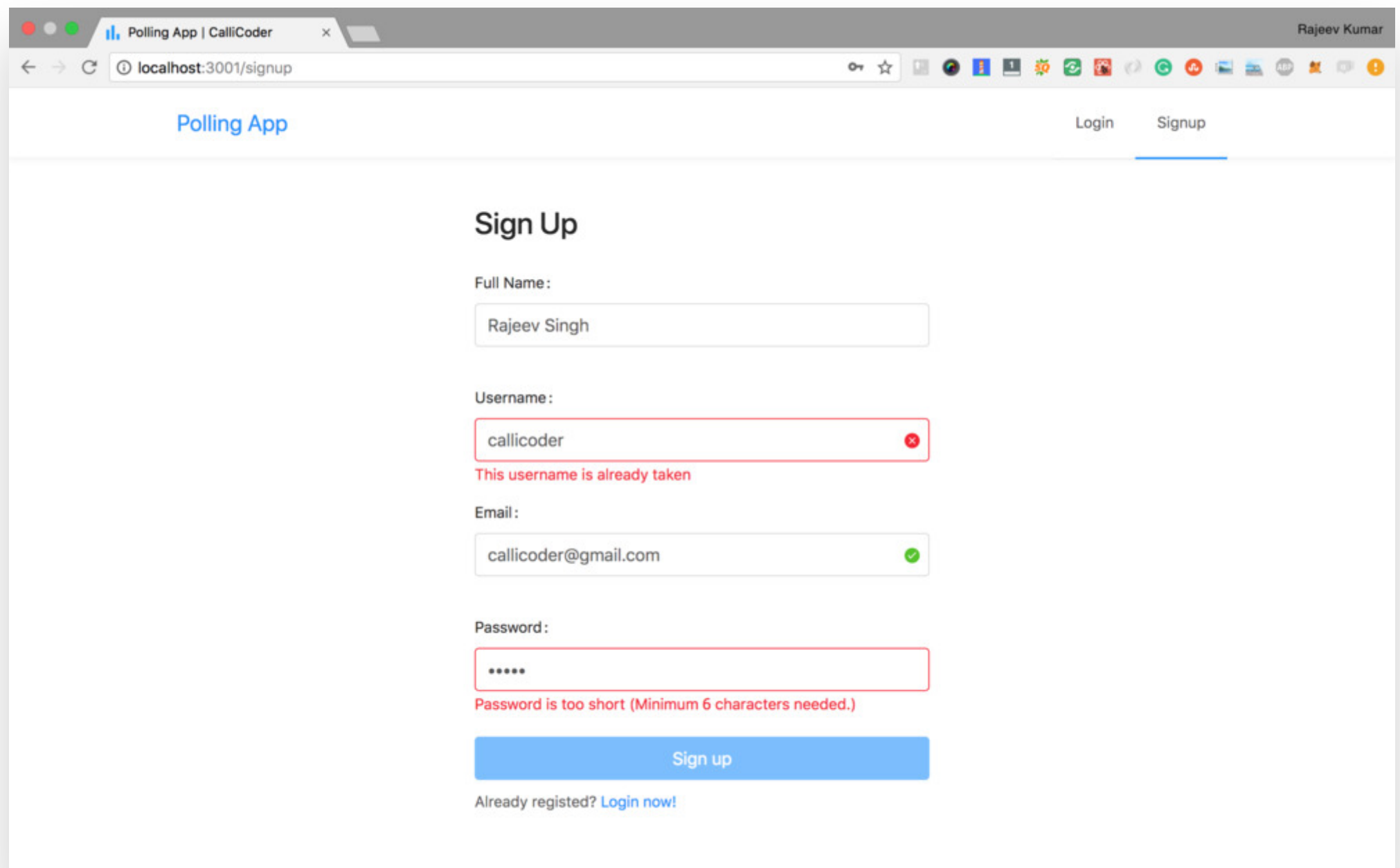
Rajeev Singh • Spring Boot • Feb 6, 2018 • 16 mins read

Welcome to the second part of my [full stack app development series](#) with Spring Boot, Spring Security, JWT, MySQL and React.

In the first part, we bootstrapped our project and created the basic domain models and repositories.

In this article, We'll configure Spring Security along with JWT authentication, and write the APIs to let users register and login to our application.

You can find the complete source code of the project on [Github](#).



An overview of the security mechanism that we're going to build

- Build an API that registers new users with their name, username, email and password.
- Build an API to let users log in using their username/email and password. After validating user's credentials, the API should generate a JWT authentication token and return the token in the response.

The clients will send this JWT token in the `Authorization` header of all the requests to access any protected resources.

- Configure Spring security to restrict access to protected resources. For example,
 - APIs for login, signup, and any static resources like images, scripts and stylesheets should be accessible to everyone.
 - APIs to create a poll, vote to a poll etc, should be accessible to authenticated users only.
- Configure Spring security to throw a 401 unauthorized error if a client tries to access a protected resource without a valid JWT token.

- require `ADMIN` role for creating a Poll, but you can easily change this behavior).
- Only users with role `USER` can vote in a `Poll`.

Configuring Spring Security and JWT

The following class is the crux of our security implementation. It contains almost all the security configurations that are required for our project.

