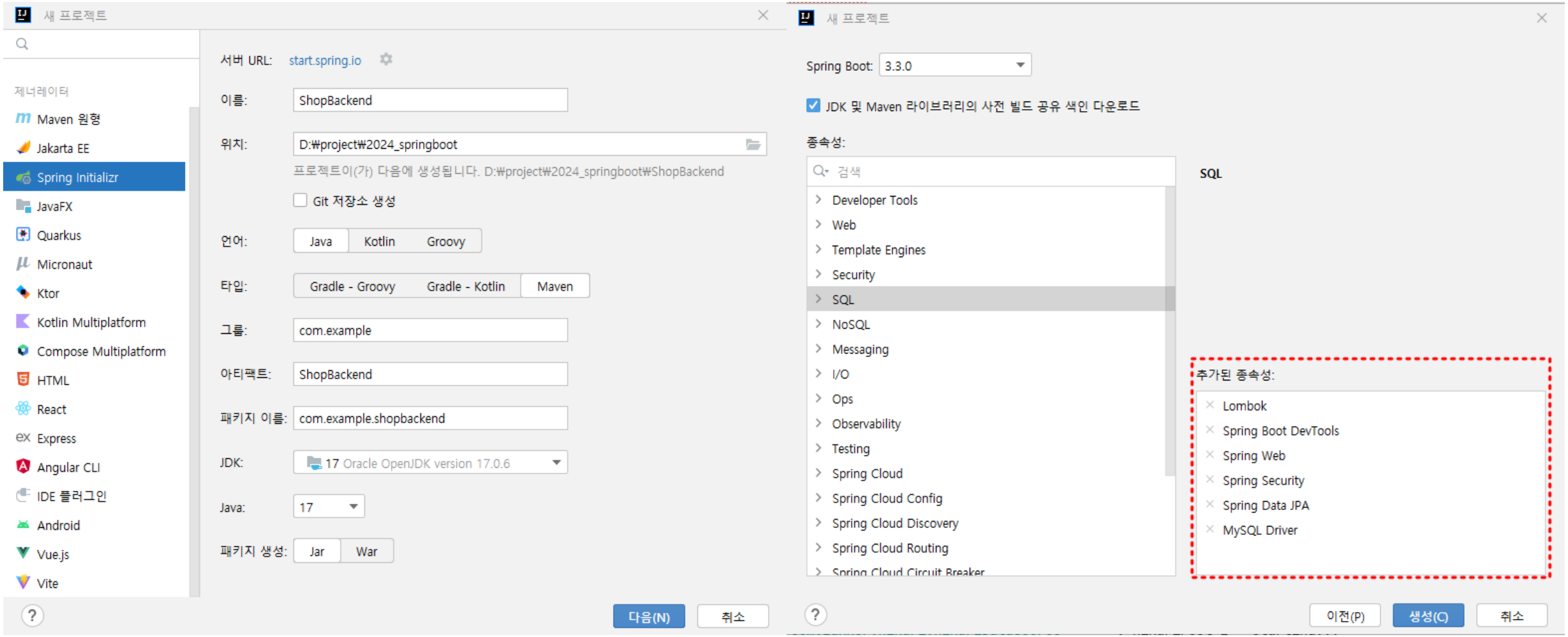
2. Security& JWT

content

- 프로젝트 생성
- Entity와 Repository 작성
- Service 작성
- Security
- JWT
- Controller

1. 프로젝트 생성


새 프로젝트
서버 URL: start.spring.io
제너레이터
Maven 원형
Jakarta EE
Spring Initializr
JavaFX
Quarkus
Micronaut
Ktor
Kotlin Multiplatform
Compose Multiplatform
HTML
React
Express
Angular CLI
IDE 플러그인
Android
Vue.js
Vite
이름: ShopBackend
위치: D:₩project₩2024_springboot
프로젝트이(가) 다음에 생성됩니다. D:₩project₩2024_springboot₩ShopBackend
Git 저장소 생성
언어: Java Kotlin Groovy
타입: Gradle - Groovy Gradle - Kotlin Maven
그룹: com.example
아티팩트: ShopBackend
패키지 이름: com.example.shopbackend
JDK: 17 Oracle OpenJDK version 17.0.6
Java: 17
패키지 생성: Jar War
다음(N)
취소
새 프로젝트
Spring Boot: 3.3.0
JDK 및 Maven 라이브러리의 사전 빌드 공유 색인 다운로드
종속성:
검색
Developer Tools
Web
Template Engines
Security
SQL
NoSQL
Messaging
I/O
Ops
Observability
Testing
Spring Cloud
Spring Cloud Config
Spring Cloud Discovery
Spring Cloud Routing
SQL
추가된 종속성:
Lombok
Spring Boot DevTools
Spring Web
Spring Security
Spring Data JPA
MySQL Driver
이전(P)
생성(C)
취소

1. 프로젝트 생성

- MySQL 워크벤치 새 유저 만들기
  - Sever > Users and Privileges
  - add Account로 새로운 유저 추가
  - Administrative Role 에서 권한 설정
  - 새 connection 추가

- 새 스키마 작성
  - 스키마 명 : shopdb
  - encoding 설정 : utf8

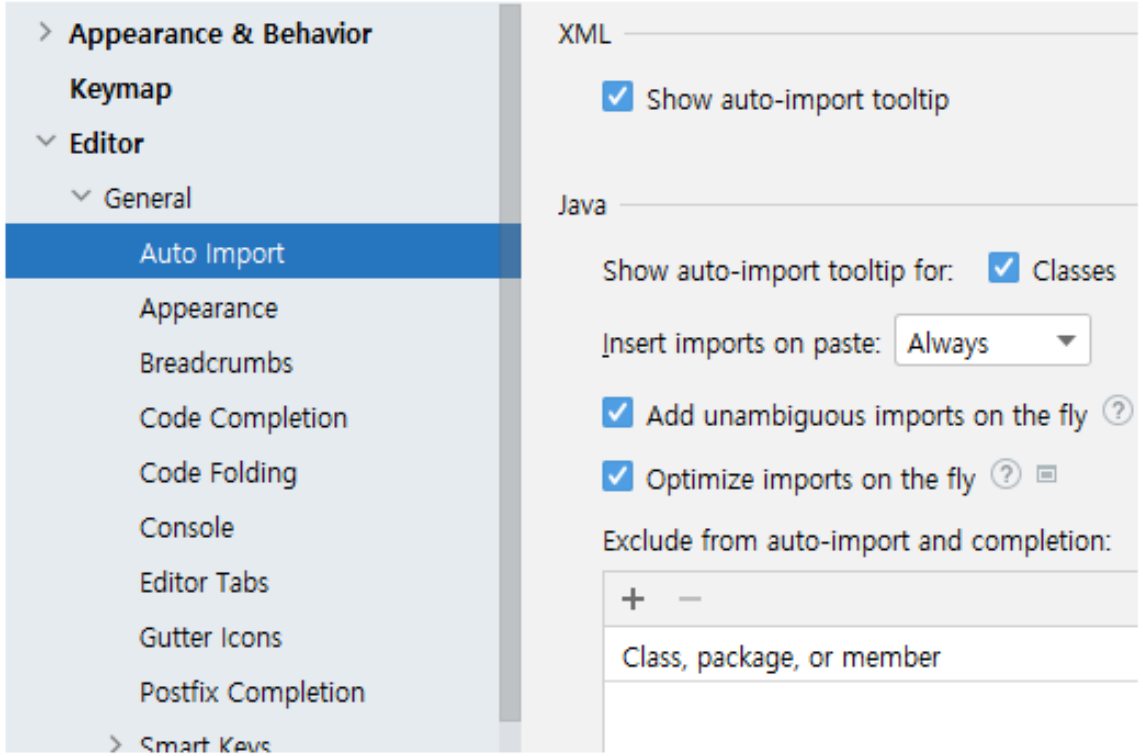
1. 프로젝트 생성

- application.properties 설정

```
server.port=8082
#server.servlet.context-path=/mysite

# MYSQL DB
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/shopdb
spring.datasource.username=pgm
spring.datasource.password=1234

logging.level.org.springframework=info
logging.level.org.zerock=debug
logging.level.org.springframework.security=trace

# JPA table ddl auto
spring.jpa.hibernate.ddl-auto=update

# Console sql show
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
```

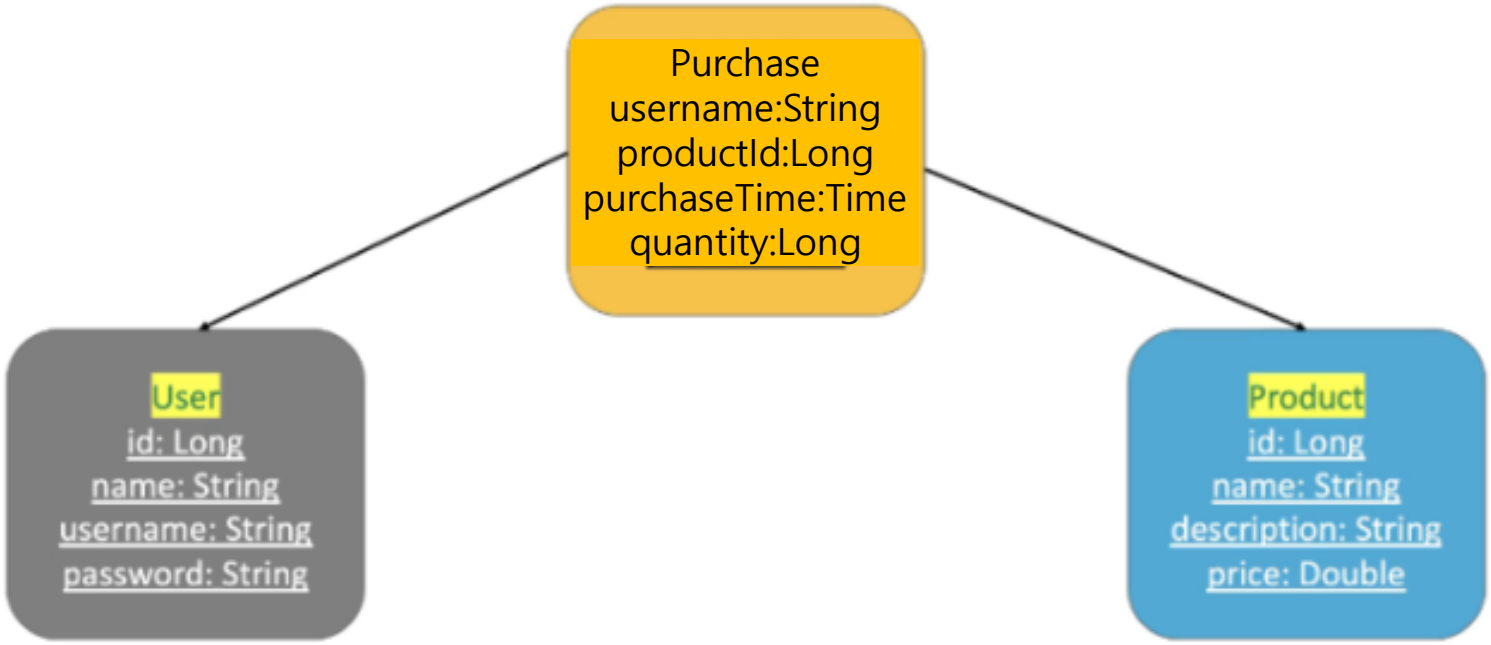
1. 프로젝트 생성

- 자동 import 설정
  - 파일 > 세팅(Ctrl + Alt + S)



2. Entity와 Repository 작성

- ER-Diagran



2. Entity와 Repository 작성

- User Entity
  - model 패키지에 작성

```java
@Data
@Entity
@Table(name="users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private  Long id;
    @Column(name = "username", unique = true, nullable = false, length = 100)
    private String username;

    @Column(name = "password", nullable = false)
    private String password;

    @Column(name = "name", nullable = false)
    private String name;

    @Column(name = "create_time", nullable = false)
    private LocalDateTime createTime;

    @Enumerated(EnumType.STRING)
    @Column(name = "role", nullable = false)
    private Role role;

    @Transient
    private String token;
}
```

2. Entity와 Repository 작성

- Role Enum
  - model 패키지에 작성

```
public enum Role {
    1개 사용 위치
    USER, ADMIN;
}
```

@Enumerated(EnumType.ORDINAL) // 데이터베이스에 순서 값이 저장
@Enumerated(EnumType.STRING)

```
ordinal
------------
ID | role |
------------
1 | 0    USER
2 | 1    ADMIN
3 | 0    USER
4 | 0    USER

string
------------
ID | role |
------------
1 | USER
2 | ADMIN
3 | USER
4 | USER
```

2. Entity와 Repository 작성

- Product Entity

```
@Data
@Entity
@Table(name="product")
public class Product {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "name", nullable = false)
    private String name;

    @Column(name = "description", nullable = false)
    private String description;

    @Column(name = "price", nullable = false)
    private Integer price;

    @CreationTimestamp
    @Column(name = "create_time", nullable = false)
    private LocalDateTime createTime;
}
```

2. Entity와 Repository 작성

- Purchase Entity

```
@Data
@Entity
@Table(name="purchase")
public class Purchase {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "user_id", referencedColumnName = "id", updatable = false)
     private User user;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "product_id", referencedColumnName = "id", updatable = false)
    private Product product;

    @Column(name = "quantity", nullable = false)
    private Integer quantity;

    @Column(name = "purchase_time", nullable = false)
    private LocalDateTime purchaseTime;
}
```

2. Entity와 Repository 작성

- **즉시 로딩(EAGER)과 지연 로딩(LAZY)**
  - Purchase를 조회할 때 User와 Product도 함께 조회 해야 할까?
  - 비즈니스 로직에서 단순히 멤버 로직만 사용하는데 함께 조회하면, 트래픽 증가
  - JPA는 이 문제를 지연로딩 LAZY를 사용해서 **필요시 조회**하는 방법으로 해결.

- insertable = false, updatable = false
  - 입력시 이 열은 제외, 업데이트시 이 열은 제외
  - 이 엔티티에서 외래키 부분(Product 와 User)을 Purchase를 입력하거나 업데이트 할 때 포함하지 않는다.

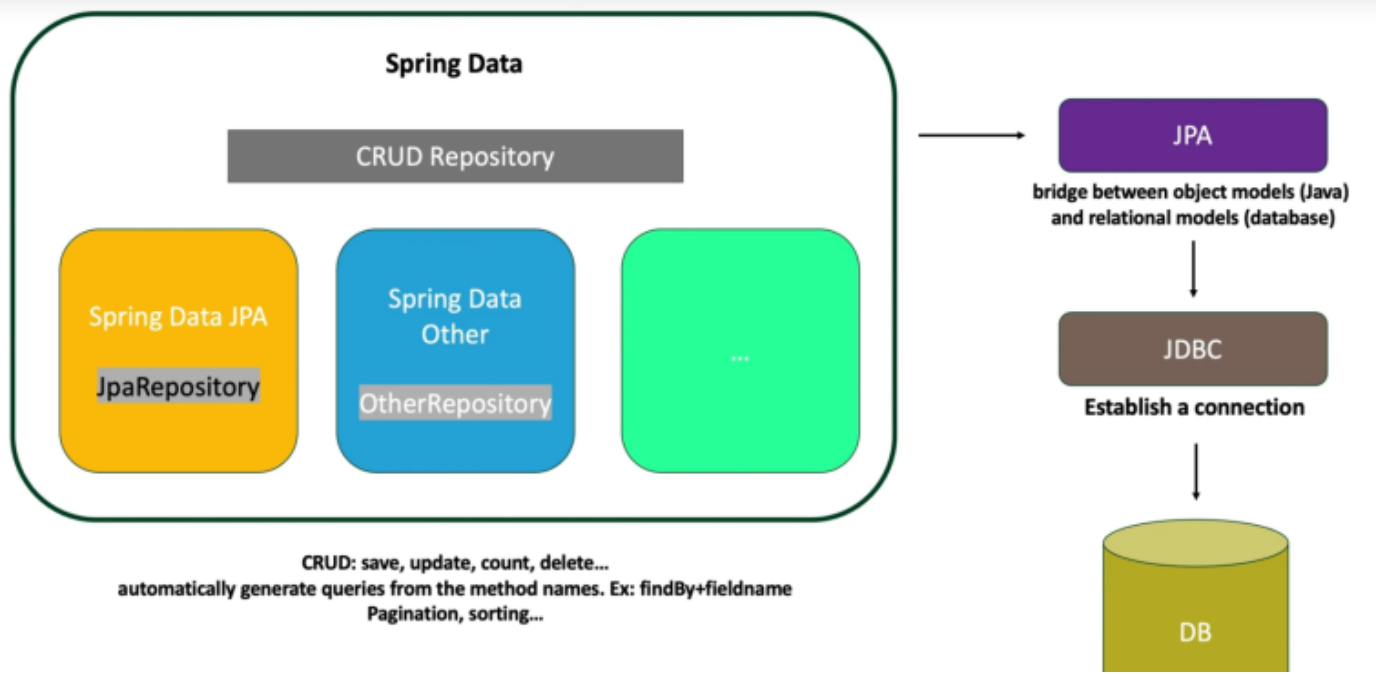
2. Entity와 Repository 작성

- UserRepository
  - repository 패키지에 작성

```
2개 사용 위치
public interface UserRepository extends JpaRepository<User, Long> {
    1개 사용 위치
    Optional<User> findByUsername(String username);

    1개 사용 위치
    @Modifying
    @Query("update User set role=:role where username=:username")
    void updateUserRole(@Param("username") String username, @Param("role") Role role);
}
```

2. Entity와 Repository 작성

- UserRepository
  - jpa 데이터 처리 방식



<Class> findBy + fieldName...

User findByName(String name);

User findByCreateTimeAfter(LocalDate threshold)

2. Entity와 Repository 작성

- UserRepository
  - **JPQL이란?**
    - JPQL은 엔티티 객체를 조회하는 객체지향 쿼리, 즉 테이블을 대상으로 쿼리하는 것이 아니라 엔티티 객체를 대상으로 쿼리
    - 문법은 SQL과 유사하며 간결하다
    - JPQL은 결국 SQL로 변환된다.
  - **JPA 방식이 아닌 JPQL 로 쿼리를 작성 예**
    - **Select Query**

```
@Query("Select c from Class c where c.fieldName=:param")
<Class> findByQueryAnnotation(@Param String params...);
```

2. Entity와 Repository 작성

- UserRepository
  - INSERT UPDATE DELETE 쿼리작성시 @Modifying을 붙임

```
@Modifying  // update or delete 쿼리에 추가하는 어노테이션
@Query("Update or Delete ... where c.fieldName=:param")
<Class> updateOrDelete(@Param String parames...)
```

  - **JPQL주의점.**
    - **대소문자 구분 :** 엔티티와 속성은 대소문자를 구분. (단 SELECT, FROM, WHERE 같은 JPQL 키워드는 대소문자를 구분하지 않는다).
    - **엔티티 이름:** JPQL에서 사용한 Member는 테이블명이 아니라 엔티티 클래스 이름이다.
    - **이름 파라미터 (: 속성명) :** 파라미터를 이름으로 구분할 때 객체명.이름 =: 파라미터 을 사용.

