# Univerzitet Singidunum

# Tehnički fakultet

# Web Blog Application

-Projektni rad-

**Predmet:** Internet Softverske Arhitekture

**Profesor:**

Nebojša Bačanin Džakula

**Asistent:**

Petar Biševac

**Student:**

Jovana Kržanović 2018/200030

Beograd, 2021 godine

# Sadržaj

# 1. Uvod

Kroz dokumentaciju biće prikazana implementacija RESTful web blog API-ja primenom tehnologija pokrivenih predmetom ''Internet Softverske Arhitekture''. Svrha navedene aplikacije je da omogući registrovanim korisnicima da objavljuju blogove, pretražuju sadržaje blogova postavljenih od strane drugih korisnika, dodavanje blogova u listu omiljenih i ostavljanje komentara. Takođe korisnici platforme imaju mogućnost da pregledaju korisničke naloge drugih korisnika, njihove liste omiljenih i objavljenih blogova. Korisniku se nudi i opcija ažuriranja podataka , kao i brisanje naloga.

# 2. Prikaz korišćenih tehnologija

API (Application Programing Interface) predstavlja posrednik koji omogućava da dve aplikacije (sa klijentske i serverske strane) međusobno komuniciraju i razmenjuju podatke, tako što klijenti šalju zahteve (Http-request), a server šalje odgovore (Http-response). Komunikacija se izvršava putem poruka, najčešće u JSON formatu. Kroz realizaciju projekta, napravljen je REST (Representional State Transfer) API u Java Spring backend tehnologiji, koji se koristi za izgradnju skalabilnih i lakih (u smislu koliko resursa koriste) veb servisa. Sledeće backend tehnologije su korišćene za realizaciju ovog projekta:

**Spring Boot** - Kao i sam Spring framework, fokusira se da pruži fleksibilnost sa dependency injecton-om, koji nam omogućava da pozovemo drugi objekat u konstruktoru sa anotacijom @Autowired. Spring Boot znatno skraćuje kompleksnost i dužinu koda, što ga poprilično čini user-friendly.

**Spring Security** - Framework koji pruža autentifikaciju i autorizaciju korisnika koji koriste aplikaciju. Koriste se JWT (JSON Web Token) koji se koristi za autentifikaciju korisnika svaki put kad podnese neki Http request ka serveru, pošto je REST API stateless, što znači da ne pamti stanja.

**Spring Data JPA** - Koristi se za implementaciju JPA-based repozitorijuma, korišćenje Hibernate-a koji pruža mogućnost komunikacije sa relacionim bazama podataka, npr. MySQL koji je korišćen za izradu projekta.

**MySQL Driver** - JDBC driver koji omogućava pristup MySQL podacima u realnom vremenu (real-time).

**Spring Web** - Pruža infrastrukturu za izgradnju i održavanje RESTful web aplikacija.

Što se tiče frontend-a, korišćen je **Angular**, framework pisan u TypeScriptu koji se koristi za pravljenje single-page aplikacija (SPAs) i dobar je za rukovanje RESTful API-jem.

Alati koji su korišćeni za realizaciju projekta su:

**IntelliJ Ultimate Edition** - okruženje u kojem je pisan backend u Java Spring framework-u.

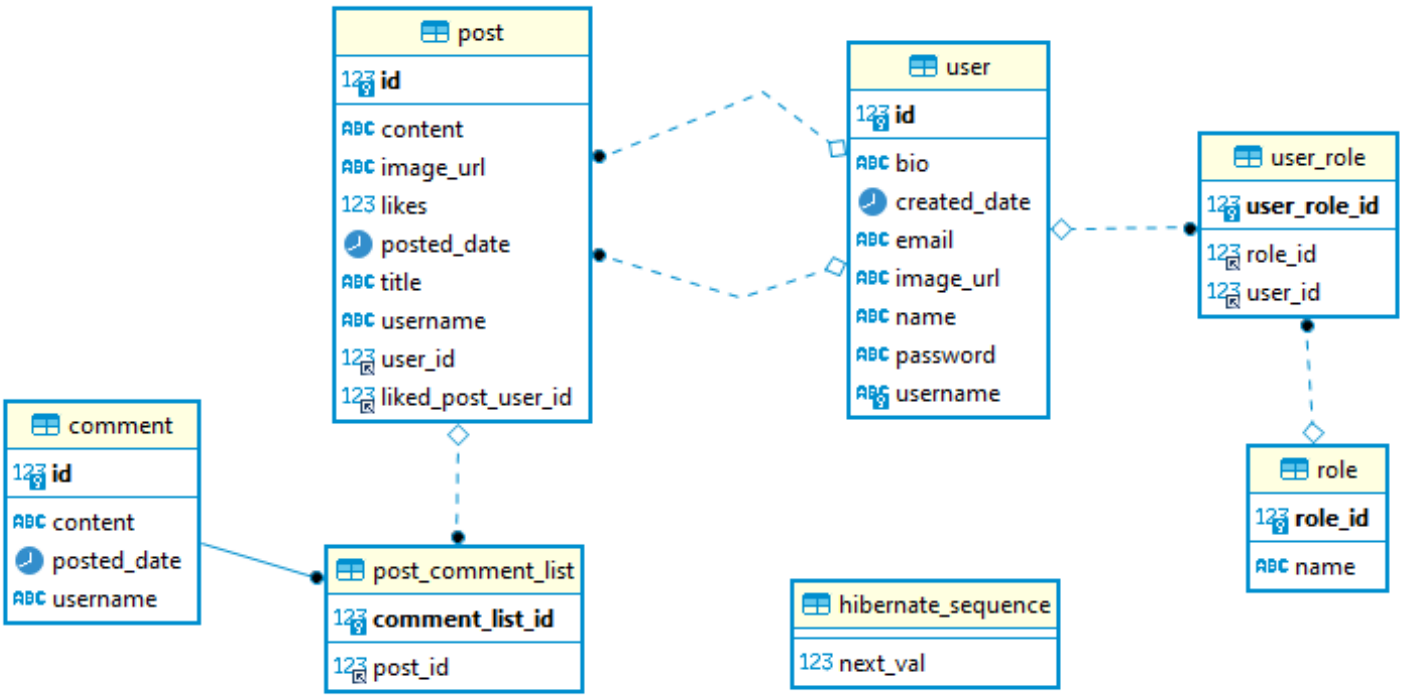
**Postman** - Aplikacija koja je korišćena za testiranje endpointa REST API-ja.

**DBeaver Community Edition** - Okruženje za administraciju i upravljanje bazama podataka.

**Maven** - Alat koji se koristi za izgradnju i upravljanje bilo kojim projektom zasnovanom na Javi. Baziran je na POM (Project Object Model), što predstavlja ključnu XML datoteku, pod nazivom pom.xml, koja sadrži informacije o samom projektu i konfiguraciji, kao što su zavisnosti (dependency), pluggin-ovi, itd.

# 3. Opis Java Spring RESTful API-ja

Spring API se sastoji od modela, repozitorijuma, servisa, kontrolera i konfiguracionih klasa. **Modeli** predstavljaju klase označene *@Entity* anotacijom koje se mogu mapirati u tabele u određenom formatu i služe za prosleđivanje informacija i podataka između front i backend-a. **Repozitorijumi** predstavljaju interfejse koji koji proširuju JPARepository koji pruža pristup osnovnim CRUD operacijama koje se mogu izvršiti nad bazom podataka. U njima se dodatno definišu metode koje je potrebno modifikovati za određene potrebe korišćenjem *@Qeury* anotacije, koja omogućava pisanje upita ka bazi podataka. Servisi su klase u kojima se izvršava cela logika API-ja. One implementiraju interfejse koji obezbeđuju tim klasama da implementiraju sve potrebne metode za funkcionalnost. **Controlleri** predstavljaju klase koje u sebi sadrže metode koje služe kao pristupne tačke (endpoint) preko kojih će frontend aplikacija, u našem slučaju Angular, moći da komunicira sa backend-om. I na kraju, konfiguracione klase sadrže logiku za konfiguraciju Spring Security i kreiranja JSON Web Tokena koji se koriste za autentifikaciju i autorizaciju korisnika svaki put kada podnosi zahteve ka serveru.