Let's first create the following `SecurityConfig` class inside the package `com.example.polls.config`, and then we'll go through the code and learn what each configuration does -

```
package com.example.polls.config;

import com.example.polls.security.CustomUserDetailsService;
import com.example.polls.security.JwtAuthenticationEntryPoint;
import com.example.polls.security.JwtAuthenticationFilter;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.HttpMethod;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.config.BeanIds;
import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;
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;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;

@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(
    securedEnabled = true,
    jsr250Enabled = true,
    prePostEnabled = true
)
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    CustomUserDetailsService customUserDetailsService;

    @Autowired
    private JwtAuthenticationEntryPoint unauthorizedHandler;

    @Bean
    public JwtAuthenticationFilter jwtAuthenticationFilter() {
        return new JwtAuthenticationFilter();
    }

    @Override
    public void configure(AuthenticationManagerBuilder authenticationManagerBuilder) throws Exception {
        authenticationManagerBuilder
            .userDetailsService(customUserDetailsService)
            .passwordEncoder(passwordEncoder());
    }
}
```

```

@Override
public AuthenticationManager authenticationManagerBean() throws Exception {
    return super.authenticationManagerBean();
}

@Bean
public PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}

@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .cors()
            .and()
        .csrf()
            .disable()
        .exceptionHandling()
            .authenticationEntryPoint(unauthorizedHandler)
            .and()
        .sessionManagement()
            .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
            .and()
        .authorizeRequests()
            .antMatchers("/",
                "/favicon.ico",
                "/*/*.png",
                "/*/*.gif",
                "/*/*.svg",
                "/*/*.jpg",
                "/*/*.html",
                "/*/*.css",
                "/*/*.js")
                .permitAll()
            .antMatchers("/api/auth/**")
                .permitAll()
            .antMatchers("/api/user/checkUsernameAvailability", "/api/user/checkEmailAvailability")
                .permitAll()
            .antMatchers(HttpMethod.GET, "/api/polls/**", "/api/users/**")
                .permitAll()
            .anyRequest()
                .authenticated();

    // Add our custom JWT security filter
    http.addFilterBefore(jwtAuthenticationFilter(), UsernamePasswordAuthenticationFilter.class);
}
}

```

The above `SecurityConfig` class will show compilation errors in your IDE because we haven't yet defined many of the classes that are being used in it. We will define them one by one in this article.

But before that, Let's understand the meaning of the annotations and configurations used in the `SecurityConfig` class -

1. @EnableWebSecurity

This is the primary spring security annotation that is used to enable web security in a project.

methods -

- **securedEnabled:** It enables the `@Secured` annotation using which you can protect your controller/service methods like so -

```
@Secured("ROLE_ADMIN")
public User getAllUsers() {}

@Secured({"ROLE_USER", "ROLE_ADMIN"})
public User getUser(Long id) {}

@Secured("IS_AUTHENTICATED_ANONYMOUSLY")
public boolean isUsernameAvailable() {}
```

- **jsr250Enabled:** It enables the `@RolesAllowed` annotation that can be used like this -

```
@RolesAllowed("ROLE_ADMIN")
public Poll createPoll() {}
```

- **prePostEnabled:** It enables more complex expression based access control syntax with `@PreAuthorize` and `@PostAuthorize` annotations -

```
@PreAuthorize("isAnonymous()")
public boolean isUsernameAvailable() {}

@PreAuthorize("hasRole('USER')")
public Poll createPoll() {}
```

3. WebSecurityConfigurerAdapter

This class implements Spring Security's `WebSecurityConfigurer` interface. It provides default security configurations and allows other classes to extend it and customize the security configurations by overriding its methods.

Our `SecurityConfig` class extends `WebSecurityConfigurerAdapter` and overrides some of its methods to provide custom security configurations.

4. CustomUserDetailsService

To authenticate a User or perform various role-based checks, Spring security needs to load users details somehow.

For this purpose, It consists of an interface called `UserDetailsService` which has a single method that loads a user based on username-

```
UserDetails loadUserByUsername(String username) throws UsernameNotFoundException;
```

We'll define a `CustomUserDetailsService` that implements `UserDetailsService` interface and provides the implementation for `loadUserByUsername()` method.

Note that, the `loadUserByUsername()` method returns a `UserDetails` object that Spring Security uses for performing various authentication and role based validations.

In our implementation, We'll also define a custom `UserPrincipal` class that will implement `UserDetails` interface, and return the `UserPrincipal` object from `loadUserByUsername()` method.

5. JwtAuthenticationEntryPoint

6. JwtAuthenticationFilter

We'll use `JwtAuthenticationFilter` to implement a filter that -

- reads JWT authentication token from the `Authorization` header of all the requests
- validates the token
- loads the user details associated with that token.
- Sets the user details in Spring Security's `SecurityContext`. Spring Security uses the user details to perform authorization checks. We can also access the user details stored in the `SecurityContext` in our controllers to perform our business logic.

7. AuthenticationManagerBuilder and AuthenticationManager

`AuthenticationManagerBuilder` is used to create an `AuthenticationManager` instance which is the main Spring Security interface for authenticating a user.

You can use `AuthenticationManagerBuilder` to build in-memory authentication, LDAP authentication, JDBC authentication, or add your custom authentication provider.

In our example, we've provided our `customUserDetailsService` and a `passwordEncoder` to build the `AuthenticationManager`.

We'll use the configured `AuthenticationManager` to authenticate a user in the login API.