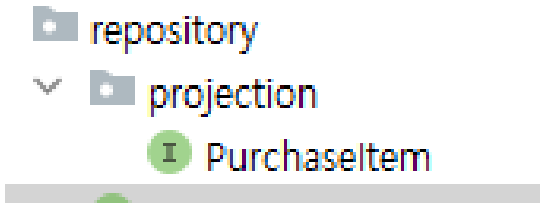
2. Entity와 Repository 작성

- ProductRepository

```
public interface ProductRepository extends JpaRepository<Product, Long> {

}
```

2. Entity와 Repository 작성

PurchaseItem 작성




```
public interface PurchaseItem {
    String getName();
    Integer getQuantity();
    LocalDateTime getPurchaseTime();
}
```

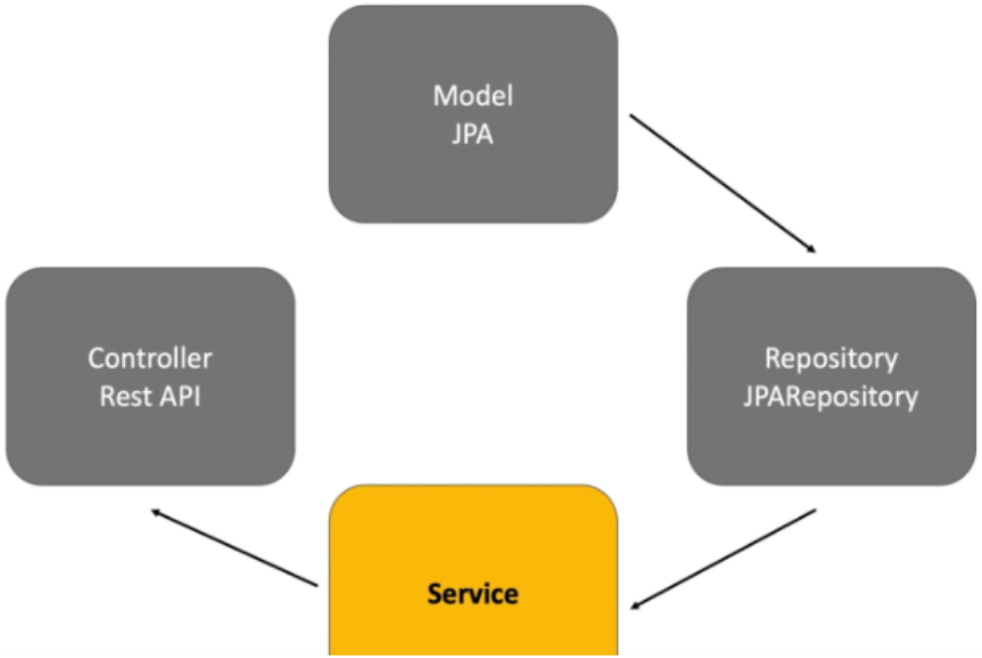
프로젝션 (Projection)이란 SELECT 절에서 조회할 대상을 지정하는 것으로
여기서 제품의 이름, 수량, 구매시간만을 가져오는 프로젝션을 작성.

2. Entity와 Repository 작성

- **PurchaseRepository**

```
public interface PurchaseRepository extends JpaRepository<Purchase, Long> {
    @Query("select " +
            "prd.name as name, pur.quantity as quantity, pur.purchaseTime as purchaseTime " +
            "from Purchase pur left join Product prd on prd.id = pur.product.id " +
            "where pur.user.username = :username")
    List<PurchaseItem> findAllPurchasesOfUser(@Param("username") String username);
}
```

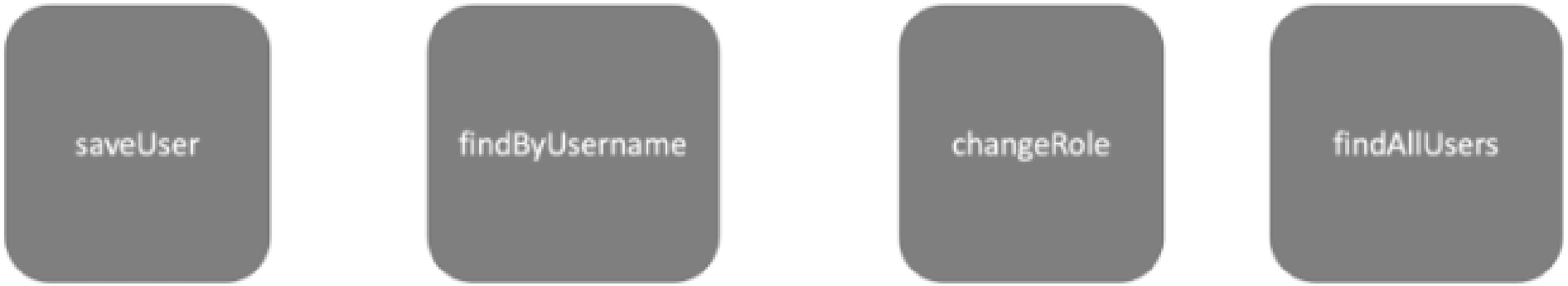
3. Service 작성



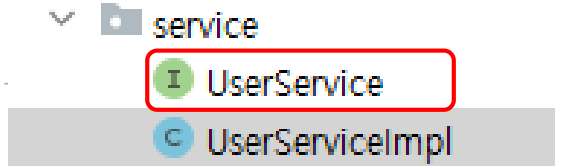
1. 모델 엔티티 설계 (테이블)
2. 리포지토리 작성 (DAO)
3. 서비스 작성(비즈니스 로직)
4. 컨트롤러에서 서비스 호출

3. Service 작성

- User 서비스
  - 유저 서비스에 다음과 같은 로직이 필요



3. Service 작성

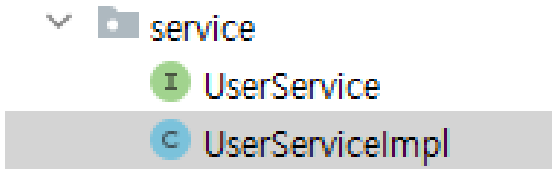


- User 서비스

```
public interface UserService {
    User saveUser(User user);
    User findByUsername(String username);
    void changeRole(Role newRole, String username);
    List<User> findAllUsers();
}
```

3. Service 작성

- User 서비스




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

패스워드 인코딩을 위한 빈을 메인 클래스에 빈 등록

```
@Service
@RequiredArgsConstructor
public class UserServiceImpl implements UserService{
    private final UserRepository userRepository;
    private final PasswordEncoder passwordEncoder;

    public User saveUser(User user){
        user.setPassword(passwordEncoder.encode(user.getPassword()));
        user.setRole(Role.USER);
        user.setCreateTime(LocalDateTime.now());
        return userRepository.save(user);
    }

    @Override
    @Transactional
    public void changeRole(Role newRole, String username) {
        userRepository.updateUserRole(username, newRole);
    }

    @Override
    public List<User> findAllUsers() {
        List<User> users=userRepository.findAll();
        return users;
    }
}
```

3. Service 작성

- ProductService

```
public interface ProductService {
    Long saveProduct(ProductDTO productDTO);
    void deleteProduct(Long id);
    List<ProductDTO> findAllProducts();

    default Product dtoToEntify(ProductDTO productDTO){
        Product product=Product.builder()
                .id(productDTO.getId())
                .name(productDTO.getName())
                .price(productDTO.getPrice())
                .description(productDTO.getDescription())
                .build();
        return  product;
    }
    default ProductDTO entityToDto(Product product){
        ProductDTO productDTO=ProductDTO.builder()
                .id(product.getId())
                .name(product.getName())
                .price(product.getPrice())
                .description(product.getDescription())
                .createTime(product.getCreateTime())
                .build();
        return productDTO;
    }
}
```

3. Service 작성

- ProductServiceImpl

```
@Service
@RequiredArgsConstructor
public class ProductServiceImpl implements ProductService{
    private final ProductRepository productRepository;

    @Override
    public Long saveProduct(ProductDTO productDTO) {
        Product product=dtoToEntify(productDTO);
        product.setCreateTime(LocalDateTime.now());
        return productRepository.save(product).getId();
    }

    @Override
    public void deleteProduct(Long id) {
        productRepository.deleteById(id);
    }
    @Override
    public List<ProductDTO> findAllProducts() {
        List<Product> productS=productRepository.findAll();
        List<ProductDTO> productDTOS=productS.stream()
            .map(product ->entityToDto(product)).collect(Collectors.toList());
        return productDTOS;
    }
}
```

3. Service 작성

- PurchaseService

```
public interface PurchaseService {
    Purchase savePurchase(Purchase purchase);
    List<PurchaseItem> findPurchaseItemsOfUser(String username);

}
```

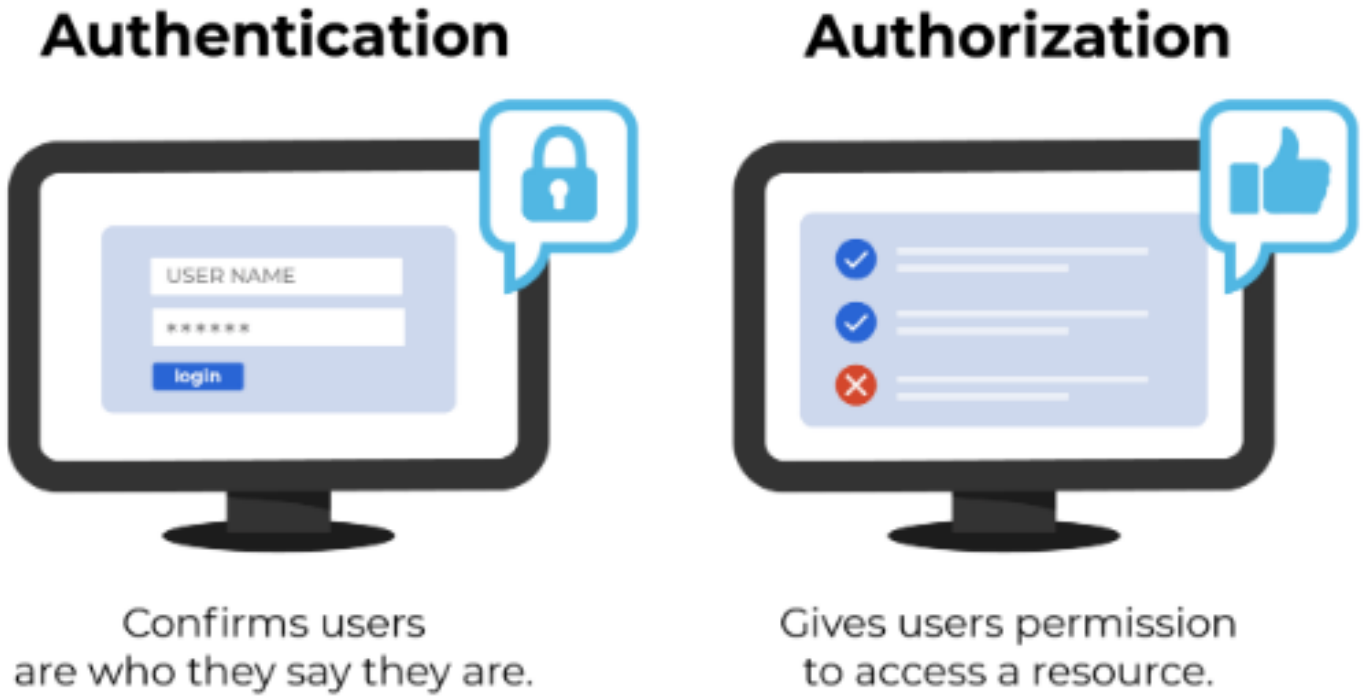
3. Service 작성

- PurchaseServiceImpl

```
@Service
@RequiredArgsConstructor
@Log4j2
public class PurchaseServiceImpl implements PurchaseService{
    private final PurchaseRepository purchaseRepository;
    private final UserRepository userRepository;
    private final ProductRepository productRepository;

    @Override
    public Purchase savePurchase(PurchaseDTO purchaseDTO) {
        Purchase purchase=Purchase.builder()
                .quantity(purchaseDTO.getQuantity())
                .build();
        User user=userRepository.findByUsername(purchaseDTO.getUsername()).orElseThrow();
        Product product=productRepository.findById(purchaseDTO.getProductId()).orElseThrow();
        purchase.setUser(user);
        purchase.setProduct(product);
        purchase.setPurchaseTime(LocalDateTime.now());

        return purchaseRepository.save(purchase);
    }
    @Override
    public List<PurchaseItem> findPurchaseItemsOfUser(String username) {
        log.info("service~~~~~~~~~~"+username);
        User user=userRepository.findByUsername(username).orElseThrow();

        return purchaseRepository.findAllPurchasesOfUser(username);
    }
}
```

4. Security와 JWT

- 인증(Authentication) vs 인가(Authorization)



4. Security

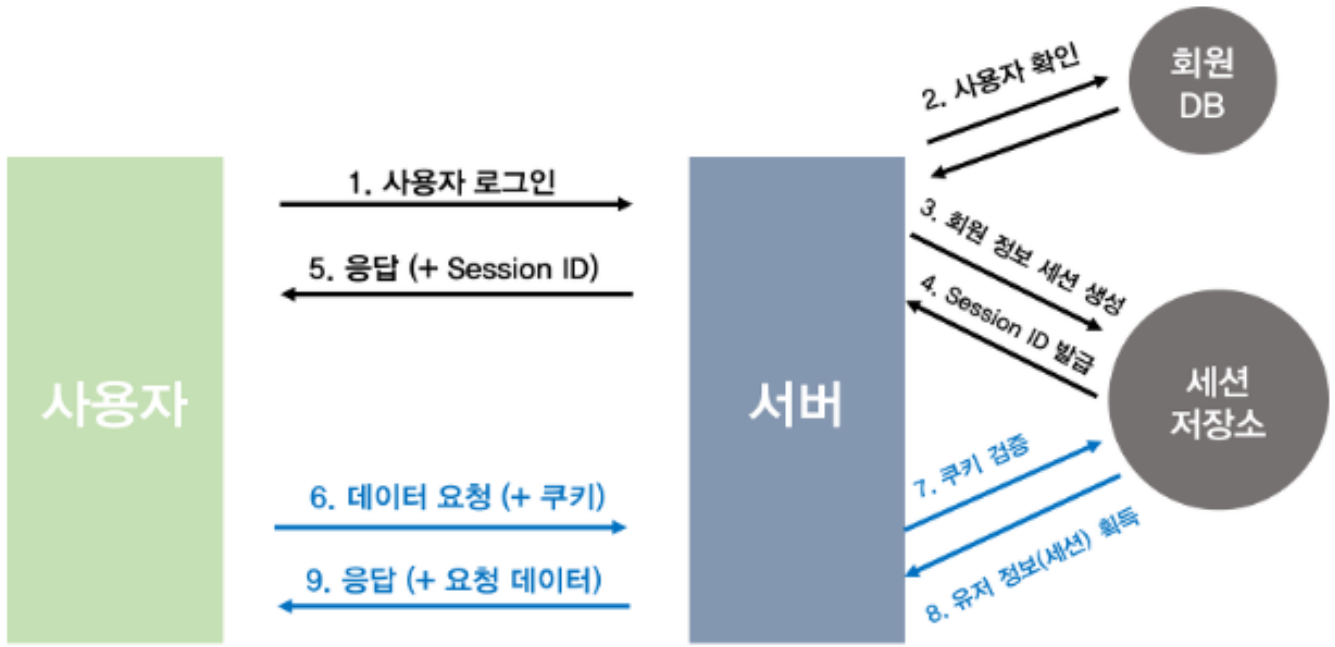
- **인증과 인가(허가)의 차이점**
  - **인증 :** 유저가 자원에 접근할 수 있는지를 확인을 받는 과정(예 : 로그인)
  - **인가 :** 유저가 자원에 접근할 때, 어떤 자원에 접근이 가능한지 확인하는 과정(예 : Security에서 사용자에 따른 접근 권한 설정)
  -
- **인증과 인가의 필요성**
  - Private API는 물론 Public API도 기본적인 인증과 인가가 요구된다.
  - 인증(Authentication)은 서비스를 누가 사용했는지 추적이 가능하므로, 제3의 타인에 대한 서비스 정보 노출을 막을 수 있다.
  - 인가(Authorozation)는 인증된 사용자에 대한 권한을 확인함으로써 A가 쓴 글을 B가 수정, 삭제 등을 할 수 없도록 막을 수 있다.

4. Security

- **인증방법**
  - **HTTP 헤더 요청 방식**
    - 인증을 받을 사용자 계정 정보를 http 요청에 담아 보내는 방식
    - 데이터를 요청할 때마다 사용자 정보를 계속해서 보내는 것이므로, 가장 보안이 낮은 인증 방식
    - 해커가 중간에 http 요청을 가로챈다면 사용자의 계정정보를 알아낼 수 있기 때문에 실제 서비스에서 거의 사용되지 않는 방식
  - **Session/ Cookie 방식**
    - 사용자의 계정 정보를 매번 모든 요청에 넣어서 보내는 것은 보안에도 취약하고 비효율적이므로 이를 보완하기 위해 나온 인증 방법
    - session/cookie 방식은 세션 저장소를 필요, 사용자가 로그인했을 때 사용자의 정보를 저장하고 열쇠가 되는 session id값을 만들어서 서버에서 관리를 하며, 웹 브라우저가 서버에 접속하여 종료할 때까지 인증 상태를 유지
    - 사용자 정보를 서버에 저장하므로 보안이 비교적 안전하지만, 사용자가 많아질수록 서버의 메모리를 많이 차지하여 서버에 과부하를 주어 성능 저하를 야기
  - **토큰 인증 방식**
    - session이 서버의 메모리를 차지하는 단점 때문에 이를 해결하기 위한 방법
    - token 방식을 사용하면 서버가 정보를 저장하지 않게 되어 사용자가 많아져도 서버의 과부하나 성능 저하가 생기지 않는다.

4. Security

- **Session/ Cookie 방식**



① 사용자가 로그인을 하면 서버에서 계정 정보를 확인한 후, 사용자에 고유한 id 값을 부여하여 세션 저장소에 저장

② 세션 저장소에 저장한 id 값에 연결되는 세션 id를 발행. 사용자는 서버에서 session id를 전달받아 쿠키에 저장.

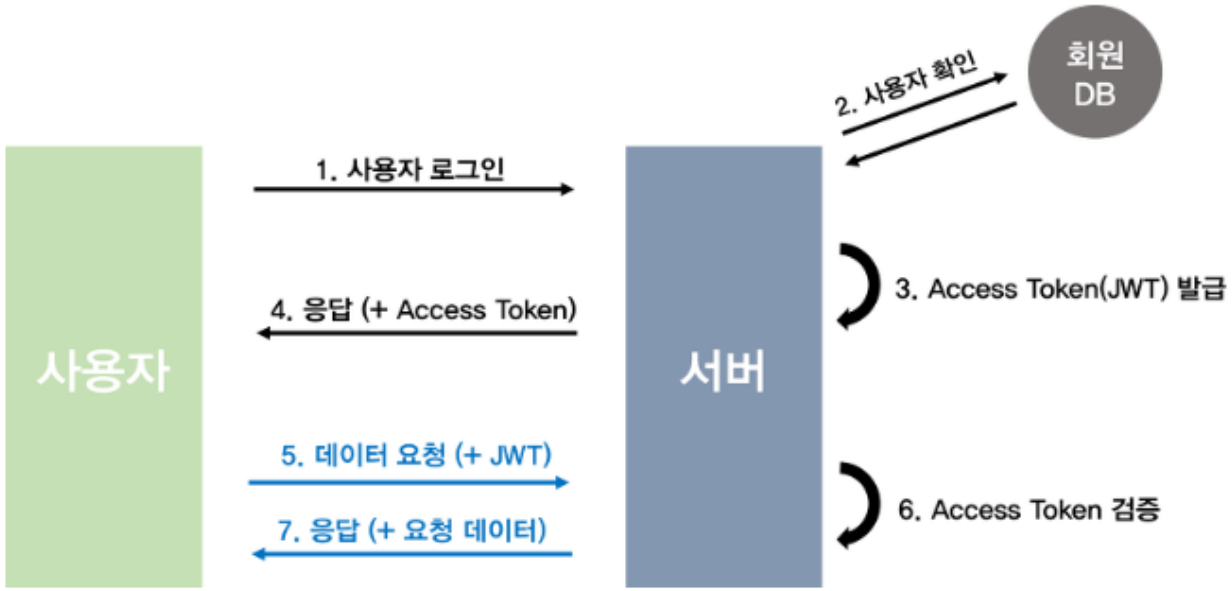
③ 이후 인증이 필요한 요청마다 쿠키를 http 헤더에 실어서 보냄.

