# Spring Security

**Core Filters in Spring Security**

1. **SecurityContextPersistenceFilter**: Manages the SecurityContext for the current request. It stores the context in the HttpSession between requests.
2. **LogoutFilter**: Handles the logout process. It clears the SecurityContext and invalidates the session.
3. **UsernamePasswordAuthenticationFilter**: Processes authentication requests using a username and password.
4. **BasicAuthenticationFilter**: Handles HTTP Basic authentication.
5. **BearerTokenAuthenticationFilter**: Processes authentication requests using a bearer token.
6. **CsrfFilter**: Protects against Cross-Site Request Forgery (CSRF) attacks.
7. **ExceptionTranslationFilter**: Handles any AccessDeniedException or AuthenticationException thrown within the filter chain.
8. **FilterSecurityInterceptor**: Intercepts requests to secure resources and enforces access control.

Data

Gallery

Login

Home

**Controller**

Filter 4

Filter 5

Filter 6

Filter **.**

Filter 3

Filter 2

Filter 1

**Servlet Container**

**Client**

**Website / Apps**

Tomcat server

Public String myfunction ( **HttpServletRequest request** )

{

String sid =  **request.getSession( ).getId ( );**

return sid ;

}

**Cross-Site Request Forgery (CSRF) :**

**@Configuration :** It tells that there is a configuration file read this configuration file and change the default settings A/C to given or defined rule in this configuration file .

**@EnableWebSecurity :** It tells that enabe this configuration setting .

For Changing the default Security Configuration

|  |
| --- |
| @Configuration  @EnableWebSecurity  public class SecurityConfig  {  @Bean  Public SecurityFilterChain securityFilterChain ( HttpSecurity http) throws Exception  {  return http.build( );  }  } |

* **http.csrf (customizer .disable( ) ); :**  To disable the CSRF
* **http.authorizeHttpRequests ( request - > request.anyRequest( ) .authenticated ( ) ); :** To start the authentication of the controller .
* **http.formLogin( Customizer.withDefaults( )); :**
* **http.httpBasic( Customizer.withDefaults( ) ); :**
* **http.sessionManagement (session -> session.sessionCreationPolicy (SessionCreationPolicy.*STATELESS*) );**

change ***STATELESS*** to ***ALWAYS*** for browser . To login one time and use

For some path that is permitted use this

**http.authorizeHttpRequests ( request - > request**

**.requestMatchers( “** *login* **”, “** *path1* **”, “** *path2.\*\****”, … … … … )**

**.permitAll()**

**.anyRequest( ) .authenticated ( ) );**

For role based

**.requestMatchers("/admin/\*\*").hasRole("ADMIN")**

**.requestMatchers("/user/\*\*").hasRole("USER")**

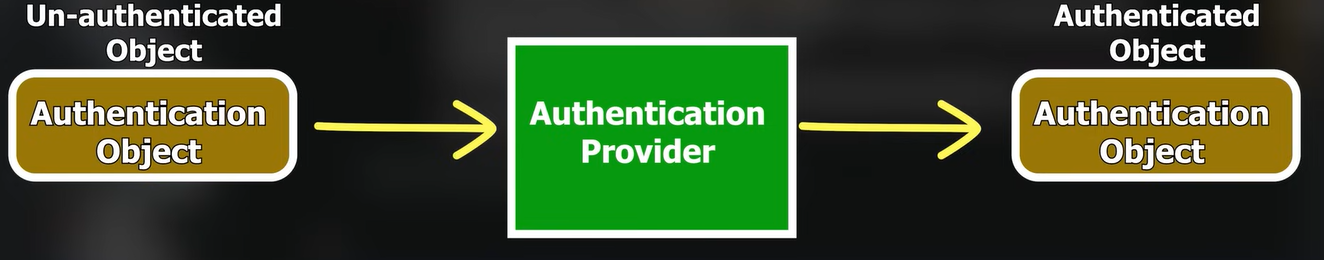
**.anyRequest( ) .authenticated ( ) );**

|  |
| --- |
| @Configuration @EnableWebSecurity public class SecurityConfig {  @Bean  public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {  return http  .csrf(customizer -> customizer.disable())  .authorizeHttpRequests(requests -> requests.anyRequest().authenticated())  .httpBasic(Customizer.*withDefaults*())  .formLogin(Customizer.*withDefaults*())  .build();  } } |

