Spring Boot Token based Authentication with Spring Security & JWT

Spring Boot Token based Authentication with Spring Security & JWT

We’re going to build a Spring Boot Application that supports Token based Authentication with JWT.

You’ll know:

- Appropriate Flow for User Signup & User Login with JWT Authentication
- Spring Boot Application Architecture with Spring Security
- How to configure Spring Security to work with JWT
- How to define Data Models and association for Authentication and Authorization
- Way to use Spring Data JPA to interact with PostgreSQL/MySQL Database

Overview of Spring Boot JWT Authentication example

We will build a Spring Boot application in that:

- User can signup new account, or login with username & password.
- By User's role (admin, moderator, user), we authorize the User to access resources

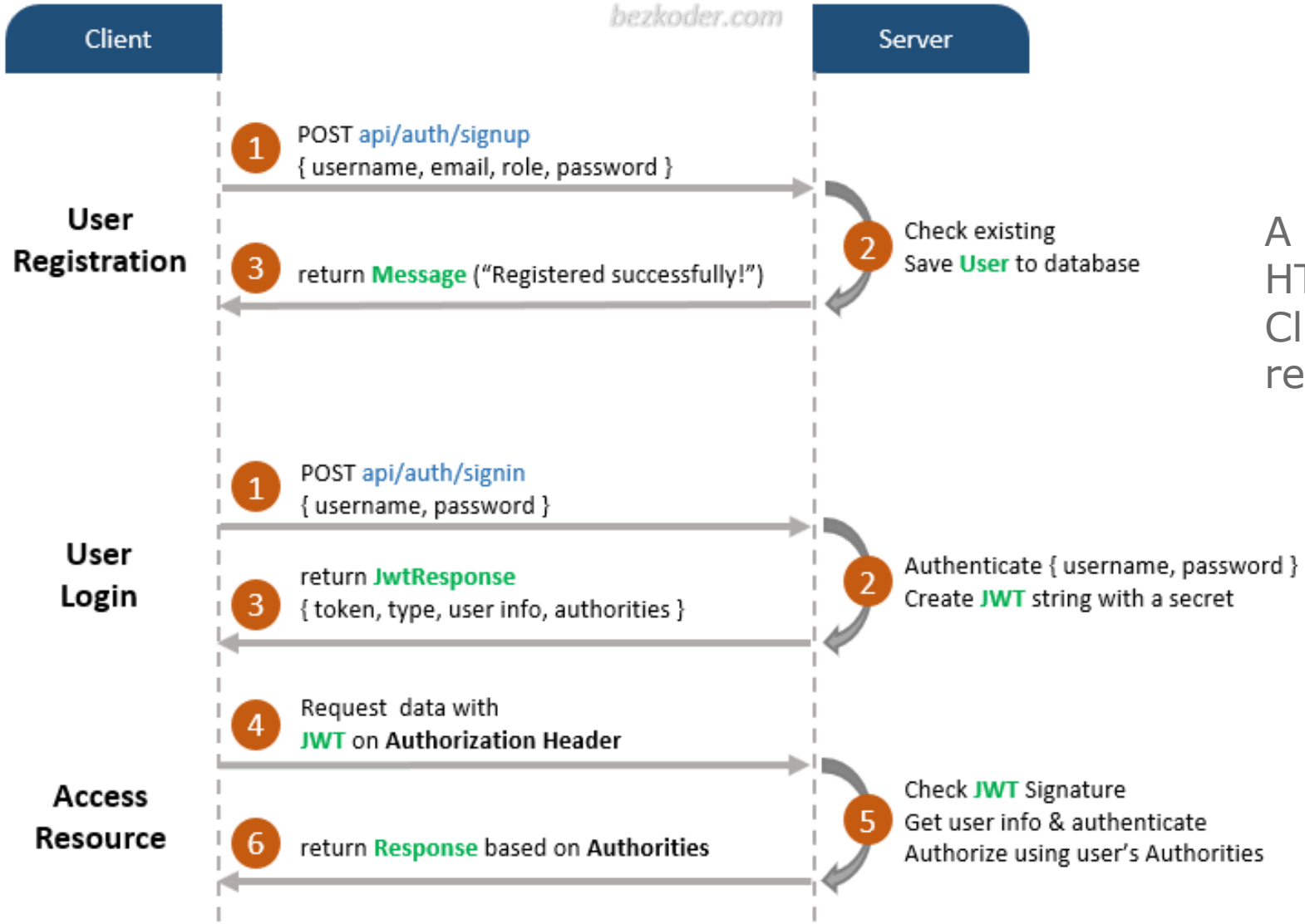
These are APIs that we need to provide:

Methods	Urls	Actions
POST	/api/auth/signup	signup new account
POST	/api/auth/signin	login an account
GET	/api/test/all	retrieve public content
GET	/api/test/user	access User's content
GET	/api/test/mod	access Moderator's content
GET	/api/test/admin	access Admin's content

The database we will use could be PostgreSQL or MySQL depending on the way we configure project dependency & data source.

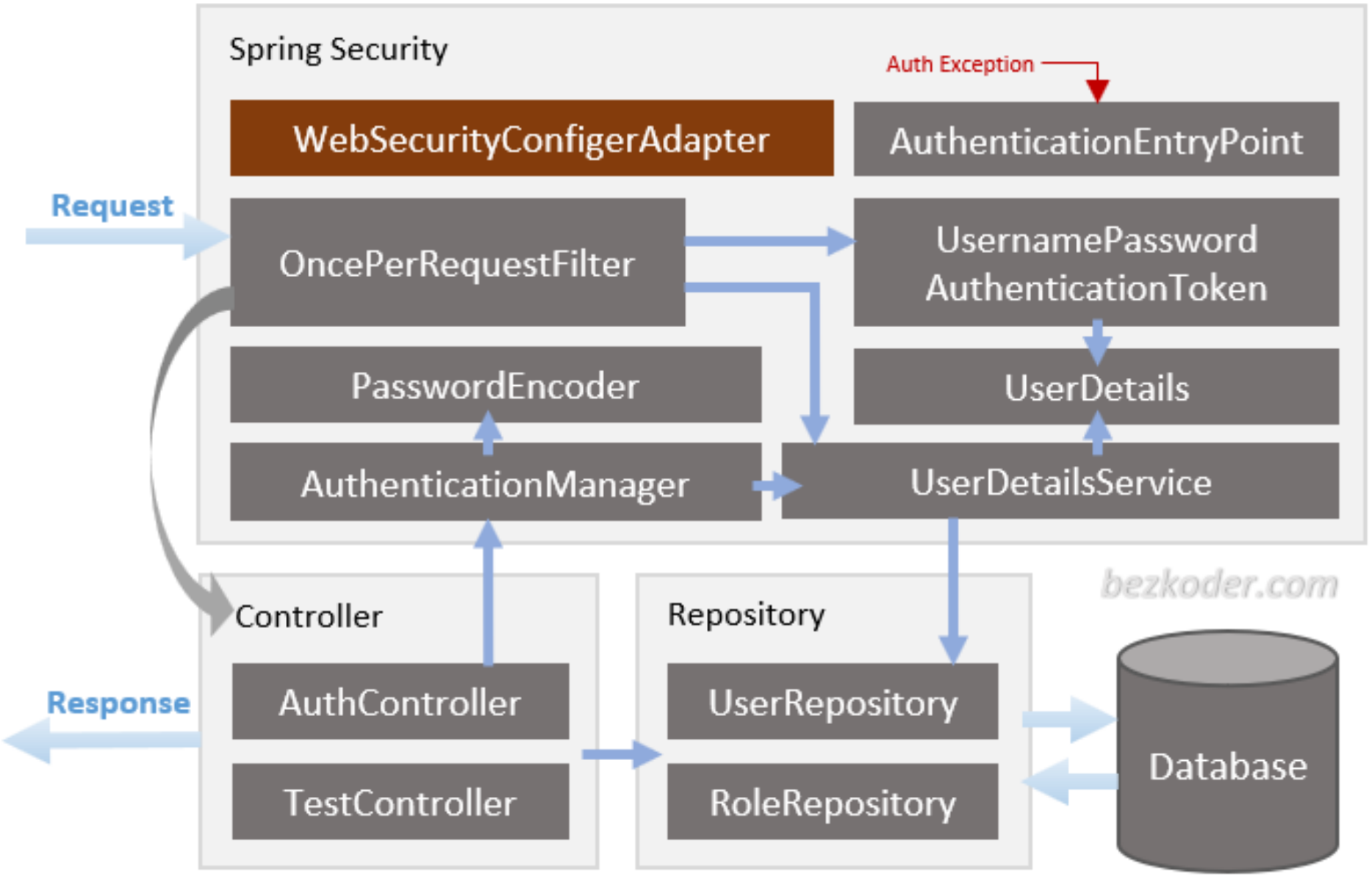
Spring Boot Signup & Login with JWT Authentication Flow

The diagram shows flow of how we implement User Registration, User Login and Authorization process.

