

Lập Trình JAVA

CyberSoft Academy

Giảng Viên : Chế Công Bình

...

Tổng quan - Spring Security



CYBERSOFT
ĐÀO TẠO CHUYÊN GIA LẬP TRÌNH

- 1) Spring Security là gì
- 2) Tại sao lại phải sử dụng Spring Security
- 3) Cách thức hoạt động của Spring Security
- 4) Thực hành spring security

Kết quả đạt được

Please sign in

Sign in

Spring Security là gì

- Spring Security là một framework cung cấp các phương thức chứng thực như **Authentication**, **Authorization** để bảo vệ ứng dụng khỏi sự tấn công thường gặp của hacker
- Thông qua Spring Security chúng ta có thể làm các chức năng liên quan tới phân quyền sử dụng tài nguyên hoặc chức năng của User



- LDAP (Lightweight Directory Access Protocol)
- Single sign-on
- JAAS (Java Authentication and Authorization Service) LoginModule
- LDAP (Lightweight Directory Access Protocol)
- Basic Access Authentication
- Digest Access Authentication
- Remember-me
- Web Form Authentication
- Authorization
- Software Localization
- HTTP Authorization

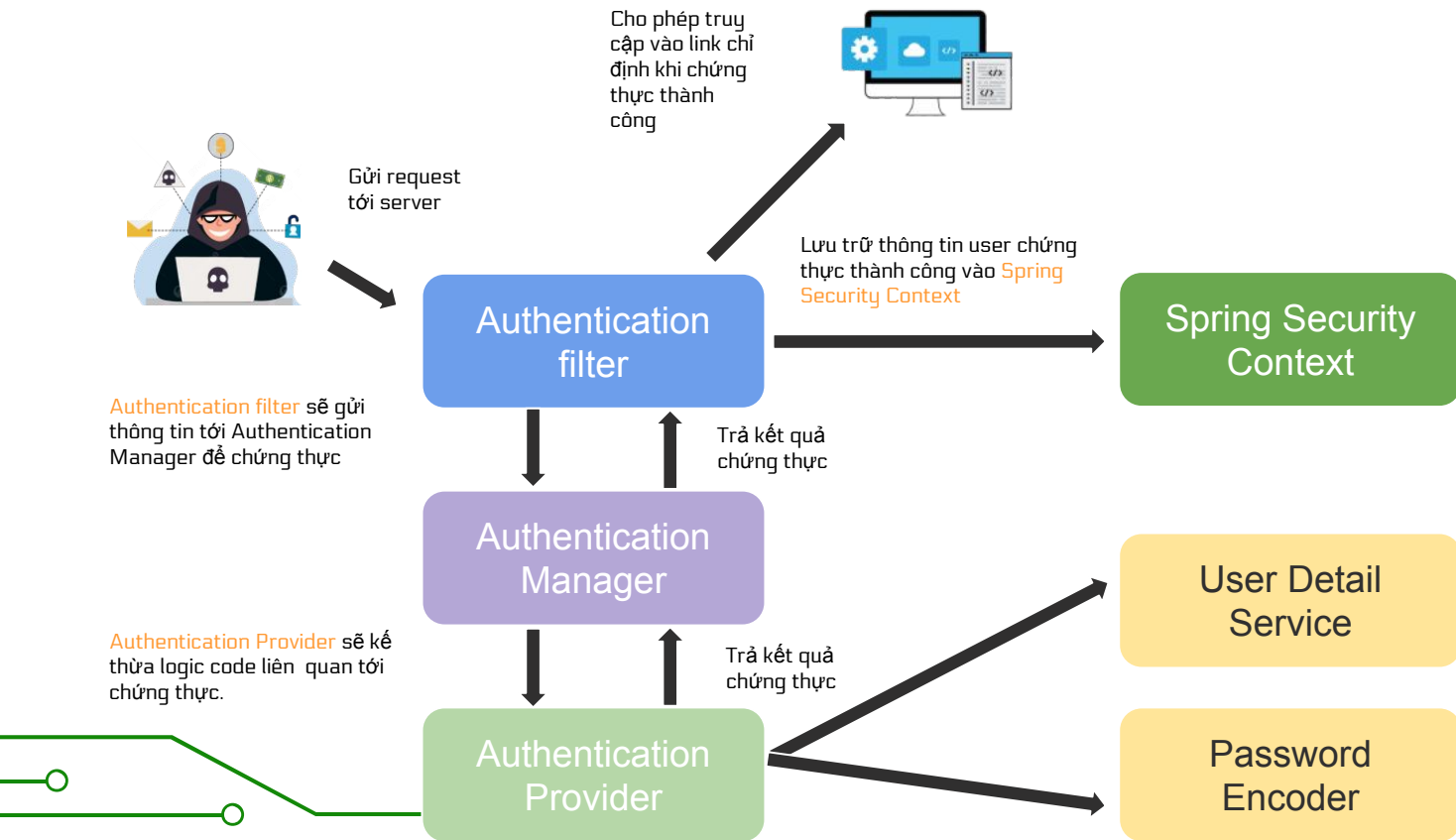


Tính năng Spring Security

Access Control và Authorization

- Trong vấn đề bảo mật tập trung vào 2 vấn đề chính : **Authentication** [Bạn là ai ?] và **Authorization** [Bạn được phép làm gì ?] . Đôi khi mọi người hay sử dụng “Access control” thay cho “Authorization” điều này dễ gây tới hiểu lầm bởi vì đây là 2 khái niệm khác nhau.
