SpringBoot Basic Security

Basic Configurations are done by Spring Framework Required for Spring Security

1. Open http://start.spring.io

2. Enter details

3. Add dependencies

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.7.5</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <groupId>com.example</groupId>
  <artifactId>spring-security-Database</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>spring-security-Database</name>
  <description>Demo project for Spring Boot</description>
  <properties>
    <java.version>17</java.version>
  </properties>
  <dependencies>
    <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>mysql</groupId>
      <artifactId>mysql-connector-java</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>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework.security</groupId>
      <artifactId>spring-security-test</artifactId>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>

</project>
```

As soon as we add the Spring Security dependency (Spring-Boot-Starter-Security) to the project the basic authentication gets activated by default. If we now start the application, Basic Security is enabled by default by Spring security due to the spring auto configurations.
In the console we get the password (Step 8) while the username is user-

4. Create REST API using Spring Boot and Run the code

5. Edit application.properties file with MySQL related properties

```
spring.datasource.url= jdbc:mysql://localhost:3306/schema_name
spring.datasource.username= username
spring.datasource.password= password
```

6. Add Hibernate properties

```
## Hibernate Properties
# The SQL dialect makes Hibernate generate better SQL for the chosen database
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5InnoDBDialect
spring.jpa.hibernate.ddl-auto = update

## Hibernate Logging
logging.level.org.hibernate.SQL= DEBUG

# Initialize the datasource with available DDL and DML scripts
spring.datasource.initialization-mode=always
spring.jpa.defer-datasource-initialization= true
```

7. Run Spring Boot Application

8. After running we will get a message on application console with Spring Security Generated Password

```
2022-11-22 10:00:02.048  INFO 3480 --- [           main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit
2022-11-22 10:00:02.182  WARN 3480 --- [           main] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefore,
2022-11-22 10:00:02.982  WARN 3480 --- [           main] .s.s.UserDetailsServiceAutoConfiguration :

Using generated security password: 20e446d8-8ea1-49e3-a5c1-ffe8d9a95609

This generated password is for development use only. Your security configuration must be updated before running your application in production.