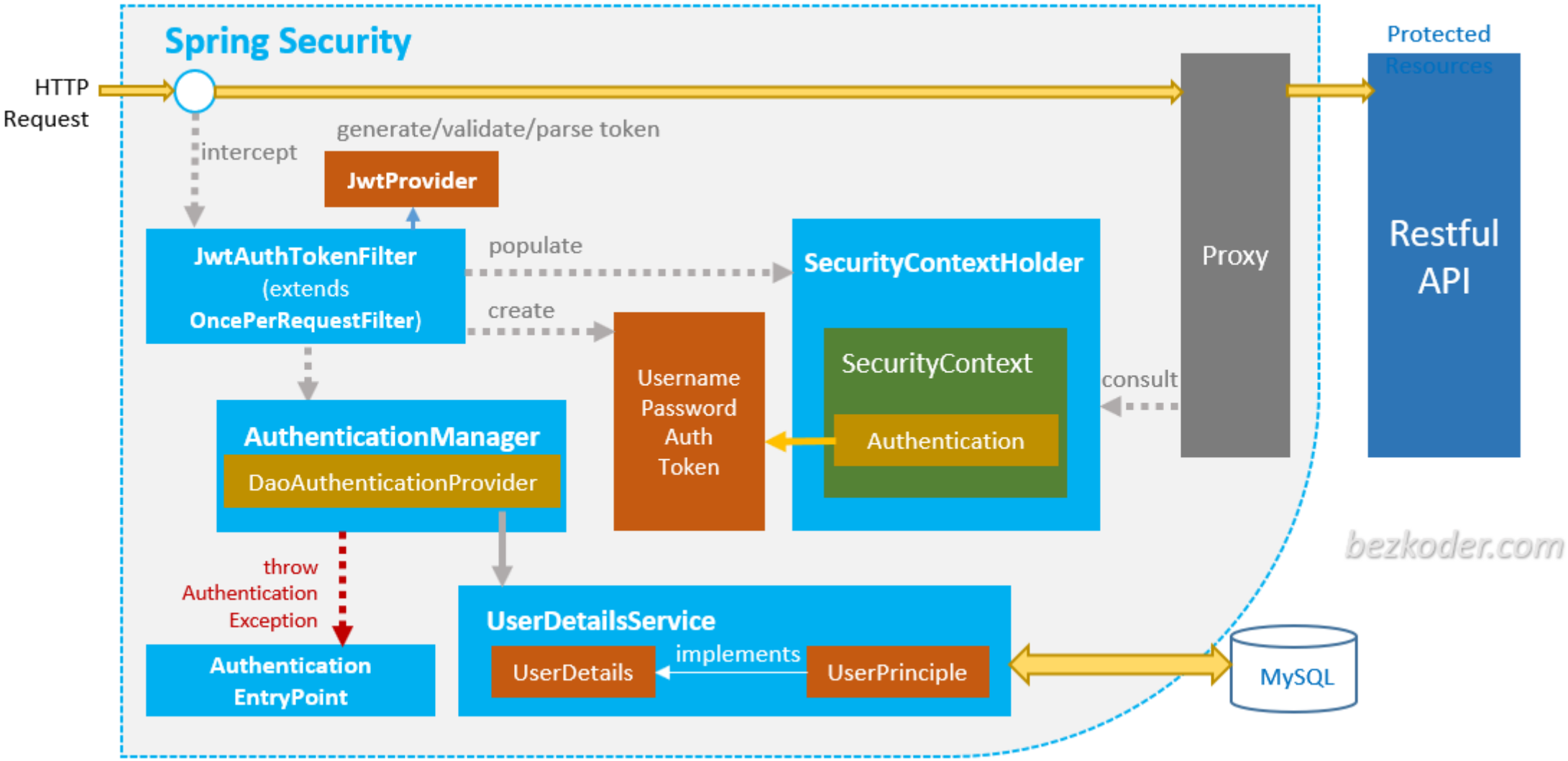


A legal JWT must be added to HTTP Authorization Header if Client accesses protected resources.

Spring Boot Server Architecture with Spring Security



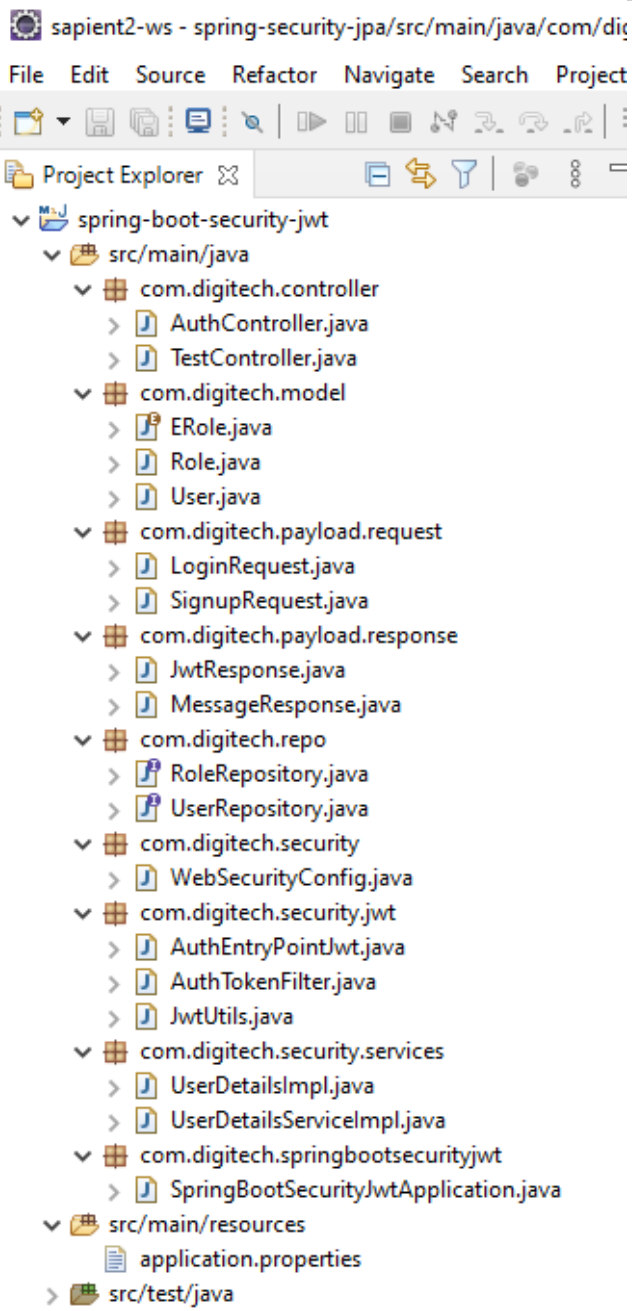
Spring Boot Server Architecture with Spring Security



Spring Boot Server Architecture with Spring Security

Spring Security
– WebSecurityConfigurerAdapter is the crux of our security implementation. It provides HttpSecurity configurations to configure cors, csrf, session management, rules for protected resources. We can also extend and customize the default configuration that contains the elements below.
– UserDetailsService interface has a method to load User by *username* and returns a UserDetails object that Spring Security can use for authentication and validation.
– **UserDetails** contains necessary information (such as: username, password, authorities) to build an Authentication object.
– UsernamePasswordAuthenticationToken gets {username, password} from login Request, AuthenticationManager will use it to authenticate a login account.
– AuthenticationManager has a DaoAuthenticationProvider (with help of UserDetailsService & PasswordEncoder) to validate UsernamePasswordAuthenticationToken object. If successful, AuthenticationManager returns a fully populated Authentication object (including granted authorities).
– OncePerRequestFilter makes a single execution for each request to our API. It provides a *doFilterInternal()* method that we will implement parsing & validating JWT, loading User details (using UserDetailsService), checking Authorization (using UsernamePasswordAuthenticationToken).
– AuthenticationEntryPoint will catch authentication error.
Repository contains UserRepository & RoleRepository to work with Database, will be imported into **Controller**.
Controller receives and handles request after it was filtered by OncePerRequestFilter.
– AuthController handles signup/login requests
– TestController has accessing protected resource methods with role based validations.