④ 서버에서는 쿠키를 받아 session id를 세션 저장소에서 대조를 하여 검증한 후에 대응되는 유저 정보를 가져옴.

⑤ 인증이 완료되고 서버는 사용자에 맞는 데이터를 보냄.

4. Security

- 토큰 인증 방식



① 사용자가 로그인을 하면 서버에서 사용자를 확인하고, 사용자의 고유한 ID값을 부여한 후, 기타 정보와 함께 payload에 담는다.

② secret key를 이용하여 access token을 발급.

③ 인증된 사용자는 응답에서 액세스 토큰(Access Token)을 받아 저장.

④ 이 access token은 이후 http 요청마다 헤더에 담아서 전달하므로 서버는 요청한 사용자를 식별할 수 있다.

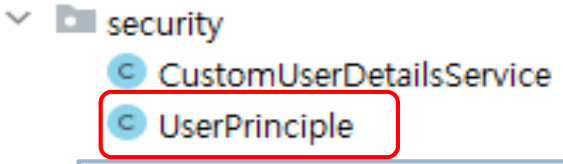
⑤ 서버에서는 해당 토큰의 Verify Signature을 Secret key로 복호화하고, 검증이 완료되면 payload 디코딩하여 사용자의 id에 맞는 데이터를 가져온다.

- 토큰은 그 자체로 유저 정보를 가지고 있으므로, 인증 정보를 서버가 저장하지 않아도 되어 애플리케이션을 확장할 수 있으며, stateless 서버를 만드는데 큰 강점이다
- 또한 한번 발급된 토큰은 탈취가 되어도 유효기간이 만료할 때까지 계속 사용이 가능
- 토큰이 탈취되어 악의적으로 이용된다면 토큰의 유효기간이 만료될때까지 기다리 것 말고는 할 수 있는 것이 없기 때문에 보안에 유의해야 한다.

4. Security

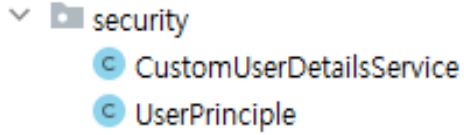
- UserDetailsService 와 UserDetails 구현

```
@Getter
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class UserPrinciple implements UserDetails {

    private Long id;
    private String username;
    transient private String password;
    transient private User user;
    private Set<GrantedAuthority> authorities;

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


    @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;
    }
    public Long getId(){
        return id;
    }
}
```

4. Security



- UserDetailsService 와 UserDetails 구현

```
@Service
@RequiredArgsConstructor
public class CustomUserDetailsService implements UserDetailsService {
    private final UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        User user=userRepository.findByUsername(username).orElseThrow(()->
                new UsernameNotFoundException(username));
        Set<GrantedAuthority> authorities= Set.of(SecurityUtils.convertToAuthority(user.getRole().name()));
        return null;
    }
}
```

4. Security

- UserDetailsService 와 UserDetails 구현
  - SecurityUtils


```
public class SecurityUtils {
    public static final String ROLE_PREFIX = "ROLE_";

    public static SimpleGrantedAuthority convertToAuthority(String role){
        String formatRole=role.startsWith(ROLE_PREFIX)? role: ROLE_PREFIX + role;
        return new SimpleGrantedAuthority(formatRole);
    }
}
```

4. Security

- **UserDetails 란**
  - Spring Security에서 사용자의 정보를 담는 인터페이스. 사용자 정보를 불러오기 위해서 구현

메소드	리턴 타입	설명	기본값
getAuthorities()	Collection<? extends GrantedAuthority>	계정의 권한 목록을 리턴	
getPassword()	String	계정의 비밀번호를 리턴	
getUsername()	String	계정의 고유한 값을 리턴 (ex : DB PK값, 중복이 없는 이메일 값)	
isAccountNonExpired()	boolean	계정의 만료 여부 리턴	true (만료 안됨)
isAccountNonLocked()	boolean	계정의 잠김 여부 리턴	true (잠기지 않음)
isCredentialsNonExpired()	boolean	비밀번호 만료 여부 리턴	true (만료 안됨)
isEnabled()	boolean	계정의 활성화 여부 리턴	true (활성화 됨)

4. Security

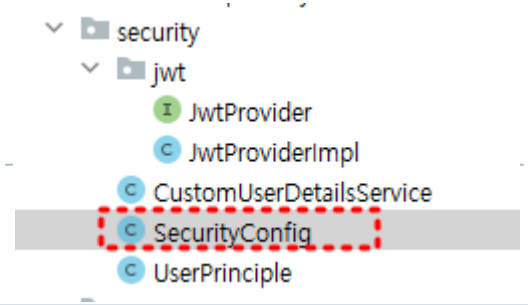
- **UserDetailsService 란?**
  - Spring Security에서 유저의 정보를 가져오는 인터페이스

메소드	리턴 타입	설명
loadUserByUsername	UserDetails	유저의 정보를 불러와서 UserDetails로 리턴

loadUserByUsername 메소드의 마지막 부분
return null 부분을 오른쪽과 같이 수정

```
return UserPrinciple.builder()
        .user(user)
        .username(user.getUsername())
        .password(user.getPassword())
        .authorities(authorities)
        .build();
```

4. Security



- @EnableWebSecurity 시큐리티 설정

```
@Configuration
@EnableWebSecurity
@RequiredArgsConstructor
public class SecurityConfig {
    private final CustomUserDetailsService userDetailsService;

    @Bean
    public AuthenticationManager authenticationManager(AuthenticationConfiguration authenticationConfiguration)
            throws Exception{
        return authenticationConfiguration.getAuthenticationManager();
    }
}
```

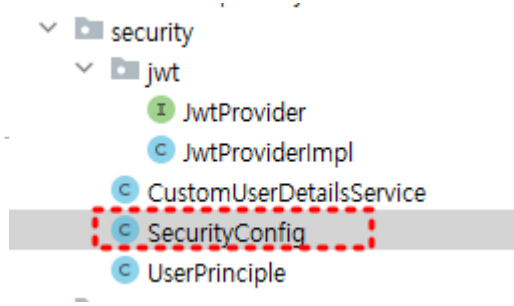
@Configuration은 스프링의 환경설정 파일을 의미하는 애너테이션

@EnableWebSecurity는 모든 요청 URL이 스프링 시큐리티의 제어를 받도록 만드는 애너테이션

AuthenticationManager : 스프링 시큐리티의 인증을 담당

AuthenticationManager 빈 생성시 **스프링의 내부 동작**으로 인해 위에서 작성한 customUserDetailsService와 PasswordEncoder가 자동으로 설정되어 유저가 로그인을 하면 유저이름으로 customUserDetailsService를 이용해 유저를 찾은 후 암호화된 패스워드와 유저가 입력한 패스워드를 비교(이때 패스워드 인코더 객체 필요) 하여 인증을 수행.

4. Security

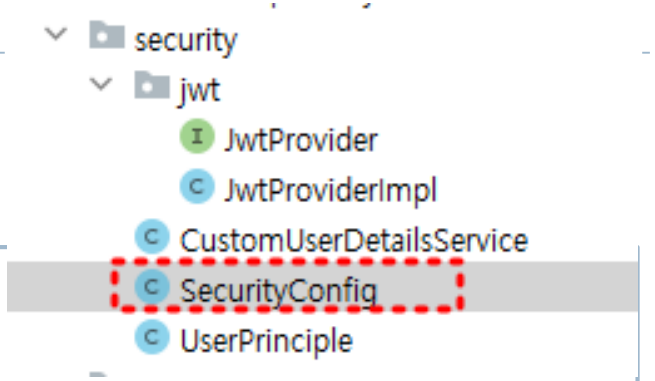


- SecurityFilterChain 빈
  - 스프링 시큐리티의 세부 설정

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception{
    http
            .csrf(AbstractHttpConfigurer::disable)
            .cors(httpSecurityCorsConfigurer -> corsConfigurationSource())
            .sessionManagement((session)->session
                    .sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .authorizeHttpRequests((authz)-> authz
                    .requestMatchers("/api/authentication/**").permitAll()
                    .requestMatchers(HttpMethod.GET, "/api/product/**").permitAll()
                    .requestMatchers("/api/product/**").hasRole(Role.ADMIN.name())
                    .anyRequest().authenticated()
            )
            .addFilterBefore(jwtAuthorizationFilter(), UsernamePasswordAuthenticationFilter.class);
    return http.build();
}
```

4. Security

- CORS 설정
  - 모든 요청시 Cross-Origin 을 허가




```
@Bean
CorsConfigurationSource corsConfigurationSource() {
    CorsConfiguration configuration = new CorsConfiguration();
    configuration.setAllowedOrigins(Arrays.asList("*"));
    configuration.setAllowedMethods(Arrays.asList("GET","POST","PUT","DELETE"));
    configuration.setAllowedHeaders(Arrays.asList("*"));
    UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
    source.registerCorsConfiguration("/**", configuration);
    return source;
}
```

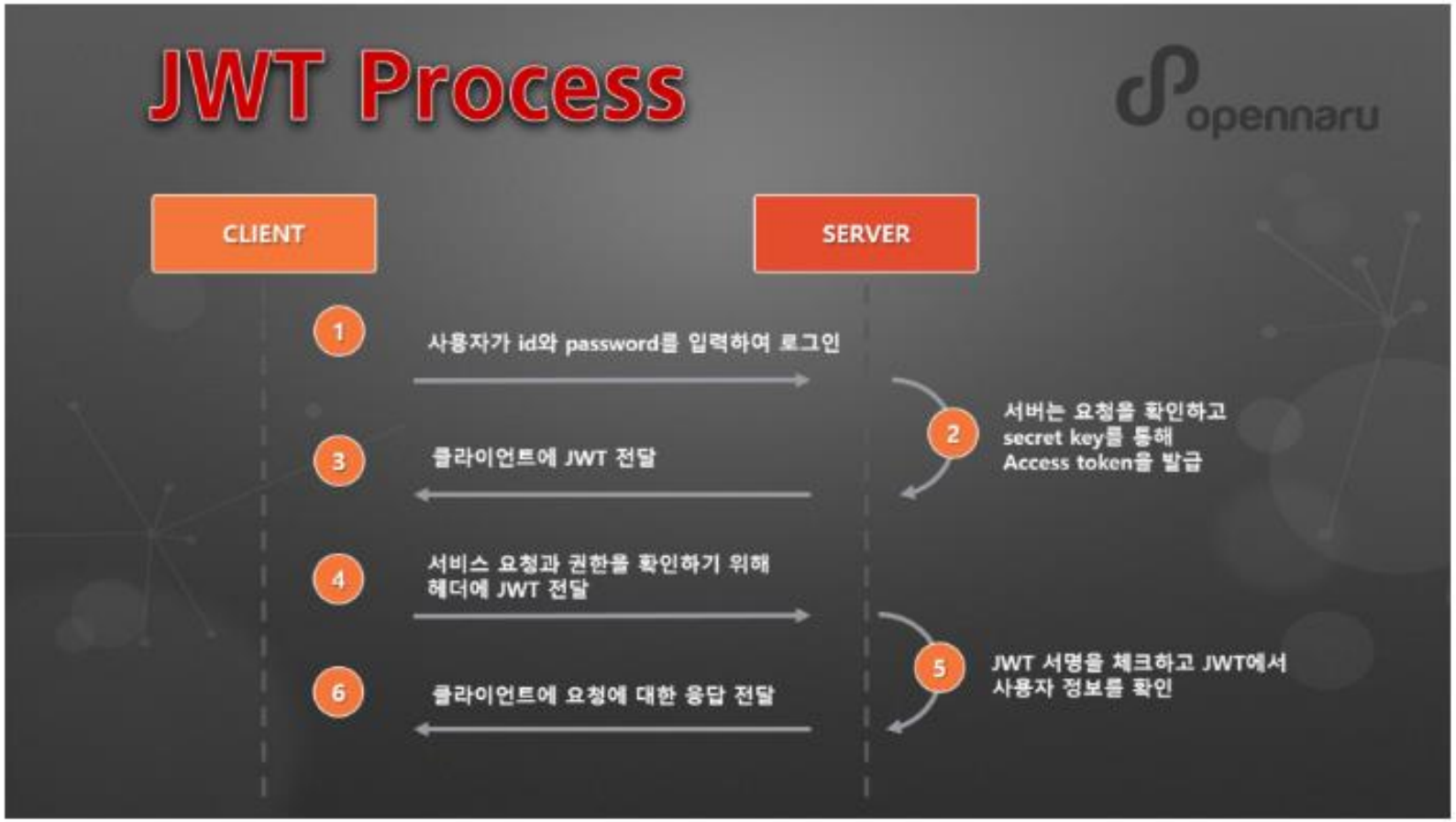
CORS (Cross-Origin Resource Sharing) : 웹 브라우저에서 다른 출처(origin)의 리소스에 접근할 수 있도록 허용하는 **보안 메커니즘**
브라우저(화면)을 보여주는 프론트 리액트가 :3000 에서 실행, 백엔드 스프링부트 : 8080에서 요청자료를 보내주면 동일한 출처가 아니므로 CORS 오류가 발생해서 요청을 거부하게 된다.
이때 서버사이드 스프링부트에서 시큐리티 설정으로 응답 헤더에 "Access-Control-Allow-Origin" 헤더를 추가하여 다른 도메인에서 접근할 수 있도록 한다.

5. JWT

- JSON Web Token (JWT)
  - **JWT 란?**
    - 웹에서 사용되는 JSON 형식의 토큰에 대한 표준 규격.
    - 주로 사용자의 인증(authentication) 또는 인가(authorization) 정보를 서버와 클라이언트 간에 안전하게 주고 받기 위해서 사용
    - JWT 토큰 웹에서 보통 Authorization HTTP 헤더를 Bearer <토큰>으로 설정하여 클라이언트에서 서버로 전송
    - 서버에서는 토큰에 포함되어 있는 서명(signature) 정보를 통해서 위변조 여부를 빠르게 검증할 수 있음
    - JWT 의 주요한 이점은 사용자 인증에 필요한 모든 정보는 토큰 자체에 포함하기 때문에 별도의 인증 저장소가 필요 없음

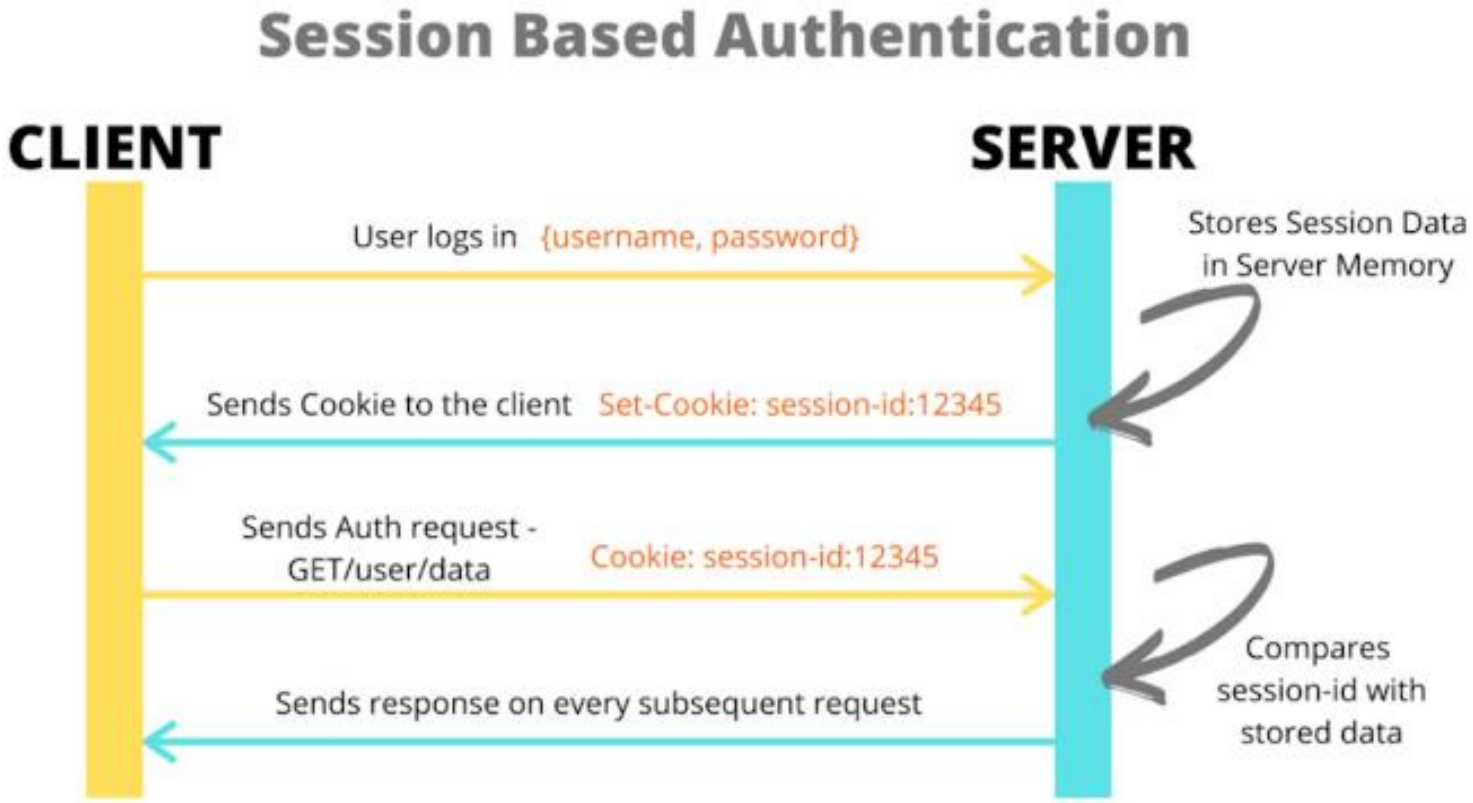
4. Security와 JWT

- JSON Web Token (JWT)
  - 회원인증 프로세스



5. JWT

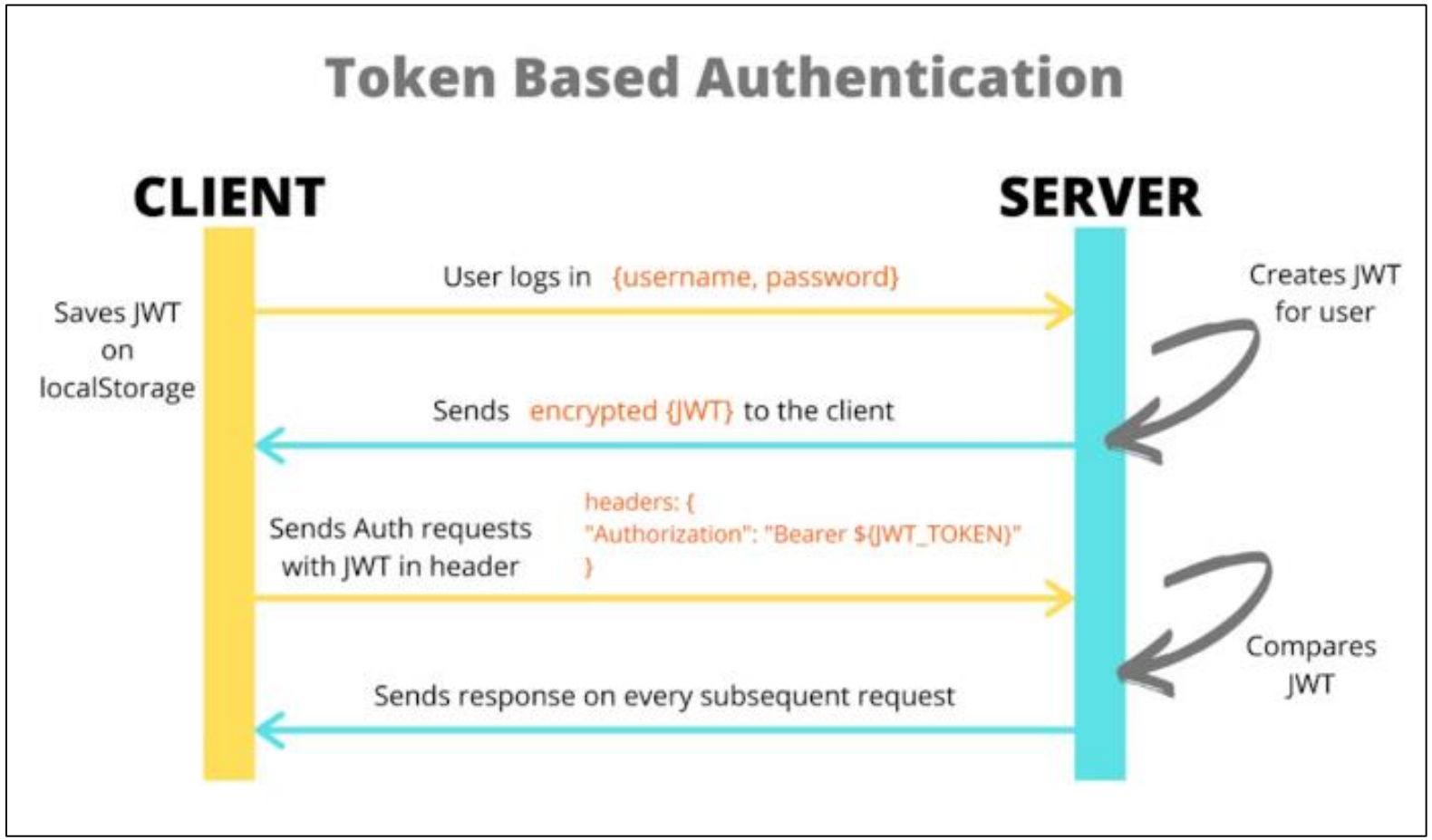
- 세션 인증 vs JWT 토큰 인증



- 사용자가 로그인하면 서버는 사용자를 위한 세션을 만들고 세션 데이터를 서버 메모리에 저장.
- 사용자가 웹 사이트에서 특정 활동을 수행하는 동안 클라이언트 브라우저의 쿠키에 저장된 세션 ID가 생성
- 사용자가 요청할 때마다 쿠키와 함께 세션 ID 전송.
- 서버는 사용자가 처음 로그인했을 때 서버 메모리에 저장된 세션 데이터로 쿠키의 세션 데이터를 확인.
- 사용자가 웹 사이트에서 로그 아웃하면 해당 세션 데이터가 데이터베이스 및 서버 메모리에서 삭제.

5. JWT

- 세션 인증 vs JWT 토큰 인증



- 사용자가 로그인 세부 정보를 사용하여 사용자 인증 요청을 보내면, 서버는 JSON 웹 토큰 (JWT) 형식으로 암호화 된 토큰을 만들어 클라이언트로 다시 보냄
- 클라이언트가 토큰을 수신하면 사용자가 클라이언트를 사용하여 모든 활동을 수행하도록 인증되었음을 의미함.

5. JWT

- JWT 토큰 인증
  - JWT는 클라이언트 localStorage에 저장되며 사용자가 서버에서 데이터를 요청하거나 수행할 때 해당 사용자의 고유 키로 사용
  - 서버가 요청을 수신하면 해당 요청에 대해 JWT를 확인한 다음 필요한 응답을 클라이언트로 다시 보냄
  - 요청시 http 헤더에 json 토큰이 추가되어 서버로 전송되는 방식

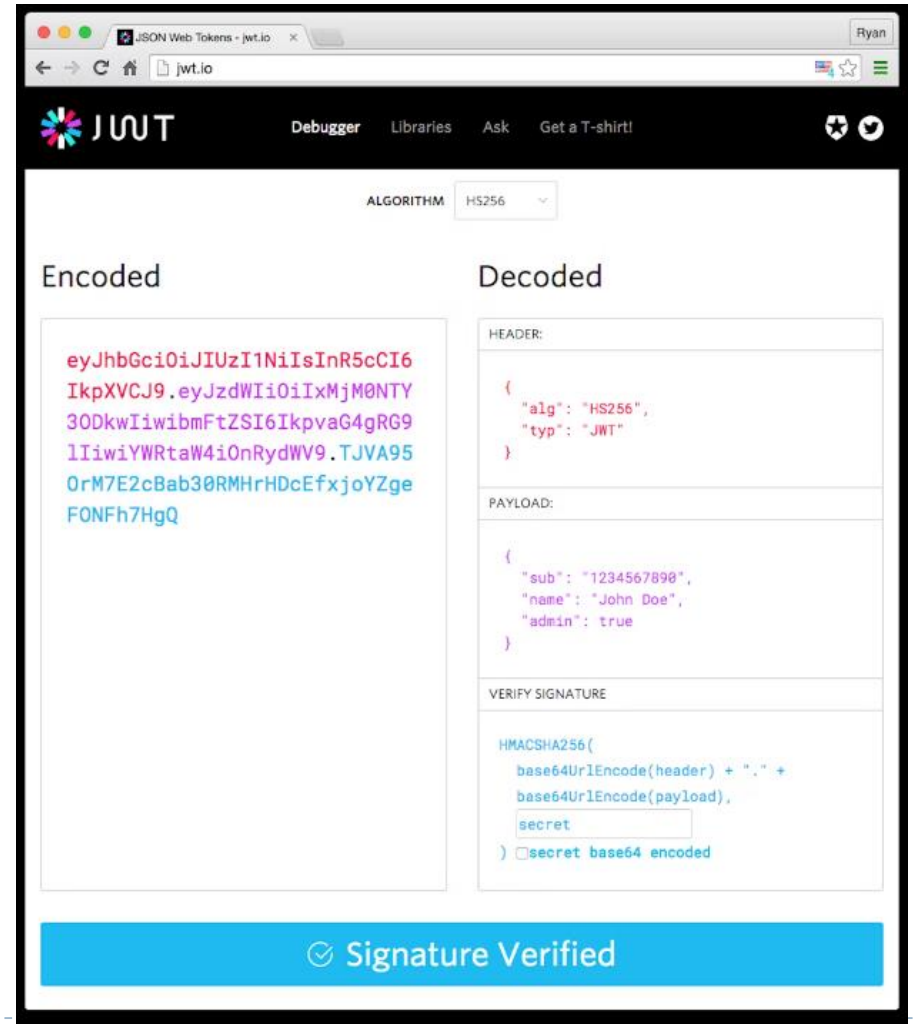
```
headers: {
"Authorization": "Bearer ${JWT_TOKEN}"
}
```

  - JSON 웹 토큰은 점 (으로 연결된 세 부분으로 구성)
    1. 헤더(Header)
    2. 페이로드(Payload) : data
    3. 서명(Signature)