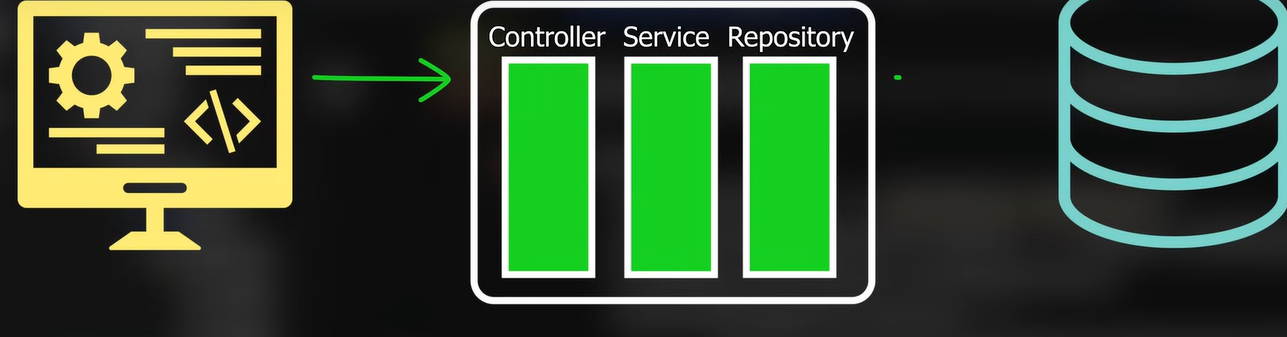
## To use our own User and password

### For the Default user

|  |
| --- |
| **@Bean**  **public UserDetailsService userDetailsService ( )**  **{**  **UserDetails** user1 = **User.withDefaultPasswordEncoder( )**  **.username(“** SK **”) .password(“** 362004 **” ) .roles( “**USER**”) .build( );**  **UserDetails** user2 = **User.withDefaultPasswordEncoder( )**  **.username(“** SK **”) .password(“** 362004 **” ) .roles( “**ADMIN**”) .build( );**  **return new** InMemoryUserDetailsManager( user1, user2, ………….. ) ; // it can take multiple args…  **}** |



For the Database using user



|  |
| --- |
| @Entity  Public class UserEntity  {  Private int id;  Private String username;  Private String password;  //------- Other Getters Setters and etc …. -------------------  } |

|  |
| --- |
| @Repository  Public interface UserRepo extends JpaRepository< UserEntity , Integer >  {  public UserEntity findByUsername( String username );  } |

|  |
| --- |
| @Service  Public class MyuserDetailsService implements **UserDetailsService**  {  @Autowired  private UserRepo qury;    public UserDetails loadUserByUsername ( String username ) throws UsernameNotFoundException  {  UserEntity user = qury.findByUsername (username );  if ( user == null ) { **throw** new UserNameNotFoundException (“ Sorry no User found with this name“);    return **new UserPrincipal (** user **); //**  returning UserDetails object here.  }  } |

|  |
| --- |
| public class UserPrincipal implements UserDetails {   private UserEntity user;   public UserPrincipal( UserEntity user) { this.user = user; }  public Collection<? extends GrantedAuthority> getAuthorities() {  return Collections.*singleton*(new SimpleGrantedAuthority("USER"));  }    public String getPassword( ) { return user.getPhone(); }   public String getUsername( ) { return user.getName(); }   public boolean isAccountNonExpired( ) { return true; //UserDetails.super.isAccountNonExpired(); }    public boolean isAccountNonLocked( ) { return true; // UserDetails.super.isAccountNonLocked(); }    public boolean isCredentialsNonExpired() { return true; //UserDetails.super.isCredentialsNonExpired(); }    public boolean isEnabled() { return true; //UserDetails.super.isEnabled(); } } |

|  |
| --- |
| @Autowired  private **UserDetailsService** myUserDetailsService ;  @Beans  public AuthenticationProvider authenticationProvider ( )  {  DaoAuthenticationProvider provider = new DaoAuthenticationProvider ( );    provider.setPasswordEncoder ( NoOpPasswordEncoder.getInstance ( ) );  provider.setUserDetailsService ( myUserDetailsService) ;  return provider;  } |

<https://github.com/mthshubham/Spring.git>

Encoding Password:

**Step 1 : --**  Use it while saving data in Database.

private **BCryptPasswordEncoder encoder = new BCryptPasswordEncoder (** Sttength of Encoding **) ; //** example : 10

user.setPassword( **encoder. encode (** user.getPassword( ) **)** ) **;**

**Step 2 : --** Use it in web security config file .

provider.setPasswordEncoder ( **new BCryptPasswordEncoder (** Sttength of Encoding **)** ) ;

**Role Based Authentication**

.authorizeHttpRequests(  
 requests -> requests  
 .requestMatchers("/","/signup","/login").permitAll() .requestMatchers("/admin/\*\*").hasAnyRole("ADMIN","USER")

.requestMatchers("/admin/allusers").hasRole("ADMIN")

.anyRequest().authenticated()

)

* When using **hasAnyRole ("ADMIN", "USER")** or **hasRole(“ADMIN”)** , Spring Security expects roles to have a default prefix of ROLE\_. This means the roles stored in your database or defined in your system should be ROLE\_ADMIN and ROLE\_USER.
* If your roles don't have the ROLE\_ prefix, you can configure Spring Security to not expect it by customizing GrantedAuthority mapping.

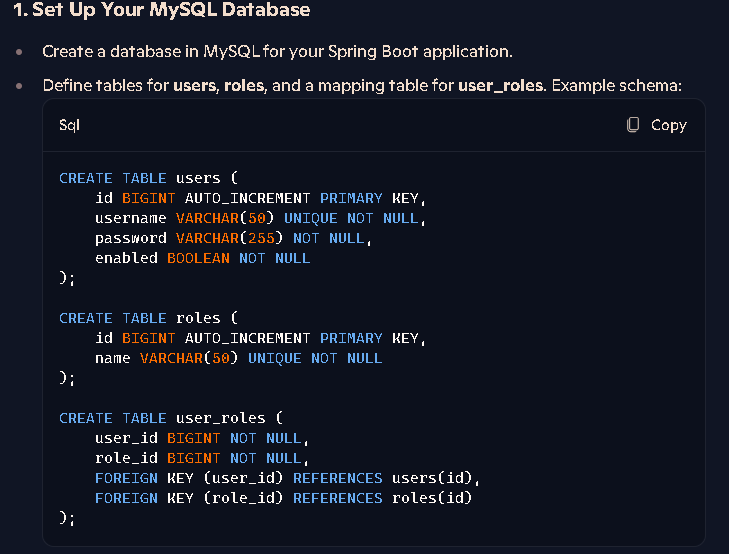
**Solution**: UserDetails implemented class function

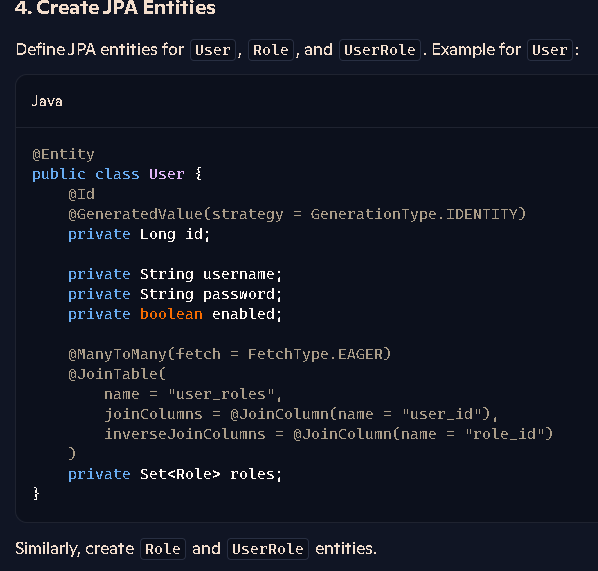
public Collection<? extends GrantedAuthority> getAuthorities() {  
 return Collections.*singleton*(new SimpleGrantedAuthority("ROLE\_ADMIN"));  
 //return List.of();  
}

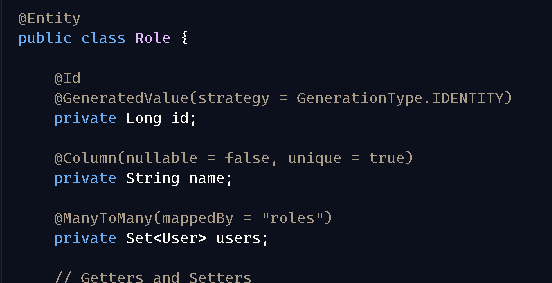
* Ensure roles are prefixed with ROLE\_, or use hasAuthority("ADMIN") instead of hasAnyRole.