Slika 1. (Prikaz relacija tabela)

```java
@Entity
public class User implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(nullable = false, updatable = false)
    private Long id;

    @Column(unique = true, nullable = false)
    private String username;

    private String email;

    @JsonProperty(access = JsonProperty.Access.WRITE_ONLY)
    private String password;

    private String name;

    @Column(columnDefinition = "text")
    private String bio;

    @CreationTimestamp
    private Date createdDate;

    @Column(columnDefinition = "text")
    private String imageUrl;

    @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
    @JoinColumn(name = "user_id")
    private List<Post> post;

    @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
    @JoinColumn(name = "liked_post_user_id")
    private List<Post> likedPost;

    @OneToMany(mappedBy = "user", cascade = CascadeType.ALL, fetch = FetchType.EAGER)
    private Set<UserRole> userRoles = new HashSet<>();
```

Slika 2. (Prikaz User modela)

Na slici 2. može se detaljnije videti logika User modela. Implementacija Serializable omogućava konverziju Java objekata u sekvencu bajtova koji se pritom mogu sačuvati u bazu podataka ili transportovati kroz mrežu.

*@Id* - predstavlja kolonu u tabeli koja će imati ulogu primarnog ključa.

*@JsonProperty* – Uz pomoć ove anotacije specificiramo da je atribut password READ_ONLY.

*@OneToMany* – Anotacija koja uspostavlja relaciju jedan prema više, gde atribut post predstavlja listu atributa drugog entiteta.

*@JoinColumn* – predstavlja entitet, tj strain ključ, koji se nalazi u Post tabeli sa kojom je uspostavljena jedan prema više relacija.

```java
public interface IUserRepository extends JpaRepository<User, Long> {

    public User findByUsername(String username);

    public User findByEmail(String email);

    @Query("SELECT user FROM User user WHERE user.id=:param")
    public User findUserById(@Param("param") Long id);

    public List<User> findByUsernameContaining(String username);

}
```

Slika 3. (prikaz UserRepository interfejsa)

```java
@Service
@Transactional
public class UserServiceImpl implements UserService {


    @Autowired
    private IUserRepository userRepository;

    @Autowired
    private IRoleRepository roleRepository;

    @Autowired
    private UserService userService;

    @Autowired
    private BCryptPasswordEncoder bCryptPasswordEncoder;

    @Override
    @Transactional
    public User saveUser(String name, String username, String email, String password){
        User user = new User();
        user.setPassword(password);
        String encryptedPassword = bCryptPasswordEncoder.encode(password);
        user.setName(name);
        user.setUsername(username);
        user.setEmail(email);
        user.setPassword(encryptedPassword);
        Set<UserRole> userRoles = new HashSet<>();
        userRoles.add(new UserRole(user, userService.findUserRoleByName("USER")));
        user.setUserRoles(userRoles);
        userRepository.save(user);
        return user;
    }

    @Override
    public User findByUsername(String username) { return userRepository.findByUsername(username); }

    @Override
    public User findByEmail(String email) { return userRepository.findByEmail(email); }

    @Override
```

Slika 4. (Prikaz UserService klase)

Na slici 4. se vidi deo User Service klase u kojoj se nalaze implementirane metode iz UserService interfejsa uz pomoć *@Override* anotacije. *@Autowired* je mehanizam koji nam omogućava da implementiramo dependency injection na klase I interfejse koji su nam potrebni u servisu.