Spring Boot Project Structure



security: we configure Spring Security & implement Security Objects here.
- WebSecurityConfig extends WebSecurityConfigurerAdapter
- UserDetailsServiceImpl implements UserDetailsService
- UserDetailsImpl implements UserDetails
- AuthEntryPointJwt implements AuthenticationEntryPoint
- AuthTokenFilter extends OncePerRequestFilter
- JwtUtils provides methods for generating, parsing, validating JWT

controllers handle signup/login requests & authorized requests.
- AuthController: @PostMapping('/signin'), @PostMapping('/signup')
- TestController: @GetMapping('/api/test/all'), @GetMapping('/api/test/[role]')

repository has interfaces that extend Spring Data JPA JpaRepository to interact with Database.
- UserRepository extends JpaRepository<User, Long>
- RoleRepository extends JpaRepository<Role, Long>

models defines two main models for Authentication (User) & Authorization (Role). They have many-to-many relationship.
- User: id, username, email, password, roles
- Role: id, name

payload defines classes for Request and Response objects

We also have **application.properties** for configuring Spring Data source, Spring Data JPA and App properties (such as JWT Secret string or Token expiration time).

Spring Boot Project Structure: pom.xml

```
<dependencies>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>

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

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

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

<dependency>
<groupId>org.postgresql</groupId>
<artifactId>postgresql</artifactId>
<scope>runtime</scope>
</dependency>
<dependency>
<groupId>mysql</groupId>
<artifactId>mysql-connector-java</artifactId>
<scope>runtime</scope>
</dependency>
<!-- https://mvnrepository.com/artifact/javax.validation/validation-api -->
<dependency>
<groupId>javax.validation</groupId>
<artifactId>validation-api</artifactId>
</dependency>

<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-test</artifactId>
<scope>test</scope>
<exclusions>
<exclusion>
<groupId>org.junit.vintage</groupId>
<artifactId>junit-vintage-engine</artifactId>
</exclusion>
</exclusions>
</dependency>
</dependencies>
```

Configure Spring Datasource, JPA, App properties

For PostgreSQL

```
spring.datasource.url= jdbc:postgresql://localhost:5432/testdb
spring.datasource.username= postgres
spring.datasource.password= 123
spring.jpa.properties.hibernate.jdbc.lob.non_contextual_creation= true
spring.jpa.properties.hibernate.dialect= org.hibernate.dialect.PostgreSQLDialect
# Hibernate ddl auto (create, create-drop, validate, update)
spring.jpa.hibernate.ddl-auto= update
# App Properties
digitech.app.jwtSecret= digitechSecretKey
digitech.app.jwtExpirationMs= 86400000
```

For MySQL

```
spring.datasource.url= jdbc:mysql://localhost:3306/testdb?useSSL=false
spring.datasource.username= root
spring.datasource.password= 123456
spring.jpa.properties.hibernate.dialect= org.hibernate.dialect.MySQL5InnoDBDialect
spring.jpa.hibernate.ddl-auto= update
# App Properties
digitech.app.jwtSecret= digitechSecretKey
digitech.app.jwtExpirationMs= 86400000
```

Create the models

We're going to have 3 tables in database: **users**, **roles** and **user_roles** for many-to-many relationship.

```
public enum ERole { ROLE_USER, ROLE_MODERATOR, ROLE_ADMIN }
```

```
@Entity
@Table(name = "roles")
public class Role {
	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	private Integer id;

	@Enumerated(EnumType.STRING)
	@Column(length = 20)
	private ERole name;

	public Role() { }

	public Role(ERole name) {
		this.name = name;		}

	public Integer getId() {
		return id;	}

	public void setId(Integer id) {
		this.id = id;	}

	public ERole getName() {
		return name;	}

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

```
@Entity
@Table(	name = "users",
		uniqueConstraints = {
			@UniqueConstraint(columnNames = "username"),
			@UniqueConstraint(columnNames = "email")
		})
public class User {
	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	private Long id;
	@NotBlank
	@Size(max = 20)
	private String username;
	@NotBlank
	@Size(max = 50)
	@Email
	private String email;
	@NotBlank
	@Size(max = 120)
	private String password;
	@ManyToMany(fetch = FetchType.LAZY)
	@JoinTable(	name = "user_roles",
			joinColumns = @JoinColumn(name = "user_id"),
			inverseJoinColumns = @JoinColumn(name = "role_id"))
	private Set<Role> roles = new HashSet<>();

	public User() {	}

	public User(String username, String email, String password) {
		this.username = username;
		this.email = email;
		this.password = password;
	}
	// getter and setter methods
}
```

Implement Repositories

```
@Repository public interface UserRepository extends JpaRepository<User, Long> {
        Optional<User> findByUsername(String username);
        Boolean existsByUsername(String username);
        Boolean existsByEmail(String email);
}
```

```
@Repository public interface RoleRepository extends JpaRepository<Role, Long> {
        Optional<Role> findByName(ERole name);
}
```

Configure Spring Security

```
@Configuration
// allows Spring to find and automatically apply the class to the global Web Security
@EnableWebSecurity
@EnableGlobalMethodSecurity( // securedEnabled = true, // jsr250Enabled = true, prePostEnabled = true)
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
@Autowired
UserDetailsServiceImpl userDetailsService;
@Autowired
private AuthEntryPointJwt unauthorizedHandler;
@Bean
public AuthTokenFilter authenticationJwtTokenFilter() {
return new AuthTokenFilter();
}

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

```
@Bean
@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("/api/auth/**").permitAll()
.antMatchers("/api/test/**").permitAll() .anyRequest().authenticated();
http.addFilterBefore(authenticationJwtTokenFilter(), UsernamePasswordAuthenticationFilter.class);
} }
```

Configure Spring Security

– @EnableWebSecurity allows Spring to find and automatically apply the class to the global Web Security.

– @EnableGlobalMethodSecurity provides AOP security on methods. It enables @PreAuthorize, @PostAuthorize, it also supports JSR-250.

– We override the configure(HttpSecurity http) method from WebSecurityConfigurerAdapter interface. It tells Spring Security how we configure CORS and CSRF, when we want to require all users to be authenticated or not, which filter (AuthTokenFilter) and when we want it to work (filter before UsernamePasswordAuthenticationFilter), which Exception Handler is chosen (AuthEntryPointJwt).

– Spring Security will load User details to perform authentication & authorization. So it has UserDetailsService interface that we need to implement.

– The implementation of UserDetailsService will be used for configuring DaoAuthenticationProvider by AuthenticationManagerBuilder.userDetailsService() method.

– We also need a PasswordEncoder for the DaoAuthenticationProvider. If we don't specify, it will use plain text.

Implement UserDetails & UserDetailsService

If the authentication process is successful, we can get User's information such as username, password, authorities from an Authentication object.

```
Authentication authentication = authenticationManager.authenticate(
        new UsernamePasswordAuthenticationToken(username, password) );

UserDetails userDetails = (UserDetails) authentication.getPrincipal();
// userDetails.getUsername()
// userDetails.getPassword()
// userDetails.getAuthorities()
```

If we want to get more data (id, email...), we can create an implementation of this UserDetails interface.