2022-11-22 10:00:03.285  INFO 3480 --- [           main] o.s.s.web.DefaultSecurityFilterChain     : Will secure any request with [org.springframework.securit
2022-11-22 10:00:03.407  INFO 3480 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context path
2022-11-22 10:00:03.429  INFO 3480 --- [           main] c.R.APIClient.ApiClientApplication       : Started ApiClientApplication in 8.617 seconds (JVM runnir
```

9. Go to browser and Run

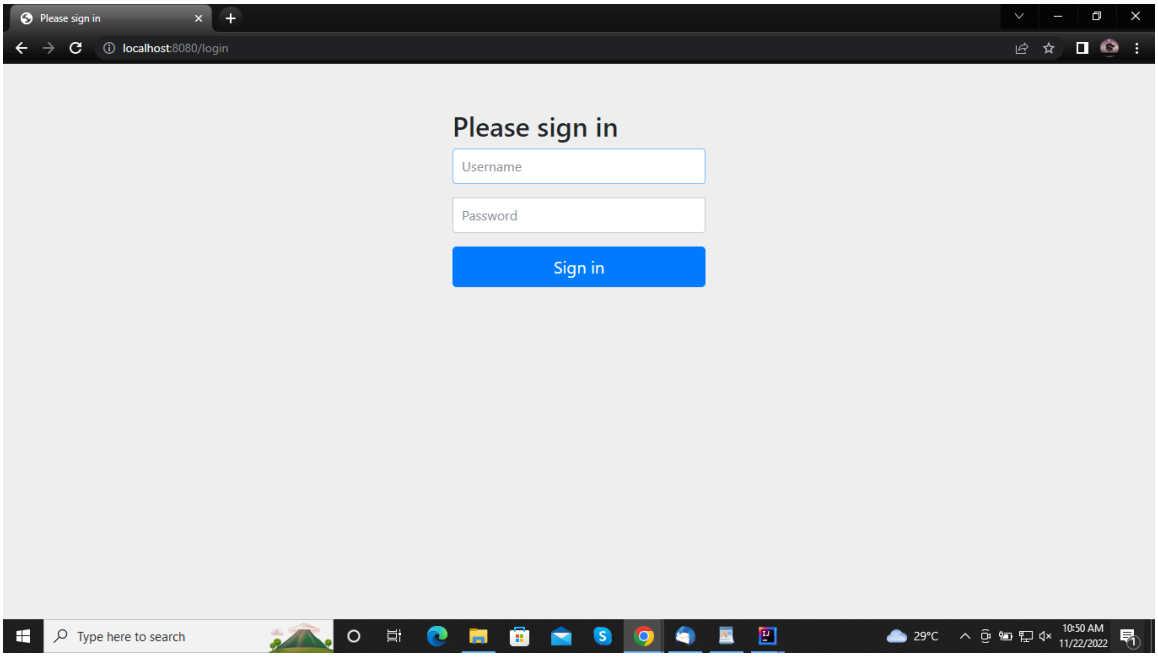
http://localhost:8080/api_endpoint,

you will be redirected to

http://localhost:8080/login

page. (This means our spring security is working properly)

10. Spring Boot Auto Configured Login window will appear



11. Enter Default username as user and password which is on console (auto generated by spring security Point-8 figure)

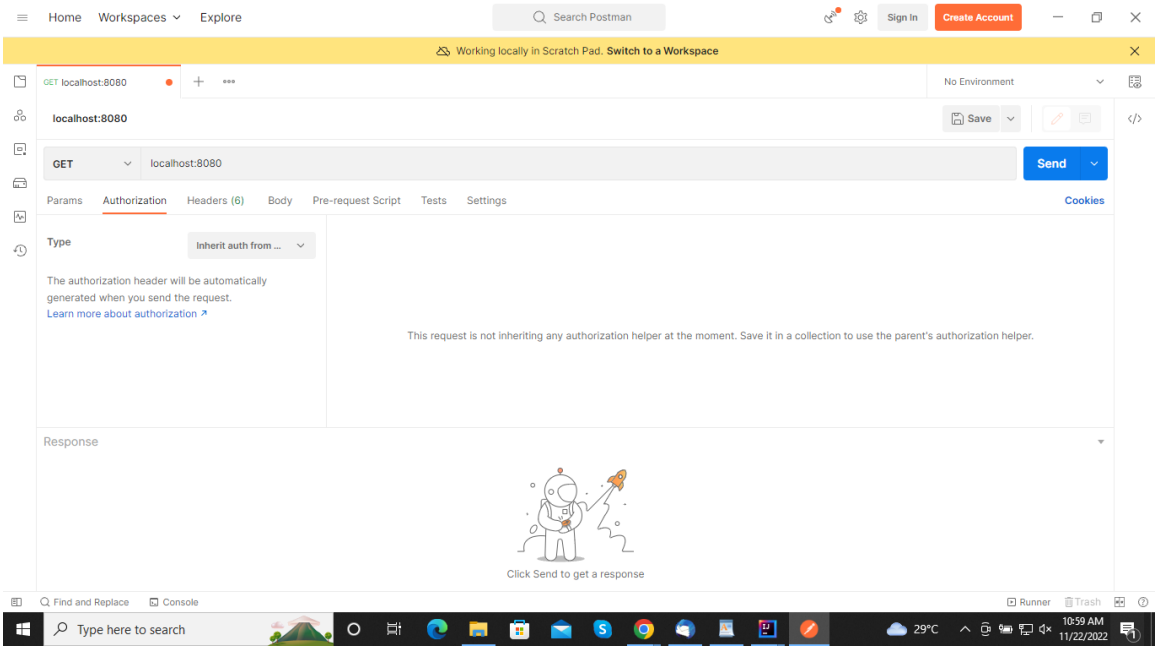
12. After successful validation you will be redirected to your end-point.

Using Postman