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

    @Autowired
    private UserService userService;


    @GetMapping("/list")
    public ResponseEntity<?> getUserList() {
        List<User> users = userService.userList();
        if (users.isEmpty()) {
            return new ResponseEntity<>( body: "No Users Found..", HttpStatus.NOT_FOUND);
        }
        return new ResponseEntity<>(users, HttpStatus.OK);
    }

    @GetMapping("/{username}")
    public ResponseEntity<?> getUserInfo(@PathVariable String username) {
        User user = userService.findByUsername(username);
        if (user == null) {
            return new ResponseEntity<>( body: "User is not found..", HttpStatus.NOT_FOUND);
        }
        return new ResponseEntity<>(user, HttpStatus.OK);
    }

    @GetMapping("/findByUsername/{username}")
    public ResponseEntity<?> getUsersByUsername(@PathVariable String username) {
        List<User> user = userService.getUsersListByUsername(username);
        if (user.isEmpty()) {
            return new ResponseEntity<>( body: "No Users Found..", HttpStatus.NOT_FOUND);
        }
        return new ResponseEntity<>(user, HttpStatus.OK);
    }

    @PostMapping("/register")
    public ResponseEntity<?> register(@RequestBody HashMap<String, String> request) {
        String username = request.get("username");
        if (userService.findByUsername(username) != null) {
            return new ResponseEntity<>( body: "usernameExists", HttpStatus.CONFLICT);
```

Slika 5. (Prikaz UserController klase)

*@RestController* – Anotacija koja obeležava klasu da je u pitanju kontroler.

*@RequestMapping* – anitacija koja specificira koja je ulazna tačka za sve navedene metode u kontroleru.

*@GetMapping, @PostMapping i @DeleteMapping* – Anotacije koje specificiraju da li je u pitanju GET, POST ili DELETE metoda.

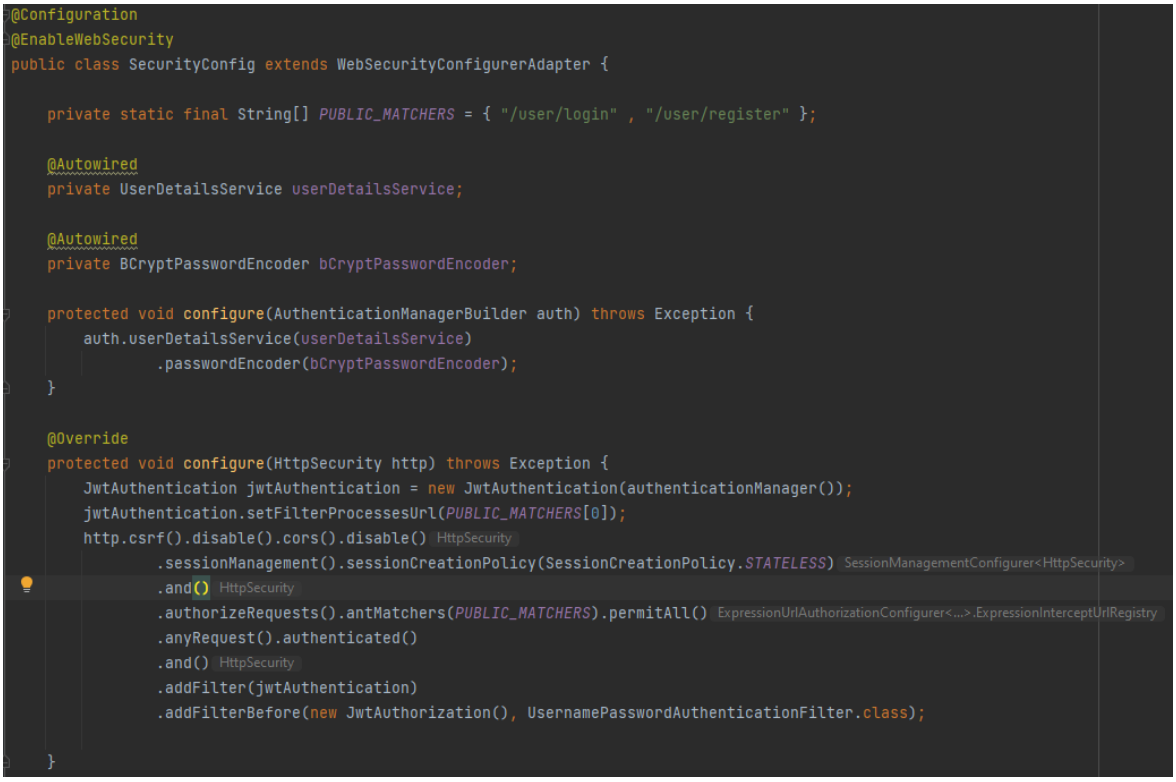
*@PathVariable* – Anotacija koja služi da izvuče informacije iz URI-ja. Koristi se u kombinaciji sa GET I DELETE anotacijama.

*@RequestBody* – Anotacija koja zahteva informacije iz body-ja jednog zahteva. Vrši automatsku deserijalizaciju JSON formata u Java objekat.