8. HttpSecurity configurations

The `HttpSecurity` configurations are used to configure security functionalities like `csrf`, `sessionManagement`, and add rules to protect resources based on various conditions.

In our example, we're permitting access to static resources and few other public APIs to everyone and restricting access to other APIs to authenticated users only.

We've also added the `JwtAuthenticationEntryPoint` and the custom `JwtAuthenticationFilter` in the `HttpSecurity` configuration.

Creating Custom Spring Security Classes, Filters, and Annotations

In the previous section, we configured spring security with many custom classes and filters. In this section, we'll define those classes one by one.

All the following custom security related classes will go inside a package named `com.example.polls.security`.

1. Custom Spring Security AuthenticationEntryPoint

The first spring security related class that we'll define is `JwtAuthenticationEntryPoint`. It implements `AuthenticationEntryPoint` interface and provides the implementation for its `commence()` method. This method is called whenever an exception is thrown due to an unauthenticated user trying to access a resource that requires authentication.

In this case, we'll simply respond with a 401 error containing the exception message.

```
package com.example.polls.security;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.security.core.AuthenticationException;
import org.springframework.security.web.AuthenticationEntryPoint;
import org.springframework.stereotype.Component;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
```

```
@Component
public class JwtAuthenticationEntryPoint implements AuthenticationEntryPoint {

    private static final Logger logger = LoggerFactory.getLogger(JwtAuthenticationEntryPoint.class);

    @Override
    public void commence(HttpServletRequest httpServletRequest,
                        HttpServletResponse httpServletResponse,
                        AuthenticationException e) throws IOException, ServletException {
        logger.error("Responding with unauthorized error. Message - {}", e.getMessage());
        httpServletResponse.sendError(HttpServletResponse.SC_UNAUTHORIZED, e.getMessage());
    }
}
```

2. Custom Spring Security UserDetails

Next, Let's define our custom `UserDetails` class called `UserPrincipal`. This is the class whose instances will be returned from our custom `UserDetailsService`. Spring Security will use the information stored in the `UserPrincipal` object to perform authentication and authorization.

Here is the complete `UserPrincipal` class -

```
package com.example.polls.security;

import com.example.polls.model.User;
import com.fasterxml.jackson.annotation.JsonIgnore;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;

import java.util.Collection;
import java.util.List;
import java.util.Objects;
import java.util.stream.Collectors;

public class UserPrincipal implements UserDetails {
    private Long id;

    private String name;

    private String username;

    @JsonIgnore
    private String email;

    @JsonIgnore
    private String password;

    private Collection<? extends GrantedAuthority> authorities;

    public UserPrincipal(Long id, String name, String username, String email, String password, Collection<? extends GrantedAuthority> authorities) {
        this.id = id;
        this.name = name;
        this.username = username;
        this.email = email;
        this.password = password;
    }
}
```



```
public static UserPrincipal create(User user) {
    List<GrantedAuthority> authorities = user.getRoles().stream().map(role ->
        new SimpleGrantedAuthority(role.getName().name())
    ).collect(Collectors.toList());

    return new UserPrincipal(
        user.getId(),
        user.getName(),
        user.getUsername(),
        user.getEmail(),
        user.getPassword(),
        authorities
    );
}

public Long getId() {
    return id;
}

public String getName() {
    return name;
}

public String getEmail() {
    return email;
}

@Override
public String getUsername() {
    return username;
}

@Override
public String getPassword() {
    return password;
}

@Override
public Collection<? extends GrantedAuthority> getAuthorities() {
    return authorities;
}

@Override
public boolean isAccountNonExpired() {
    return true;
}

@Override
public boolean isAccountNonLocked() {
    return true;
}

@Override
public boolean isCredentialsNonExpired() {
    return true;
}
```

```

    return true;
}

@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    UserPrincipal that = (UserPrincipal) o;
    return Objects.equals(id, that.id);
}

@Override
public int hashCode() {

    return Objects.hash(id);
}
}

```

3. Custom Spring Security UserDetailsService

Now let's define the custom `UserDetailsService` which loads a user's data given its username -

```

package com.example.polls.security;

import com.example.polls.model.User;
import com.example.polls.repository.UserRepository;
import org.springframework.beans.factory.annotation.Autowired;
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 org.springframework.transaction.annotation.Transactional;

@Service
public class CustomUserDetailsService implements UserDetailsService {

    @Autowired
    UserRepository userRepository;

    @Override
    @Transactional
    public UserDetails loadUserByUsername(String usernameOrEmail)
        throws UsernameNotFoundException {
        // Let people login with either username or email
        User user = userRepository.findByUsernameOrEmail(usernameOrEmail, usernameOrEmail)
            .orElseThrow(() ->
                new UsernameNotFoundException("User not found with username or email : " + usernameOrEmail)
            );

        return UserPrincipal.create(user);
    }

    // This method is used by JWTAuthenticationFilter
    @Transactional
    public UserDetails loadUserById(Long id) {

```



```

        return UserPrincipal.create(user);
    }
}

```

The first method `loadUserByUsername()` is used by Spring security. Notice the use of `findByUsernameOrEmail` method. This allows users to log in using either username or email.

The second method `loadUserById()` will be used by `JWTAuthenticationFilter` that we'll define shortly.