UserDetailsImpl

```
public class UserDetailsImpl implements UserDetails {
private static final long serialVersionUID = 1L;

private Long id;
private String username;
private String email;
@JsonIgnore
private String password;

private Collection<? extends GrantedAuthority> authorities;

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

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

        return new UserDetailsImpl(
                                user.getId(),
                                user.getUsername(),
                                user.getEmail(),
                                user.getPassword(),
                                authorities);
}
```

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

        public Long getId() {           return id;      }
        public String getEmail() {      return email;   }

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

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

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

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

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

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

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

UserDetailsImpl

In the code above, you can notice that we convert Set<Role> into List<GrantedAuthority>.

It is important to work with Spring Security and Authentication object later.
We need UserDetailsService for getting UserDetails object.

UserDetailsService interface that has only one method:

```
public interface UserDetailsService {
        UserDetails loadUserByUsername(String username) throws UsernameNotFoundException;
}
```

```
@Service public class UserDetailsServiceImpl implements UserDetailsService {
@Autowired
UserRepository userRepository;
@Override
@Transactional
public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
User user = userRepository.findByUsername(username) .orElseThrow(() -> new UsernameNotFoundException("User Not Found
with username: " + username));
return UserDetailsImpl.build(user);
}
}
```

Filter the Requests

Let's define a filter that executes once per request. So we create AuthTokenFilter class that extends OncePerRequestFilter and override doFilterInternal() method

```
public class AuthTokenFilter extends OncePerRequestFilter {
	@Autowired
	private JwtUtils jwtUtils;
	@Autowired
	private UserDetailsServiceImpl userDetailsService;
	private static final Logger logger = LoggerFactory.getLogger(AuthTokenFilter.class);
	@Override
	protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
			throws ServletException, IOException {
		try {
			String jwt = parseJwt(request);
			if (jwt != null && jwtUtils.validateJwtToken(jwt)) {
				String username = jwtUtils.getUserNameFromJwtToken(jwt);

				UserDetails userDetails = userDetailsService.loadUserByUsername(username);
				UsernamePasswordAuthenticationToken authentication = new UsernamePasswordAuthenticationToken(
						userDetails, null, userDetails.getAuthorities());
				authentication.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));

				SecurityContextHolder.getContext().setAuthentication(authentication);
			}
		} catch (Exception e) {
			logger.error("Cannot set user authentication: {}", e);
		}

		filterChain.doFilter(request, response);
	}

	private String parseJwt(HttpServletRequest request) {
		String headerAuth = request.getHeader("Authorization");

		if (StringUtils.hasText(headerAuth) && headerAuth.startsWith("Bearer ")) {
			return headerAuth.substring(7, headerAuth.length());
		}
		return null;
	}
}
```

What we do inside doFilterInternal():
- get JWT from the Authorization header (by removing Bearer prefix)
- if the request has JWT, validate it, parse username from it
- from username, get UserDetails to create an Authentication object
- set the current UserDetails in SecurityContext using setAuthentication(authentication) method.

After this, everytime you want to get UserDetails, just use SecurityContext like this:

```
UserDetails userDetails = (UserDetails)
SecurityContextHolder.getContext()
.getAuthentication().getPrincipal();
// userDetails.getUsername() // userDetails.getPassword()
// userDetails.getAuthorities()
```

Create JWT Utility class

```
@Component
public class JwtUtils {
		private static final Logger logger = LoggerFactory.getLogger(JwtUtils.class);

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

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

		public String generateJwtToken(Authentication authentication) {

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

				return Jwts.builder()

		.setSubject((userPrincipal.getUsername()))
		.setIssuedAt(new Date())
		.setExpiration(new Date((new Date()).getTime() + jwtExpirationMs))

		.signWith(SignatureAlgorithm.HS512, jwtSecret)
								.compact();
		}

		public String getUserNameFromJwtToken(String token) {
				return
Jwts.parser().setSigningKey(jwtSecret).parseClaimsJws(token).getBody().getSubject();
		}
```

This class has 3 functions:

- generate a JWT from username, date, expiration, secret
- get username from JWT
- validate a JWT

```
public boolean validateJwtToken(String authToken) {
				try {
						Jwts.parser().setSigningKey(jwtSecret).parseClaimsJws(authToken);
						return true;
				} catch (SignatureException e) {
						logger.error("Invalid JWT signature: {}", e.getMessage());
				} catch (MalformedJwtException e) {
						logger.error("Invalid JWT token: {}", e.getMessage());
				} catch (ExpiredJwtException e) {
						logger.error("JWT token is expired: {}", e.getMessage());
				} catch (UnsupportedJwtException e) {
						logger.error("JWT token is unsupported: {}", e.getMessage());
				} catch (IllegalArgumentException e) {
						logger.error("JWT claims string is empty: {}", e.getMessage());
				}

				return false;
		}
}
```

Remember that we've added digitech.app.jwtSecret and digitech.app.jwtExpirationMs properties in application.properties file.

Handle Authentication Exception

Now we create AuthEntryPointJwt class that implements AuthenticationEntryPoint interface. Then we override the commence() method. This method will be triggered anytime unauthenticated User requests a secured HTTP resource and an AuthenticationException is thrown.

```
@Component
public class AuthEntryPointJwt implements AuthenticationEntryPoint {

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

	@Override
	public void commence(HttpServletRequest request, HttpServletResponse response,
				AuthenticationException authException) throws IOException, ServletException {
		logger.error("Unauthorized error: {}", authException.getMessage());
		response.sendError(HttpServletResponse.SC_UNAUTHORIZED, "Error: Unauthorized");
	}

}
```

HttpServletResponse.SC_UNAUTHORIZED is the 401 Status code. It indicates that the request requires HTTP authentication.
We've already built all things for Spring Security. The next sections of this tutorial will show you how to implement Controllers for our RestAPIs.

Define payloads for Spring RestController

Controller for Authentication

This controller provides APIs for register and login actions.

– /api/auth/signup
- check existing username/email
- create new User (with ROLE_USER if not specifying role)
- save User to database using UserRepository

– /api/auth/signin
- authenticate { username, password }
- update SecurityContext using Authentication object
- generate JWT
- get UserDetails from Authentication object
- response contains JWT and UserDetails data

AuthController

```
@CrossOrigin(origins = "*", maxAge = 3600)
@RestController
@RequestMapping("/api/auth")
public class AuthController {
		@Autowired
		AuthenticationManager authenticationManager;
		@Autowired
		UserRepository userRepository;
		@Autowired
		RoleRepository roleRepository;
		@Autowired
		PasswordEncoder encoder;
		@Autowired
		JwtUtils jwtUtils;

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

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

			SecurityContextHolder.getContext().setAuthentication(authentication);
			String jwt = jwtUtils.generateJwtToken(authentication);

			UserDetailsImpl userDetails = (UserDetailsImpl) authentication.getPrincipal();
			List<String> roles = userDetails.getAuthorities().stream()
					.map(item -> item.getAuthority())
					.collect(Collectors.toList());

			return ResponseEntity.ok(new JwtResponse(jwt,  userDetails.getId(),  userDetails.getUsername(),  userDetails.getEmail(),  roles));
		}
```

contd..

contd..