```
xxxxx.yyyyy.zzzzz
```

5. JWT

- **JWT의 구조:**



header
토큰의 타입과 Signature 해싱 알고리즘

payload(정보, data)
정보의 한 조각을 claim이라고 하며, 크레임은 Json(Key/Value) 형태의 한 쌍으로 이뤄져 있음

서명 (Signature)
토큰을 인코딩하거나 유효성 검증을 할 때 사용하는 고유한 암호화 코드

Jwt의 단점
쿠키/세션과 다르게 토큰의 길이가 길어, 인증 요청이 많아질 수록 네트워크 부하가 심해 짐.
쿠키/세션보다 복잡함.

5. JWT

- JWT 설정 과 properties 설정
  - dependency 추가

```
// https://mvnrepository.com/artifact/io.jsonwebtoken/jjwt-api
implementation group: 'io.jsonwebtoken', name: 'jjwt-api', version: '0.12.6‘

// https://mvnrepository.com/artifact/io.jsonwebtoken/jjwt-impl
runtimeOnly group: 'io.jsonwebtoken', name: 'jjwt-impl', version: '0.12.6‘

// https://mvnrepository.com/artifact/io.jsonwebtoken/jjwt-jackson
runtimeOnly group: 'io.jsonwebtoken', name: 'jjwt-jackson', version: '0.12.6'
```

```
<!-- https://mvnrepository.com/artifact/io.jsonwebtoken/jjwt-api -->
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-api</artifactId>
    <version>0.11.5</version>
</dependency>

<!-- https://mvnrepository.com/artifact/io.jsonwebtoken/jjwt-impl -->
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-impl</artifactId>
    <version>0.11.5</version>
    <scope>runtime</scope>
</dependency>

<!-- https://mvnrepository.com/artifact/io.jsonwebtoken/jjwt-jackson -->
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-jackson</artifactId>
    <version>0.11.5</version>
    <scope>runtime</scope>
</dependency>
```

5. JWT

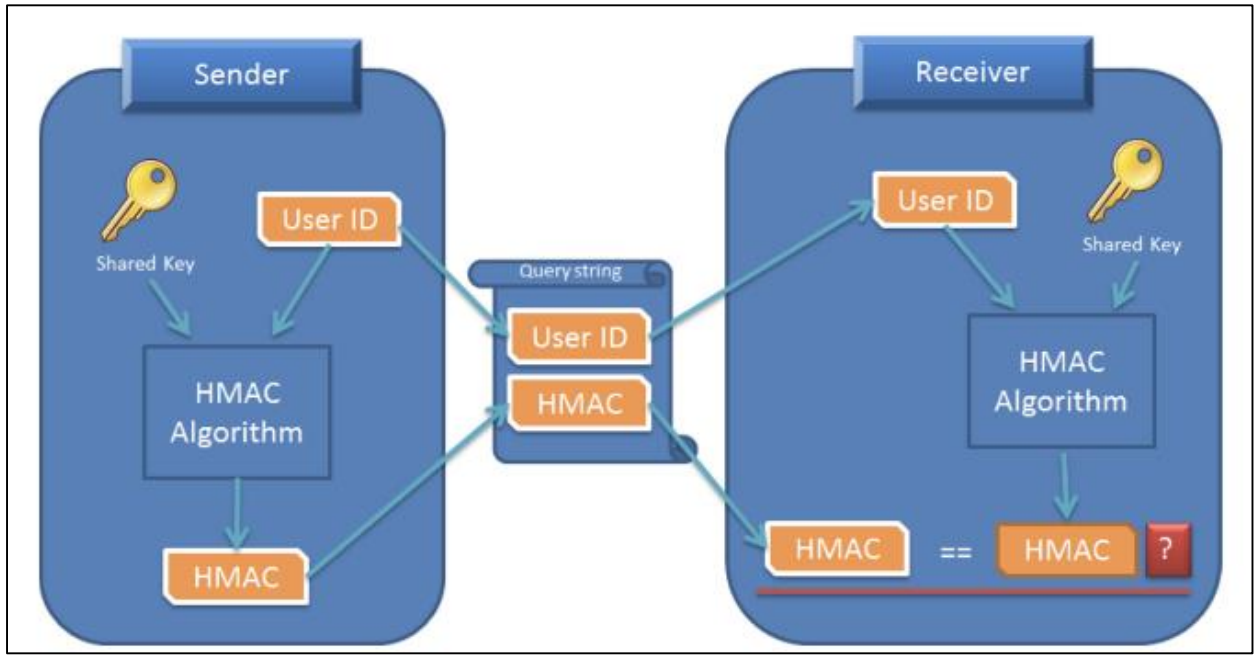
- JWT 설정 과 properties 설정
  - **application.properties**

```
#Key depends on JWT alogrithm HMAC ( >= 64bytes )
app.jwt.secret=RandomSecretKey1234567890!RandomSecretKey1234567890!RandomSecretKey1234567890!
#1 day
app.jwt.expiration-in-ms=86400000
```

- HMAC(Hash-based Message Authentication Code)
- 메시지 인증 코드(Message Authentication Code, MAC)는 메시지의 인증에 쓰이는 정보(코드)이다.
- 메시지의 무결성 및 신뢰성을 보장하는 용도로 MAC을 사용한다

5. JWT

- HMAC(Hash-based Message Authentication Code)
  - 메시지 인증 코드(Message Authentication Code, MAC)는 메시지의 인증에 쓰이는 정보(코드
  - 메시지의 무결성 및 신뢰성을 보장하는 용도로 MAC을 사용



① 사전에 Sender와 Receiver는 별도 채널로 해시에 사용할 키(Share key)를 공유, 양쪽에서 사용할 해시 알고리즘을 정함.

② Sender는 공유키를 사용해서 UserId를 해싱.

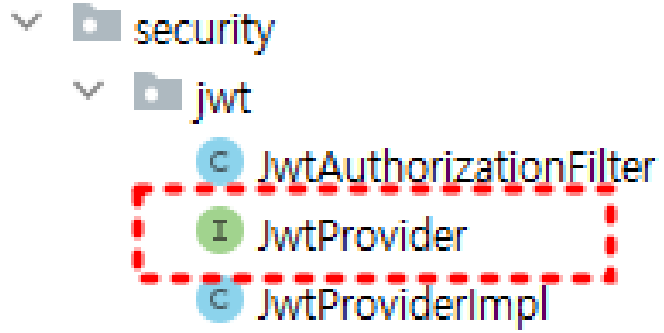
③ Sender는 원본 UserId와 그 해시결과(HMAC)을 쿼리스트링 값으로 Receiver에게 전달.

④ Receiver는 받은 UserId를 공유키를 사용하여 같은 알고리즘으로 해시한 결과(Receiver's HMAC)를 만생

⑤ Receiver가 만든 HMAC과 쿼리스트링으로 받은 HMAC이 같다면 UserId는 변경되지 않았다고 신뢰할 수 있다.

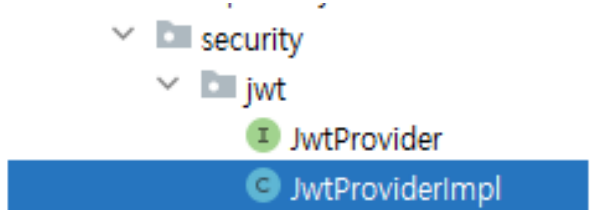
5. JWT

- 토큰 생성 메서드
  - JwtProvider 인터페이스




```
package org.pgm.productseller.security.jwt;

import jakarta.servlet.http.HttpServletRequest;
import org.pgm.productseller.security.UserPrinciple;
import org.springframework.security.core.Authentication;

public interface JwtProvider {

    String generateToken(UserPrinciple auth);

    Authentication getAuthentication(HttpServletRequest request);

    boolean isTokenValid(HttpServletRequest request);
}
```

5. JWT



- JwtProviderImpl 토큰 생성 메서드

```
@Component
public class JwtProviderImpl implements JwtProvider {

    @Value("${app.jwt.secret}")
    private String JWT_SECRET;

    @Value("${app.jwt.expiration-in-ms}")
    private Long JWT_EXPIRATION_IN_MS;

    @Override
    public String generateToken(UserPrinciple auth){

        String authorites = auth.getAuthorities().stream().map(GrantedAuthority::getAuthority).collect(Collectors.joining(","));
        Key key = Keys.hmacShaKeyFor(JWT_SECRET.getBytes(StandardCharsets.UTF_8));

        return Jwts.builder()
                .setSubject(auth.getUsername())
                .claim("roles", authorites)
                .claim("userId", auth.getId())
                .setExpiration(new Date(System.currentTimeMillis() + JWT_EXPIRATION_IN_MS))
                .signWith(key, SignatureAlgorithm.HS512)
                .compact();
    }
}
```

```
"authorities": [
  "ROLE_ADMIN",
  "ROLE_USER"
]
```

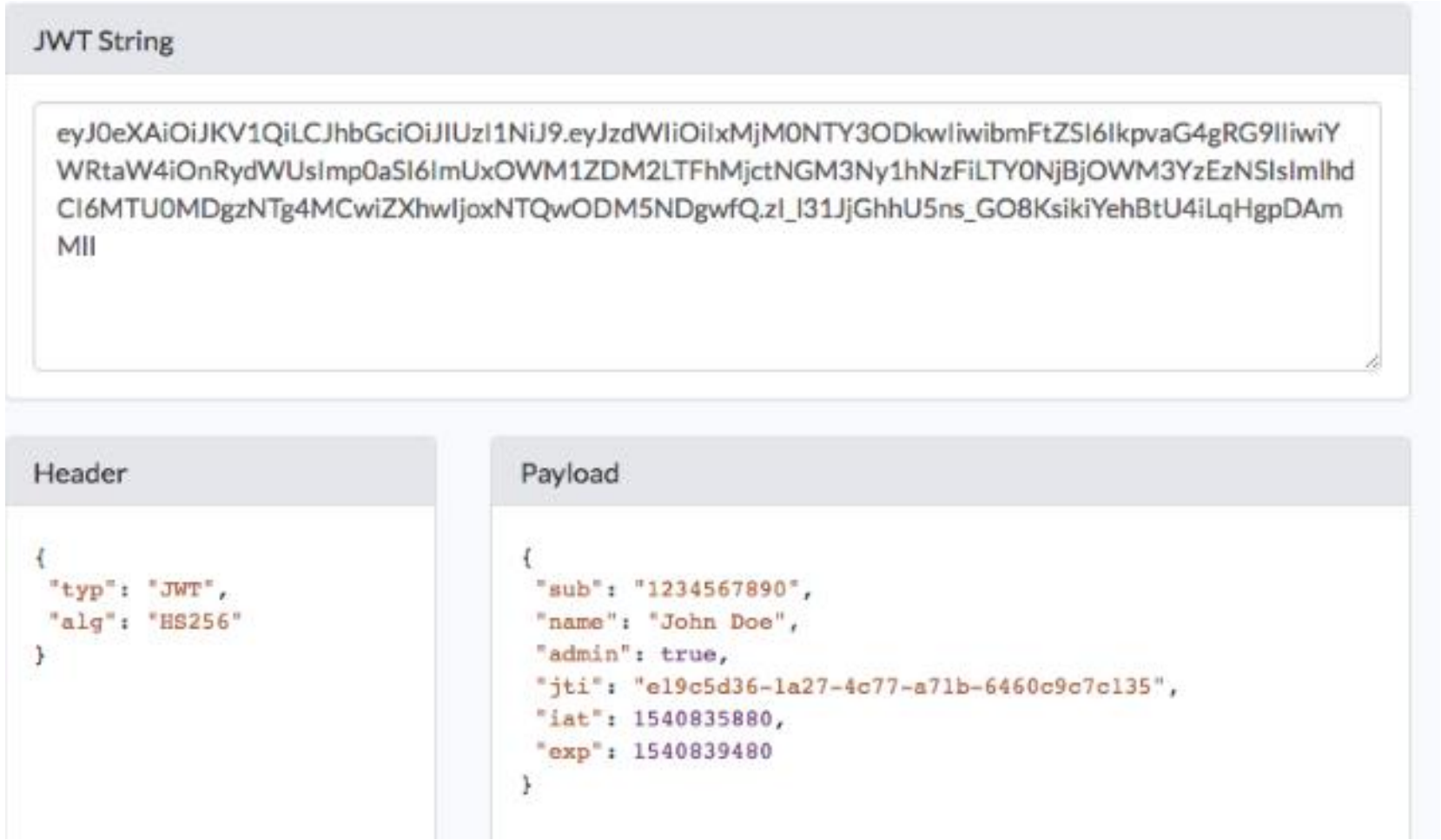
토큰 생성 함수

UserPrinciple : 인증 후 유저정보가 저장된 객체

authorites : 문자열로 ["ROLE_ADMIN", "ROLE_USER"] 을 생성

5. JWT

- JwtProviderImpl 토큰 생성 메서드



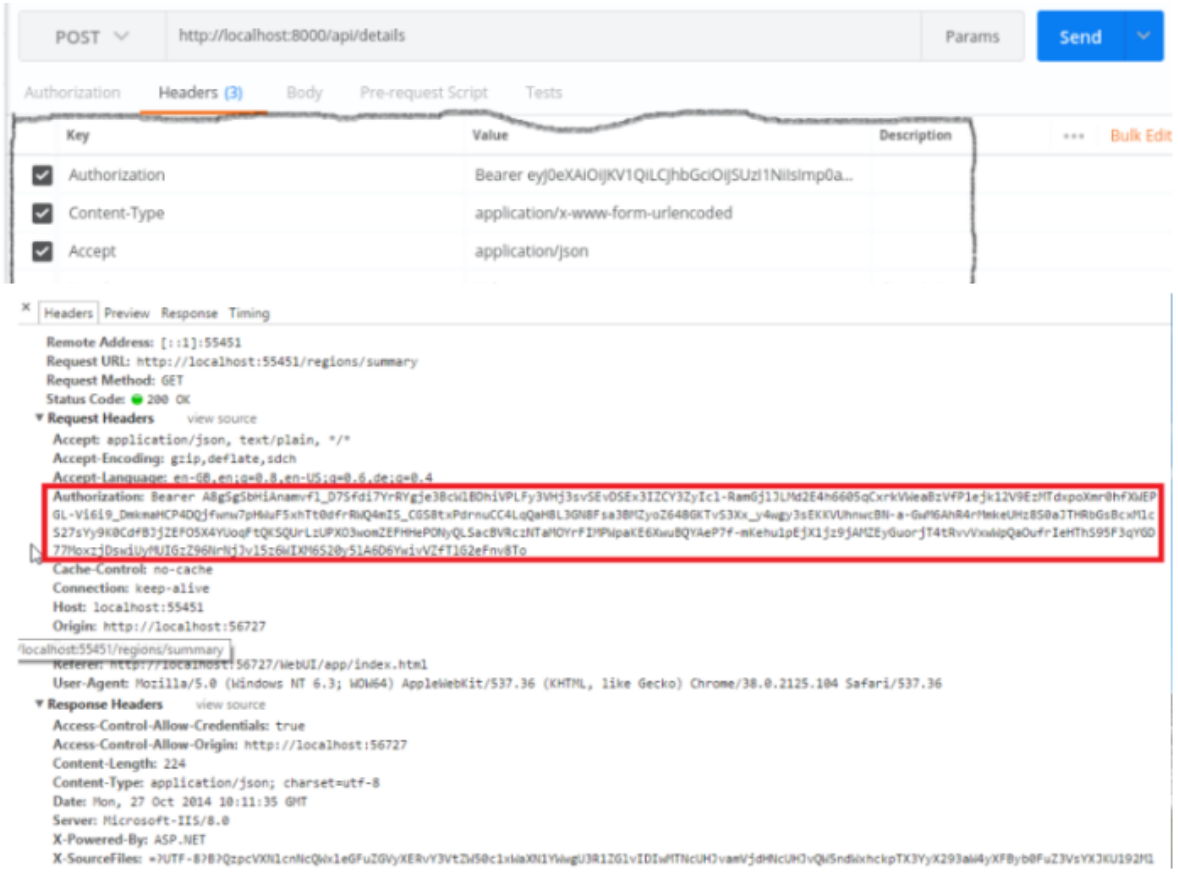
5. JWT

- 요청시 static token가져오는 메서드