4. Utility class for generating and verifying JWT

The following utility class will be used for generating a JWT after a user logs in successfully, and validating the JWT sent in the Authorization header of the requests -

```

package com.example.polls.security;

import io.jsonwebtoken.*;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.security.core.Authentication;
import org.springframework.stereotype.Component;
import java.util.Date;

@Component
public class JwtTokenProvider {

    private static final Logger logger = LoggerFactory.getLogger(JwtTokenProvider.class);

    @Value("${app.jwtSecret}")
    private String jwtSecret;

    @Value("${app.jwtExpirationInMs}")
    private int jwtExpirationInMs;

    public String generateToken(Authentication authentication) {

        UserPrincipal userPrincipal = (UserPrincipal) authentication.getPrincipal();

        Date now = new Date();
        Date expiryDate = new Date(now.getTime() + jwtExpirationInMs);

        return Jwts.builder()
            .setSubject(Long.toString(userPrincipal.getId()))
            .setIssuedAt(new Date())
            .setExpiration(expiryDate)
            .signWith(SignatureAlgorithm.HS512, jwtSecret)
            .compact();
    }

    public Long getUserIdFromJWT(String token) {
        Claims claims = Jwts.parser()
            .setSigningKey(jwtSecret)
            .parseClaimsJws(token)

```

```

    return Long.parseLong(claims.getExpiration());
}

public boolean validateToken(String authToken) {
    try {
        Jwts.parser().setSigningKey(jwtSecret).parseClaimsJws(authToken);
        return true;
    } catch (SignatureException ex) {
        logger.error("Invalid JWT signature");
    } catch (MalformedJwtException ex) {
        logger.error("Invalid JWT token");
    } catch (ExpiredJwtException ex) {
        logger.error("Expired JWT token");
    } catch (UnsupportedJwtException ex) {
        logger.error("Unsupported JWT token");
    } catch (IllegalArgumentException ex) {
        logger.error("JWT claims string is empty.");
    }
    return false;
}
}

```

The utility class reads the JWT secret and expiration time from `properties` .

Let's add the `jwtSecret` and `jwtExpirationInMs` properties in the `application.properties` file -

JWT Properties

```

## App Properties
app.jwtSecret= JWTSuperSecretKey
app.jwtExpirationInMs = 604800000

```

5. Custom Spring Security AuthenticationFilter

Finally, Let's write the `JWTAuthenticationFilter` to get the JWT token from the request, validate it, load the user associated with the token, and pass it to Spring Security -

```

package com.example.polls.security;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.web.authentication.WebAuthenticationDetailsSource;
import org.springframework.util.StringUtils;
import org.springframework.web.filter.OncePerRequestFilter;
import javax.servlet.FilterChain;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

public class JwtAuthenticationFilter extends OncePerRequestFilter {

```

```

private CustomUserDetailsService customUserDetailsService;

@Autowired
private CustomUserDetailsService customUserDetailsService;

private static final Logger logger = LoggerFactory.getLogger(JwtAuthenticationFilter.class);

@Override
protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain) throws ServletException, IOException {
    try {
        String jwt = getJwtFromRequest(request);

        if (StringUtils.hasText(jwt) && tokenProvider.validateToken(jwt)) {
            Long userId = tokenProvider.getUserIdFromJWT(jwt);

            UserDetails userDetails = customUserDetailsService.loadUserById(userId);
            UsernamePasswordAuthenticationToken authentication = new UsernamePasswordAuthenticationToken(userDetails, null, userDetails.getAuthorities());
            authentication.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));

            SecurityContextHolder.getContext().setAuthentication(authentication);
        }
    } catch (Exception ex) {
        logger.error("Could not set user authentication in security context", ex);
    }

    filterChain.doFilter(request, response);
}

private String getJwtFromRequest(HttpServletRequest request) {
    String bearerToken = request.getHeader("Authorization");
    if (StringUtils.hasText(bearerToken) && bearerToken.startsWith("Bearer ")) {
        return bearerToken.substring(7, bearerToken.length());
    }
    return null;
}
}

```

In the above `filter`, We're first parsing the JWT retrieved from the `Authorization` header of the request and obtaining the user's Id. After that, We're loading the user's details from the database and setting the authentication inside spring security's context.

Note that, the database hit in the above `filter` is optional. You could also encode the user's username and roles inside JWT claims and create the `UserDetails` object by parsing those claims from the JWT. That would avoid the database hit.

However, Loading the current details of the user from the database might still be helpful. For example, you might wanna disallow login with this JWT if the user's role has changed, or the user has updated his password after the creation of this JWT.

6. Custom annotation to access currently logged in user

Spring security provides an annotation called `@AuthenticationPrincipal` to access the currently authenticated user in the controllers.

The following `currentUser` annotation is a wrapper around `@AuthenticationPrincipal` annotation.

```

package com.example.polls.security;

import org.springframework.security.core.annotation.AuthenticationPrincipal;
import java.lang.annotation.*;

```

```
@Documented
@AuthenticationPrincipal
public @interface CurrentUser {

}
```

We've created a meta-annotation so that we don't get too much tied up of with Spring Security related annotations everywhere in our project. This reduces the dependency on Spring Security. So if we decide to remove Spring Security from our project, we can easily do it by simply changing the `CurrentUser` annotation-

Writing the Login and Signup APIs

All right folks! We're done with all the security configurations that were required. It's time to finally write the login and signup APIs.

But, Before defining the APIs, we'll need to define the request and response payloads that the APIs will use. So let's define these payloads first.