AuthController

```
@PostMapping("/signup")
public ResponseEntity<?> registerUser(@Valid @RequestBody SignupRequest signUpRequest) {
                                        if (userRepository.existsByUsername(signUpRequest.getUsername())) {
                                                                return ResponseEntity
                                                                                                        .badRequest()
                                                                                                        .body(new MessageResponse("Error: Username is already taken!"));
                                        }

                                        if (userRepository.existsByEmail(signUpRequest.getEmail())) {
                                                                return ResponseEntity
                                                                                                        .badRequest()
                                                                                                        .body(new MessageResponse("Error: Email is already in use!"));
                                        }

                                        // Create new user's account
                                        User user = new User(signUpRequest.getUsername(),
                                                                                                                                                signUpRequest.getEmail(),
                                                                                                                                                encoder.encode(signUpRequest.getPassword()));


                                        Set<String> strRoles = signUpRequest.getRole();
                                        Set<Role> roles = new HashSet<>();

                                        if (strRoles == null) {
                                                                Role userRole = roleRepository.findByName(ERole.ROLE_USER)
                                                                                                        .orElseThrow(() -> new RuntimeException("Error: Role is not found."));
                                                                roles.add(userRole);
                                        } else {
                                                                strRoles.forEach(role -> {
                                                                                        switch (role) {
                                                                                        case "admin":
                                                                                                                Role adminRole = roleRepository.findByName(ERole.ROLE_ADMIN)
                                                                                                                                                                .orElseThrow(() -> new RuntimeException("Error: Role is not found."));
                                                                                                                roles.add(adminRole);

                                                                                                                break;
                                                                                        case "mod":
                                                                                                                Role modRole = roleRepository.findByName(ERole.ROLE_MODERATOR)
                                                                                                                                                                .orElseThrow(() -> new RuntimeException("Error: Role is not found."));
                                                                                                                roles.add(modRole);

                                                                                                                break;
                                                                                        default:
                                                                                                                Role userRole = roleRepository.findByName(ERole.ROLE_USER)
                                                                                                                                                                .orElseThrow(() -> new RuntimeException("Error: Role is not found."));
                                                                                                                roles.add(userRole);
                                                                                        }
                                                                });
                                        }

                                        user.setRoles(roles);
                                        userRepository.save(user);

                                        return ResponseEntity.ok(new MessageResponse("User registered successfully!"));
                }
}
```

Controller for testing Authorization

There are 4 APIs:
- /api/test/all for public access
- /api/test/user for users has ROLE_USER or ROLE_MODERATOR or ROLE_ADMIN
- /api/test/mod for users has ROLE_MODERATOR
- /api/test/admin for users has ROLE_ADMIN

Remember we used @EnableGlobalMethodSecurity(prePostEnabled = true) for WebSecurityConfig class

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter { ... }
```

Now we can secure methods in our Apis with @PreAuthorize annotation easily.

Enable Method Level Security
By annotating the class with **@EnableGlobalMethodSecurity**, we can enable method level security using annotations. We can optionally configure which annotations we'll allow. You can enable one of the following.
securedEnabled – enables the spring **@Secured** annotation.

TestController

Now we can secure methods in our Apis with @PreAuthorize annotation easily.

```
@CrossOrigin(origins = "*", maxAge = 3600)
@RestController
@RequestMapping("/api/test")
public class TestController {
	@GetMapping("/all")
	public String allAccess() {
		return "Public Content.";
	}

	@GetMapping("/user")
	@PreAuthorize("hasRole('USER') or hasRole('MODERATOR') or hasRole('ADMIN')")
	public String userAccess() {
		return "User Content.";
	}

	@GetMapping("/mod")
	@PreAuthorize("hasRole('MODERATOR')")
	public String moderatorAccess() {
		return "Moderator Board.";
	}

	@GetMapping("/admin")
	@PreAuthorize("hasRole('ADMIN')")
	public String adminAccess() {
		return "Admin Board.";
	}
}
```

Enable Method Level Security

By annotating the class with **@EnableGlobalMethodSecurity**, we can enable method level security using annotations. We can optionally configure which annotations we'll allow. You can enable one of the following.

securedEnabled – enables the spring **@Secured** annotation.

jsr250Enabled – enables the JSR-250 standard java security annotations.

prePostEnabled – enables the spring **@PreAuthorize and @PostAuthorize** annotations.

Run & Test

Run Spring Boot application with command: mvn spring-boot:run

Tables that we define in models package will be automatically generated in Database.
If you check PostgreSQL for example, you can see things like this:

```
\d users
                        Table "public.users"
 Column  |         Type          |                    Modifiers
---------+-----------------------+-------------------------------------------------
 id      | bigint                | not null default nextval('users_id_seq'::regclass)
 email   | character varying(50) |
 password | character varying(120) |
 username | character varying(20) |
Indexes:
    "users_pkey" PRIMARY KEY, btree (id)
    "uk6dotkott2kjsp8vw4d0m25fb7" UNIQUE CONSTRAINT, btree (email)
    "ukr43af9ap4edm43mmtq01oddj6" UNIQUE CONSTRAINT, btree (username)
Referenced by:
    TABLE "user_roles" CONSTRAINT "fkhfh9dx7w3ubf1co1vdev94g3f" FOREIGN KEY (user_id) REFERENCES users(id)
```

Contd..

contd..

Run & Test

```
\d roles;
                        Table "public.roles"
 Column |         Type          |                    Modifiers
--------+-----------------------+-------------------------------------------------
 id     | integer               | not null default nextval('roles_id_seq'::regclass)
 name   | character varying(20) |
Indexes:
    "roles_pkey" PRIMARY KEY, btree (id)
Referenced by:
    TABLE "user_roles" CONSTRAINT "fkh8ciramu9cc9q3qcqiv4ue8a6" FOREIGN KEY (role_id) REFERENCES roles(id)


\d user_roles
   Table "public.user_roles"
 Column  |  Type   | Modifiers
---------+---------+-----------
 user_id | bigint  | not null
 role_id | integer | not null
Indexes:
    "user_roles_pkey" PRIMARY KEY, btree (user_id, role_id)
Foreign-key constraints:
    "fkh8ciramu9cc9q3qcqiv4ue8a6" FOREIGN KEY (role_id) REFERENCES roles(id)
    "fkhfh9dx7w3ubf1co1vdev94g3f" FOREIGN KEY (user_id) REFERENCES users(id)
```

contd..

Run & Test

We also need to add some rows into roles table before assigning any role to User.
Run following SQL insert statements:

```
INSERT INTO roles(name) VALUES('ROLE_USER');
INSERT INTO roles(name) VALUES('ROLE_MODERATOR');
INSERT INTO roles(name) VALUES('ROLE_ADMIN');
```

Then check the tables:

```
> SELECT * FROM roles;

 id |      name
----+----------------
  1 | ROLE_USER
  2 | ROLE_MODERATOR
  3 | ROLE_ADMIN
(3 rows)
```

Register some users with /signup API:

admin with ROLE_ADMIN
mod with ROLE_MODERATOR and ROLE_USER
asr with ROLE_USER

contd..

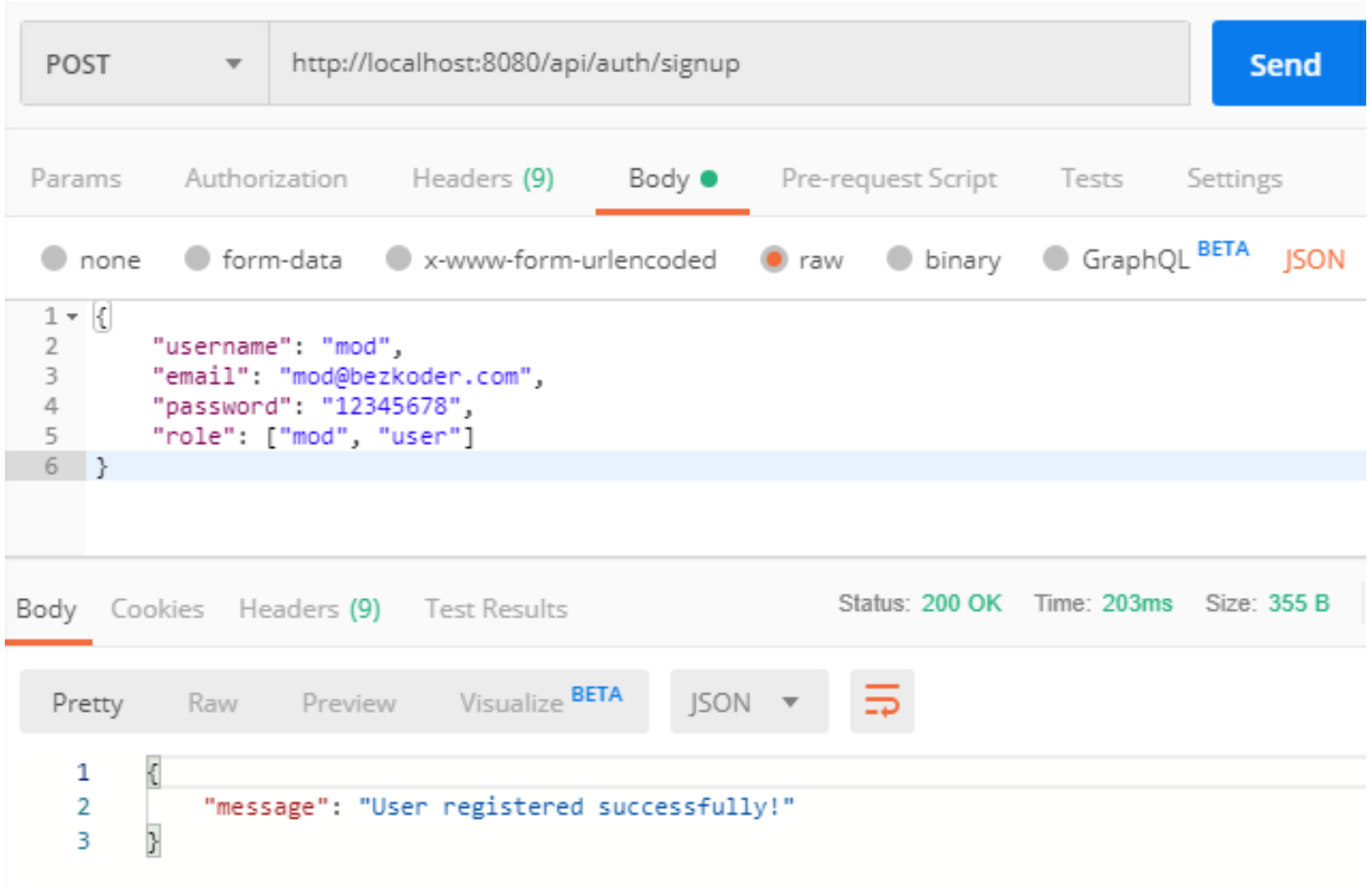
Run & Test

Our tables after signup could look like this.