- **Access control** : Quản lý quyền được phép sử dụng chức năng trong ứng dụng. Tức là user đã chức thực thành công nhưng không được phép thực hiện một số chức năng trong ứng dụng.
- **Authorization** : Quản lý user có được phép truy cập vào ứng dụng hay không

Sơ đồ hoạt động của Spring Security



Authentication Provider sẽ dùng **User Detail Service** để tìm kiếm thông tin user đăng nhập và kiểm tra mật khẩu theo chuẩn mã hóa đã được quy định

Authentication Manager

```
public interface AuthenticationManager {  
  
    Authentication authenticate(Authentication authentication)  
        throws AuthenticationException;  
}
```

- Trong authentication sẽ hỗ trợ interface cho việc chứng thực là **AuthenticationManager**.
- AuthenticationManager hỗ trợ phương thức chứng thực là **authenticate()**.
- Phương thức **authenticate()** sẽ trả về một đối tượng **Authentication** nếu chứng thực thành công hay không sẽ dựa vào các rule Validation mà khách hàng hoặc người lập trình quy định.



Authentication Manager


```
public interface AuthenticationProvider {  
    Authentication authenticate(Authentication authentication)  
        throws AuthenticationException;  
    boolean supports(Class<?> authentication);  
}
```

- Thông thường khi custom chứng thực sẽ kế thừa interface **AuthenticationProvider** đây là một interface con của **AuthenticationManager**.
- **AuthenticationProvider** sẽ giống như **AuthenticationManager** nhưng khác là **AuthenticationProvider** có hỗ trợ loại chứng thực thông qua hàm **supports()**.