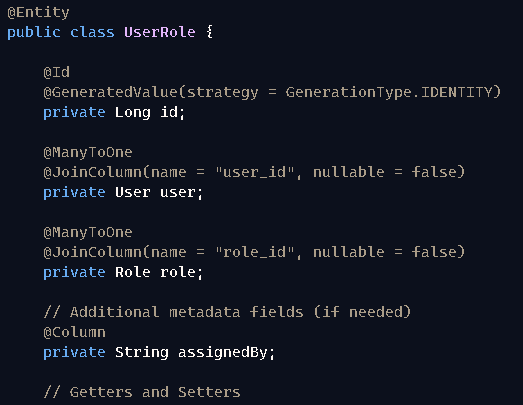
Or use this

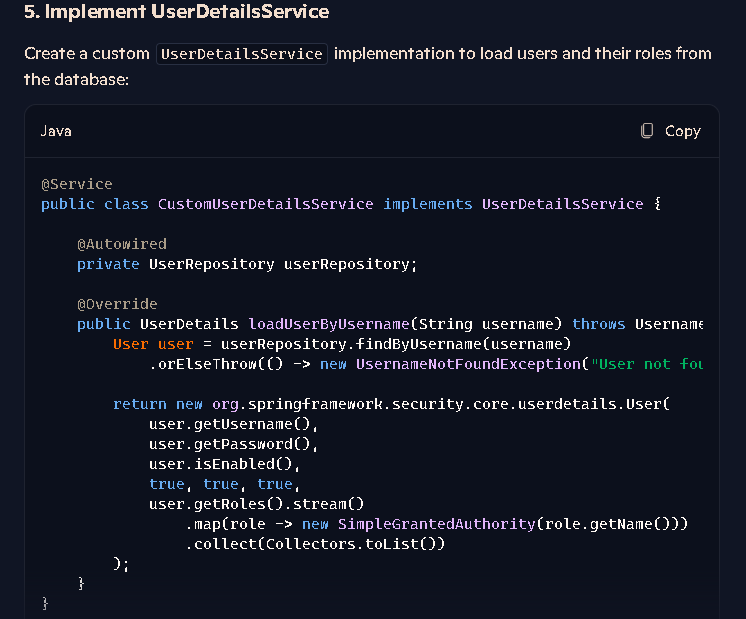
example:--- .**requestMatchers("/admin/\*\*").hasAuthority("ADMIN")**

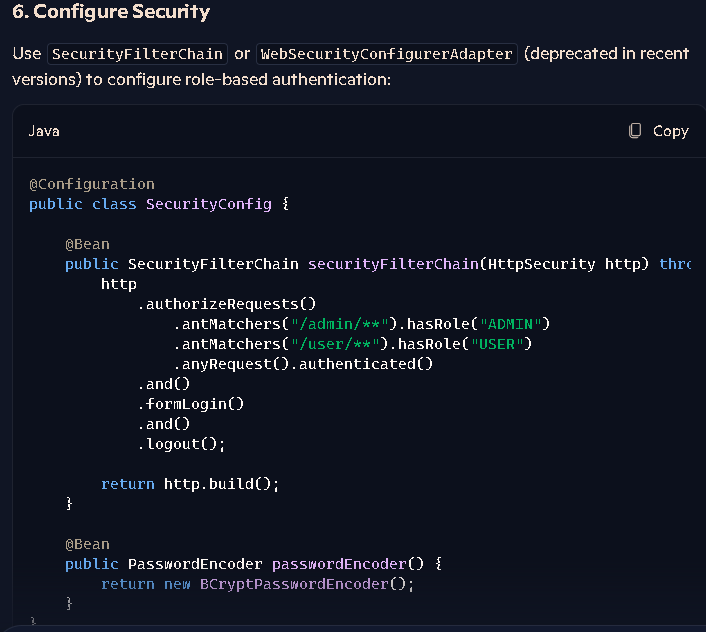












JWT Jason( JavaScript Object Notaion ) Web Token

Add dependency :