```java
@Service
@Transactional
public class UserDetailsServiceImpl implements UserDetailsService {


    @Autowired
    UserService userService;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        User user = userService.findByUsername(username);
        if (user == null) {
            throw new UsernameNotFoundException("Username " + username + " was not found");
        }
        Collection<GrantedAuthority> authorities = new ArrayList<>();
        Set<UserRole> userRoles = user.getUserRoles();
        userRoles.forEach(role -> {
            authorities.add(new SimpleGrantedAuthority(userRoles.toString()));
        });
        return new org.springframework.security.core.userdetails.User(user.getUsername(), user.getPassword(), authorities);
    }

}
```

Slika 6. (Prikaz UserDetailsService klase)

Klasa UserDetailsService daje informacije Spring Security-ju na koji način želimo da se korisnik učitava i autentifikuje svaki put kada korisnik podnosi zahteve ka API-ju.



```java
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    private static final String[] PUBLIC_MATCHERS = { "/user/login" , "/user/register" };

    @Autowired
    private UserDetailsService userDetailsService;

    @Autowired
    private BCryptPasswordEncoder bCryptPasswordEncoder;

    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.userDetailsService(userDetailsService)
                .passwordEncoder(bCryptPasswordEncoder);
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        JwtAuthentication jwtAuthentication = new JwtAuthentication(authenticationManager());
        jwtAuthentication.setFilterProcessesUrl(PUBLIC_MATCHERS[0]);
        http.csrf().disable().cors().disable() HttpSecurity
                .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS) SessionManagementConfigurer<HttpSecurity>
                .and() HttpSecurity
                .authorizeRequests().antMatchers(PUBLIC_MATCHERS).permitAll() ExpressionUrlAuthorizationConfigurer<...>.ExpressionInterceptUrlRegistry
                .anyRequest().authenticated()
                .and() HttpSecurity
                .addFilter(jwtAuthentication)
                .addFilterBefore(new JwtAuthorization(), UsernamePasswordAuthenticationFilter.class);

    }
```

Slika 7. (Prikaz SecurityConfig klase)

U klasi SecurityConfig klasi će se omogućiti Spring Security na nivou cele aplikacije. Ona sadrži metodu configure koja sadrži logiku konfiguracije autentifikacije za stranice aplikacije na kojima će se korisnik autentifikovati svaki put kada podnosi zahteve ka serveru. Login i register stranice su stranice na kojima se neće primenjivati navedena pravila.

```java
public class JwtAuthentication  extends UsernamePasswordAuthenticationFilter {

    private AuthenticationManager authenticationManager;

    public JwtAuthentication(AuthenticationManager authenticationManager) {
        this.authenticationManager = authenticationManager;
    }

    @Override
    public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response)
            throws AuthenticationException {
        ObjectMapper objectMapper = new ObjectMapper();
        objectMapper.configure(Feature.AUTO_CLOSE_SOURCE,  state: true);
        User user;
        try {
            user = objectMapper.readValue(request.getInputStream(), User.class);
        } catch (Exception e) {
            e.printStackTrace();
            throw new RuntimeException("Unable to convert JSON format into Java Object: " + e );
        }
        return authenticationManager.authenticate(new UsernamePasswordAuthenticationToken(
                user.getUsername(),
                user.getPassword()));
    }