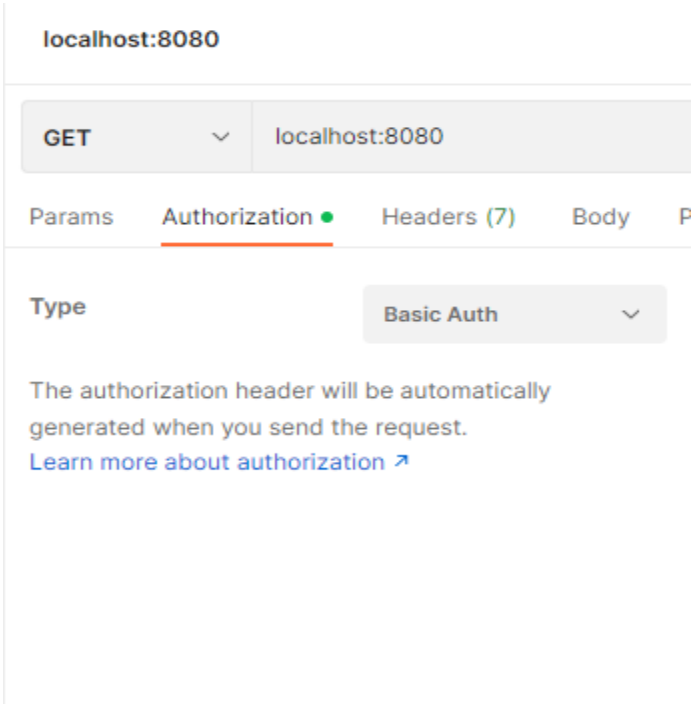
1. To use without browser i.e, with postman Do following steps.

2. Enter end-point in postman with appropriate request method

3. Go to Authorization tab



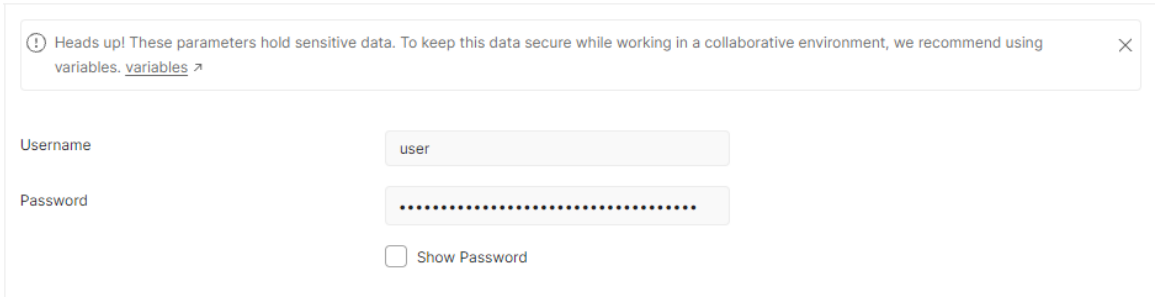
4. In Authorization tab select Basic Auth under Type Drop down.

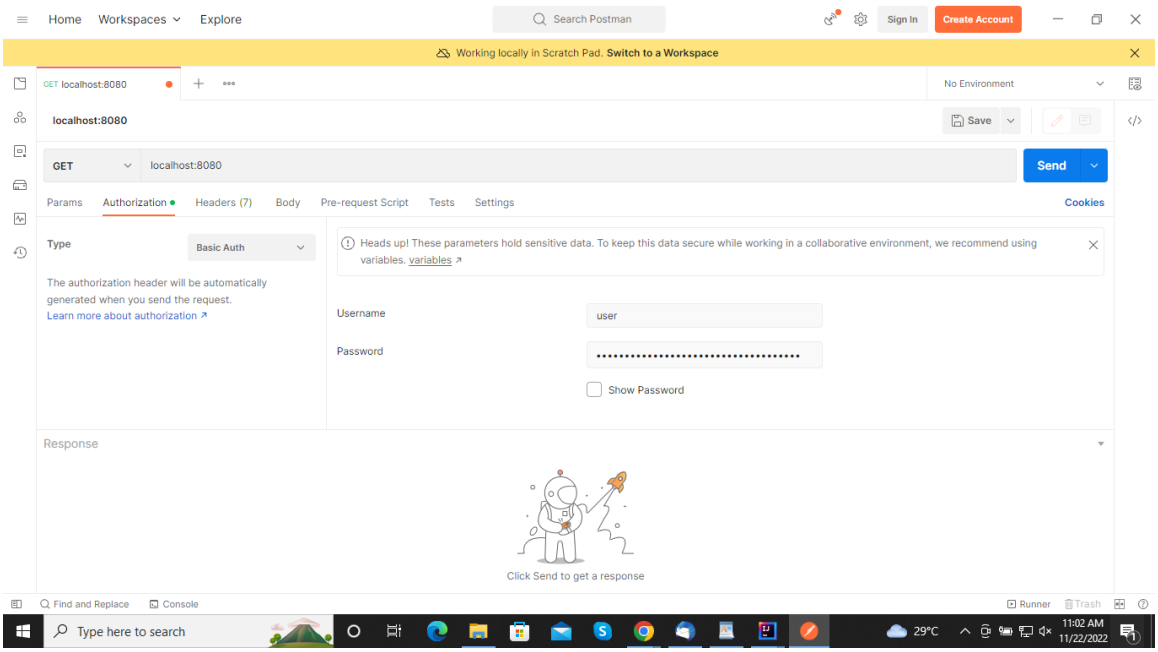


5. Enter Username and Password.

Username - user

Password - Spring Auto Generated Password (From Console)





6. After entering valid credentials click Send and you will receive response from Spring Boot Application.

7. Above login is provided by Spring as default configuration, to make our own configuration to create configuration class.

8. Create new package as configuration class.

9. Create a class named as SecurityConfiguration annotate this class with @Configuration and @EnableWebSecurity

10. To configure our own Spring Security, create a bean class for SecurityFilterChain..