1. JJWT api
2. JJWT impl
3. JJWT Jackson

Security Config file

|  |
| --- |
| **@**Bean  public **AuthenticationManager** authenticationManagerBean(**AuthenticationConfiguration** config )  throws Exception {  return config.getAuthenticationManager(); } |

Controller.

|  |
| --- |
| **@**Postmapping (“/login”)  public String login (@ReuestBody User user)  {  return myser.verify( user );  } |

|  |
| --- |
| @Autowired  private **AuthenticationManager auth;**  public String verify ( User user)  {  Authentication authentication = auth. authenticate **(**  new **UsernamePasswordAuthenticationToken** (  user.getUsername ( ) , user.getPassword ( ) ) **)** ;  if (authentication.isAuthenticated ( ) )  {  return “Success” ;  }  return “ Sorry Failed ” ;  } |
|  |

|  |
| --- |
| import java.awt.image.BufferedImage;  import java.io.File;  import java.io.IOException;  import javax.imageio.ImageIO;  public class ImageExample {  public static void main(String[] args) {  try {  // Read an image from file  File inputFile = new File("path/to/your/input/image.jpg");  BufferedImage image = ImageIO.read(inputFile);  // Write the image to a new file  File outputFile = new File("path/to/your/output/image.jpg");  ImageIO.write(image, "jpg", outputFile);  System.out.println("Image successfully read and saved!");  } catch (IOException e) {  e.printStackTrace();  }  }  } |

**------------------------------ IMAGE Processing ---------------------------------------**

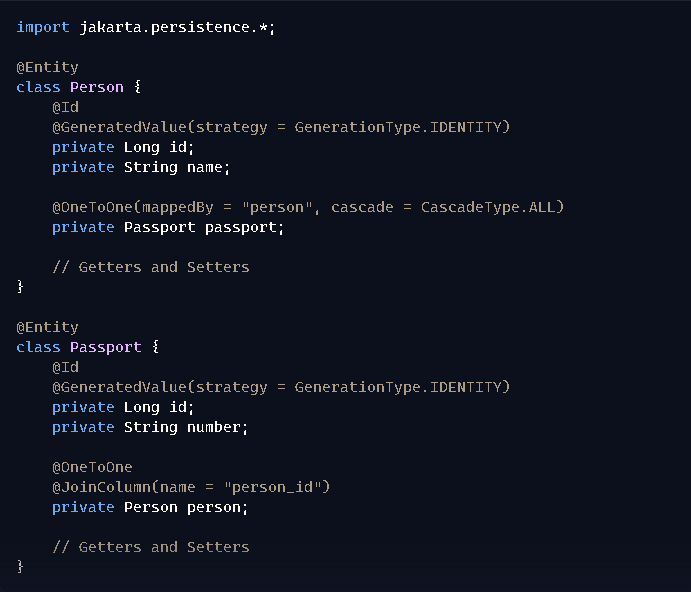
|  |
| --- |
| import java.awt.image.BufferedImage;  import java.io.ByteArrayOutputStream;  import java.io.File;  import java.io.IOException;  import java.sql.Connection;  import java.sql.DriverManager;  import java.sql.PreparedStatement;  import javax.imageio.ImageIO;  public class ImageToDatabase {  public static void main(String[] args) {  Connection connection = null;  PreparedStatement statement = null;  try {  // Read the image  File imageFile = new File("path/to/your/image.jpg");  BufferedImage image = ImageIO.read(imageFile);  ByteArrayOutputStream baos = new ByteArrayOutputStream();  ImageIO.write(image, "jpg", baos);  byte[] imageBytes = baos.toByteArray();  // Connect to the database  connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/yourdatabase", "username", "password");  String sql = "INSERT INTO images (image) VALUES (?)";  statement = connection.prepareStatement(sql);  statement.setBytes(1, imageBytes);  // Execute the statement  statement.executeUpdate();  System.out.println("Image successfully saved to the database!");  } catch (IOException | SQLException e) {  e.printStackTrace();  } finally {  try {  if (statement != null) statement.close();  if (connection != null) connection.close();  } catch (SQLException e) {  e.printStackTrace();  }  }  }  } |

**Database Relationship**

1. **One-to-One Relationship**

**Example**: A person and their passport.

* **Person**: Only one passport belongs to this person.
* **Passport**: It is issued to only one person.

**How it works**:

* The @OneToOne annotation establishes a one-to-one relationship.
* The mappedBy attribute defines which entity owns the relationship.