Tạo dự án với Spring Security

start.spring.io

 **spring** initializr

Project

☒ **Gradle - Groovy** ☐ Gradle - Kotlin ☐ Maven

Language

☒ **Java** ☐ Kotlin ☐ Groovy

Spring Boot

☐ 3.1.0 (SNAPSHOT) ☐ 3.1.0 (RC1) ☐ 3.1.0 (M2) ☐ 3.0.7 (SNAPSHOT) ☒ **3.0.6** ☐ 2.7.12 (SNAPSHOT) ☐ 2.7.11

Project Metadata

Group

com.cybersoft

Artifact

springsecurity

Name

springsecurity

Description

The demo excuted by the cybersoft academy

Package name

com.cybersoft.springsecurity

Packaging

☒ **Jar** ☐ War

Java

☐ 20 ☒ **17** ☐ 11 ☐ 8

Dependencies

ADD DEPENDENCIES... % + B

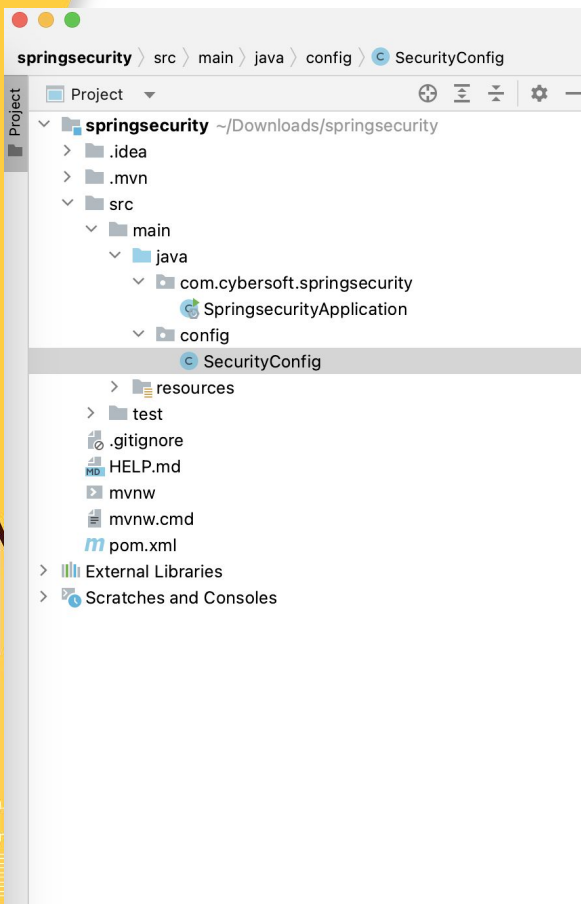
Spring Web **WEB**

Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Spring Security **SECURITY**

Highly customizable authentication and access-control framework for Spring applications.

Tạo dự án với Spring Security



- Tạo class **SecurityConfig** để tiến hành kế thừa Spring Security và quy định rule Security cho ứng dụng.
- Package **config** là nơi chứa tất cả các class liên quan tới cầu hình của ứng dụng.



Tạo dự án với Spring Security

Run: SpringsecurityApplication x

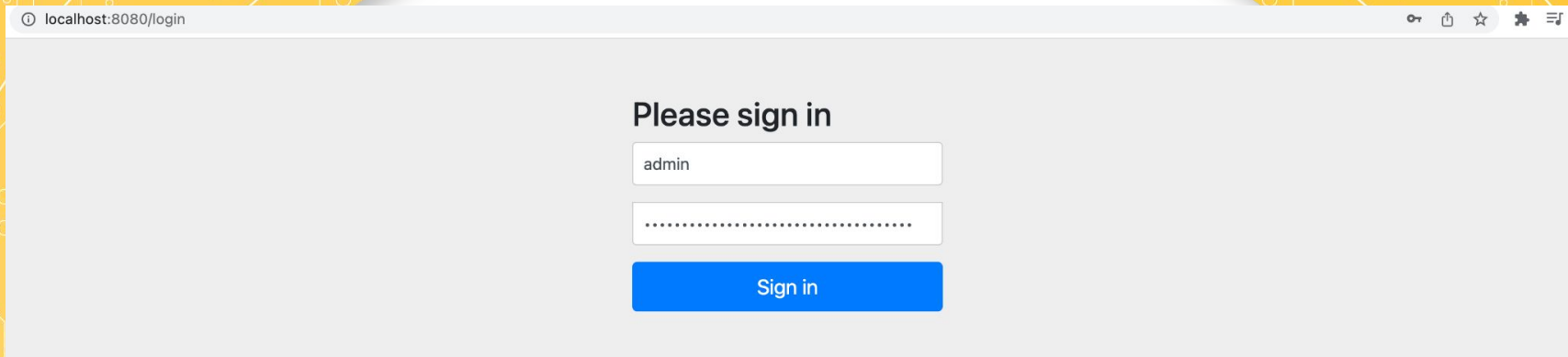
Console Actuator

Using generated security password: d651e626-0b1b-47fc-a9db-432c7b00eff4

This generated password is for development use only. Your security configuration must be updated before

- Mặc định khi chưa ghi đè lại các phương thức chứng thực của **Spring Security**, ứng dụng sẽ tự tạo ra một mật khẩu mặc định và hiển thị ở màn hình console

Tạo dự án với Spring Security



localhost:8080/login

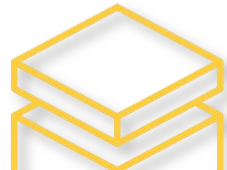
Please sign in

admin

.....

Sign in

- Đăng nhập với tài khoản mặc định là **user** và mật khẩu đã được tạo trước đó.



Làm sao có thể
chứng thực kết
hợp với database
?

Làm sao để có thể
phân quyền truy
cập chức năng
hoặc link ?

Nghe nói có tích
hợp được JWT
nữa rồi phải làm
sao ?



Tạo tài khoản chứng thực dùng memory



Bước 1 : Tạo class `UserController.java` định nghĩa link với `"/user"`.

Bước 2 : Tạo class `AdminController.java` định nghĩa link với `"/admin"`.

Bước 3 : Tạo class `HelloController.java` định nghĩa link với `"/hello"`.

Bước 4 : Tạo class `SecurityConfig.java` để cấu hình bảo mật.