```
public class SecurityUtils {
    public static final String ROLE_PREFIX = "ROLE_";
    public static final String AUTH_HEADER = "authorization";
    public static final String AUTH_TOKEN_TYPE = "Bearer";
    public static final String AUTH_TOKEN_PREFIX = AUTH_TOKEN_TYPE + " ";


     public static String extractAuthTokenFromRequest(HttpServletRequest request){
         String bearerToken = request.getHeader(AUTH_HEADER);

         if(StringUtils.hasLength(bearerToken) && bearerToken.startsWith(AUTH_TOKEN_PREFIX)){
             log.info(bearerToken.substring(7));
             return bearerToken.substring(7); //토큰 부분만 잘라서 리턴
         }
         return null; //없을경우
     }

}
```

5. JWT

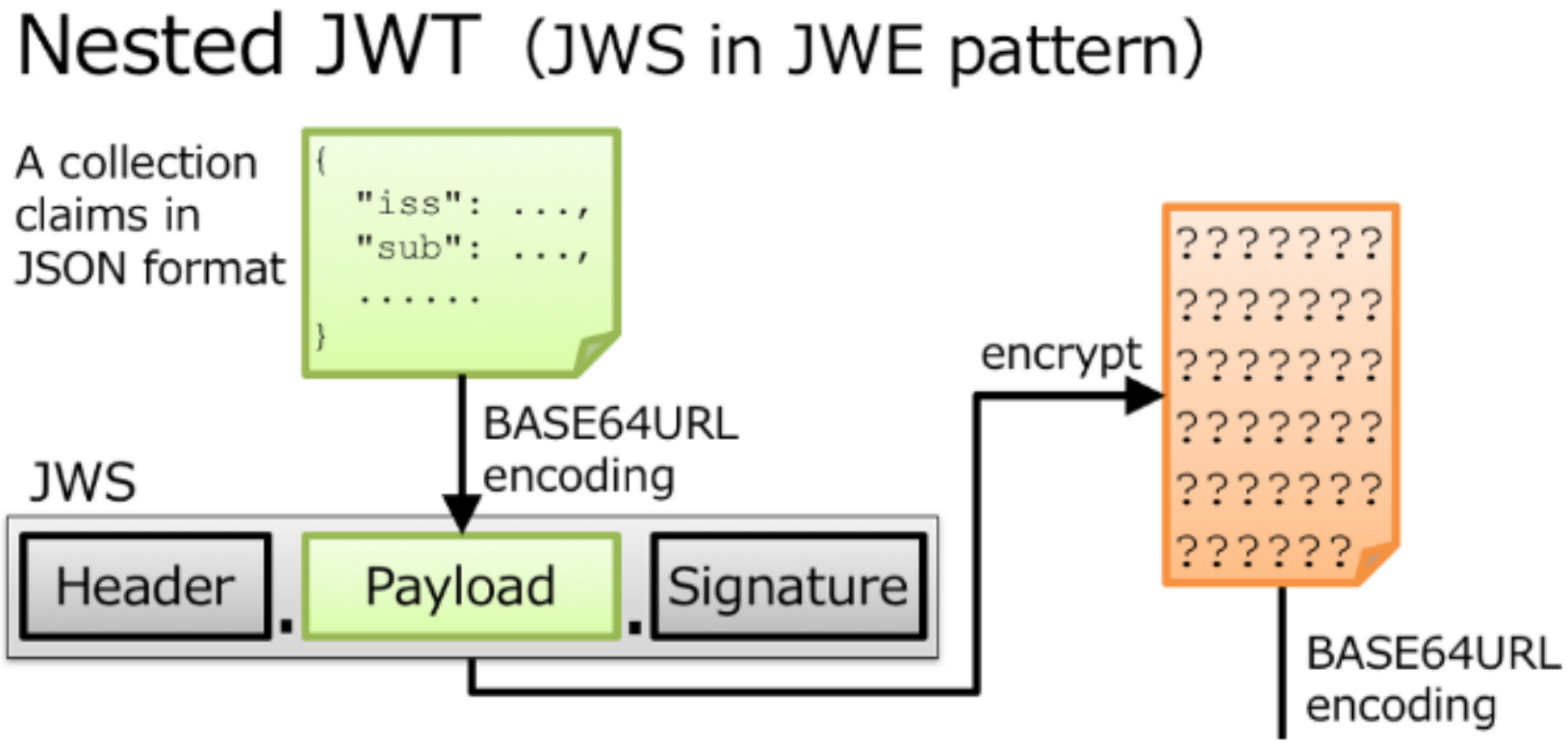
- 요청시 static token가져오는 메서드



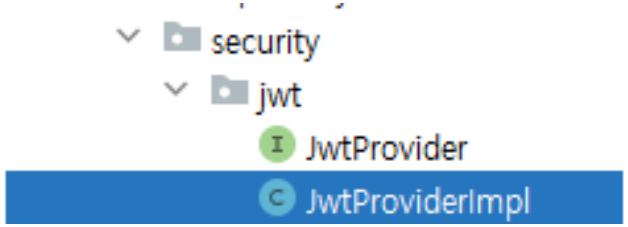

```
Authorization: Bearer <token>
```

5. JWT

- Request의 토큰에서 암호풀어 Claims 가져오기



5. JWT



- request의 토큰에서 암호풀어 Claims 가져오기
  - 사용자 인증 정보를 기반으로 **JWT(JSON Web Token)**을 생성

```
private Claims extractClaims(HttpServletRequest request){
        String token = SecurityUtils.extractAuthTokenFromRequest(request);

        if(token == null) {
            System.out.println("token null error");
            return null;
        }

        Key key = Keys.hmacShaKeyFor(JWT_SECRET.getBytes(StandardCharsets.UTF_8));

        return  Jwts.parserBuilder()
                .setSigningKey(key)
                .build()
                .parseClaimsJws(token)
                .getBody();
    }
}
```

request에서 토큰을 가져오고 토큰이 null이 아닐 경우에
시크릿키를 사용해 다시 암호를 풀어
Claims (name/value 쌍으로 이루어진 값들) 형식으로 가져 옴

5. JWT

- 이름,비번,권한으로 인증토큰 생성 , 토큰이 사용가능한지 체크 하는 메서드


```
@Override
public Authentication getAuthentication(HttpServletRequest request){

    Claims claims = extractClaims(request); //클레임 추출
    if(claims == null) return null;
    String username = claims.getSubject();  //클레임에서 사용자 정보 추출
    Long userId = claims.get("userId", Long.class);

    Set<GrantedAuthority> authorities = Arrays.stream(claims.get("roles").toString().split(","))
            .map(SecurityUtils::convertToAuthority)
            .collect(Collectors.toSet());

    UserDetails userDetails = UserPrinciple.builder()
            .username((username))
            .authorities(authorities)
            .id(userId)
            .build();
    if(username == null) {
        log.info("username null error");
        return null;
    }
    return new UsernamePasswordAuthenticationToken(userDetails, null, authorities);  // 인증객체 생성 후 반환
}
```

5. JWT

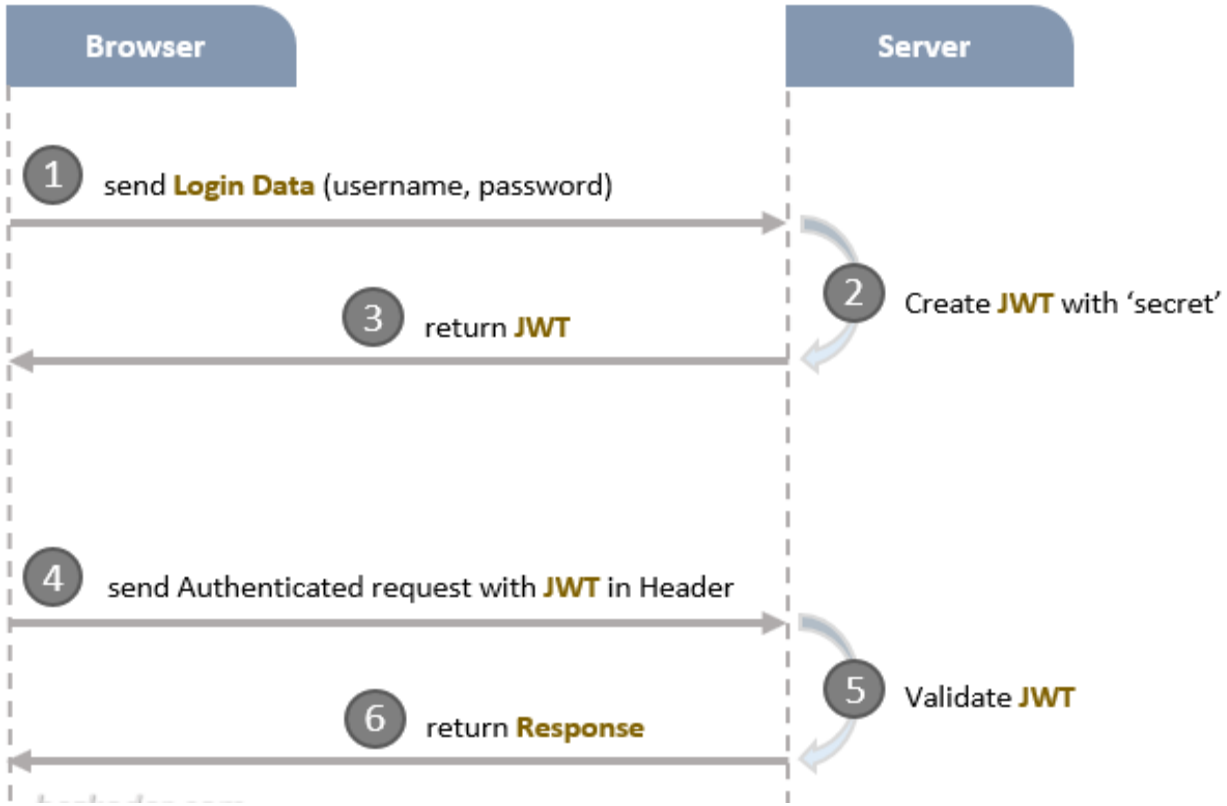
- 토큰 유효성 체크

```
@Override
public boolean isTokenValid(HttpServletRequest request){

    Claims claims = extractClaims(request);

    if(claims == null) {
        log.info("claims null error");
        return false;
    }

    if(claims.getExpiration().before(new Date())) return false;

    return true;
}
```

토큰에 claims 데이터가 없으면 사용불가 false 리턴
토큰사용기간 만료시에도 사용불가 false 리턴
아니면 true 리턴

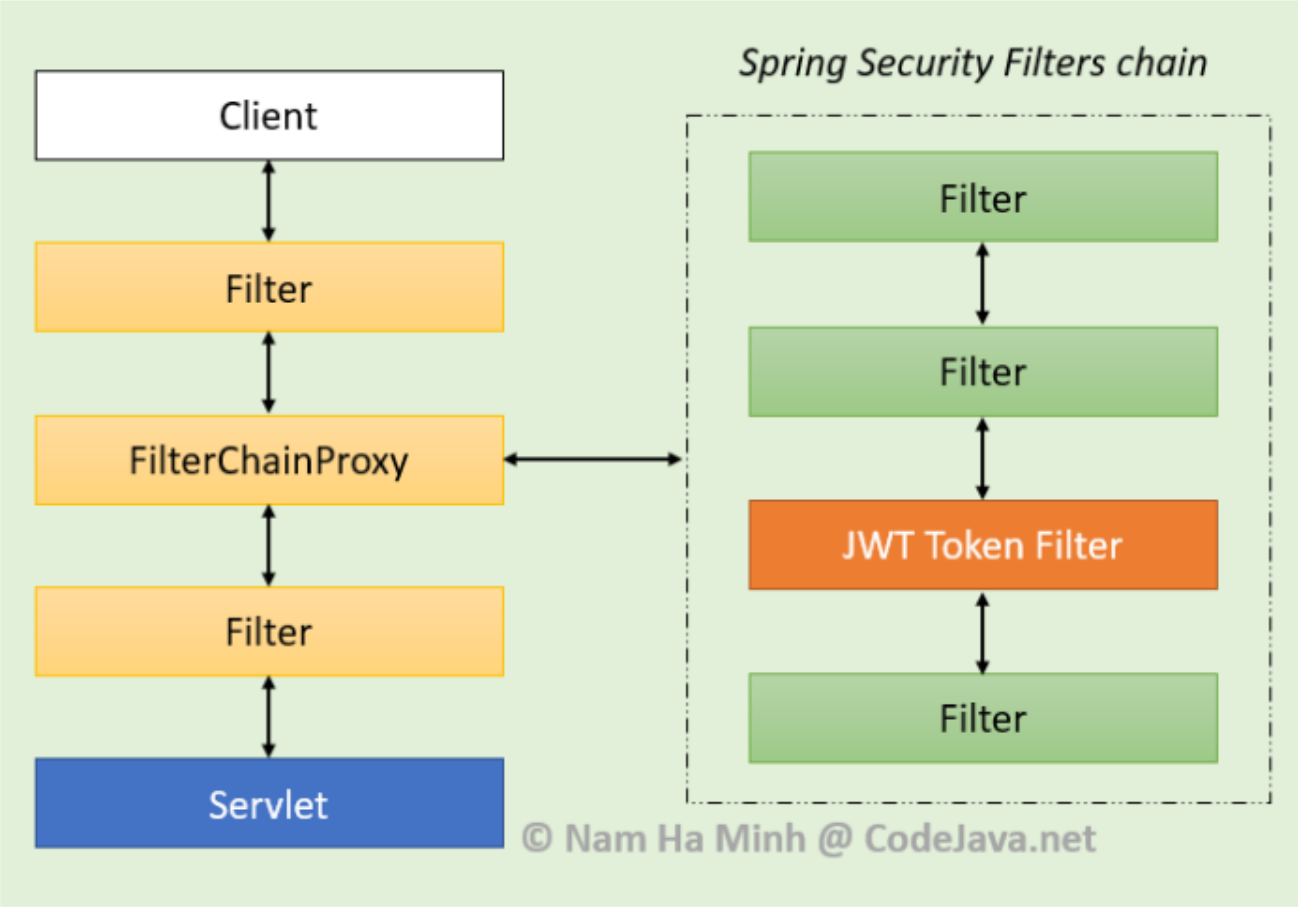
5. JWT

- 시큐리티 JWT 필터 만들기
  - JWT 인증 과정



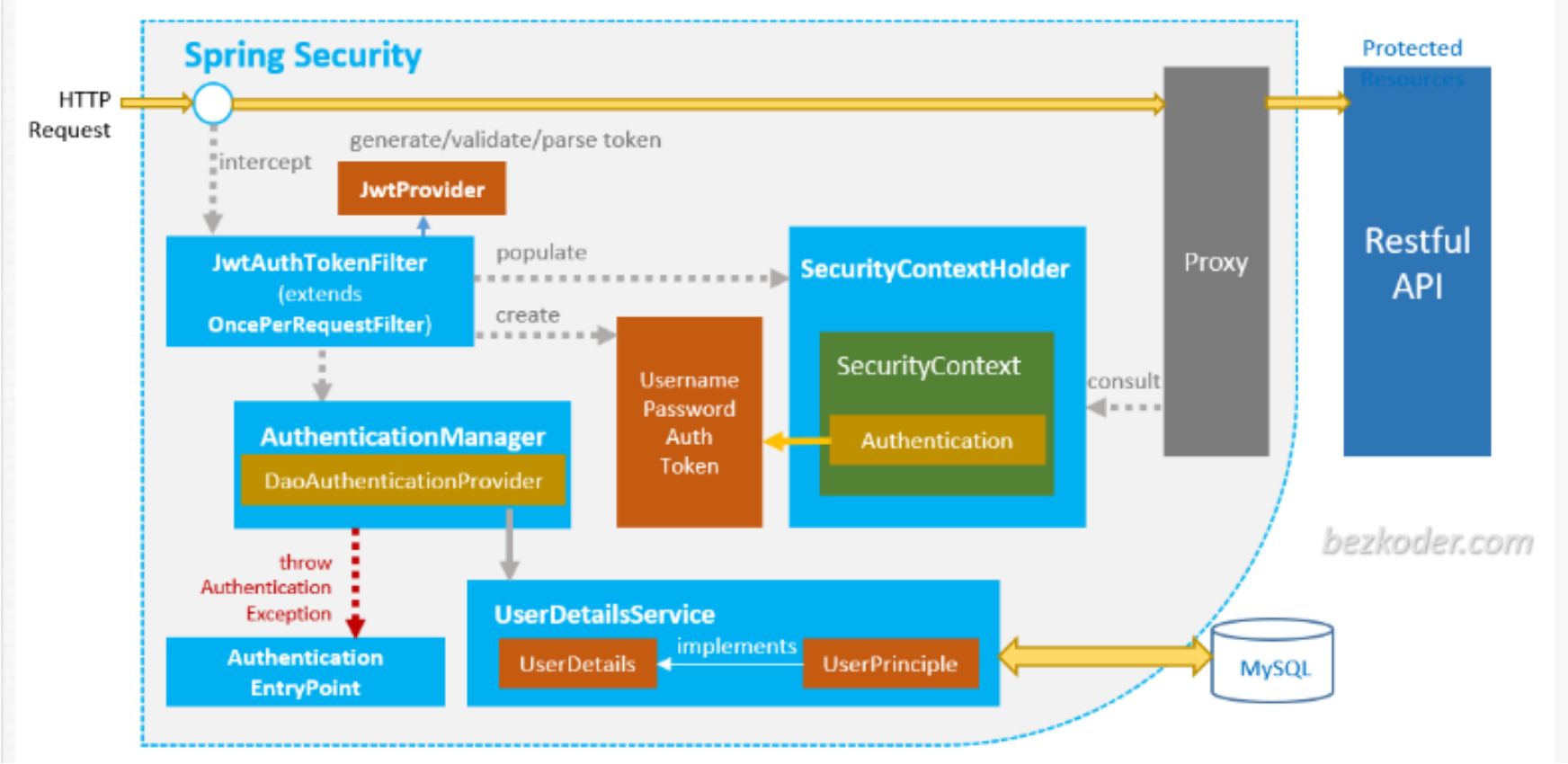
5. JWT

- **스프링 시큐리티 필터 체인**

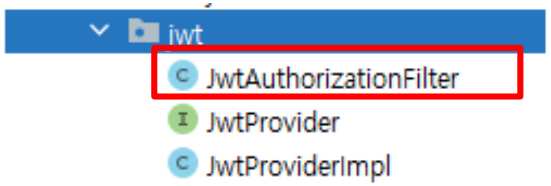


5. JWT

- **스프링 시큐리티 필터 체인** 한번의 인증처리 -> 계속 인증체크하면 작업이 중복되니까 체크는 한번만 하게
  - 여러 개의 필터체인 중 JWT 토큰을 인증하기 위한 필터를 생성
  - 단 한번에 인증으로 다시 인증 체크하는 일이 없도록 보장해야 한다!



5. JWT



JwtAuthorizationFilter

인증 체킹을 한번만 하게 세팅. -> 다른 작업동안 인증 유지