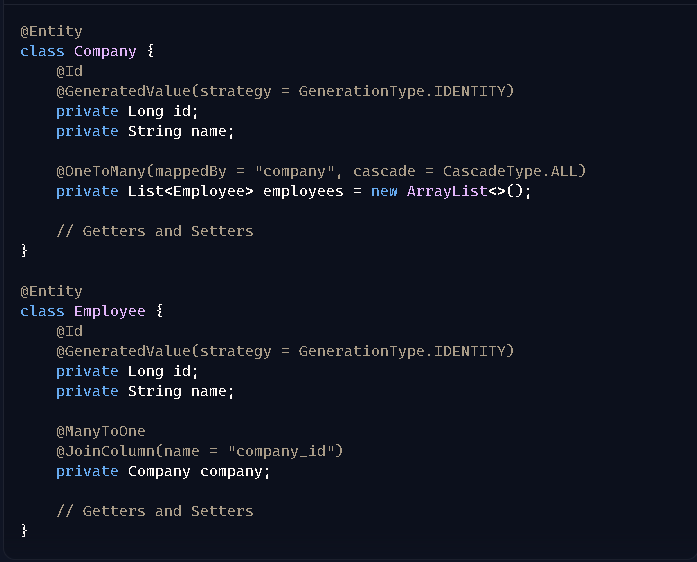
**2. One-to-Many Relationship**

**Example**: A company and its employees.

* A company can have many employees, but each employee works for one company.

**Example**: Company and employees.

* **Company**: Can hire multiple employees.
* **Employee**: Works for one company.



**How it works**:

* @OneToMany establishes the relationship on the **Company** side.
* @ManyToOne establishes the relationship on the **Employee** side.

**3. Many-to-One Relationship**

**Example**: Orders and a customer.

* Many orders can belong to one customer.



**How it works**:

* Similar to the one-to-many relationship, except **one side (customer)** is the focus here.

**4. Many-to-Many Relationship**

**Example**: Students and courses.

* A student can enroll in multiple courses, and a course can have multiple students.

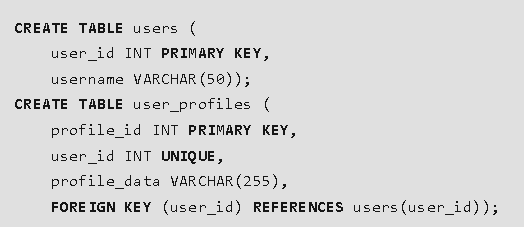


**How it works**:

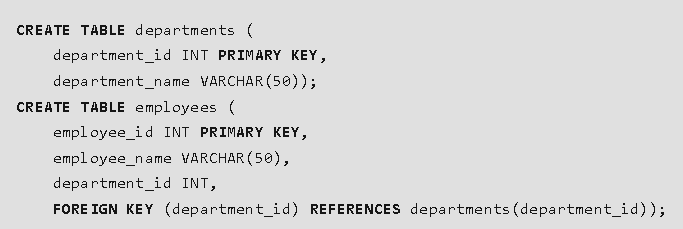
* @ManyToMany establishes the relationship.
* @JoinTable specifies the join (or association) table.

**Key Points:**

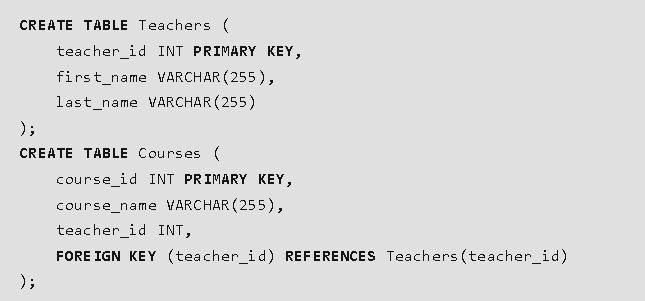
1. **Annotations**: JPA annotations (@OneToOne, @OneToMany, @ManyToOne, and @ManyToMany) define how relationships work.
2. **Cascading**: The cascade property allows related entities to be persisted/updated/deleted automatically.
3. **Join Column**: Used to link foreign keys between tables.
4. **One to One**



1. **One to Many**



1. **Many to One**



Many Courses can be teach by a single teacher

1. **Many to Many**