Bước 5 : Khởi tạo `@Bean` khai báo chuẩn password sử dụng cho Spring Security

Bước 6 : Khởi tạo danh sách user lưu trữ trên memory

Bước 7 : Cấu hình `Authentication` và Authorization



Tạo Class UserController.java

Bước 1:

Tạo class `UserController.java` định nghĩa link với `"/user"` với `@RestController` và `@RequestMapping()`.

`@GetMapping` định nghĩa `"/user"` với phương thức GET



```
@RestController
@RequestMapping("/user")
public class UserController {

    @GetMapping("/")
    public String index(){
        return "Hello this is the user page";
    }

}
```


Tạo class AdminController

Bước 2 :

Tạo class `AdminController.java` định nghĩa link với `"/admin"` với `@RestController` và `@RequestMapping()`.

`@GetMapping` định nghĩa `"/admin"` với phương thức `GET`.



```
@RestController
@RequestMapping("/admin")
public class AdminController {

    @GetMapping("")
    public String index(){
        return "Hello this is the admin page";
    }

}
```

Tạo class HelloController

Bước 3 :

Tạo class `HelloController.java` định nghĩa link với `"/hello"` với `@RestController` và `@RequestMapping[]`.

`@GetMapping` định nghĩa `"/hello"` với phương thức GET.



```
@RestController
@RequestMapping("/hello")
public class AdminController {

    @GetMapping("")
    public String index(){
        return "Hello this is the admin page";
    }

}
```

Tạo class SecurityConfig.Java

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

}
```

- `@Configuration` : Đây là Annotation sẽ được Spring Boot đọc và cấu hình khi chạy ở tầng config.
- `@EnableWebSecurity` : Khai báo cho Spring Security biết đây là class sẽ custom rule của Security



Tạo class SecurityConfig.Java

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {
    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```

- `@Bean` : Giúp tạo class và đưa lên IOC dùng chung
- Ở đây chúng ta tạo ra một hàm tên là `passwordEncoder` trả ra kiểu dữ liệu là `PasswordEncoder`
- Trong hàm này sẽ tạo ra `BCryptPasswordEncoder`
- `BCryptPasswordEncoder` là chuẩn mã hóa không thể giải mã được chỉ có thể so sánh là có giống nhau hay không

Tạo class SecurityConfig.Java

```
@Bean
public UserDetailsService
userDetailsService(PasswordEncoder passwordEncoder) {
    UserDetails admin = User.withUsername("cybersoft")
        .password(passwordEncoder().encode("123456"))
        .roles("ADMIN")
        .build();

    UserDetails user = User.withUsername("user")
        .password(passwordEncoder().encode("user123456"))
        .roles("USER")
        .build();
    return new InMemoryUserDetailsManager(admin, user);
}
```

- `@Bean`
`public UserDetailsService`
`userDetailsService(PasswordEncoder`
`passwordEncoder)` : Khai báo hàm
`userDetailService` và đưa lên IOC dùng chung và
sử dụng `PasswordEncoder` đã được khai báo và
đưa lên IOC như một tham số
- `UserDetails user = User.withUsername("user")`
`.password(passwordEncoder().encode("user1234`
`56"))`
`.roles("USER")`
`.build();` : Tạo ra tài khoản đăng nhập cho security,
tài khoản user nhập vào sẽ được đối chiếu với tài
khoản này nếu tồn tại thì sẽ chứng thực thành
công và ngược lại.
- `new InMemoryUserDetailsManager(admin, user);`
: Tài khoản được tạo ra sẽ lưu trên RAM

Tạo class SecurityConfig.Java

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http)
throws Exception{
    return http.csrf().disable()
        .authorizeHttpRequests()
        .requestMatchers("/hello").permitAll()
        .requestMatchers("/admin").hasRole("ADMIN")
        .requestMatchers("/user").hasAnyRole("ADMIN", "USER")
        .anyRequest().authenticated()
        .and().build();
}
```

- **SecurityFilterChain** : Là một class dùng để config rule liên quan tới bảo mật cho các link của web.
- **csrf** (Cross-site Request Forgery) : Là một hình thức tấn công của hacker dùng để giả mạo request của người dùng.
- **authorizeHttpRequests** : Cho phép quy định rule chứng thực cho web.
- **requestMatchers** : Quy định link sẽ được áp dụng rule chứng thực. **"/**"** tất cả các link.
- **permitAll** : Cho phép truy cập vào link mà ko cần chứng thực.
- **hasAnyRole, hasRole** : Quy định quyền để truy cập được vào link
- **anyRequest** : Toàn bộ request của ứng dụng
- **authenticated** : bắt buộc chứng thực đối với request.