```
public class JwtAuthorizationFilter extends OncePerRequestFilter {

    @Autowired
    private JwtProvider jwtProvider;

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
            throws ServletException, IOException { //FilterChain: 다음 필터로 요청을 전달하기 위한 객체.

        Authentication authentication = jwtProvider.getAuthentication(request); //인증(Authentication) 객체 가져오기

        if(authentication != null && jwtProvider.isTokenValid(request)){ //JWT 유효성 검증 및 인증 설정
            SecurityContextHolder.getContext().setAuthentication(authentication); //인증된 사용자 정보를 SecurityContext에 설정
        }

        filterChain.doFilter(request, response); //다음 필터로 요청 전달
    }
}
```

OncePerRequestFilter Class : Http Request의 한 번의 요청에 대해 한 번만 실행하는 Filter
예: 컨트롤러에서 Forwarding이 발생하면 Filter Chain이 다시 동작하여 인증처럼 딱 한번만 수행하도록 하여 Logic이 불필요하게 여러 번 수행될 수 있는 것을 막아 줌.

다음 필터로 요청을 전달

-> 토큰 받아서 정보 뽑고 다시 토큰 만드는걸 여기서 ?

userprincipal이 null 아니면

<- 객체가 얻어

토큰에 정보를 얻어서 시큐리티 컨텍스트에 정보를 넣고 -> 다음 작업하는 필터에 넣어

5. JWT

- SecurityConfig에 빈등록 및 필터 추가

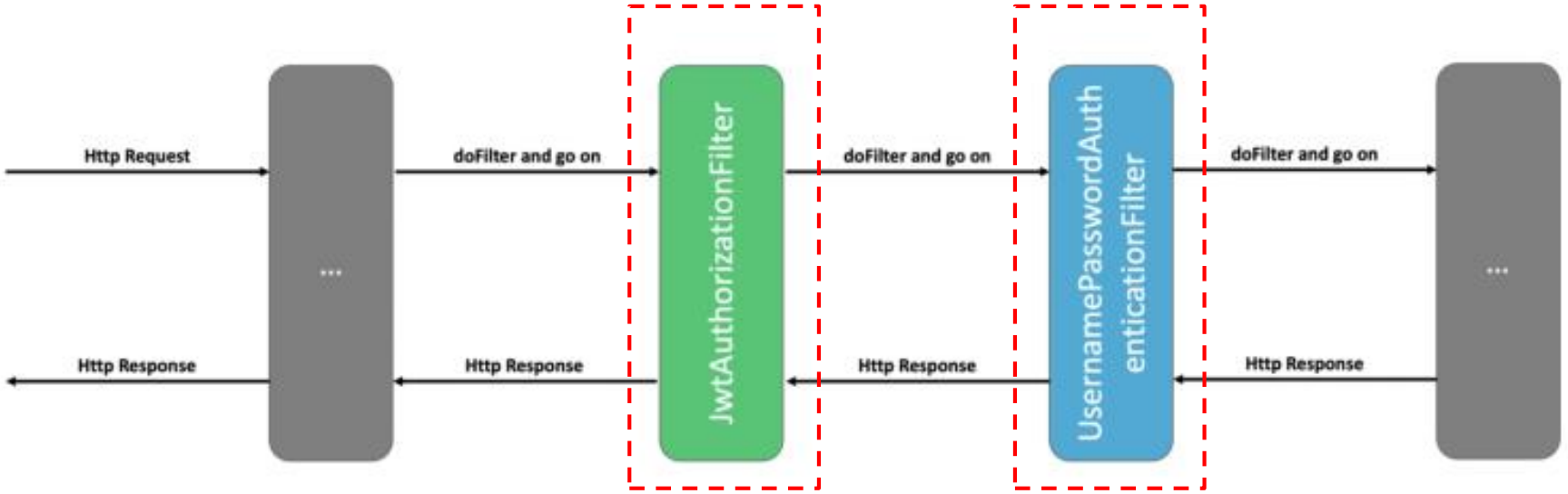
```
@Bean
public JwtAuthorizationFilter jwtAuthorizationFilter(){
    return new JwtAuthorizationFilter();
}
```

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception{
    http
            .csrf(AbstractHttpConfigurer::disable)
            .cors(httpSecurityCorsConfigurer -> corsConfigurationSource())
            .sessionManagement((session)->session
                    .sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .authorizeHttpRequests((authz)-> authz
                    .requestMatchers("/api/authentication/**").permitAll()
                    .requestMatchers(HttpMethod.GET, "/api/product/**").permitAll()
                    .requestMatchers("/api/product/**").hasRole(Role.ADMIN.name())
                    .anyRequest().authenticated()
            )
            // 사용자 정의 JWT 인증 필터(jwtAuthorizationFilter)를 Spring Security의 필터 체인에 추가
            // 이 필터는 Spring Security의 기본 인증 필터(UsernamePasswordAuthenticationFilter)보다 먼저 실행
            .addFilterBefore(jwtAuthorizationFilter(), UsernamePasswordAuthenticationFilter.class);
    return http.build();
}
```

-> 인정 처리 끝! 컨트롤러 등 작업을 ㅅ

5. JWT

- 시큐리티 설정에 JWT필터 추가하기

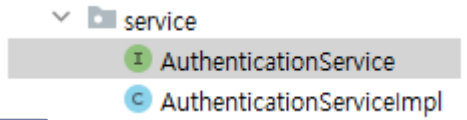


JWTAuthorizationFilter : 직접 만든 커스텀 필터
UsernameAndPasswordAuthenticationFilter : 스프링 시큐리티의 기본 로그인 필터

반드시 커스텀 JWT 필터는 시큐리티의 기본 필터보다 먼저 처리되어야 함.

5. JWT

- AuthenticationServiceImpl 인증 메소드 구현




```
public interface AuthenticationService {
    public User signInAndReturnJWT(User signInRequest);
}
```

5. JWT

- AuthenticationServiceImpl 인증메소드 구현

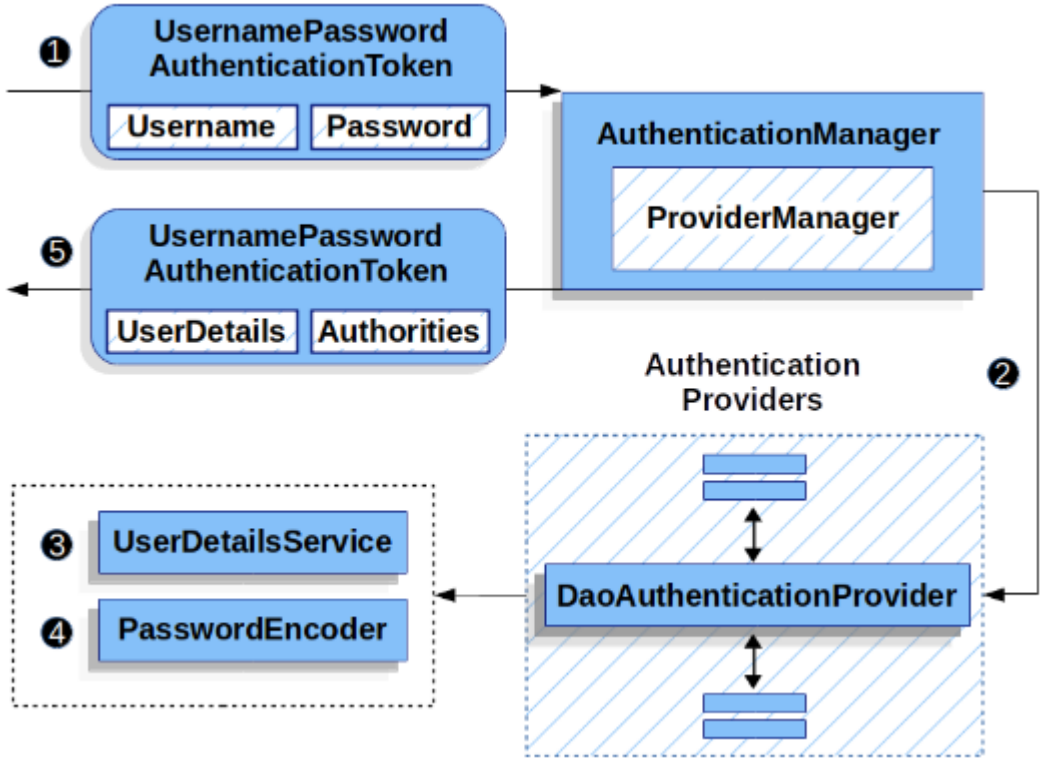
```
@Service
@RequiredArgsConstructor
public class AuthenticationServiceImpl implements AuthenticationService{
    private final AuthenticationManager authenticationManager;
    private final JwtProvider jwtProvider;

    @Override
    public User signInAndReturnJWT(User signInRequest) {
        Authentication authentication = authenticationManager.authenticate(  //사용자 인증 처리
                // username과 password를 담은 인증 토큰 생성
                new UsernamePasswordAuthenticationToken(signInRequest.getUsername(), signInRequest.getPassword())
        );

        UserPrinciple userPrinciple = (UserPrinciple) authentication.getPrincipal(); // 인증성공한 사용자 정보 얻기
        String jwt = jwtProvider.generateToken(userPrinciple); // UserPrinciple 정보를 기반으로 JWT 생성

        User signInUser = userPrinciple.getUser();
        log.info("로그인 처리 유저:``````````````````"+signInUser);
        signInUser.setToken(jwt);

        return signInUser;
    }
}
```

5. JWT

AuthenticationServiceImpl 인증메소드 구현

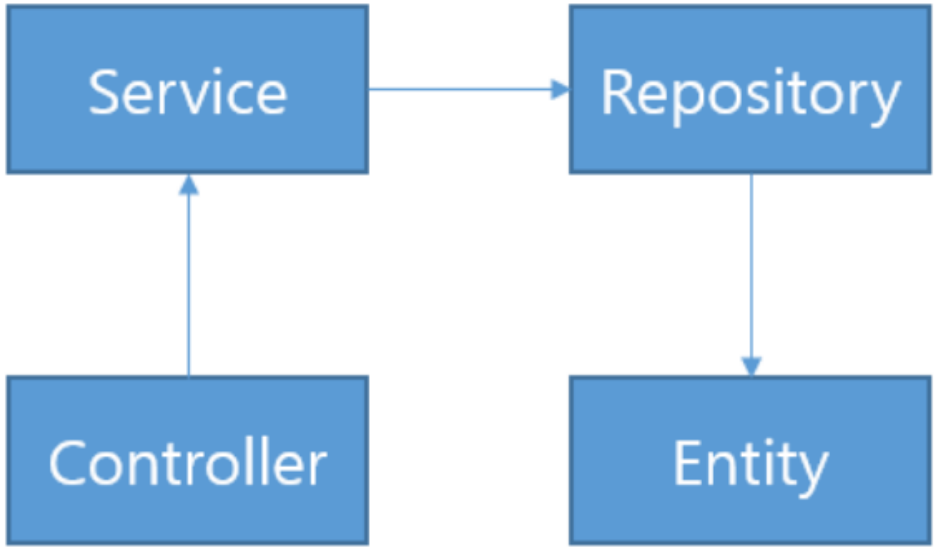
- 유저의 로그인 요청시 AuthenticationManager로 인증하고 인증되면 JwtProvider 에서 인증된 d사용자 정보로 토큰을 발행
- 발행된 토큰은 유저의 필드변수에 저장장후 리턴(DB에는 저장되지 않도록 @Transient 붙임)


```
@Transient
private String token; //DB에 저장되지 않음
```



5. JWT

- 로그인과 가입하기



Entity => Repository => Service => 컨트롤러 작성

인증관련 엔드포인트 주소
sign-in 로그인, sign-up 회원가입

Sign-In => POST /api/authentication/sign-in

Sign-Up => POST /api/authentication/sign-up

```
.authorizeHttpRequests((authz)-> authz
        .requestMatchers("/api/authentication/**").permitAll()
        .anyRequest().authenticated()
)
```

시큐리티 설정에서 모든 사용자에게 허용되는 엔드포이트 주소는 /api/authentication/모든주소 이므로 인증 로그인, 회원가입 요청은 인증되지 않은 사용자(모든 사용자)에게 허용됨

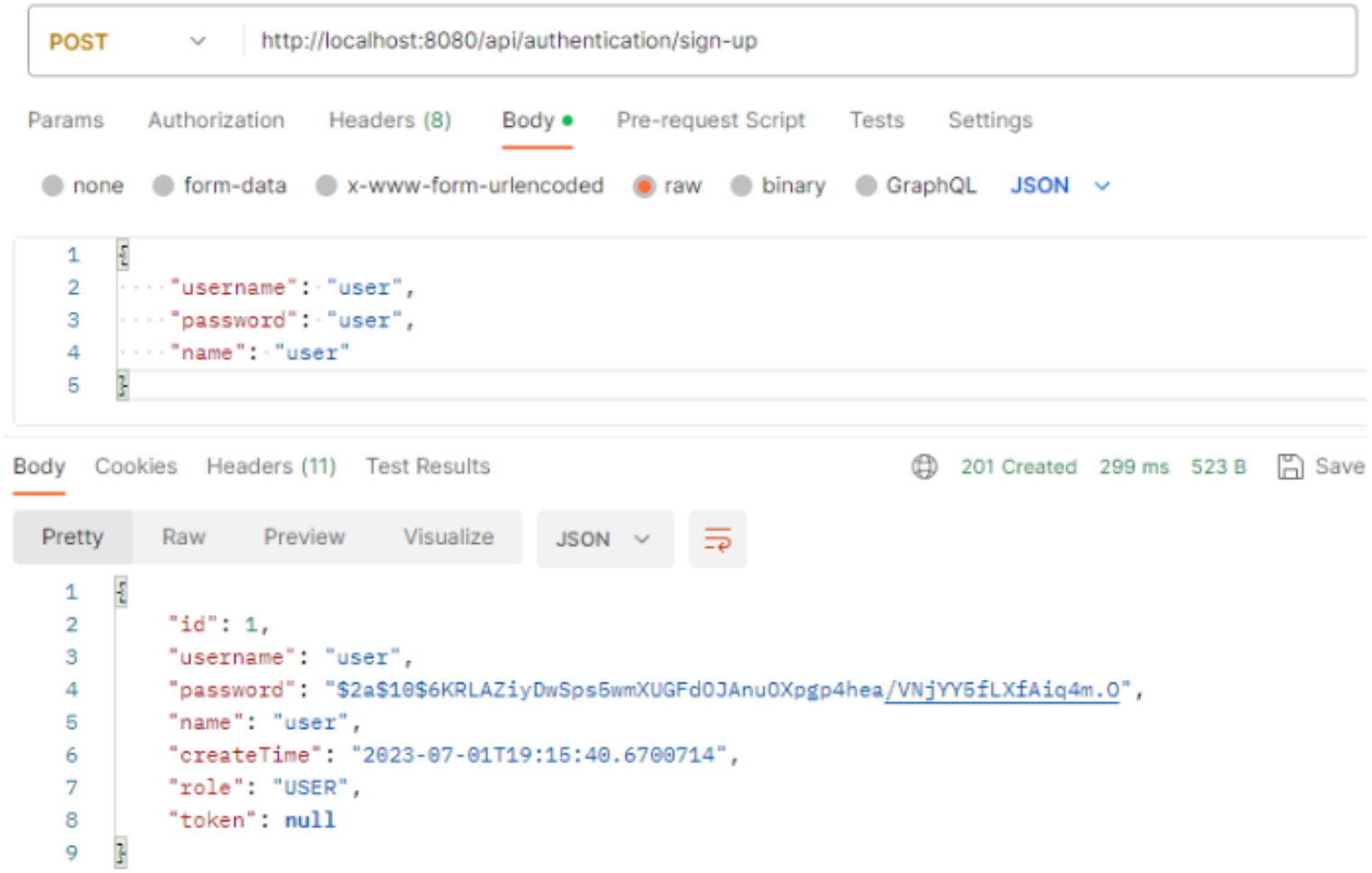
5. JWT

AuthenticationController

```
@RestController
@RequestMapping("api/authentication")
@RequiredArgsConstructor
public class AuthenticationController {

    private final AuthenticationService authenticationService;
    private final UserService userService;

    @PostMapping("sign-up")
    public ResponseEntity<Object> signUp(@RequestBody User user) {
        if(userService.findByUsername(user.getUsername())!=null){  // 중복 값이 존재하면
            return new ResponseEntity<>(HttpStatus.CONFLICT);  // 409 CONFICT  : Spring Security에서 인증실패 예외 처리
        }
        // 새 User DB에 저장하고 , 201 상태코드 반환
        return new ResponseEntity<>(userService.saveUser(user), HttpStatus.CREATED); }

    @PostMapping("sign-in")
    public ResponseEntity<Object> signIn(@RequestBody User user){
        // 로그인 성공 시, 생성된 JWT와 함께 상태코드 200 반환
        return new ResponseEntity<>(authenticationService.signInAndReturnJWT(user), HttpStatus.OK);
    }
}
```