All the request and response payloads will go inside a package named `com.example.polls.payload`.

Request Payloads

1. LoginRequest

```
package com.example.polls.payload;

import javax.validation.constraints.NotBlank;

public class LoginRequest {

    @NotBlank
    private String usernameOrEmail;

    @NotBlank
    private String password;

    public String getUsernameOrEmail() {
        return usernameOrEmail;
    }

    public void setUsernameOrEmail(String usernameOrEmail) {
        this.usernameOrEmail = usernameOrEmail;
    }

    public String getPassword() {
        return password;
    }

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

}
```

2. SignUpRequest

```
import javax.validation.constraints.*;

public class SignUpRequest {

    @NotBlank
    @Size(min = 4, max = 40)
    private String name;

    @NotBlank
    @Size(min = 3, max = 15)
    private String username;

    @NotBlank
    @Size(max = 40)
    @Email
    private String email;

    @NotBlank
    @Size(min = 6, max = 20)
    private String password;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getUsername() {
        return username;
    }

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

    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;
    }
}
```

Response Payloads

1. JwtAuthenticationResponse

```
public class JwtAuthenticationResponse {  
    private String accessToken;  
    private String tokenType = "Bearer";  
  
    public JwtAuthenticationResponse(String accessToken) {  
        this.accessToken = accessToken;  
    }  
  
    public String getAccessToken() {  
        return accessToken;  
    }  
  
    public void setAccessToken(String accessToken) {  
        this.accessToken = accessToken;  
    }  
  
    public String getTokenType() {  
        return tokenType;  
    }  
  
    public void setTokenType(String tokenType) {  
        this.tokenType = tokenType;  
    }  
}
```

2. ApiResponse

```
package com.example.polls.payload;  
  
public class ApiResponse {  
    private Boolean success;  
    private String message;  
  
    public ApiResponse(Boolean success, String message) {  
        this.success = success;  
        this.message = message;  
    }  
  
    public Boolean getSuccess() {  
        return success;  
    }  
  
    public void setSuccess(Boolean success) {  
        this.success = success;  
    }  
  
    public String getMessage() {  
        return message;  
    }  
  
    public void setMessage(String message) {  
        this.message = message;  
    }  
}
```

The APIs will throw exceptions if the request is not valid or some unexpected situation occurs.

We would also want to respond with different HTTP status codes for different types of exceptions.

Let's define these exceptions along with the corresponding `@ResponseStatus` (All the exception classes will go inside a package named `com.example.polls.exception`) -

1. AppException

```
package com.example.polls.exception;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ResponseStatus;

@ResponseStatus(HttpStatus.INTERNAL_SERVER_ERROR)
public class AppException extends RuntimeException {
    public AppException(String message) {
        super(message);
    }

    public AppException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

2. BadRequestException

```
package com.example.polls.exception;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ResponseStatus;

@ResponseStatus(HttpStatus.BAD_REQUEST)
public class BadRequestException extends RuntimeException {

    public BadRequestException(String message) {
        super(message);
    }

    public BadRequestException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

3. ResourceNotFoundException

```
package com.example.polls.exception;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ResponseStatus;

@ResponseStatus(HttpStatus.NOT_FOUND)
public class ResourceNotFoundException extends RuntimeException {
    private String resourceName;
    private String fieldName;
```



```

public ResourceNotFoundException( String resourceName, String fieldName, Object fieldValue) {
    super(String.format("%s not found with %s : '%s'", resourceName, fieldName, fieldValue));
    this.resourceName = resourceName;
    this.fieldName = fieldName;
    this.fieldValue = fieldValue;
}

public String getResourceName() {
    return resourceName;
}

public String getFieldName() {
    return fieldName;
}

public Object getFieldValue() {
    return fieldValue;
}
}

```

Authentication Controller

Finally, Here is the complete code for `AuthController` that contains APIs for login and signup (All the controllers in our project will go inside a package named `com.example.polls.controller`) -

```

package com.example.polls.controller;

import com.example.polls.exception.AppException;
import com.example.polls.model.Role;
import com.example.polls.model.RoleName;
import com.example.polls.model.User;
import com.example.polls.payload.ApiResponse;
import com.example.polls.payload.JwtAuthenticationResponse;
import com.example.polls.payload.LoginRequest;
import com.example.polls.payload.SignUpRequest;
import com.example.polls.repository.RoleRepository;
import com.example.polls.repository.UserRepository;
import com.example.polls.security.JwtTokenProvider;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.crypto.password.PasswordEncoder;
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 org.springframework.web.servlet.support.ServletUriComponentsBuilder;

import javax.validation.Valid;
import java.net.URI;
import java.util.Collections;