Như vậy chúng ta vừa mới cấu hình xong security dùng với tài khoản được lưu trong memory

Vậy bây giờ làm sao có thể chuyển sang dữ liệu đăng nhập ở database ?



Security và Database

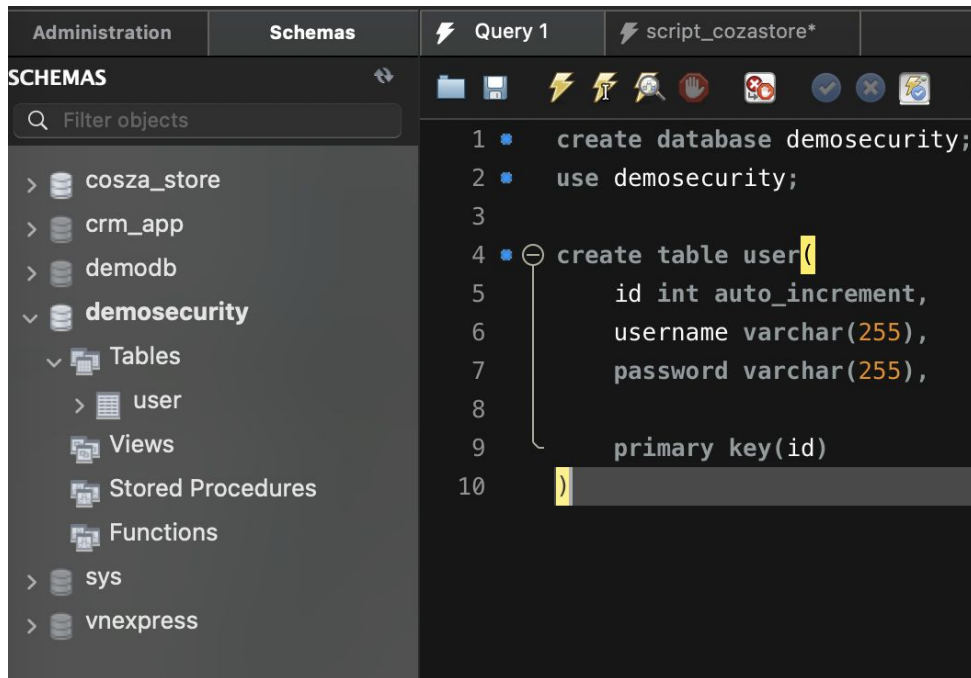
Bước 1: Bổ sung thư viện JPA và MySQLConnector vào project. Thêm vào file pom.xml

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

<dependency>
  <groupId>com.mysql</groupId>
  <artifactId>mysql-connector-j</artifactId>
  <scope>runtime</scope>
</dependency>
```


Security và Database

Bước 2 : Tạo database demosecurity và tạo bảng user bằng script như hình bên ở mysql tool.



Security và Database

Bước 3 : Khai báo thông tin cấu hình kết nối tới csdl Mysql

Spring.datasource.url : Định nghĩa đường dẫn kết nối tới csdl mysql và database cần sử dụng.

Spring.datasource.username : Thông tin tài khoản đăng nhập vào csdl.

Spring.datasource.password : Thông tin mật khẩu đăng nhập vào csdl.

Spring.jpa.properties.hibernate.dialect : Loại truy vấn hỗ trợ giành cho JPA.

Spring.jpa.hibernate.ddl-auto : Quy định cho phép tạo bảng hoặc column thông qua code trong entity

logging.level.org.hibernate : Xuất thông tin log

```
server.port=8080
spring.datasource.url=jdbc:mysql://localhost:3307/demos
security
spring.datasource.username=root
spring.datasource.password=admin123

spring.jpa.properties.hibernate.dialect=
org.hibernate.dialect.MySQL8Dialect
spring.jpa.hibernate.ddl-auto= none
logging.level.org.hibernate= ERROR
```

Security và Database

Bước 4 : Tạo class UserEntity để mapping với bảng User trong Database cho JPA.

```
@Entity(name = "user")
public class UserEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    @Column(name = "username")
    private String username;

    @Column(name = "password")
    private String password;
}
```

Chỉnh sửa lại class SecurityConfig.java

Bước 5 :Chỉnh sửa lại Bean

authenticationManager nhằm thông báo cho AuthenticationManager biết sẽ sử dụng class **CustomerDetailsService** để xác thực tài khoản thay vì xài mặc định trong Memory như trước.

`httpSecurity.getSharedObject(AuthenticationManagerBuilder.class)` : Lấy

AuthenticationManagerBuilder của Security để chỉ định lại các class chứng thực hoặc mọi thứ liên quan tới chứng thực muốn custom.