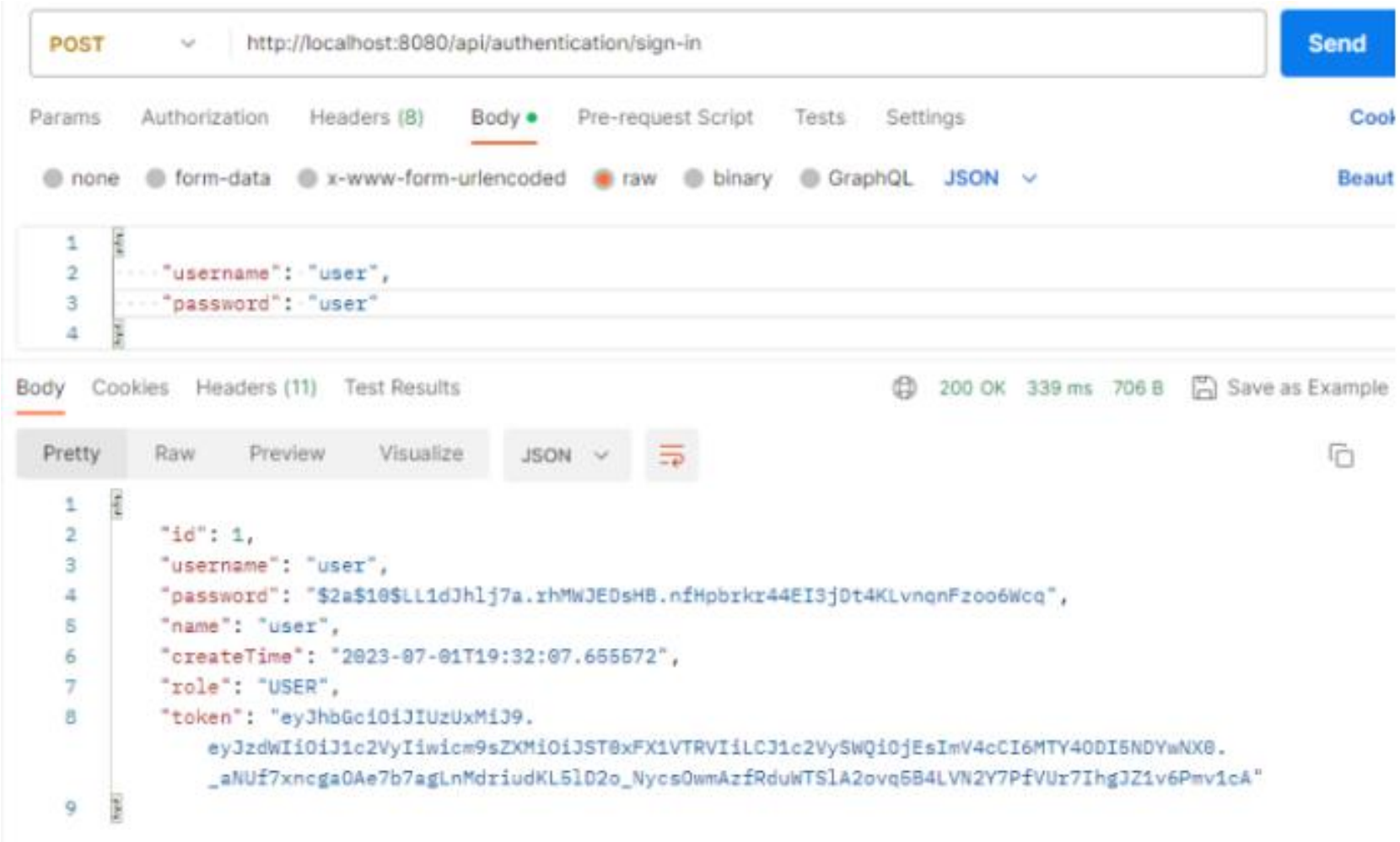
5. JWT

- 포스트맨 테스트 (sign-up 가입하기)



5. JWT

- 포스트맨 테스트 (sign-in 로그인 인증)



5. JWT

- 로그인(인증)된 사용자 정보 가져오는 방법(Tip)
  - **Bean 에서 사용자 정보 얻기**

```
Object principal = SecurityContextHolder.getContext().getAuthentication().getPrincipal();
UserDetails userDetails = (UserDetails)principal;
String username = principal.getUsername();
```

  - **Controller 에서 사용자 정보 얻기(Principal 객체로 얻기)**

```
@Controller
public class SecurityController {
	@GetMapping("/username")
	@ResponseBody
	public String currentUserName(Principal principal) {
		return principal.getName();
	}
}
```

5. JWT

- 로그인(인증)된 사용자 정보 가져오는 방법(Tip)
  - **Controller 에서 사용자 정보 얻기(Authentication 객체로 얻기)**

```
@Controller
public class SecurityController{
	@GetMapping("/username")
	@ResponseBody
	public String currentUserName(Authentication authentication) {
		UserDetails userDetails = (UserDetails)
		authentication.getPrincipal();
		return userDetails.getUsername();
	}
}
```

5.JWT

- 로그인(인증)된 사용자 정보 가져오는 방법(Tip)
  - **request 객체에서 얻기**

```
@RequestMapping("/username")
@ResponseBody
public String currentUserNameSimple(HttpServletRequest request) {
        Principal principal = request.getUserPrincipal();
        return principal.getName();
}
```

5. JWT

- 로그인(인증)된 사용자 정보 가져오는 방법(Tip)
  - **@AuthenticationPrincipal**
    - Spring Security 3.2 부터는 annotation을 이용하여 현재 로그인한 사용자 객체를 인자에 주입할 수 있음.
    - @AuthenticationPrincipal를 이용하여 CustomUser 객체에 사용자 객체를 주입

```
@Controller
public class SecurityController {
	@GetMapping("/messages/inbox")
	public ModelAndView currentUserName(@AuthenticationPrincipal CustomUser customUser) {
		String username = customUser.getUsername();
		// .. find messages for this user and return them ...
	}
}
```

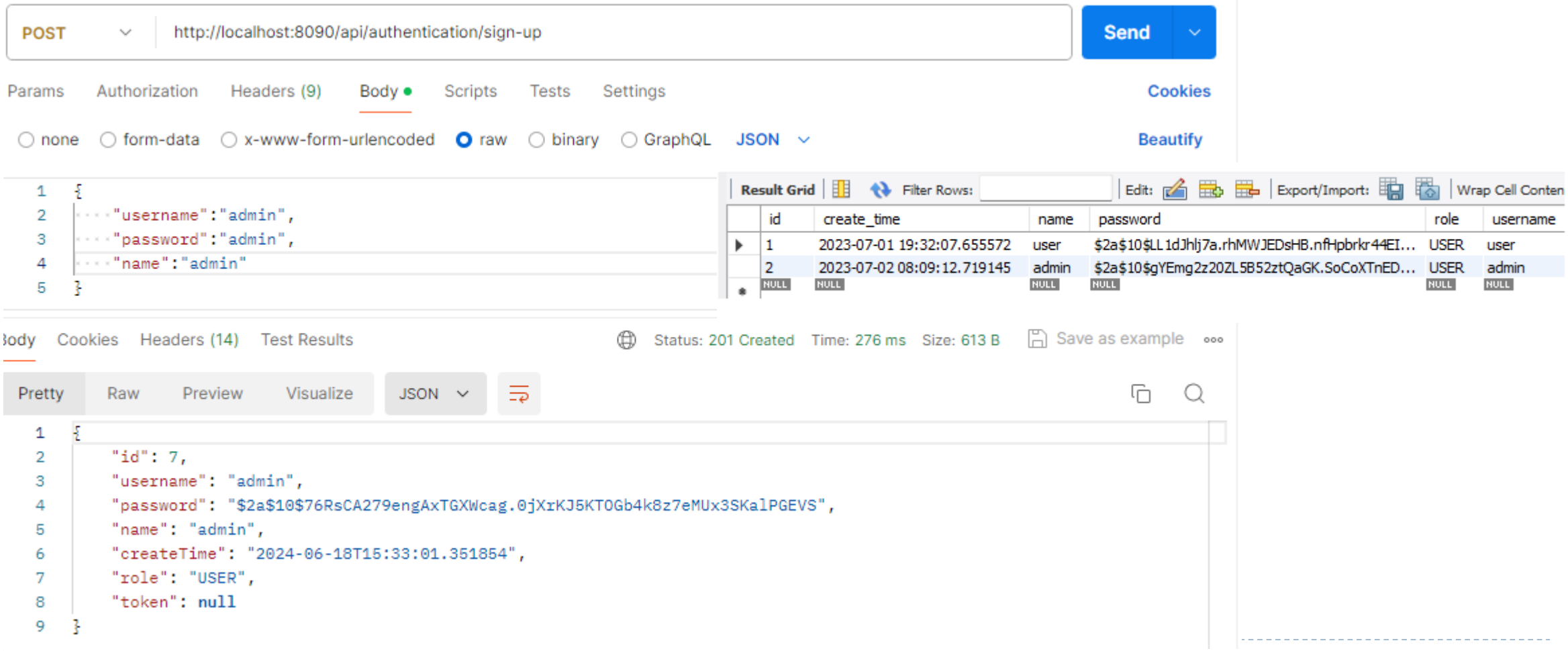
6. Controller

- 유저 컨트롤러
  - 새유저 생성(sign-up), 로그인(sign-in)등은 AuthController에서 처리
  - role 업데이트 처리

```java
@RestController
@RequestMapping("/api/user")
@RequiredArgsConstructor
public class UserController {
    private final UserService userService;

    @PutMapping("change/{role}")
    public ResponseEntity<Object> changeRole(@AuthenticationPrincipal UserPrinciple userPrinciple,
                                             @PathVariable Role role) {
        userService.changeRole(role,userPrinciple.getUsername());
        return ResponseEntity.ok( body: true);
    }
}
```

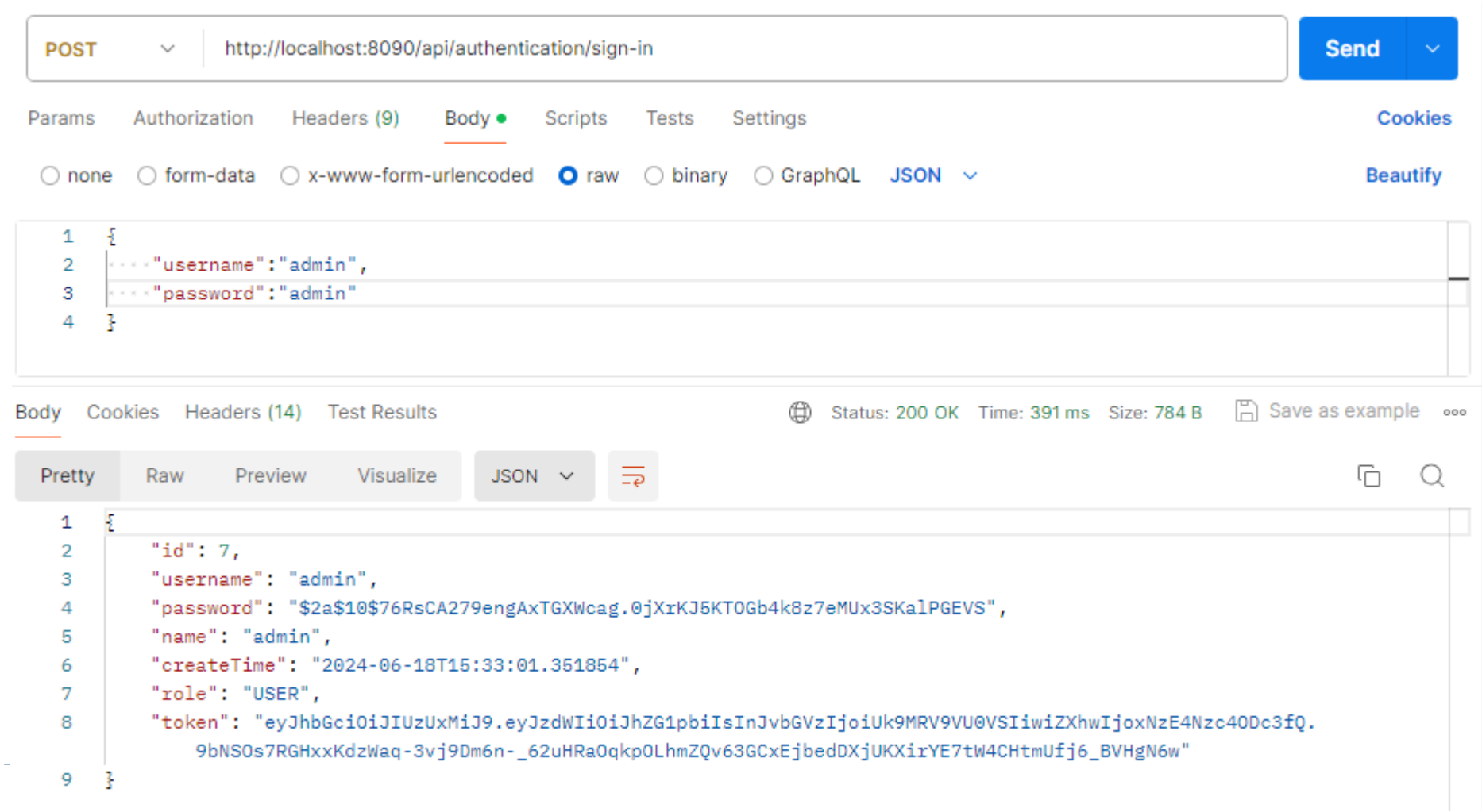
6. Controller

- 유저 컨트롤러- 테스트
  - 테스트 포스트맨에서 admin 유저 가입하기



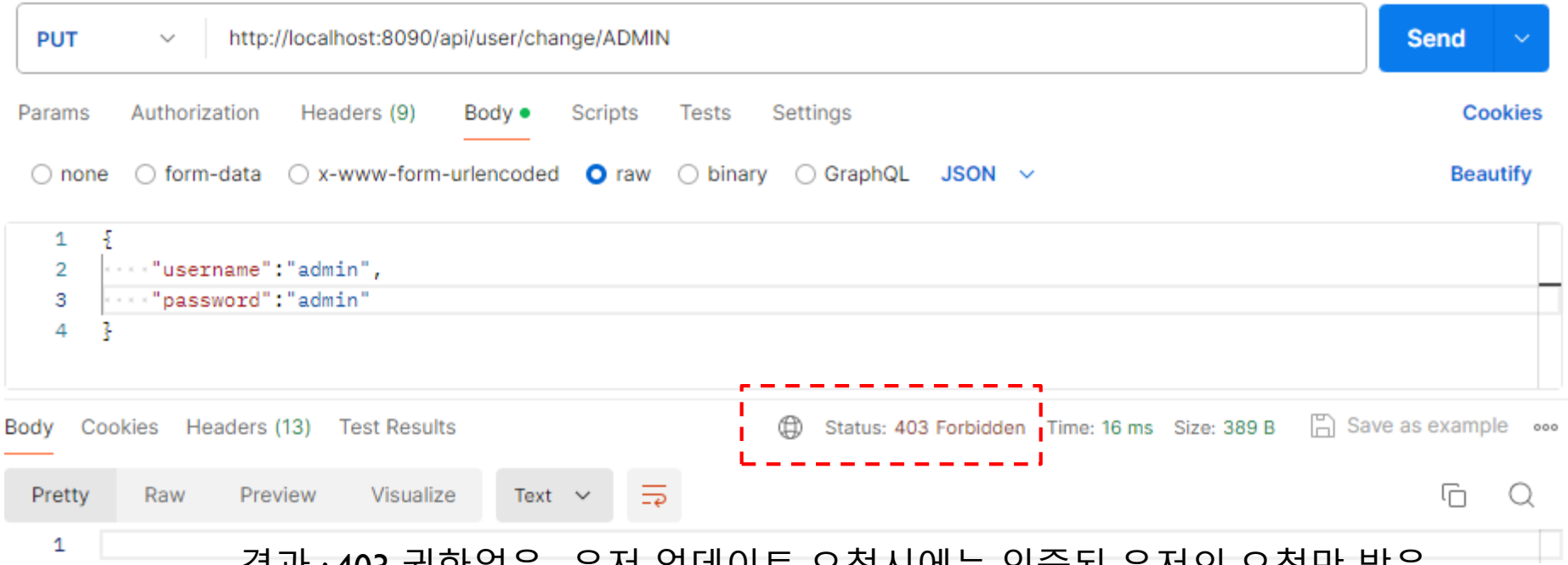
6. Controller

- 유저 컨트롤러- 테스트
  - 로그인 하기



6. Controller

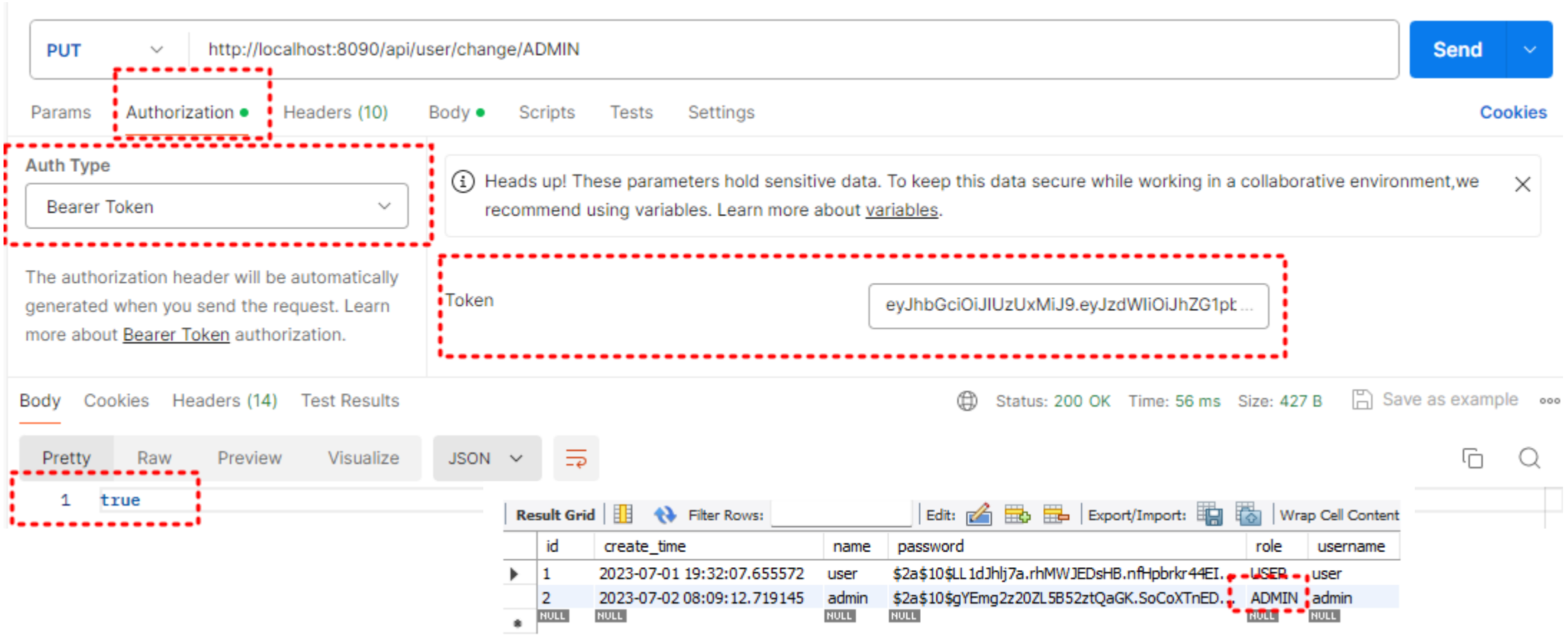
- 유저 컨트롤러- 테스트
  - 유저 롤 업데이트



결과 : 403 권한없음, 유저 업데이트 요청시에는 인증된 유저의 요청만 받음.
이유는 ?
기존의 시큐리티 로그인시 인증된 정보가 세션에 남아있기 때문에 유지되지만
JWT 토큰 방식은 요청시 헤더에 토큰을 가지고 있어야 한다.