```
testdb=# select * from users;
 id |      email      |                          password                          | username
----+-----------------+------------------------------------------------------------+----------
  2 | admin@gmail.com | $2a$10$I6.nJ0V.DTZxcvBGVmUOnuMuyOAXat.SY1IENTzUjFfKzEwmRbKZq | admin
  3 | mod@gmail.com   | $2a$10$aOGaaCjtzPwTPPnHRhUNz.JtV.gybrWrZpAhGhzmBzmWFKKkP0ID6 | mod
  4 | asr@gmail.com   | $2a$10$.HALk5pRyuBO5JJwdrsg7e8pH6tx6rk.EeGLgnXIvsNXa70XRcZ6W | asr
```

```
> SELECT * FROM roles;

 id |      name
----+----------------
  1 | ROLE_USER
  2 | ROLE_MODERATOR
  3 | ROLE_ADMIN
(3 rows)
```

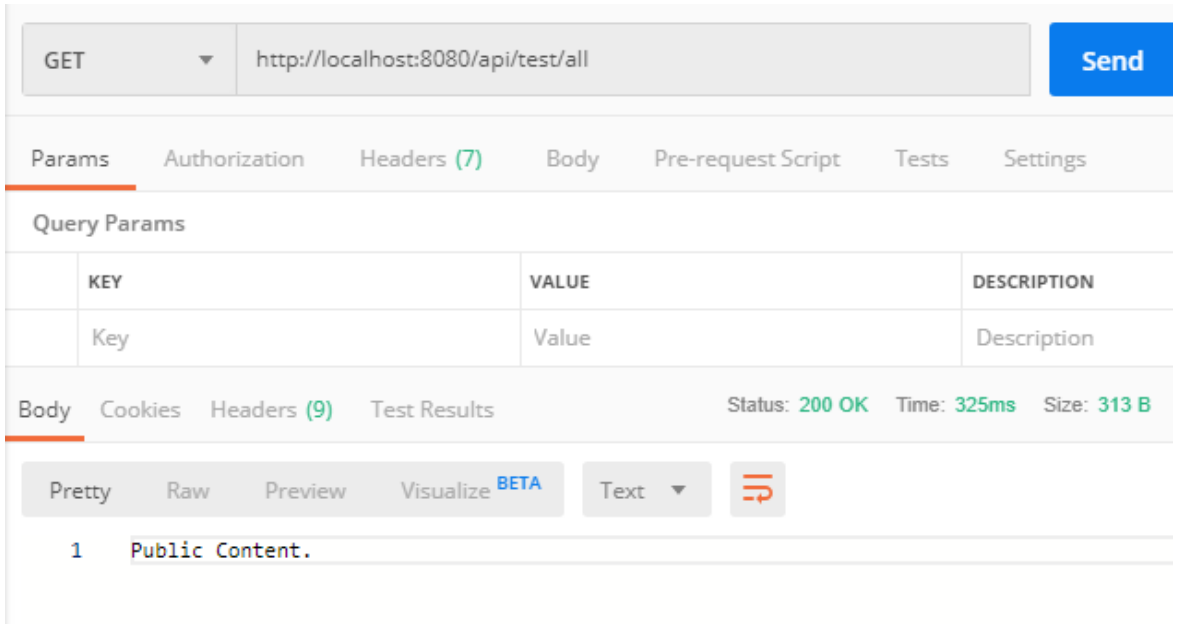
```
>SELECT * FROM user_roles;