```
@Bean
public AuthenticationManager authenticationManager(HttpSecurity
httpSecurity) throws Exception {
    CustomUserDetailsService customUserDetailsService = new
CustomUserDetailsService();
    return
httpSecurity.getSharedObject(AuthenticationManagerBuilder.class)
        .userDetailsService(customUserDetailsService)
        .passwordEncoder(passwordEncoder())
        .and().build();
}
```

Tạo interface UserRepository.java

Bước 6: Khởi tạo interface

UserRepository.java đây là class sẽ đảm nhận các câu query liên quan tới bảng user.

findByUsername : Đây là hàm khai báo theo cấu trúc của Query Criteria của JPA dùng để lấy User theo username;

```
@Repository
public interface UserRepository extends
    JpaRepository<UserEntity,Integer> {
    UserEntity findByUsername(String username);
}
```

Tạo interface CustomUserService.java

Bước 7:

loadUserByUsername : Hàm này sẽ được AuthenticationManager sử dụng và load khi user tiến hành chứng thực ở bước 5.

UserEntity userEntity =

userRepository.findByUsername(username) :

Dùng để tìm thông tin user trong database với tham số là username do người dùng truyền vào.

User user1 = new

User(userEntity.getUsername(),userEntity.get

Password(), new ArrayList<>()) : Tạo ra thông tin chứng thực cho AuthenticationManager sử dụng để so sánh và kiểm tra chứng thực.

```
@Service
public class CustomUserService implements
UserDetailsService {
    @Autowired
    UserRepository userRepository;
    @Override
    public UserDetails loadUserByUsername(String
username) throws UsernameNotFoundException {
        UserEntity userEntity =
userRepository.findByUsername(username);
        if(userEntity == null){
            throw new UsernameNotFoundException("Không
            tìm thấy thông tin user");
        }else {
            User user1 = new
            User(userEntity.getUsername(),userEntity.getPassword(),
            new ArrayList<>());
            return user1;
        }
    }
}
```

Như vậy chúng ta vừa mới cấu hình xong security kết hợp với kết nối database

Nhưng tính bảo mật của hệ thống chưa được đảm bảo và vẫn còn lưu trữ session. Tiếp theo chúng ta sẽ sử dụng một thủ thuật khác tên là JWT để cải thiện bảo mật và giao tiếp database

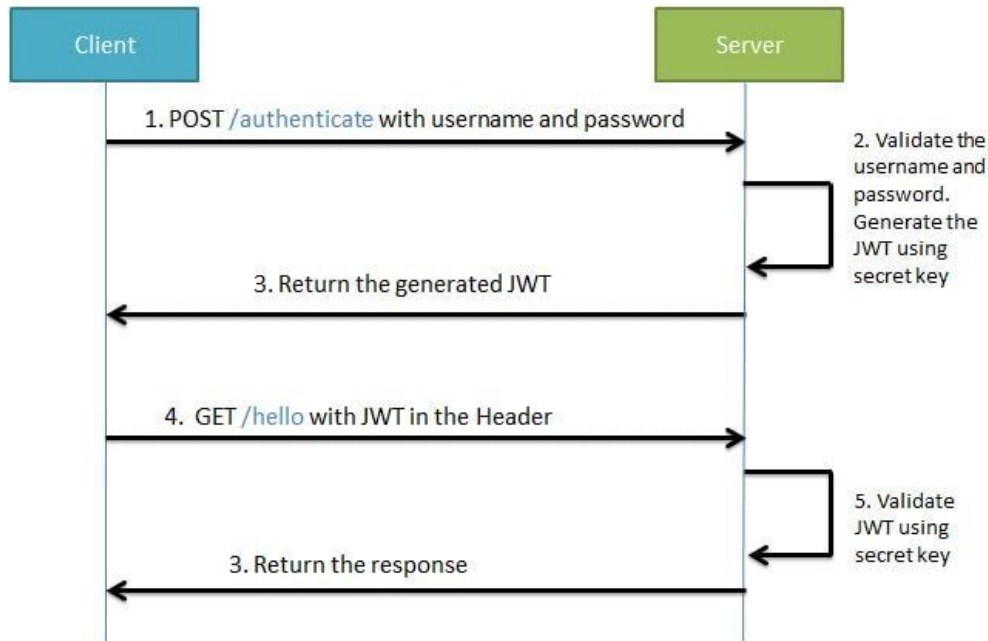


JWT (JSON Web Token)

- **Json Web Token (JWT)** là một chuẩn để truyền tải thông tin một cách an toàn giữa các bên bằng một đối tượng Json.
- **Json Web Token** có kích thước nhỏ gọn do đó nó có thể được gửi qua Url, tham số POST hoặc bên trong tiêu đề HTTP.
- **Json Web Token** thường được dùng để xác thực người dùng [authentication], chuỗi JWT sẽ được gửi kèm trong phần header của request và server sẽ thông qua token đó để xác thực request.



JWT - Cách Thức Hoạt Động



- **Bước 1:** Client sẽ gọi link chứng thực gửi username và password về server
- **Bước 2:** Server sẽ xác thực username và password người dùng có hợp lệ hay không. Nếu hợp lệ sẽ tạo ra một JWT Token với secret key.
- **Bước 3:** Trả Token đã tạo cho client
- **Bước 4:** Client sẽ gọi một link bất kỳ và truyền token đã nhận được thông qua header để lấy dữ liệu
- **Bước 5:** Server sẽ kiểm tra Token có hợp lệ hay không bằng secret key đã tạo token trước đó nếu hợp lệ sẽ trả dữ liệu cho client.

JWT - Cấu Trúc

Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFpbCI6ImN5YmVyc29mdEBnbWFpbC5jb20iLCJ1YW1lIjoiaQ3liZXJzb2Z0In0.2AD4yKbuC_u-TZ8MJJhIm2a4TMeM116LE4vHmObWFZk
```