6. Controller

- 유저 컨트롤러- 테스트
  - 유저 롤 업데이트



6. Controller

- ProductController
  - 제품 추가
  - 제품리스트 가져오기
  - 제품삭제

```
@RestController
@RequiredArgsConstructor
@RequestMapping("api/product")
public class ProductController {
    private final ProductService productService;

    @PostMapping
    public ResponseEntity<Object> saveProduct(@RequestBody ProductDTO productDTO){
        return new ResponseEntity<>(productService.saveProduct(productDTO) , HttpStatus.CREATED);
    }

    @GetMapping
    public ResponseEntity<List<ProductDTO>> getAllProducts(){
        System.out.println("getList");
        return new ResponseEntity<>(productService.findAllProducts(), HttpStatus.OK);
    }

    @DeleteMapping("{productId}")
    public ResponseEntity<Object> deleteProduct(@PathVariable  Long productId){
        productService.deleteProduct(productId);
        return new ResponseEntity<>(HttpStatus.OK);
    }
}
```

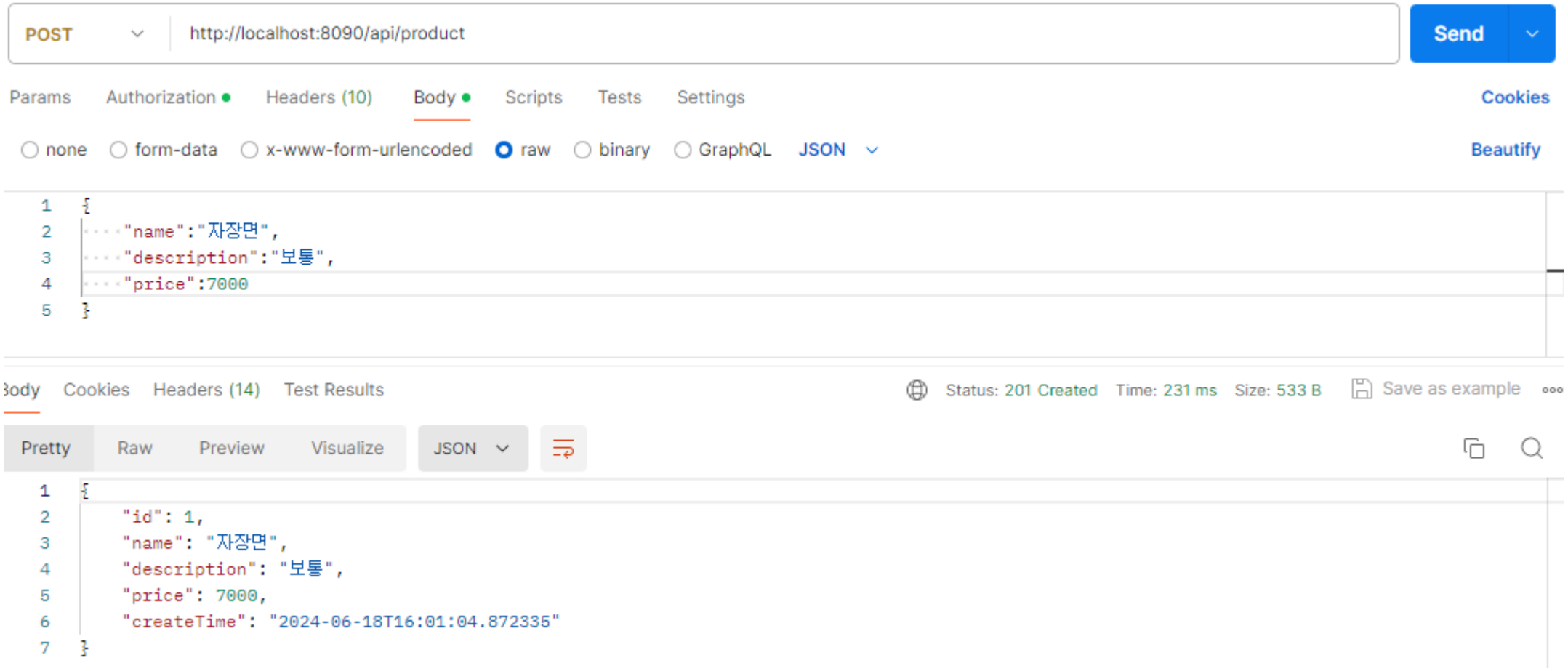
6. Controller

- ProductController
  - 시큐리티의 필터체인 수정 제품리스트는 누구든지 가능
  - 그 외 제품 수정,삭제,입력은 ADMIN 롤만 가능

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception{
    http
            .csrf(AbstractHttpConfigurer::disable)
            .cors(httpSecurityCorsConfigurer -> corsConfigurationSource())
            .sessionManagement((session)->session
                    .sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .authorizeHttpRequests((authz)-> authz
                    .requestMatchers("/api/authentication/**").permitAll()
                    .requestMatchers(HttpMethod.GET, "/api/product/**").permitAll()
                    .requestMatchers("/api/product/**").hasRole(Role.ADMIN.name())
                    .anyRequest().authenticated()
            )
            .addFilterBefore(jwtAuthorizationFilter(), UsernamePasswordAuthenticationFilter.class);
    return http.build();
}
```

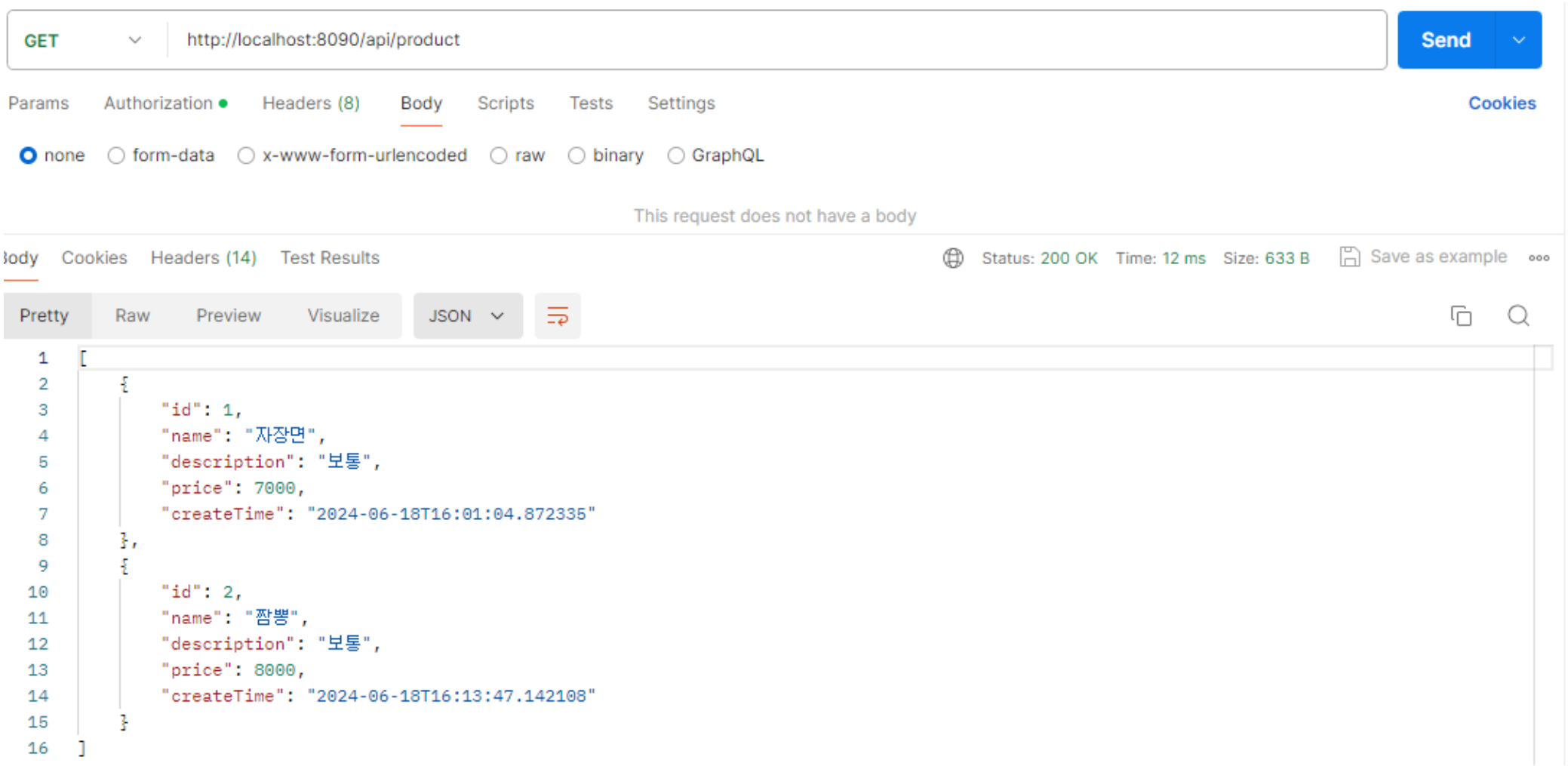
6. Controller

- ProductController 테스트- 제품추가 :ADMIN Role만 접근 가능



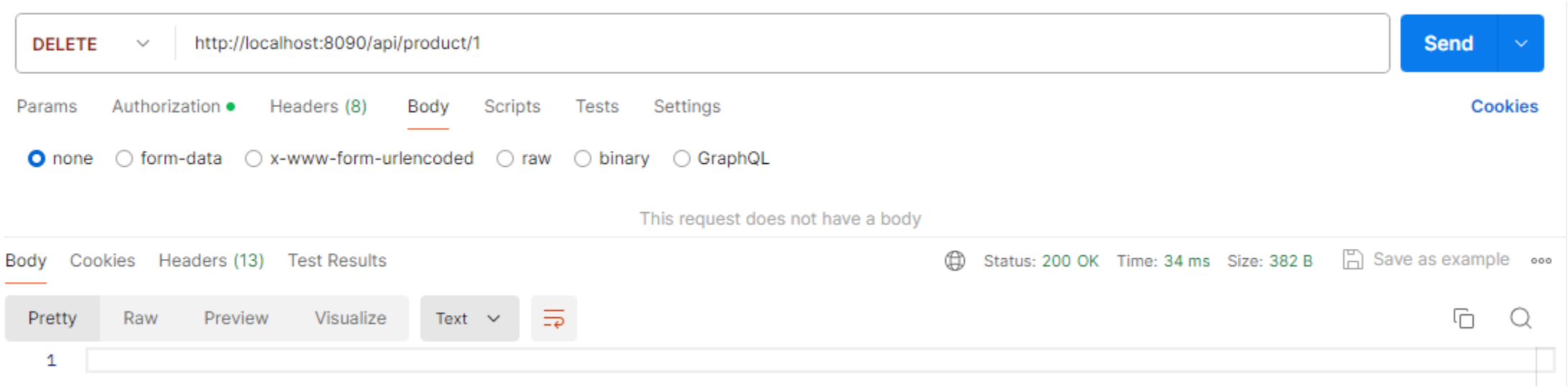
6.Controller

- ProductController 테스트- 제품 리스트 얻기



6. Controller

- ProductController 테스트- 제품 삭제



6. Controller

- PurchaseController
  - 구매추가, 사용자별 구매리스트

```
@Log4j2
@RestController
@RequiredArgsConstructor
@RequestMapping("/api/purchase")
public class PurchaseController {
    private final PurchaseService purchaseService;

    @PostMapping
    public ResponseEntity<Object> savePurchase(@RequestBody PurchaseDTO purchaseDTO){
        log.info("PurchaseController purchaseDTO:"+purchaseDTO);
        return new ResponseEntity<>(purchaseService.savePurchase(purchaseDTO), HttpStatus.CREATED);
    }

    @GetMapping
    public ResponseEntity<Object> getAllPurchasesOfUser(@AuthenticationPrincipal UserPrinciple userPrinciple){
        System.out.println("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~"+userPrinciple.getUser());
        return ResponseEntity.ok(purchaseService.findPurchaseItemsOfUser(userPrinciple.getUsername()));
    }
}
```

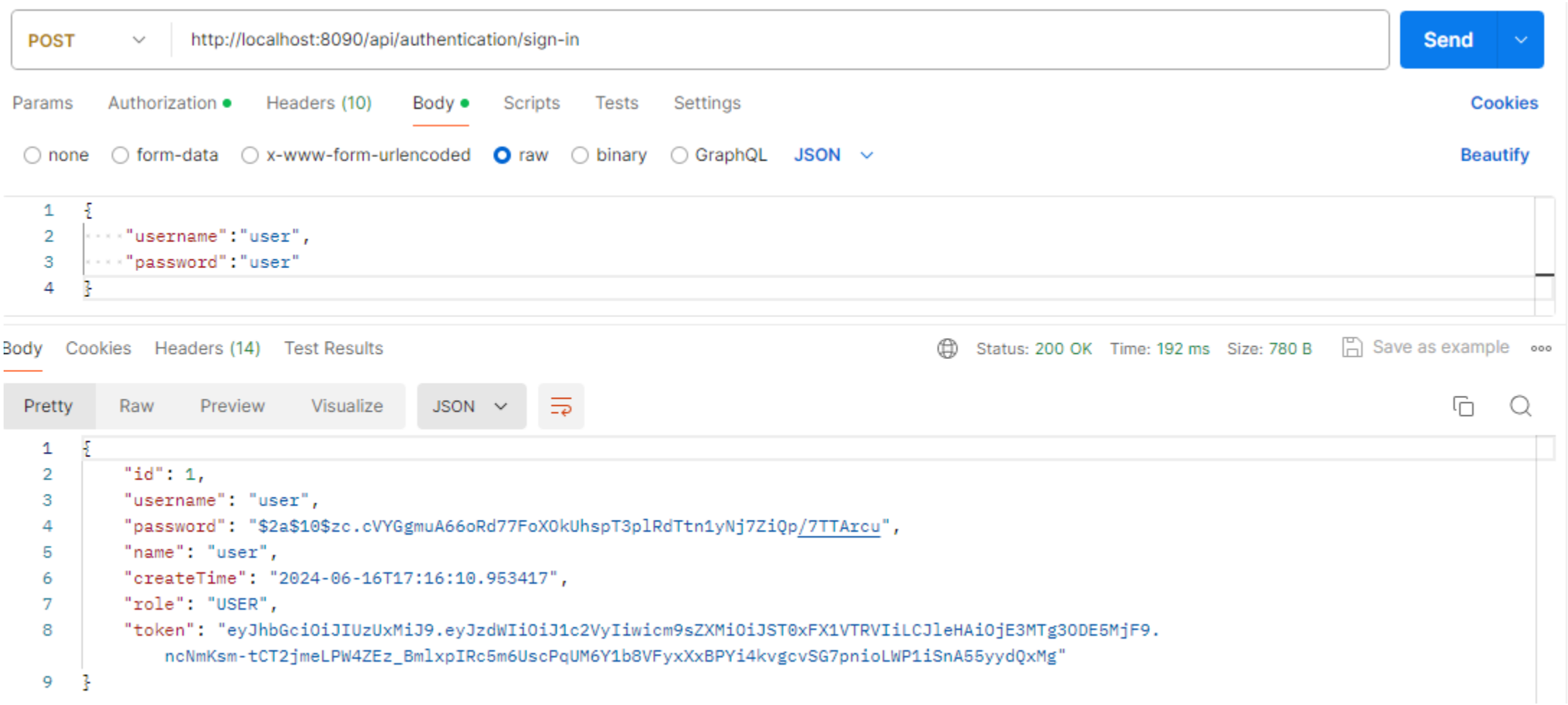
6. Controller

- PurchaseServiceImpl 의 findPurchaseItemofUser() 수정

```
@Override
public List<PurchaseItem> findPurchaseItemsOfUser(String username) {
    System.out.println("service^^^^^^^^^^^^"+username);
    User user=userRepository.findByUsername(username).orElseThrow();

     return purchaseRepository.findAllPurchasesOfUser(user.getId());
}
```

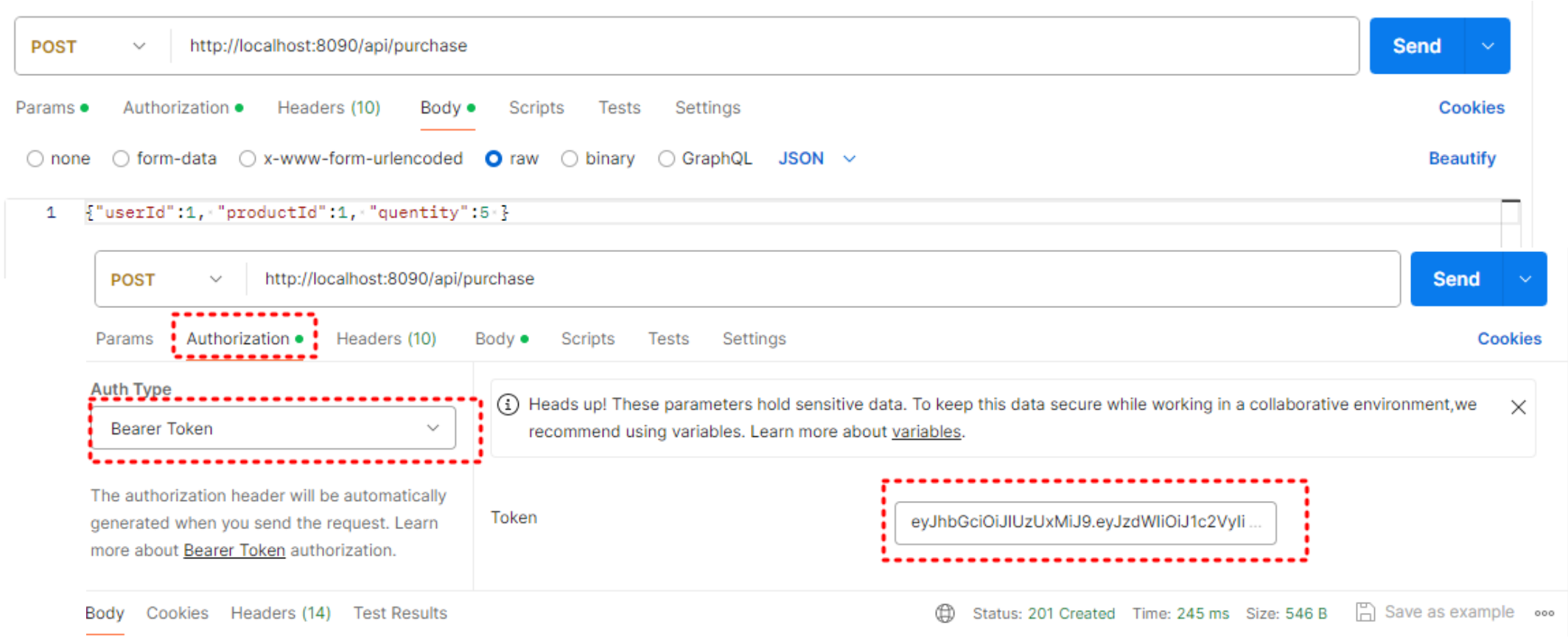
6. Controller

- user 사용자로 로그인



6. Controller

- 구매추가 : body에 추가 데이터, Authorization에 Token 설정



6. Controller

- 구매리스트 확인