```java
@Configuration
@EnableWebSecurity
public class SecurityConfig {
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception{
        //http configurations
        return http.build();
    }
}
```

11. To permit access to all the end-points following configuration can be used

```java
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception{
    http
            .authorizeHttpRequests() AuthorizeHttpRequestsConfigurer<...>.AuthorizationManagerRequestMatcherRegistry
            .antMatchers( ...antPatterns: "/**") AuthorizeHttpRequestsConfigurer<...>.AuthorizedUrl
            .permitAll();
    return http.build();
}
```

here we are authorizing requests matching with "/" endpoint.**

12. following bean method creates a user with username test and password as password with USER role.

```java
@Bean
public UserDetailsService userDetailsService(){
    UserDetails user = User.withDefaultPasswordEncoder()
            .username("test")
            .password("password")
            .roles("USER")
            .build();
    return new InMemoryUserDetailsManager(user);
}
```

Spring Security with MySQL and Role Based Login and API

1. **Create an Entity class for User**

   User will be Registered officially using the field and save the credentials in database which will be validated with login credentials.

The User model contains the following fields -

id - Primary Key

username - username

email - unique email id

password - A password which will be stored in encrypted format.

roles - Many-to-Many relationship with Role entity

```
@Entity
@Table(
	name = "users",
	uniqueConstraints = @UniqueConstraint(columnNames = {
		"email"
	})
)
public class User {

   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;
   private String username;
   private String email;
   private String password;

   @ManyToMany(fetch = FetchType.EAGER)
   @JoinTable(name = "user_roles",
		joinColumns = @JoinColumn(name = "user_id"),
		inverseJoinColumns = @JoinColumn(name = "role_id")
   )
   private Set<Role> roles;

   public User() {}

   public User(Long id, String username, String email, String password, Set<Role> roles)
{
	this.id = id;
	this.username = username;
	this.email = email;
```

```
        this.password = password;
        this.roles = roles;
    }

    public User(String username, String email, String password, Set<Role> roles) {
        this.username = username;
        this.email = email;
        this.password = password;
        this.roles = roles;
    }

    public Long getId() {
        return id;
    }

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

    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;
    }

    public Set<Role> getRoles() {
        return roles;
    }

    public void setRoles(Set<Role> roles) {
        this.roles = roles;
    }
}
```

2. Create Role class

Fields from Role class –

Name field is an Enum as we have fixed set of pre-defined roles

```
@Entity
@Table(name = "roles")
public class Role {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "role_id")
    private Long id;