Chuỗi token
sau khi mã hóa

Chuỗi token có dạng: **header.payload.signature**

Signature được tạo ra bằng cách mã hóa **header** và **payload** bằng thuật toán **base64UrlEncode** sau đó mã hóa 2 chuỗi trên kèm theo **Secret** bằng thuật toán **HS256**.

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{  "alg": "HS256",  "typ": "JWT"}
```

Thuật toán mã hóa

Loại định dạng token

PAYLOAD: DATA

```
{  "email": "cybersoft@gmail.com",  "name": "Cybersoft"}
```

Payload chứa thông tin muốn đặt trong token như email, fullname, avatar

VERIFY SIGNATURE

```
HMACSHA256(  base64UrlEncode(header) + "." +  base64UrlEncode(payload),  cyber  ) ☐ secret base64 encoded
```

Khóa bí mật (secret)

JWT - Cấu trúc

- **Header** : bao gồm hai phần chính: Loại token (mặc định là JWT). Thuật toán đã dùng để mã hóa (**HMAC SHA256 - HS256** hoặc **RSA**).
- **Payload**: Chứa **claims** (dữ liệu mà chúng ta muốn truyền đi như **username**, **email**, **fullname**,...), chứa các thông tin như **subject** (chủ đề), **issuer** (tổ chức phát hành token), **expired time** (ngày hết hạn).
- **Signature**: Là một chuỗi được mã hóa bởi header, payload cùng với một chuỗi bí mật (secret) theo nguyên tắc sau:

```
HMACSHA256(  
    base64UrlEncode(header) + "." +  
    base64UrlEncode(payload),  
    secret)
```

Thông tin payload

- **iss (issuer):** tổ chức phát hành token
- **sub (subject):** chủ đề của token
- **aud (audience):** đối tượng sử dụng token
- **exp (expired time):** thời điểm token sẽ hết hạn
- **nbf (not before time):** token sẽ chưa hợp lệ trước thời điểm này
- **iat (issued at):** thời điểm token được phát hành, tính theo UNIX time
- **jti:** JWT ID



Thư viện sử dụng

- <https://github.com/jwt/jjwt>
-

```
<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt-api</artifactId>
  <version>0.11.5</version>
</dependency>
<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt-impl</artifactId>
  <version>0.11.5</version>
  <scope>runtime</scope>
</dependency>
<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt-jackson</artifactId> <!-- or jjwt-gson if Gson is preferred -->
  <version>0.11.5</version>
  <scope>runtime</scope>
</dependency>
```

Thực hành chứng thực với JWT

- **Bước 1:** Tạo file **LoginController** và định nghĩa đường dẫn **/signin** và nhận vào 2 tham số là username và password.
- **Bước 2:** Import thư viện jjwt vào pom.xml
- **Bước 3:** tại hàm xử lý cho link **/signin** kiểm tra username và password xem có hợp lệ không nếu hợp lệ thì trả ra token

Lưu ý: Ở đây chúng ta sẽ không sử dụng **UserDetailsService** nữa mà sẽ custom chứng thực bằng cách custom lại **AuthenticationManager**.



Thực hành chứng thực với JWT

Bước 1: Tạo class

CustomerAuthProvider và kế thừa **AuthenticationProvider** để custom lại logic đăng nhập của Security

- **authenticate()**: Là phương thức giúp custom lại logic chứng thực
- **supports()**: là phương thức khai báo loại chứng thực cho Security sử dụng khi muốn so sánh.