 user_id | role_id
---------+---------
       1 |       3
       2 |       1
       2 |       2
       3 |       1
(4 rows)
```



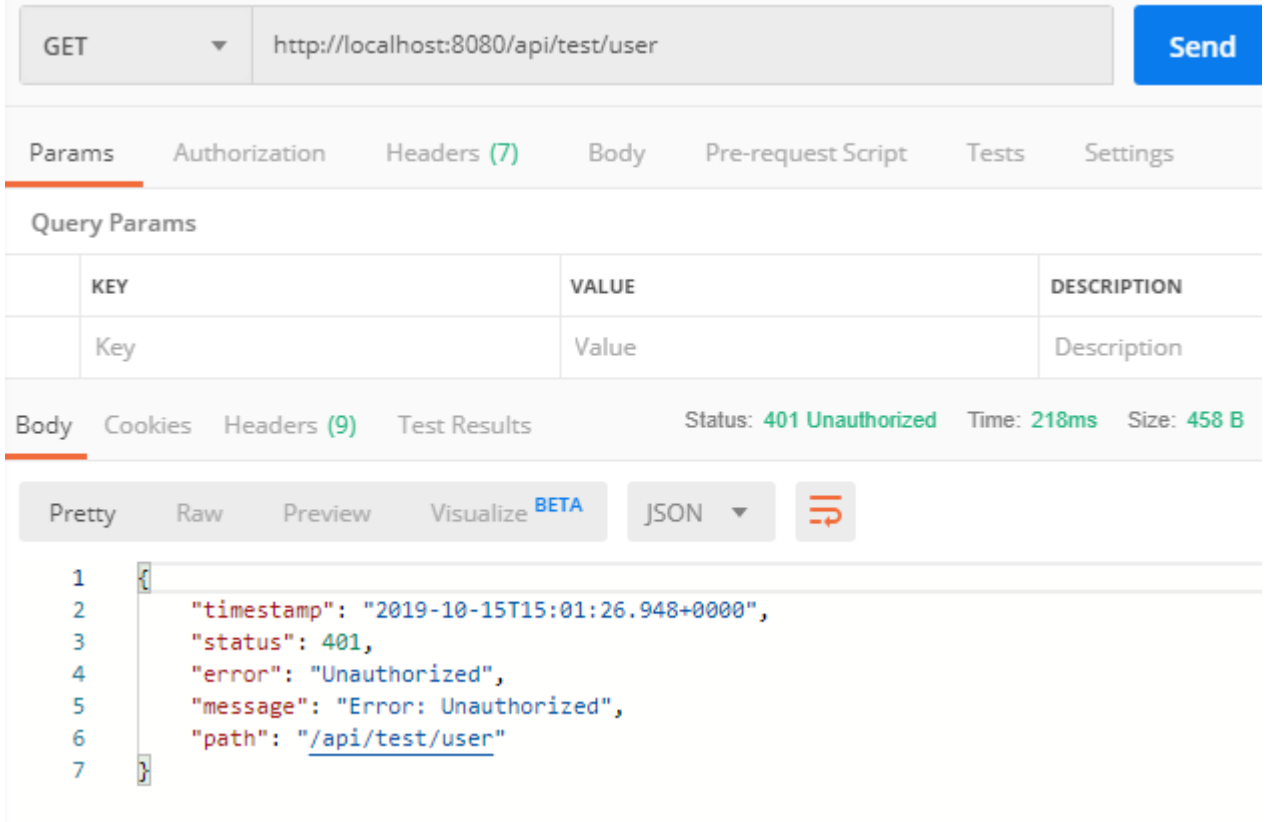
contd..

Run & Test

Access public resource: GET /api/test/all



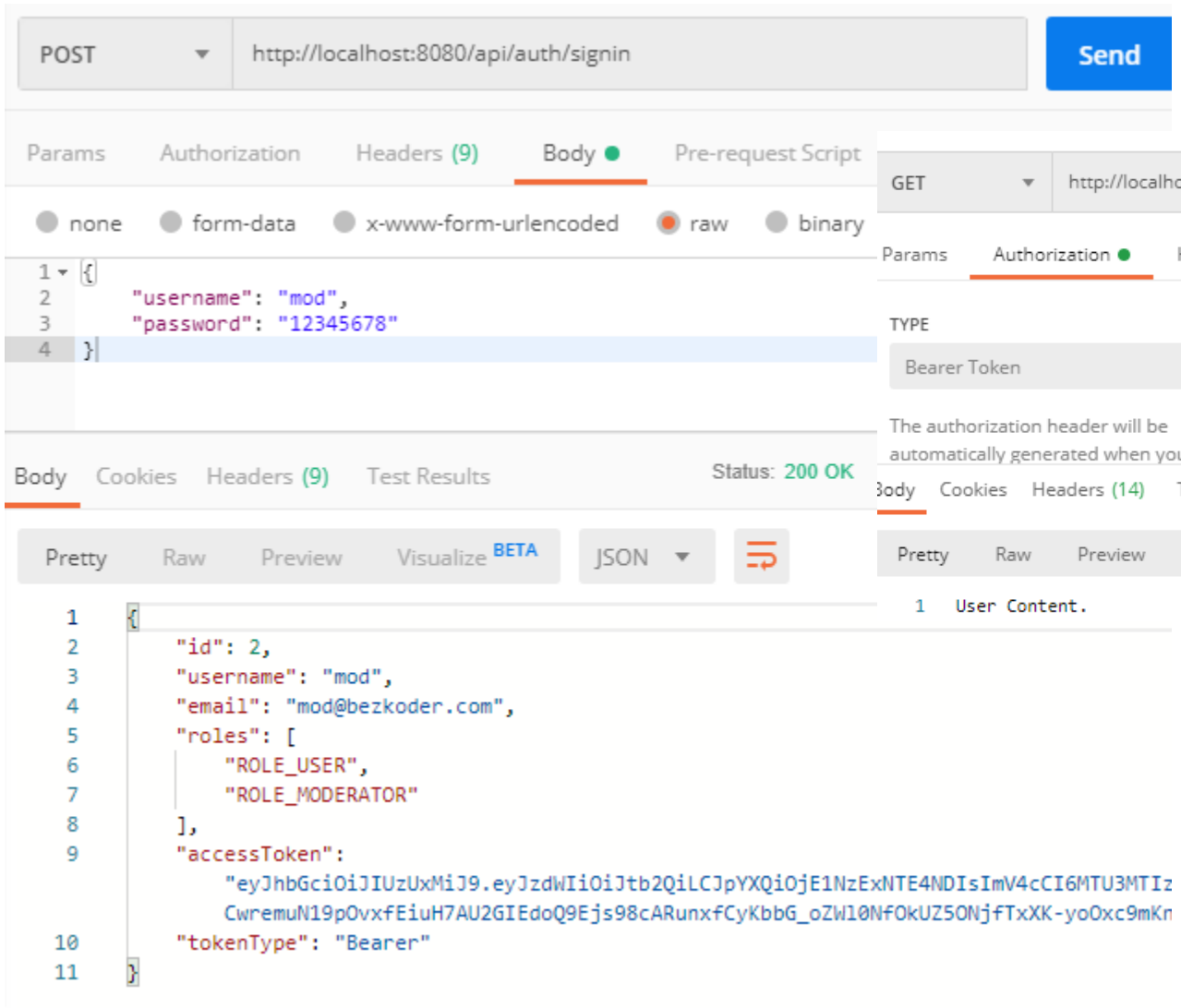
Access protected resource: GET /api/test/user



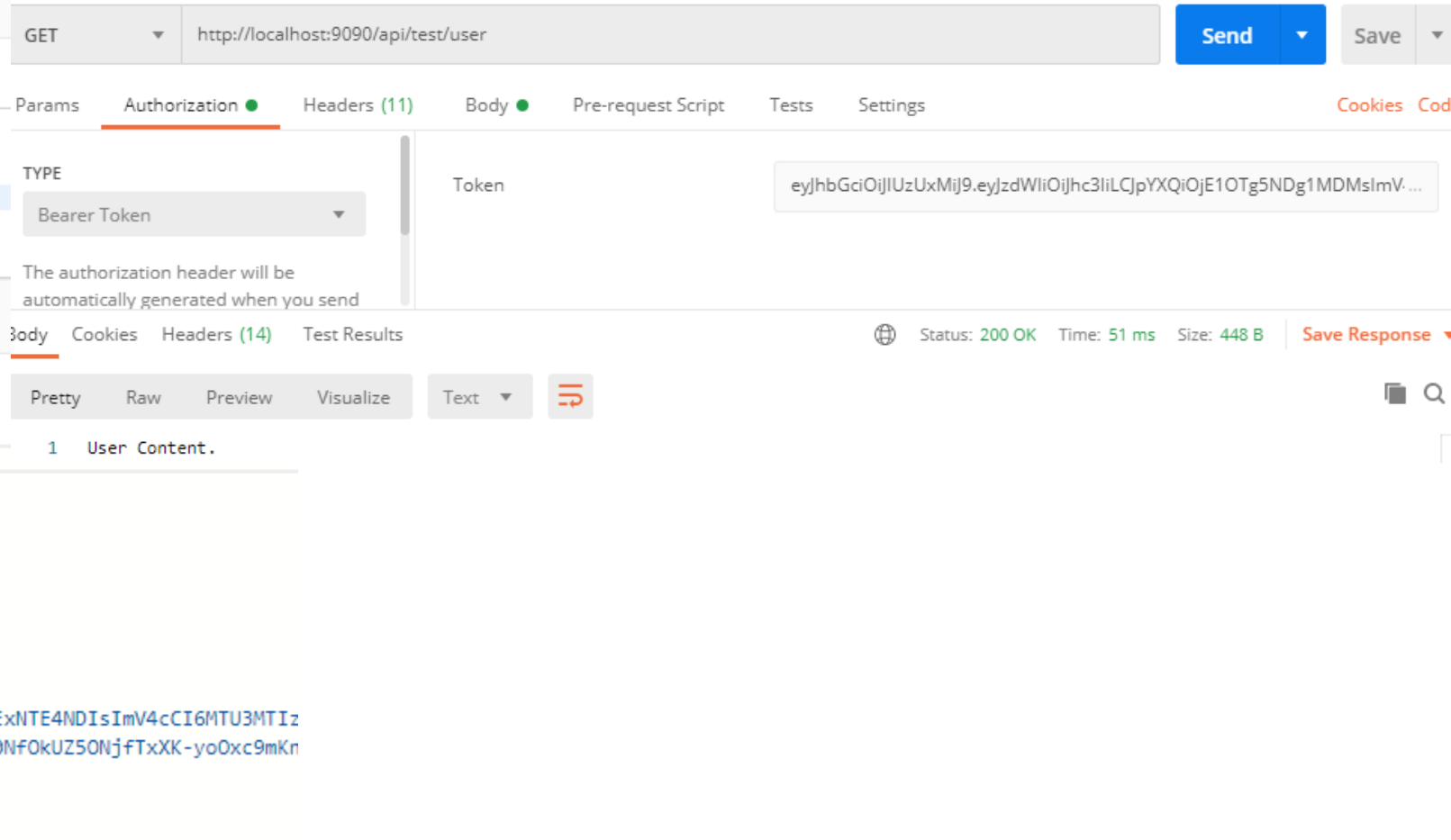