    @Enumerated(EnumType.STRING)
    @NaturalId
    private RoleName name;

    public Role() {
    }

    public Role(Long id, RoleName roleName) {
        this.id = id;
        this.name = roleName;
    }

    public Long getId() {
        return id;
    }

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

    public RoleName getRoleName() {
        return name;
    }

    public void setRoleName(RoleName roleName) {
        this.name = roleName;
    }
}
```

3. Create RoleName enum.

Enum with pre-defined roles (ROLE_USER and ROLE_ADMIN)

```
public enum RoleName {
    ROLE_USER,
    ROLE_ADMIN
}
```

4. Creating repositories for entity classes.

```
public interface RoleRepository extends JpaRepository<Role, Long> {
   Optional<Role> findByName(RoleName roleName);
}

@Repository
public interface UserRepository extends JpaRepository<User, Long> {
   User findByEmail(String email);
}
```

5. We need to create custom UserDetails class called UserPrincipal

Instances of this class will be returned from our custom UserDetailsService.

This instance will be used by Spring Security to perform Authentication and Authorization

```
public class UserPrincipal implements UserDetails {

   private Long id;

   private String username;

   private String email;

   private String password;

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

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

   private Collection<? extends GrantedAuthority> authorities;

   public UserPrincipal(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;
    }

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

    public Long getId() {
        return id;
    }

    public String getEmail() {
        return email;
    }

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

    @Override
    public String getUsername() {
        return this.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;
    }
}
```

6. Creating CustomUserDetailsService class

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

For this purpose, we will create CustomUserDetailsService class which implements UserDetailsService interface and we need to provide implementation for loadByUsername() method.

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

```
@Service
public class CustomUserDetailsService implements UserDetailsService {

   @Autowired
   private UserRepository userRepository;

   @Override
   @Transactional
   public UserDetails loadUserByUsername(String email) throws
UsernameNotFoundException {
      User user = userRepository.findByEmail(email);
      return UserPrincipal.create(user);
   }

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

We have also defined a bean for PasswordEncoder which will return BCryptPasswordEncoder.

7. Creating a RestController for Registration

We will need a Controller to persist the user in database

This end-point will need users data i.e, username, email and password

To pass this data we will create DTO/payload class

```
@RestController
public class AuthController {

   @Autowired
   private PasswordEncoder passwordEncoder;
```

```
    @Autowired
    private RoleRepository roleRepository;

    @Autowired
    private UserRepository userRepository;

    @PostMapping("/register")
    public String registerUser(@RequestBody UserRegistration userPayload){
        User user = new User();
        user.setUsername(userPayload.getUsername());
        user.setEmail(userPayload.getEmail());
        user.setPassword(passwordEncoder.encode(userPayload.getPassword()));

        Role userRole = roleRepository.findByName(RoleName.ROLE_USER).get();

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

        userRepository.save(user);

        return "Registered Successfully";

    }

    @PostMapping("/registerAdmin")
    public String registerAdmin(@RequestBody UserRegistration adminPayload){
        User user = new User();
        Role userRole = roleRepository.findByName(RoleName.ROLE_ADMIN).get();

        user.setUsername(adminPayload.getUsername());
        user.setEmail(adminPayload.getEmail());
        user.setPassword(passwordEncoder.encode(adminPayload.getPassword()));
        user.setRoles(Collections.singleton(userRole));

        userRepository.save(user);
        return "Registered Successfully";
    }


}
```

8. Creating dto/payload classes

This class will be used to transfer data from request payload to User class object, which will saved in databse with password encryption

```
public class UserRegistration {
    private String username;
    private String email;
    private String password;

    public UserRegistration() {
    }
```

```
    public UserRegistration(String username, String email, String password) {
        this.username = username;
        this.email = email;
        this.password = password;
    }

    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;
    }
}
```

9. Updating SecurityConfiguration

In this configuration we will add role based authorization. Specific set of endpoints will be accessed by different different roles.