```

```

@Autowired
AuthenticationManager authenticationManager;

@Autowired
UserRepository userRepository;

@Autowired
RoleRepository roleRepository;

@Autowired
PasswordEncoder passwordEncoder;

@Autowired
JwtTokenProvider tokenProvider;

@PostMapping("/signin")
public ResponseEntity<?> authenticateUser(@Valid @RequestBody LoginRequest loginRequest) {

    Authentication authentication = authenticationManager.authenticate(
        new UsernamePasswordAuthenticationToken(
            loginRequest.getUsernameOrEmail(),
            loginRequest.getPassword()
        )
    );

    SecurityContextHolder.getContext().setAuthentication(authentication);

    String jwt = tokenProvider.generateToken(authentication);
    return ResponseEntity.ok(new JwtAuthenticationResponse(jwt));
}

@PostMapping("/signup")
public ResponseEntity<?> registerUser(@Valid @RequestBody SignUpRequest signUpRequest) {
    if(userRepository.existsByUsername(signUpRequest.getUsername())) {
        return new ResponseEntity(new ApiResponse(false, "Username is already taken!"),
            HttpStatus.BAD_REQUEST);
    }

    if(userRepository.existsByEmail(signUpRequest.getEmail())) {
        return new ResponseEntity(new ApiResponse(false, "Email Address already in use!"),
            HttpStatus.BAD_REQUEST);
    }

    // Creating user's account
    User user = new User(signUpRequest.getName(), signUpRequest.getUsername(),
        signUpRequest.getEmail(), signUpRequest.getPassword());

    user.setPassword(passwordEncoder.encode(user.getPassword()));

    Role userRole = roleRepository.findByName(RoleName.ROLE_USER)
        .orElseThrow(() -> new AppException("User Role not set."));

    user.setRoles(Collections.singleton(userRole));

    User result = userRepository.save(user);

```

```
        return ResponseEntity.created(location).body(new ApiResponse(true, "User registered successfully"));
    }
}
```

Enabling CORS

We'll be accessing the APIs from the react client that will run on its own development server. To allow cross origin requests from the react client, create the following `WebMvcConfig` class inside `com.example.polls.config` package -

```
package com.example.polls.config;

import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.CorsRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

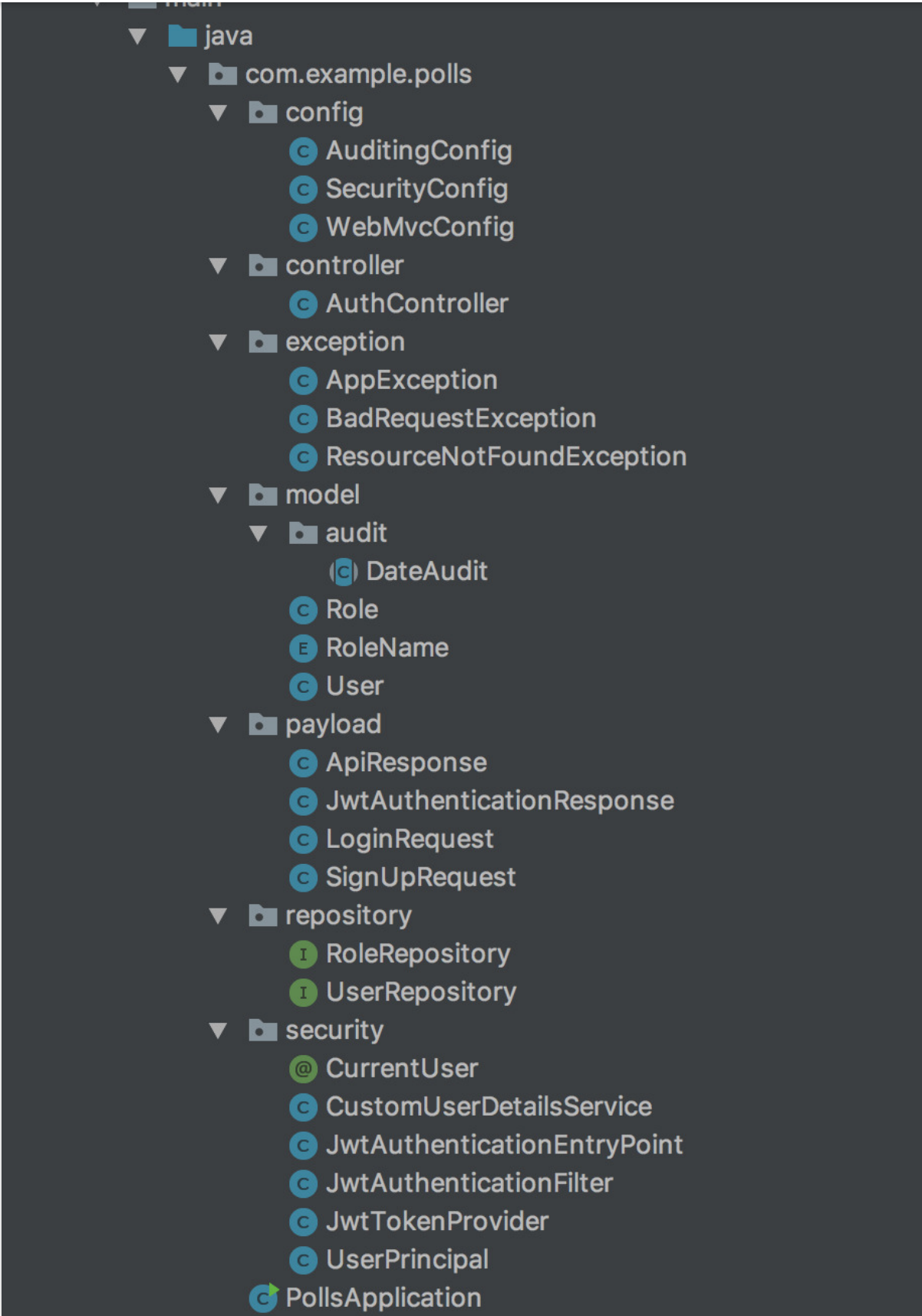
@Configuration
public class WebMvcConfig implements WebMvcConfigurer {

    private final long MAX_AGE_SECS = 3600;

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/**")
            .allowedOrigins("*")
            .allowedMethods("HEAD", "OPTIONS", "GET", "POST", "PUT", "PATCH", "DELETE")
            .maxAge(MAX_AGE_SECS);
    }
}
```