contd..

Run & Test

Login an account: POST /api/auth/signin



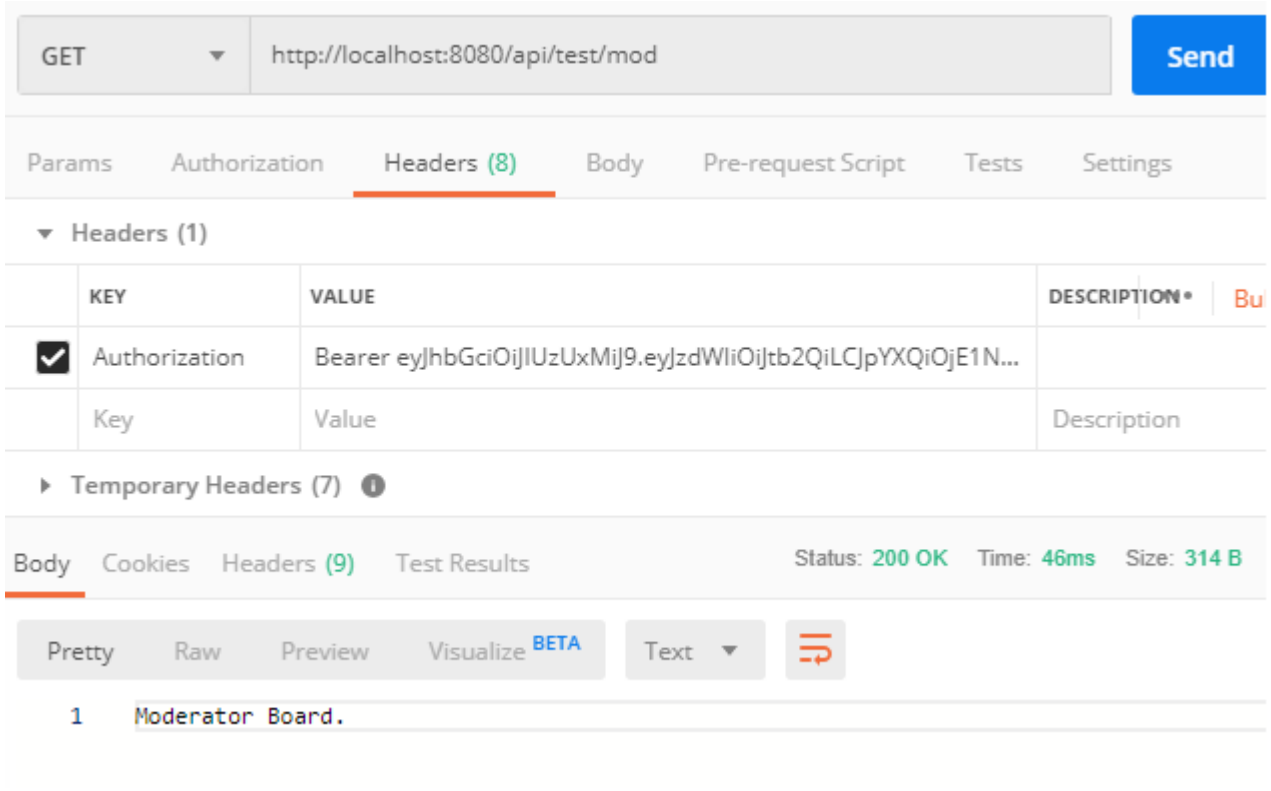
Access ROLE_USER resource: GET /api/test/user



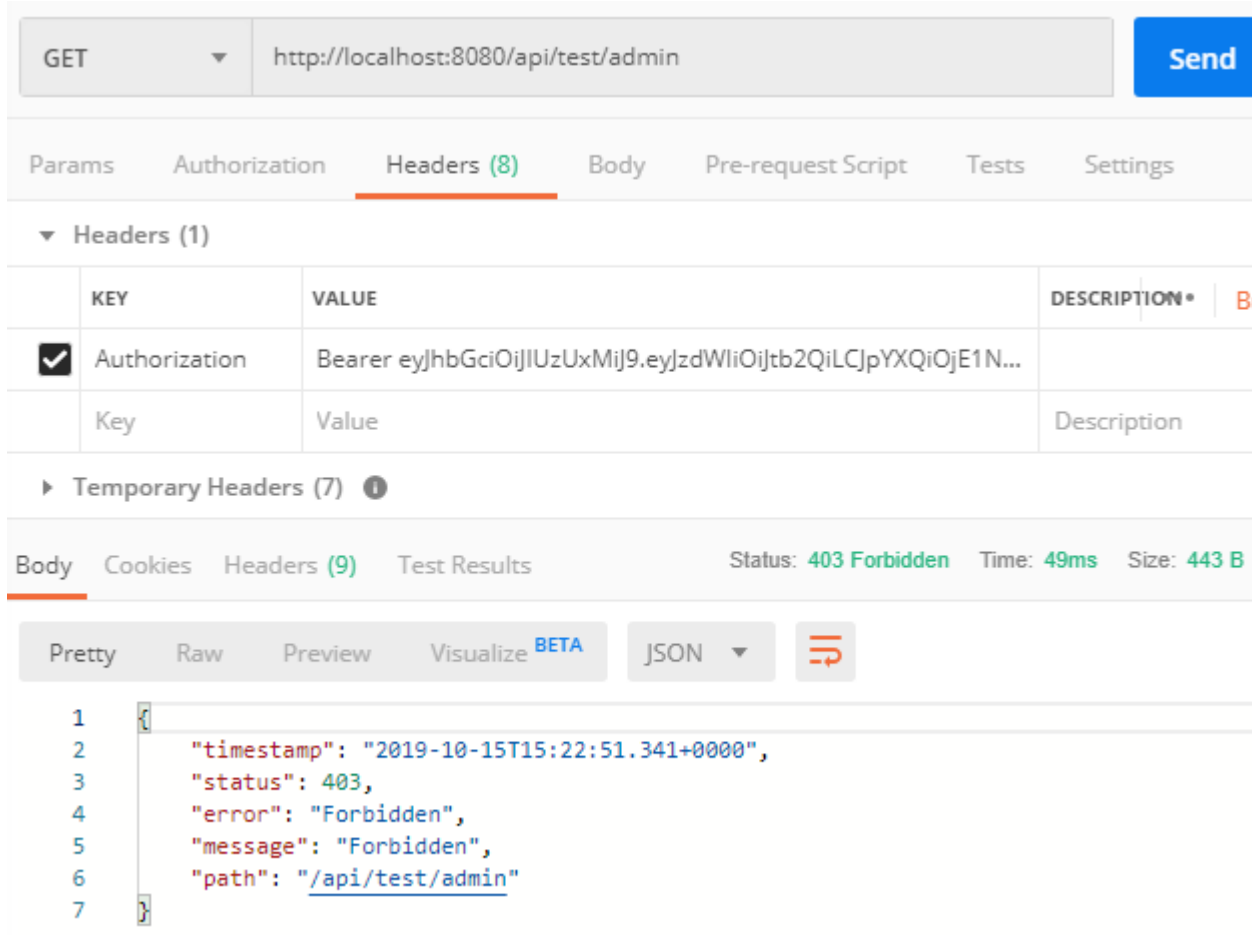
contd..

Run & Test

Access ROLE_MODERATOR resource: GET /api/test/mod



Access ROLE_ADMIN resource: GET /api/test/admin



Thank You!