Here api with "/" will be accessible to all, "/secured" will be accessible to users having roles USER and ADMIN and "/admin" will be accessible to users with ADMIN role

```
@Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception{
        http
                .cors()
                .and()
                .csrf()
                .disable()
                .authorizeRequests()
                .antMatchers("/")
                .permitAll()
```

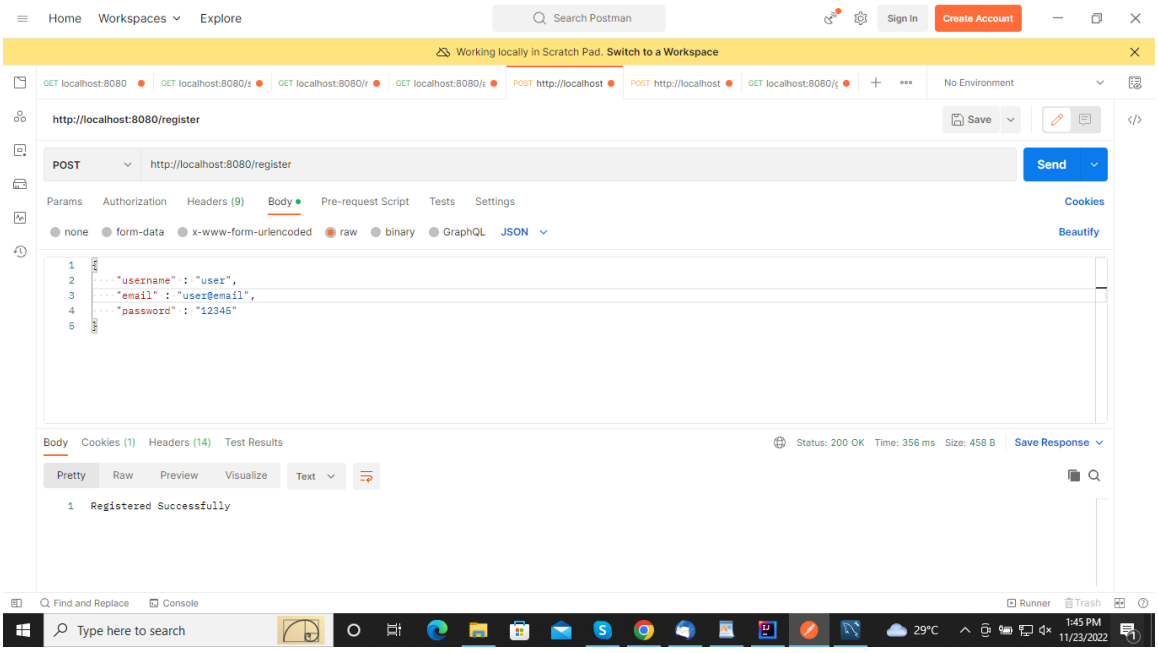
```
                .antMatchers("/secured")
                .hasAnyRole("USER", "ADMIN")
                .antMatchers("/admin")
                .hasRole("ADMIN")
                .and()
                .formLogin()
                .and()
                .httpBasic();
        return http.build();
    }
```