Exploring the current project setup & Running the Application

If you have followed along and defined all the classes presented in this article, then your project's directory structure should look like this-



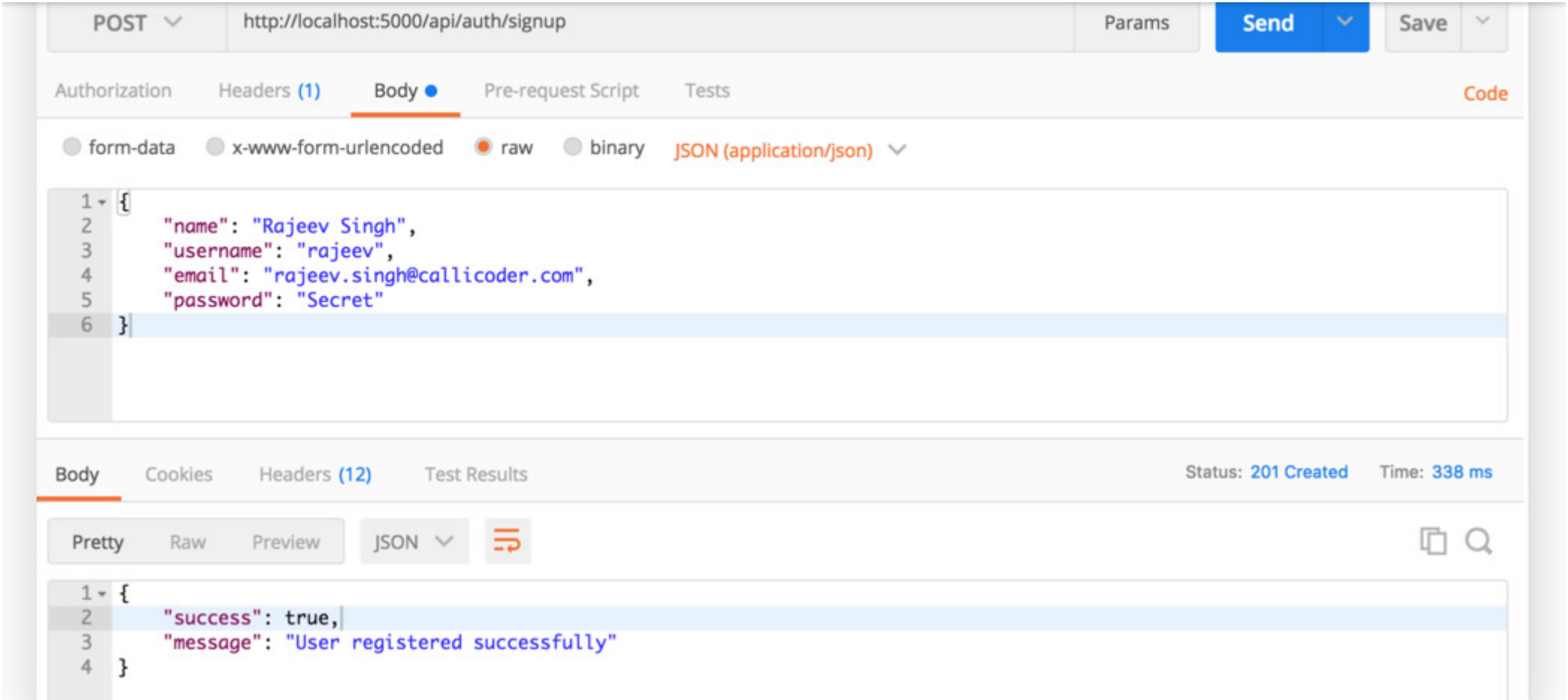
You can run the application by typing the following command from the root directory of your project -