```
@Component
public class CustomerAuthProvider implements AuthenticationProvider {
    1 usage
    @Autowired
    private UserRepository userRepository;
    1 usage
    @Autowired
    @Lazy
    private PasswordEncoder passwordEncoder;

    @Override
    public Authentication authenticate(Authentication authentication) throws AuthenticationException {
        String username = authentication.getName();
        String password = authentication.getCredentials().toString();
        UserEntity user = userRepository.findByUsername(username);

        if(user != null && passwordEncoder.matches(password, user.getPassword())){
            return new UsernamePasswordAuthenticationToken(
                user.getUsername(), user.getPassword(), new ArrayList<>());
        }

        return null;
    }

    @Override
    public boolean supports(Class<?> authentication) {
        return authentication.equals(UsernamePasswordAuthenticationToken.class);
    }
}
```

Thực hành chứng thực với JWT

- **Bước 2** : Ở file `SecurityConfig.java` chỉnh sửa lại Bean `AuthenticationManager` sẽ sử dụng `CustomerAuthenProvider` mới khai báo.

```
@Autowired
CustomerAuthenProvider customerAuthenProvider;

/**
 * Khai báo dạng mã hóa giành cho password
 */
@Bean
public PasswordEncoder passwordEncoder(){
    return new BCryptPasswordEncoder();
}

// Tạo ra AuthenticationManager để custom lại thông tin chứng thực
@Bean
public AuthenticationManager authenticationManager(HttpSecurity httpSecurity) throws
    return httpSecurity.getSharedObject(AuthenticationManagerBuilder.class)
        .authenticationProvider(customerAuthenProvider)
        .build();
}
```


Thực hành chứng thực với JWT

- **Bước 3** : Tạo file LoginController và định nghĩa đường dẫn `/login/signin` và tiến hành chứng thực

```
@RestController
@RequestMapping("/login")
public class LoginController {
    usage
    Autowired
    private AuthenticationManager authenticationManager;

    @PostMapping("/signin")
    public ResponseEntity<?> signin(
        @RequestParam String username,
        @RequestParam String password
    ){
        authenticationManager.authenticate(
            new UsernamePasswordAuthenticationToken(username,password)
        );
        return new ResponseEntity<> ( body: "", HttpStatus.OK);
    }
}
```

Thực hành chứng thực với JWT

- **Bước 4** : Tạo class **JwtUtilHeplers** để quản lý việc tạo token và giải mã token.
- Hàm **generateToken()** : xử lý việc tạo token với các thông tin **SecretKey** được sử dụng và các thông tin liên quan

```
@Component
public class JwtUtilsHeplers {

    @Value("${jwt.privateKey}")
    private String privateKey;

    private long expiredTime = 8 * 60 * 60 * 1000;

    public String generateToken(String data){
        SecretKey key = Keys.hmacShaKeyFor(Decoders.BASE64.decode(privateKey));
        // Calendar calendar = Calendar.getInstance();
        // calendar.getTimeInMillis();

        Date date = new Date();
        long currentDateMilis = date.getTime() + expiredTime;
        Date expiredDate = new Date(currentDateMilis);

        String jwt = Jwts.builder()
            .setSubject(data) //Dữ liệu muốn lưu kèm khi mã hóa JWT để sau này lấy ra xử dụng
            .signWith(key) //Key mã hóa
            .setExpiration(expiredDate)
            .compact();

        System.out.println("token : " + jwt);
        return jwt;
    }
}
```

Thực hành chứng thực với JWT

- **Bước 5** : Tại class `LoginController` bổ sung logic code trả ra token khi chứng thực thành công

```
@RestController
@RequestMapping("/login")
public class LoginController {
    1 usage
    @Autowired
    private AuthenticationManager authenticationManager;

    @PostMapping("/signin")
    public ResponseEntity<?> signin(
        @RequestParam String username,
        @RequestParam String password
    ){
        authenticationManager.authenticate(
            new UsernamePasswordAuthenticationToken(username,password)
        );

        return new ResponseEntity<>(jwtUtilsHepLers.generateToken(username), HttpStatus.OK);
    }
}
```