add one more bean method to provide user data from database

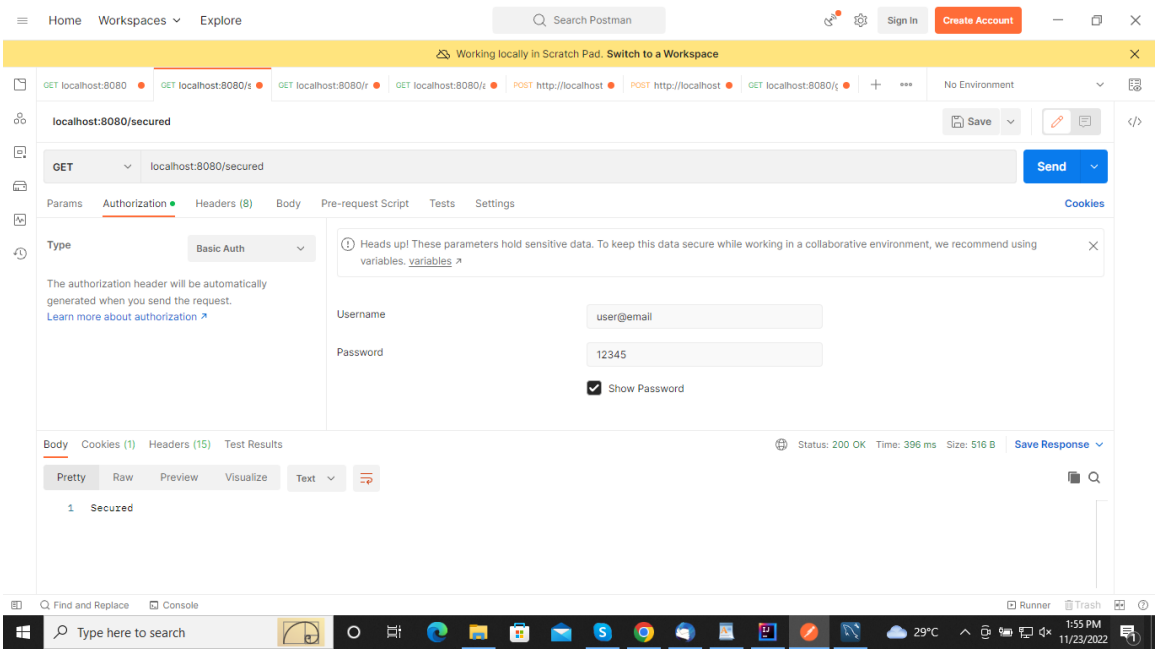
```
@Bean
public DaoAuthenticationProvider authenticationProvider(){
    DaoAuthenticationProvider authenticationProvider = new DaoAuthenticationProvider();
    authenticationProvider.setUserDetailsService(userDetailsService);
    authenticationProvider.setPasswordEncoder(passwordEncoder);

    return authenticationProvider;
}
```

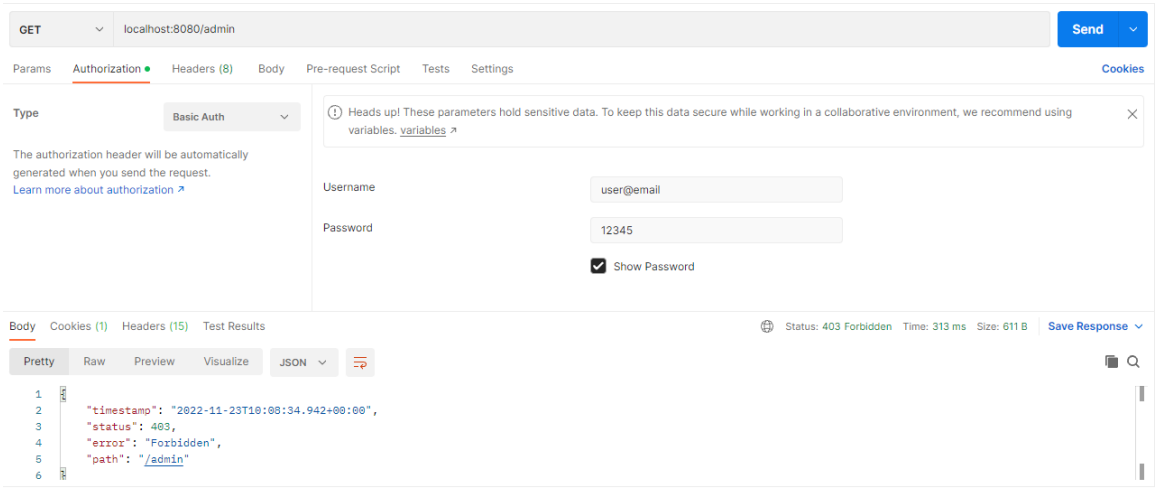
10. Registring Users using postman



11. Using above credentials to use authenticated api



12. Accessing unauthorized endpoints



here we are trying to access "/admin" endpoint with "USER" role, so we are not able to access that endpoint as it is only accessible to "ADMIN" user.

- **Source**
- **Spring Boot JavaDoc :-**
- https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/config/annotation/web/builders/HttpSecurity.html
- https://spring.io/guides/gs/securing-web/