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

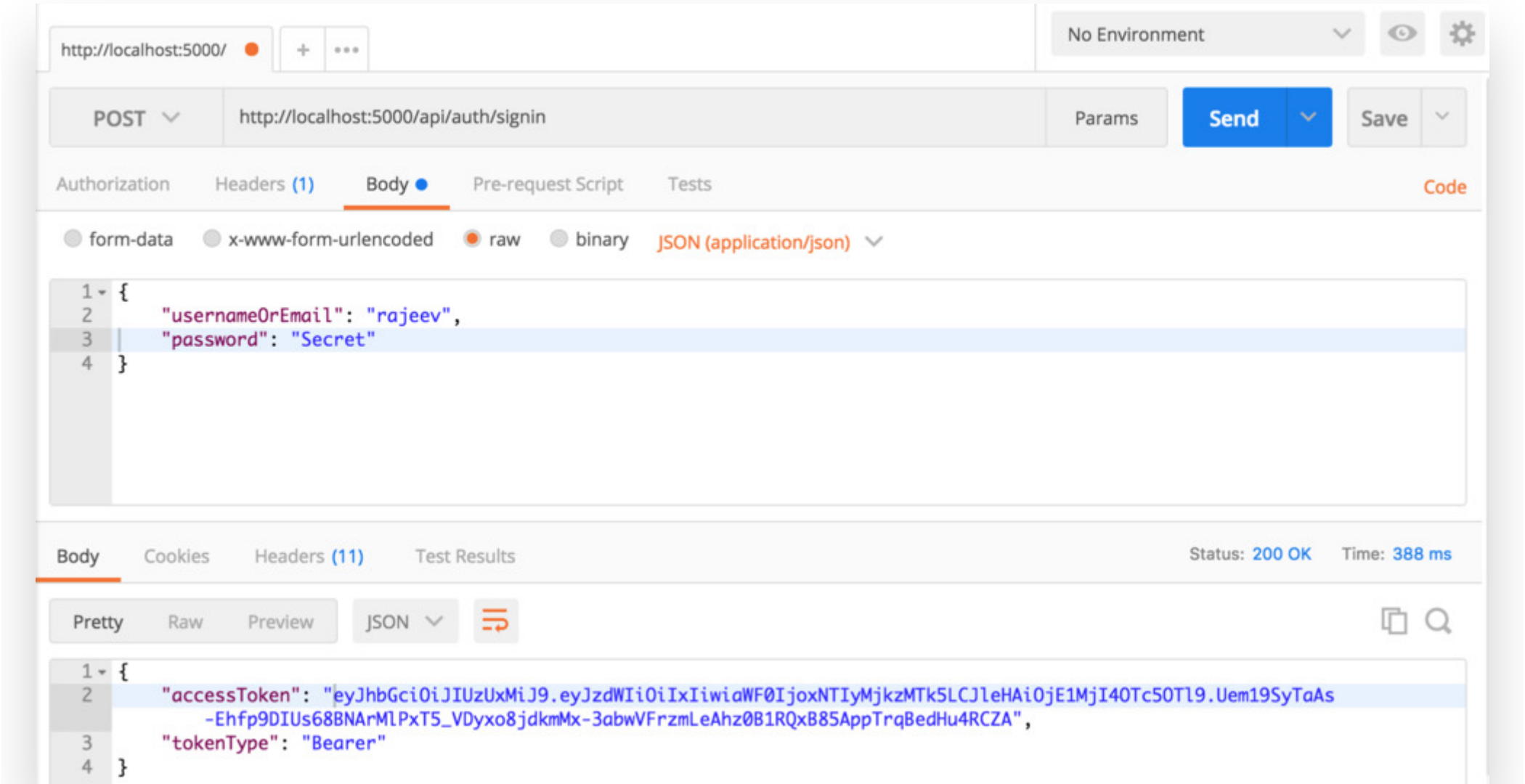
Testing the Login and Signup APIs

Let's now test the login and signup APIs from Postman.

SignUp



Login



Calling Protected APIs

Once you’ve obtained the access token using the login API, you can call any protected API by passing the accessToken in the `Authorization` header of the request like so -

```
Authorization: Bearer <accessToken>
```

The `JwtAuthenticationFilter` will read the accessToken from the header, verify it, and allow/deny access to the API.

Wooh! We covered a lot in this article, and built a solid authentication and authorization logic with Spring Security and JWT. Thank you for reading and congratulations for making it till the end.

In the next article, We'll write the APIs to create Polls, vote for a choice in a Poll, get users profile etc.

Read Next: [Full Stack Polling App with Spring Boot, Spring Security, JWT, MySQL and React - Part 3](#)

Liked the Article? Share it on Social media!


 Twitter

 Facebook

 LinkedIn

 Reddit

Featured Comment



Rajeev Singh Mod ➔ Abdur Rehman • a year ago • edited

I've created a Github repository that demonstrates how to perform **Social Login with Google, Facebook and Github** using the OAuth2 capabilities provided in Spring Security 5. It uses Spring Boot 2.0 in the backend and React in the frontend. Check out the repository here -




[spring-boot-react-oauth2-social-login-demo](#)


The project is complete. A blog post will follow soon.

Cheers,
Rajeev.

3 ^ | v 1 • Share ›

254 Comments CalliCoder


 Recommend 15  Tweet  Share

 Fredi Tansari ▾


Sort by Newest ▾




Join the discussion...

- 

Victor Hugo Ribeiro Tiburtino • a month ago




HOW to fix this error?

^ | v • Reply • Share ›
- 

Simran Sharma • a month ago

Sir i am not able to execute the code in postman also . I am geting the error

```
{
  "timestamp": "2019-11-03T08:21:44.502+0000",
  "status": 401,
  "error": "Unauthorized",
  "message": "Full authentication is required to access this resource",
  "path": "/api/auth/signup"
}
```

^ | v • Reply • Share ›
- 

Olebogeng Mongale • a month ago

Hello, I would like to know how to assign a user a role, i have been trying to register a user but getting "User Role not set" on postman, what

CalliCoder

Software Development Tutorials written from the heart!

Copyright © 2017-2019

RESOURCES

[About & Contact](#)

[Advertise](#)

[Recommended Books](#)

[Recommended Courses](#)

[Algorithms](#)

[Privacy Policy](#)

[Sitemap](#)

TOOLS

[URL Encoder](#)

[URL Decoder](#)

[Base64 Encoder](#)

[Base64 Decoder](#)

[QRCodeBit](#)

[ASCII Table](#)

[JSON Formatter](#)

CONNECT

[Twitter](#)

[Github](#)

[Facebook](#)

[Linkedin](#)

[Reddit](#)