    @Override
    protected void successfulAuthentication(HttpServletRequest request, HttpServletResponse response, FilterChain chain,
                                            Authentication authentication) throws IOException, ServletException {
        org.springframework.security.core.userdetails.User sUser = (org.springframework.security.core.userdetails.User) authentication.getPrincipal();
        List<String> roles = new ArrayList<>();
        sUser.getAuthorities()
                .forEach(authority -> {
                    roles.add(authority.getAuthority());
                });
        String jwtToken = JWT.create()
                .withIssuer("blog app")
                .withSubject(sUser.getUsername())
                .withArrayClaim( name: "roles", roles.stream().toArray(String[]::new))
                .withExpiresAt(new Date(System.currentTimeMillis()+SecurityConstants.EXPIRATION_TIME))
                .sign(Algorithm.HMAC256(SecurityConstants.SECRET));
        response.addHeader(SecurityConstants.HEADER_TYPE,  s1: SecurityConstants.TOKEN_PREFIX + jwtToken);
```

Slika 8. (Prikaz JwtAuthentication klase)

Metoda attemptAuthentication() se svaki put poziva kada Spring Security treba da autentifikuje korisnika. Uz pomoć ObjectMapper-a, ima ulogu da konvertuje JSON u Java objekat User klase.
Metoda successfulAuthenticaion() se poziva onog trenutka kada se attemptAuthentication() uspešno izvršila, i u njoj se kreiraju JSON Web Tokeni, koji predstavlja response servera na zahtev korisnika, i token ima funkciju da autentifikuje korisnika svaki put kad šalje zahteve serveru. U Header-u, pod ključem „Authorization“, će biti prosleđena vrednost tokena.

```java
public class JwtAuthorization extends OncePerRequestFilter {


    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
            throws ServletException, IOException {

        response.addHeader( s: "Access-Control-Allow-Origin", SecurityConstants.CLIENT_DOMAIN_URL);

        response.addHeader( s: "Access-Control-Allow-Headers", s1: "Origin, Accept, X-Requested-With, "
                + "Content-Type, Access-Control-Request-Method, " + "Access-Control-Request-Headers, Authorization");

        response.addHeader( s: "Access-Control-Expose-Headers",
                s1: "Access-Control-Allow-Origin, " + "Access-Control-Allow-Credentials, " + "Authorization");

        response.addHeader( s: "Access-Control-Allow-Methods", s1: "GET, " + " POST, " + " DELETE, " + " PUT");

        if ((request.getMethod().equalsIgnoreCase( anotherString: "OPTIONS"))) {
            try {
                response.setStatus(HttpServletResponse.SC_OK);
            } catch (Exception e) {
                e.printStackTrace();
            }
        } else {
            String jwtToken = request.getHeader(SecurityConstants.HEADER_TYPE);
            if (jwtToken == null || !jwtToken.startsWith(SecurityConstants.TOKEN_PREFIX)) {
                filterChain.doFilter(request, response);
                return;
            }
            JWT.require(Algorithm.HMAC256(SecurityConstants.SECRET));
            DecodedJWT jwt = JWT.decode(jwtToken.substring(SecurityConstants.TOKEN_PREFIX.length()));
            String username = jwt.getSubject();
            List<String> roles = jwt.getClaims().get("roles").asList(String.class);
            Collection<GrantedAuthority> authorities = new ArrayList<>();
            roles.forEach(role -> authorities.add(new SimpleGrantedAuthority(role)));
            UsernamePasswordAuthenticationToken authenticatedUser = new UsernamePasswordAuthenticationToken(
                    username, credentials: null, authorities);
            SecurityContextHolder.getContext().setAuthentication(authenticatedUser);
            filterChain.doFilter(request, response);
        }
    }
```

Slika 9. (Prikaz JwtAuthorization klase)

Uloga JwtAuthorization klase je da posmatra razmenu zahteva i odgovora. Unutar klase su je specificiran opseg dozvoljenih Header-a u aplikaciji. Izvršava se dekodovanje JSOn Web tokena iz koje se izvlače informacije o korisniku, koje se dalje prosleđuju Spring Security-ju za autentifikaciju.

# 4. Zaključak

Dekumentovani API, s obzirom na njegovu jednostavnost, ima puno prostora za nadogradnju i poboljšanje same strukture u odnosu na druge API-je dostupne na internetu kojima mi, kao korisnici, svakodnevno podnosimo zahteve. Svrha pravljenja navedenog API-ja je primenjivanje znanja i veština stečenih na predmetu „Internet Softverske Arhitekture“.